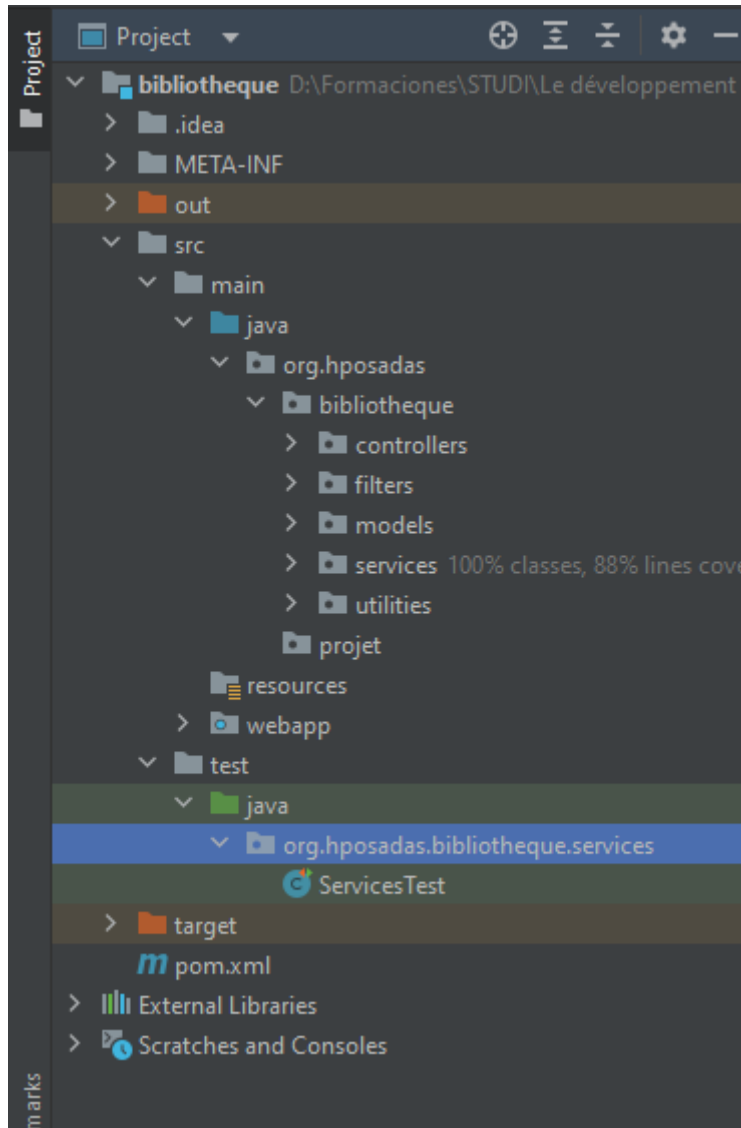


Documentation Test Fonctionnel et d'intégration

Pour vérifier la qualité de mon code ainsi que du test je me suis servi du plug-in **Sonar Lint** qui m'a permis de supprimer des lignes ainsi que de créer des règles nécessaires pour optimiser la qualité du code. Après avoir installé Sonar Lint j'ai pu modifier l'structure de mon code en supprimant le package Repository ainsi que ses classes. La structure de mon code est meilleure.



Tests Fonctionnels

Pour tester les fonctions de mon projet j'utilisé **JUnit** et **Mockito**, ces outils ont été rajouté à mon projet Maven en tant que dépendances de la façon suivante :

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.9.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
```

```

        <artifactId>mockito-core</artifactId>
        <version>4.8.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>4.8.0</version>
    </dependency>

```

Les fonctionnalités qu'ont été testés appartient aux services de mon projet. Le package services contient les interfaces et classes qui apportent les opérations centrales les plus importantes qui seront après utilisées par les servlets pour traiter les requests et la récupération/insertion de données vers la base de données.

Le package Services contient 3 interfaces qui sont importées par 3 classes. Dans ces classes nous trouvons 5 fonctions centrales qui sont :

Classe LivreServiceImpl

- Fonction `list()` : Permet de récupérer tous les livres présents dans la table livres de la base de données, le retour de cette fonction renvoi une `List<Livre>`.
- Fonction `byName()` : Permet de rechercher dans la table livres occurrences dans son titre. Cette fonction renvoi une `List<Livre>`.
- Fonction `parId()` : Permet de trouver un livre dans la table livre par son Id. Cette fonction renvoi un `Optional<Livre>` de retour.

```

@Override
public List<Livre> lister() {
    List<Livre> livres = new ArrayList<>();

    try (Statement stmt = conn.createStatement();
         ResultSet res = stmt.executeQuery("SELECT * FROM livres")) {
        while (res.next()) {
            Livre l = getLivre(res);
            livres.add(l);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return livres;
}

@Override
public List<Livre> byName(String name) throws SQLException {
    List<Livre> livres = new ArrayList<>();

    try (PreparedStatement stmt = conn.prepareStatement("SELECT * FROM livres as l WHERE l.titre LIKE ?")) {
        stmt.setString(1, "%" + name + "%");

        try (ResultSet res = stmt.executeQuery()) {
            while (res.next()) {
                Livre l = getLivre(res);
                livres.add(l);
            }
        }
    }

    return livres;
}

```

```

    }
}
return livres;
}

@Override
public Optional<Livre> findById(Long id) {
    return lister().stream().filter(l->l.getId().equals(id)).findAny();
}

```

Test de la fonctionnalité lister()

Pour tester cette fonctionnalité j'ai instancié un objet de la classe LivreServiceImpl (classe qui contient la fonctionnalité lister()). Après j'ai créé une variable du type List<Livre> qui contient le retour de l'appel à la fonction lister(). D'un autre côté j'ai effectué dans le test une consultation directement sur la base de données que j'ai enregistrée dans une List<Livre> pour comparer ces deux résultats. Voici le test de cette fonctionnalité :

```

@Test
@DisplayName("Afficher les livres.")
void lister() {
    LivreService livreService = new LivreServiceImpl(conn);
    List<Livre> livresActuel = livreService.lister();
    List<Livre> livresExpected = new ArrayList<>();

    try (Statement stmt = conn.createStatement();
         ResultSet result = stmt.executeQuery("SELECT * FROM livres")) {
        while (result.next()) {
            Livre l = getLivre(result);
            livresExpected.add(l);
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    };
    assertAll(
        ()-> assertNotEquals(livresExpected, livresActuel, ()-> "Les
elements que vous avez comparé sont le meme objet"),
        //verifier que meme si les deux requetes ont le meme resultat, les deux
objets ne sont pas les memes objets mais des instances differents du meme
type.
        ()-> assertTrue(livreService.lister().size()>0, ()-> "On
s'attendait a avoir du contenu dans la liste de livres mais la liste est
vide"), //verifier que le retour de la fonction lister rend une liste
non vide
        ()-> assertEquals(5, livresActuel.size(), ()-> "La quantité
d'elements dans la liste livres ne correspond pas a la quantité d'elements
présents dans la table livres") //nous savons que la bibliotheque a
5 livres renseignés, nous allons verifier que la taille de la liste de
livres obtenu est egale a 5.
    );
}

```

Test de la fonctionnalité `byName()`

Pour tester cette fonctionnalité je suis allé regarder directement dans la table livres de la base de données pour trouver un mot d'un des livres et le rechercher, ce mot est « voyage ». l'assertion est la suivante

```
@Test
@DisplayName("Recherche des livres par titre.")
void byName() throws SQLException {
    LivreService livreService = new LivreServiceImpl(conn);

    assertNotNull(livreService.byName("voyage"), () -> "Aucune
correspondance"); //verifier qu'au moins un livre a été trouvé par la
fonction byName en utilisant la chaine "voyage".
}
```

Test de la fonctionnalité `byId()`

Pour tester cette fonctionnalité je suis allé chercher un livre et son id dans la table livres pour en suite utiliser cette fonction avec l'id que j'avais trouvé. Les assertions sont les suivantes :

```
@Test
@DisplayName("Recherche des livres par ID.")
void byId() {
    LivreService livreService = new LivreServiceImpl(conn);
    Optional<Livre> livreParId = livreService.byId(1L);
    String titreReel = livreParId.get().getTitre();
    String titreExpected = "Le petit prince";

    assertAll(
        () -> assertTrue(livreParId.isPresent(), () -> "Aucune
correspondance avec la chaine "+ titreExpected +""), //Nous
savons par avance que le livre "Le petit prince" porte l'ID 1. Nous allons
maintenant verifier que la fonction parId(1L) est capable de trouver un
resultat
        () -> assertEquals(titreReel, titreExpected, () -> "Le titre
obtenu et le titre recherché ne correspondent pas") //verifier que
le titre obtenu lors de la recherche correspond bien au titre du livre
enregistre avec l'ID 1.
    );
}
```

Classe UtilisateurServiceImpl

- Fonction `enregistrer()` : Permet d'insérer un nouvel utilisateur dans la table utilisateurs.

```
@Override
public void enregistrer(Utilisateur utilisateur) throws SQLException {
    try(PreparedStatement stmt = conn.prepareStatement("INSERT INTO
utilisateurs(nom, prenom, email, password) VALUES (?, ?, ?, ?)")) {
        stmt.setString(1, utilisateur.getNom());
        stmt.setString(2, utilisateur.getPrenom());
        stmt.setString(3, utilisateur.getEmail());
        stmt.setString(4, utilisateur.getPassword());
        stmt.executeUpdate();
    }
}
```

Test de la fonctionnalité enregistrer()

Pour tester cette fonctionnalité j'ai d'abord créé une instance de la classe Utilisateur et après avoir rempli ses attributs je me suis disposé à enregistrer cet utilisateur en utilisant la fonction enregistrer() en lui passant par argument l'utilisateur déjà créé. Pour tester que l'utilisateur a été bien inséré dans la table utilisateurs j'ai effectué une recherche dans le test avec l'email qui identifie a cet utilisateur pour prouver que l'utilisateur a été bien créé. Les assertions sont les suivantes :

```
@Test
@DisplayName("Creation du compte utilisateur")
void enregistrer() throws SQLException {

    List<Utilisateur> users = new ArrayList<>();

    Utilisateur u = new Utilisateur();           //Creation de l'utiliasteur
de test que nous allons rajouter
    u.setId(90L);
    u.setNom("Hernandez");
    u.setPrenom("Gustavo");
    u.setEmail("gh@test.com");
    u.setPassword("12345");

    UserService<Utilisateur> userService = new
UtilisateurServiceImpl(conn);
    userService.enregistrer(u);                   //insertion de
l'utilisateur dans la base de données

    try(PreparedStatement stmt = conn.prepareStatement("SELECT * FROM
utilisateurs WHERE email=?")){
        stmt.setString(1, u.getEmail());

        try(ResultSet result = stmt.executeQuery()){
            while (result.next()){

                Utilisateur user = getUtilisateur(result);

                users.add(user);
            }
        }
    }

    assertAll(
        ()-> assertFalse(users.isEmpty(), ()-> "L'email "+ u.getEmail()
+"n'a pas été trouvé"),           //verification que l'utilisateur que nous
avons enregistré a été trouvé dans la base de données
        ()-> assertFalse(users.equals(u), ()-> "L'utilisateur "+
u.getEmail() +" et l'utilisateur "+ users.get(0).getEmail() +" sont le meme
objet")           //verification que nous avons enregistré
lorsqu'il a été récupéré de la base de données n'est pas le meme objet que
nous avons enregistré mais une autre instance avec les memes
caracteristiques.
    );
}
```

Classe LoginServiceImpl

- Fonction getUsername() : Permet de savoir si le request qu'on lui passe en argument possède l'attribut « username », si c'est le cas on dit que cet utilisateur est authentifié mais si

le request ne possède pas cet attribut alors l'utilisateur n'est pas authentifié. Par rappel, un utilisateur authentifié est capable de demander un ou plusieurs prêts.

```
@Override
public Optional<String> getUsername(HttpServletRequest req) {
    HttpSession session = req.getSession();
    String username = (String) session.getAttribute("username");
    if (username != null) {
        return Optional.of(username);
    }
    return Optional.empty();
}
```

Test de la fonctionnalité getUsername() :

Le test de cette fonctionnalité a été plus complexe car pour pouvoir appeler la fonction getUsername() en lui passant une instance de la classe HttpServletRequest, cette fonction récupère la session du request reçu par paramètre mais lorsque dans le test je n'ai pas de request à récupérer, j'ai dû en instancier un objet de la classe HttpServletRequest moi-même, le problème se trouve dans le fait que cette instance request n'a pas de session ni de headers, alors j'ai choisi de moquer l'appel à la fonctionnalité en utilisant **Mockito** et de moquer son retour pour simuler dans un cas que le request porte l'attribut username et dans un autre cas un request qui n'a pas l'attribut username.

Par info, lorsque le request porte l'attribut « username » la fonction getUsername() renvoie un Optional<String> qui contient le mail de l'utilisateur authentifié. Lorsque l'attribut username n'est pas présent alors la fonction getUsername renvoie un Optional.empty().

Les assertions de cette fonctionnalité sont les suivantes :

```
@Test
@DisplayName("Vérifier l'authentification de l'utilisateur")
void getUsername() {
    //Mock
    HttpServletRequest requestMockLoggedIn =
mock(HttpServletRequest.class);
    HttpServletRequest requestMockNotLoggedIn =
mock(HttpServletRequest.class);
    LoginService service = new LoginServiceImpl();

    LoginService loginService = mock(LoginServiceImpl.class);

    when(loginService.getUsername(requestMockNotLoggedIn)).thenReturn(Optional.empty());

    when(loginService.getUsername(requestMockLoggedIn)).thenReturn(Optional.of(
"test@test.com"));

    assertAll(
        () ->
assertNotNull(loginService.getUsername(requestMockNotLoggedIn)),
//vérifier que lorsqu'on appelle la fonction getUsername, il y a bien un
retour

        () ->
assertNotNull(loginService.getUsername(requestMockLoggedIn)),
//vérifier que lorsqu'on appelle la fonction getUsername, il y a bien un
retour

        () ->
```

```

assertTrue(loginService.getUsername(requestMockLoggedIn).isPresent()),
//Verifier que lorsque l'utilisateur est authentifié, l'optional avec
l'username est present
        () ->
assertFalse(loginService.getUsername(requestMockNotLoggedIn).isPresent())
//Verifier que lorsque l'utilisateur n'est pas authentifié, l'optional avec
l'username n'est pas present
        );
}



```





Couverture d'au moins 60%








En effectuant les tests de toutes les fonctionnalités de mon projet on observe une couverture de **87%** (7/8) pour les méthodes et de **88%** (45/51) pour les lignes.

J'ai utilisé JaCoCo qui est par défaut intégré dans mon IDE IntelliJ Idea.

Voici le tableau de couverture JaCoCo lors de l'exécution des tests.

Coverage: ServicesTest  

Element ▲	Class, %	Method, %	Line, %
▼  org.hposadas.bibliotheque.services	100% (3/3)	87% (7/8)	88% (45/51)
 LivreService	100% (0/0)	100% (0/0)	100% (0/0)
 LivreServiceImpl	100% (1/1)	100% (5/5)	97% (35/36)
 LoginService	100% (0/0)	100% (0/0)	100% (0/0)
 LoginServiceImpl	100% (1/1)	0% (0/1)	16% (1/6)
 UserService	100% (0/0)	100% (0/0)	100% (0/0)
 UtilisateurServiceImpl	100% (1/1)	100% (2/2)	100% (9/9)

Ainsi que le rapport de l'exécution des tests :

The screenshot shows the IntelliJ IDEA interface with the 'ServicesTest' test results. The 'Coverage' tab is active, displaying a list of test methods and their execution times. The 'Test Results' tab shows the command used to run the tests: 'C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe ...'.

Test Method	Execution Time
Recherche des livres par titre.	411 ms
Creation du compte utilisateur	271 ms
Afficher les livres.	368 ms
Recherche des livres par ID.	121 ms
Verifier l'authentification de l'utilisateur	1 sec 866 ms

Test Results: Tests passed: 5 of 5 tests – 3 sec 37 ms

Command: "C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe" ...

IntelliJ IDEA coverage runner

sampling ...

include patterns:

org\.hposadas\.bibliotheque\.services\.*

exclude patterns:

Class transformation time: 0.121301s for 2740 classes or 4.4270437956204385E-5s per class

Process finished with exit code 0

Tests d'intégration

Pour la partie Test d'intégration j'ai utilisé la Dependency **Cucumber** ainsi que des plugins nécessaires pour le bon fonctionnement et intégration.

Pour les dépendances j'ai rajouté les dépendances suivantes à mon projet Maven :

```
<groupId>io.cucumber</groupId>
<artifactId>cucumber-java</artifactId>
<version>6.9.1</version>
</dependency>
```

```

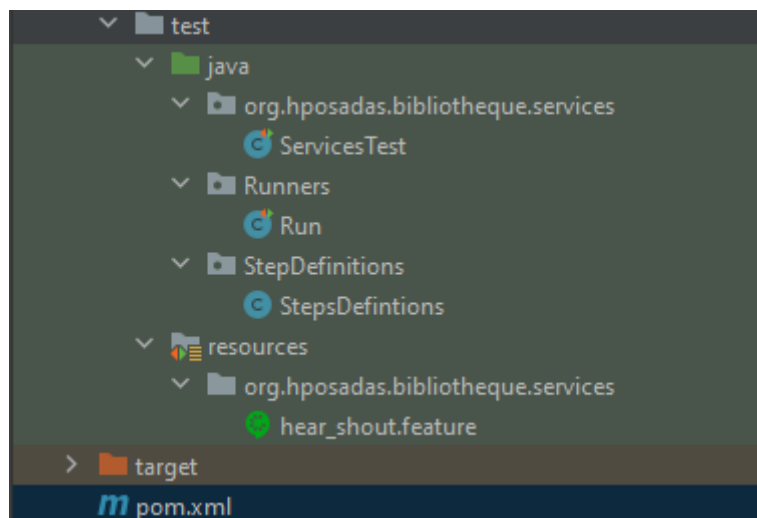
<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-testng -->
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-testng</artifactId>
  <version>7.3.3</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-simple -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>2.0.5</version>
  <scope>test</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-core -->
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-core</artifactId>
  <version>7.9.0</version>
</dependency>

```

Les plugins nécessaires rajoutés à mon IDE IntelliJ IDEA sont Gherkin et Cucumber for Java.

Structure de dossiers de test

Lorsqu'on utilise Cucumber c'est très important de respecter une structure pour ne pas se perdre, cette structure est la suivante :



Dans le package **Runners** il se trouve la classe **Run** qui sert à faire le lien entre le fichier **feature** qui contient les différents scénarios de test d'intégration et le fichier **StepDefinitions** trouvé dans le package **StepDefinitions**, ce fichier contient le code correspondant à chaque ligne de scénario défini dans le fichier **feature**.

Contenu du fichier **Feature** :

Le fichier Feature contient les scénarios d'interaction nécessaires pour que l'utilisateur puisse effectuer toutes les activités tels que créer un compte, afficher la liste des livres dans la bibliothèque, demander un livre en prêt et chercher un livre par son nom.

Afficher la liste des livres

```

Scenario: Show the books list in the library
  Given User is in the homepage

```



```
When user clicks into the Parcourir la bibliotheque button
Then user is able to see the books list
```

Créer une nouvelle compte utilisateur

```
Scenario: Create a new User account
  Given User lands in the homepage
  When user clicks into the Parcourir la bibliotheque button
  And User has filled all the information inputs with valid information
  And User clicked in the Créer button
  Then Account has been created
```

Recherche des livres par son titre

```
Scenario: search a book by his title
  Given User lands in the homepage
  When User Clicks in the parcourir la bibliotheque button
  And User complete the recherche par titre form input
  And User clicks into the rechercher button
  Then User receive the list of occurrences between the string and book titles
```

Demander un livre en prêt

```
Scenario: Ask for a book on loan
  Given User lands in the homepage
  And User clicked in the Se connecter button
  And User filled the connection form with valid information
  Then User is successfully logged in
  And User clicked into the Parcourir la bibliotheque button
  Then user is able to see the books list
  And User clicked into the Demander en pret in the selected book
  Then User have reached his book loan
```

Contenu du fichier StepDefinitions

Ce fichier contient les fonctions qui correspondent aux phrases dans le scenario permettant de tester leur intégration.

```
package StepDefinitions;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import jakarta.servlet.http.HttpServletRequest;
import org.hposadas.bibliotheque.models.Livre;
import org.hposadas.bibliotheque.models.Utilisateur;
import org.hposadas.bibliotheque.services.*;
import org.hposadas.bibliotheque.utilities.ConnectionDB;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;
import java.util.Optional;

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

public class StepsDefintions {
```

```

        Connection conn = ConnectionDB.getConnection();

        public StepsDefintions() throws SQLException {
        }

        @Given("User is in the homepage")
        public void userIsInTheHomepage() {
            System.out.println("L'utilisateur a atterri sur la page d'accueil.");
        }

        @When("user clicks into the Parcourir la bibliotheque button")
        public void userClicksIntoTheParcourirLaBibliothequeButton() {
            System.out.println("L'utilisateur a clique sur le bouton 'Parcourir la bibliotheque'.");
        }

        @Then("user is able to see the books list")
        public void userIsAbleToSeeTheBooksList() {
            LivreService service = new LivreServiceImpl(conn);
            List<Livre> livres = service.lister();
            String message = livres.size() > 0 ?
                "L'utilisateur peut regarder la liste des livres dans la bibliotheque." :
                "L'utilisateur n'a pas reussi a avoir la liste des ivres";
            System.out.println(message);
        }

        @Given("User lands in the homepage")
        public void userLandsInTheHomepage() {
            System.out.println("L'utilisateur se trouve sur la page d'accueil");
        }

        @When("User clicks in the créer un compte button")
        public void userClicksInTheCréerUnCompteButton() {
            System.out.println("L'utilisateur a clique sur le bouton Creer un compte");
        }

        @And("User has filled all the information inputs with valid information")
        public void userHasFilledAllTheInformationInputsWithValidInformation() {
            System.out.println("L'utilisateur a rempli correctement le formulaire");
        }

        @And("User clicked in the Créer button")
        public void userClickedInTheCréerButton() {
            System.out.println("L'utilisateur a appuye sur le bouton creer");
        }

        @Then("Account has been created")
        public void accountHasBeenCreated() throws SQLException {

            Utilisateur u = new Utilisateur();
            u.setId(90L);
            u.setNom("Deloitte");
            u.setPrenom("Francois");
            u.setEmail("df@test.com");
        }
    }

```

```

        u.setPassword("12345");

        UserService<Utilisateur> userService = new
UtilisateurServiceImpl(conn);
        userService.enregistrer(u);
    }

    @When("User Clicks in the parcourir la bibliotheque button")
    public void userClicksInTheParcourirLaBibliothequeButton() {
        System.out.println("L'utilisateur se trouve sur la page
d'accueil");
    }

    @And("User complete the recherche par titre form input")
    public void userCompleteTheRechercheParTitreFormInput() {
        System.out.println("L'utilisateur a renseigne un string avec
lequel il cherche un livre");
    }

    @And("User clicks into the rechercher button")
    public void userClicksIntoTheRechercherButton() {
        System.out.println("L'utilisateur a appuye sur le bouton creer");
    }

    @Then("User receive the list of occurrences between the string ant book
titles")
    public void
userReceiveTheListOfOccurrencesBetweenTheStringAntBookTitles() throws
SQLException {
        LivreService livreService = new LivreServiceImpl(conn);
        List<Livre> livres = livreService.byName("voyage");
        String message = livres.size()>0 ?
            "L'utilisateur retrouve la liste des livres trouvee avec sa
recherche" :
            "Rien n'a ete trouve pour cette recherche";
        System.out.println(message);
    }

    @And("User clicked in the Se connecter button")
    public void userClickedInTheSeConnecterButton() {
        System.out.println("L'utilisateur a appuye sur le buton se
connecter");
    }

    @And("User filled the connection form with valid information")
    public void userFilledTheConnectionFormWithValidInformation() {
        System.out.println("L'utilisateur s'est authentifie avec ses
identifiants");
    }

    @Then("User is succesfully logged in")
    public void userIsSuccesfullyLoggedIn() {
        HttpServletRequest requestMockLoggedIn =
mock(HttpServletRequest.class);

        LoginService loginService = mock(LoginServiceImpl.class);

        when(loginService.getUsername(requestMockLoggedIn)).thenReturn(Optional.of(
"test@test.com"));

        String message =

```

```

loginService.getUsername(requestMockLoggedIn).isPresent() ?
    "L'utilisateur s'est correctement authentifié" :
    "L'utilisateur ne peut pas demander des prêts tant qu'il
n'est pas authentifié";
}

@And("User clicked into the Parcourir la bibliotheque button")
public void userClickedIntoTheParcourirLaBibliothequeButton() {
    System.out.println("L'utilisateur a appuyé sur le bouton parcourir
la bibliotheque");
}

@And("User clicked into the Demander en pret in the selected book")
public void userClickedIntoTheDemanderEnPretInTheSelectedBook() {
    System.out.println("L'utilisateur a cliqué sur le bouton demander
en pret du livre souhaité");
}

@Then("User have reached his book loan")
public void userHaveReachedHisBookLoan() {
    System.out.println("L'utilisateur a demandé le prêt d'un livre");
}
}

```

Résultats des tests d'intégration

J'ai programmé 4 tests d'intégration alors les résultats sont les suivants :

The screenshot shows the 'Run' tab of an IDE with the following test results:

Test Name	Duration	Status
runScenario["Show the books list in the library", "Test Itnegration"]	1 sec 684 ms	Passed
runScenario["Crate a new User account", "Test Itnegration Bibliotl"]	1 sec 71 ms	Passed
runScenario["search a book by his title", "Test Itnegration Bibliothequ"]	989 ms	Passed
runScenario["Ask for a book on loan", "Test Itnegration Bibliothe"]	2 sec 676 ms	Passed

Summary:
 Tests passed: 4 of 4 tests - 7 sec 933 ms
 Default Suite
 Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
 Process finished with exit code 0