ARCHITECTURE DEFINITION DOCUMENT

YOUR CAR YOUR WAY

SOMMAIRE

Objet du document

Ce document permet de modéliser l'architecture de l'application web unifiée Your Car Your Way App qui sera réalisée par les équipes de développement. Au-delà de la modélisation graphique, il s'agit également d'énoncer les principes sur lesquels l'architecture s'appuie, de justifier cette approche architecturale, mais également d'indiquer des transitions si nécessaire.

Objectifs du projet

- Unifier les différentes plateformes clients existantes en une seule application web.
- Offrir une expérience utilisateur fluide, cohérente et personnalisée à l'échelle internationale.
- Permettre une montée en charge grâce à une architecture modulaire.
- Garantir une conformité aux réglementations internationales.
- Permettre une évolutivité et une intégration facile avec les systèmes internes (API).

Principes d'architecture

- **Modularité** : séparation claire des domaines (authentification, réservation, support client)
- **Conformité aux principes S.O.L.I.D et design pattern** : Les modules métiers et techniques sont conçus en respectant les principes **S.O.L.I.D**, afin de garantir un code maintenable, testable et évolutif.
  L'utilisation de **design patterns** adaptés (Factory, Strategy, MVC, etc.) permet d'assurer une structure claire et réutilisable à long terme.
- **API First** : tous les services sont exposés via une API REST sécurisée avec JWT.
- **Sécurité by design** :
  - Gestion centralisée de l'authentification et chiffrement des données sensible.

    - Validation des entrées et protections contre les attaques courantes (CSRF, XSS..)
- **Responsiveness** : interface utilisateur adaptée à tous les devices.
- **Qualité :** Test unitaire + intégration + E2E + CI/CD permettant d'assurer un moindre risque de régression et une qualité de livraison du projet.

Architecture existante

Le projet ne s'appuie pas sur un produit existant qui serait repris puis mis à jour. Il n'y a donc pas d'architecture existante à définir.

Architectures

Les 3 sections ci-dessous permettent de modéliser l'architecture selon des angles de vue différents : métier, données, technologique. La vue métier correspond à la description du produit d'un point de vue fonctionnel.
La vue de données correspond à la description de l'architecture de données – cette modélisation va jusqu'à la formalisation du schéma de la base de données relationnelle si le projet mobilise ce type de base. La vue technologique correspond à la description des composants logiciels, de leurs interactions, de leurs infrastructures.

Métier

Ce diagramme décrit les interactions fonctionnelles entre les acteurs du système et les fonctionnalités proposées par l'application Your Car Your Way.

Les Acteurs :

Client

Représente l'utilisateur final de l'application (client de l'entreprise), qui interagit avec l'interface web pour réserver un véhicule, gérer son compte et contacter le support.

Service support

Représente le personnel du service client chargé de répondre aux demandes des clients via différents canaux (chat, visio, message).

Agent

Représente le personnel en agence ou le back-office, qui accède à une interface technique et a besoin d'une API pour effectuer des opérations CRUD métier.

De données

Ce schéma relationnel modélise les principales entités de l'application Your Car Your Way ainsi que leurs relations. Il s'inscrit dans une architecture orientée domaines, en cohérence avec la séparation fonctionnelle décrite dans la vue métier. Les entités sont organisées autour de trois axes principaux : gestion utilisateur, réservation de véhicules, et support client.

Gestion utilisateur

La table **users** représente l'ensemble des comptes utilisateurs de la plateforme. Chaque utilisateur dispose d'un identifiant, d'une adresse email, d'un mot de passe, et d'un rôle. Le rôle est un champ ENUM permettant de distinguer les clients, les agents en agence, et les membres du support. Cette table centralise l'authentification et l'autorisation.

La table **profiles** stocke les informations personnelles des clients : nom, prénom, date de naissance et adresse. Elle est liée à la table users via une clé étrangère user_id, qui est également sa clé primaire. Ce lien direct permet d'assurer qu'un profil n'existe que pour un utilisateur identifié comme client.

La table **support_infos** est dédiée aux utilisateurs de type support. Elle contient le prénom et le nom du membre du support, liés à l'utilisateur via user_id (clé primaire et clé étrangère). Cela permet de séparer clairement les données spécifiques à chaque type d'utilisateur tout en gardant une structure centralisée.

Les **adresses** sont stockées dans la table **addresses**, qui contient les informations nécessaires à la localisation (numéro et nom de rue, pays, canton, code postal). Ces adresses peuvent être associées à des profils clients ou à des agences.

Réservation véhicule

La table **vehicles** définit les types de véhicules disponibles à la location, avec un champ acriss_code permettant de les classifier selon la norme ACRISS. Les colonnes complémentaires comme la catégorie, la transmission, le type de carburant, la climatisation, et une image permettent d'enrichir la description de l'offre.

Les **agences** de location sont enregistrées dans la table **agencies**, avec leur nom et l'identifiant d'adresse. Cela permet de gérer les différents points de départ ou de retour des réservations.

Les **réservations** sont modélisées dans la table **rentals**. Une réservation comprend un client (via son profil), une date et une heure de début et de fin, une agence de départ, une agence d'arrivée, un type de véhicule, ainsi qu'un prix. Les agences sont référencées deux fois (départ et arrivée) via des clés étrangères, ce qui reflète le fonctionnement des réservations inter-agences.

Support client

Le support client est géré via plusieurs tables.

La table **support_requests** enregistre chaque demande formulée par un utilisateur : sujet, contenu, date d'envoi, statut. Elle est liée à la table users.

Les tables **messages**, **conversations** et **conversation_participants** permettent de modéliser un système conversationnel.

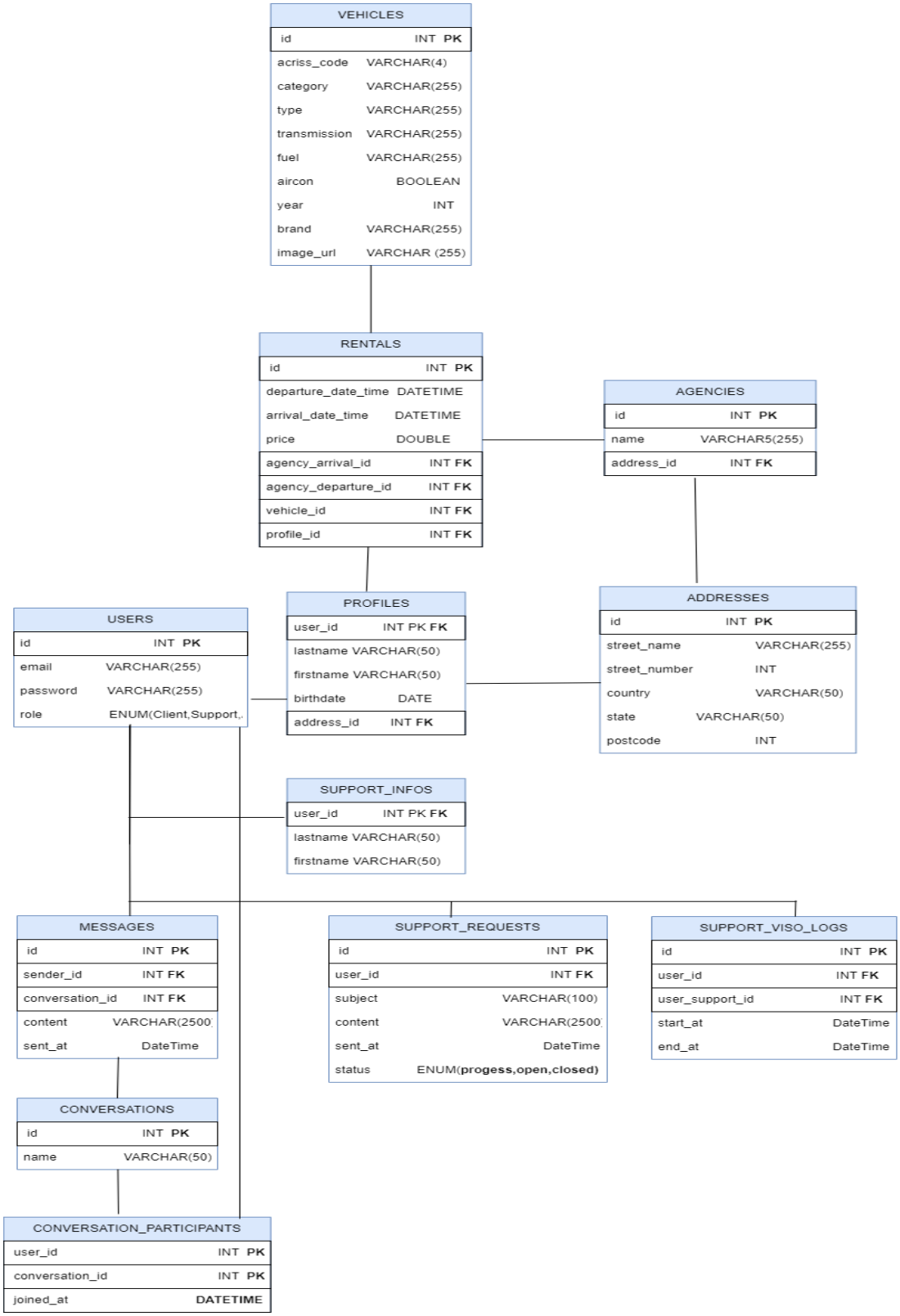
- **conversations**: représente une discussion
- **conversation_participants** : lien entre utilisateurs et conversations (support ou client)
- **messages**: stocke chaque message envoyé, avec expéditeur, contenu, date

Ce découpage respecte le **principe de séparation des responsabilités** et permet :

- Le support multiclient ou multiagents dans une même conversation
- Une future extension vers des salons ou discussions de groupe

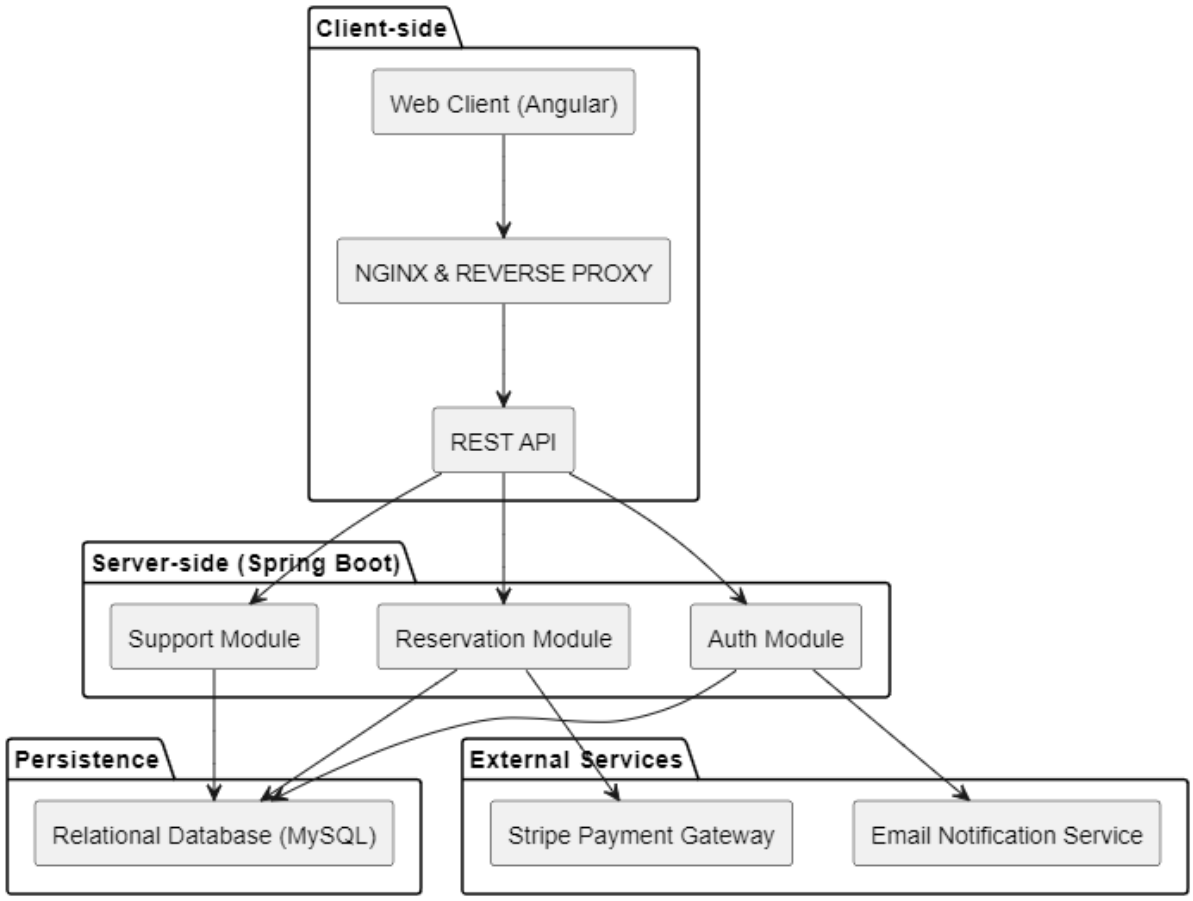
Enfin, la table **support_viso_logs** enregistre les sessions de visioconférence entre un client et un agent support, avec les horodatages de début et de fin. Cette table permet de tracer les échanges synchrones sans stocker de données multimédia.

Ce modèle relationnel offre une base solide pour le développement de l'application. Il isole les responsabilités, respecte les bonnes pratiques de conception de base de données, et permet une extension future des fonctionnalités tout en garantissant la cohérence et l'intégrité des données.



Technologiques

Diagramme de composant et deploiement



Ce diagramme de composants présente l'architecture logicielle de l’application en distinguant la partie **client-side** (frontend) de la partie **server-side** (backend), ainsi que les systèmes externes.

Le frontend repose sur une application **Angular** exécutée dans le navigateur du client. Ce client communique avec un **serveur NGINX faisant office de reverse proxy**, qui redirige les requêtes vers une **API REST unique** exposée par le backend.

Le backend est développé avec **Spring Boot** et est structuré en plusieurs **modules fonctionnels indépendants** :

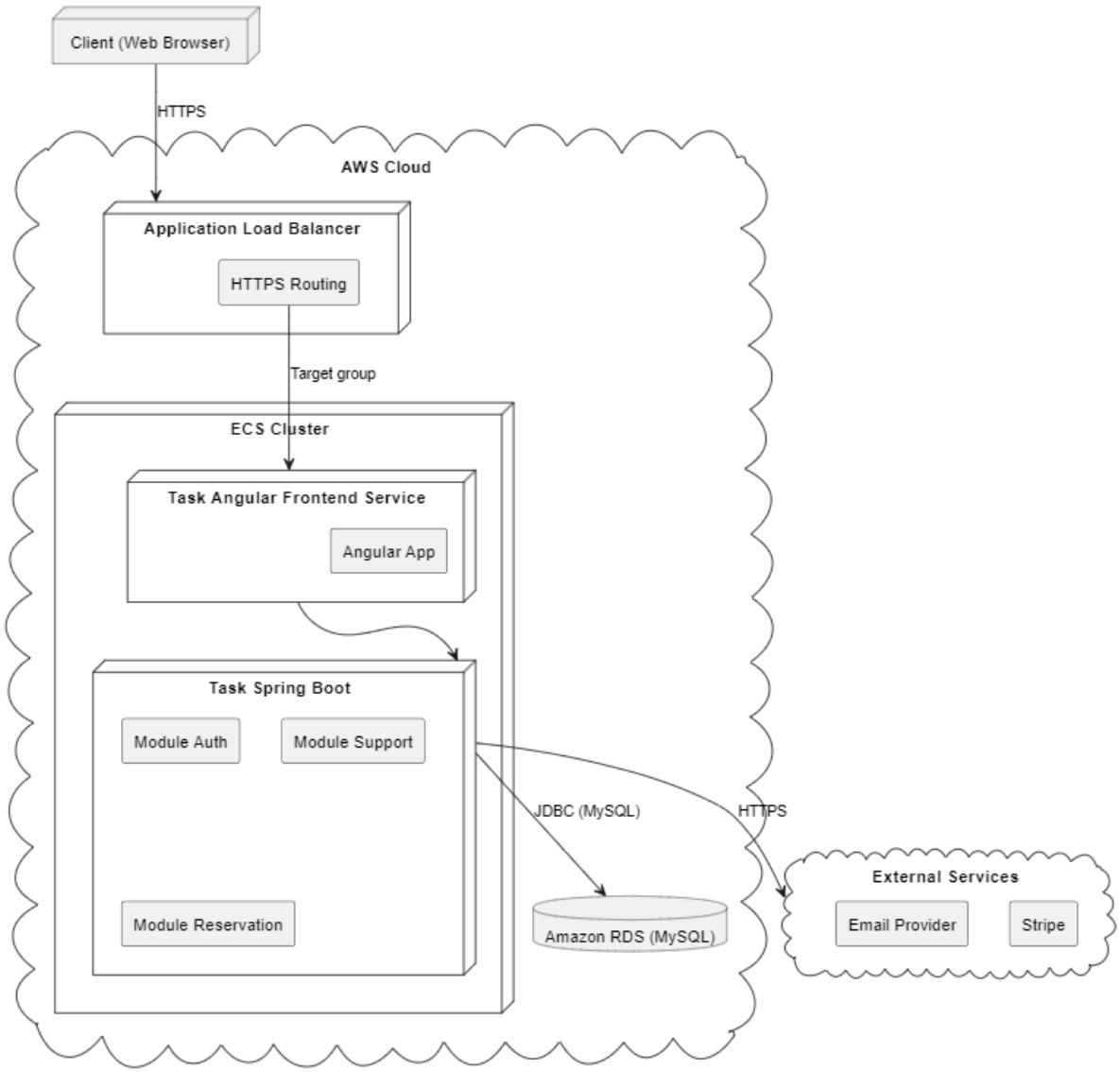
- Le **module d’authentification** gère l’inscription, la connexion, et la gestion des utilisateurs.
- Le **module de réservation** gère la recherche, la réservation et la modification des locations de voitures.
- Le **module de support** gère la communication entre les clients et le service d’assistance, y compris le chat et la visio.

Chaque module peut accéder à :

- Une base de données **relationnelle (MySQL)** pour la persistance.
- Des **services externes** spécifiques à leurs responsabilités :

- La **passerelle de paiement Stripe** pour les réservations.
- Le **service de notification par email** pour les confirmations ou réinitialisations.

Ce découpage modulaire permet une meilleure maintenabilité, une possible scalabilité indépendante de chaque module, et une adaptation facile vers une architecture microservices si nécessaire.



L'architecture repose sur le déploiement de l'application via **Amazon ECS** (Elastic Container Service) en mode EC2. Un cluster ECS est provisionné avec un ou plusieurs nœuds EC2. Les différents services de l'application (frontend et backend) sont conditionnés sous forme de conteneurs Docker, puis déployés dans des tâches ECS gérées automatiquement par le cluster.

L’accès au frontend Angular se fait à travers un **Application Load Balancer (ALB)** configuré pour router le trafic HTTPS vers la tâche ECS correspondante. Le **navigateur du client** envoie donc ses requêtes vers le Load Balancer, qui les redirige dynamiquement vers le conteneur Angular. Ce dernier communique ensuite directement avec les différents services backend spécialisés :

- Le service d’**authentification** (Spring Boot Auth)
- Le service de **réservation**
- Le service de **support client**

Tous les services accèdent à une base de données **Amazon RDS (MySQL)** pour la persistance des données. Ils interagissent également avec des services tiers : **Stripe** pour le paiement, et un **fournisseur d’email** pour l’envoi de notifications.

Les tâches ECS peuvent être automatiquement réparties entre les différentes instances EC2 du cluster. Il n’y a pas une instance EC2 par service : ECS planifie la répartition selon les ressources disponibles (RAM, CPU), garantissant une gestion efficace et évolutive des ressources.

Communication en temps réel dans l'application de support

L'application intègre deux mécanismes de communication en temps réel pour répondre aux besoins du support client : **le chat** et **la visioconférence**.

Chat : WebSocket + STOMP + RxStomp

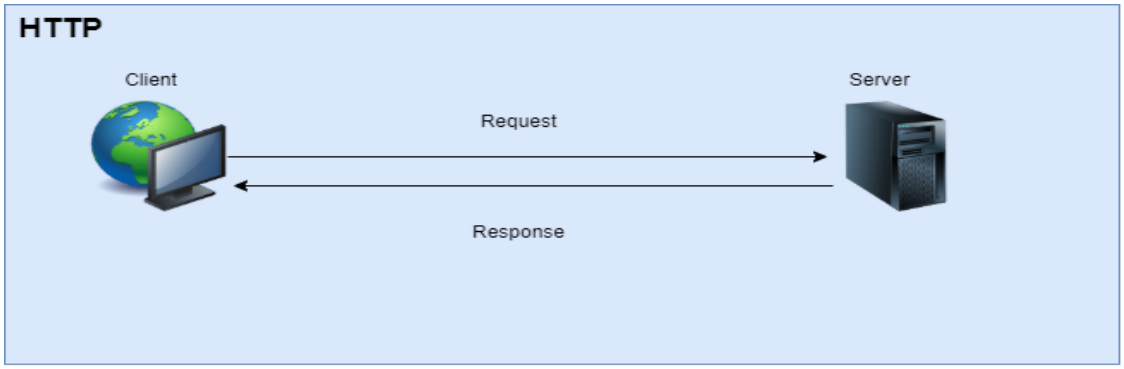
Pour permettre une **communication instantanée** entre un utilisateur et un agent du support, l'application utilise **WebSocket**, un protocole permettant une communication **bidirectionnelle** persistante entre le client et le serveur, contrairement à HTTP où chaque requête est indépendante et unidirectionnelle.

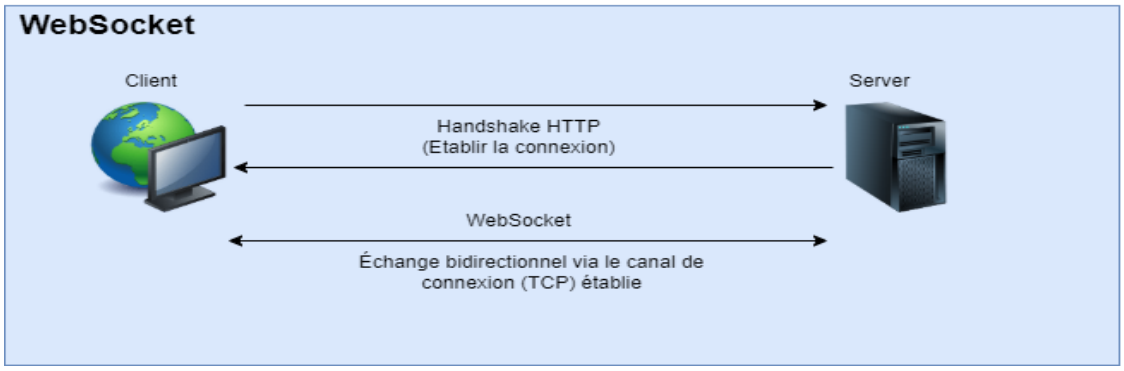
Afin de structurer les échanges WebSocket, nous utilisons le protocole **STOMP (Simple Text Oriented Messaging Protocol)**. Ce protocole de plus haut niveau apporte des fonctionnalités essentielles telles que :

- des **destinations** (topics),
- des **en-têtes** pour enrichir les messages,
- une gestion claire des **abonnements** et **envois**.

Pourquoi RxStomp ? Nous utilisons **RxStomp** côté Angular car il s'intègre parfaitement avec **RxJS** (programmation réactive), ce qui facilite la gestion des flux asynchrones de messages, des abonnements et des erreurs.

Schéma HTTP vs WebSocket





Visioconférence : WebRTC + STUN + WebSocket (Signaling)

Pour les échanges vidéo et audio en temps réel, l'application s'appuie sur **WebRTC**, une technologie native des navigateurs permettant une connexion directe **peer-to-peer (P2P)** entre les utilisateurs.

Fonctionnement :

1. **Signaling via WebSocket** : Avant d'établir la connexion P2P, les navigateurs doivent échanger des informations (SDP, ICE candidates). Cette phase, appelée signaling, est assurée par notre backend avec **WebSocket + STOMP**, de la même manière que pour le chat.
2. **STUN** : Chaque client contacte un serveur **STUN** pour obtenir son IP publique (ICE candidate). Si la connexion directe échoue (firewall, NAT...), un serveur

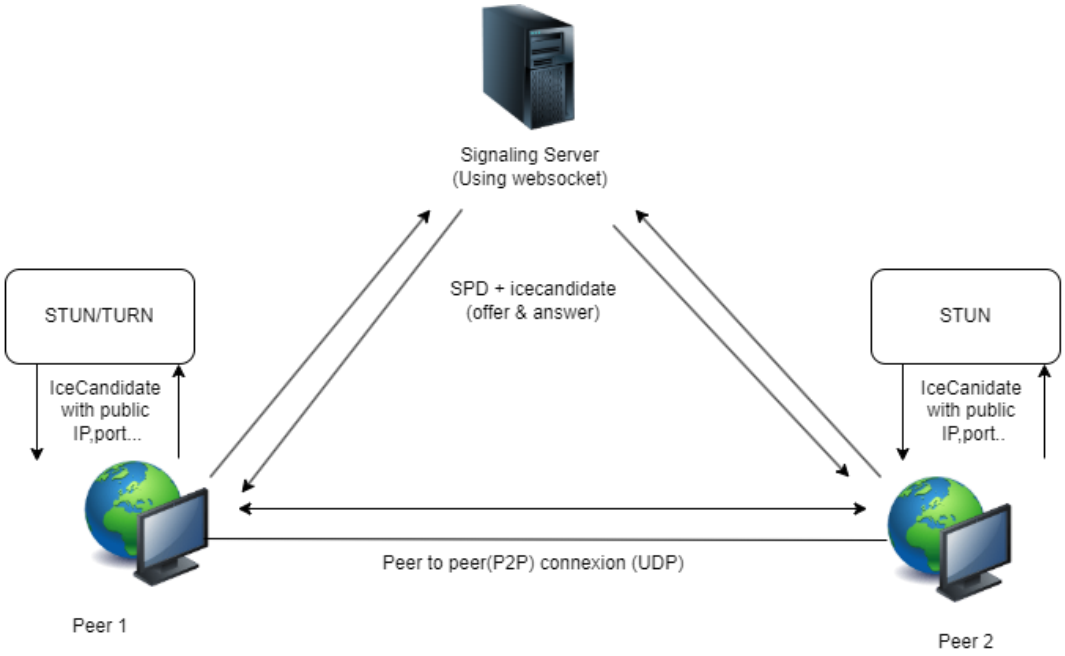
TURN peut être utilisé pour relayer le flux, même si cela n'est plus totalement P2P.

3. **Établissement de la connexion** : Une fois les échanges terminés, si une route directe est possible, une connexion **UDP P2P** est établie entre les deux pairs.

Avantages de cette architecture

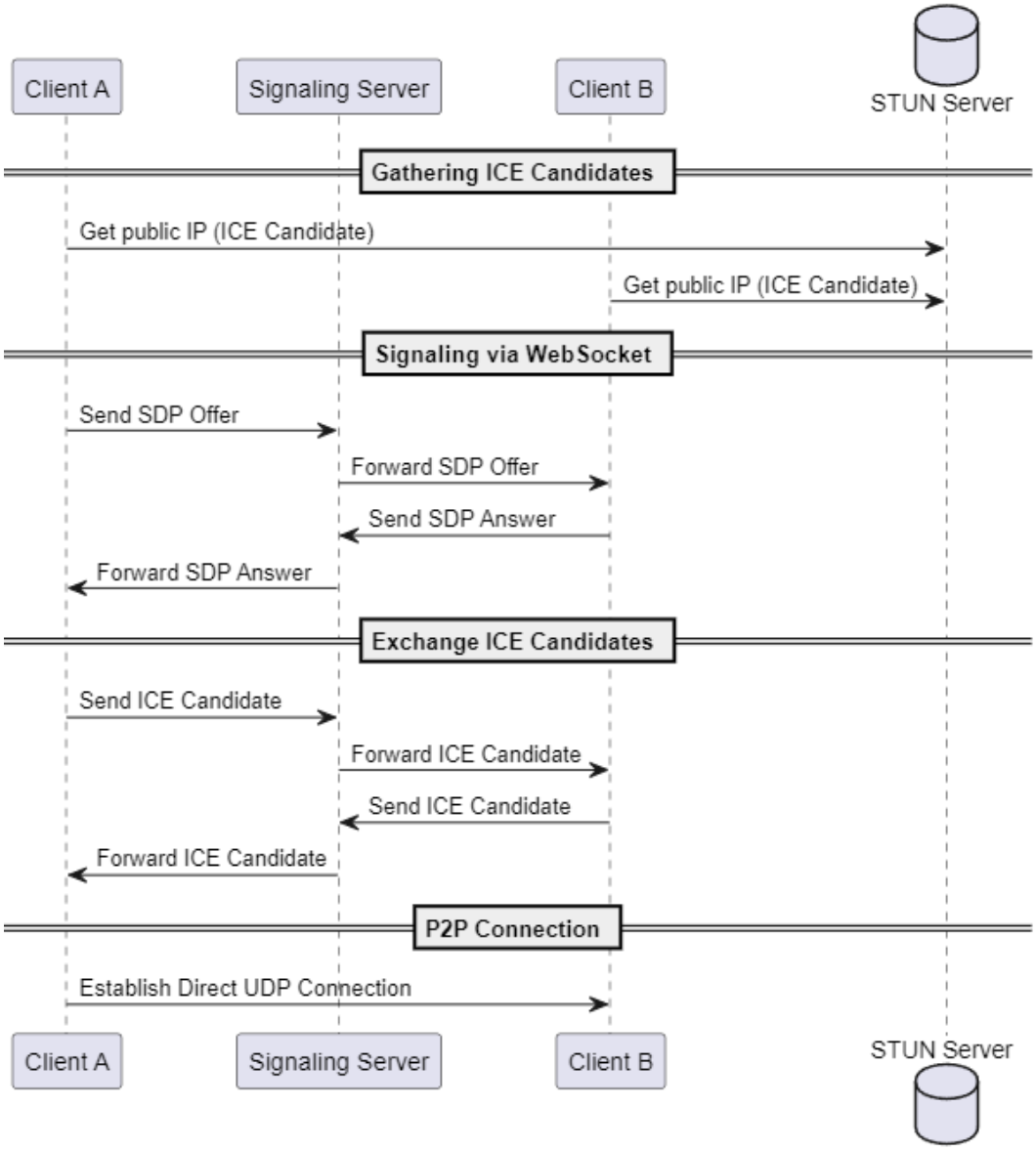
- Communication **bidirectionnelle et en temps réel**.
- Intégration naturelle avec **RxJS** via **RxStomp**, favorisant une architecture réactive et maintenable.
- Utilisation de **STUN/TURN** pour s'adapter à différents environnements réseau.

Schéma



1. Chaque peer contacte son serveur STUN → obtient ses ICE candidates (IP publique, port.. permet d'identifier le peer)
2. Les ICE et les SDPs sont envoyés à l'autre pair via le serveur de signaling
3. Si une route est trouvée, une connexion P2P (UDP) est établie

Flux entre deux clients pour établir le P2P.



Justification de l'approche architecturale

L'architecture proposée repose sur une structure modulaire et évolutive, qui répond aux objectifs de modernisation, de centralisation et de déploiement international de la plateforme Your Car Your Way. Elle respecte les **principes d'architecture modulaire** en isolant les responsabilités dans des services distincts (authentification, réservation, support), ce qui garantit une meilleure maintenabilité et un déploiement indépendant en cas de montée en charge ou de corrections ciblées.

L'utilisation d'**Amazon ECS avec des conteneurs Docker** permet une gestion fine des ressources, un **déploiement automatisé** et une **scalabilité horizontale** des

services. L’intégration d’un **load balancer** en frontal assure la haute disponibilité, la redondance et une répartition intelligente du trafic. La séparation claire entre les couches client et serveur respecte les standards MVC (Model View Controller), tout en facilitant l’évolution vers une architecture microservices complète.

L’architecture suit également les **principes S.O.L.I.D.**, favorise le découplage via des interfaces REST, et applique le paradigme **API First**, assurant l’interopérabilité entre les services internes et les applications tierces. Elle est compatible avec les **bonnes pratiques DevOps** : conteneurisation, CI/CD, déploiement via des environnements cloud managés.

Enfin, des mesures de **sécurité by design** sont intégrées dès la conception : chiffrement des communications, gestion centralisée des accès, protection des données sensibles, et séparation des environnements.