Documentation technique

YANG Chen

30/11/2023

Table des matières

1 Intégration de JavaFX et Spring Boot et résolution de problèmes

1.1 Vue d'ensemble

Dans notre projet actuel, où nous utilisons JavaFX comme framework front-end et Spring comme framework back-end, nous avons rencontré un problème critique : malgré le fait que nous voulions implémenter l'injection de dépendances à travers Spring, les composants Controller et Service ont des références 'null' au moment de l'exécution. Cela indique que ces composants ne sont pas correctement identifiés et gérés par le conteneur Spring.

1.2 La nécessité de l'injection de dépendances

L'injection de dépendances (DI) est un modèle de conception logicielle qui permet de gérer les dépendances d'une classe de manière externe (par exemple, fichiers de configuration ou annotations) plutôt que de les coder en dur à l'intérieur de la classe :

- **Découplage** : réduit le couplage étroit entre les classes et améliore la modularité et la maintenabilité du code.
- **Testabilité** : facilite les tests unitaires car les dépendances peuvent être facilement modélisées ou remplacées.
- **Flexibilité et extensibilité** : permet une plus grande flexibilité dans l'extension et la modification des applications en changeant l'injection des dépendances.

1.3 Solution : Intégrer Spring Boot et JavaFX

Pour résoudre le problème de l'injection de dépendances, nous avons décidé d'intégrer Spring Boot et JavaFX :

- **Créer un projet principal Spring Boot** : nous avons configuré Spring Boot comme cadre de base pour le projet afin de profiter de ses capacités d'auto-configuration et de gestion des dépendances.
- **Intégrer JavaFX** : Dans le projet Spring Boot, nous intégrons JavaFX en ajoutant les dépendances Maven appropriées.
- **Configurer la classe principale** : Spécifiez la classe principale de votre application Spring Boot explicitement dans le fichier 'pom.xml' pour éviter les échecs de construction dus à de multiples méthodes 'main'.

1.4 Dépendances Maven et changements de version

Pendant la construction du projet, nous avons ajusté les dépendances et les versions Maven pour assurer la compatibilité avec Spring Boot et JavaFX :

- Mise à jour de la version de Spring Boot pour correspondre à JavaFX.
- Suppression du fichier 'module-info.java' car le projet ne nécessite pas l'encapsulation et la modularité du système de modules Java.

1.5 Résolution des problèmes rencontrés

1.5.1 Échec du démarrage du serveur Tomcat

Le serveur Tomcat ne démarre pas parce que le projet a été construit à l'origine pour Java 11 et qu'il utilise actuellement Java 19. La solution consiste à reconstruire le projet en utilisant Java 19.

1.5.2 Choix du stockage des données

Le plan initial était d'utiliser une base de données SQLite, mais en raison du manque de support des serveurs distants et de la nécessité de la conteneurisation Docker, nous avons décidé d'émuler la base de données en utilisant des fichiers JSON. Comme les fichiers contenus dans le fichier Jar sont en lecture seule, nous avons créé une nouvelle méthode de création de fichiers JSON dans un répertoire externe pour nous assurer que le projet fonctionnerait correctement dans différents environnements et systèmes.

1.6 Conclusion

Avec l'intégration profonde de Spring Boot et JavaFX, nous avons résolu avec succès le problème de l'injection de dépendances et amélioré la flexibilité et la maintenabilité du projet. En outre, les ajustements de la configuration Maven et les modifications du schéma de stockage des données ont permis d'assurer le bon fonctionnement et la compatibilité multiplateforme du projet.

1.7 Exemple

1.7.1 Initialisation de l'application Spring Boot

```
@SpringBootApplication
@ComponentScan(basePackages = {"com.isep.eleve.javaproject"})
public class App extends Application {
    private static ConfigurableApplicationContext context;
    private static Stage primaryStage;

    @Override
    public void init() throws Exception {
        context = SpringApplication.run(App.class);
    }
}
```

Commentaire : Ce code montre comment initialiser une application Spring Boot avec l'annotation `@SpringBootApplication` et comment scanner les composants Spring dans des packages spécifiques avec `@ComponentScan`.

1.7.2 Configuration de l'intégration JavaFX et Spring

```
FXMLLoader fxmlLoader = new FXMLLoader(App.class.getResource(LOGIN_VIEW_PATH));
fxmlLoader.setControllerFactory(context::getBean);
Parent root = fxmlLoader.load();
primaryStage.setScene(new Scene(root));
```

Commentaire : Cette section explique comment intégrer JavaFX avec Spring, permettant ainsi à Spring de gérer les contrôleurs JavaFX via l'injection de dépendances.

1.7.3 Création du contrôleur de connexion

```
@Controller
public class LoginController {
    @Autowired
    private AuthenticationService authenticationService;
}
```

Commentaire : Ici, un contrôleur de connexion est défini avec l'annotation `@Controller`, et l'authentification est gérée automatiquement par Spring à l'aide de `@Autowired`.

1.7.4 Mise en œuvre du service d'authentification

```
@Service
public class AuthenticationService {
    private final UserRepository userRepository;
    private final SecurityService securityService;

    @Autowired
    public AuthenticationService(UserRepository userRepository,
    SecurityService securityService) {
        this.userRepository = userRepository;
        this.securityService = securityService;
```

```
    }
}
```

Commentaire : Ce code illustre un service d'authentification marqué avec `@Service`, indiquant qu'il fait partie de la couche logique métier. Le constructeur injecte `UserRepository` et `SecurityService`.

1.7.5 Définition du dépôt utilisateur

```
@Repository
public class UserRepository {
}
```

Commentaire : `UserRepository` est annoté avec `@Repository`, indiquant qu'il fait partie de la couche d'accès aux données, géré par Spring pour l'intégration avec d'autres services et contrôleurs.

1.8 Références

https://www.cnblogs.com/lingkang/p/16698421.html

2

3 Classe Outils(Tools)

3.1 Vue d'ensemble

La classe Tools est une classe utilitaire qui fournit des méthodes statiques pour effectuer des opérations de base sur les chaînes de caractères, les fichiers et les images.

3.2 Constants

3.2.1 Chemin des Vues

- `LOGIN_VIEW_PATH` : Chemin d'accès à la vue de connexion.
  Valeur : `"\com\isep\eleve\javaproject\view\LoginView.fxml"`.
- `REGISTRATION_VIEW_PATH` : Chemin d'accès à la vue d'inscription.
  Valeur : `"\com\isep\eleve\javaproject\view\RegistrationView.fxml"`.

3.2.2 Types d'Encryptage

- `MD5` : Utilisé pour l'encryptage en MD5.
- `SHA1` : Utilisé pour l'encryptage en SHA1.
- `SHA256` : Utilisé pour l'encryptage en SHA256.
- `SHA512` : Utilisé pour l'encryptage en SHA512.

3.2.3 Types d'Échec d'Inscription

- `PASSWORD_NOT_SAME` : Indique que les mots de passe entrés ne sont pas identiques.
- `USERNAME_ALREADY_EXIST` : Indique que le nom d'utilisateur choisi existe déjà.

3.3 UUIDUtils

3.3.1 Introduction

UUIDUtils est une classe utilitaire qui fournit des méthodes statiques pour générer des UUID aléatoires.

3.3.2 Méthodes

— `getUUID()` : Génère un UUID aléatoire.
Paramètres : Aucun.
Valeur de retour : `String` - UUID aléatoire.
— `getUUIDInOrderID()` : Génère un UUID aléatoire qui contiant que des chiffres.
Valeur de retour : `int` - UUID aléatoire.

3.3.3 Exemple

```
String uuid = UUIDUtils.getUUID();
int uuidInOrderID = UUIDUtils.getUUIDInOrderID();
```

3.4 EncryptUtils

3.4.1 Introduction

EncryptUtils est une classe utilitaire qui fournit des méthodes statiques pour effectuer des opérations d'encryptage sur les chaînes de caractères.

3.4.2 Méthodes

— `encrypt(String str, String type)` : Effectue un encryptage sur la chaîne de caractères spécifiée.
Paramètres :
— `str` - Chaîne de caractères à encrypter.
— `type` - Type d'encryptage.
Valeurs possibles : `MD5`, `SHA1`, `SHA256`, `SHA512`.
Valeur de retour : `String` - Chaîne de caractères encryptée.

3.4.3 Exemple

```
encryptedP = EncryptUtils.encrypt(password, Constants.EncryptType.MD5);
```

3.5 FileOperation

3.5.1 Introduction

La classe FileOperation fournit un ensemble d'outils pour gérer les opérations de lecture et d'écriture de fichiers, en particulier pour les données au format JSON. Cette classe utilise la bibliothèque Jackson pour sérialiser et désérialiser des listes d'objets.

3.5.2 Constructeur

— `FileOperation(ObjectMapper objectMapper)` : Crée un objet FileOperation avec le objectMapper.
Paramètres :
— `objectMapper` - Instance de Jackson ObjectMapper pour le traitement JSON.

3.5.3 Méthodes

— `public <T> List<T> readListFromFile(String filePath, Class<T> clazz)` : Lit une liste d'objets à partir d'un fichier JSON.
— **Paramètres :**
— `filePath` - Chemin d'accès au fichier JSON.
— `clazz` - Classe de l'objet à lire.
— **Valeur de retour :** `List<T>` - Liste d'objets lus.
— **Exceptions :** `IOException` - Si une erreur d'entrée/sortie se produit lors de la lecture du fichier.

— `public <T> void writeListToFile(String filePath, List<T> list)` : Écrit une liste d'objets dans un fichier JSON.
  — **Paramètres :**
    — `filePath` - Chemin d'accès au fichier JSON.
    — `list` - Liste d'objets à écrire.
  — **Exceptions :** `IOException` - Si une erreur d'entrée/sortie se produit lors de l'écriture du fichier.

3.5.4 Exemple

Dans les applications Spring Boot, la classe FileOperation est automatiquement assemblée. Par conséquent, les instances de cette classe peuvent être injectées directement dans le service ou le contrôleur.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final FileOperation fileOperation;

    @Autowired
    public MyService(FileOperation fileOperation) {
        this.fileOperation = fileOperation;
    }

    public void exampleMethod() {
        List<User> users = fileOperation.
        readListFromFile("users.json", User.class);
        fileOperation.writeListToFile("users.json", users);
    }
}
```