TPIA4GL - l'automatisation des tests fonctionnels basé sur l'IA (LLM)

HAI913I

2025-2026

Auteurs

BENOMAR Fadel | N° étudiant : 22015967
M2 Génie Logiciel

Table des matières

Introduction

La qualité logicielle est un enjeu crucial, et les tests fonctionnels sont une étape indispensable pour valider la conformité aux exigences métier. Traditionnellement, l'écriture et la maintenance de ces tests, notamment des scénarios en langage naturel type Gherkin et des scripts d'automatisation type Selenium, exigent un investissement conséquent en temps et en ressources. Cette charge est amplifiée par l'évolution rapide des projets agiles.

Dans ce tp nous explorerons l'utilisation des modèles de langage de grande taille, comme outils d'assistance pour le processus de test fonctionnel. L'objectif est d'évaluer concrètement la capacité de l'IA à accélérer la production de scénarios de tests, à générer des scripts d'exécution automatisés, et à maintenir la cohérence entre les spécifications et le code de test, sans pour autant remplacer l'expertise humaine.

L'approche repose sur un cas d'étude concret qui constitue une application de gestion de produits avec authentification, implémentée avec une architecture moderne Spring Boot API RESTful, React/Material-UI, MongoDB. Ce contexte technique représentatif nous permet d'explorer les tests d'API, d'interface utilisateur et les scénarios de bout en bout.

Ce rapport détaillera la démarche suivie, de l'analyse du cahier des charges à l'implémentation d'une suite de tests automatisés. Nous analyserons les résultats, les apports et les limites de cette assistance par IA, fournissant ainsi une analyse critique et pédagogique sur les enjeux de l'automatisation des tests fonctionnels dans un contexte professionnel moderne.

Le code source complet du projet, incluant l'application et la suite de tests automatisés, est disponible sur GitHub à l'adresse suivante : `https://github.com/fb2001/automatisation_des_tests_fonctionnels-`

Chapitre 1

Fondements théoriques , Tests fonctionnels et approche BDD

1.1 Les tests fonctionnels dans le cycle de développement logiciel

Les tests fonctionnels constituent une catégorie fondamentale de tests logiciels, distincte des tests unitaires ou des tests d'intégration par leur niveau d'abstraction et leur objectif. Alors que les tests unitaires se concentrent sur la validation de composants isolés du code, généralement au niveau des fonctions ou des classes, les tests fonctionnels adoptent une perspective de plus haut niveau, centrée sur le comportement observable du système du point de vue de l'utilisateur final ou du métier. Cette distinction est essentielle pour comprendre leur rôle dans la stratégie globale de qualité logicielle.

Un test fonctionnel ne s'intéresse pas aux détails d'implémentation interne du code. Il ne vérifie pas comment une fonction effectue un calcul ou comment une classe gère ses données internes. Au contraire, il se concentre sur une question fondamentale, étant donné un contexte initial défini, lorsqu'une action spécifique est effectuée par un utilisateur, le système produit-il le résultat attendu conformément aux exigences métier ? Cette approche black-box permet de valider que le logiciel répond effectivement aux besoins pour lesquels il a été conçu, indépendamment des choix techniques d'implémentation qui peuvent évoluer au fil du temps.

L'importance des tests fonctionnels dans le cycle de développement moderne ne peut être sous-estimée. Dans un contexte agile où les fonctionnalités sont développées de manière itérative et incrémentale, ces tests constituent une forme de documentation vivante des exigences métier. Ils permettent de détecter les régressions fonctionnelles lors des évolutions du code, garantissant qu'une nouvelle fonctionnalité n'introduit pas de dysfonctionnement dans les comportements existants. De plus, ils facilitent la communication entre les différents acteurs du projet, développeurs, testeurs, product owners et utilisateurs finaux peuvent tous comprendre et discuter d'un scénario de test fonctionnel exprimé en termes métier, sans nécessiter de compétences techniques approfondies. Cependant, les tests fonctionnels présentent également des défis spécifiques. Leur exécution est généralement plus lente que celle des tests unitaires, car ils impliquent l'activation de composants multiples du système, voire la simulation d'interactions utilisateur complètes à travers une interface graphique. Leur maintenance peut s'avérer coûteuse lorsque l'interface utilisateur ou l'API évoluent fréquemment. De plus, la rédaction de tests fonctionnels nécessite une compréhension approfondie du domaine métier et des cas d'usage réels, ce qui requiert une collaboration étroite entre les équipes techniques et métier.

1.2 L'approche Behavior Driven Development "BDD"

Face aux défis de communication et de compréhension mutuelle entre équipes techniques et métier, l'approche Behavior Driven Development a émergé comme une évolution naturelle des pratiques de développement piloté par les tests "Test Driven Development". Le BDD ne se limite pas à une simple méthodologie de test, mais représente plutôt une approche du développement logiciel où le comportement attendu du système est placé au centre des décisions.

Le langage Gherkin, qui constitue l'outil central du BDD. Sa syntaxe structurée autour des mots-clés Given, When, Then correspond directement à la façon dont nous décrivons des comportements ,nous définissons d'abord un contexte initial, nous décrivons ensuite une action ou un événement, et nous énonçons finalement les conséquences ou résultats attendus. Cette structure logique permet de capturer non seulement ce que le système doit faire, mais aussi pourquoi il doit le faire, en rendant explicites les règles métier sous-jacentes.

Considérons le scénario suivant en Gherkin, "Étant donné que je suis un utilisateur enregistré avec l'email **alice@test.com**, lorsque je me connecte avec le bon mot de passe, alors je devrais voir mon nom affiché dans la barre de navigation et avoir accès à la liste de mes produits." Ce scénario capture une exigence fonctionnelle claire sans aucune référence aux détails techniques , il ne mentionne ni les endpoints d'API, ni les composants React, ni les requêtes HTTP. Pourtant, il décrit précisément un comportement attendu qui peut être validé de manière automatisée.

L'un des avantages majeurs du BDD réside dans le fait que les scénarios Gherkin peuvent servir simultanément de spécification fonctionnelle, de documentation et de tests automatisés. Les outils comme Cucumber permettent de lier ces scénarios à du code d'implémentation "les step definitions" qui effectue les actions décrites et vérifie les assertions. Cette approche garantit que la documentation reste synchronisée avec le code réel, puisqu'un scénario qui ne correspond plus au comportement du système échouera lors de son exécution.

1.3 Outils et frameworks, Cucumber et Selenium

L'écosystème BDD s'appuie sur deux catégories d'outils complémentaires, d'une part, les frameworks d'exécution de scénarios comme **Cucumber** qui permettent d'interpréter les fichiers Gherkin et de les lier au code de test, et d'autre part, les outils d'automatisation d'interactions comme **Selenium** qui permettent de simuler les actions d'un utilisateur réel sur une interface web.

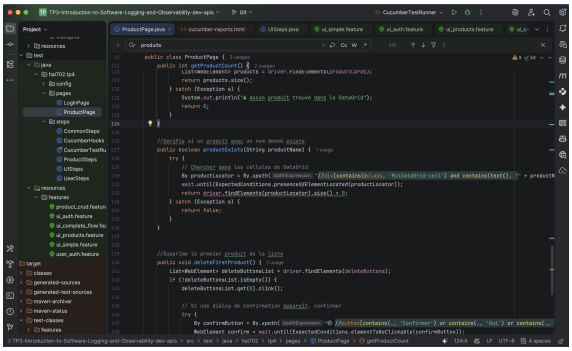
Lorsqu'un scénario Gherkin est exécuté, Cucumber analyse chaque étape "Given, When, Then" et recherche dans les step definitions une méthode Java correspondante dont l'annotation contient l'expression régulière ou le texte exact de l'étape. Cette méthode contient le code technique qui effectue réellement l'action décrite. Prenons l'exemple l'étape "When je me connecte avec l'email 'alice@test.com' et le mot de passe 'admin123'" sera liée à une méthode Java annotée qui effectuera un appel HTTP POST vers l'endpoint de connexion du backend avec les paramètres appropriés.

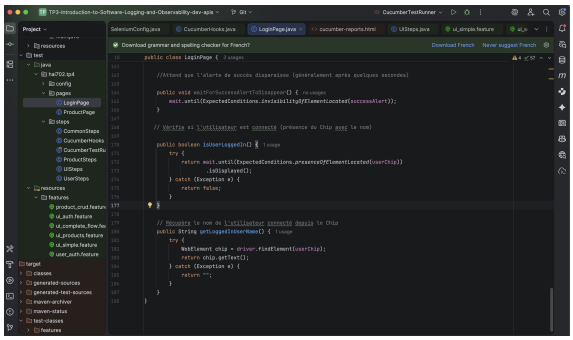
Cette séparation entre la description du comportement les features Gherkin et l'implémentation technique les step definitions présente plusieurs avantages. Elle permet aux scénarios de rester stables même lorsque l'implémentation technique change, si l'architecture du backend évolue, seules les step definitions doivent être modifiées, pas les features. Elle facilite éga-

lement la réutilisation, une même étape Gherkin peut être utilisée dans plusieurs scénarios différents, et son implémentation technique sera partagée. Enfin, elle améliore la lisibilité et la maintenabilité en évitant de mélanger la logique métier et les détails techniques.

Selenium WebDriver représente l'outil de référence pour l'automatisation des tests d'interface utilisateur web. Contrairement aux outils d'enregistrement-rejeu simplistes, Selenium offre un contrôle programmatique fin sur les navigateurs web réels, permettant de simuler des interactions utilisateur complexes, navigation entre pages, saisie de formulaires, clics sur des boutons, vérification de l'état de la page, attente de conditions spécifiques. Selenium prend en charge tous les navigateurs majeurs et permet d'écrire des tests robustes et fiables.

L'utilisation de Selenium présente toutefois des défis spécifiques. Les tests UI sont intrinsèquement plus fragiles que les tests d'API car ils dépendent de la structure du DOM, qui peut varier selon le framework frontend utilisé, et du comportement asynchrone de JavaScript. Pour atténuer ces problèmes, nous avons adopté le pattern Page Object Model, une pratique recommandée dans l'industrie. Ce pattern consiste à encapsuler les interactions avec une page web dans une classe Java dédiée, qui expose des méthodes métier de haut niveau et masque les détails techniques des sélecteurs CSS ou XPath. Ainsi, si la structure HTML de la page change, seule la classe Page Object doit être modifiée, pas tous les tests qui l'utilisent. Dans notre implémentation, nous avons créé deux Page Objects principaux, **LoginPage** pour gérer l'authentification et **ProductPage** pour gérer les opérations CRUD sur les produits. Chaque Page Object contient les sélecteurs des éléments de la page, des méthodes pour interagir avec ces éléments "fillLoginForm, clickLogin", et des méthodes pour vérifier l'état de la page "isUserLoggedIn, productExists". Cette organisation permet aux step definitions Cucumber de rester concises et lisibles, se concentrant sur la logique du test plutôt que sur les détails techniques de Selenium.





Chapitre 2

Intelligence artificielle et automatisation des tests : état de l'art

2.1 Les modèles de langage dans le génie logiciel

Les Modèles de Langage de Grande Taille apportent une assistance significative dans les tests fonctionnels grâce à leur double capacité de compréhension et de génération. Leur aptitude à traiter le langage naturel leur permet d'analyser des descriptions fonctionnelles exprimées en termes métier pour les traduire en scénarios de tests structurés, notamment au format Gherkin. Simultanément, leur connaissance approfondie des frameworks de l'écosystème comme Cucumber ou Selenium leur permet de générer rapidement du code syntaxiquement correct et souvent basé sur les bonnes pratiques. Néanmoins, il est impératif de considérer que les LLMs ne constituent pas une solution autonome. Ils opèrent sur des patterns statistiques appris de leur corpus d'entraînement, ce qui leur confère une compréhension sémantique limitée du domaine métier spécifique à l'application. Cette limite se manifeste par le risque de produire du code qui, bien que techniquement viable, pourrait omettre ou déformer des règles métier complexes essentielles. De plus, l'efficacité des modèles est directement contrainte par leur absence de contexte réel. Un LLM ne peut pas examiner la base de code complète, l'architecture précise du système en cours, ni interroger les utilisateurs finaux. Par conséquent, leur utilité dépend entièrement de la qualité et de la précision des informations fournies par l'ingénieur en tests via le prompt initial. Un cahier des charges vague produira inévitablement des tests incomplets, soulignant que l'IA est un outil d'accélération et non de substitution à l'analyse métier humaine. Enfin, en conséquence directe de ces limitations, tout code généré par l'IA doit être considéré comme un brouillon : il requiert systématiquement une révision, une validation, et une maintenance humaine pour garantir qu'il couvre les cas critiques et qu'il est en phase avec les exigences fonctionnelles réelles.

2.2 Cas d'Usage Pertinents de l'IA dans les Tests Fonctionnels

Un premier cas d'usage majeur concerne la génération des scénarios Gherkin à partir de la spécification fonctionnelle, fournissant une base solide de scénarios comme l'authentification et les opérations CRUD qui a réduit significativement le temps de rédaction initiale. Dans la continuité, l'IA a prouvé son efficacité pour l'implémentation des step definitions Cucumber, proposant des structures Java appropriées, notamment pour les tests d'API backend.

Un autre apport clé réside dans la génération du code Selenium et des Page Object Models.

L'écriture de sélecteurs XPath ou CSS robustes pour les interfaces complexes comme celle de Material-UI. l'IA, en s'appuyant sur sa connaissance des structures DOM fréquentes, a pu proposer des sélecteurs plus maintenables. Enfin, l'assistance IA s'est montrée utile pour la gestion des aspects techniques du framework de tests configuration Cucumber-Spring Boot, WebDriverManager, permettant de se concentrer sur la logique métier plutôt que sur les détails d'intégration.

2.3 Limites et Considérations Éthiques

Au-delà de savoir si l'outil d'automatisation fonctionne ou non, nous devons nous poser une question essentielle, en tant que futurs ingénieurs, quel est notre rôle si la machine fait une grande partie du travail ? Le danger le plus réel est de perdre nos réflexes. Quand l'outil nous donne un gros bloc de code complexe, il est tentant de l'accepter sans vraiment comprendre comment il fonctionne. Cette facilité peut vite nous rendre dépendants, et le jour où le code plante ou doit être mis à jour, nous sommes perdus parce que nous n'avons pas la connaissance en tête. C'est pourquoi, dans ce TP, nous avons insisté pour ne pas être passifs : chaque morceau de code proposé par l'assistance, nous l'avons décortiqué. Notre objectif est clair : l'outil est là pour aller plus vite, pour faire le gros du travail répétitif, mais il doit rester un accélérateur de notre apprentissage, jamais un remplaçant de notre expertise technique.

Il faut également faire preuve d'un sens critique permanent. Les outils d'aide sont créés par des humains et font des erreurs : ils peuvent proposer des solutions qui ne sont pas les plus propres, qui datent un peu, ou qui oublient des aspects de sécurité cruciaux. C'est pour cette raison qu'on ne peut jamais se permettre d'intégrer du code sans le valider nous mêmes rigoureusement. C'est notre responsabilité de garantir la qualité finale, s'habituer trop vite à ne compter que sur un seul outil nous fragilise. Si cet outil n'est plus disponible ou ne comprend pas un contexte technique très spécifique, nous devons pouvoir revenir à nos fondamentaux. En clair, il est vital de continuer à muscler nos compétences de base en programmation et en analyse, car c'est cela qui nous rendra toujours irremplaçables face à n'importe quelle technologie.

Chapitre 3

Présentation du cas d'étude et analyse du cahier des charges

3.1 Description fonctionnelle du système

Le cas d'étude proposé pour ce travail pratique consiste en une application complète de gestion de produits avec système d'authentification utilisateur. Bien que volontairement simplifiée, cette application intègre néanmoins les principales caractéristiques que l'on retrouve dans des systèmes d'information réels, séparation frontend-backend, API RESTful, persistance de données, gestion d'état utilisateur, et opérations CRUD complètes.

L'application répond à un besoin simple mais représentatif permettre à des utilisateurs authentifiés de gérer un catalogue de produits. Chaque utilisateur est caractérisé par un identifiant unique, un nom complet, un âge, un email servant d'identifiant de connexion, et un mot de passe pour sécuriser l'accès.

Les produits constituent l'entité métier centrale de l'application. Chaque produit est défini par quatre attributs, un identifiant unique permettant de le distinguer dans le système, un nom descriptif, un prix exprimé en valeur décimale, et une date d'expiration qui pourrait servir dans un contexte réel à gérer la fraîcheur de produits alimentaires ou la validité de produits périssables. Cette modélisation, bien que simple, permet de tester des opérations variées, saisie de différents types de données (texte, nombres, dates), validation de contraintes, affichage et tri de listes.

Le cahier des charges spécifie clairement les opérations métier attendues. Les utilisateurs doivent pouvoir visualiser l'ensemble des produits disponibles dans le référentiel, ce qui nécessite une interface de consultation efficace. Ils doivent pouvoir rechercher un produit spécifique par son identifiant, une opération courante lorsqu'on connaît la référence exacte d'un article. L'ajout de nouveaux produits doit être possible, avec une contrainte métier importante, l'unicité de l'identifiant doit être garantie par le système, qui lèvera une exception si on tente d'ajouter un produit avec un ID déjà existant. La suppression de produits constitue une opération sensible dans tout système de gestion. Le cahier des charges précise qu'une tentative de suppression d'un produit inexistant doit également lever une exception, plutôt que d'échouer silencieusement.De même, la mise à jour des informations d'un produit doit échouer de manière explicite si le produit ciblé n'existe pas dans le système.

Create a simple backend application that allows you to :

1. create a user with an ID, name, age, email, and password.
2. provide a user with a menu through which (s)he can :
    - display products in a repository, where every product has an ID, a name, a price, and a expiration date.
    - fetch a product by its ID (if no product with the provided ID exists, an exception must be thrown).
    - add a new product (if a product with the same ID already exists, an exception must be thrown).
    - delete a product by its ID (if no product with the provided ID exists, an exception must be thrown).
    - update a product's info (if no product with the provided ID exists, an exception must be thrown).

The backend can be consumed with a simple frontend CLI. Also, adding a database (e.g., MongoDB) is a bonus, which would be very beneficial for the second bonus exercise.

FIGURE 3.1 – Cahier de charge

3.2 Architecture technique retenue

L'architecture technique que nous avons mise en oeuvre pour répondre à ce cahier des charges s'appuie sur une séparation claire entre trois couches principales, **le backend** responsable de la logique métier et de la persistance, **le frontend** gérant l'interface utilisateur, et la couche de tests validant le bon fonctionnement de l'ensemble.

Le backend a été développé avec Spring Boot, un choix technique qui s'impose naturellement dans l'écosystème Java moderne pour le développement d'API RESTful. Notre backend expose plusieurs endpoints REST : /api/users/register pour l'inscription de nouveaux utilisateurs, /api/users/login pour l'authentification, et une série d'endpoints /api/products pour les opérations CRUD sur les produits. La communication avec le frontend se fait via des requêtes HTTP JSON, un standard de l'industrie pour les API web modernes.

La persistance des données est assurée par MongoDB, une base de données NoSQL orientée documents. L'intégration avec Spring Boot via Spring Data MongoDB est transparente et permet de bénéficier du pattern Repository avec très peu de code.

Le frontend a été développé avec React, pour la construction d'interfaces utilisateur modernes. Nous avons choisi d'utiliser Material-UI comme framework de composants, ce qui nous fournit une collection riche de composants pré-stylisés (TextField, Button, Card, DataGrid) respectant les principes du Material Design de Google.

Le frontend communique avec le backend via l'API Fetch standard de JavaScript, en gérant l'état local avec les hooks React "useState, useEffect".

3.3 Identification des exigences testables

L'analyse du cahier des charges nous a permis d'identifier plusieurs catégories d'exigences testables, chacune nécessitant des approches de test spécifiques. Cette identification constitue une étape obligatoire dans la démarche de test, car elle permet de structurer le plan de test et de s'assurer d'une couverture exhaustive des fonctionnalités.

- **Exigences d'Authentification (Tests Fondamentaux)**

Ce groupe de tests valide le bon fonctionnement du système de connexion et d'inscription. Nous devons couvrir les cas de succès et les cas d'échec :

- **Cas de Succès :** Inscription réussie d'un nouvel utilisateur et connexion valide avec des identifiants corrects.
- **Cas d'Échec (Négatifs) :** Connexion avec un email inexistant ou un mot de passe incorrect ; tentative d'enregistrement avec un email déjà utilisé (vérification du message d'erreur et du refus d'accès).

- **Opérations CRUD sur les Produits**

  Les opérations CRUD (Create, Read, Update, Delete) génèrent de nombreux cas de test, y compris la validation des contraintes :

  - **Lecture (Read) :** Vérification de la récupération de la liste complète des produits et de la récupération d'un produit spécifique par son ID.
  - **Création (Create) :** Validation de l'ajout de produits avec des données valides (types de données corrects) et surtout le respect de la **contrainte d'unicité** sur l'identifiant.
  - **Mise à Jour (Update) :** Vérification de la modification et de la persistance des attributs d'un produit existant, ainsi que le traitement d'erreur si le produit est inexistant.
  - **Suppression (Delete) :** Validation de la suppression d'un produit existant et vérification que la tentative de suppression d'un produit inexistant lève bien une exception.

- **Scénarios de Bout en Bout "E2E"**

  Ces tests sont essentiels car ils valident des flux utilisateur complets, garantissant que toutes les parties du système fonctionnent correctement ensemble :

  - Validation d'un flux typique : inscription → connexion → série d'opérations CRUD → déconnexion.
  - Ces tests, bien que plus longs, confirment la validité des flux utilisateur réels à travers toutes les couches de l'architecture.

Chapitre 4

Démarche Proposée - Méthodologie d'Automatisation Assistée par IA

4.1 Workflow et Rôle de l'Assistance IA

Notre méthodologie repose sur un workflow itératif associant systématiquement la rapidité de génération de l'IA à la validation humaine. La démarche a été structurée en trois phases principales. La première phase consistait à fournir au modèle de langage le cahier des charges pour générer des scénarios Gherkin initiaux. Ces scénarios ont ensuite été soumis à une révision pour vérifier leur pertinence métier et y ajouter manuellement les cas d'erreur critiques. La deuxième phase a utilisé l'IA pour générer le squelette des step definitions Java correspondantes, en incluant des détails contextuels sur les frameworks Spring Boot, TestRestTemplate. Bien que le code généré soit techniquement cohérent, il nécessitait des ajustements fins pour correspondre aux noms d'attributs JSON et aux endpoints spécifiques de notre backend. La troisième phase, la plus complexe, concernait l'implémentation des tests d'interface utilisateur UI.

L'IA a généré les Page Objects Selenium en analysant des extraits du code React/Material-UI. Ces objets, excellents comme base de départ, ont demandé des modifications, notamment pour rendre les sélecteurs XPath pertinantes face à l'asynchronisme de React et aux spécificités de Material-UI.

4.2 L'Impact du Prompt Engineering et Processus de Validation

La qualité du résultat final a été directement proportionnelle à la précision des informations fournies, validant l'importance du "prompt engineering". Les entrées fournies à l'IA incluaient systématiquement le contexte général du projet, les exigences spécifiques à tester, et des exemples concrets "format JSON, endpoints exacts, codes HTTP attendus". Cette précision a permis de générer des sorties de haute niveau pour les scénarios Gherkin et les tests d'API. Cependant, même avec des prompts optimaux, la validation humaine est restée indispensable. Notre processus de validation systématique impliquait la vérification de la compilabilité du code, l'examen de sa logique pour détecter des erreurs sémantiques, et le contrôle de sa conformité aux bonnes pratiques. Bien que l'IA ait accéléré la production initiale, cette phase de validation garantissait que les tests étaient non seulement fonctionnels mais aussi maintenables et pertinents pour notre domaine.

4.3 Limites et Rôle Crucial de l'Intervention Humaine

L'expérience a clairement mis en lumière les limites fondamentales des LLMs, justifiant le rôle central de l'ingénieur en tests.
Premièrement, l'IA, entraînée sur des données génériques, ne peut pas comprendre les subtilités du contexte métier spécifique de notre application. Elle ne génère que des solutions basées sur des patterns courants, omettant souvent les cas limites critiques. Ainsi que des problèmes techniques liés à l'exactitude des détails d'implémentation tels que versions des bibliothèques, comportement asynchrone des composants UI nécessitaient des corrections manuelles.En plus le code généré est un point de départ. La maintenance et l'évolution des tests face aux changements de l'application exigent une compréhension profonde de la logique de test, ce qui ne peut être délégué.

l'IA est un outil puissant pour accélérer la production de code, mais elle est inefficace sans le jugement critique, l'expertise contextuelle et la capacité de maintenance d'un humain.

Chapitre 5

Implémentation des tests fonctionnels backend

5.1 Scénarios de test d'authentification

Les tests d'authentification constituent le premier niveau de validation fonctionnelle de notre application. Sans authentification fonctionnelle, aucune autre fonctionnalité ne peut être testée de manière significative, ce qui en fait une priorité absolue dans notre stratégie de test.
Le scénario d'enregistrement d'un nouvel utilisateur illustre l'approche BDD. Dans le fichier `user-authentication.feature`, nous avons :

```
Scenario: S'enregistrer avec un email et mot de passe valides
  Given l'utilisateur n'existe pas avec l'email "newuser@test.com"
  When je m'enregistre avec:
    | name   | email              | password | age |
    | Julien | newuser@test.com | pwd1234 | 25 |
  Then l'enregistrement reussit et l'utilisateur existe avec l'email
     "newuser@test.com"
```

Ce scénario capture l'essence du comportement attendu en trois étapes claires. L'étape Given établit le contexte, nous nous assurons que l'utilisateur n'existe pas déjà, évitant ainsi des interférences avec des données de tests précédents. L'étape When décrit l'action, l'enregistrement d'un utilisateur avec des données complètes fournies sous forme de table Cucumber. L'étape Then vérifie le résultat, l'enregistrement doit réussir et l'utilisateur doit désormais exister dans le système.
L'implémentation de ce scénario dans UserSteps.java démontre comment les step definitions font le pont entre le langage métier Gherkin et le code technique. La méthode annotée @When("je m'enregistre avec :") reçoit un paramètre DataTable de Cucumber, qui est automatiquement parsé en une map de valeurs. Nous construisons ensuite un objet JSON correspondant à la structure attendue par notre API, avec les champs name, email, password et age. Cette structure est envoyée via une requête POST à l'endpoint **/api/users/register** en utilisant TestRestTemplate de Spring Boot.
TestRestTemplate est un outil puissant pour tester des API REST dans un contexte Spring Boot. Il encapsule les complexités de la communication HTTP tout en offrant une API simple et expressive. Dans notre implémentation, nous créons une HttpEntity contenant le corps de la requête et les headers nécessaires, puis nous utilisons restTemplate.postForEntity() pour envoyer la requête et récupérer la réponse. La vérification du résultat, dans la méthode annotée @Then, se concentre sur deux aspects, le code de statut HTTP doit être **200 (succès)**,

et l'utilisateur doit effectivement exister dans le système. Cette double vérification est importante, elle valide à la fois que l'API répond correctement et que l'effet de bord attendu "création de l'utilisateur en base de données" a bien eu lieu.

Le scénario de connexion avec des identifiants valides suit une structure similaire mais introduit un aspect supplémentaire, la récupération et le stockage de l'identifiant utilisateur pour les tests ultérieurs. Après une connexion réussie, notre step definition parse la réponse JSON pour extraire l'ID de l'utilisateur connecté, stocké dans une variable d'instance currentUserId. Cette variable sera ensuite utilisée dans les tests de produits pour identifier l'utilisateur effectuant les opérations, illustrant comment les tests peuvent maintenir un état entre les étapes.

Les cas d'échec d'authentification sont tout aussi importants à tester que les cas de succès. Bien que nous n'ayons pas détaillé tous ces scénarios dans le rapport par souci de concision, notre suite de tests inclut des scénarios pour, tentative de connexion avec un email inexistant **doit échouer avec un code 401 ou 404**, tentative de connexion avec un mauvais mot de passe **doit échouer avec un code 401**, tentative d'enregistrement avec un email déjà utilisé **doit échouer avec un code approprié et un message d'erreur explicite**. Ces tests de cas d'erreur sont cruciaux pour garantir la robustesse de l'API.

5.2 Scénarios de test des opérations CRUD sur les produits

Les tests des opérations CRUD (Create, Read, Update, Delete) sur les produits constituent le coeur de la validation fonctionnelle de notre application. Ces tests vérifient non seulement que les opérations de base fonctionnent, mais aussi que les règles métier et les contraintes sont respectées. Le scénario de listing des produits est conceptuellement simple mais important. Il vérifie qu'un utilisateur authentifié peut récupérer la liste de tous les produits disponibles. Dans notre implémentation, nous avons défini ce scénario comme suit :

```
gherkinScenario: Lister tous les produits
  When je demande la liste des produits en tant que "1"
  Then je reois au moins 1 produit
```

L'utilisation de au moins 1 produit plutôt qu'un nombre exact reflète une bonne pratique de test : nous testons le comportement fonctionnel sans créer de dépendance rigide sur l'état exact de la base de données. L'implémentation de ce scénario dans ProductSteps.java démontre un pattern important pour les tests d'API authentifiées, l'injection du header X-User-Id dans chaque requête. Ce header, bien que simpliste par rapport à des mécanismes d'authentification réels basés sur JWT, permet de simuler le contexte utilisateur. Nous créons une HttpHeaders, y ajoutons le header avec l'ID de l'utilisateur, puis créons une HttpEntity contenant ces headers. La requête GET est ensuite effectuée via restTemplate.exchange() qui offre plus de flexibilité que les méthodes spécialisées comme getForEntity().Le scénario d'ajout d'un produit introduit une complexité supplémentaire : la validation de la contrainte d'unicité de l'ID. Notre scénario Gherkin est défini ainsi :

```
Scenario: Ajouter un produit avec un ID unique
  When j'ajoute un produit:
    | id | name          | price | expirationDate |
    | 10 | Clavier AZERTY | 19.99 | 2026-12-31 |
```

```
  Then le produit avec l'id "10" existe
```

L'implémentation de ce scénario nécessite une attention particulière à la construction du corps de la requête. Nous devons parser les valeurs de la DataTable Cucumber et les convertir dans les types appropriés, String pour id et name, Double pour price en utilisant Double.parseDouble, et String pour expirationDate qui sera parsée côté serveur. Cette conversion de types illustre une responsabilité importante des step definitions, faire le pont entre la représentation textuelle des données dans les features Gherkin et les types de données utilisés par l'API.

Le test de suppression introduit un aspect intéressant, la gestion des exceptions métier. Le cahier des charges spécifie que la tentative de suppression d'un produit inexistant doit lever une exception ProductNotFoundException. Notre scénario Gherkin capture cette exigence.

```
Scenario: Supprimer un produit inexistant renvoie une erreur
  When je supprime le produit avec l'id "9999"
  Then une erreur ProductNotFoundException est leve
```

L'implémentation de ce test nécessite une gestion explicite des exceptions HTTP. Dans notre step definition, nous utilisons un bloc try-catch pour capturer les HttpClientErrorException et HttpServerErrorException que RestTemplate lance lorsqu'il reçoit un code de statut d'erreur. Nous stockons ensuite le ResponseEntity avec le code d'erreur, permettant à l'étape Then de vérifier que le code de statut indique bien une erreur "4xx ou 5xx". Cette approche est plus robuste qu'une vérification sur un code de statut spécifique, car elle reste valide même si l'implémentation backend change entre 404 (Not Found) et 400 (Bad Request).

Un aspect que nous avons particulièrement soigné est la gestion du contexte entre les étapes. Les tests de produits nécessitent un utilisateur authentifié, dont l'ID doit être fourni dans chaque requête. Plutôt que de répéter la connexion dans chaque scénario, nous utilisons le Background Gherkin pour définir un contexte commun :

```
Background:
  Given a running backend
  And un utilisateur admin existe avec l'email "alice@test.com" et le mot de passe
      "admin123"
```

Ce Background est exécuté avant chaque scénario du fichier feature, garantissant que nous avons toujours un utilisateur authentifié disponible. L'utilisation de l'utilisateur "alice" qui est créé par notre DataInitializer au démarrage de l'application garantit la répétabilité des tests.

5.3 Gestion des données de test et isolation

L'un des principaux défis du test fonctionnel est d'assurer l'isolation des scénarios. Notre stratégie pour gérer les données repose sur plusieurs piliers. Premièrement, nous utilisons une base de données MongoDB dédiée aux tests, configurée via application-test.properties. Deuxièmement, une classe DataInitializer injecte un ensemble de données de test prévisibles "utilisateurs et produits de base" au démarrage de l'application via CommandLineRunner, établissant une base stable. Troisièmement, nous avons adopté une convention de nommage simple pour éviter les conflits entre les données initiales et celles créées par les tests. Bien que fonctionnelle, cette approche pourrait être améliorée dans un environnement de production par des techniques plus avancées comme le nettoyage de la base de données entre chaque scénario via un hook Cucumber After, afin de garantir une isolation parfaite. L'utilisation de

TestRestTemplate dans cet environnement offre également l'avantage d'être automatiquement configuré pour fonctionner avec le serveur embedded de Spring, simplifiant l'écriture des tests.

Chapitre 6

Implémentation des tests fonctionnels frontend avec Selenium

6.1 Pattern Page Object Model - architecture et bénéfices

L'implémentation de tests Selenium robustes et maintenables nécessite une architecture qui sépare la logique de test de l'interaction technique avec le navigateur. Le pattern **Page Object Model (POM)** constitue la meilleure pratique reconnue. Sa compréhension et sa mise en œuvre correcte ont représenté l'un des apprentissages les plus significatifs de ce TP.

Le principe fondamental du POM est d'encapsuler toutes les interactions avec une page web dans une classe Java dédiée, appelée **Page Object**. Cette classe expose des méthodes publiques représentant les actions métier que l'on peut effectuer sur la page (login, addProduct, logout) tout en masquant les détails techniques d'implémentation "sélecteurs XPath, attentes Selenium, gestion des éléments du DOM". Cette séparation apporte plusieurs bénéfices cruciaux pour la maintenabilité à long terme.

Le premier bénéfice est la **localisation des changements**. Lorsque l'interface utilisateur évolue et elle évolue invariablement dans tout projet actif, seul le Page Object correspondant doit être modifié. Sans Page Objects, chaque changement nécessiterait potentiellement la modification de nombreuses step definitions. Avec le POM, le changement est localisé dans une seule classe, et tous les tests bénéficient automatiquement de la mise à jour.

Le deuxième bénéfice est la **réutilisabilité du code**. Une action comme "se connecter" peut être nécessaire dans de nombreux scénarios de test différents. Avec un Page Object `LoginPage` exposant une méthode `login(email, password)`, nous écrivons cette logique une seule fois et la réutilisons partout où nécessaire.

Le troisième bénéfice est l'amélioration de la **lisibilité et compréhensibilité des tests**. Comparons deux approches :

```
// Sans Page Object
driver.findElement(By.xpath("//input[@type='email']")).sendKeys("alice@test.com");
driver.findElement(By.xpath("//input[@type='password']")).sendKeys("admin123");
driver.findElement(By.xpath("//button[contains(., 'Se connecter')]")).click();
```

```
// Avec Page Object
loginPage.fillLoginForm("alice@test.com", "admin123");
loginPage.clickLogin();
```

La seconde version est incomparablement plus claire et exprime l'intention du test plutôt que les détails techniques de son implémentation.

Notre implémentation comporte deux Page Objects principaux,`LoginPage` et `ProductPage`. La classe `LoginPage` contient d'abord un ensemble de constantes privées définissant les sélecteurs des éléments de la page. Ces sélecteurs utilisent principalement XPath pour naviguer dans le DOM, garantissant une robustesse face aux changements de structure.

```
private By emailInput = By.xpath("//input[@type='email']");
private By passwordInput = By.xpath("//input[@type='password']");
private By ageInput = By.xpath("//label[contains(text(),
    'ge')]//following::input[1]");
```

La classe contient également un `WebDriver` et un `WebDriverWait` injectés via le constructeur, essentiels pour gérer l'asynchronisme du JavaScript moderne.
Les méthodes publiques du Page Object représentent les actions utilisateur, par exemple `fillLoginForm` :

```
public void fillLoginForm(String email, String password) {
    wait.until(ExpectedConditions.visibilityOfElementLocated(emailInput));
    clearAndSendKeys(driver.findElement(emailInput), email);
    clearAndSendKeys(driver.findElement(passwordInput), password);
}
```

La méthode utilitaire `clearAndSendKeys` permet de vider les champs de manière fiable avant d'envoyer les données :

```
private void clearAndSendKeys(WebElement element, String text) {
    ((JavascriptExecutor) driver).executeScript("arguments[0].value = '';",
        element);
    element.sendKeys(text);
}
```

Les méthodes de vérification comme `isUserLoggedIn()` encapsulent les assertions sur l'état de la page, gérant proprement les cas où l'élément attendu n'existe pas.

6.2 Gestion des spécificités Material-UI et React

Material-UI et React introduisent des défis spécifiques pour les tests Selenium. Les composants MUI génèrent des structures DOM complexes avec de nombreux wrappers, rendant certains sélecteurs fragiles. Nous privilégions donc les sélecteurs basés sur des attributs sémantiques ou le texte visible, plutôt que sur la hiérarchie exacte des divs.
React ajoute des rendus conditionnels et animations. Par exemple, après un clic sur "S'inscrire", le DOM est mis à jour progressivement. Nos tests utilisent `WebDriverWait` pour attendre explicitement que les éléments soient présents avant toute interaction :

```
public void switchToRegisterMode() {
```

```
    WebElement button =
        wait.until(ExpectedConditions.elementToBeClickable(switchModeButton));
    button.click();
    wait.until(ExpectedConditions.presenceOfElementLocated(idInput));
}
```

Les animations Material-UI "fade-in des alertes" nécessitent parfois des pauses explicites pour garantir la stabilité des tests.

6.3 Step definitions Selenium et intégration avec Cucumber

Les step definitions dans `UISteps.java` font le lien entre les scénarios Gherkin et les Page Objects. Elles orchestrent les actions sur les Page Objects et effectuent les assertions nécessaires. Exemple pour le formulaire de connexion :

```
@When("je remplis le formulaire de connexion avec l'email {string} et le mot de
    passe {string}")
public void fill_login_form(String email, String password) {
    loginPage.fillLoginForm(email, password);
    System.out.println("    Formulaire de connexion rempli: " + email);
}
```

Les scénarios E2E complexes utilisent plusieurs Page Objects. L'initialisation lazy garantit que chaque Page Object est créé avec le bon WebDriver et réutilisé entre les étapes :

```
@Autowired private WebDriver driver;
@Autowired private CommonSteps common;
private LoginPage loginPage;
private ProductPage productPage;

private void initPages() {
    if (loginPage == null) {
        loginPage = new LoginPage(driver);
    }
    if (productPage == null) {
        productPage = new ProductPage(driver);
    }
}
```

Notre stratégie de test combine tests backend via API REST et frontend. Cette approche respecte les bonnes pratiques industrielles, beaucoup de tests backend rapides, moins de tests frontend plus lents, assurant une couverture efficace et une validation complète de l'application.

Chapitre 7

Exécution, résultats et analyse critique

7.1 Exécution de la suite de tests

L'exécution de notre suite complète de tests s'effectue via Maven avec la commande `mvn test`. Cette commande déclenche le processus de build Maven qui compile le code de production et de test, puis exécute tous les tests via le plugin Surefire.
Cucumber commence à exécuter les fichiers feature un par un. Pour chaque scénario, Cucumber affiche le nom du scénario, puis exécute séquentiellement chaque étape (Given, When, Then). Pour les tests backend, l'exécution est relativement rapide.
Pour les tests Selenium, l'exécution est plus lente. Le lancement du navigateur Chrome prend environ 2 secondes. La navigation vers la page, le chargement des assets JavaScript et CSS, l'exécution du code React ajoutent plusieurs secondes. Chaque interaction utilisateur "clic, saisie de texte" nécessite du temps pour l'exécution du JavaScript et le re-rendering React. Un scénario Selenium complet peut prendre 10 à 20 secondes, contre 1 à 2 secondes pour un scénario backend équivalent.

La console affiche en temps réel les étapes en cours d'exécution, colorées en vert quand elles réussissent. Nos logs personnalisés, rendent la progression des tests facilement lisible. Cette attention améliore significativement le confort lors du débogage.

À la fin de l'exécution, Cucumber génère un rapport de synthèse indiquant le nombre total de scénarios exécutés, combien ont réussi, combien ont échoué, et combien ont été ignorés. Pour notre suite complète, nous obtenons :

- 14 scénarios (14 passés)

Ce résultat indique que l'ensemble de nos scénarios passe avec succès, validant que l'application répond aux exigences fonctionnelles testées.

7.2 Analyse des rapports Cucumber

Les rapports générés par Cucumber fournissent une vue détaillée des résultats de test sous plusieurs formats complémentaires.

7.2.1 Rapport HTML

Le rapport HTML, généré dans `target/cucumber-reports.html`, présente les résultats de manière visuellement attractive et interactive.

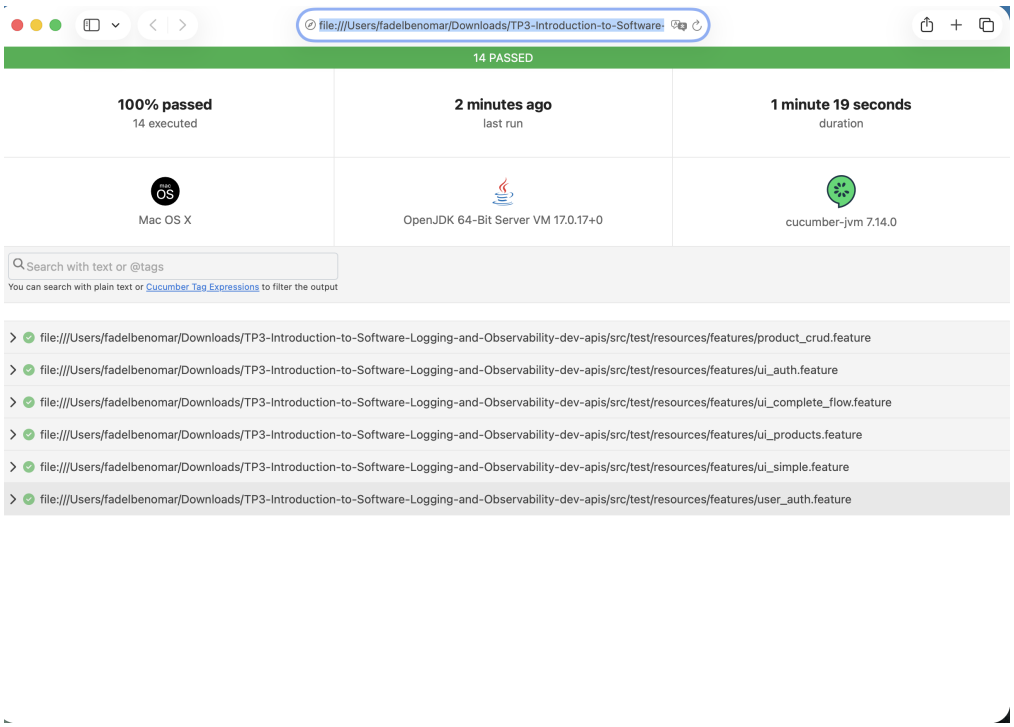


FIGURE 7.1 – Rapport HTML Cucumber montrant la liste des features exécutées

Le rapport HTML liste toutes les features testées (`user-authentication.feature`, `products-crud.feat` `ui-authentication.feature`, etc.) avec pour chacune le nombre de scénarios passés/échoués. Les features avec 100% de succès sont marquées en vert. En cliquant sur une feature particulière, on accède au détail de chaque scénario avec ses étapes colorées selon le résultat : vert pour succès, rouge pour échec, jaune pour skipped. Le temps d'exécution de chaque étape est également affiché.

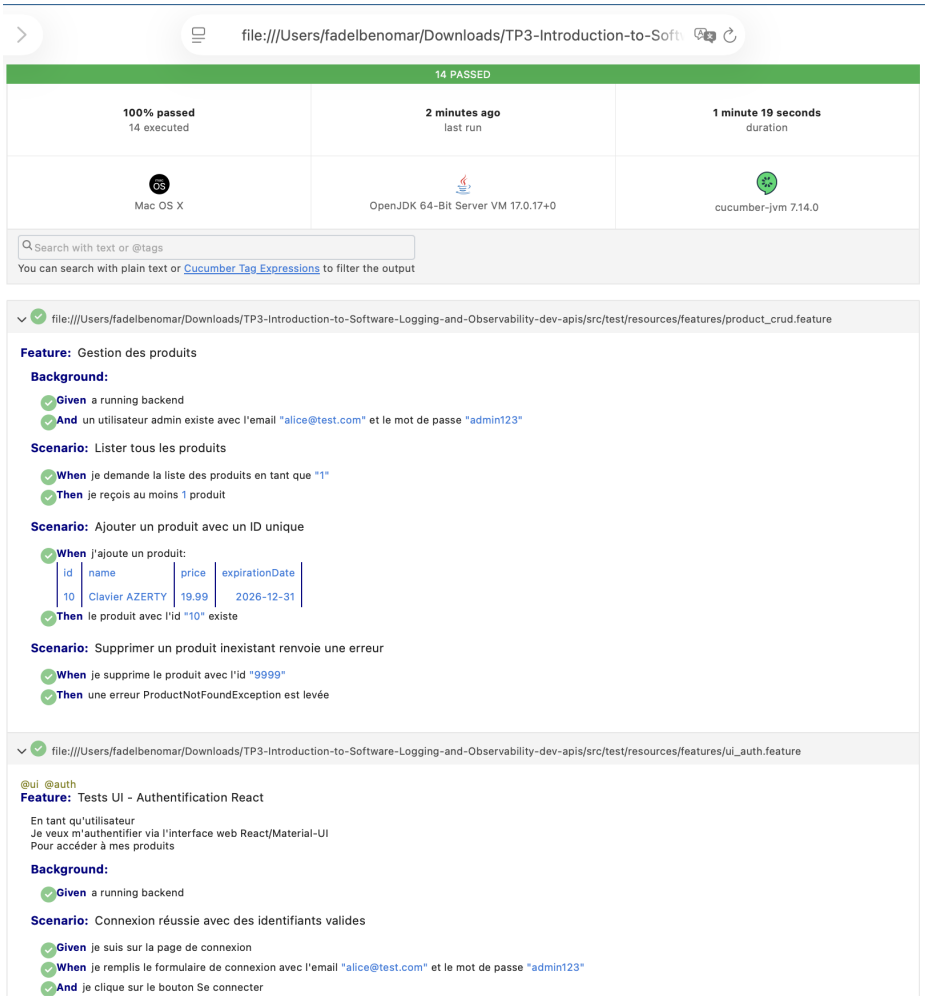


FIGURE 7.2 – Détail d'un scénario dans le rapport Cucumber

7.2.2 Rapport JSON

Le rapport JSON généré (`target/cucumber.json`) contient les mêmes informations dans un format machine-readable. Ce fichier peut être consommé par des outils de CI/CD comme Jenkins, GitLab CI, ou GitHub Actions pour générer des graphiques d'évolution de la couverture de tests ou des alertes en cas de régression.

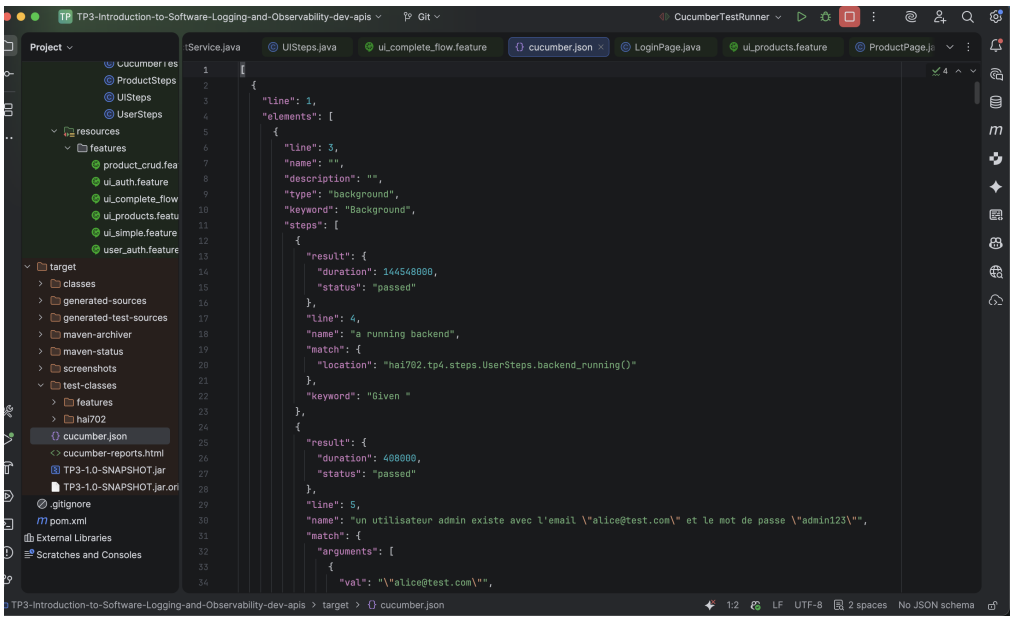


FIGURE 7.3 – cucumber.json

7.3 Screenshots et preuve de fonctionnement

La capture d'écran suivante démontre le fonctionnement effectif de notre application et de nos tests.

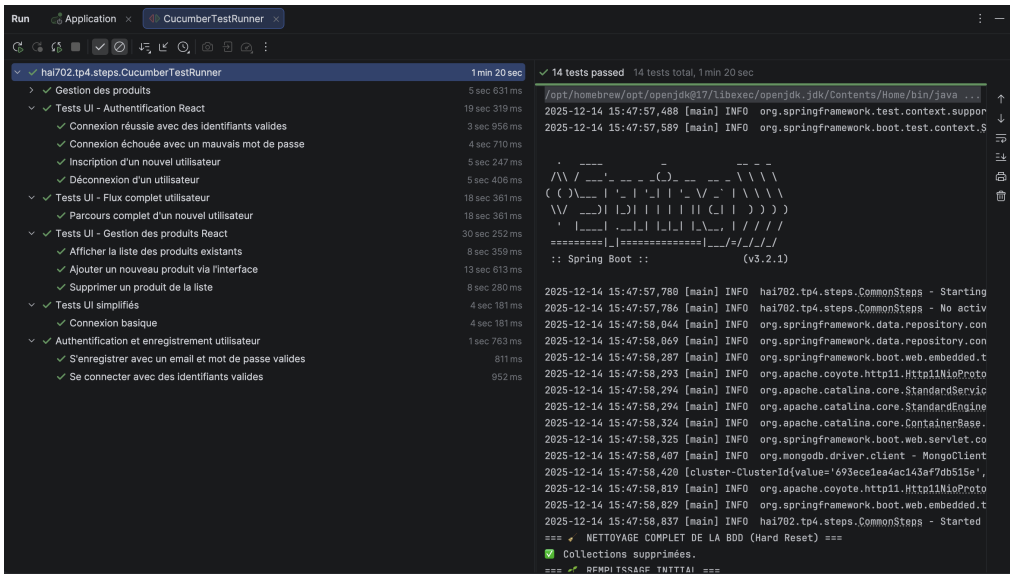


FIGURE 7.4 – Terminal montrant l'exécution des tests Selenium

7.4 Analyse de l'apport de l'IA dans le processus

L'analyse quantitative et qualitative de l'utilisation de l'IA dans ce TP révèle des résultats nuancés.

7.4.1 Apport quantitatif

Nous estimons que l'assistance IA nous a permis de gagner environ 40 à 50% du temps nécessaire pour développer la suite de tests. Les gains les plus significatifs se sont manifestés dans la génération des scénarios Gherkin et des squelettes de step definitions.
Pour les step definitions backend, environ 70% du code final était directement utilisable. Les 30% restants nécessitaient des ajustements spécifiques à notre architecture endpoints, format JSON, gestion d'erreurs. Pour les Page Objects Selenium, 50% du code généré était directement utilisable, 50% nécessitant des ajustements.

7.4.2 Apport qualitatif

Le plus significatif n'a pas été la génération de code, mais l'apprentissage accéléré et la découverte de bonnes pratiques :

- Utilisation de JavaScriptExecutor pour vider les champs de formulaire
- Patterns d'attente explicite avec `ExpectedConditions`
- Structure optimale d'un Page Object avec séparation claire des sélecteurs et des méthodes
- Gestion des exceptions dans les tests REST

7.4.3 Limites identifiées

L'IA ne pouvait pas :

- Comprendre les règles métier spécifiques pour proposer des cas de test pertinents
- Identifier les cas limites critiques nécessitant une expertise métier
- Déboguer les échecs résultant d'interactions complexes React/Selenium/Material-UI
- Optimiser les temps d'exécution en détectant les attentes ou opérations redondantes

7.4.4 Observation sur l'amélioration progressive

La qualité de l'assistance IA s'est améliorée au fil du TP à mesure que nous affinions nos prompts. Les premières requêtes généraient des résultats génériques, tandis que les prompts structurés et précis produisaient un code de qualité nettement supérieure.

Conclusion

Ce travail pratique a permis de valider une approche d'automatisation des tests fonctionnels assistée par l'Intelligence Artificielle, tout en renforçant notre maîtrise de la chaîne de test moderne. Nous avons implémenté avec succès une suite de tests exhaustive en suivant la méthodologie Behavior Driven Development, utilisant Cucumber pour structurer les scénarios métier et Selenium WebDriver pour valider les flux de bout en bout sur une application Spring Boot, React, MongoDB. Ce processus nous a confrontés aux défis réels de l'ingénierie logicielle, notamment la complexité d'intégration, la nécessité de la configuration "WebDriverManager, SpringTest" et l'importance d'appliquer le Page Object Model pour garantir la maintenabilité des tests d'interface utilisateur. Notre principale conclusion technique est que les tests bien écrits sont un investissement crucial qui assure la stabilité et la confiance dans les évolutions futures du code.

L'intégration de l'IA a confirmé son rôle d'accélérateur de productivité pour les tâches répétitives. Nous avons mesuré un gain de temps significatif, principalement dans la génération de code boilerplate "squelette des step definitions et des Page Objects" et dans la traduction initiale des spécifications en scénarios Gherkin. Cependant, notre expérience a clairement défini les limites,l'IA est un outil d'assistance, non un substitut. L'expertise humaine est restée indispensable pour l'analyse métier, la définition des cas limites critiques, le débogage des problèmes de synchronisation et la validation finale de la pertinence des tests. Nous avons adopté une méthodologie rigoureuse où chaque suggestion de l'IA était soumise à une révision critique, assurant ainsi la qualité finale et évitant le risque de dépendance ou d'erreurs sémantiques.

En définitive, ce TP est une étape décisive vers la maîtrise des compétences requises pour le développement logiciel moderne. Nous avons acquis une vision sur les tests fonctionnels sont une forme de documentation exécutable qui préserve la connaissance métier. La méthodologie que nous avons établie combinant la vitesse de génération de l'IA avec le jugement critique et l'expertise humaine est directement transférable à un environnement professionnel. Ce travail nous positionne pour tirer pleinement parti des outils d'Intelligence Artificielle, non pas en déléguant notre responsabilité, mais en optimisant notre temps pour nous concentrer sur les tâches de haute valeur ajoutée.