

# Test & Sécurité

## Travaux Pratiques



Karim EL GHARBI

Florian REMY

DII5A

Promotion 2016 -2019

Test & Sécurité – Année 5

Décembre 2018

# TABLE DES MATIERES

Introduction.....	3
I. Présentation du projet .....	4
1. Rappel : Cahier des charges .....	4
2. Présentation des solutions .....	5
II. Aspect « Back-End ».....	6
1. Création d'une API .....	6
2. RUST .....	7
Installations.....	7
Développement .....	8
3. Tests & Validation .....	12
III. Aspect « Front-End ».....	13
1. JavaFx : Mise en œuvre .....	13
2. Développement .....	14
Structure de l'application .....	14
Programmation des interactions.....	16
3. Tests unitaires.....	19
4. Contrôles de qualité.....	21
SonarLint.....	21
SonarQube .....	22
IV. Cahier de recettes.....	26
V. Gestion de projet.....	28
1. GIT .....	28
2. Méthode agile.....	28
Conclusion.....	29

# Introduction

Le présent document est un rapport qui résume les réalisations dans le cadre de la mise en œuvre du projet nommé « RoverLand ».

Le but de ce projet était de concevoir et de mettre en place une application proposant des annonces de véhicules de marque Rover.

Ce projet est basé sur un ensemble de technologies, certaines maîtrisées, d'autres totalement nouvelles.

Dans le cadre de ce travail, les problématiques étaient du type tests et sécurité. Pour répondre à ces problématiques, il a été choisi de décomposer le projet avec une partie front et une partie back de manière à pouvoir traiter l'aspect sécurité côté serveur et l'aspect tests du côté interface graphique. Ce sont ces deux aspects qui sont traités dans la suite de ce rapport.



*Figure 1 : Logo Rover*

# I. Présentation du projet

## 1. Rappel : Cahier des charges

L'application à développer devra comporter une interface listant les annonces de véhicules disponibles. La particularité de cette application de petites annonces est qu'elle offre la possibilité d'acheter immédiatement un ou plusieurs véhicules au moyen d'un panier. La nouveauté est donc ce croisement entre site de petites annonces et site de e-commerce.

L'utilisateur pourra donc sélectionner un ou plusieurs véhicules pour les ajouter à son panier. Une fois que l'utilisateur aura fini ses achats, il devra avoir la possibilité d'afficher son panier. Ainsi, l'utilisateur pourra régler la quantité et supprimer un article du panier (véhicule). La partie paiement ne sera pas gérée dans le cadre de ce projet.

Le développement sera réalisé en deux blocs. Une partie « back-end » s'occupera de tout l'aspect traitement et gestion des données. Ces données seront ensuite mises à disposition du second bloc « front-end » via une API WEB (API REST). Ainsi, via un certain nombre d'URL mises à disposition par le serveur WEB, le client aura accès à toutes les données qu'il souhaite afficher (liste des annonces, contenu du panier...). Le client pourra de même effectuer des requêtes via cette interface (API).



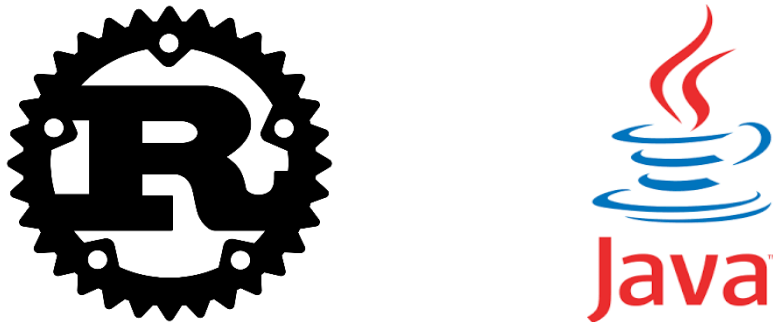
Figure 2: Front-End / Back-End

Remarque : Depuis la seconde livraison du cahier de spécifications, il a été décidé de supprimer la fonctionnalité permettant de gérer les quantités pour des critères de complexité.

## 2. Présentation des solutions

Il a été choisi de réaliser l'application suivant deux technologies différentes. Il a été décidé de programmer la partie « back » en RUST et la partie « front » en Java. En effet, le langage RUST étant totalement inconnu, il était préférable de choisir un langage connu (ex : Java) et d'utiliser de nouveaux outils compatibles avec ce langage. Il s'agira donc de créer une « application bureau » qui viendra interagir avec un serveur WEB.

À partir de ces outils, l'objectif est donc de mettre en œuvre certain aspects du CRUD (create, read, update, delete). Il s'agit des quatre opérations de base pour la persistance des données. Ce projet devra mettre en œuvre à minima le create et le delete.



*Figure 3: RUST / Java*

Remarque : En ce qui concerne le CRUD, il a été décidé de mettre en place le read comme fonctionnalité supplémentaire.

## II. Aspect « Back-End »

### 1. Création d'une API

La première étape a été de définir toute l'interface de communication entre les parties front et back. Cette interface devait être à la fois facile d'utilisation et la plus complète possible. Un aspect sécurité et robustesse était également intéressant. C'est pourquoi nous avons utilisé l'outil Swagger, nous permettant de construire une documentation claire grâce à un fichier YAML. C'est dans ce fichier que nous avons défini les interactions entre nos deux systèmes. Une première partie montre toutes les URLs proposées ainsi que leur construction.

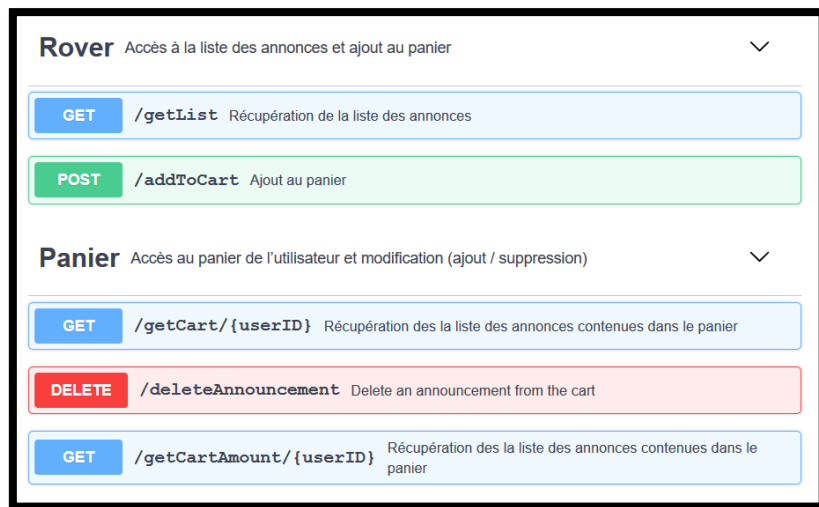


Figure 4: URLs API

Il est également possible de définir les messages de retour des différentes requêtes. Ces messages peuvent être utilisés par le front pour augmenter la robustesse de communication tout en informant l'utilisateur de la réussite, ou non, de ses actions.

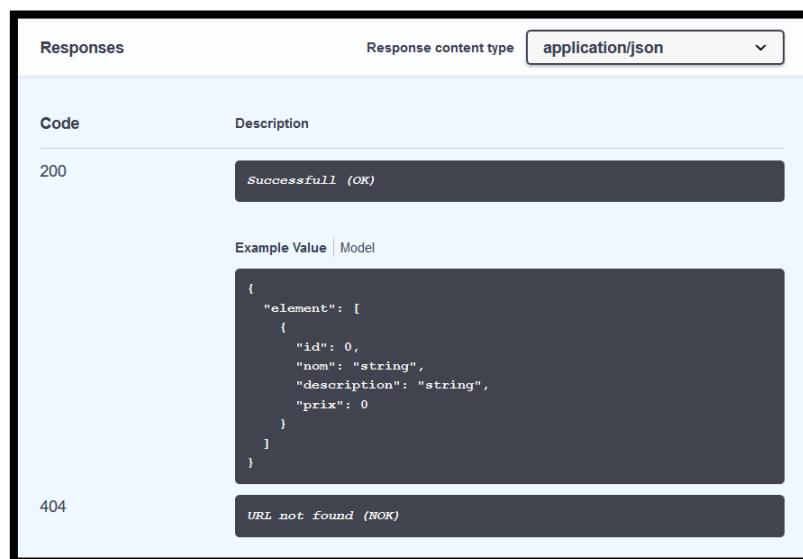


Figure 5: Codes retour de l'URL getList

De plus, il est possible de préciser toutes les structures de données JSON nécessaire au fonctionnement du système. Il s'agit des paramètres à passer à une requête HTTP.

The screenshot shows the 'Parameters' section of a Swagger UI. At the top right is a 'Try it out' button. Below it is a table with two columns: 'Name' and 'Description'. A parameter named 'body' is listed, marked as 'required' in red. Its description is 'encapsulation of articleID to delete and userID where to delete'. Below the description, there are tabs for 'Example Value' and 'Model'. The 'Example Value' tab is active, showing a JSON object: 

```
{  "id_article": 0,  "id_user": 0}
```

. At the bottom, there is a 'Parameter content type' dropdown menu with 'application/json' selected.

Figure 6: Paramètres de l'URL deleteAnnouncement

Enfin, il est important de préciser que Swagger est facile à prendre en main et permet de se mettre d'accord sur les URL de façon à travailler séparément. Cet outil a fortement été utile pour garder le groupe synchronisé.

## 2. RUST

### Installations

Il faut dans un premier temps installer RUST. Pour cela, il est possible de se rendre sur le site <https://www.rust-lang.org/> et de suivre les instructions après avoir cliqué sur « getting started ». Une fois l'installation terminée, vous devrez, si cela n'a pas déjà été fait, cloner le répertoire du projet situé à l'adresse <https://github.com/KEG01/RoverLand>.

- Aller sur <https://www.rust-lang.org/>, cliquer sur getting started et se laisser guider.
- Cloner le projet.
- Ouvrir un terminal.
- Aller dans le répertoire RoverLand/back/api\_soft/.
- Saisir les commandes suivantes :

```
rustup override set nightly
rustup update && cargo update
cargo run
```

Un écran de ce type devrait apparaître, indiquant que le serveur est lancé, prêt à répondre aux requêtes.

```
Finished dev [unoptimized + debuginfo] target(s) in 0.32s
Running `target/debug/api_soft`
🔧 Configured for development.
=> address: localhost
=> port: 8000
=> log: normal
=> workers: 8
=> secret key: generated
=> limits: forms = 32KiB
=> keep-alive: 5s
=> tls: disabled
🚀 Mounting /:
=> GET /getList (get_list)
=> GET /getCart/<id> (get_cart)
=> GET /getCartAmount/<id> (get_cart_amount)
=> POST /addToCart application/json (add_to_cart)
=> DELETE /deleteAnnouncement application/json (delete_announcement)
🚀 Rocket has launched from http://localhost:8000
```

Figure 7: Lancement du serveur RUST

## Développement

Pour la partie développement, tout le code RUST a été basé sur le crate (package en rust) Rocket, qui est spécialisé dans la gestion haut niveau de connexions HTTP.



Figure 8: Crates.io

L'architecture est basée sur une série de routes qu'il est possible de monter. Il faut spécifier le type de requêtes (dans notre cas GET, POST et DELETE) ainsi que les paramètres possibles.

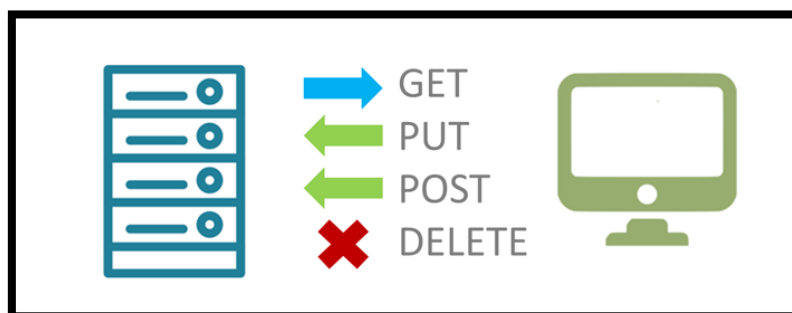


Figure 9: Requêtes HTTP



Un serialiseur/deserialiseur JSON faisant partie de Rocket, a été utilisé pour lire et créer les données au format JSON portant les informations de communication. L'enregistrement des informations en local sur les articles disponibles est réalisé via deux fichiers « txt ». Le premier, list.txt, liste toutes les annonces disponibles pour le système. Le second, cart.txt, est le fichier contenant sous forme textuelle, tous les paniers de tous les utilisateurs. Ce fichier est lu et écrit pour l'ajout ou la suppression d'un élément dans le panier d'un utilisateur. **Nous avons à ce titre, eu des problèmes du fait des caractères de fin de ligne. En effet, sous Windows, les caractères de fin de ligne sont CRLF (Carriage Return et Line Feed) alors que sous tous les systèmes Unix, le caractère est juste un LF. L'application gère maintenant cette différence et est donc pleinement portable d'un système à un autre.**

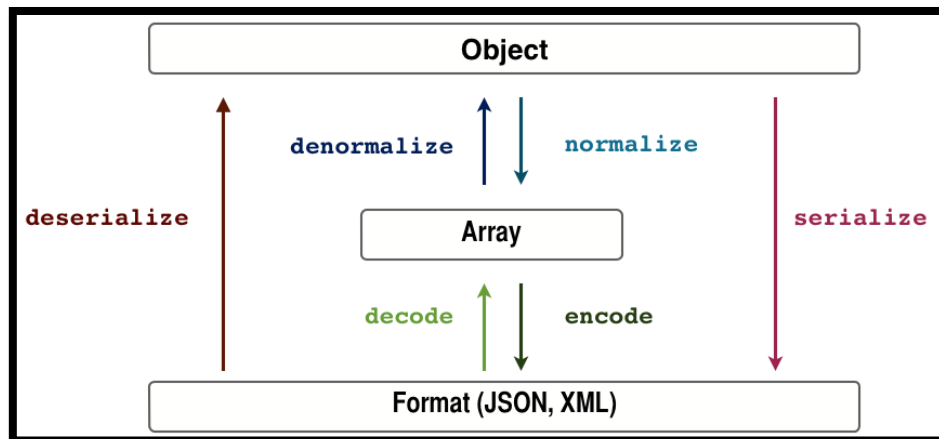


Figure 10: Serializer / Deserializer

Le développement a été réalisé en plusieurs couches. On a d'abord les couches « bas niveau », qui vont s'occuper d'écrire, de lire et de parser les fichiers. Ensuite, les résultats sont passés à des fonctions de plus haut niveau, chargées d'extraire l'information utile des résultats. Enfin, un troisième niveau s'occupe de construire la réponse à la requête envoyée.

Tout le fonctionnement est basé sur les événements d'arrivée d'une nouvelle requête. Le main de l'application ne s'occupe alors que de monter et de lancer les différentes routes proposées par le serveur.

Les structures spécifiées dans le Swagger sont retrouvables sous le même nom ici aussi. Elles implémentent le trait Serialize et/ou Deserialize, qui permet de parser ou de créer des Json à partir de ces structures de manière transparente.

Le gros point fort de Rocket, outre la gestion simplifiée des connexions HTTP, est la sécurité. En effet, nativement, ce crate peut gérer les aspects de sécurité important en vérifiant que les URL entrées sont correctes et valides par rapport à des routes proposées par le serveur par exemple. C'est ce qui a été mis en place.

De plus, il est techniquement possible de gérer les connexions HTTPS, car Rocket peut supporter le protocole TLS. Cependant, du fait de la jeunesse de RUST et de son écosystème, le support de TLS n'est encore pas considéré comme complet et stable. Il est donc déconseillé de le mettre en place pour les applications autres que du test, c'est pourquoi nous ne l'avons pas utilisé dans ce projet.

Voici ci-dessous le fonctionnement interne qui a été programmé pour l'URL addToCart. Il s'agit de la procédure permettant d'ajouter un article au panier. L'API reçoit l'identifiant de l'utilisateur et celui de l'article au format JSON. C'est ce qui explique la recherche de l'article par l'identifiant et l'ajout au panier appartenant au client.

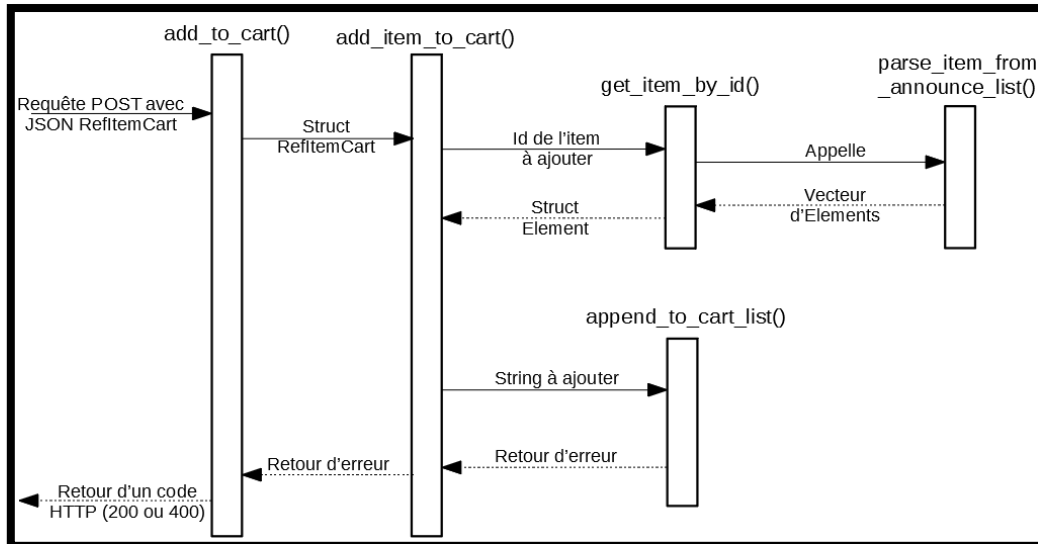


Figure 11: addToCart

Voici maintenant la procédure permettant de supprimer un article du panier qui a été mise en place (deleteAnnouncement). Cette fois également, on retrouve la recherche de l'utilisateur et de l'article avant la suppression.

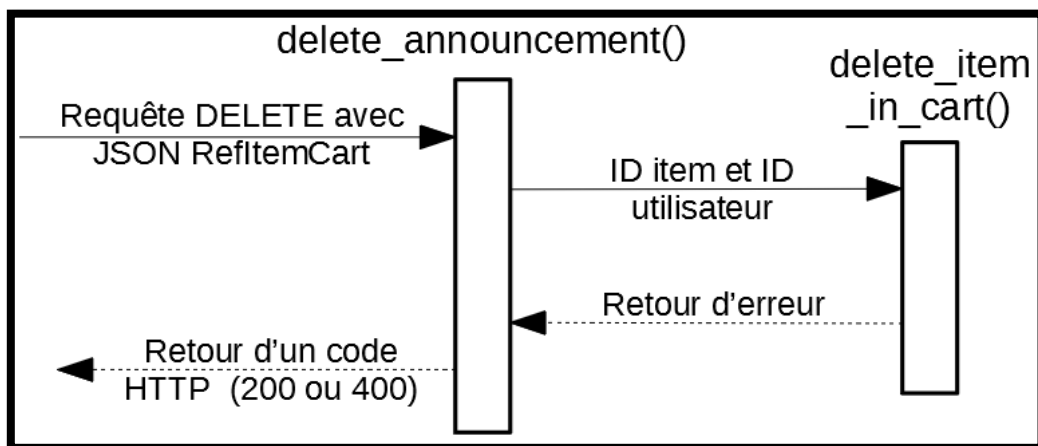


Figure 12: deleteAnnouncement

Ensuite, l'URL `getCart` a également été mise en place de façon à récupérer la liste des articles ajoutés au panier pour chaque client. C'est justement pour afficher le bon panier que le paramètre `id_user` est ajouté à l'URL. L'API renvoi la liste au format JSON.

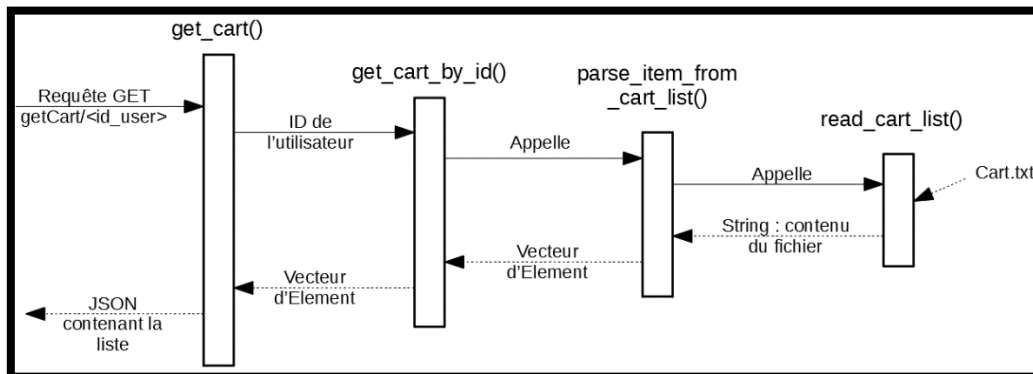


Figure 13: `getCart`

Lorsque le client effectue une requête sur l'URL `getCartAmount`, le programme appelle la fonction permettant de récupérer le total associé au panier, d'où la référence au panier utilisateur qui est récupérée.

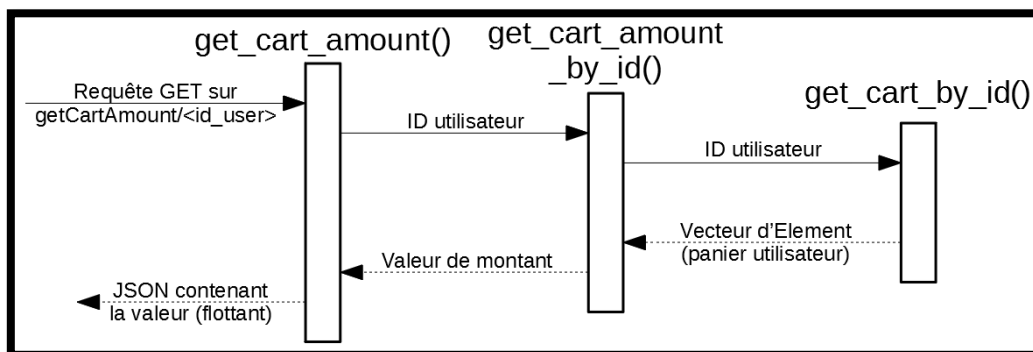


Figure 13: `getCartAmount`

Enfin, la dernière requête mise en œuvre est l'URL qui doit normalement être utilisée au début. Il s'agit de l'URL `getList`. Celle-ci consiste à appeler une fonction de parsing (comme pour les autres URL) mais cette fois-ci pour récupérer la liste de tous les véhicules à vendre.

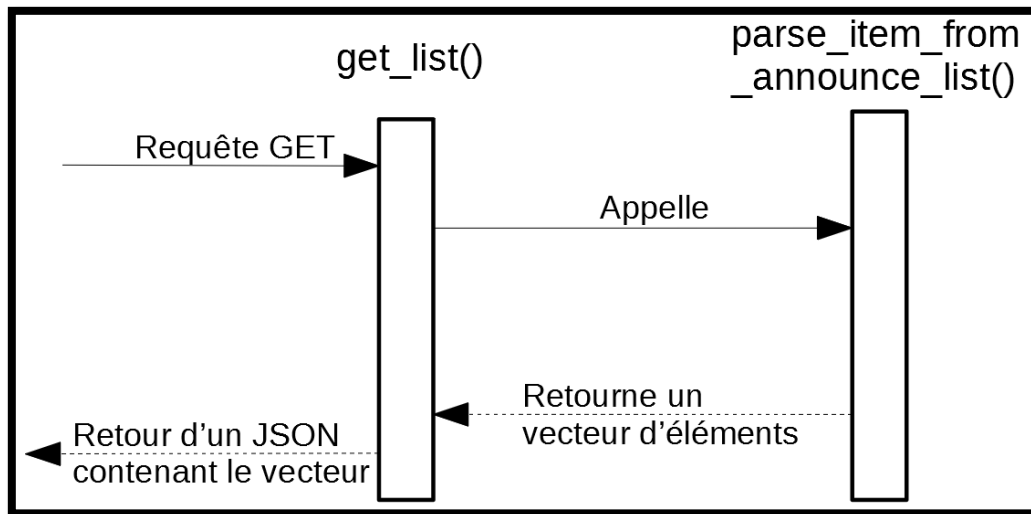


Figure 14: `getList`

### 3. Tests & Validation

La validation du back-end s'est fait au travers de tests fonctionnels arbitrairement choisis, tout en essayant de couvrir un maximum de cas. Les requêtes GET ont été lancées depuis un navigateur (Firefox) alors que les requêtes POST et DELETE l'ont été via l'outil CURL, spécialisé dans ce domaine.



Figure 15: Curl

Toutes les routes disponibles ont été testées selon plusieurs configurations (plusieurs fois la même requête ou une requête équivalente par exemple). Cela a permis dès le début du développement de remonter plusieurs bug ou incohérences de fonctionnement dans le serveur. Cela a également permis de réaliser du développement en continu, car les fonctionnalités élémentaires ont pu être testées rapidement, de manière à pouvoir les réutiliser et développer les parties de plus haut niveau.

### III. Aspect « Front-End »

#### 1. JavaFx : Mise en œuvre

Pour mettre en œuvre l'aspect front, il a été utilisé l'API officielle de Java permettant de faire des interfaces graphiques. Il s'agit de JavaFx. JavaFx appartient aujourd'hui à Oracle. Cette API Java est considérée comme l'héritière de son prédécesseur Swing. Elle est facilement intégrable à l'environnement de développement Eclipse. De plus, les interfaces graphiques sont assez faciles à construire à l'aide d'un outil nommé SceneBuilder.



Figure 16: Plugin E(fx)clipse + Scenebuilder

Ce choix a été effectué de par le rendu des interfaces graphiques, mais également pour pouvoir mettre en œuvre des outils de tests sur le langage Java en lui-même.

En ce qui concerne le fonctionnement de JavaFx. Ce dernier prend en charge des feuilles CSS et XML. En effet, pour créer une interface graphique, il est possible d'utiliser des fichiers XML qui seront édités par le logiciel SceneBuilder en fonction de nos modifications effectuées en drag and drop.

Voici ci-dessous une illustration qui schématise la structure qui a été mise en œuvre du côté front. Celle-ci est basée sur le célèbre modèle MVC (Modèle-Vue-Contrôleur).

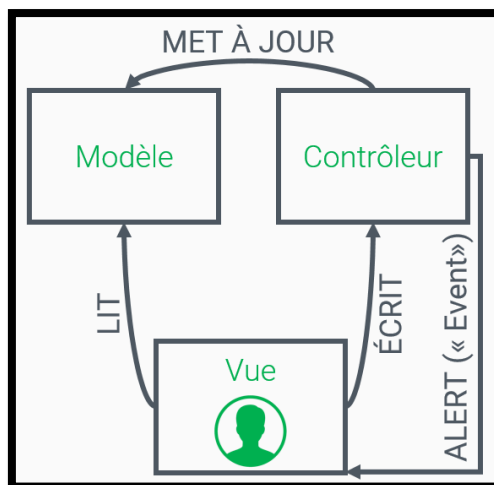


Figure 17: Modèle MVC

MVC est un modèle d'architecture logicielle destiné aux interfaces graphiques. Ce modèle est lui-même composé de trois modules : le modèle, la vue et le contrôleur. La vue contient l'interface graphique, le contrôleur contient la logique de l'application (ex : action suite à l'appui sur un bouton) et le modèle contient les données ou une structure de données à afficher. L'avantage de ce modèle est d'être modulaire et par conséquent plus facile à tester au fil de l'eau.

Pour préciser, le fonctionnement est le suivant. Lorsqu'un utilisateur vient cliquer sur un bouton « ajouter » ou « supprimer » sur l'interface graphique, une action est effectuée au niveau du contrôleur pour venir interagir avec la donnée, c'est-à-dire, la modifier. Ainsi, ces données (ex : Article, Contenu du panier...) sont ensuite remontées à l'utilisateur qui vient les lire à son tour pour mettre à jour la vue.

## 2. Développement

### Structure de l'application

Pour mettre en œuvre cette application, il a été nécessaire d'effectuer une petite analyse pour savoir comment structurer l'application de manière à ce qu'elle soit le plus modulaire possible. On rappelle que l'application doit être fonctionnelle sans API avant d'effectuer l'interfaçage. Par conséquent, toutes les classes doivent pouvoir être testées unitairement.

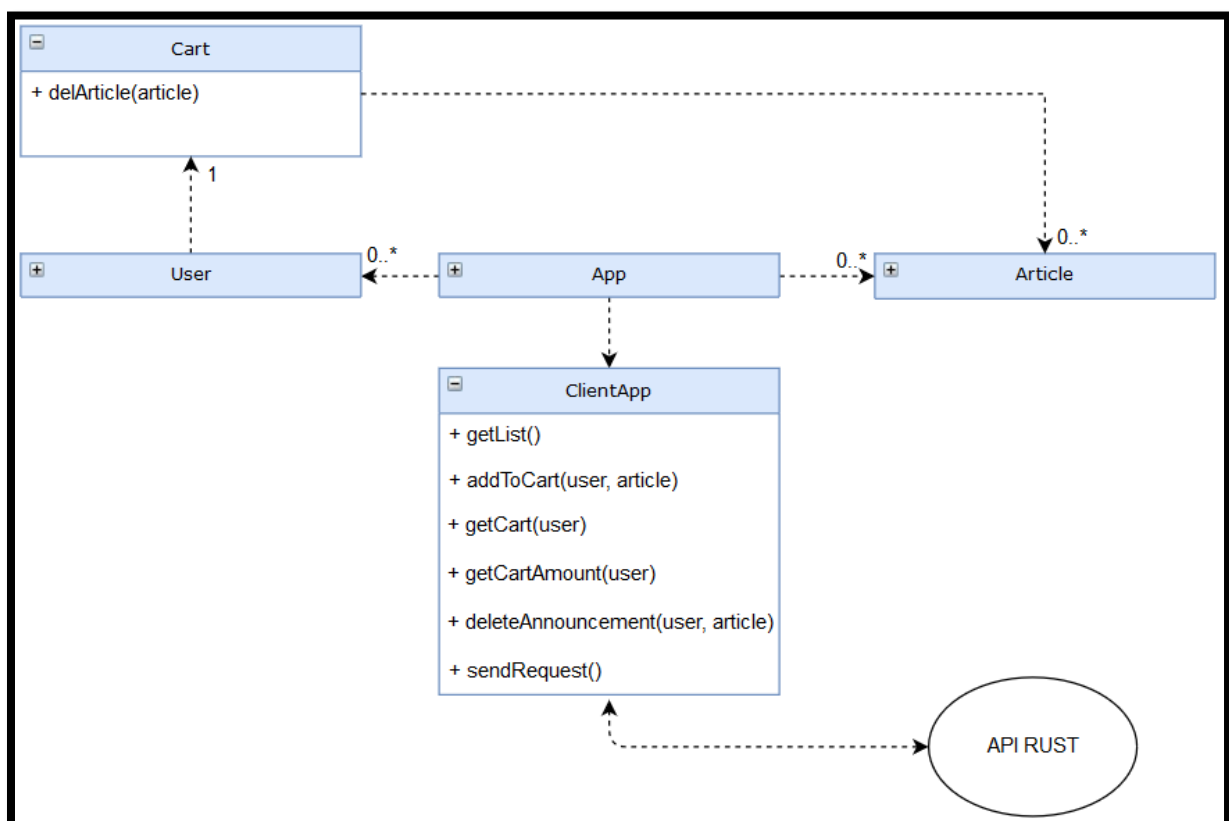


Figure 18: Diagramme des classes

Pour que cela soit possible, il a été décidé de construire l'application selon le diagramme des classes ci-dessus. En résumé, l'idée est d'interagir avec l'API RUST avec une classe qui fait l'interface entre le front et le back. C'est dans cette classe que toutes les requêtes http ont été programmées.

Outre cet aspect, la partie front est basée sur une classe application. Cette application peut posséder plusieurs utilisateurs qui eux même possèdent un panier plein d'articles ou vide.

Voici ci-dessous l'arborescence du projet (front uniquement) avec la volonté de suivre le modèle MVN en gardant l'IHM indépendante du reste. Lors du développement de cette application, il a été nécessaire de mettre en place des tests unitaires pour valider les fonctions en cours de travail.

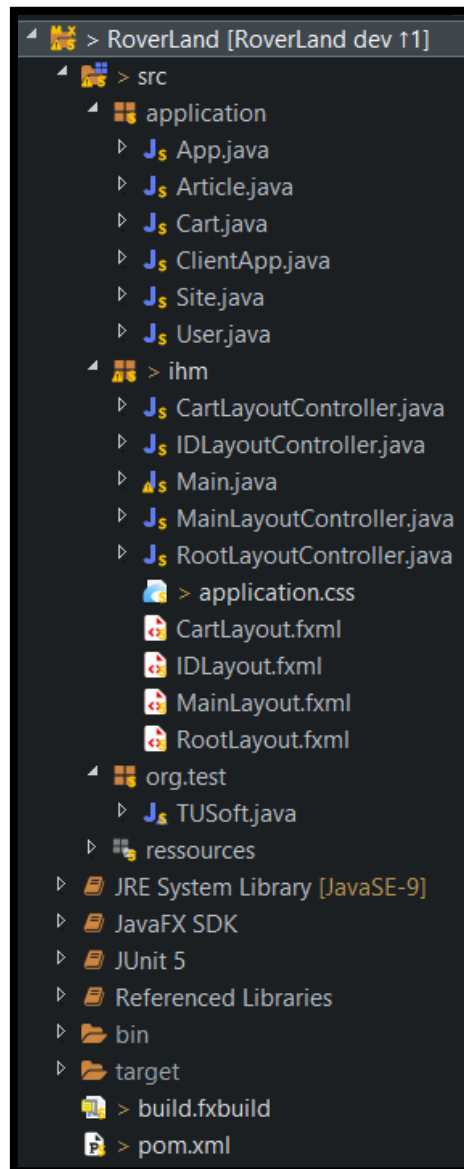


Figure 19: Arborescence des fichiers

Une fois cela effectué, il ne manque plus que les interactions à programmer de manière à interfacier les commandes avec les fonctions permettant d'ajouter des articles au panier et de les supprimer (conforme au CRUD).

## Programmation des interactions

Voici la première vue de l'application qui a été construite. Il a été fait en sorte que l'on ne puisse pas laisser de champ manquant et que l'utilisateur soit obligé de saisir un identifiant numérique. Toute la logique de cette vue est gérée dans la classe nommée IDLayoutController. Ainsi, lors de l'appui sur le bouton valider de l'utilisateur, le programme va enregistrer l'identifiant et créer un panier au client.

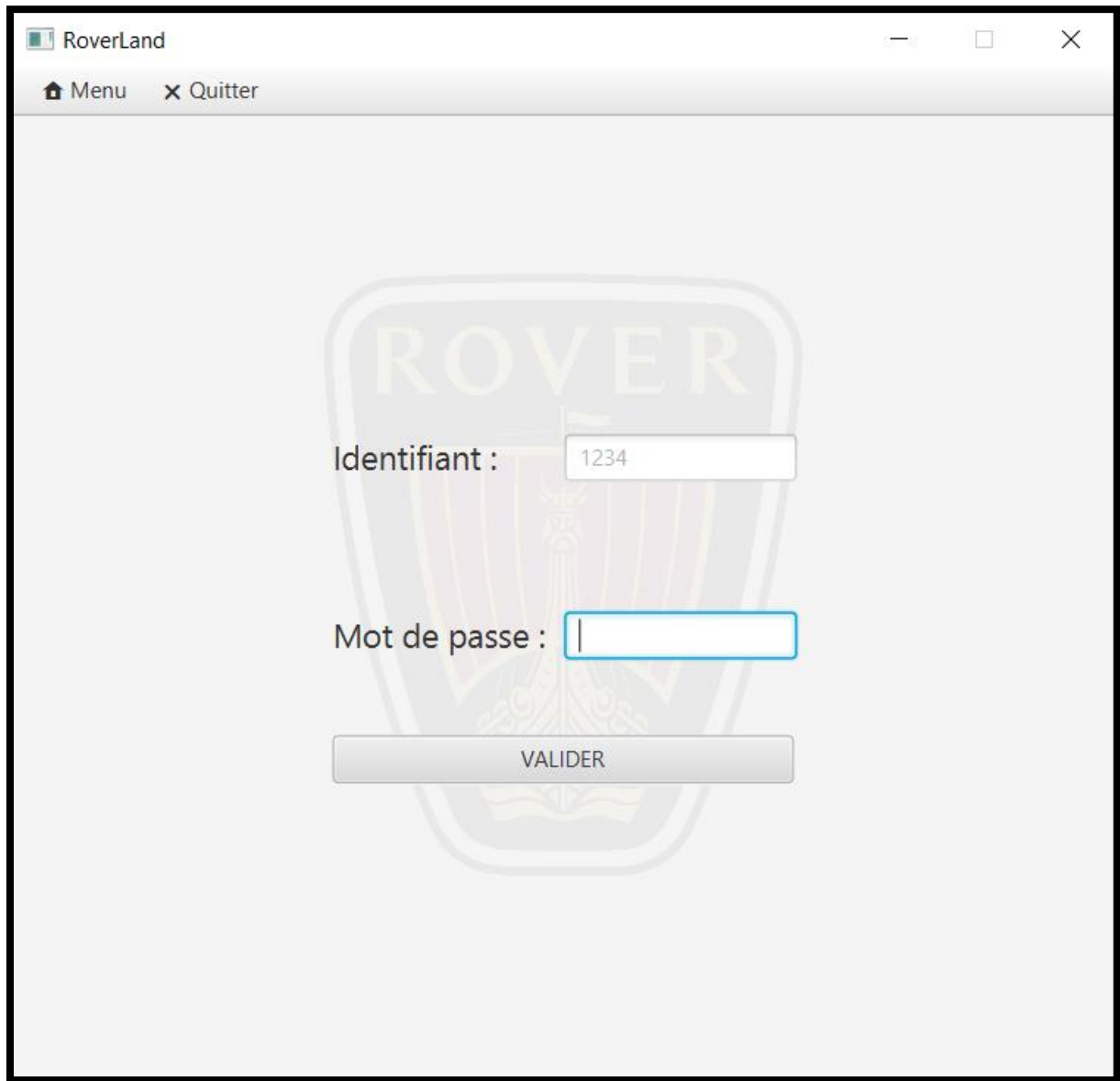


Figure 20: Vue « Login »

*Remarque : C'est dès lors que l'utilisateur est autorisé à accéder à l'application que le programme effectue sa première requête vers l'API. Cela est utile de manière à initialiser la liste des annonces en mémoire.*



La deuxième vue correspond à la vue principale. Il s'agit de celle qui présente toutes les annonces disponibles. Le contenu du tableau et la gestion des actions utilisateurs sont gérés dans la classe MainLayoutController. De plus, des listeners ont été implémentés pour détecter l'appui sur le bouton ajouter un article et sur bouton afficher le panier. Ainsi, dès lors de l'utilisateur interagira avec l'interface via ces boutons, l'application réalisera l'action attendue. En l'occurrence, il s'agit soit d'ajouter l'article au panier (requête addToCart), soit d'afficher la vue « Panier » (requêtes getCart et getCartAmount).

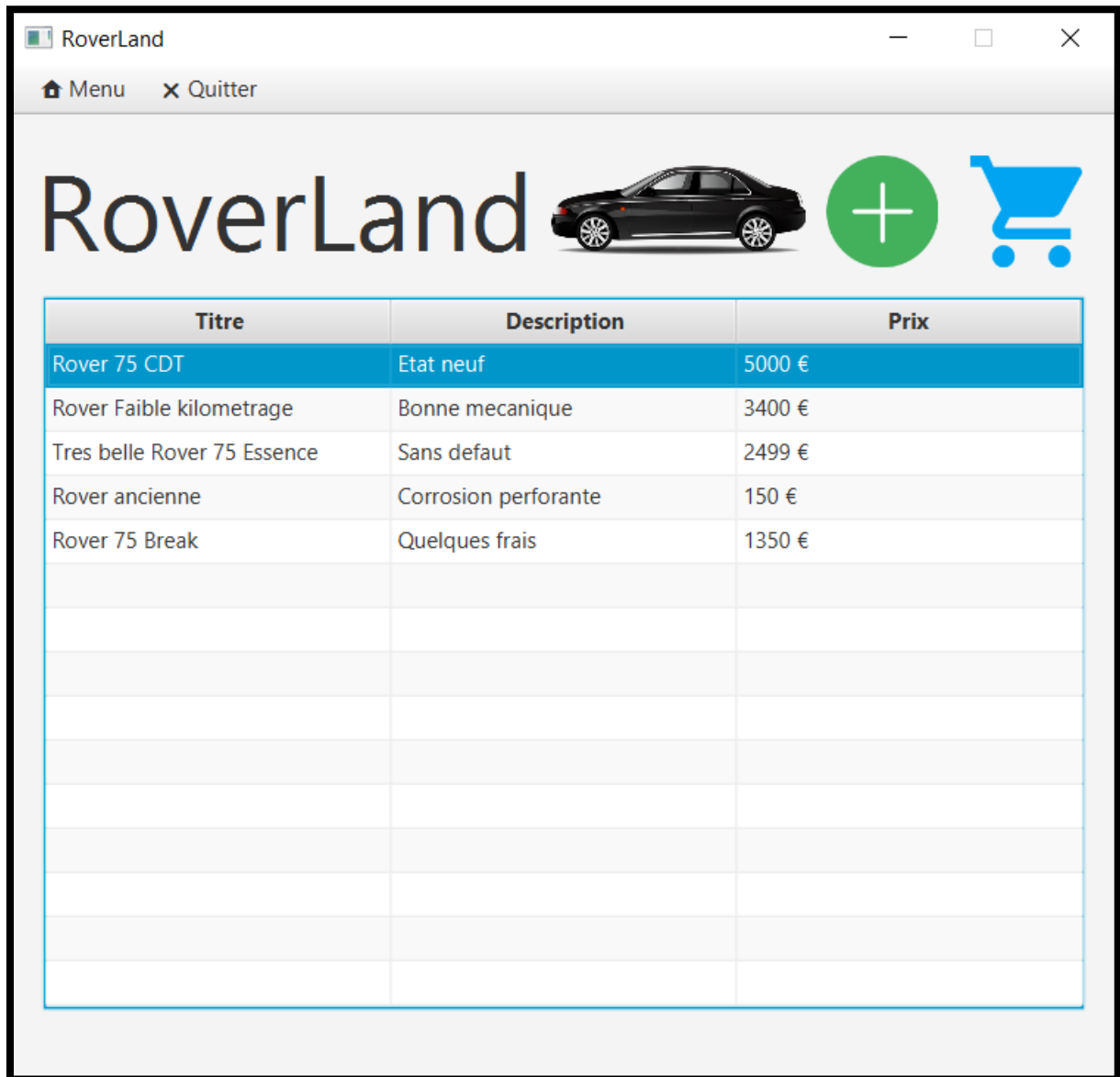


Figure 21: Vue « Annonces »

Voici enfin la dernière page qui correspond à la vue « Panier ». Cette page regroupe tous les articles précédemment sélectionnés. La logique de cette interface a été gérée dans la classe CartLayoutController. Ainsi, dès lors que l'utilisateur sera amené à supprimer un article de son panier (requête deleteAmount), l'application effectuera les requêtes getCart et getCartAmount pour mettre à jour le contenu du panier et son coût total.

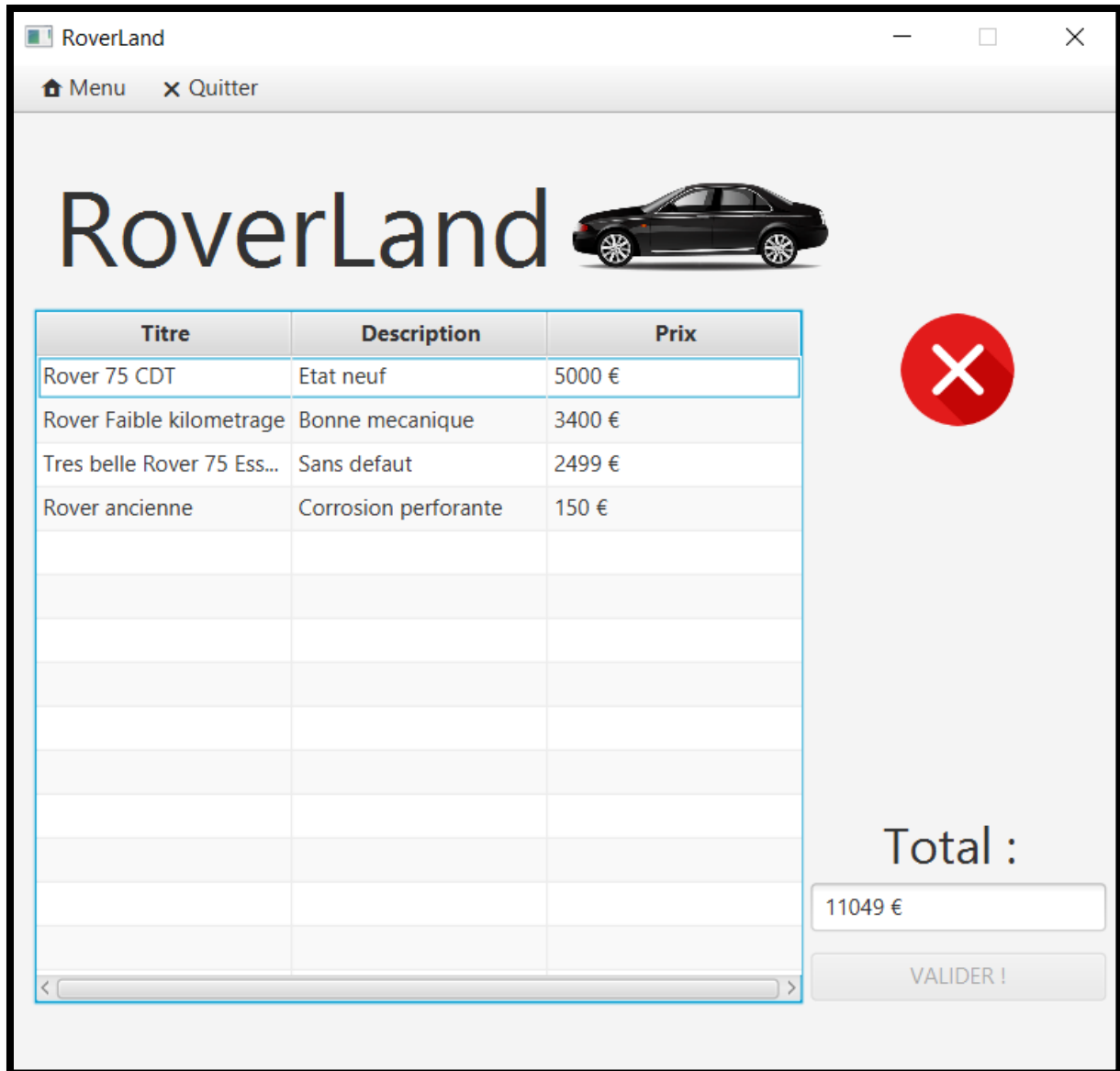


Figure 22: Vue « Panier »

### 3. Tests unitaires

Le choix s'est porté sur Java en grande partie grâce à un logiciel propre à Java : JUnit. JUnit est un framework de test unitaire pour le langage de programmation Java. Il permet de dérouler des tests automatiquement tout en vérifiant la cohérence des résultats. Les tests sont bien évidemment programmés par le développeur.



Figure 23 : Junit5

Il a donc été décidé de mettre en œuvre cet outil dans le cadre de ce projet de manière à valider les quelques fonctions nécessaires à l'application. Cela a permis de valider la partie front seule.

Voici ci-dessous la liste des classes avec les méthodes à tester :

- Classe App : Aucun test
- Classe Article : Aucun test
- **Classe Cart : delArticle**
- **Classe Site : addArticle / addNewCart**
- Classe User : Aucun test

En résumé, dans le cas présent, il paraît pertinent de tester les trois fonctions addArticle, addNewCart et delArticle, ceci étant à nouveau en accord avec le CRUD. Voici ci-dessous les tests qui ont été programmés :

```
@org.junit.jupiter.api.Test
@DisplayName("Add a new cart to the hashmap of the app")
public void test1() {
    int targetNumberOfCartsAfter = roverLand.getHmCart().size() + 1;
    roverLand.addNewCart(roverLand.getUser().getId());
    assertEquals(targetNumberOfCartsAfter, roverLand.getHmCart().size());
}

@org.junit.jupiter.api.RepeatedTest(10)
@DisplayName("Add a new article to the cart of the current user")
public void test2() {
    int targetNumberOfArticlesInCart = roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList().size();
    roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList().add(new Article("titleTest", "descTest"));
    assertEquals(targetNumberOfArticlesInCart, roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList().size());
}

@org.junit.jupiter.api.Test
@DisplayName("Delete an article from the cart of the current user")
public void test3() {
    roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList().add(article1);
    int targetNumberOfArticlesInCart = roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList().size();
    System.out.println("List : " + roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList());
    roverLand.getHmCart().get(roverLand.getUser().getId()).delArticle(article1);
    assertEquals(targetNumberOfArticlesInCart, roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList().size());
    assertTrue(!roverLand.getHmCart().get(roverLand.getUser().getId()).getArticleList().contains(article1));
}
```

Figure 23 : Tests unitaires

Sur les trois tests effectués on retrouve des tests classiques (Annotation @Test) avec l'utilisation de la fonction assert pour vérifier que le résultat obtenu est bien celui souhaité. Le deuxième test a également été répété un certain nombre de fois (Annotation @RepeatedTest)

de façon à bien vérifier que le résultat attendu est bien le bon lorsque l'on refait le test plus d'une seule fois.

Remarque : JUnit propose encore plein d'autres types de tests qui peuvent être appliqués selon le besoin (exemples d'annotations : `@ParameterizedTest`, `@BeforeEach`, `@AfterEach`, `@BeforeAll`, `@AfterAll`...). Pour créer un test, il suffit d'annoter une méthode de test avec une de ces annotations par exemple.

Voici ci-dessous les résultats obtenus suite au déroulement des tests précédemment introduits :

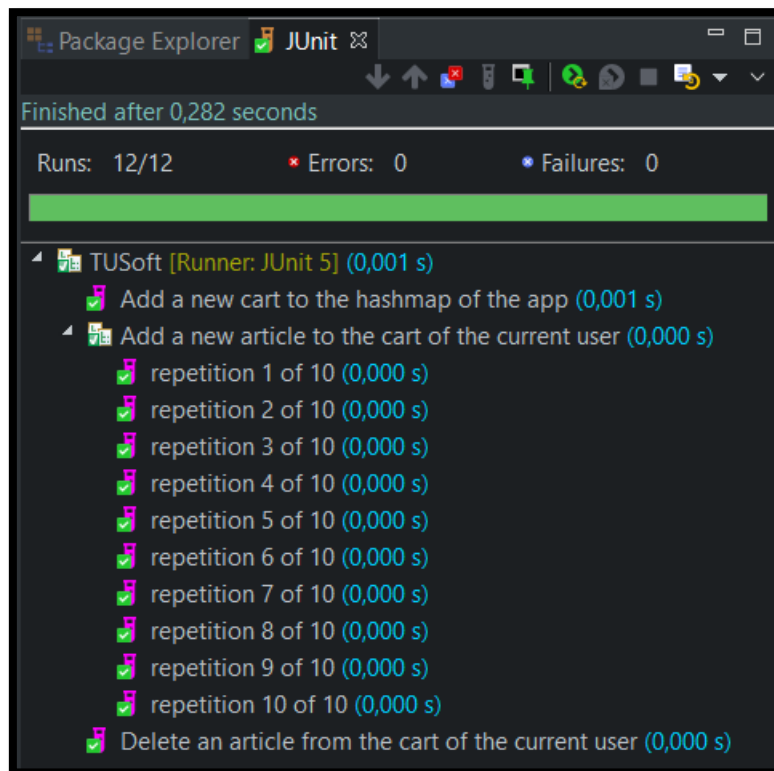


Figure 24 : Résultat des tests (JUnit)

Cette procédure a été très utile pour tester unitairement les fonctions et pour valider le fonctionnement global de la partie front. JUnit est vraiment un logiciel simple à mettre en œuvre et à utiliser. De plus, il compte plein de ressources à exploiter pour développer des tests.

## 4. Contrôles de qualité

### SonarLint

Toujours dans une démarche de tests, il a été nécessaire d'effectuer une analyse des programmes en continu pour assurer une bonne qualité des livrables.

Étant donné que l'IDE utilisé pour la partie front est Eclipse, il a été décidé d'utiliser un outil nommé SonarLint. Il permet d'anticiper les erreurs avant qu'elles n'arrivent. Cet outil s'ajoute à Eclipse comme un plugin et analyse la qualité des programmes, puis signale les problématiques en temps réel.



Figure 25 : SonarLint

SonarLint a été utile pour corriger des problèmes de plusieurs types, en voici quelques exemples :

- Commentaires inutiles
- Duplication de code
- Nom des variables
- Variables non utilisées
- Mauvaise gestion des exceptions
- Mauvaise gestion des logs

Ci-dessous, on peut observer de quelle manière SonarLint effectue le retour de ses analyses. Cela permet de les corriger au fil de l'eau, en même temps que le développement. Le point intéressant de cet outil est qu'en plus d'isoler les problèmes, il fournit en général une bonne indication permettant de corriger cette erreur.

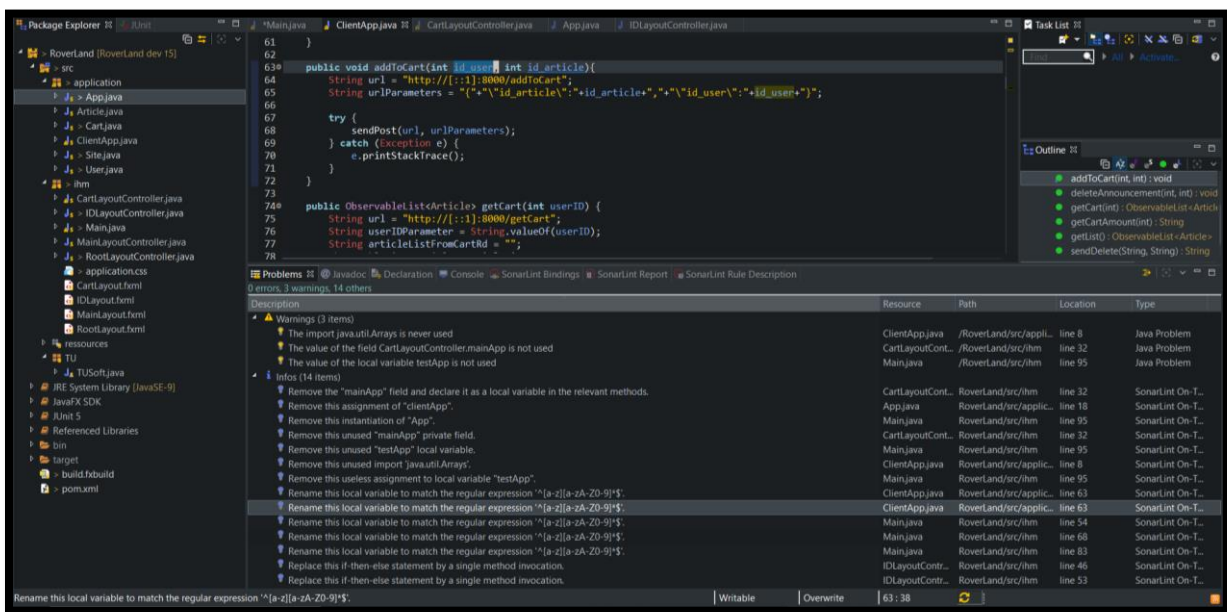


Figure 26 : SonarLint (Détection d'une variable mal nommée)

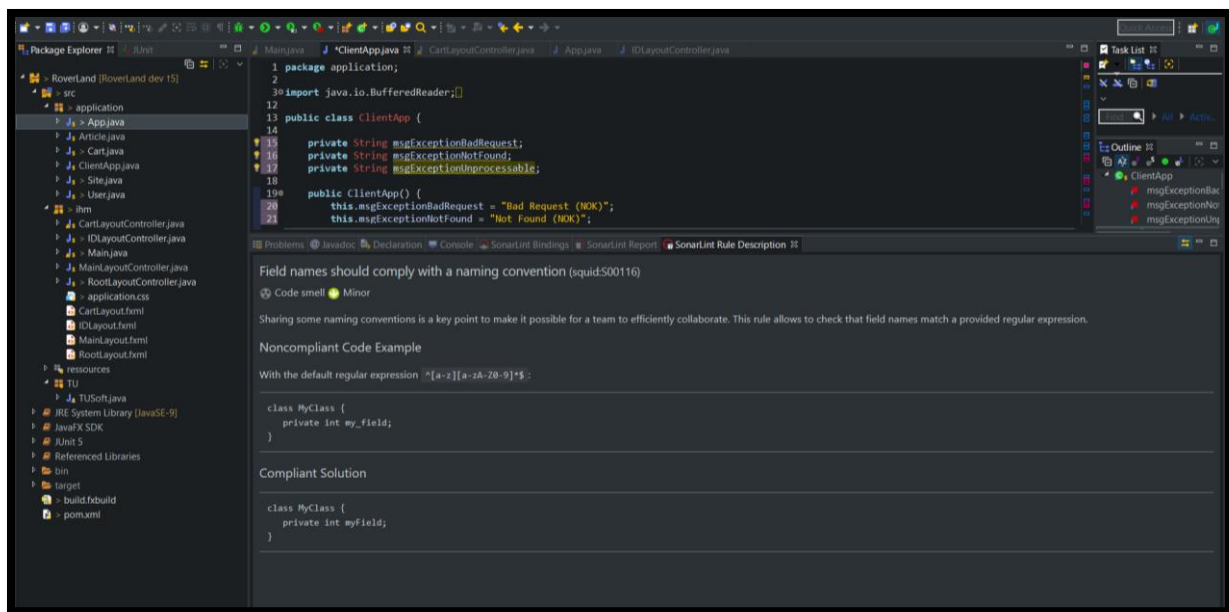


Figure 27 : SonarLint (Suggestion d'un nom de variable)

Remarque : la majorité des erreurs détectées ont été corrigées sauf éventuellement quelques-unes qui ont été conservées. Il s'agissait principalement de variables non utilisées. Celles-ci sont utiles si on souhaite tester l'application sans API. Par conséquent, ces variables n'ont pas été retirées.

## SonarQube

L'autre outil qui a été utilisé se nomme SonarQube. SonarQube permet également d'effectuer du contrôle de qualité de programmes et intègre lui-même les fonctionnalités de SonarLint. La différence est que cet outil est bien plus complet mais fonctionne avec un serveur qu'il faut lancer en local. Il est également possible d'utiliser un serveur sur un cloud de manière à générer des rapports de qualité sur les programmes analysés. C'est ce qui a été effectué.



Figure 28 : SonarQube

Remarque : L'installation de SonarQube est quelque chose de très fastidieux sous Windows. Il faut dans un premier temps vérifier la variable d'environnement associée au JRE (Java) utilisé, ensuite télécharger et installer Maven<sup>1</sup> et également ajouter la variable d'environnement associée à Maven. Ensuite, une fois le projet créé dans SonarCloud, il faut permettre à Sonar d'analyser du code. Pour cela, Sonar génère un token qu'il faut utiliser au moment de l'analyse pour se connecter au serveur. Enfin Sonar génère la ligne de commande qu'il faut exécuter dans le bon répertoire pour effectuer l'analyse.

<sup>1</sup> Maven : Maven est un outil de construction de projets Java (= « Compilateur » Java)

Sonar nous fournit plusieurs informations en ce qui concerne la qualité du code. Premièrement, il fournit le nombre de bugs qu'il a détecté et les potentielles vulnérabilités qui peuvent être une faille de sécurité pour l'application. Deuxièmement, Sonar effectue une analyse des mauvaises pratiques en programmation. C'est ce que l'on appelle les « code smells » en génie logiciel. À cela vient s'ajouter un calcul des dettes techniques (coûts supplémentaires liés à un non-respect des bonnes pratiques de programmation). Ensuite, on constate une mesure du coverage, qui concerne la couverture de tests de l'application. Enfin cet outil fournit le niveau de duplication dans le rapport général. Il s'agit de portions de programmes répliqués plusieurs fois pour faire plus ou moins la même chose. Il s'agit clairement d'un critère important car il permet de savoir s'il faut factoriser le code. Voici les résultats obtenus pour le programme « front » :

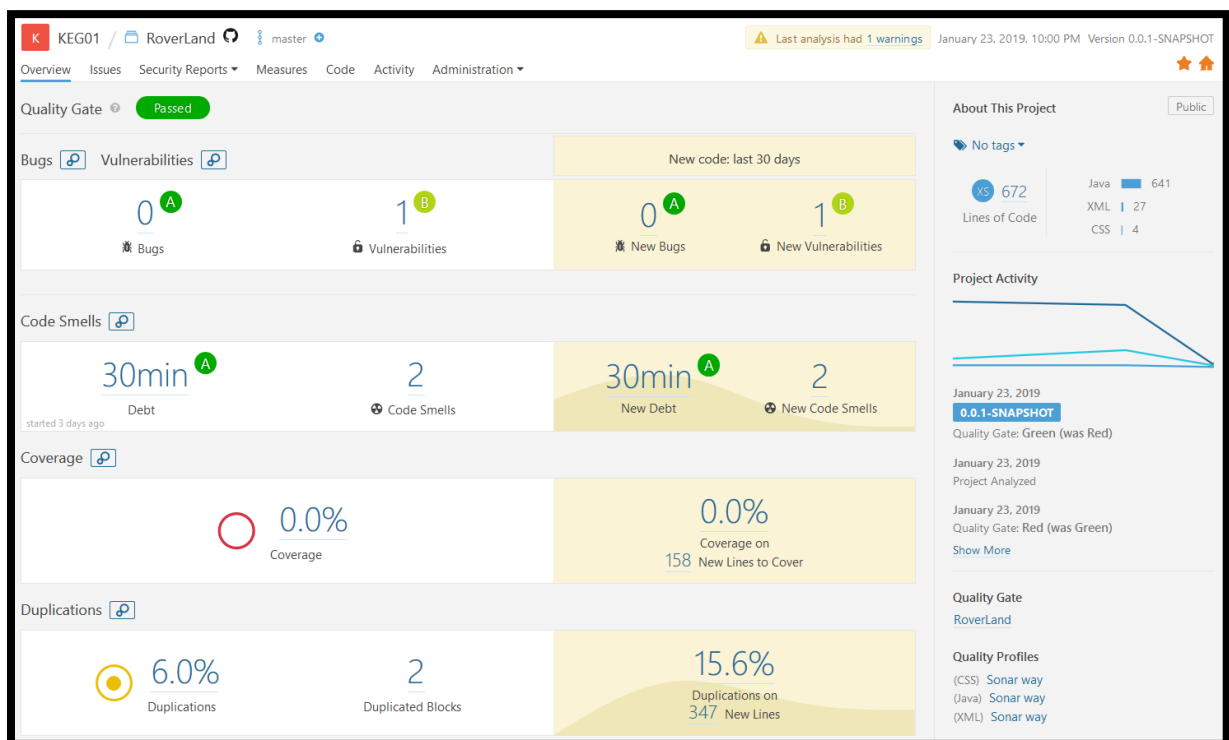


Figure 29 : Rapport SonarQube (SonarCloud)

Globalement, la majorité des vulnérabilités et des « code smells » ont été supprimées ou corrigées. Les seuls qui restent, sont liées aux quelques variables de tests qui ont été conservées. En ce qui concerne justement les tests, on observe que la couverture de tests est à 0.0 %, ce qui n'est pas forcément problématique. En effet, comme cela a été expliqué précédemment, une série de tests unitaires a été réalisée précédemment. En revanche, il semblerait que SonarQube n'arrive pas à faire le lien avec la classe liée aux tests. SonarCloud nous offre un suivi de l'évolution et fournit un résultat valide ou non valide selon les règles qui ont été définies.

**Remarques :** Le plugin Eclipse de SonarQube permet là également de corriger les erreurs directement lors des phases de développement. Il suffit de relancer une analyse.



10 items		
Resource	Date	Description
Article.java		⚠️ Immediately return this expression instead of assigning it to the temporary variable "temp".
CartLayoutCon		⚠️ Remove the "mainApp" field and declare it as a local variable in the relevant methods.
CartLayoutCon		⚠️ Remove this unused "mainApp" private field.
Main.java		⚠️ Remove this unused "testApp" local variable.
Main.java		⚠️ Remove this instantiation of "App".
Main.java		⚠️ Remove this useless assignment to local variable "testApp".
RootLayoutCoi	1 hour ago	⚠️ Add a nested comment explaining why this method is empty, throw an UnsupportedOperationException or comp...
TUSoft.java	12 hours ago	⚠️ Remove the declaration of thrown exception 'java.lang.Exception', as it cannot be thrown from method's body.
TUSoft.java	19 days ago	⚠️ Rename this field "RoverLand" to match the regular expression '^([a-z][a-zA-Z0-9]*)\$'.
TUSoft.java		⚠️ Rename this package name to match the regular expression '^([a-z_]+)([a-z_][a-z0-9_]*)*\$'.

Figure 30 : Rapport SonarQube (Eclipse)

La majorité des vulnérabilités et des erreurs ont également été traitées sur la partie WEB. Certains « warning » ont dû être omis étant donné qu'ils sont utiles si jamais on souhaite utiliser la partie front sans API (ex : Tests, Démonstrations...). C'est le cas pour les lignes commentées. Ces « code smells » ont été ignorés pour cette raison.

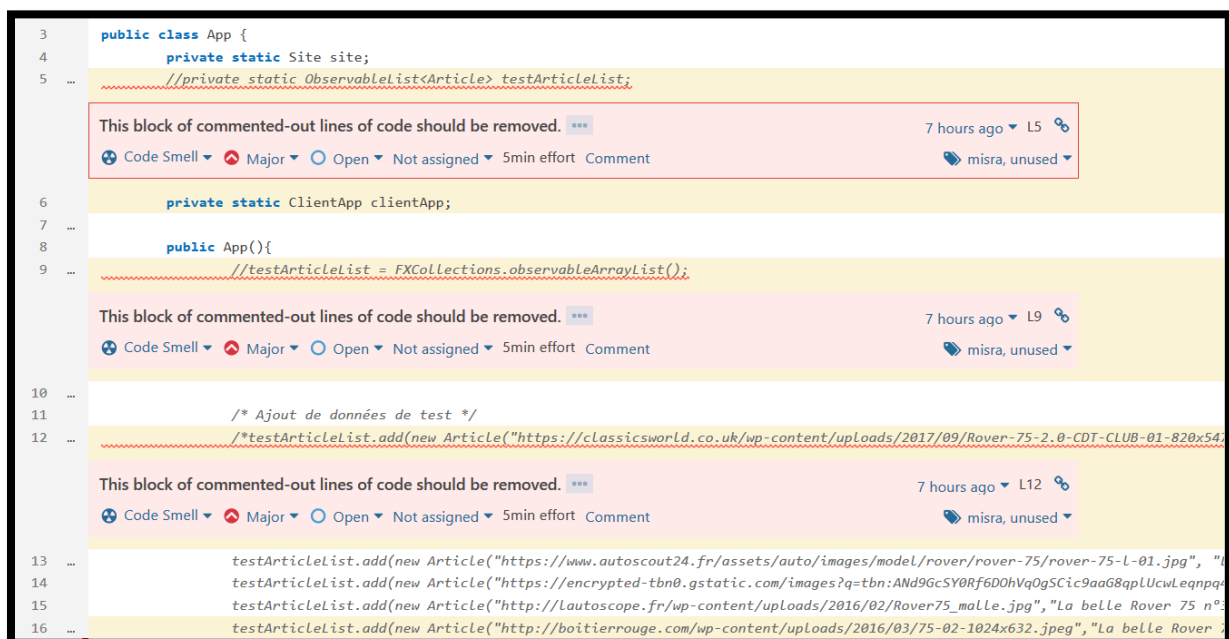


Figure 31 : Exemple de « Code Smells »

Il est également intéressant de remarquer le niveau de précision de cet outil pour détecter les portions de code dupliquées. Au vu des résultats, il a été nécessaire de revoir le programme et de le factoriser en fonctions (voir la classe ClientApp).

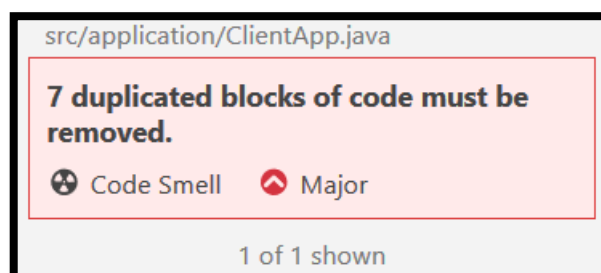
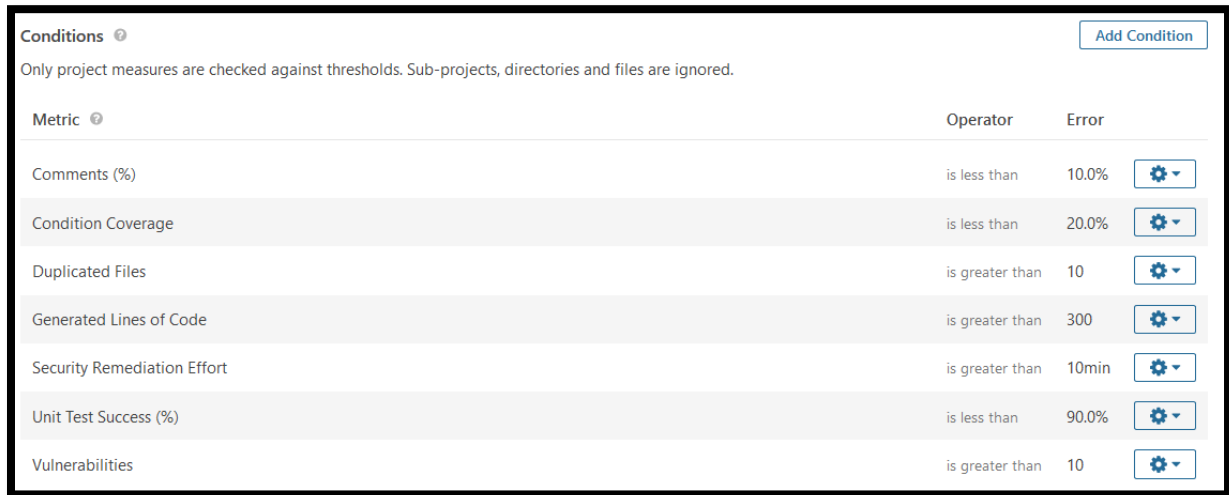




Figure 32 : Duplications

Il faut savoir qu'il existe des règles prédéfinies selon lesquelles les analyses sont effectuées, mais il est possible d'en ajouter tout en configurant le niveau de criticité. Voici ci-dessous un exemple de configuration des règles.



The screenshot shows the 'Conditions' configuration page in SonarQube. It lists various metrics and their corresponding operators and error thresholds. Each row has a gear icon to the right for configuration.

Metric	Operator	Error
Comments (%)	is less than	10.0%
Condition Coverage	is less than	20.0%
Duplicated Files	is greater than	10
Generated Lines of Code	is greater than	300
Security Remediation Effort	is greater than	10min
Unit Test Success (%)	is less than	90.0%
Vulnerabilities	is greater than	10

Figure 30 : Quality Gates

La majorité des tests sur SonarQube étaient basés sur les critères par défaut. Suite à cela, la qualité du programme a été validée.

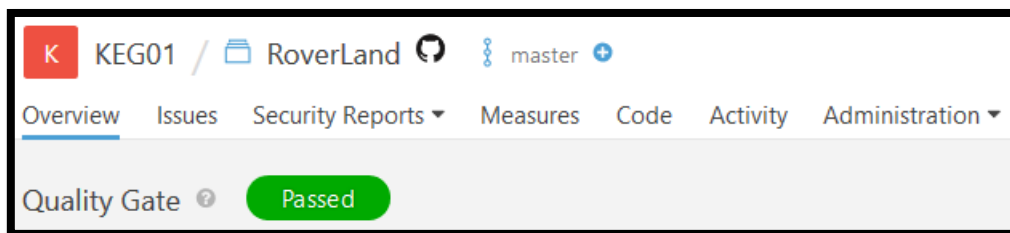


Figure 29 : Résultat SonarQube

## IV. Cahier de recettes

Comme dans tout projet, une fois les tests d'intégration effectués, il a été réalisé une série de tests pour effectuer la validation de l'application finale. En théorie, ce cheminement correspond plutôt au cycle en V. Néanmoins, notre méthodologie a consisté à valider au fil de l'eau même si l'idée était de suivre une méthode agile, soit plus flexible avec plusieurs livrables.

Voici ci-dessous le cahier de recettes avec tous les tests qui ont été effectués manuellement sur l'interface graphique à la fin du projet. On retrouve bien toutes les règles propres à chaque cas d'utilisation.

Cas d'utilisation	Test	Scénario	Résultat attendu	Validé ?	Conséquence(s)	Opérateur	Date	Commentaire
S'identifier	#login-0	Saisie de l'identifiant seul	Champ mot de passe en rouge	OK		Karim	23/01/2019	RAS
	#login-1	Saisie du mot de passe seul	Champ identifiant en rouge	OK		Karim	23/01/2019	RAS
	#login-2	Validation sans saisie	Champ identifiant / mot de passe en rouge	OK		Karim	23/01/2019	RAS
	#login-3	Saisie de l'identifiant et du mot de passe + validation	Identifiant enregistré en mémoire	OK		Karim	23/01/2019	RAS
Faire ses achats	#list-0	Clic sur un véhicule	Survollement de la ligne en bleu	OK		Karim	23/01/2019	RAS
	#list-1	Clic sur une colonne	Classement des articles par ordre alphabétique	OK		Karim	23/01/2019	Si on ajoute plusieurs fois le même article, celui-ci est dupliqué
	#list-2	Clic sur le bouton « + » (avec sélection)	Ajout de l'article sélectionné au panier	OK		Karim	23/01/2019	RAS
	#list-3	Clic sur le bouton « + » (sans sélection)	Aucune action	OK		Karim	23/01/2019	RAS
Modifier son panier	#list-4	Clic sur le bouton « panier »	Affichage de la page « panier »	OK		Karim	23/01/2019	RAS
	#cart-0	Clic sur un véhicule	Survollement de la ligne en bleu	OK		Karim	23/01/2019	RAS
	#cart-1	Clic sur une colonne	Classement des articles par ordre alphabétique	OK		Karim	23/01/2019	RAS
	#cart-2	Clic sur le bouton « delete » (avec sélection)	Suppression de l'article sélectionné du panier	OK		Karim	23/01/2019	Si plusieurs articles sont du même type, ils seront tous supprimés
	#cart-3	Clic sur le bouton « delete » (sans sélection)	Aucune action	OK		Karim	23/01/2019	RAS
	#cart-4	Affichage de la page	Envoi d'une requête au serveur pour récupérer le prix total	OK		Karim	23/01/2019	RAS

Figure 30 : Cahier de recette

VERDICT
Résultat Final : <b>VALIDE</b>
<b>Commentaires supplémentaires :</b> <i>Entièrement fonctionnel, conforme aux spécifications</i>

Figure 31 : Validation finale

Suite au déroulement de cette série de tests, l'application a été validée étant donné que chaque test a répondu aux spécifications fournies lors du premier livrable.

# V. Gestion de projet

## 1. GIT

Au démarrage de ce projet, beaucoup de temps a été passé dans la mise en œuvre d'un dépôt GIT pour pouvoir gérer le versionning de notre application. GIT étant plus ou moins connu pour nous, cela était l'opportunité de le reprendre en main sur ce projet, d'autant que c'est un outil qui permet à plusieurs membres d'une équipe de travailler en parallèle.

Pour utiliser GIT, il existe un outil assez intéressant nommé GitKraken. C'est un logiciel qui permet de grandement simplifier l'utilisation des logiciels de versionning et en particulier Git. Cet outil est très attractif pour son interface graphique simple et intuitive.

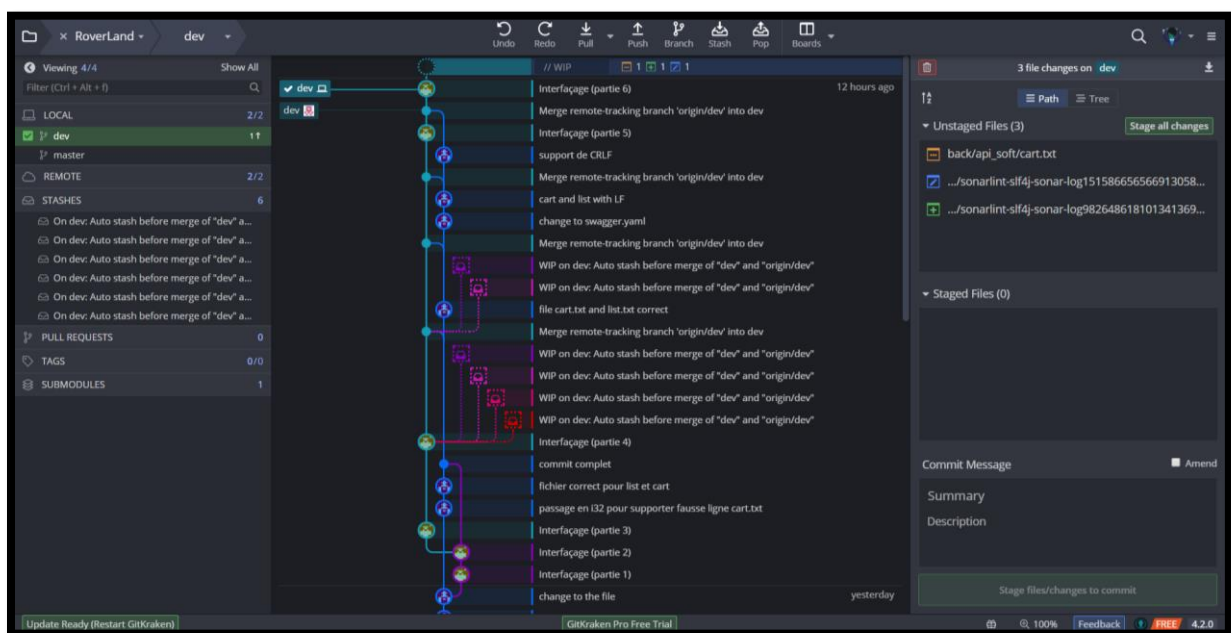


Figure 32 : GITKRAKEN

## 2. Méthode agile

Comme indiqué dans le cahier de spécifications, nous avons l'ambition de mettre en place une méthode agile. Nous avons essayé de le faire dans la mesure du possible notamment en rendant plusieurs versions du cahier de spécifications. Cela n'a pas pu se faire pour la partie développement. En revanche, nous avons fait en sorte de décomposer les tâches en îlots de manière à pouvoir travailler en parallèle et passer un peu de temps à l'intégration. Cette stratégie a d'ailleurs fonctionné étant donné que cela s'est fait très rapidement. De plus, une des particularités de la méthode agile était la flexibilité du projet. En effet, nous avons fait en sorte de réagir rapidement aux changements selon les erreurs constatées lors de l'intégration.

# Conclusion

En résumé, le projet est fonctionnel et répond parfaitement aux attentes du cahier des spécifications avec quelques évolutions. Nous avons fait en sorte de traiter le maximum de points sur le plan des tests avec JUnit puis Sonar et de la sécurité avec RUST. Il reste encore des optimisations à faire sur ces points-là et sur l'interface graphique. Néanmoins, nous estimons avoir couvert un large panel d'outils permettant de traiter beaucoup de problématiques dans ces domaines.

Nous avons rencontré de plus ou moins grandes difficultés au cours de ce projet. En effet, il a été difficile de prendre en main GIT au moment de la mise en place du dépôt. Néanmoins, nous avons su palier à la majorité de nos difficultés en s'entraidant et en effectuant des recherches en autonomie. Le seul point qui a été contraignant pour nous a été de trouver assez de créneaux horaires pour avancer sur le projet.

Toutes les compétences acquises dans cette discipline vont sûrement nous être utiles à l'avenir sur le plan professionnel. En effet, ceci est une certitude étant donné que c'est la première fois que l'on traite ces aspects-là, ce qui fait que nous disposons maintenant des outils pour effectuer des tests et répondre aux contraintes de sécurité.

Il est possible de retrouver les codes sources du projet et les analyses aux adresses suivantes :

<https://github.com/KEG01/RoverLand>

[https://sonarcloud.io/dashboard?id=KEG01\\_RoverLand](https://sonarcloud.io/dashboard?id=KEG01_RoverLand)

# Table des illustrations

Figure 1 : Logo Rover .....	3
Figure 2: Front-End / Back-End.....	4
Figure 3: RUST / Java .....	5
Figure 4: URLs API.....	6
Figure 5: Codes retour de l'URL getList .....	7
Figure 6: Paramètres de l'URL deleteAnnouncement.....	7
Figure 7: Lancement du serveur RUST .....	8
Figure 8: Crates.io.....	8
Figure 9: Requêtes HTTP .....	8
Figure 10: Serializer / Deserializer.....	9
Figure 11: addToCart .....	10
Figure 12: deleteAnnouncement.....	10
Figure 13: getCart.....	11
Figure 13: getCartAmount.....	11
Figure 14: getList .....	12
Figure 15: Curl .....	12
Figure 16: Plugin E(fx)clipse + Scenebuilder.....	13
Figure 17: Modèle MVC.....	13
Figure 18: Diagramme des classes .....	14
Figure 19: Arborescence des fichiers .....	15
Figure 20: Vue « Login » .....	16
Figure 21: Vue « Annonces ».....	17
Figure 22: Vue « Panier » .....	18
Figure 23 : Junit5 .....	19
Figure 23 : Tests unitaires .....	19
Figure 24 : Résultat des tests (JUnit).....	20
Figure 25 : SonarLint.....	21
Figure 26 : SonarLint (Détection d'une variable mal nommée) .....	21
Figure 27 : SonarLint (Suggestion d'un nom de variable) .....	22
Figure 28 : SonarQube.....	22
Figure 29 : Rapport SonarQube (SonarCloud).....	23
Figure 30 : Rapport SonarQube (Eclipse) .....	24
Figure 31 : Exemple de « Code Smells ».....	24
Figure 32 : Duplications.....	25
Figure 30 : Quality Gates .....	25
Figure 29 : Résultat SonarQube.....	25
Figure 30 : Cahier de recette .....	26
Figure 31 : Validation finale.....	27
Figure 32 : GITKRAKEN .....	28