

SOMMAIRE

[.. Objet du document 1](#_Toc175997214)

[.. Objectifs du projet 1](#_Toc175997215)

[.. Principes d’architecture 2](#_Toc175997216)

[.. Architecture existante 3](#_Toc175997217)

[.. Architectures 3](#_Toc175997218)

[De données 8](#_Toc175997219)

[Technologiques 10](#_Toc175997220)

[Choix de la Stack Technologique pour les WebSockets 20](#_Toc175997221)

[.. Justification de l’approche architecturale 20](#_Toc175997222)

[.. Architectures de transition 23](#_Toc175997223)

# .. Objet du document

Ce document permet de modéliser l’architecture 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

Les objectifs du projet *Your Car Your Way* sont les suivants :

1. **Unification et standardisation** : L'objectif est de centraliser les fonctionnalités de réservation de voitures dans une seule plateforme et de standardiser les processus pour garantir une cohérence et une uniformité à l'échelle internationale.
2. **Amélioration de l'Expérience Client** : L'objectif est d'améliorer l'expérience des clients en fournissant une application conviviale, intuitive et réactive, facilitant ainsi la réservation de voitures et offrant un service client de qualité.
3. **Adaptabilité et Progression** : L'objectif est de concevoir une architecture technique capable de supporter une croissance rapide du nombre d'utilisateurs et de réservations, ainsi que l'ajout de nouvelles fonctionnalités et de nouveaux services à l'avenir.
4. **Sécurisation et Conformité** : L'objectif est de garantir la sécurité des données des utilisateurs, en mettant en place des mesures de sécurité avancées telles que le cryptage des données, l'authentification à deux facteurs, et en assurant la conformité aux réglementations en matière de protection des données personnelles.
5. **Déploiement Rapide** : L'objectif est de déployer rapidement la nouvelle application *Your Car Your Way*, en minimisant les temps d'arrêt et en assurant une transition fluide depuis les anciennes plateformes vers la nouvelle.

# .. Principes d’architecture

Les principes d'architecture qui serviront de socle à l'architecture de la nouvelle application Your Car Your Way sont les suivants :

1. **S.O.L.I.D.**
   * **Single Responsibility Principle (Responsabilité Unique)** : Chaque classe ou module ne doit avoir qu'une seule responsabilité.
   * **Open/Closed Principle (Ouvert/Fermé)** : Les entités logicielles doivent être ouvertes à l'extension mais fermées à la modification.
   * **Liskov Substitution Principle (Substitution de Liskov)** : Les objets d'un programme doivent pouvoir être remplacés par des instances de leurs sous-classes sans altérer le bon fonctionnement du programme.
   * **Interface Segregation Principle (Ségrégation des Interfaces)** : Les interfaces spécifiques sont préférées aux interfaces générales.
   * **Dependency Inversion Principle (Inversion des Dépendances)** : Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, les deux devraient dépendre d'abstractions.
2. **Sécurité**
   * Mise en place de mesures de sécurité avancées telles que le cryptage des données, l'authentification à deux facteurs, et la gestion des autorisations.
   * Conformité aux réglementations en matière de protection des données personnelles (par exemple, RGPD).
3. **Cohérence**
   * Utilisation de normes et de pratiques de développement cohérentes à travers l'ensemble du projet.
   * Uniformité dans les interfaces utilisateur et les interactions pour offrir une expérience utilisateur homogène.
4. **Évolutivité**
   * Conception d'une architecture modulaire permettant d'ajouter facilement de nouvelles fonctionnalités sans perturber les systèmes existants.
   * Utilisation de technologies et de pratiques permettant de gérer efficacement une croissance rapide du nombre d'utilisateurs et de transactions.
5. **Testabilité**
   * Facilitation de la mise en place de tests unitaires, d'intégration et de bout en bout pour assurer la qualité du code.
   * Mise en place de pipelines CI/CD intégrant des étapes de tests automatiques.
6. **Performance**
   * Optimisation des performances de l'application pour garantir des temps de réponse rapides et une utilisation efficace des ressources.
   * Surveillance continue des performances et mise en place de mécanismes de mise à l'échelle automatique pour gérer les pics de charge.

# .. 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

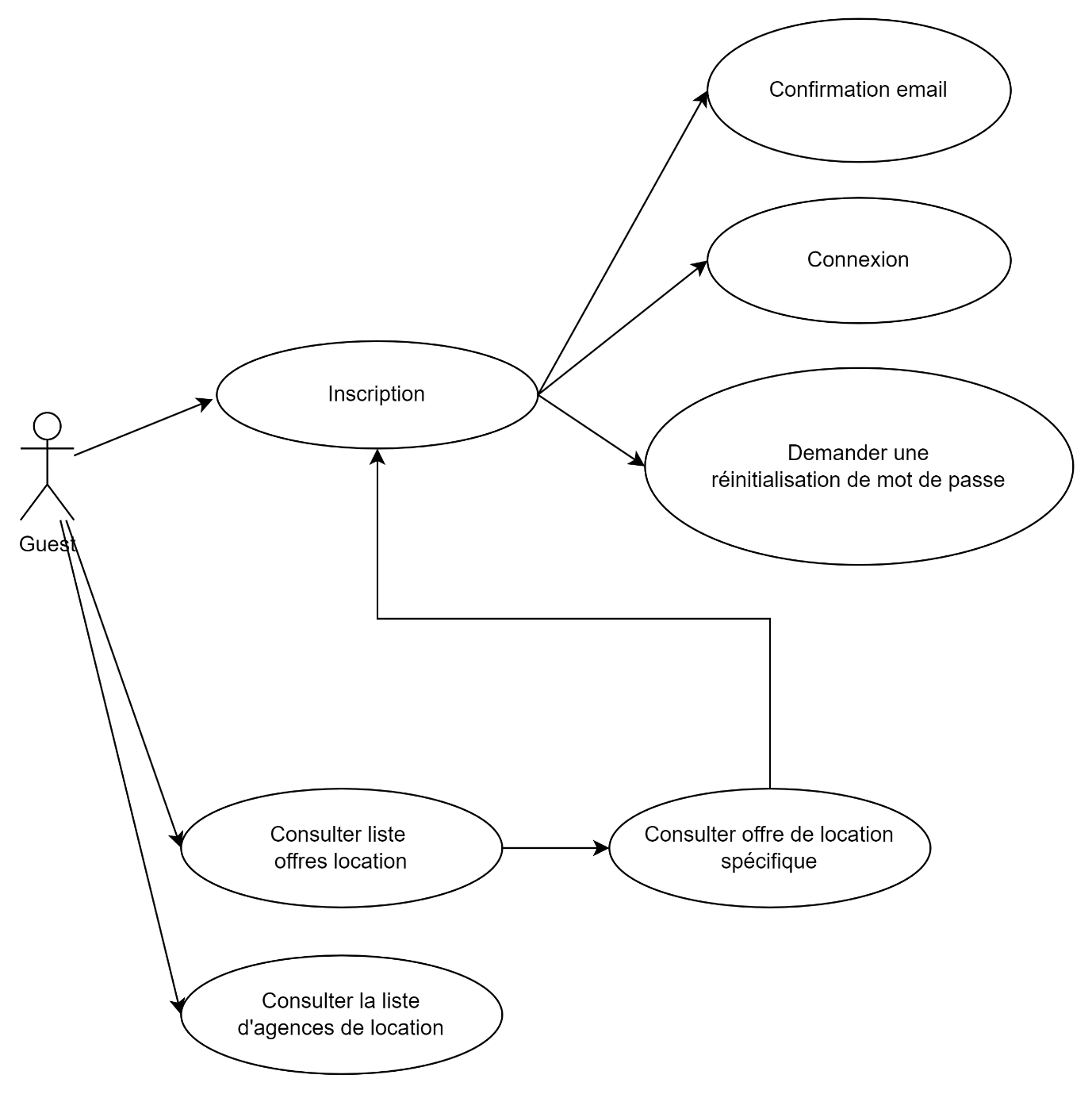
Les 3 sections ci-dessous permettent de modéliser l’architecture selon des angles de vue différents : métier, données, technologique.

Métier

La vue métier correspond à la description du produit d’un point de vue fonctionnel.

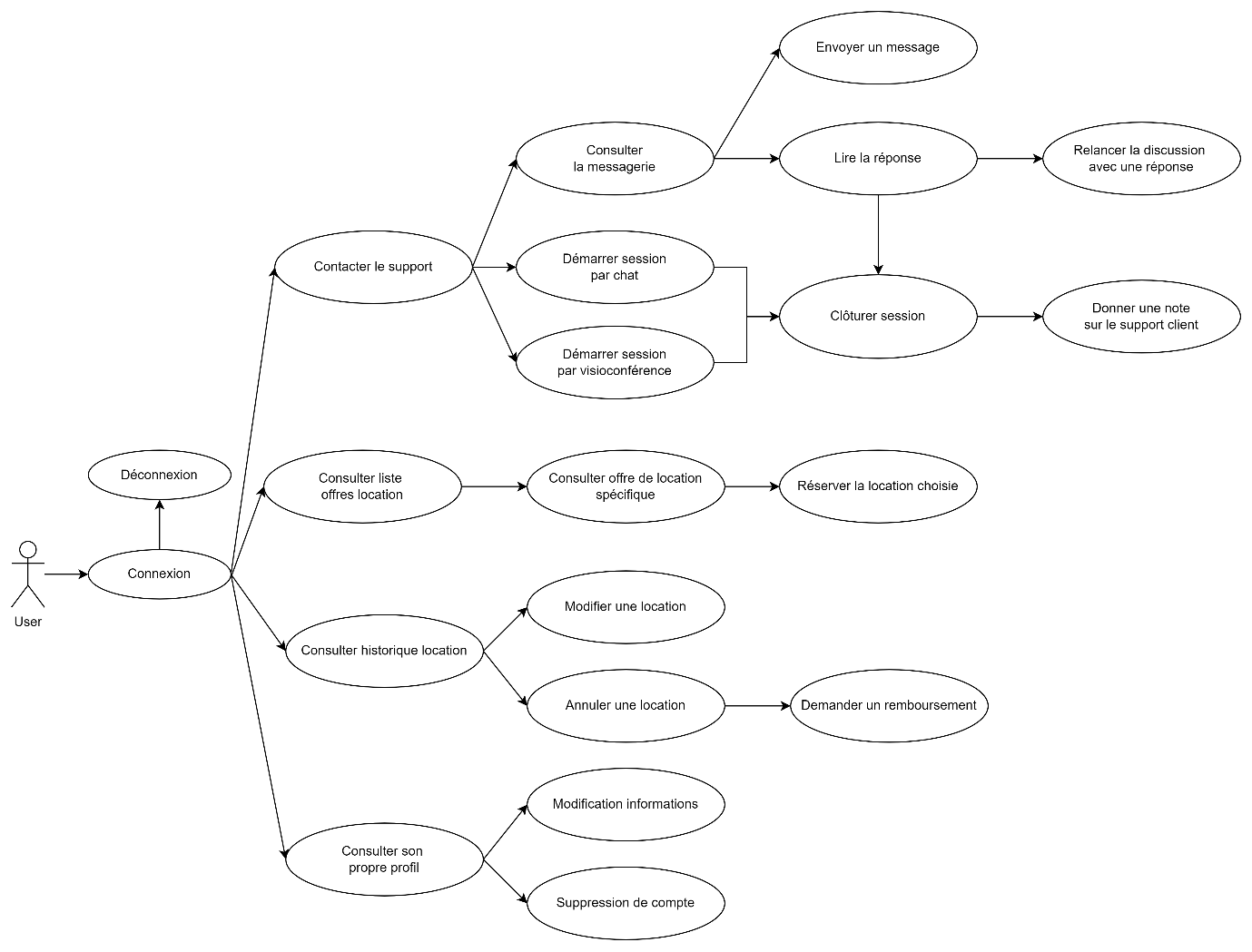
Elle se concentre sur les fonctionnalités offertes aux différents utilisateurs (utilisateurs non connectés, utilisateurs connectés, et employés) et décrit comment ces fonctionnalités interagissent pour satisfaire les besoins métier de l'application.

* **Diagramme UML pour Utilisateur Non Connecté**
  + **Inscription :** Permettre aux utilisateurs de s'inscrire sur la plateforme pour ensuite pouvoir louer un véhicule.
  + **Confirmation email :** Vérification de l'inscription via un lien de confirmation.
  + **Consulter liste d'offres de location :** Visualisation des différentes offres disponibles sur la plateforme.
  + **Consulter agence de location :** Recherche et consultation des agences de location partenaires.

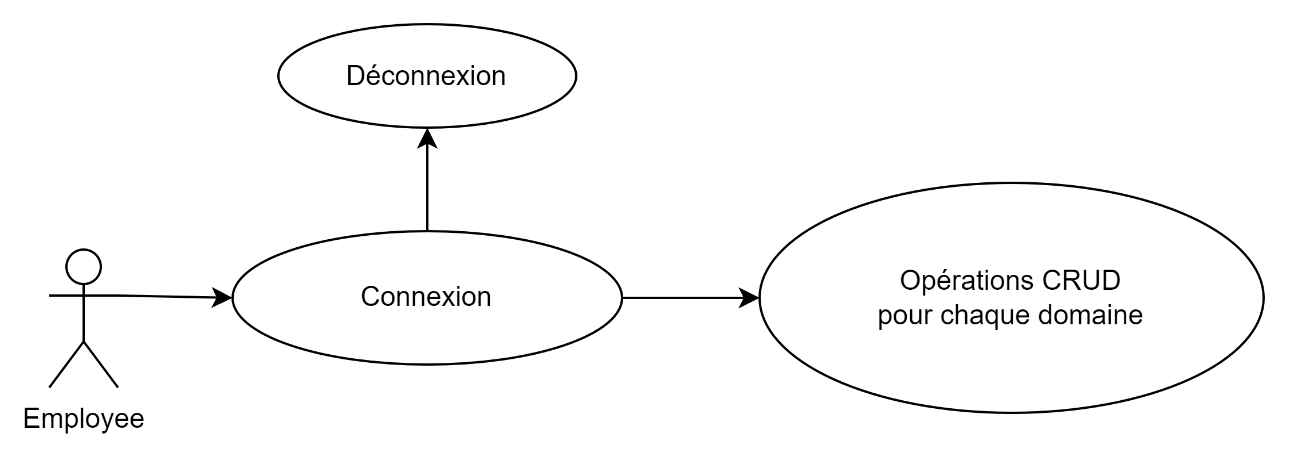


* **Diagramme UML pour Utilisateur Connecté**
  + **Connexion :** Authentification de l'utilisateur pour accéder à son espace personnel.
  + **Déconnexion :** Permet à l'utilisateur de se déconnecter de son compte.
  + **Consulter son propre profile :** Permet à l'utilisateur de consulter et de modifier ses informations personnelles telles que son adresse, numéro de téléphone, etc. Il peut également supprimer son compte.
  + **Consulter offre de location :** Permet à l'utilisateur de visualiser les offres de location disponibles sur la plateforme, de consulter les détails d'une offre spécifique et de procéder à une réservation.
  + **Consulter historique de location :** Permet à l'utilisateur de voir ses locations passées, de les modifier ou d'annuler une réservation. En cas d'annulation, l'utilisateur peut demander un remboursement.
  + **Contact support** Permet à l'utilisateur de contacter le support client de deux manières:
  + **Messagerie asynchrone** : Permet à l'utilisateur d'envoyer un message, de recevoir et lire la réponse du support. Il peut relancer la discussion ou la clôturer.
  + **Support synchrone** : Comprend une session chat et une session visioconférence pour une assistance en temps réel, le client peut aussi la clôturer.

À la suite de la clôture d'une session de support, qu'elle soit asynchrone ou synchrone. L'utilisateur est alors invité à évaluer la qualité du support reçu en attribuant une note appropriée.



* **Diagramme UML pour Employé d'agence**
  + **Connexion :** Authentification de l'utilisateur pour accéder au tableau de bord administrateur.
  + **Déconnexion :** Permet à l'employé de se déconnecter de son compte.
  + **Accès endpoints API d'agence :** Donne accès sécurisé via une API permettant aux applications utilisées en agence de consulter et de modifier les données des clients selon les opérations CRUD standard

****

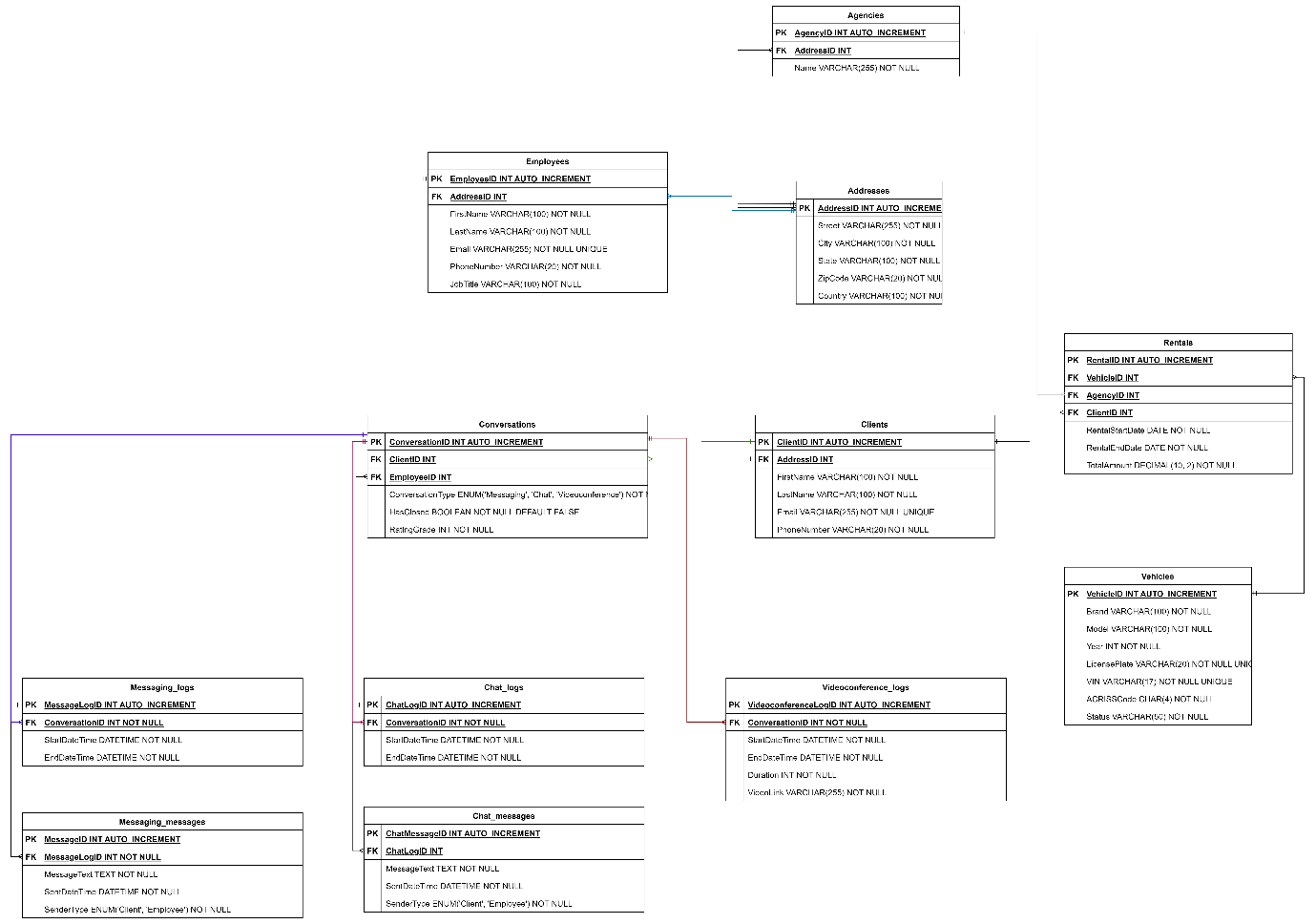
## De données

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.

Le schéma de la base de données comprend plusieurs entités principales :

1. **Addresses** : Contient les adresses avec des détails tels que la rue, la ville, l'état, le code postal et le pays.
2. **Clients** : Représente les clients avec des informations comme le prénom, le nom, l'e-mail, le numéro de téléphone et une clé étrangère vers Addresses pour l'adresse du client.
3. **Employees** : Définit les employés avec des attributs tels que le prénom, le nom, l'e-mail, le numéro de téléphone, le titre du poste et une clé étrangère vers Addresses pour leur adresse.
4. **Vehicles** : Contient des détails sur les véhicules tels que la marque, le modèle, l'année, la plaque d'immatriculation, le numéro de série (VIN), le code ACRISS et le statut du véhicule.
5. **Agencies** : Représente les agences avec des informations comme le nom et une clé étrangère vers Addresses pour l'adresse de l'agence.
6. **Rentals** : Associe les locations avec les clients, les agences et les véhicules, enregistrant les dates de début et de fin de la location ainsi que le montant total.
7. **Conversations** : Permet de suivre les conversations entre les clients et les employés, incluant le type de conversation (messagerie, chat, visioconférence), l'état de clôture et une note facultative donnée par le client.
8. **Messaging\_logs**, **Messaging\_messages**, **Chat\_logs**, **Chat\_messages**, **Videoconference\_logs** : Ces tables détaillent les logs et les messages échangés dans les différentes formes de communication (messagerie, chat, visioconférence) associées aux types de conversations.

Chaque table est liée par des clés primaires et des clés étrangères pour maintenir l'intégrité référentielle des données. Ce modèle supporte les opérations CRUD (Create, Read, Update, Delete) nécessaires pour gérer les informations clients, les locations de véhicules, et les communications entre utilisateurs et support.



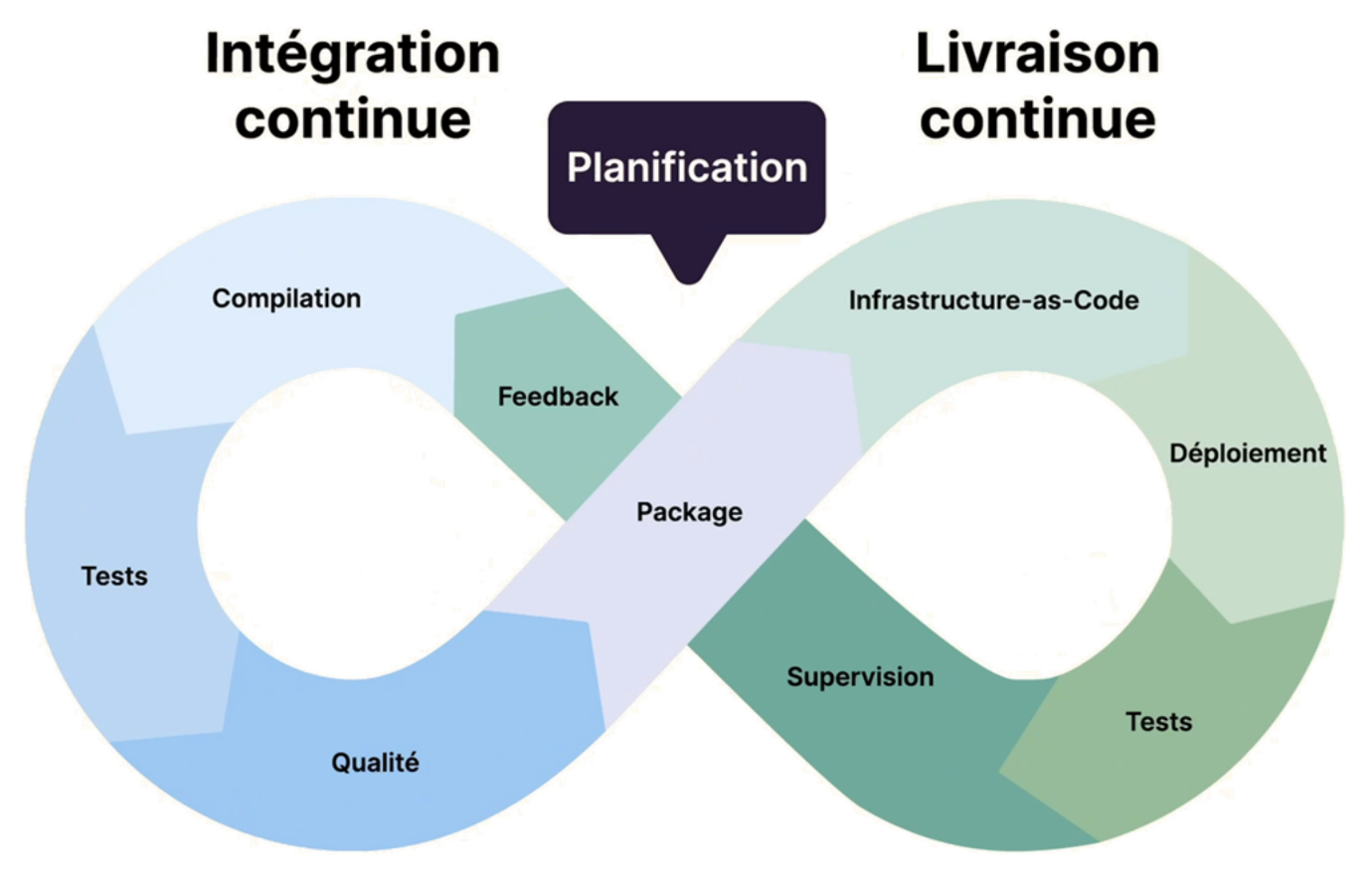
## Technologiques

La vue technologique correspond à la description des composants logiciels, de leurs interactions, de leurs infrastructures. Cela inclut la définition des technologies mobilisées pour implémenter l’architecture.

**Pipelines CI/CD**

Une pipeline CI/CD (Continuous Integration/Continuous Deployment) est un ensemble automatisé de processus permettant aux équipes de développement de livrer du code de manière rapide et sûre. Elle intègre les phases de construction, de test, de qualité, de packaging, de gestion de l'infrastructure, de déploiement, de supervision et de feedback, garantissant ainsi un cycle de développement continu et sans interruption.

**Mise en place d'une pipeline CI/CD :**

****

1. **Intégration & Compilation :**
   * Cette étape consiste à récupérer le code source depuis le référentiel, le compiler et générer une version exécutable de l'application.
2. **Tests :**
   * Les tests sont effectués à trois niveaux : unitaires, d'intégration et end-to-end (e2e). Les tests unitaires vérifient le bon fonctionnement des composants individuels, tandis que les tests d'intégration et d'e2e valident le comportement de l'application dans son ensemble.
3. **Qualité :**
   * Une revue de code est effectuée pour évaluer la qualité du code source et identifier les problèmes potentiels. Cela peut inclure des vérifications de conformité aux normes de codage, des analyses de sécurité et des conseils de bonnes pratiques.
4. **Packaging :**
   * Le code source est conditionné dans une image Docker prête à être déployée. Cela facilite la portabilité et la gestion des dépendances de l'application.
5. **Infrastructure as Code :**
   * L'infrastructure requise pour déployer l'application est définie et gérée sous forme de code. Cela permet une gestion efficace et reproductible de l'infrastructure.
6. **Déploiement :**
   * L'application est déployée dans l'environnement de production ou de staging. Cela peut impliquer un déploiement sur des serveurs physiques, des machines virtuelles ou des conteneurs.
7. **Tests :**
   * Des tests supplémentaires sont effectués pour évaluer la sécurité, les performances et la stabilité de l'application. Cela comprend des analyses de sécurité, des tests de charge et des tests de fumée pour vérifier les fonctionnalités de base.
8. **Supervision :**
   * Des mesures métriques sont collectées pour surveiller la performance de l'application en production. Cela inclut des métriques de latence, de trafic, d'erreurs retournées et de saturation des ressources.
9. **Feedback :**
   * Les retours des utilisateurs et des développeurs sont recueillis pour identifier les problèmes et les améliorations potentielles. Cela alimente le processus d'amélioration continue du produit.

**Docker**

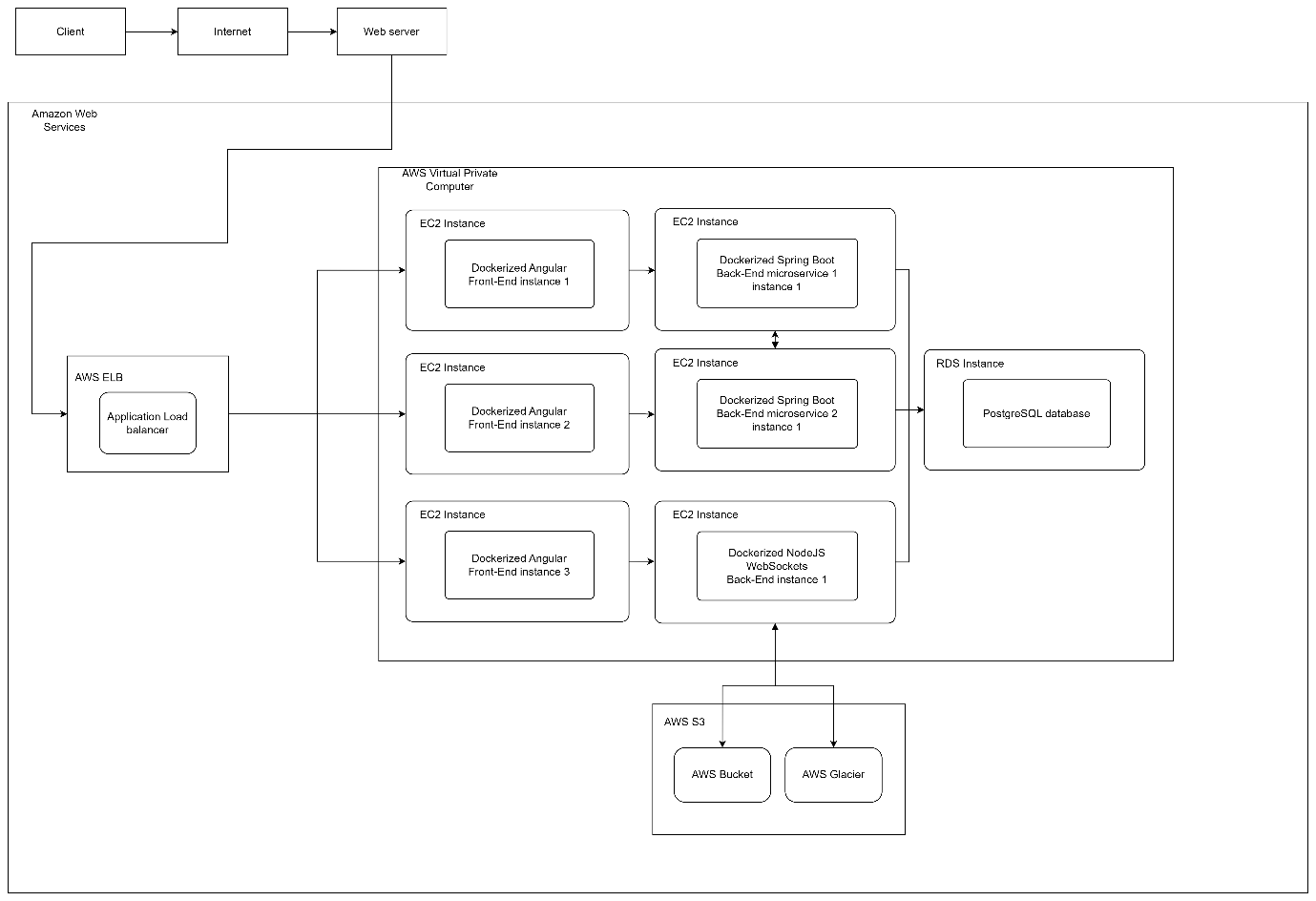
Docker est une plateforme de conteneurisation qui permet de créer, déployer et exécuter des applications de manière isolée et cohérente. En empaquetant les applications et leurs dépendances dans des conteneurs, Docker assure la portabilité et la scalabilité, facilitant ainsi le développement et le déploiement continu dans différentes plateformes de Cloud.

Docker apporte plusieurs choses dans notre projet :

* **Isolation des Environnements :** Chaque composant de notre architecture (par exemple, les services Back-End, la base de données, les services Front-End) sera conteneurisé. Cela garantit que chaque composant fonctionne dans un environnement isolé, ce qui évite les conflits de dépendances et facilite la gestion des versions.
* **Portabilité :** Les conteneurs Docker peuvent être exécutés de manière cohérente sur différents environnements, qu'il s'agisse de machines de développement locales, de serveurs de production ou de services cloud. Cette portabilité simplifie le déploiement et le scaling de l'application.
* **CI/CD :** Les pipelines CI/CD intégreront Docker pour construire des images de conteneurs à chaque commit. Ces images seront ensuite testées et déployées automatiquement. Par exemple, lors de la phase de packaging de notre pipeline CI/CD, le code source sera conditionné dans une image Docker prête à être déployée, ce qui facilite la portabilité et la gestion des dépendances de l'application.

**Diagramme de Déploiement**

Le diagramme de déploiement illustre la configuration des composants matériels et logiciels dans le système. Il montre comment les différentes entités logicielles sont déployées sur l'infrastructure physique ou virtuelle, mettant en évidence les nœuds, les artefacts, et les connexions entre eux.



**Client** :

Les utilisateurs finaux qui interagissent avec le système via une interface web.

**Internet** :

Le réseau global permettant la communication entre les clients et les composants de ton architecture.

**Load Balancer (LB)** :

Distribue les requêtes entrantes aux instances appropriées pour équilibrer la charge et améliorer la disponibilité. Il peut s'agir d'un AWS Application Load Balancer (ALB) ou Network Load Balancer (NLB).

**Web Server** :

Reçoit les requêtes du Load Balancer et les redirige vers les instances de services backend ou frontend. Ce serveur peut être Nginx, Apache ou tout autre serveur web.

**AWS Virtual Private Cloud (VPC)** :

Un environnement réseau isolé dans AWS, où sont hébergées les ressources telles que les instances EC2 et la base de données RDS.

**Groupes de composants au sein de l'AWS VPC** :

**EC2 Instance 1** :

Héberge un service backend Dockerisé utilisant Spring Boot pour les requêtes, désigné comme "Spring Boot Back-End microservice 1 instance 1".

**EC2 Instance 2** :

Héberge un service backend Dockerisé utilisant NodeJS pour les requêtes WebSockets, désigné comme "NodeJS WebSockets Back-End microservice 1 instance 1".

**EC2 Instance 3** :

Héberge une application frontend Dockerisée développée avec Angular, désignée comme "Front-End instance 1".

**RDS Instance** :

Un service de base de données relationnelle hébergé sur AWS RDS, utilisant PostgreSQL comme système de gestion de base de données.

**AWS S3** :

Amazon S3 est utilisé pour stocker les fichiers multimédias, tels que les photos et les vidéos des annonces de location.

Un bucket S3 permet de stocker ces fichiers de manière durable et sécurisée.

**AWS Glacier** :

Amazon Glacier est utilisé pour l'archivage des données, comme les discussions. Cela permet de conserver des données historiques à moindre coût.

Glacier est conçu pour le stockage à long terme et le back-up, assurant la sécurité des données archivées.

**Architecture Microservice Back-End**

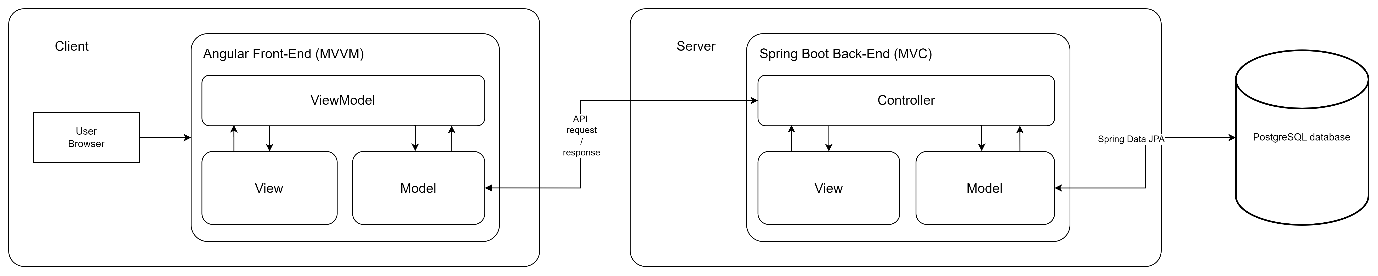
L'architecture microservices consiste à structurer une application en un ensemble de services indépendants, chacun étant responsable d'une fonctionnalité spécifique. Chaque microservice peut être développé, déployé et mis à l'échelle indépendamment, facilitant ainsi l'agilité et la résilience du système global.

**Diagramme Architecture Client-Serveur et Protocoles HTTPS**

Le diagramme d'architecture client-serveur montre comment les clients (navigateurs web, applications mobiles) interagissent avec les serveurs via des requêtes HTTP/HTTPS. HTTPS (Hypertext Transfer Protocol Secure) assure la sécurité des communications en chiffrant les données échangées entre le client et le serveur.

L'architecture client-serveur sépare les responsabilités entre le client (qui gère l'interface utilisateur et la logique de présentation) et le serveur (qui gère les données, la logique métier et les règles de l'application). Cette séparation permet une meilleure modularité, évolutivité et maintenabilité de l'application.

* **Client** : Ici, le client utilise Angular et suit le modèle MVVM (Model-View-ViewModel).
* **Serveur** : Le serveur HTTP utilise Spring Boot et suit le modèle MVC (Model-View-Controller).
* **Communication** : Le client et le serveur communiquent via des requêtes HTTP, où le client envoie des requêtes pour récupérer ou modifier des données, et le serveur répond avec les résultats ou les mises à jour



**Détails du Client (Front-End) dans le Diagramme**

1. **User** : Représente l'utilisateur final interagissant avec l'application via un navigateur.
2. **Browser** : Le navigateur que l'utilisateur utilise pour accéder à l'application.
3. **Angular Front-End (MVVM)** :
   * **View** : C'est la partie visible de l'application avec laquelle l'utilisateur interagit. Elle est composée de composants Angular responsables de l'affichage des données.
   * **ViewModel** : Gère la logique de présentation et l'interaction entre la vue et le modèle. Il assure la liaison de données et les opérations de traitement des données côté client.
   * **Model** : Représente les données côté client et inclut les services Angular qui interagissent avec le back-end pour récupérer ou envoyer des données.

**Détails du Serveur HTTP (Back-End) dans le Diagramme**

1. **Server** : La machine ou l'instance exécutant l'application Spring Boot.
2. **Spring Boot Back-End (MVC)** :
   * **Model** : Contient les entités et la logique métier côté serveur. Utilise JPA pour la persistance des données.
   * **View** : La couche de présentation côté serveur, souvent constituée de templates HTML ou de réponses JSON rendues par Spring MVC.
   * **Controller** : Les contrôleurs reçoivent les requêtes HTTP du client, interagissent avec le modèle pour traiter la logique métier et renvoient les réponses appropriées (vues ou données JSON).

**Communication entre Client et Serveur**

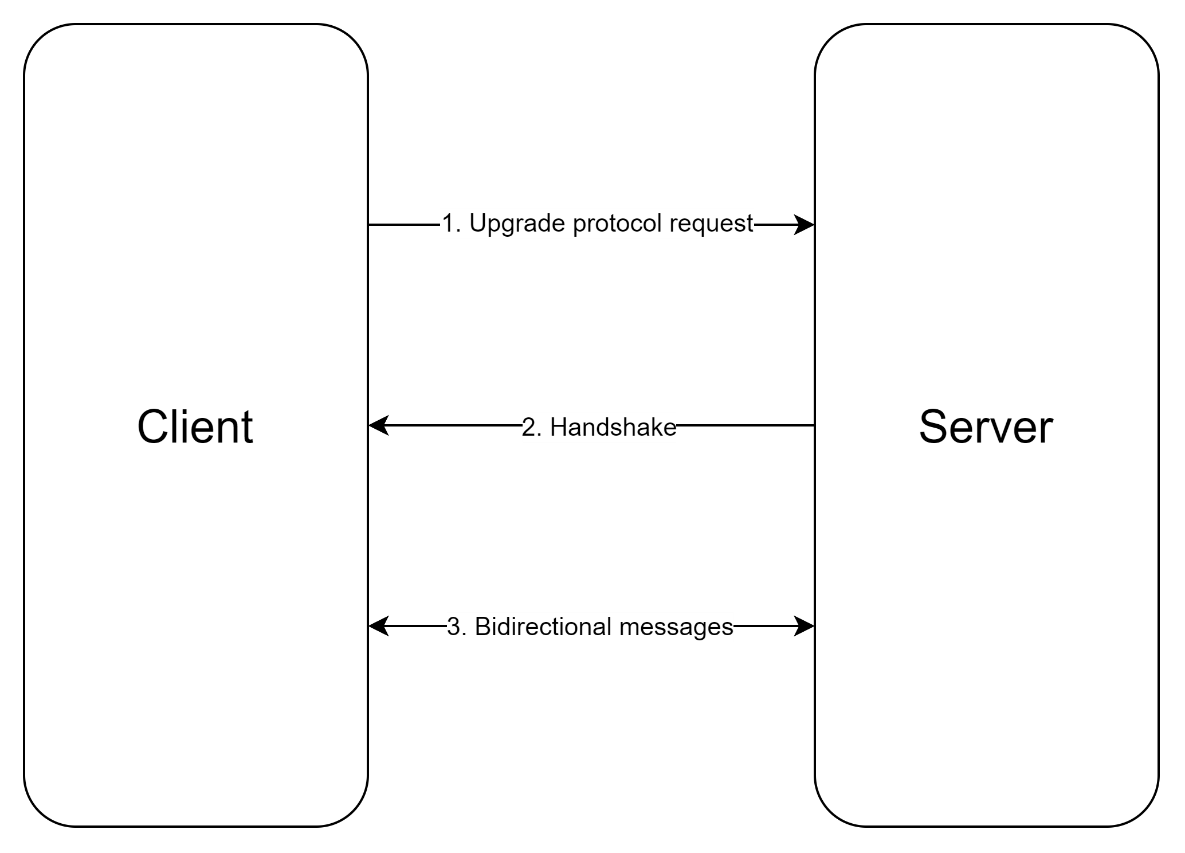
1. **HTTP Requests** :
   * Le **ViewModel** envoie des requêtes HTTP au serveur via des services Angular.
   * Les **Controllers** Spring Boot reçoivent ces requêtes, traitent les données nécessaires via le modèle, et renvoient les réponses appropriées.

**Relations et Flux de Données**

1. **View ↔ ViewModel** : La vue interagit avec le ViewModel en transmettant les actions de l'utilisateur et en mettant à jour l'affichage selon les données fournies par le ViewModel.
2. **ViewModel ↔ Model** (Client) : Le ViewModel utilise le modèle pour récupérer les données nécessaires à la vue et pour envoyer les modifications de données au serveur.
3. **ViewModel ↔ Server** : Les services Angular dans le ViewModel envoient des requêtes HTTP au serveur pour récupérer ou modifier des données.
4. **Controller ↔ Model** (Server) : Les contrôleurs Spring Boot interagissent avec le modèle pour traiter les requêtes et appliquer la logique métier.
5. **Controller ↔ View** (Server) : Les contrôleurs renvoient les vues rendues ou les données JSON en réponse aux requêtes du client.

**Diagramme Protocole WebSocket**

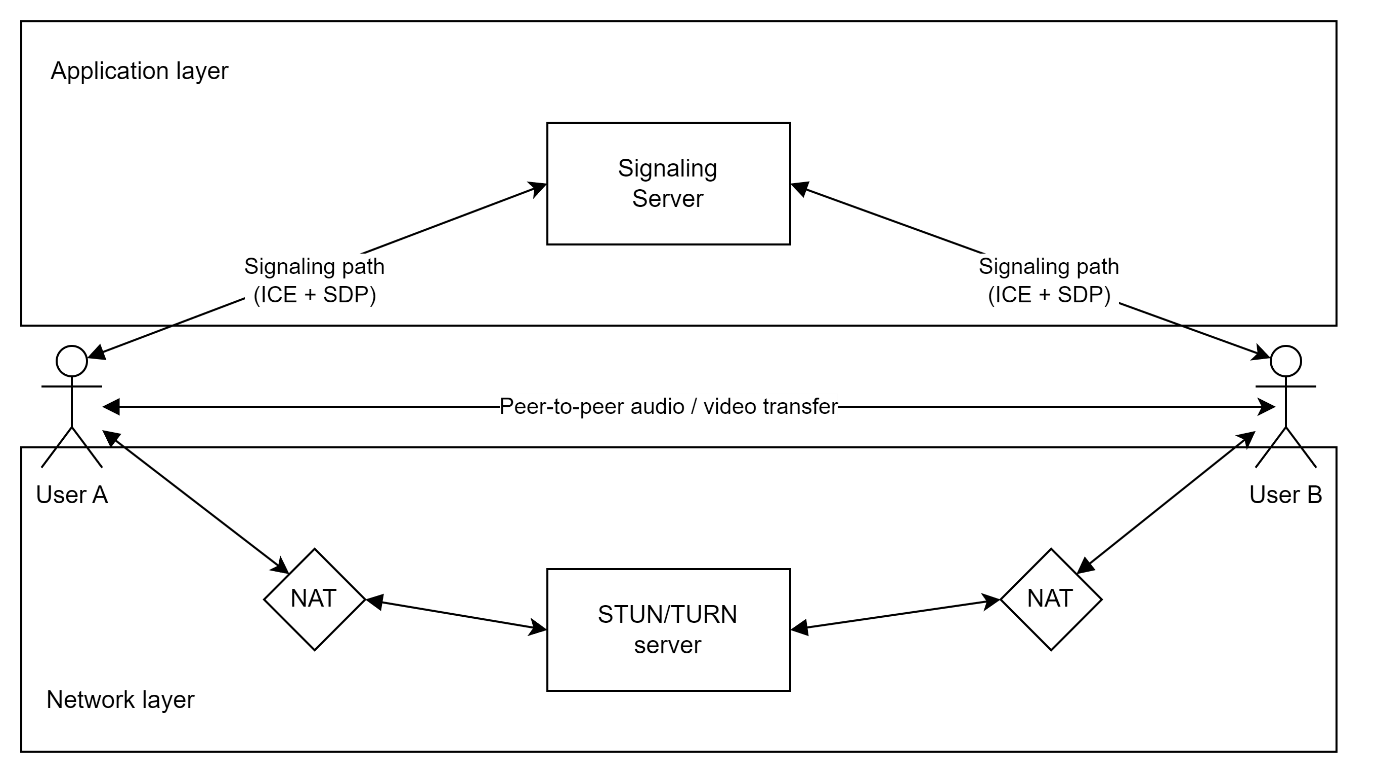
WebSocket est un protocole de communication bidirectionnelle en temps réel entre un client et un serveur. Le diagramme de WebSocket montre comment une connexion persistante est établie, permettant l'échange de messages instantanés et interactifs sans la surcharge des requêtes HTTP traditionnelles.



1. **Établissement de la Connexion**:
   * Le client initie une connexion WebSocket avec le serveur en envoyant une requête HTTP spéciale.
   * Si le serveur accepte la requête, il envoie une réponse positive qui établit la connexion WebSocket via un processus de handshake. Cette connexion reste ouverte tant que ni le client ni le serveur ne la ferment explicitement.
2. **Communication Bidirectionnelle**:
   * Une fois la connexion établie, le client et le serveur peuvent envoyer des messages dans les deux sens de manière asynchrone.
   * Les messages peuvent contenir des données en texte ou en binaire.
3. **Fermeture de la Connexion**:
   * La connexion peut être fermée par le client ou le serveur à tout moment. La fermeture est initiée en envoyant un message de fermeture spécifique, après quoi la connexion est fermée proprement.

**Diagramme Protocole WebRTC**

WebRTC (Web Real-Time Communication) est un ensemble de technologies qui permet aux navigateurs web et aux applications mobiles de réaliser des communications audio, vidéo et de partage de données en temps réel sans nécessiter de plugins supplémentaires. Le diagramme de WebRTC illustre comment les pairs (clients) établissent des connexions directes, négocient les flux multimédias et échangent des données de manière sécurisée.



1. **Clients (User A et User B) :**
   * Ce sont les utilisateurs finaux qui souhaitent établir une connexion audio/vidéo.
2. **Serveur de Signalisation (Signaling Server) :**
   * Il gère l'échange des informations de signalisation nécessaires pour établir la connexion WebRTC entre les clients. Cela inclut l'échange des descriptions de session (SDP) et des informations sur les candidats ICE.
3. **Serveur STUN/TURN :**
   * **STUN (Session Traversal Utilities for NAT) :** Il aide les clients à découvrir leur adresse publique et à traverser le NAT (Network Address Translation).
   * **TURN (Traversal Using Relays around NAT) :** Si la connexion directe n'est pas possible, le serveur TURN relaie le trafic entre les clients.
4. **NAT :**
   * Les routeurs NAT sont couramment utilisés pour permettre à plusieurs appareils sur un réseau local de partager une seule adresse IP publique.

**Processus de connexion WebRTC**

1. **Établissement de la signalisation :**
   * Les clients (Peer A et Peer B) envoient leurs informations de signalisation au serveur de signalisation via le chemin de signalisation. Le serveur de signalisation transmet ces informations entre les clients pour qu'ils puissent se découvrir et négocier la connexion.
2. **Découverte des candidats ICE :**
   * Les clients utilisent les serveurs STUN pour découvrir leurs adresses IP publiques et privées. Si nécessaire, ils utilisent les serveurs TURN pour relayer le trafic.
3. **Négociation de la connexion :**
   * Les clients échangent des descriptions de session (SDP) via le serveur de signalisation pour négocier les paramètres de la connexion.
4. **Transfert peer-to-peer :**
   * Une fois la connexion établie, les clients peuvent transférer directement l'audio/vidéo de peer-to-peer (pair à pair), contournant le serveur de signalisation pour un transfert plus efficace et avec une latence réduite.

**Technologies choisies :**

* Front-end : Angular 18
* Back-end : Java 22 et Spring Boot 3.3.0 et Node.js 20.11.0
* Base de données : PostgreSQL 16
* Conteneurisation: Docker
* Infrastructure: AWS

## Choix de la Stack Technologique pour les WebSockets

Dans le cadre de la conception de notre architecture backend, nous avons opté pour une approche *hybride* en utilisant **Java + Spring Boot pour les requêtes HTTP** et **NodeJS pour les WebSockets**. Ce choix est motivé par les raisons suivantes :

1. **Obsolescence des bibliothèques WebSocket de Spring Boot :**

Les bibliothèques disponibles pour gérer les WebSockets avec Spring Boot sur Spring Initializr, notamment [StompJS](https://github.com/stomp-js/stompjs) et [SockJS](https://github.com/sockjs/sockjs-client), ne sont plus régulièrement mises à jour. Elles requièrent une configuration complexe côté serveur, une intégration front-end laborieuse, et ne prennent pas en charge certaines fonctionnalités avancées, comme l'envoi de fichiers binaires, qui sont mieux gérées par des solutions comme [Socket.io](https://github.com/socketio/socket.io).

1. **Manque de documentation :**

Bien qu'il existe une bibliothèque non officielle pour implémenter [Socket.io](https://github.com/socketio/socket.io), nommée "[Netty-Scoket.io](https://github.com/mrniko/netty-socketio)" avec Spring Boot, celle-ci souffre d’un manque de documentation depuis 2012. Ce manque de ressources officielles complique considérablement l’intégration et la configuration des serveurs WebSocket.

Afin d'éviter les retards et les problèmes d'intégration, nous avons décidé d'utiliser NodeJS, qui bénéficie d'un support robuste et d'une documentation extensive pour la gestion des WebSockets.

Cette approche nous permet de bénéficier de la robustesse de Spring Boot pour les requêtes HTTP tout en profitant de la flexibilité et des fonctionnalités avancées offertes par NodeJS pour les communications en temps réel via WebSocket.

# .. Justification de l’approche architecturale

L'architecture proposée pour le projet "Your Car Your Way" a été soigneusement conçue pour répondre aux objectifs définis tout en respectant les principes d'architecture modernes et les normes de l'industrie. Voici une justification détaillée de cette approche :

**Objectifs et Réponse de l'Architecture**

1. **Unification et standardisation :**
   * **Approche adoptée :** La centralisation des fonctionnalités de réservation dans une seule plateforme est facilitée par une architecture microservices. Chaque microservice est dédié à une fonctionnalité spécifique, permettant une standardisation et une cohérence dans les processus.
   * **Justification :** Cette approche permet de gérer de manière homogène les différents services offerts par la plateforme, assurant une cohérence à l'échelle internationale.
2. **Amélioration de l'Expérience Client :**
   * **Approche adoptée :** Utilisation d'une application front-end développée avec Angular, offrant une interface utilisateur intuitive et réactive.
   * **Justification :** Angular permet de créer des applications web dynamiques et performantes, ce qui améliore l'expérience utilisateur en rendant l'application plus interactive et agréable à utiliser.
3. **Adaptabilité et Progression :**
   * **Approche adoptée :** Architecture modulaire et scalable grâce à Docker, intégrée dans une infrastructure cloud AWS.
   * **Justification :** Cette modularité permet de facilement ajouter de nouvelles fonctionnalités ou services sans perturber les systèmes existants, assurant ainsi la capacité de l'architecture à supporter une croissance rapide. AWS offre une infrastructure scalable avec un modèle de tarification "pay-as-you-go", permettant de s'adapter rapidement aux variations de la demande tout en optimisant les coûts.
4. **Sécurisation et Conformité :**
   * **Approche adoptée :** Mise en place de mesures de sécurité telles que le cryptage des données, l'authentification à deux facteurs et la conformité aux réglementations RGPD. De plus, l'utilisation de Spring Security offre une sécurité renforcée avec des fonctionnalités telles que la gestion des identités et des accès, la protection contre les attaques CSRF (Cross-Site Request Forgery), et les contrôles d'accès basés sur les rôles.
   * **Justification :** En suivant ces bonnes pratiques, nous assurons la protection des données des utilisateurs et la conformité légale, ce qui est crucial pour gagner la confiance des utilisateurs et éviter les sanctions réglementaires. L'intégration de Spring Security permet de mettre en place une gestion fine et robuste des accès, assurant que seules les personnes autorisées puissent accéder aux ressources sensibles, tout en offrant des protections intégrées contre les vulnérabilités courantes.
5. **Déploiement Rapide :**
   * **Approche adoptée :** Utilisation de pipelines CI/CD intégrant des étapes de tests automatiques et de déploiement continu.
   * **Justification :** Les pipelines CI/CD permettent de déployer les nouvelles versions de l'application rapidement et de manière fiable, réduisant ainsi les temps d'arrêt et facilitant une transition fluide depuis les anciennes plateformes vers la nouvelle.

**Respect des Principes d’Architecture**

1. **S.O.L.I.D. :**
   * **Approche adoptée :** Les principes S.O.L.I.D. ont été appliqués dans la conception des microservices, chaque service ayant une responsabilité unique et étant ouvert à l'extension mais fermé à la modification.
   * **Justification :** Ces principes garantissent un code modulaire, facile à maintenir et à faire évoluer, réduisant ainsi les risques de régression lors des ajouts de fonctionnalités.
2. **Sécurité :**
   * **Approche adoptée :** Intégration de pratiques de sécurité dès la phase de conception, avec un focus sur le cryptage des données, l'authentification à deux facteurs et la gestion des autorisations.
   * **Justification :** La sécurité est intégrée au cœur de l'architecture, ce qui permet de minimiser les vulnérabilités et d'assurer la protection des informations sensibles des utilisateurs.
3. **Cohérence :**
   * **Approche adoptée :** Utilisation de standards de développement et de conception cohérents, notamment en ce qui concerne les interfaces utilisateur et les interactions.
   * **Justification :** La cohérence dans le développement permet d'assurer une expérience utilisateur homogène et facilite la collaboration entre les équipes de développement.
4. **Évolutivité :**
   * **Approche adoptée :** Architecture modulaire et utilisation de technologies de conteneurisation comme Docker.
   * **Justification :** Cette approche permet de scaler horizontalement les services en fonction de la demande, assurant ainsi la performance et la disponibilité de l'application même en période de forte affluence.
5. **Testabilité :**
   * **Approche adoptée :** Mise en place de tests unitaires, d'intégration et de bout en bout, intégrés dans des pipelines CI/CD.
   * **Justification :** Les tests automatisés permettent d'assurer la qualité du code et de détecter rapidement les régressions, ce qui est essentiel pour maintenir la fiabilité de l'application.
6. **Performance :**
   * **Approche adoptée :** Optimisation des performances via des pratiques de développement efficaces et la mise en place de mécanismes de mise à l'échelle automatique.
   * **Justification :** Une application performante améliore l'expérience utilisateur et réduit les coûts opérationnels liés à l'infrastructure.

# .. Architectures de transition

Étant donné la complexité potentielle de la mise en œuvre complète de l'architecture finale, une approche par étapes progressives peut être envisagée. Voici une proposition d'architectures de transition :

**Phase 1 : Mise en Place de l'Infrastructure de Base**

* **Objectif :** Déployer une infrastructure de base stable permettant de supporter les premières fonctionnalités de la plateforme.
* **Composants :**
  + Mise en place de l'infrastructure AWS VPC, EC2 instances, et RDS.
  + Déploiement des premiers microservices essentiels (gestion des utilisateurs, gestion des véhicules) puis du serveur WebSockets.
  + Configuration initiale du pipeline CI/CD pour les déploiements automatisés.

**Phase 2 : Déploiement des Fonctionnalités Clés**

* **Objectif :** Ajouter des fonctionnalités clés et intégrer les services front-end et back-end.
* **Composants :**
  + Déploiement du front-end Angular.
  + Intégration des services de réservation et de consultation d'offres.
  + Mise en place des mécanismes de sécurité de base (authentification, autorisation).

**Phase 3 : Évolution et Optimisation**

* **Objectif :** Optimiser l'architecture et ajouter des fonctionnalités avancées.
* **Composants :**
  + Intégration de services avancés (support client, évaluation des services).
  + Optimisation des performances et mise en place de la scalabilité automatique via AWS.
  + Renforcement des mesures de sécurité et conformité RGPD.

**Phase 4 : Finalisation et Mise en Production Complète**

* **Objectif :** Finaliser l'architecture et assurer une mise en production complète et stable.
* **Composants :**
  + Achèvement de la migration des données et des utilisateurs.
  + Tests de performance et de sécurité complets.
  + Mise en place de la surveillance continue et des mécanismes de feedback utilisateur.

En suivant cette approche progressive, il est possible de gérer efficacement la complexité du projet, en assurant une transition fluide vers l'architecture finale tout en minimisant les risques et en garantissant la qualité et la stabilité du produit final. L'utilisation d'AWS offre des avantages significatifs en termes de scalabilité, de flexibilité et de coût, assurant ainsi une infrastructure robuste et adaptable aux besoins changeants de l'application.

