

Projet JDR

Rapport

Présentation du projet

JDR est un jeu de rôle où des personnages appartenant à différentes classes affrontent divers types de monstres. Chaque classe dispose de caractéristiques propres. Un système d'objets est également intégré, permettant d'améliorer les caractéristiques d'attaque des joueurs. Lorsqu'un joueur attaque, les dégâts infligés au monstre sont calculés en combinant les caractéristiques de base du joueur avec celles des objets qu'il possède.

Sommaire

Projet JDR	1
Présentation du projet.....	2
Sommaire	3
1 – Les choix technologiques	4
2 – Installation	5
2.0 – Prérequis	5
2.1 – Déploiement	5
2.2 – Exécution des tests.....	6
3 – Choix de réalisation des tests	7
4 – Processus de tests	8
5 – Description des tests Selenium.....	9
5.1 – Test 1 : Vérification de l’affichage des caractéristiques	9
5.2 – Test 2 : Absence de nom dans l’input.....	9
5.3 – Test 3 : Absence de classe dans l’input.....	9
5.4 – Test 4 : Ouverture du plateau de jeu.....	9
5.5 – Test 5 : Combat contre le rat	9

1 – Les choix technologiques

L'application a été développée en trois parties :

- **Backend** : avec le framework Java Spring Boot
Les tests ont été réalisés à l'aide de
 - Spring Test Suite
 - JUnit
 - Mockito
 - H2 (pour faire office de base de données)
- **Frontend** : avec le framework Javascript React (à l'aide de Vite.JS)
Les tests ont été réalisés à l'aide de
 - Jest (et Jest Dom)
 - React Test Suite (React testing library)
 - ESLint
- **Tests Selenium** : avec unittest en Python

Le déploiement de l'application s'effectue à l'aide de Docker.

Pour éviter tout problème, les deux images pré-construites ont été ajoutées au projet.

Les liens de téléchargement sont présent dans le fichier README du projet.

Si, lors de l'installation, vous obtenez une erreur en vérifiant les logs des containers, se référer à la [documentation officielle de Docker](#) pour les importer.

2 – Installation

2.0 – Prérequis

Il est recommandé de lancer l'application sous n'importe quelle distribution Linux car quelques problèmes de compatibilités avec Docker pour Windows (notamment les installations avec Docker Desktop) ont été remarqués lors des développements.

Le déploiement a été testé sous Debian 12.

- Docker Engine (développements réalisés sous la version 20.10.24+dfsg1)
 - o [Documentation d'installation pour Linux](#)
- Docker Compose v2 (la version Go)
 - o [Documentation d'installation pour Linux](#)
- Python 3.11 (pour l'exécution des tests Selenium)
 - o [Page de téléchargement de Python](#)
- JDK 23.0.1 (pour l'exécution des tests Java)
 - o [Page des archives JDK d'Oracle](#)
- NodeJS 23.4.0 / NPM ^10.9.2
 - o [Site de NodeJS maintenu par l'OpenJS Foundation](#)
- Make (optionnel)
 - o Présent dans la majorité des repositories pour toute disto Linux sous le nom « make »
- N'avoir aucun processus écoutant les ports TCP 5173 et TCP 8080

2.1 – Déploiement

Le build du frontend et du backend est géré par la construction des images Docker.

Les étapes de cette construction peuvent être trouvées dans le fichier **Dockerfile** de chaque partie. De plus, leurs tests respectifs sont exécutés lors de ce build et bloque le déploiement de l'application s'ils échouent.

Le déploiement a été pensé pour être simple et ne consiste qu'en une seule commande.

Déploiement	Arrêt
Solution avec Make \$ make up	Solution avec Make \$ make down
Solution sans Make \$ docker compose up -d --build	Solution sans Make \$ docker compose down --remove-orphans

2.2 – Exécution des tests

Concernant les tests Selenium, il est recommandé de déployer un venv et d'installer les dépendances du requirements.txt.

Pour lancer les tests manuellement, voici pour chaque partie la commande à exécuter :

Backend	Frontend	Selenium
Solution avec Make \$ make testJava	Solution avec Make \$ make testJs	Solution avec Make \$ make testSelenium
Solution sans Make \$./back/gradlew -p ./back test	Solution sans Make \$ npm --prefix ./front test	Solution sans Make \$ python3 -m unittest discover - s ./python-selenium/tests/

3 – Choix de réalisation des tests

Initialement, et afin de redécouvrir les technologies et de garantir que les développements pourraient aboutir, le projet n'avait pas suivi une approche TDD (Test-Driven Development), notamment sur la partie Java. Il a été décidé en amont de par la suite repartir de zéro pour intégrer progressivement les fonctionnalités métier et techniques schématisées, tout en ajoutant des classes et des méthodes de test au fur et à mesure.

Toujours côté Java, l'implémentation avec Mockito a aidé à faciliter le processus. Il a été décidé de ne pas mettre de tests sur les controllers, car ils ne font office que de passe-plat.


Cependant, il a été décidé d'instaurer des tests sur l'intégralité des **records**, utilisés dans les différents endpoints, pour s'assurer de la conformité des modèles de données d'input de l'extérieur et des différents blocages dans le cas inverse, et sur l'intégralité des **services** car ils renferment pour ce projet le plus gros de la logique métier et traitent avec les repositories (et donc avec la base de données)..

Pour le frontend, nous avons opté pour Jest afin de réaliser des tests unitaires sur les composants. L'objectif était de vérifier que les interactions avec l'utilisateur fonctionnaient comme prévu et que les données s'affichaient correctement. Étant donné que la logique métier est gérée principalement par le backend, les tests frontend se sont concentrés sur l'affichage des informations et le comportement des composants. Par exemple, pour la création du personnage (composant `CreationPersonnage`), nous avons vérifié que des messages d'erreur apparaissent si les champs obligatoires (nom et classe) sont laissés vides. Cela nous a poussés à structurer le code pour intégrer cette validation dès le départ et garantir une meilleure expérience utilisateur.

4 – Processus de tests

Pour le backend, l'écriture des tests en parallèle de la refonte du code a drastiquement changé la conception du projet.

Au départ, les méthodes écrites n'étaient pas pleinement réfléchies et assez abstraites, mais par la suite elles sont devenues bien plus claires et logiques.

Par exemple, les tests sur la partie aléatoire,  **chance** (ou « luck » du personnage) ont permis, en suivant la méthodologie du TDD, de s'assurer de son bon fonctionnement et de prévenir les éventuelles erreurs qui pourraient survenir côté Frontend.

La première partie consistait uniquement à mocker la méthode `hasLuck` de la classe `CharacterService` afin qu'elle retourne la valeur désirée, afin de s'assurer que les méthodes l'utilisant répondent correctement.

Ensuite, des tests ont été instaurés sur cette méthode, en testant chaque cas indépendamment tout en restant certain que le test passerait en toute circonstances.

Écrire et exécuter les tests pendant le développement a clairement influencé la façon dont nous avons écrit notre code frontend. Au début, les composants étaient simples, mais au fur et à mesure des tests, le code est devenu plus clair et plus fiable. Par exemple, les tests ont rapidement montré qu'il était essentiel d'ajouter une validation des champs avant de permettre à l'utilisateur de continuer. Cela nous a poussés à mettre en place une vérification systématique des entrées dans le formulaire de création de personnage, empêchant ainsi la progression en cas de données manquantes. En travaillant avec Jest, nous avons pu tester chaque modification du composant au fur et à mesure, ce qui nous a permis de garantir que l'interface répondait bien aux attentes tout en simplifiant le développement.

5 – Description des tests Selenium

5.1 – Test 1 : Vérification de l’affichage des caractéristiques

Ce test vérifie que l’encart avec les caractéristiques du personnage s’affiche correctement après que l'utilisateur ait saisi les informations nécessaires. On s'assure aussi que les caractéristiques (Force, Intelligence, Chance) sont bien calculées et affichées avec les bonnes valeurs par défaut.

5.2 – Test 2 : Absence de nom dans l’input

Ici, on teste ce qui se passe si l'utilisateur essaie de continuer sans entrer un nom. Une alerte doit apparaître pour lui demander de renseigner un nom et une classe. On vérifie également que le plateau de jeu ne s’affiche pas, garantissant que l'utilisateur ne peut pas continuer tant que les informations essentielles ne sont pas saisies.

5.3 – Test 3 : Absence de classe dans l’input

Ce test vérifie le comportement de l'application si l'utilisateur ne choisit pas de classe. Comme dans le test précédent, une alerte est attendue pour rappeler à l'utilisateur de choisir une classe avant de continuer. Cela empêche le processus de création de personnage de se poursuivre sans cette information.

5.4 – Test 4 : Ouverture du plateau de jeu

Une fois que le personnage est créé, on vérifie que le plateau de jeu s'affiche correctement. On s'assure que le titre "Combat contre un monstre" est bien visible, et que les informations du personnage (nom et classe) sont correctement affichées sur le plateau.

5.5 – Test 5 : Combat contre le rat

Ce test simule un combat contre un rat. On commence par vérifier que le personnage commence avec 100 points de vie. Ensuite, on s’assure que le texte indiquant l’apparition du rat ("Un RAT apparaît ! Préparez-vous au combat.") est bien affiché. On vérifie aussi que les points de vie du personnage et du rat diminuent correctement à chaque tour de combat.