Projet Master Informatique MIAGE
Interopérabilité des systèmes d'information

Tokotoko Jannaï

2023

1 Introduction

Ce projet consiste à réaliser une API *RESTful* pour rendre inter-opérable deux applications : une application de délimitation foncière et une application de génération de contrat intelligent (smart contrat) de type baux ruraux. L'utilité du contrat intelligent "BailRural" est de stocker les informations de propriété foncière de manière transparente sur la blockchain (Ethereum). Cela permet de résoudre les problèmes liés aux litiges fonciers et à la falsification de documents fonciers.

1.1 L'application Sprint Boot à développer

L'application Spring contiendra

Le contrôleur : Expose une API (REST) au travers d'un controleur, pour interagir avec le front-end et recevoir les demandes de création de smart contracts. : Le contrôleur envoie une réponse au front-end pour informer de la réussite ou de l'échec de la demande.

Le service de vérification de données : "checkService"

- Vérifie par l'algorithme de Jarvis (cf. fin d'énoncé) si les points envoyés forment bien un polygone. Il envoie le message "*Le polygone n'est pas valide*" si les points ne forment pas de polygone.
- Envoie une requête au service "deploy" pour la l'enregistrement du contrat, si les points forment un polygone.

Le service de déploiement de smart contrat : "deployService" :

- Est responsable de la création de smart contracts de baux ruraux.
- Déploie le smart contrat dans Ganache, Utilisez le code tout fait par ChatGPT pour le déploiement : *DeployContract.java*
- Renvoie une réponse au contrôleur pour informer de l'insertion effectif du smart contrat dans la base de données.

1.2 Diagramme de séquence

Le diagramme de séquence de l'image 1 représente les interactions entre les clients et les deux services.

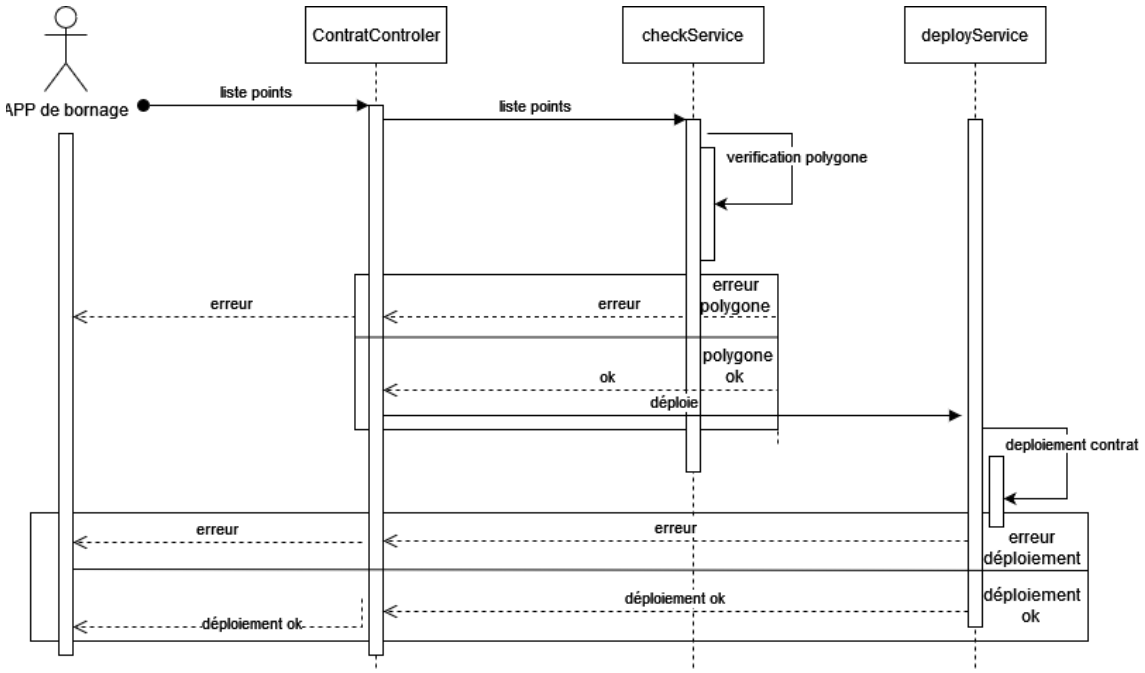

Figure 1: Diagramme de séquences

1.3 Format de données

L'image 2 montre un polygone et le format GeoJson de données est décrit ce dessous :

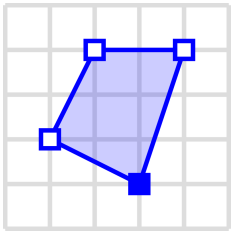

Figure 2: Polygone et point GPS

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {

      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [
              166.21704,
              -21.615572
            ],
            [
              166.219218,
              -21.616968
            ],
            [
              166.217426,
              -21.618046
            ],
            [
              166.215205,
              -21.617367
            ],
            [
              166.21704,
              -21.615572
            ],
          ]
        ]
      }
    }
  ]
}
```


L'objet *FeatureCollection* contient une liste de feature. Dans cette liste l'objet *geometry* de type *polygone* enregistre l'ensemble des points GPS.

Le client à compléter permet l'envoie de ces points.

1.4 Le client

Folium permet de manipuler des données géographiques avec *Python* et *Leaflet*. *Leaflet* permet de générer le visuel de la cartographie. Le fichier App_a_completer.java permet de lancer une map à partir de la bibliothèque folium. L'objet *folium.map* est instancié dans la fonction *loadPage(self)*.

Les composantes de l'application qui permettent de dessiner un polygone sur une image sont ajouté à la map à partir de la class *Draw* (**folium.plugins.draw import Draw**)

Avec ce client servez vous de la bibliothèque *requests* de *Python* pour l'envoie des requêtes au serveur spring boot. Exemple d'envoi de points en utilisant *requests* sur un point de terminaison "*http://localhost:8080/polygons*"

```
import requests

url = "http://localhost:8080/polygons"  # URL de l'API à appeler
points = [
    [166.21704, -21.615572],
    [166.219218, -21.616968],
    [166.217426, -21.618046],
    [166.215205, -21.617367],
    [166.21704, -21.615572]
]  # Liste des points à envoyer dans le corps de la requête

# Envoi de la requête POST avec les points dans le corps de la requête
response = requests.post(url, json=points)

# Vérification du code de retour de la réponse
if response.status_code == requests.codes.ok:
    # La requête a été traitée avec succès
    print("Requête traitée avec succès.")
else:
    # La requête a échoué
    print("La requête a échoué avec le code de retour {}".format(response.status_code))
```

2 Description du contrat

Le contrat "BailRural" est un contrat intelligent (smart contract) écrit en langage de programmation Solidity, destiné à être exécuté sur une blockchain Ethereum.

Descriptif de ses fonctions et de ses événements :

- Struct "LandInformation" : structure de données utilisée pour stocker les informations de propriété de la terre. "owner" est une chaîne de caractères pour stocker le nom du propriétaire de la terre, et "gpsPoints" qui est un tableau de type uint pour stocker les délimitation foncière (points GPS).

- Event "LandRecorded" : Cet événement déclenché chaque fois qu'un enregistrement de propriété foncière est ajouté avec succès au contrat.
- Fonction "recordLand" : fonction utilisée pour enregistrer les informations de propriété de la terre. avec en entrée le nom du propriétaire et les points GPS de la terre.
- Fonction "getLandInformation" : fonction utilisée pour récupérer les informations de propriété de la terre pour une adresse Ethereum donnée. Elle renvoie le nom du propriétaire et les points GPS de la terre associés à l'adresse Ethereum.

3 les étapes pour déployer vos contrats

Voici les étapes générales pour utiliser Web3j avec Ganache:

- Installez Ganache sur votre ordinateur.
- Ajoutez la dépendance *Web3j* à votre projet Java en utilisant *Maven* ou *Gradle*.
- Créez une instance Web3j en utilisant l'URL de votre nœud Ganache local.
- Chargez votre contrat intelligent dans Web3j en utilisant son adresse.

Utilisez fichier DeployContract.java pour le déploiement de votre contrat sur Ganache. Vous pouvez aussi déployez un contrat intelligent sur Ganache en utilisant *Remix*, comme outil de développement de contrats intelligents disponible sur *https://remix.ethereum.org/*.

3.1 Vérification d'un polygone par l'algorithme de Jarvis

```
import java.awt.geom.Line2D;
import java.util.List;

public boolean isPolygon(List<Point> points) {
if (points.size() < 3) { // Un polygone doit avoir au moins 3 points
    return false;
}
// Vérifier si le dernier point est identique au premier point pour former une boucle
if (!points.get(0).equals(points.get(points.size() - 1))) {
    return false;
}
// Vérifier si les segments formés par les points se croisent
    for (int i = 0; i < points.size() - 2; i++) {
        for (int j = i + 2; j < points.size() - 1; j++) {
            if (Line2D.linesIntersect(
                points.get(i).getLatitude(),
```

```
                points.get(i).getLongitude(),
                points.get(i + 1).getLatitude(),
                points.get(i + 1).getLongitude(),
                points.get(j).getLatitude(),
                points.get(j).getLongitude(),
                points.get(j + 1).getLatitude(),
                points.get(j + 1).getLongitude())) {
            return false;
            }
        }
    }
    return true;
}
```