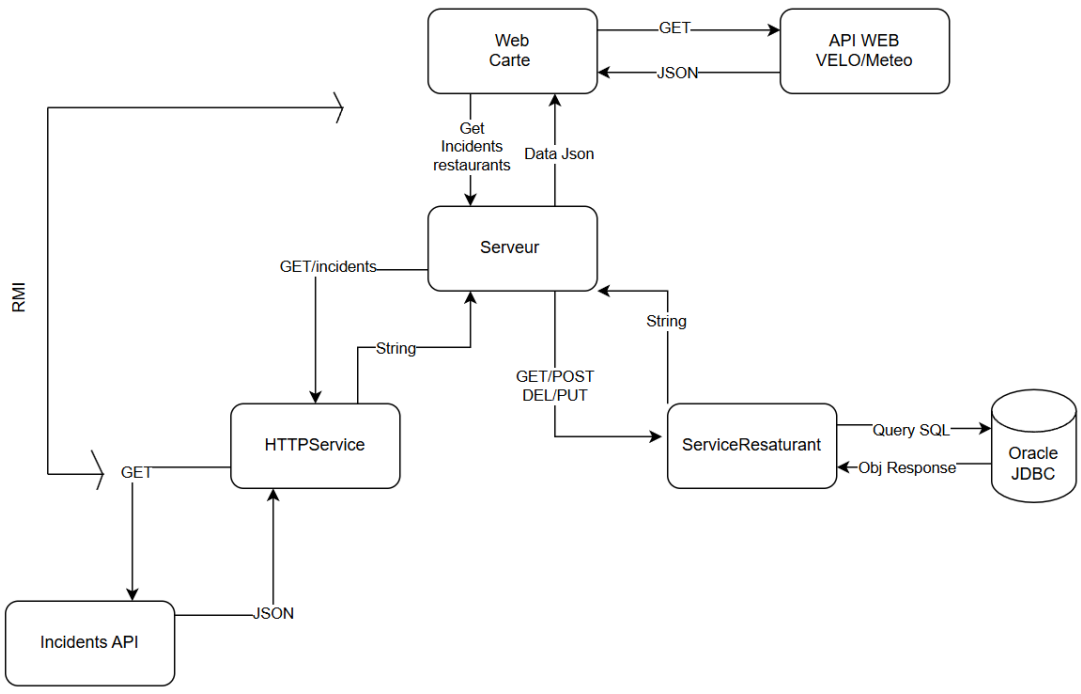
Rapport – SAE S4-IL-02 – Application Répartie

Groupe : ROCHEDREUX Hugo - KLINGLER Emma - FÉNARD Dorian - DROUVOT Melody

Projet : Carte interactive et services distribués

1. Architecture de l'application

- Frontend : site web avec carte Leaflet (JS)
- Serveur HTTPS Java : intermédiation entre le site et les services (via /data et /incidents)
- Service RMI pour les restaurants (information, réservation, annulation)
- Service RMI pour les incidents
- Données ouvertes : stations vélos
- Base de données Oracle (sur Charlemagne)



2. Répartition du travail

Nom	Partie réalisée
Dorian FÉNARD	Carte Leaflet, frontend JS
Emma KLINGLER	Service RMI Restaurant, accès DB Oracle
Hugo ROCHEDREUX	Serveur HTTPS, gestion CORS, handlers
Melody DROUVOT	Service RMI Incidents, proxy

3. Fonctionnalités implémentées

- Carte Leaflet centrée sur Nancy
- Affichage dynamique des stations vélos, incidents, restaurants
- Réservation de tables (nom, prénom, téléphone, heure)
- Annulation de réservation
- Interaction avec base Oracle via RMI
- Proxy HTTP Java (HTTPS ou HTTP)
- Sélecteur dynamique d'adresse IP (non codée en dur)
- Rapport et source en ligne (Git)

4. Lancement du projet

1) Lancer le serveur principal : MainServeur/src/main/java/Main.java
2) Lancer le service pour les incidents : MainServeur/src/main/java/Main.java

3) Lancer le service pour la base de données : DB_restaurant/src/main/java/Main.java

4) S'assurer d'être connecté sur le réseau de l'IUT, ou d'utiliser le VPN correspondant.

5. Liens utiles

Dépôt Git : https://github.com/Mitsouille/SAE_prog_repartie
Webetu : https://webetu.iutnc.univ-lorraine.fr/www/rochedre2u/SAE_projet_repartie/

Partie RMI et base de données pour la gestion

Tout d'abord, nous avons modélisé la base de données autour de quatre entités principales :

- **restaurants** : chaque établissement dispose d'un identifiant automatique, de son nom, de son adresse, de ses coordonnées géographiques, d'un téléphone, d'une catégorie, d'un site web et d'une note moyenne.
- **tables_restaurant** : lien entre un restaurant et ses tables physiques, avec pour chaque table un identifiant automatique, un numéro, une capacité et un indicateur intérieur/extérieur.
- **utilisateurs** : stockage des comptes clients avec e-mail unique, mot de passe hashé et date de création (prévu pour de futurs développements d'authentification).
- **reservations** : chaque réservation enregistre la table choisie, les informations du client (nom, prénom, téléphone), le nombre de convives, l'intervalle horaire, un statut (en_attente, confirmee, annulee), ainsi que les dates de création et d'éventuelle annulation.

Pour automatiser la gestion des clés primaires et tracer les dates d'insertion/modification, nous avons créé une séquence et un trigger par table. Le trigger le plus critique est prevent_overlap sur la table reservations : avant chaque insertion ou mise à jour, il vérifie en base qu'aucune autre réservation non annulée sur la même table ne se recoupe. Si un conflit est détecté, il lève une exception contrôlée (ORA-20001), garantissant qu'il est impossible de bloquer deux fois la même table pour un même créneau, même en cas d'appels concurrentiels depuis plusieurs machines Java.

Côté Java, l'implémentation ServiceRestaurantImpl hérite de UnicastRemoteObject et expose via RMI toutes les opérations :

1. **getTousLesRestaurantsJson** renvoie la liste des restaurants au format JSON.
2. **getTablesParRestaurantJson** renvoie, pour un restaurant donné, ses tables disponibles.
3. **getPlacesDisponiblesJson** calcule le nombre total de places libres sur un créneau, en soustrayant des capacités totales la somme des convives déjà réservés.
4. **getToutesLesReservationsJson** liste toutes les réservations existantes.
5. **reserverTableJson** vérifie d'abord la capacité globale du restaurant, recherche la plus petite table libre pouvant accueillir le groupe (requête SQL NOT EXISTS), puis insère la réservation avec le statut en_attente. La réponse indique la réussite ou l'échec et précise, en cas de succès, l'ID de la table choisie.
6. **annulerReservationJson** ne s'appuie pas sur l'ID interne de la réservation (inaccessible au client) : on supprime la ligne correspondante en filtrant sur le prénom, le nom, le téléphone et la date de début du créneau. La réponse JSON

confirme le nombre de lignes supprimées ou signale qu'aucun enregistrement ne correspondait.

Pour valider l'ensemble, nous avons développé la classe de test TestServiceRestaurant, qui simule un parcours complet :

- récupération initiale des restaurants,
- lecture des tables d'un restaurant,
- calcul de capacité disponible,
- création d'une réservation,
- tentative de doublon pour tester le trigger de chevauchement,
- annulation de la réservation par les critères métier,
- vérification de l'état final des réservations.

Grâce à cette approche, nous garantissons que :

- **la capacité globale** d'un restaurant n'est jamais dépassée,
- **aucune table** ne peut être réservée deux fois pour un même créneau,
- **les groupes** ne sont jamais affectés à des tables trop petites,
- **l'interface JSON** reste simple à consommer pour un futur client HTTP/CORS,
- **les accès concurrents** via RMI sont sécurisés par les triggers Oracle.