

SAE 4.02

—

INGENIEUR LOGICIEL 1

—

PROJET D'APPLICATION
REPARTIE

Bastien JALLAIS, Titouan LETONDAL, Alann MONNIER, Fabien TOUBHANS

18/06/2024

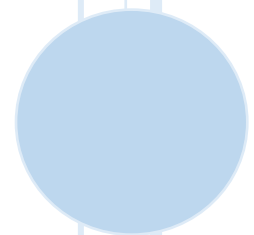


Table des matières

1. Introduction	2
1.1 Contexte et objectifs du projet	2
1.2 Informations complémentaire sur le projet	2
1.3 Etapes d'installation pour lancer le projet	2
2. Introduction du projet	4
2.1 Schémas représentant l'architecture du projet	4
3. Service RMI qui interroge une base de données	5
3.1 Présentation	5
3.2 Partie RMI	6
3.3 Gestion des transactions sur la base de données	8
3.4 Renvoi des données sous le format JSON	10
4. Des données ouvertes...	11
4.1 Données partout	11
4.2 Visualisation des données	11
5. Un service qui interroge les données bloquées...	12
5.1 Des données bloquées sur le client (Observations)	12
6. Un proxy qui interroge des services	12
6.1 Présentation	12
5.2 Un service qui interroge des données ouvertes	13
6. Un proxy qui interroge des services	14
6.1 Présentation	14
6.1 Question	14
7. Travail complémentaire	15
7.1 Présentation	15
8. Conclusion	15
8.1 Présentation des résultats obtenus	15

SAE 4.02-IL1 : PROJET D'APPLICATION REPARTIE

1. Introduction

1.1 Contexte et objectifs du projet

Ce rapport détaille le processus de conception et de développement entrepris lors du projet de fin de module, visant à créer une application répartie d'informations sur la ville de Nancy. Dans le cadre de la SAE 4.02, nous avons abordé le développement d'une application répartie en récupérant des données via des services RMI qui interroge des bases de données transactionnelles distantes.

Le document s'articule autour de la réalisation de différents modules afin de développer une application web qui affiche sur une carte des informations hétérogènes continue sur la ville de Nancy. La totalité de ces informations mises à jour transite pour la plupart via un proxy qui fait la passerelle entre le navigateur et les services RMI.

Dans ce rapport, nous exposons les différentes démarches, méthodes et stratégies adoptées à la réalisation du projet. Nous avons mis en œuvre des services RMI, des transactions sur les bases de données et de développement web.

Tout au long du rapport, des schémas, des images ainsi que des remarques accompagnent nos différentes explications pour faciliter la présentation de notre travail.

1.2 Informations complémentaire sur le projet

- Notre site web est disponible sur le serveur Webetu au lien suivant :
[Carte LeafletJS - Nancy \(univ-lorraine.fr\)](http://Carte.LeafletJS-Nancy.univ-lorraine.fr)
- Le dépôt Git contenant tous nos codes sources de l'application :
git@github.com:Titoufeur/SAE_Application_repartie.git

1.3 Etapes d'installation pour lancer le projet

Etape 1 :

- Sur une machine **A**, dans un terminal du dossier `./src/Service` lancer la commande `rmiregistry` afin de lancer l'annuaire

Etape 2 :

- Dans une base de données Oracle SQLDeveloper, lancer le script « `database.sql` » dans la racine du projet afin de créer les schémas de données, ainsi que d'ajouter un jeu de données de test.

Etape 3 :

- Lancer dans un autre terminal de la machine **A** dans le dossier `./src/Service` lancer la classe `LancerService`.
- `java -cp '.;ojdbc11.jar' LancerService identifiantOracle mdpOracle`

Etape 4 :

- Sur une machine **B**, dans un terminal dans le dossier `src/main/java` du projet lancer la classe `LancerProxy <port>`.
- `java LancerProxy 8080`

Etape 5 :

- Pour tester avec 3 machines différentes, modifiez le fichier « `index.js` » et modifiez la variable ligne 12 « **urlProxy** » et remplacez « **localhost** » par **l'adresse IP** de la machine **B**.
- Lancer sur une machine **C**, en local, le fichier « **index.html** » dans un navigateur.

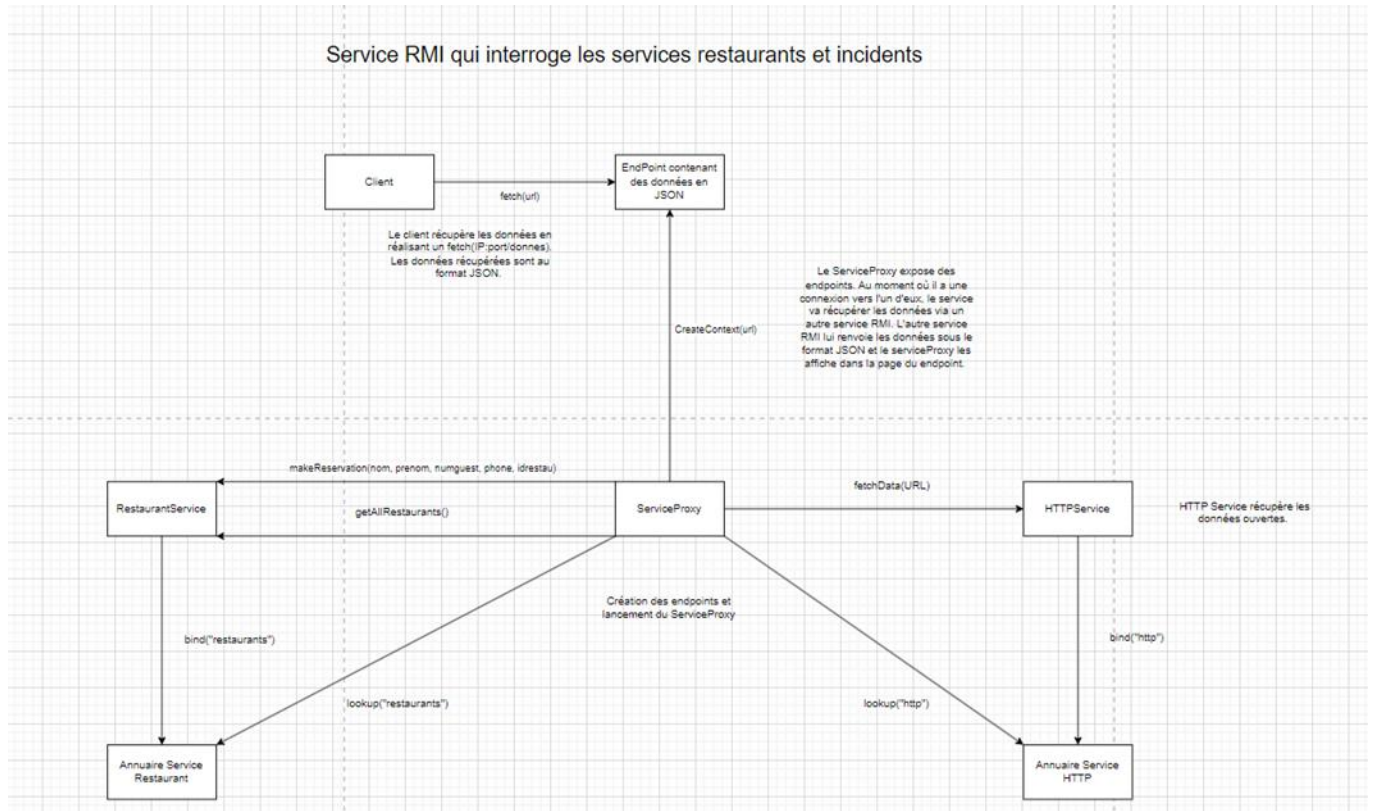
Etape 6 :

- Sinon, pour lancer depuis webetu, lancez le Proxy depuis la machine B et depuis cette même machine, cliquez sur le lien suivant pour utiliser l'application : https://webetu.iutnc.univ-lorraine.fr/~letondal3u/SAE_Application_repartie/

Pour pouvoir accéder au proxy depuis webetu, il est important que le Proxy soit lancé sur la même machine que celle qui exécute l'application, car si on tente de fetch une adresse IP externe, il nous oblige à que la requête soit en « `https` », et nous n'avons pas eu le temps de configurer le serveur pour qu'il puisse accepter les requêtes `https`. L'application hébergée sur webetu effectue donc un fetch vers <http://localhost:8080>, qui est autorisée en non `https` car ça ne change pas de domaine.

2. Introduction du projet

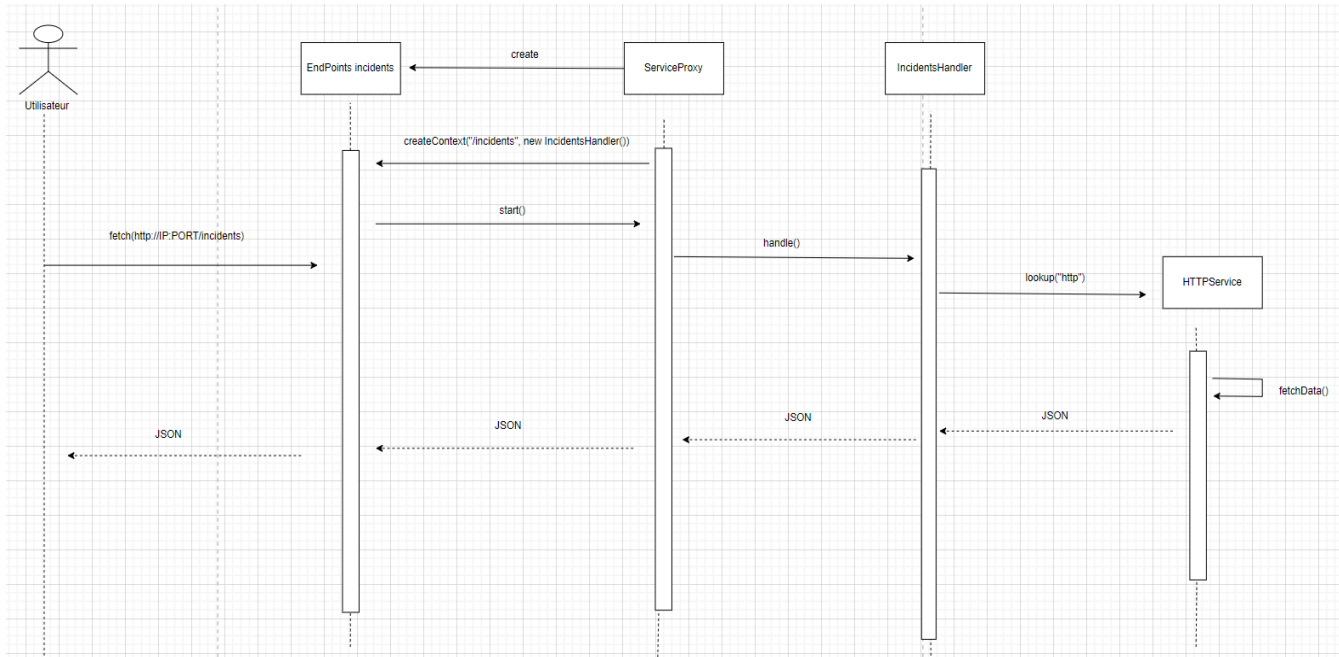
2.1 Schémas représentant l'architecture du projet



Ce premier schéma représente l'architecture de notre projet pour pouvoir récupérer les différentes données telles que les informations sur les restaurants ou les incidents.

Le client (côté JS) va fetch vers l'adresse suivante : 'IPMachine :port/typeDonnées'. Par exemple, le client peut fetch pour récupérer les données des incidents vers cette url : 192.168.0.1:8080/incidents. Ce lien va comporter le JSON des informations sur les incidents, cette page joue le rôle d'endpoints. (Extrémité d'un canal de communication. Point de contact entre deux systèmes).

Le service proxy quant à lui va créer des endpoints pour chaque type de données que l'on veut récupérer. (incidents, restaurants). Le service proxy à chaque requête vers le endpoint va appeler le service rmi comportant les données souhaitées en appelant la méthode du service. Il affiche ensuite leur code JSON sur la page et le client va donc pouvoir les récupérer.



Ce diagramme de séquence explique comment l'utilisateur peut récupérer les données JSON sur les incidents. Au moment où l'utilisateur pointe vers le endpoint d'incidents avec sa méthode fetch, le service proxy va alors détecter l'évènement avec la méthode handle. Cette méthode va récupérer le service rmi http sur une machine distante. Ce service va alors avec la méthode fetchData récupérer le JSON de l'url passé en paramètre sur les incidents. Enfin ces informations sont remontées jusqu'à l'utilisateur.

3. Service RMI qui interroge une base de données

3.1 Présentation

Mettre en place un service RMI et un client de test pour pouvoir accéder à la base de données restaurant. Nous devons ajouter des données d'entrées pour les restaurants telles qu'un nom, une adresse, et leurs coordonnées.

Le service mise en place doit pouvoir s'inscrire sur le service central, récupérer les informations sur les restaurants et permettre de réserver en précisant des informations sur la personne réalisant la réservation.

Toutes les informations renvoyées par le service se font au format JSON.

3.2 Partie RMI

```

1  import java.rmi.registry.Registry;
2  import java.rmi.registry.LocateRegistry;
3  import java.rmi.NotBoundException;
4  import java.rmi.RemoteException;
5
6  public class Client {
7      Run | Debug
8      public static void main(String[] args) {
9          if (args.length < 1) {
10             System.err.println("Usage: java Client <server_ip>");
11             return;
12         }
13         try {
14             Registry reg = LocateRegistry.getRegistry(args[0], 1099);
15             RestaurantService rs = (RestaurantService) reg.lookup("restaurants");
16
17             System.out.println(rs.getAllRestaurants());
18             System.out.println("Essai de faire une réservation : ");
19             boolean response = rs.makeReservation("Bastien", "Jallais", 1, "06 06 06 06 06", 6);
20             System.out.println(response);
21         } catch (Exception e) {
22             System.err.println("Erreur client : " + e.getMessage());
23             e.printStackTrace();
24         }
25     }

```

La classe Client Restaurant nous a permis de tester notre service RMI pour récupérer les données sur le service Restaurant. On va pouvoir récupérer les différents restaurants et réaliser une réservation.

```

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RestaurantService extends Remote {
    String getAllRestaurants() throws RemoteException;
    boolean makeReservation(String firstName, String lastName, int numPersons, String contactNumber,
                           int time) throws RemoteException;
}

```

L'interface restaurant que nous partageons entre le client et le serveur nous permet d'implémenter les méthodes de réservation et de récupération de restaurant.

```
public class ServiceRestaurant implements RestaurantService {

    private String motdepasse;
    private String identifiant;

    public ServiceRestaurant(String identifiant, String motdepasse){
        this.motdepasse = motdepasse;
        this.identifiant = identifiant;
    }
}
```

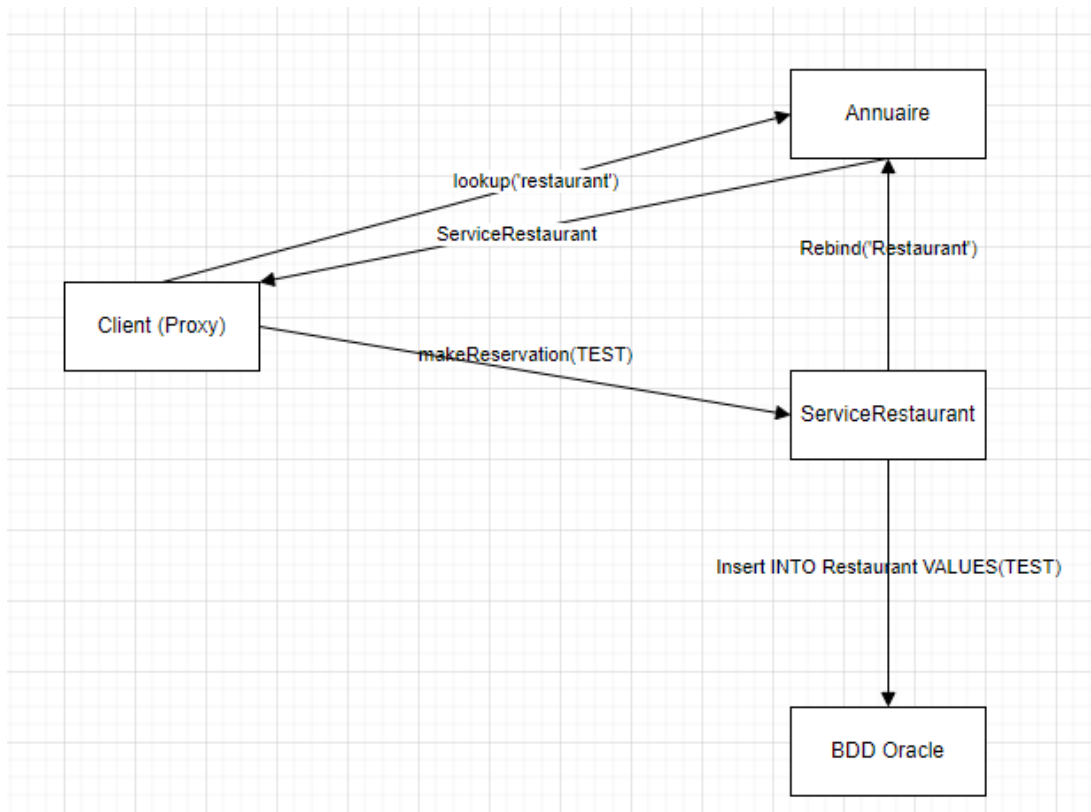
La classe ServiceRestaurant implémente l'interface précédente et crée les deux méthodes.

```
public class LancerServiceRestaurant {
    Run | Debug
    public static void main(String[] args){
        System.out.println("Avant lancement.");
        try {
            ServiceRestaurant sr = new ServiceRestaurant(args[0], args[1]);
            System.out.println("Avant export.");
            RestaurantService rs = (RestaurantService) UnicastRemoteObject.exportObject(sr, 0);

            System.out.println("Après export.");
            Registry reg = LocateRegistry.getRegistry();
            reg.rebind("restaurants", rs);

            System.out.println("Service de restaurant lancé avec succès.");
        } catch (RemoteException rm){
            rm.printStackTrace();
        }
    }
}
```

La classe LancerServiceRestaurant va enregistrer le service restaurant dans l'annuaire avec la méthode bind.



Ce schéma présente comment nous avons mis en place le service RMI pour réserver un restaurant.

3.3 Gestion des transactions sur la base de données

```

public Connection connect() throws SQLException {
    try {
        Class.forName("oracle.jdbc.OracleDriver");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    Connection connection = DriverManager.getConnection("jdbc:oracle:thin:@charlemagne.iutnc.univ-lorraine.fr:1521:xe");
    if (connection != null) {
        System.out.println("Connexion reussie a la base de donnees !");
    } else {
        System.out.println("Échec de la connexion à la base de données !");
    }
    return connection;
}
  
```

Dans la classe ServiceRestaurant, la méthode connect va nous permettre de se connecter à la base de données via oracle du Restaurant.

```

public String getAllRestaurants() throws RemoteException { no usages  Titoufeur
    List<Restaurant> restaurants = new ArrayList<>();
    try {
        Connection conn = connect();
        Statement stmt = conn.createStatement();
        String sql = "SELECT * FROM RESTAURANTS";
        ResultSet rs = stmt.executeQuery(sql);
        while (rs.next()) {
            int id = rs.getInt( columnLabel: "id");
            String name = rs.getString( columnLabel: "name");
            String address = rs.getString( columnLabel: "address");
            String gpsCoordinates = rs.getString( columnLabel: "gps_coordinates");
            System.out.println(id + " | " + name + " | " + address + " | " + gpsCoordinates);
            restaurants.add(new Restaurant(id, name, address, gpsCoordinates));
        }
        rs.close();
        stmt.close();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new RemoteException("Erreur de base de données.");
    }

    JSONArray jsonArray = new JSONArray();
    for (Restaurant restaurant : restaurants) {
        JSONObject jsonObject = new JSONObject();
        jsonObject.put("id", restaurant.getId());
        jsonObject.put("name", restaurant.getName());
        jsonObject.put("address", restaurant.getAddress());
        jsonObject.put("gpsCoordinates", restaurant.getGpsCoordinates());
        jsonArray.put(jsonObject);
    }

    return jsonArray.toString();
}

```

```

    }
    rs.close();
    stmt.close();
} catch (SQLException e) {
    e.printStackTrace();
    throw new RemoteException("Database error.");
}
System.out.println("Restaurants retournes : ");
System.out.println(restaurantsListToJson(restaurants));
return restaurantsListToJson(restaurants);
}

```

Dans la méthode `getRestaurant`, nous récupérons la totalité des informations sur les restaurants. Nous n'utilisons pas de transactions dans cette méthode, car en suivant les principes ACID, passer la base de données en mode transaction ici est inutile et ne causerait qu'une perte de performances. En cas d'erreur, on affiche un message d'erreur.

```

public boolean makeReservation(String firstName, String lastName, int numGuests, String phone, int restaurantId)
    String sql = "INSERT INTO RESERVATIONS (id, first_name, last_name, num_guests, phone, restaurant_id) VALUES
    Connection conn = null;
    try {
        conn = connect();
        // Mets l'autocommit à false
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, firstName);
        pstmt.setString(2, lastName);
        pstmt.setInt(3, numGuests);
        pstmt.setString(4, phone);
        pstmt.setInt(5, restaurantId);
        pstmt.executeUpdate();
        System.out.println("la réservation est bien enregistré dans la base de donnée");
        // Valide la modification
        conn.commit();
        return true;
    } catch (SQLException e) {
        // Annule la mise à jour
        //conn.rollback();
        e.printStackTrace();
    }

```

```

    } catch (SQLException e) {
        e.printStackTrace();
        try{
            if (conn != null){
                conn.rollback(); //On annule la transaction en cas d'erreur
            }
        } catch (SQLException sqlE){
            sqlE.printStackTrace();
        }
        System.out.println("Erreur lors de la réservation.");
        throw new RemoteException("Erreur de base de données.");
    } finally {
        try {
            if (conn != null) {
                conn.setAutoCommit(true);
            }
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}

```

Dans la méthode réservation, on gère les transactions en imposant une isolation de type sérialisable pour éviter qu'un autre utilisateur accède à la table pendant que l'utilisateur courant ajoute une donnée ou non. Pour ce faire, on ajoute également le commit à false et on valide l'insertion que s'il n'y a pas eu d'erreur. En cas d'erreur, on fait un rollback pour retirer l'insertion et annuler la transaction.

3.4 Renvoi des données sous le format JSON

```
JSONArray jsonArray = new JSONArray();
for (Restaurant restaurant : restaurants) {
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("id", restaurant.getId());
    jsonObject.put("name", restaurant.getName());
    jsonObject.put("address", restaurant.getAddress());
    jsonObject.put("gpsCoordinates", restaurant.getGpsCoordinates());
    jsonArray.put(jsonObject);
}

return jsonArray.toString();
```

Afin de renvoyer les données au format JSON, on utilise la classe JSONObject. On construit l'objet petit à petit en associant chaque attribut à une valeur, avec les valeurs que nous venons de récupérer, puis créons cet objet afin de le retourner.

4. Des données ouvertes...

4.1 Données partout...

Via le lien fourni, nous pouvons récupérer différentes informations sur les vélos Stan de la ville de Nancy. Nous pouvons récupérer des informations sur les stations comme son adresse, sa capacité d'accueil de vélo, ses coordonnées, son nom et son identifiant.

```
{"data":{"stations":[{"address":"au milieu du boulevard des Nations à Vandoeuvre dans le prolongement de l'arrêt de bus Nations devant le centre commercial","capacity":15,"lat":48.662356,"lon":6.173442,"name":"00030 - NATIONS","station_id":"30"},{"address":"Place du
```

Image : Données disponible sur les stations de vélo à Nancy

De même, on peut retrouver des informations sur la disponibilité des vélos sur une station donnée.

```
{"data":{"stations":[{"is_installed":1,"is_renting":1,"is_returning":1,"last_reported":1718211972,"num_bikes_available":4,"num_docks_available":11,"station_id":"30"},
```

Image : Vélos disponible sur les stations de vélo à Nancy

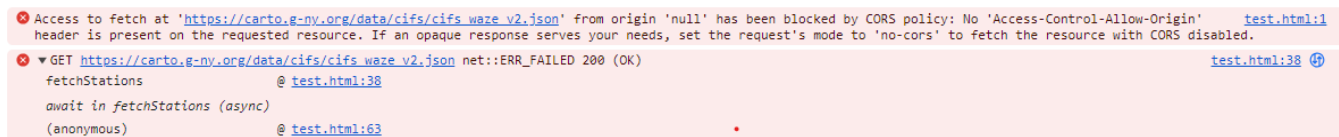
4.2 Visualisation des données



5. Un service qui interroge les données bloquées...

5.1 Des données bloquées sur le client (Observations)

L'objectif de cette partie, est d'essayer de récupérer des données précises du data grand est.



Les données sont bloquées à cause de la politique CORS. En effet, le client web (notre navigateur) tente de faire une requête HTTP à un domaine externe de celui à partir duquel l'application est chargée. C'est donc pour des raisons de sécurité que notre navigateur bloque ces requêtes, car elles peuvent être des vulnérabilités importantes.

5.2 Un service qui interroge des données ouvertes

Pour pouvoir accéder aux données ouvertes, on crée un service RMI qui va récupérer ces données via un URL.

On commence tout d'abord par créer l'interface qui est partagée avec le client.

```
import java.io.IOException;
import java.net.URISyntaxException;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HTTPService extends Remote{
    String fetchData(String url) throws IOException, InterruptedException, URISyntaxException;
}
```

Ensuite, on crée la classe qui va implémenter la méthode fetchData.

La méthode va récupérer les données accessibles à l'URL donnée. La première ligne va générer et construire une requête http. La seconde ligne envoie la requête et récupère les informations souhaitées. La troisième ligne récupère le statut de la requête (code 200 si succès). Si celle-ci est correcte on retourne le code obtenu à l'URL indiqué.

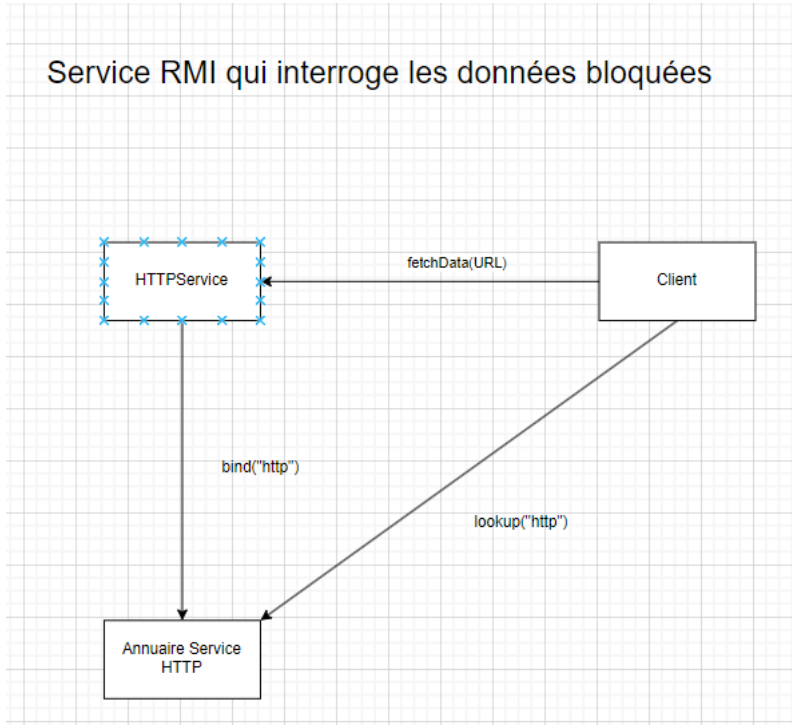
```
@Override
public String fetchData(String url) throws IOException, InterruptedException, URISyntaxException {
    HttpRequest request = HttpRequest.newBuilder().uri(new URI(url)).build();
    HttpResponse<String> response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());
    int statusCode = response.statusCode();
    if (statusCode == 200) {
        return response.body();
    } else {
        throw new IOException("HTTP Error: " + statusCode);
    }
}
```

Pour réaliser nos tests, on crée un client qui va récupérer le service dans l'annuaire et récupérer les informations de l'URL via la méthode précédente.

```
public class ClientHTTP{
    //passer en argument l'IP de la machine hébergeant le service
    Run|Debug
    public static void main(String[] args) throws RemoteException, NotBoundException, ServerNotActiveException {
        try{
            //Récupérer l'annuaire
            Registry reg = LocateRegistry.getRegistry(args[0], 1099);
            /*
             * Créer une instance de l'objet du service
             * Récupérer le service grâce à son nom dans l'annuaire avec la méthode lookup
             */
            HTTPService hs = (HTTPService) reg.lookup("http");
            /*Lancer le service*/
            System.out.println(hs.fetchData("https://carto.g-nv.org/data/cifs/cifs_waze_v2.json"));
            //System.out.println("Maintenant on essaye de faire une réservation : ");
            //boolean response = rs.makeReservation("Titouan", "LETONDAL", 1, "06 07 09 37 39", 1);
            //System.out.println(response);
        } catch (Exception e){
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
    }
}
```

6. Un proxy qui interroge des services

6.1 Présentation



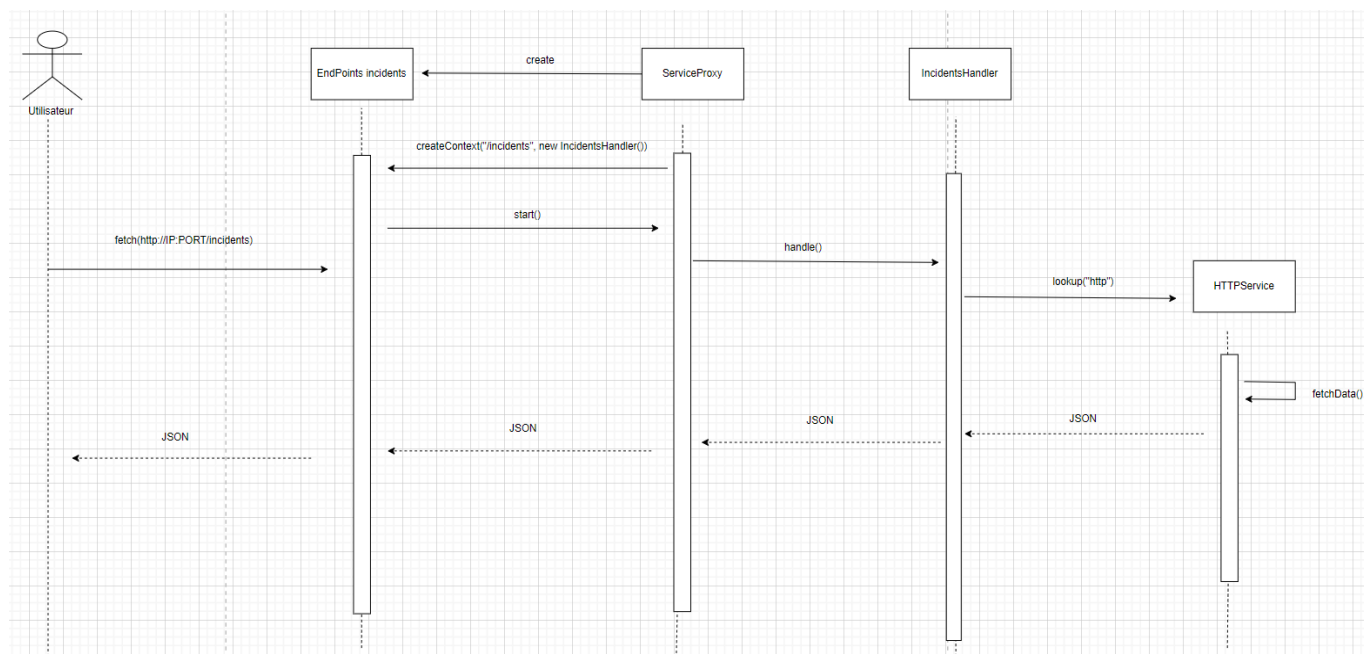
6. Un proxy qui interroge des services

6.1 Présentation

Afin de contourner la politique CORS du navigateur, nous allons donc développer un proxy afin de passer par java (qui sera donc un client côté serveur, et n'auras pas ces problèmes de CORS) pour fetch les données bloquées. L'objectif est le suivant : notre client javascript fetch notre proxy, qui lui à son tour ira fetch l'URL souhaitée. Après avoir obtenu la réponse de l'API tierce, il la rendra à son tour au client, afin de pouvoir afficher les données sur notre carte interactive.

6.2 Proxy

Le fonctionnement du proxy est représenté par ce diagramme de séquence présenté dans la partie 2.1 de ce rapport.



```

import ...

public class LancerProxy {
    // Titoufeur +1 *
    public static void main(String[] args) throws IOException {
        // on crée un serveur HTTP écoutant sur le port 8080
        HttpServer server = HttpServer.create(new InetSocketAddress(Integer.parseInt(args[0])), 0);

        // On crée les endpoints et on associe une classe à chacun d'entre eux
        server.createContext(path: "/restaurants", new RestaurantHandler());
        server.createContext(path: "/fetch", new GenericHandler());

        // et on démarre le serveur
        server.setExecutor(null);
        server.start();
        System.out.println("Server started on port " + args[0]);
    }
}
  
```


La classe `LancerProxy` va lancer le proxy à l'aide de la classe `HTTPServer`. La première ligne va créer un serveur qui va écouter sur le port donné en paramètre (comme 8080). Ensuite, les lignes « `createContext` » définissent des endpoints, qui seront à placer après l'adresse du proxy (exemple : <http://1.1.1.1:8080/endpoint>). On définit donc, par exemple, que lorsqu'un client demande l'endpoint `restaurants`, c'est la classe `restaurantHandler` qui s'occupe de répondre à la requête. Le proxy va créer une instance de cette classe et lancer la méthode « `handle()` » qui doit être définie.

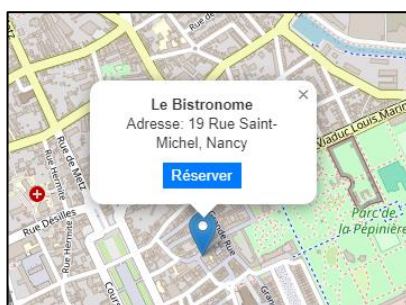
```
class RestaurantHandler implements Handler { 1 usage 1 Titoufleur +1*
    @Override 1 Titoufleur +1*
    public void handle(HttpExchange exchange) throws IOException {
        exchange.getResponseHeaders().add("Access-Control-Allow-Origin", "*"); // Permettre l'accès depuis n'importe quelle origine
        exchange.getResponseHeaders().add("Access-Control-Allow-Methods", "GET, POST, OPTIONS, PUT"); // Méthodes autorisées
        exchange.getResponseHeaders().add("Access-Control-Allow-Headers", "Content-Type, Authorization"); // En-têtes autorisés
        if ("OPTIONS".equalsIgnoreCase(exchange.getRequestMethod())) {
            exchange.sendResponseHeaders(200, -1);
            return;
        }
        //Traitement de la requête GET
        if ("GET".equalsIgnoreCase(exchange.getRequestMethod())) {
            try {
                // on récupère les restaurants depuis le service
                Registry registry = LocateRegistry.getRegistry("localhost", 1099);
                RestaurantService sr = (RestaurantService) registry.lookup("restaurants");
                String response = sr.getAllRestaurants();
                // On renvoie la réponse
                exchange.sendResponseHeaders(200, response.getBytes().length);
                OutputStream os = exchange.getResponseBody();
                os.write(response.getBytes());
                os.close();
            } catch (Exception e) {
                e.printStackTrace();
                exchange.sendResponseHeaders(500, 0);
                exchange.getResponseBody().close();
            }
        }
        //Traitement de la requête POST
        else if ("POST".equalsIgnoreCase(exchange.getRequestMethod())) {

```

Donc dans cette classe `Restaurant`, comme cité auparavant on définit une fonction `handle`, et ici on commence par traiter le cas d'une requête utilisant la méthode `http GET`.

Il existe plusieurs méthodes `http` (dont `GET`, `POST`, `PUT`, `PATCH`) et chacune d'entre elles visent à obtenir des comportements différents. La méthode `GET` sert à obtenir des données, c'est donc ce qu'on fait ici. On récupère l'annuaire, qui nous permet de récupérer le service créé auparavant, afin de s'en servir pour qu'il nous fetch les données des restaurants (qui n'étaient pas des données venant d'une API mais de notre base de données). Une fois ces données obtenues, on les renvoie simplement au client.

Nous avons ensuite implémenté une fonctionnalité visant à réserver une table d'un restaurant depuis la carte. Pour ce faire, nous avons créé, sur chaque bulle d'information de restaurant, un bouton « réserver » qui affiche un formulaire `HTML` :



Réserver pour Le Bistrone

Prénom:
 Nom:
 Nombre de convives:
 Téléphone:

Lorsqu'un utilisateur rentre des informations et clique sur « réserver », ça envoie une requête POST au proxy, qui ouvre un mode transactionnel dans la base de données pour effectuer une réservation d'un restaurant.

6.3 Question

Au fait, est-ce responsable d'ainsi contourner la politique de sécurité de votre navigateur ?

Non cela n'est pas raisonnable de les contourner. D'un point de vue sécurité, l'application peut toujours subir des attaques et expose les données. Également, il a des plus de risques que les données soient incorrectes car modifiées par d'autres personnes. Enfin, d'un point de vue légal, on peut se faire poursuivre en justice car on ne respecte pas les conditions d'utilisations.

7. Travail complémentaire

7.1 Présentation

Après avoir terminé les étapes précédentes, nous avons implémenté quelques fonctionnalités bonus. Premièrement, nous avons ajouté une rubrique sur notre site affichant la météo grâce à des données ouvertes fournies par infoclimat.fr.

Nancyclopédie

Stations de Vélos


Météo

Etablissements du supérieur

Restaurants

Infos trafic

Rapport SAE



Jeu

di 20 Juin 2024 06:00


Température: 13.35 °C

Risque de neige: non

Risque de pluie: 1.5 mm

Iso Zero: 3648 m

Vent: 7 m/s, Rafales: 8.3 m/s



Jeu

di 20 Juin 2024 08:00


Température: 14.55 °C

Risque de neige: non

Risque de pluie: 0 mm

Iso Zero: 3759 m

Vent: 13.3 m/s, Rafales: 28.5 m/s



Jeu

di 20 Juin 2024 11:00

Température: 18.85 °C

Risque de neige: non

Risque de pluie: 2.4 mm

Iso Zero: 3722 m

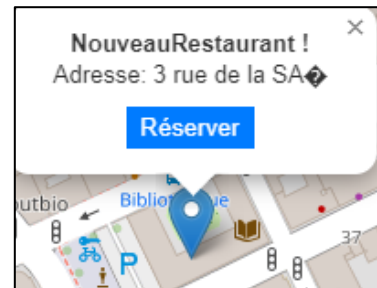
Vent: 14.7 m/s, Rafales: 32.1 m/s

Nous avons également implémenté la fonctionnalité permettant de cliquer sur la carte pour ajouter un nouveau restaurant.

Ajouter un nouveau restaurant

Nom du restaurant:

Adresse:



Ce dernier s'affiche de suite sur la carte, et est enregistré dans la base de données et sera donc visible par les autres utilisateurs. Il est aussi possible d'effectuer des réservations sur ces nouveaux restaurants.

Enfin, nous avons développé un service générique, permettant au client de, s'il rencontre un blocage dans le fetch d'une URL, appeler ce service pour que un programme côté serveur fasse le fetch à sa place et lui renvoie les données. Nous avons d'ailleurs remplacé le serviceHTTP par ce service générique. Premièrement, nous avons développé une fonction dans le code javascript permettant de fetch une URL. Si cela échoue, on effectue un fetch vers notre proxy, à l'endroit « fetch », en passant en paramètre (via le ?) un url, qui sera l'URL à fetch :

```

async function fetchUrl(url) : Promise<...> { Show usages new *
  try {
    const response : Response = await fetch(url);
    return await response.json();
  } catch (error) {
    console.log('Erreur lors d'un fetch. Tentative de passer par le proxy. ' + error);
    try {
      const proxyResponse : Response = await fetch( input: urlProxy + '/fetch?url=' + url, {
        mode: 'cors'
      });
      console.log('Fetch réalisé via le proxy');
      return await proxyResponse.json();
    } catch (proxyError) {
      console.log('Erreur lors du fetch par Proxy. Abandon.' + proxyError);
      return null;
    }
  }
}

```

Le proxy de son côté appelle donc la classe « genericHandler » qui va s'occuper de faire le travail nécessaire : récupérer l'URL à fetch, puis renvoyer le resultat :

```
class GenericHandler implements Handler {
    @Override
    public void handle(HttpExchange exchange) throws IOException {
        exchange.getResponseHeaders().add("Access-Control-Allow-Origin", "*"); // Permette l'accès depuis n'importe quelle origine
        exchange.getResponseHeaders().add("Access-Control-Allow-Methods", "GET, POST, OPTIONS"); // Méthodes autorisées
        exchange.getResponseHeaders().add("Access-Control-Allow-Headers", "Content-Type, Authorization"); // En-têtes autorisés
        if ("GET".equals(exchange.getRequestMethod())) {
            // On regarde si un paramètre a été passé (l'url donc)
            String query = exchange.getRequestURI().getQuery();
            if (query != null && query.startsWith("url=")) {
                String urlString = query.substring(5);
                // Si c'est le cas, on va alors récupérer le service afin de fetch

                try {
                    // On récupère l'annuaire
                    Registry registry = LocateRegistry.getRegistry("localhost", 1099);
                    // On récupère le service
                    GenericService sg = (GenericService) registry.lookup("generic");
                    // On fetch avec l'URL demandé.
                    String response = sg.fetchData(urlString);

                    // Spécifier l'encodage UTF-8 dans l'en-tête de la réponse sinon ça met des caractères point d'interrogation
                    Headers headers = exchange.getResponseHeaders();
                    headers.set("Content-Type", "application/json; charset=UTF-8");

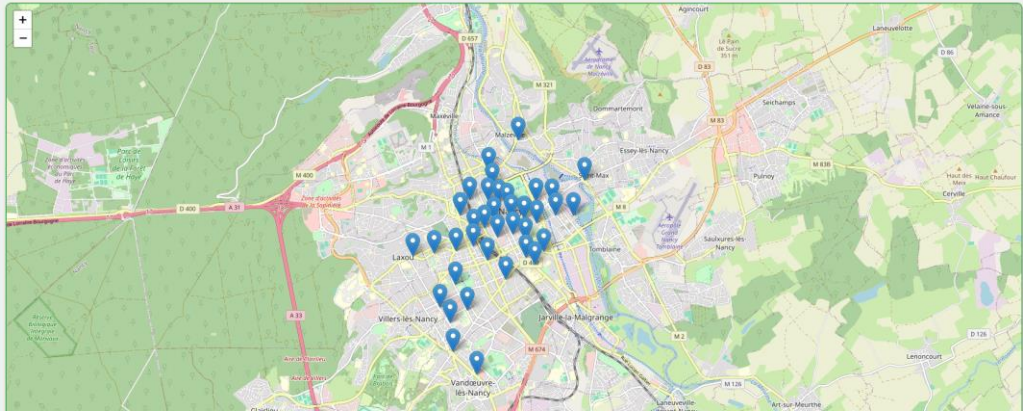
                    byte[] responseBytes = response.getBytes(StandardCharsets.UTF_8);
                    // Envoyer la réponse avec la longueur correcte
                    exchange.sendResponseHeaders(200, responseBytes.length);
                    OutputStream os = exchange.getResponseBody();
                    os.write(responseBytes);
                    os.close();
                } catch (Exception e) {
                    e.printStackTrace();
                    exchange.sendResponseHeaders(500, -1);
                    exchange.getResponseBody().close();
                }
            }
        } else {
            exchange.sendResponseHeaders(405, -1);
            exchange.getResponseBody().close();
        }
    }
}
```

8. Conclusion

8.1 Présentation des résultats obtenus

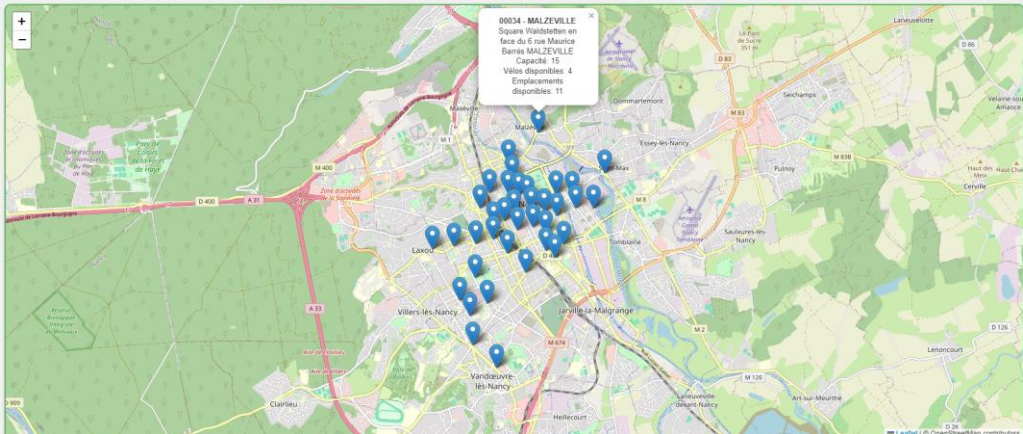
Nancyclopédie Stations de Vélos Météo Etablissements du supérieur Restaurants Infos trafic

Rapport SAE



Nancyclopédie Stations de Vélos Météo Etablissements du supérieur Restaurants Infos trafic

Rapport SAE



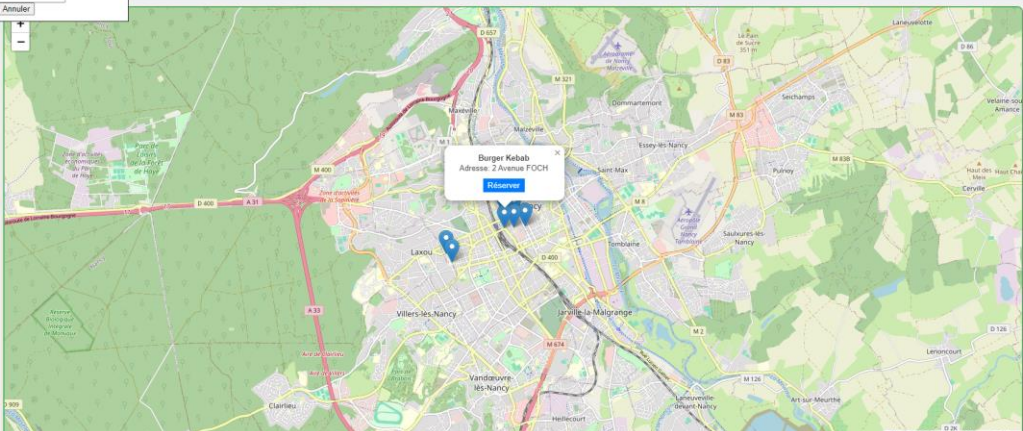
Nancyclopédie Stations de Vélos Météo Etablissements du supérieur Restaurants Infos trafic

Rapport SAE

Ajouter un nouveau restaurant

Nom du restaurant:

Adresse:



Cette SAE a été l'occasion de découvrir une nouvelle pratique en développement web, en découvrant notamment l'existence des CORS. Développer un proxy pour contourner cette restriction était une étape très intéressante, et le fonctionnement de celui-ci reste totalement clair pour nous. Ce fut également une bonne occasion d'utiliser les services RMI, et avec toute cette architecture, nous pouvons en 3 parties différentes obtenir l'accès à des données, qu'une seule partie n'aurait pas pu obtenir. Nous avons également aimé la découverte de tout ce qui est données ouvertes, c'est super intéressant et cela donne envie de développer des outils à partir de ces données, que ce soit à des fins personnelles, ou alors pour partager des outils et aider des personnes.