



**WelHome.**  
Maison, bien-être et habitat durable



# Welhome SA

## Membres :

- Kerim HUDAYBERDIYEV
- Ratiba KADI
- Shayma KHELLADI
- Meryem KÖSE
- Euphraïm LUZOLO
- Quang Viet NGUYEN
- Ousseynou SAKHO
- Jessy VY
- Zain ZAFAR
- Malak ZEMRANI

*Lien vers la page web de notre Notion groupé sur lequel nous avons collaboré tout au long du projet :*

[Welhome SA](#) (veuillez vous créer un compte Notion pour pouvoir accéder à l'ensemble de nos collaborations)

## Préambule

Dans le cadre de la matière Architecture des systèmes d'informations, nous avons eu à concevoir une application de logements basée sur l'architecture des micro-services.

Dans ce rapport, nous allons présenter les détails de chaque micro-service que nous avons implémenté en passant de la conception à la réalisation.

Le travail a été effectué en groupe, pour concevoir une solution facile à utiliser et offrant une expérience fluide, pour les locataires et les propriétaires.

Nous allons expliquer comment nous avons mis en place chaque micro-service, et les technologies que nous avons utilisées pour les développer.

Enfin, nous montrerons comment ces micro-services interagissent les uns avec les autres pour offrir une expérience utilisateur cohérente et intuitive.

Nous tenons à préciser que notre solution est accessible via <http://zain.ovh:9091> et que le code de tous nos projets est disponible sur notre repo git <https://github.com/zainzafar/welhome>.

[Préambule](#)

[Objectif du projet](#)

[Problématiques](#)

[L'architecture générale](#)

[Organisation et déroulement du projet](#)

[Les équipes](#)

[Les rôles](#)

[Suivi des tâches](#)

[I. Base de données & sa persistance](#)

[Donner vie à un site : création et gestion de la base de données](#)

[Du papier à l'écran : comment les diagrammes se transforment en une base de données exploitable](#)

[Les diagrammes papiers se transforment en un script de base de données finalisée](#)

[Du script de la base données à l'API](#)

[Bilan & axes d'améliorations](#)

[II. Le fonctionnement en coulisses : le back-end](#)

[Premières impressions sur les tâches à réaliser](#)

[Répartition des tâches](#)

[Implémentation des fonctionnalités dans Spring Boot 3](#)

[Endpoints et fonctionnalités ajoutées](#)

[Implémentation des différents endpoints](#)

[Bilan & axes d'améliorations](#)

[III. La partie visible de l'iceberg : le front-end](#)

[Conception : choix de conception UI/UX et présentation de la maquette](#)

[Implémentation](#)

[Fonctionnalités](#)

[Les fonctionnalités clés d'Angular utilisées dans notre application](#)

[Composant Reservation-list](#)

[Composant Reservation-card](#)

[Routage](#)

[Guards](#)

[Intégration avec d'autres services de l'application](#)

[Intégration des fonctions backend pour une application web stable et performante](#)

[Intégration des fonctionnalités du service d'authentification](#)

[Bilan & axes d'améliorations](#)

[IV. Gérer les accès et autorisations à un site : le micro-service d'authentification](#)

[Initiation à l'authentification avec OAuth2](#)

[OAuth2](#)

[Diagramme fonctionnel high-level](#)

[Technologies employées](#)

[Implémentation](#)

[Authentification auprès de Google](#)

[Récupération et vérification de l'access token](#)

[Vérification de l'existence de l'utilisateur dans la base de données](#)  
[Gestion des utilisateurs existants et nouveaux](#)  
[Enregistrement des nouveaux utilisateurs](#)  
[Finalisation du processus d'authentification](#)  
[Rafraîchissement de l'accès token](#)  
[Collaboration avec le Frontend, le Backend et le Déploiement](#)  
[Bilan & axes d'amélioration](#)

**V.** Toujours rester informé : le mailing

[Implémentation](#)  
[Utilité du token d'authentification](#)  
[Finalité](#)

**VI.** Finalisation : le déploiement

[Repartition des tâches](#)  
[Architecture finale](#)  
[Plan de déploiement](#)  
[Résultats de déploiement](#)

**VII.** Conclusion générale

[Liens utiles](#)  
[Comment lancer le projet en localhost ?](#)

## Objectif du projet

L'objectif est de créer une application de location de logement basée sur l'architecture des micro services. L'application doit être facile à utiliser et offrir une expérience fluide pour les locataires et les propriétaires. Nous devons mettre en place 4 micro-services fonctionnels, pouvant interagir les uns avec les autres, avec une IHM si possible en bonus.

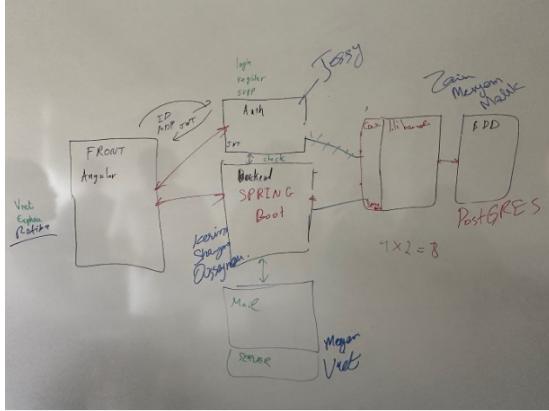
## Problématiques

De prime abord, les 2 questions principales qu'on s'est posées étaient les suivantes :

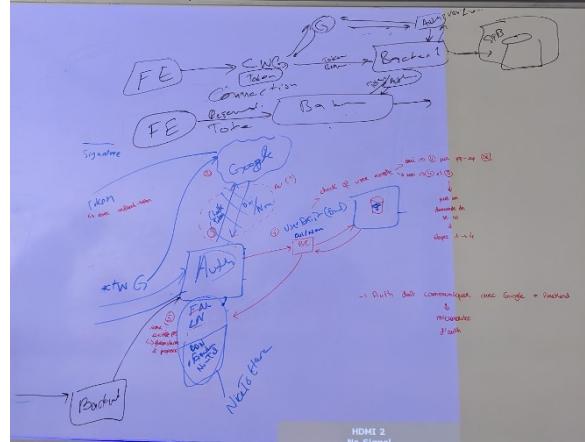
- Quels micro-services allons-nous implémenter ?
- Comment vont-ils être agencés les uns avec les autres ?

C'est pourquoi, dès le commencement du projet, nous avons décidé ensemble des micro-services qu'on allait implémenter.

Après plusieurs réflexions, sur non seulement la façon dont nos micro-services allaient être agencés, mais aussi sur la façon dont ces derniers vont interagir entre eux, on s'est quitté sur des trames d'idées, dont voici des aperçus :



Réflexions à propos des micro-services



Idées au sujet de l'implémentation

Comme il est visible sur la première image, nous avons essayé au mieux de trouver une répartition équitable des tâches au sein de notre équipe.

Il s'agissait là d'une introduction générale à l'ensemble des services et technologies que nous avons utilisés.

Afin que l'explication de chacun des services soit le plus ludique possible, nous avons décidé que chaque groupe s'occupe de la rédaction de leur rapport concernant la partie implémentée.

Les 4 micro-services qu'on a souhaité implémenter sont les suivants :

- **Backend** : gérer la jonction entre le frontend et bases de données
- **Authentification** : à l'aide de OAuth2
- **Mailing-Service** : afin d'informer les utilisateurs de nouveaux événements
- **Database-service** : qui représente le point d'entrée et de sortie vers la base de données

## L'architecture générale

Comme le montre le diagramme ci-dessous, un grand nombre de technologies différentes ont été utilisées dans le cadre de ce projet.

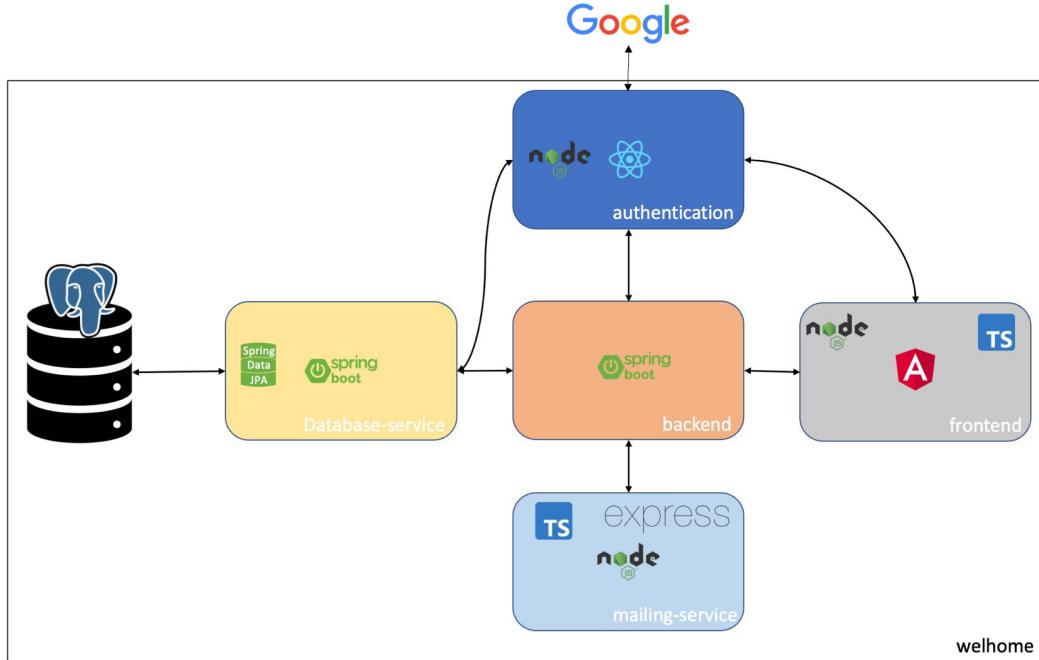


Schéma global exposant les différentes technologies dont regorge notre site

Nous allons suivre une approche en entonnoir inversé pour cette fois, afin que nos choix d'implémentation soit justifiés.

## Organisation et déroulement du projet

### Les équipes

Frontend	Backend	Database	Authentification	Mailing service	CI/CD
Ratiba KADI	Kerim HUDAYBERDIYEV	Meryem KÖSE	Jessy Vy	Quang Viet NGUYEN	Euphraïm LUZOLO
Euphraïm LUZOLO	Shayma KHELLADI	Zain ZAFAR			Quang Viet NGUYEN
Quang Viet NGUYEN	Ousseynou SAKHO	Malak ZEMRANI			Zain ZAFAR

### Les rôles

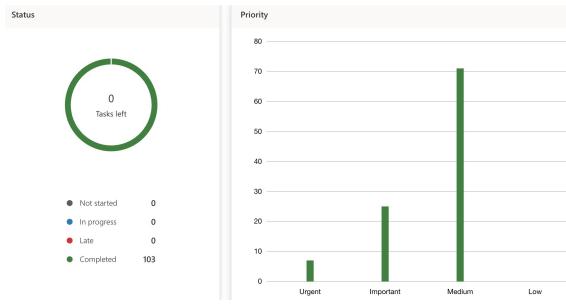
Role	Person
Project Manager	Zain ZAFAR
Frontend Lead	Euphraïm LUZOLO
Backend Lead	Kerim Serdarovitch HUDAYBERDIYEV
Authentication Lead	Jessy VY
Database/database-service Lead	Meryem KÖSE
Mailing-service Lead	Quang Viet NGUYEN

## Suivi des tâches

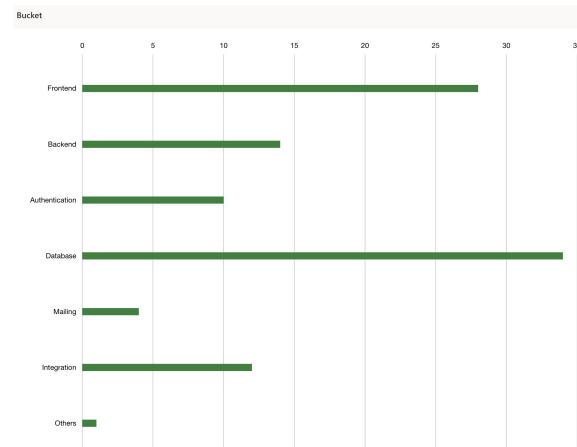
Dans le cadre de notre projet, nous avons implémenté un tableau Kanban afin d'assurer un suivi efficace des différentes tâches. Cette méthode s'est révélée extrêmement utile pour garantir un déroulement fluide de notre projet.

Au total, plus de 100 tâches ont été créées, dont 7 ont été identifiées "très prioritaires", 25 "prioritaires", et des tâches comme moyennement prioritaires.

L'évaluation des tâches était effectuée lors de nos réunions hebdomadaires, qui est devenue quotidienne lors de la dernière semaine pour assurer une gestion plus proactive du projet.



Aperçu de l'ensemble des priorités de tâches



Rapport de l'ensemble de la répartition des tâches

## I. Base de données & sa persistance

Le micro-service de base de données a pour objectif principal d'assurer l'isolation de la base de données par rapport aux autres composants de l'application. En effet, il représente l'unique point d'entrée et de sortie de la base de données, offrant ainsi une sécurité accrue et un meilleur contrôle de l'accès à cette dernière.

En séparant la base de données du reste des services, nous avons pu garantir une gestion des données fiable et efficace, tout en minimisant les risques de conflits et de pertes de données. Dans ce rapport, nous allons détailler les choix techniques et les fonctionnalités implémentés pour réaliser ce micro-service.

### Donner vie à un site : création et gestion de la base de données

Cette partie était liée à la gestion de la base de données allant permettre de réaliser les opérations CRUD et plus, sur les données dont nous disposons, afin de rendre vivante notre application.

Le but de cette section sera d'expliquer la procédure qu'on a suivie afin d'élaborer notre base de données relationnelle, sous PostgreSQL.

## Du papier à l'écran : comment les diagrammes se transforment en une base de données exploitable

### Les diagrammes papiers se transforment en un script de base de données finalisée

Avant de mettre en place notre base de données définitive, on a tout d'abord réalisé un brainstorming des informations global que l'on pensait utile d'avoir dans notre application.

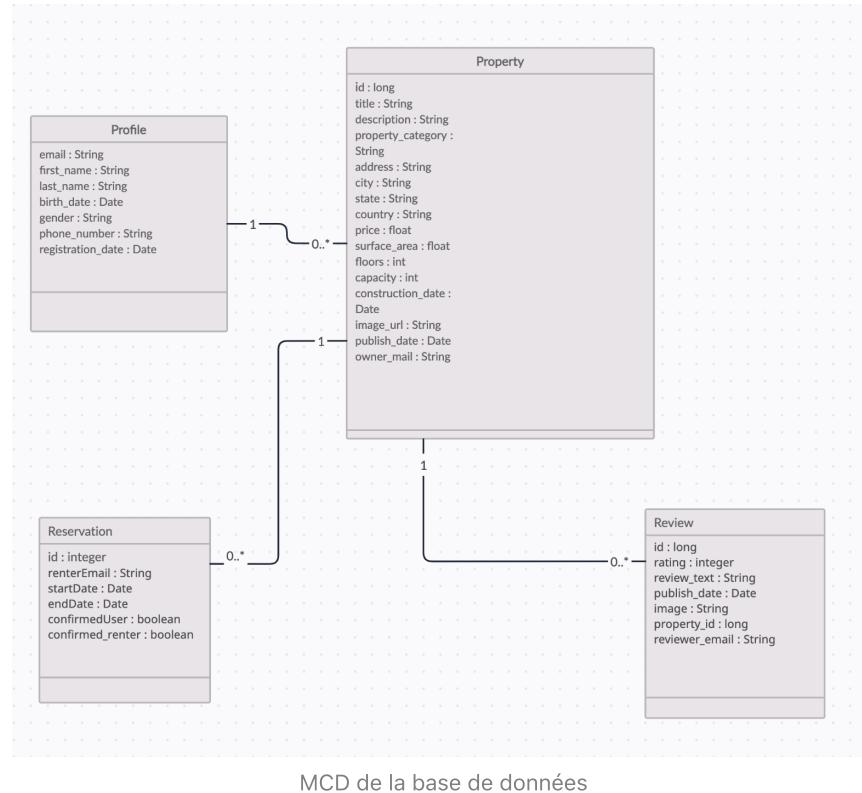
Voici un aperçu :



Un brouillon de réflexion en équipe

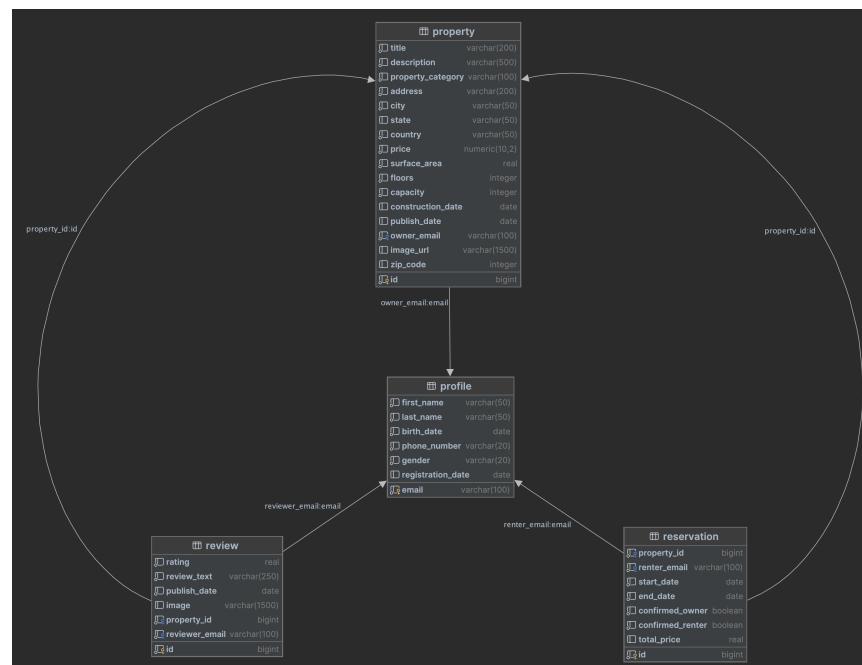
Bien qu'elle puisse paraître brouillon, cette ébauche nous a permis d'avoir une première idée du squelette de notre base de données. Puis après plusieurs modifications, nous avons élaboré notre premier modèle : le modèle conceptuel de données (MCD).

Le voici :



En s'appuyant sur ce dernier, on est passé par la modélisation logique des données (MLD) qui nous a permis de réaliser les liens entre nos tables .

Notre base de données relationnelle commençait à prendre forme :



On a ensuite mis en place un diagramme de classe:

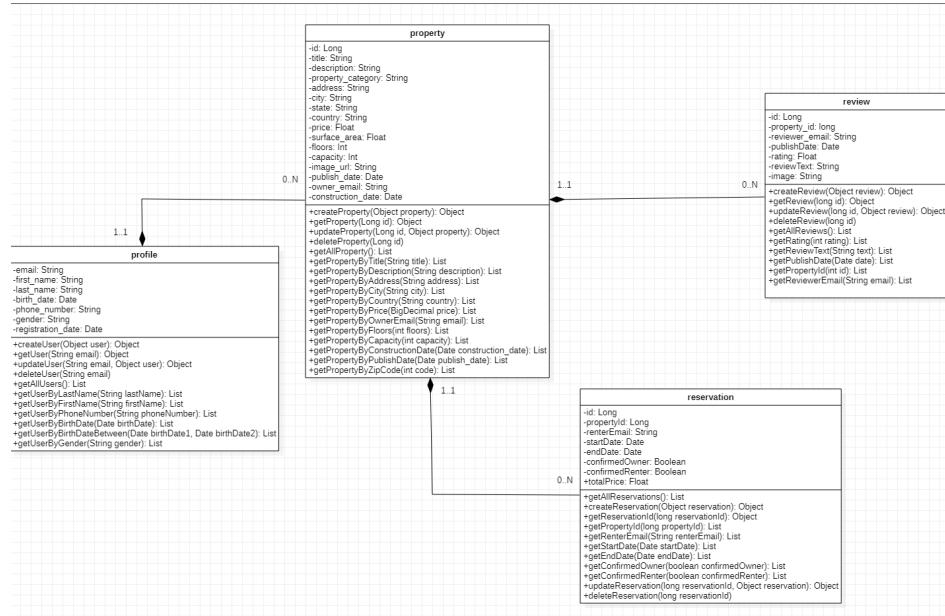


Diagramme de classe

On a suivi une logique REST (REpresentational STate Transfer), en faisant appel à des méthodes HTTP permettant le CRUD pour manipuler nos données :

- **Create** : méthode POST, va créer une nouvelle ressource à partir des données fournies dans le corps de la requête.
- **Read** : méthode GET, va récupérer une ou plusieurs ressources existantes en fonction des paramètres de la requête (par exemple, en spécifiant un ID).
- **Update** : méthode PUT, elle permet de mettre à jour une ressource existante en fonction de l'ID spécifié dans la requête, à partir des données fournies dans le corps de la requête.
- **Delete** : méthode DELETE, permet de supprimer une ressource existante en fonction de l'ID spécifié dans la requête.

→ Bien sûr, la mise en place définitive de notre base de données a nécessité plusieurs itérations en groupe, sur le modèle afin de compléter les données qui nous semblaient essentielles.

Dès qu'on s'est tous mis d'accord, on a élaboré le script final de la base de données qu'on va mettre en place :

```

CREATE TABLE profile
(

```

```

email          VARCHAR(100) UNIQUE NOT NULL PRIMARY KEY,
first_name     VARCHAR(50)      NOT NULL,
last_name      VARCHAR(50)      NOT NULL,
birth_date     DATE           NOT NULL,
phone_number   VARCHAR(20)      NOT NULL,
gender         VARCHAR(20)      NOT NULL CHECK (gender IN ('Male', 'Female', 'Non-Binary')),
registration_date DATE DEFAULT CURRENT_DATE
);

CREATE TABLE property
(
    id            BIGSERIAL PRIMARY KEY,
    title         VARCHAR(200)  NOT NULL,
    description   VARCHAR(500)  NOT NULL,
    property_category VARCHAR(100) NOT NULL CHECK (property_category IN ('House', 'Apartment', 'Room')),
    address       VARCHAR(200)  NOT NULL,
    city          VARCHAR(50)   NOT NULL,
    state         VARCHAR(50),
    country       VARCHAR(50)   NOT NULL,
    price          NUMERIC(10, 2) NOT NULL CHECK (price > 0),
    surface_area  REAL          NOT NULL CHECK (surface_area > 0),
    floors         INT           NOT NULL,
    zip_code      INT           NOT NULL,
    capacity       INT           NOT NULL CHECK (capacity > 0),
    construction_date DATE CHECK (construction_date <= CURRENT_DATE),
    publish_date   DATE DEFAULT CURRENT_DATE,
    owner_email    VARCHAR(100)  NOT NULL,
    image_url     VARCHAR(1500),
    CONSTRAINT property_email_fk FOREIGN KEY (owner_email) REFERENCES profile (email) ON DELETE CASCADE
);
CREATE INDEX property_property_id_idx ON property (id);

CREATE TABLE reservation
(
    id            BIGSERIAL PRIMARY KEY,
    property_id   BIGSERIAL      NOT NULL,
    renter_email  VARCHAR(100)  NOT NULL,
    start_date    DATE          NOT NULL,
    end_date      DATE          NOT NULL,
    confirmed_owner BOOLEAN      NOT NULL,
    confirmed_renter BOOLEAN      NOT NULL,
    CONSTRAINT reservation_property_fk FOREIGN KEY (property_id) REFERENCES property (id) ON DELETE SET NULL,
    CONSTRAINT reservation_email_fk FOREIGN KEY (renter_email) REFERENCES profile (email) ON DELETE SET NULL,
    CONSTRAINT reservation_dates_ck CHECK (end_date >= start_date)
);
CREATE TABLE review
(
    id            BIGSERIAL PRIMARY KEY,
    rating        REAL          NOT NULL CHECK (rating >= 0 AND rating <= 5),
    review_text   VARCHAR(250) NOT NULL,
    publish_date  DATE          NOT NULL DEFAULT CURRENT_DATE,
    image         VARCHAR(1500),
    property_id   BIGINT        NOT NULL,
    reviewer_email VARCHAR(100) NOT NULL,
    CONSTRAINT review_property_fk FOREIGN KEY (property_id) REFERENCES property (id) ON DELETE SET NULL,
    CONSTRAINT review_email_fk FOREIGN KEY (reviewer_email) REFERENCES profile (email) ON DELETE SET NULL
);
CREATE INDEX review_property_id_idx ON review (property_id);

```

⇒ On a veillé à ce que des vérifications logiques soient réalisées avant l'insertion dans la base de données

*Une fois que l'on s'est mis d'accord sur le script, on a décidé de mettre en place les endpoints : ponts d'entrée à une API (Application Programming Interface), allant permettre d'interagir avec la base de*

données.

## Du script de la base données à l'API

Avant de commencer à coder les points d'entrée, on a décidé de faire un nouveau brainstorming pour se mettre d'accord sur les méthodes qu'on allait utiliser sur nos données.

Redirection vers la page où nos endpoints ont été définis :  [Endpoints](#)

Suite à la définition des endpoints, nous avons collaboré par le biais de GitHub, et l'utilisation de IntelliJ Ultimate. Notre code a inclus l'utilisation de plugins comme le plugin JPA Support, qui nous a permis de mettre en place des structures d'entités en concordance avec les éléments de notre base de données. Par la suite, nous avons principalement utilisé les fonctionnalités d'Hibernate et Spring Boot Data, notamment les annotations, afin de réaliser des requêtes directes vers nos données.

→ Pour de plus amples informations :

- Voici le site nous ayant guidé tout au long de l'élaboration des endpoints à l'aide de Spring Boot Data :

**Spring Data Annotations | Baeldung**  
Learn about the most important annotations we need to handle persistence using the Spring Data project

 <https://www.baeldung.com/spring-data-annotations>



A propos de l'implémentation des différents endpoints, on s'est répartis les tâches de cette façon :

Endpoints en charge	Malak	Meryem	Zain
profile			<input checked="" type="checkbox"/>
property		<input checked="" type="checkbox"/>	
reservations	<input checked="" type="checkbox"/>		
review		<input checked="" type="checkbox"/>	

→ Cette définition des tâches est globale, bien sûr quand une personne parmi nous avait des problèmes liés à un endpoint, on essayait au mieux possible de s'entraider.

Le rendu final liée à cette partie de gestion des données de notre base de données a été conclue correctement. Nous avons réalisé l'ensemble des tâches que l'on jugeait nécessaire.

Voici le lien vers la vidéo de démonstration exposant la façon dont nous avons intégré l'utilisation de notre base de données à notre site : [voir vidéo démo](#)

## Bilan & axes d'améliorations

En guise de fermeture liée à la base de données, on peut en conclure que nous pensons avoir atteint les objectifs qu'on s'était fixé au début, dans les temps adéquats.

En revanche, nous pensons que tout travail peut être perfectionné, c'est pourquoi nous avons essayé d'améliorer notre script.

Voici une version plus performante du script :

```

CREATE TABLE profile(
    email VARCHAR(100) UNIQUE NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    birth_date DATE NOT NULL,
    phone_number VARCHAR(50) NOT NULL,
    gender VARCHAR(50),
    is_owner BOOLEAN DEFAULT FALSE,
    registration_date DATE DEFAULT CURRENT_DATE,
    PRIMARY KEY (email),
);

CREATE INDEX profile_email_idx ON profile(email);

CREATE TABLE location(
    location_id BIGSERIAL PRIMARY KEY,
    city VARCHAR(100) NOT NULL,
    state_province VARCHAR(100),
    country VARCHAR(100) NOT NULL,
    latitude NUMERIC(10, 6),
    longitude NUMERIC(10, 6)
);

CREATE TABLE amenity(
    amenity_id BIGSERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);

CREATE TABLE property_amenity(
    property_id BIGINT NOT NULL REFERENCES property(property_id) ON DELETE CASCADE,
    amenity_id BIGINT NOT NULL REFERENCES amenity(amenity_id) ON DELETE CASCADE,
    PRIMARY KEY (property_id, amenity_id)
);

CREATE TABLE property(
    property_id BIGSERIAL PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    address VARCHAR(200) NOT NULL,
    description VARCHAR(500) NOT NULL,
    image_url VARCHAR(1500),
    price NUMERIC(10,2) NOT NULL CHECK (price > 0),
    location_id BIGINT NOT NULL REFERENCES location(location_id) ON DELETE CASCADE,
    construction_date DATE CHECK (construction_date <= CURRENT_DATE),
    area REAL NOT NULL CHECK (capacity > 0),
    capacity INT NOT NULL CHECK (capacity > 0),
    property_type VARCHAR(100) NOT NULL,
    floors INT NOT NULL,
    publish_date DATE DEFAULT CURRENT_DATE,
    email VARCHAR(100) NOT NULL REFERENCES profile(email) ON DELETE SET NULL,
    CONSTRAINT property_location_fk FOREIGN KEY (location_id) REFERENCES location(location_id),
    CONSTRAINT property_email_fk FOREIGN KEY (email) REFERENCES profile(email)
    CONSTRAINT property_property_id_idx INDEX (property_id)
);

CREATE TABLE reservation(
    reservation_id BIGSERIAL PRIMARY KEY,
    start_date DATE NOT NULL,
    end_date DATE NOT NULL,
    total_price NUMERIC(10,2) NOT NULL,
    status VARCHAR(20) NOT NULL,
    confirmed_owner BOOLEAN NOT NULL,

```

```

confirmed_renter BOOLEAN NOT NULL,
property_id BIGINT NOT NULL REFERENCES property(property_id) ON DELETE CASCADE,
email VARCHAR(100) NOT NULL REFERENCES profile(email) ON DELETE SET NULL,
CONSTRAINT reservation_property_fk FOREIGN KEY (property_id) REFERENCES property(property_id),
CONSTRAINT reservation_property_id_idx INDEX (property_id),
CONSTRAINT reservation_dates_ck CHECK (end_date >= start_date),
);

CREATE TABLE review(
review_id BIGSERIAL PRIMARY KEY,
rating REAL NOT NULL CHECK (rating >= 0 AND rating <= 5),
review_comment VARCHAR(250) NOT NULL,
review_date DATE NOT NULL DEFAULT CURRENT_DATE,
reviewer_name VARCHAR(100) NOT NULL,
reviewer_profile_image VARCHAR(1500),
property_id BIGINT NOT NULL REFERENCES property(property_id) ON DELETE CASCADE,
email VARCHAR(100) NOT NULL REFERENCES profile(email) ON DELETE SET NULL,
CONSTRAINT review_property_fk FOREIGN KEY (property_id) REFERENCES property(property_id),
CONSTRAINT review_property_id_idx INDEX (property_id),
CONSTRAINT review_email_fk FOREIGN KEY (email) REFERENCES profile(email) ON DELETE SET NULL
);

```

En adoptant une vision plus globale vis-à-vis de notre projet, on peut s'intéresser à la structure des composants qui aurait pu être modifiée, si le déploiement de notre application avait été à une plus grande échelle.

En effet, dans le cas où notre application avait été distribuée sur plusieurs serveurs, il aurait fallu par exemple, que la partie liée à la gestion des dates d'état courant au niveau des tables profile, property et review soit pris en charge par des triggers, afin d'assurer une sécurité plus accrue.

Voici l'exemple de triggers qu'il aurait été possible de créer dans ce cas :

```

CREATE OR REPLACE FUNCTION update_registration_date()
RETURNS TRIGGER AS $$ 
BEGIN
NEW.registration_date = CURRENT_DATE;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER set_registration_date_on_insert
BEFORE INSERT ON profile
FOR EACH ROW
EXECUTE FUNCTION update_registration_date();

CREATE TRIGGER set_registration_date_on_update
BEFORE UPDATE ON profile
FOR EACH ROW
EXECUTE FUNCTION update_registration_date();

```

Au fur et à mesure que la partie base de données voyait le jour, le back-end a aussi commencé à être implémenté. Les endpoints de nos données commencent alors à trouver un fil conducteur, une raison d'être.

## II. Le fonctionnement en coulisses : le back-end

La finalité de cette partie est de pouvoir implémenter un microservice fonctionnel allant permettre de faire le lien entre le visuel : l'IHM final, et l'invisible : la base de données.

### Premières impressions sur les tâches à réaliser

Avant de commencer l'implémentation de cette partie, on a essayé de prendre conscience de la finalité du microservice qu'on allait mettre en place. Pour cela, il a fallu qu'on identifie quels allaient être les différents cas d'utilisations de notre interface, afin de pouvoir bien raccorder chacune des fonctionnalités aux endpoints correspondant.

C'est pourquoi nous avons réalisé ce diagramme de cas d'utilisations :

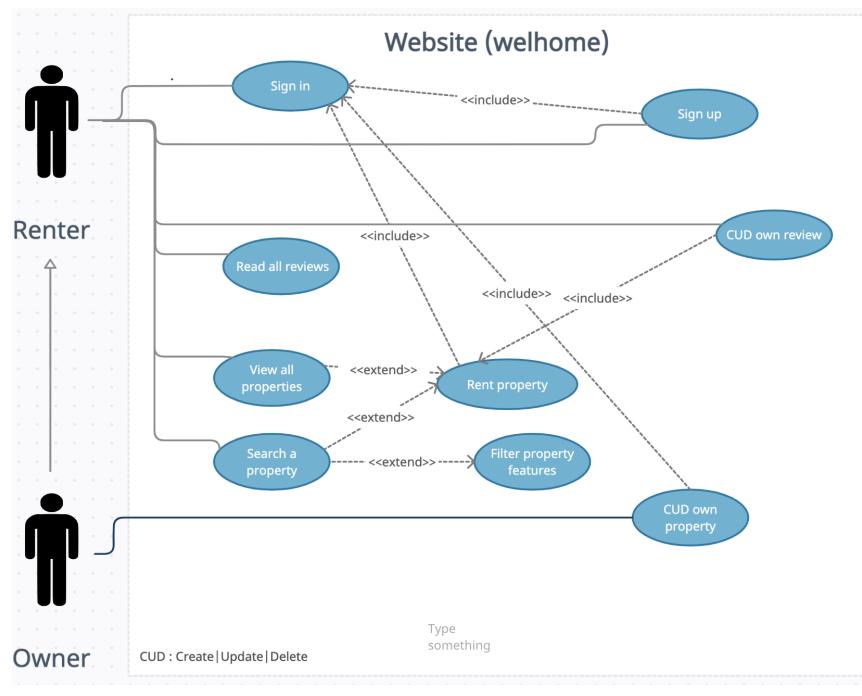


Diagramme de cas d'utilisations de notre site

A la suite de cela, nous nous sommes basés sur le diagramme de classes élaboré par l'équipe base de données. Ce dernier nous a permis de dispatcher les différentes fonctionnalités à implémenter, en fonction des différentes entités que l'on a à disposition.

Puis, en rapprochant nos réflexions précédentes, avec le swagger implanté par nos camarades de l'équipe base de données : [Swagger UI](#), nous nous sommes mis au travail pour être l'intermédiaire entre le front-end et les datas.

Aperçu de l'API : Swagger UI

## Répartition des tâches

Suite à plusieurs discussions, et après une démo commune sur certains endpoints, les tâches ont été réparties de la sorte :

Membres	Review	Profiles	Properties	Réservation
Kerim		✓	✓	✓
Ousseynou	✓			
Shayma	✓			

## Implémentation des fonctionnalités dans Spring Boot 3

Notre backend se base sur la technologie de Spring Boot 3. Afin de nous éclairer sur le sujet au commencement, nous nous sommes référé au site suivant :

Spring Boot 3 and Spring Framework 6.0 – What's New | Baeldung

Learn about new features that come with Spring Boot 3 and Spring 6.

 <https://www.baeldung.com/spring-boot-3-spring-6-new>



## Endpoints et fonctionnalités ajoutées

Le travail effectué par nos camarades du groupe base de données & persistance, a permis de poser une base sur les endpoints requis.

Un accès à leurs endpoints a donc été créé afin de former une sorte de pont entre le backend et le database-service.

Cependant, au fur et à mesure que nos réflexions ont mûries suite à des meetings réguliers, on a décidé de rajouter d'autres endpoints, afin d'améliorer la navigation/fluidité de notre site.

→ Exemples : rechercher des propriétés selon plusieurs critères tels que le prix, la localisation, le pays, la ville, la taille ou encore le nombre d'étages dont dispose la propriété.

De cette manière, il est possible de filtrer sur un ou plusieurs de ces critères en fonction de ce que l'on recherche car évidemment, les critères sont des options facultatives mais nécessaires pour permettre une recherche de propriétés à louer par le client plus flexible.

Un endpoint permettant de connaître les réservations pour une propriété spécifique d'un propriétaire donné, a également vu le jour. On a aussi décidé de prendre en compte le statut de la réservation : confirmée, en attente ou annulée

Ainsi, une réservation sera considérée :

- **Annulée** si elle n'a pas été confirmée (par le propriétaire ET le client) dans les 48 heures précédant le début de la location
- **En attente** si cette double confirmation n'a pas encore été faite et que le début de la location est dans plus de 48 heures.
- **Confirmée** : dans le cas où les deux partis ont confirmés

→ Cet endpoint permet d'ajouter un filtre sur les biens d'un propriétaire, de son point de vue, afin de pouvoir toujours suivre visuellement et efficacement les statuts des locations.

## Implémentation des différents endpoints

Afin d'implémenter chacun des endpoints, il a fallu veiller à ce que nos références correspondent bien aux endpoints exposés par le groupe base de données & persistance.

Pour cela, dans l'implémentation, chacune des entités est associée à une classe Controller spécifique, dans laquelle nous avons implémentés toutes les requêtes de cette entité telles qu'elles sont listées dans notre swagger présenté auparavant.

⇒ Nous avons donc les controllers suivants :

- ProfilesController
- PropertiesController
- ReservationsController
- ReviewsController
- QueriesController.

Comme on peut le remarquer chaque controller correspond à l'entité associé, sauf QueriesController. En effet, ce controller permet de réaliser des "queries composées", autrement dit si l'on souhaite avoir accès à des informations plus poussées pour chacune des données cela sera possible.

Par exemple, il va permettre de rechercher toutes les propriétés réservées qui appartiennent à un propriétaire. Ou bien, réaliser des recherches sur le statut (annulé, confirmé, en attente) des réservations qui, de base n'est pas enregistrée directement dans la base mais nécessite une certaine reflexion, comme nous l'avons exposé auparavant.

```

@RestController
@Tag(name = "Reservations", description = "This API allows to carry out different operations on the reservations made by the renters")
@RequestMapping("/api/reservations")
public class ReservationsController {
    @Value("${databaseService.url}/reservations")
    private String URL;

    // Singleton instances of ResponseGenerator
    private ResponseGenerator<List<Reservation>> listGenerator;           // to send out JSON array in Response
    private ResponseGenerator<Reservation> generator;                         // to send out JSON object in Response

    @PostConstruct
    @Autowired
    private void init() {
        listGenerator = new ResponseGenerator<>();
        generator = new ResponseGenerator<>();
    }

    @GetMapping
    @Operation(summary = "This endpoint allows to retrieve all reservations made by renters")
    public ResponseEntity<List<Reservation>> getReservations() {
        ResponseEntity<List<Reservation>> result = listGenerator.buildRequest(URL, HttpMethod.GET, new ParameterizedTypeReference<List<Reservation>>() {});
        return result;
    }

    @GetMapping("/{id}")
    @Operation(summary = "This endpoint allows to display one of the bookings by providing the relevant reservation id")
    public ResponseEntity<Reservation> getReservation(@PathVariable String id) {
        ResponseEntity<Reservation> result = generator.buildRequest(URL.concat("/") + id, HttpMethod.GET, new ParameterizedTypeReference<Reservation>() {});
        return result;
    }
}

```

Aperçu de la classe ReservationsController dans le backend

Tout d'abord, pour spécifier à Spring que notre classe est un controller (basé sur REST), nous y ajoutons l'annotation `@RestController`.

Ensuite, la deuxième étape consiste à créer les Endpoints. Pour cela, il nous a suffit de créer des méthodes publiques java classiques, qui font les traitements escomptés et retournent les types d'éléments attendus par l'API.

Par exemple, pour récupérer toutes les réservations, nous avons créé une méthode `"getReservations"` qui retourne `"result"` qui est une liste contenant des éléments de type `"Reservation"`.

Enfin, il nous reste à définir le chemin et le type de la requête que l'on souhaite implémenter par le biais de cet Endpoint. Par exemple, pour `"getReservations"`, qui est une requête de type `"GET"`, nous rajoutons avant la signature de la méthode l'annotation `@GetMapping`.

Pour le chemin, nous pouvons prendre l'exemple du deuxième endpoint présenté dans la capture d'écran précédente: `getReservation`. En effet, dans les premières lignes de la classe, on remarque l'annotation `@RequestMapping` pour que tous les endpoints soient rattachés à `"/api/reservations"`. On comprend désormais, avec cet exemple, plus aisément l'annotation `@GetMapping("/{id}")` qui spécifie que cet endpoint devra être rattaché au chemin `/api/reservations/id`.

Enfin, pour indiquer à Spring que cette requête attend une variable `"id"` en entrée et qu'elle doit être passée par le biais de l'url, nous utilisons l'annotation `@PathVariable` dans la signature devant le paramètre d'entrée.

## Bilan & axes d'améliorations

Les principales difficultés rencontrées ont été de comprendre exactement quelles sont les fonctionnalités requises pour le frontend, ou encore celles qui sont absolument nécessaires. Il a donc

fallu faire preuve d'une très bonne coordination avec l'équipe frontend, et comprendre l'ensemble des éléments clés au bon fonctionnement du site.

L'autre souci a été celui avec CORS pour lequel il fallait rechercher la bonne configuration dans un serveur avec Spring Boot 3.

Enfin, la dernière contrainte était de réussir à concevoir l'implémentation de vérification du token fournie par le frontend au backend avant de traiter la requête

Malgré ces nombreux obstacles, nous sommes restés dans une cohérence d'équipe concernant l'implémentation des endpoints

Comme toutes implémentations, la nôtre nécessite aussi des améliorations. Celles que nous pouvons entrevoir, de notre côté, seraient :

- l'ajout d'un tri sur les résultats des requêtes comme par exemple si l'on recherche des propriétés avec un certain filtre, de pouvoir les représenter dans un ordre alphabétique ou alphanumérique;
- permettre aux utilisateurs d'obtenir des recommandations s'ils ne savent pas précisément ce qu'ils recherchent. La mise en œuvre d'un système de recommandation n'est pas une tâche facile et nécessiterait des recherches de librairies de systèmes de recommandations disponibles.

Malheureusement, nous n'avons pas eu le temps d'implémenter ces fonctionnalités.

On vient de réaliser une explication détaillée de l'ensemble de la partie non visible par l'utilisateur, mais qui garantit une grande partie de la cohérence de ses actions. désormais, passons à la concrétisation visuelle d'un gros travail : le front-end.

### III. La partie visible de l'iceberg : le front-end

L'interface de notre application de location de logement basée sur l'architecture des microservices doit offrir une expérience utilisateur fluide et personnalisable. Les locataires et les propriétaires doivent être en mesure de trouver facilement des logements disponibles selon leurs besoins.

La répartition des tâches au sein de notre équipe frontend est la suivante :

Tâches à effectuer	Ratiba	Euphraïm	Quang Viet
Elaboration de la maquette	✓	✓	✓
Implémentation de la page des propriétés (toutes & celles réservées)		✓	✓
Implémentations des réservations		✓	✓
Implémentation des reviews		👤	👤

Tâches à effectuer	Ratiba	Euphraïm	Quang Viet
Déploiement (Jenkins, Docker..)		✓	✓

→ Comme le temps imparti s'écoulait rapidement certaines fonctionnalités ont été élaborées en collaboration avec d'autres volontaires du groupe : Jessy & Meryem ⇒  : indique la partie concernée

Nous avons choisi d'utiliser des technologies modernes tel que Angular (framework complet pour le développement web), avec des intégrations de Bootstrap pour la conception de notre interface utilisateur.

## Conception : choix de conception UI/UX et présentation de la maquette

La conception d'une application de location de logements est un projet ambitieux qui nécessite une planification minutieuse pour offrir une expérience utilisateur agréable et intuitive. Dans ce développement, nous allons décrire les différentes fonctionnalités que nous avons implémenté pour notre application de location de logements. Nous avons utilisé l'outil Balsamiq pour concevoir la maquette de l'application, et ainsi avoir une vue d'ensemble de l'interface utilisateur.

Pour concevoir cette application, nous nous sommes inspirés des plateformes des locations existantes tels que Airbnb, Booking, etc. C'est pourquoi nous avons choisi de présenter immédiatement la liste des logements disponibles dès que l'utilisateur accède à la page d'accueil du site, sans nécessiter de connexion préalable pour voir les logements.

Commençons par la maquette de la page d'accueil :



Maquette préalable du site : page d'accueil

La page d'accueil de l'application comporte le logo à gauche et, à droite, le bouton d'appel à l'action pour inciter les utilisateurs à se connecter. En dessous se trouve une barre de recherche qui permet aux utilisateurs de rechercher des logements en fonction de leur destination (ville, pays).

Les utilisateurs peuvent également spécifier les dates de début et de fin de leur séjour ainsi que le nombre de personnes pour lesquelles ils recherchent un logement. De cette manière, seuls les logements disponibles pour les dates et le nombre de personnes spécifiés seront affichés, permettant une recherche plus précise. Enfin, un élément de filtrage des résultats est disponible pour permettre aux utilisateurs d'affiner leur recherche.

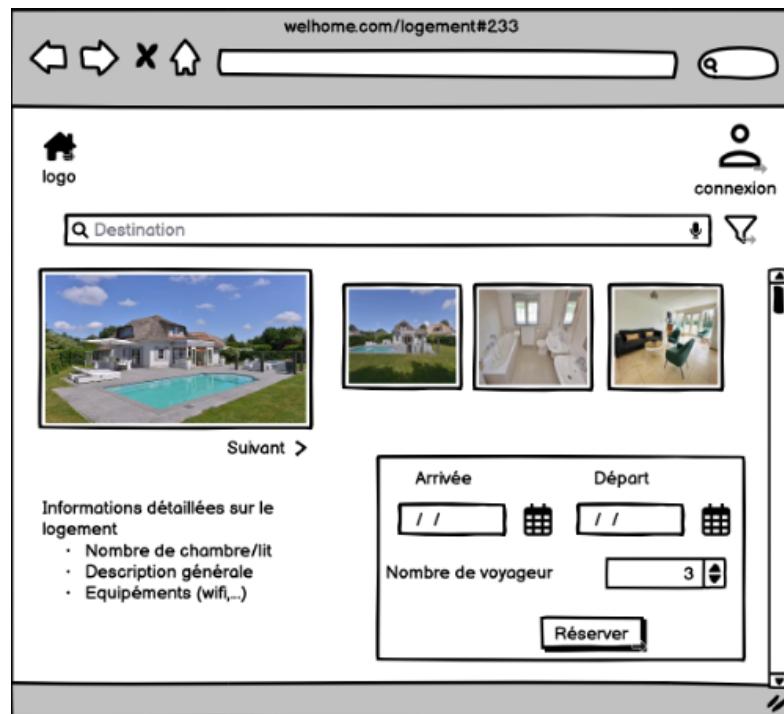
Lorsque les utilisateurs cliquent sur l'élément de filtrage, plusieurs critères sont proposés pour affiner leur recherche :

- Un filtre pour le type de logement, tels que maisons ou appartements.
- Un filtre pour les équipements disponibles, tels que la connexion Wi-Fi, la climatisation, etc.
- Un filtre pour les prix, permettant aux utilisateurs de rechercher des logements dans leur budget.



Maquette préalable du site : filtrage

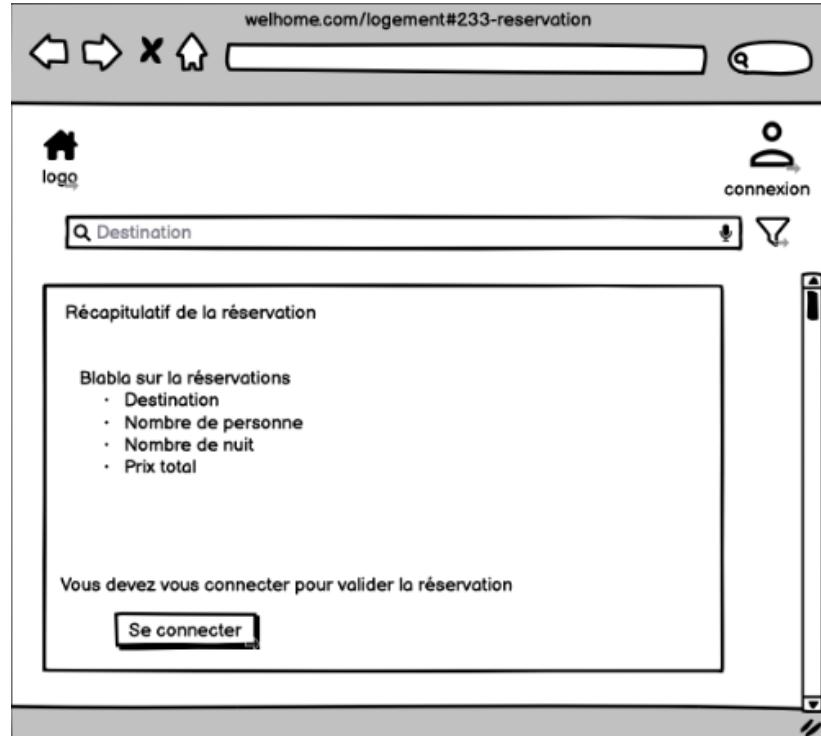
Lorsqu'un utilisateur clique sur un logement, il est dirigé vers une page de détails du logement. Cette page fournit des informations détaillées sur le logement.



Maquette préalable du site : réservation

En effet, il est possible pour les utilisateurs de visualiser les photos et les détails de chaque logement, tels que le nombre de chambres, la superficie, les commentaires et les évaluations des autres utilisateurs. Les utilisateurs peuvent également réserver directement un logement depuis la page de détails du logement, en indiquant les dates de séjour et le nombre de personnes.

Une fois que l'utilisateur saisit toutes les informations requises et clique sur le bouton "Réserver", l'application le redirige vers une page de récapitulation des informations. Si l'utilisateur n'est pas connecté, l'application l'invite à se connecter pour valider sa demande de réservation.



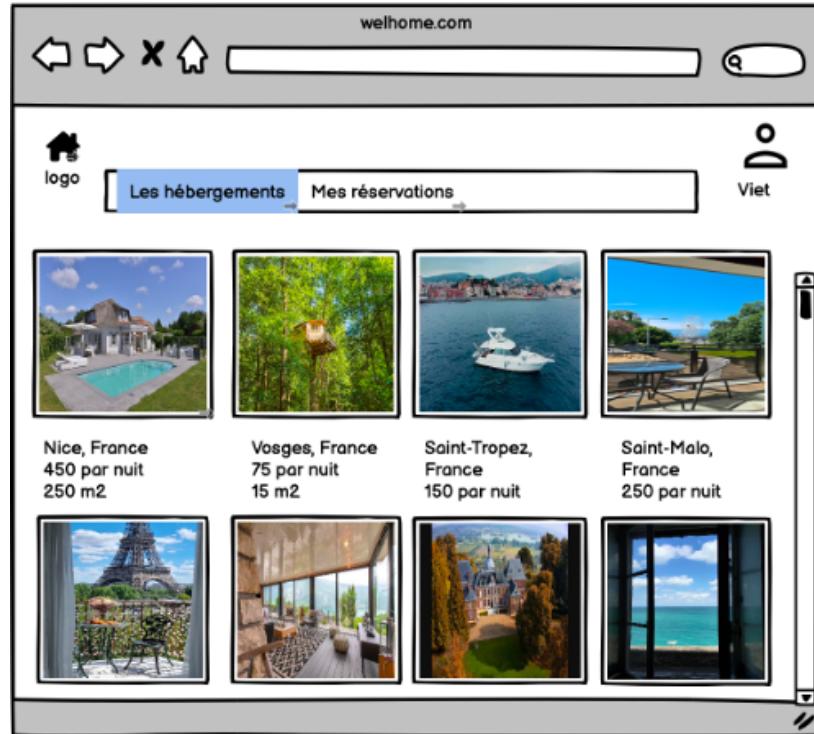
Maquette préalable du site : bilan de la réservation

En cliquant sur « Se connecter », l'utilisateur est redirigé vers une page de connexion. Cette étape se fait uniquement si l'utilisateur n'est pas connecté.

Si celui-ci est connecté, il ne verra qu'un bouton « valider » qui lui permet de valider sa demande de réservation.

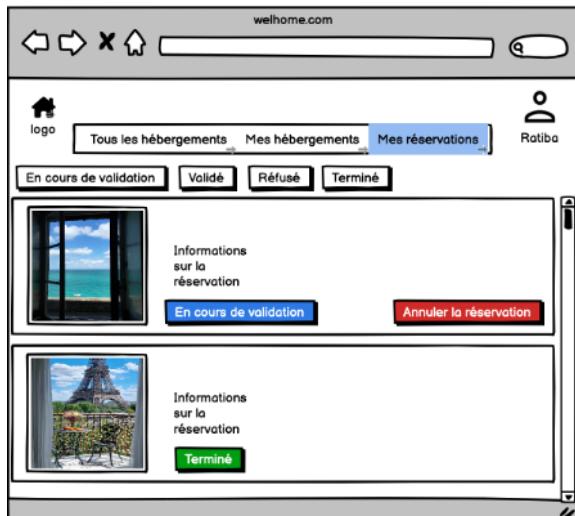
Lorsqu'un utilisateur se connecte, il sera face à une interface selon son statut :

- **Locataire** : il aura accès à seulement deux onglets de navigation



Maquette préalable du site : page "Les hébergements"

→ Le premier onglet est "Tous les hébergements" où il pourra visualiser l'ensemble des hébergements disponibles en ligne, tandis que le second est "Mes réservations" qui lui permettra de suivre l'état de ses demandes de réservation ainsi que leur historique. Voici un aperçu de l'onglet "Mes réservations" :

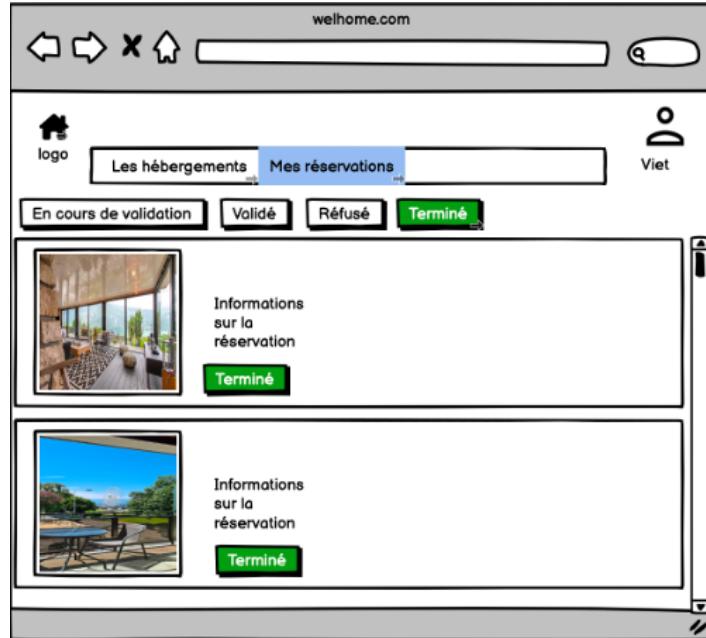


Maquette préalable du site : page "Mes réservations" | 1



Maquette préalable du site : page "Mes réservations" | 2

Comme on peut le remarquer, il est également possible de trier les réservations en fonction de leur état, qui peut être "En cours de validation", "Validé", "Refusé", "Terminé" ou "Annulé".



Maquette préalable du site : page "Mes réservations" | 3

- **Propriétaire** : il aura accès à 3 onglets : les deux précédemment évoqués, ainsi qu'un onglet supplémentaire intitulé "Mes hébergements". Via cet onglet, il pourra mettre à jour les informations concernant ses logements et également ajouter ou supprimer des logements

Les deux maquettes montrent l'interface pour les propriétaires. La gauche montre "Tous les hébergements" avec une liste de destinations et leurs détails. La droite montre "Mes hébergements" avec deux cartes détaillées d'un logement, chacune avec un bouton "Voir details".

→ On remarque que l'on retrouve l'onglet "Mes réservations". Ce dernier est présenté de manière similaire pour tous les utilisateurs, qu'ils soient locataires ou propriétaires. Cette section permet d'afficher une liste de toutes les réservations effectuées, avec la possibilité de les annuler ou non.

⇒ En élaborant la maquette de notre application, nous avons eu l'avantage de pouvoir observer l'interface utilisateur avant d'entamer la phase de codage. Cette démarche nous a permis d'identifier les points faibles et de les corriger en amont, ce qui nous a permis d'optimiser notre développement.

## Implémentation

### Fonctionnalités

L'interface doit permettre aux locataires et aux propriétaires de :

- Consulter la liste des logements disponibles dès la page d'accueil
- Rechercher des logements en fonction de la destination, des dates et du nombre de personnes avec **la barre de recherche**
- Affiner la recherche des hébergements (en fonction du type, des équipements et du prix) avec l'élément de **filtrage**
- Consulter les informations plus détaillées de chaque logement avant de réserver (l'emplacement, les notes et commentaires, des photos etc)
- Visualiser le récapitulatif de la réservation avant de confirmer la réservation
- Consulter l'historique de réservation (les réservations en cours de validation, ceux validé, terminé, annulé ou encore refusé)
- Annuler une réservation (dans le cas où la réservation a été validé par le propriétaire ou est en cours de validation). Si la réservation est classée comme terminé ou refusé, celle-ci ne peut être annulé
- Ajouter les commentaires et la note sur un hébergement que l'on a loué (dans le cas où la réservation est classée comme terminé)

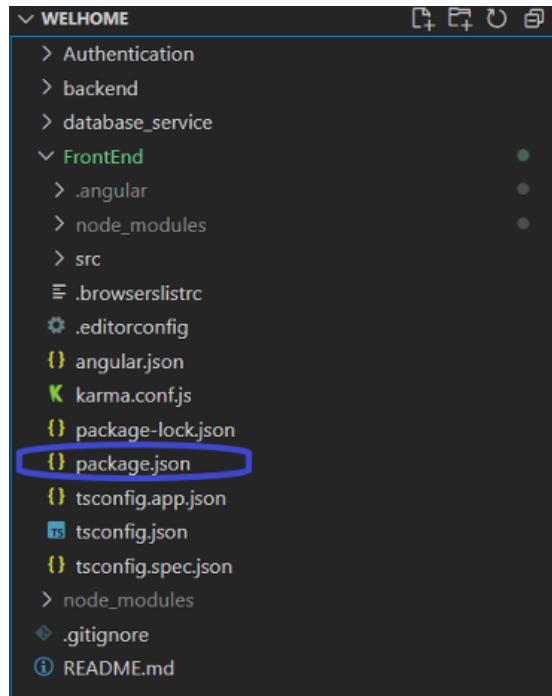
De plus, elle doit permettre aux propriétaires de :

- Ajouter, modifier, supprimer les logements qu'ils ont mis en ligne

### Les fonctionnalités clés d'Angular utilisées dans notre application

#### Fichier de conf des modules

Dans un projet Angular, les modules Node.js sont définis dans le fichier *package.json*



Emplacement du fichier package.json

Ce fichier contient une section *dependencies* qui spécifie les dépendances nécessaires pour faire fonctionner l'application. Ces dépendances sont des bibliothèques tierces et des modules Angular. Et une section *devDependencies* qui contient les dépendances nécessaires pour développer et tester l'application.

Parmi les modules utilisés nous pouvons citer :

- *@angular/platform-browser-dynamic*: contient les fonctionnalités pour le développement d'applications web sur navigateur qui sont dynamiques.
- *@angular/router*: permet de gérer la navigation entre les différentes pages de l'application.
- *"tslib"*: est une bibliothèque pour aider à écrire des fichiers TypeScript plus concis.
- *"rxjs"*: est une bibliothèque pour programmer de manière *réactive*.

## Les composants

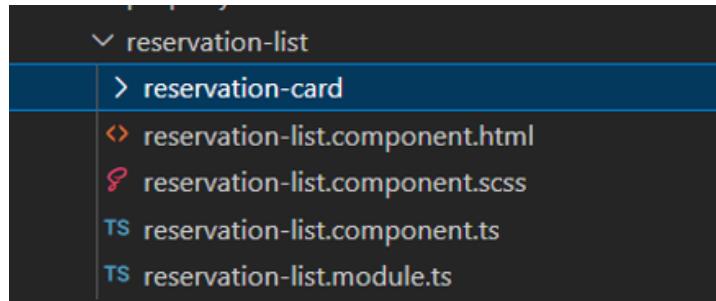
Les composants sont l'une des principales briques de construction d'une application Angular. Ceux-ci permettent de découper l'interface utilisateur en plusieurs morceaux réutilisables, ce qui facilite la maintenance et l'évolutivité de l'application.

Chaque composant encapsule une partie de la logique métier et une partie de l'interface utilisateur (HTML, CSS).

Les composants ont un cycle de vie qui leur permet de réagir à des événements tels que la création, la mise à jour et la destruction. Ils peuvent également recevoir des données en entrée (via des propriétés) et émettre des événements en sortie (via des événements personnalisés), ce qui permet de communiquer avec d'autres composants.

Enfin, les composants sont définis dans des fichiers TypeScript qui contiennent leur code ainsi que leurs métadonnées (comme leur nom, leur sélecteur et leur template). Ils sont ensuite utilisés dans d'autres fichiers TypeScript (les modules) qui regroupent les composants et les autres fonctionnalités de l'application.

Un composant peut contenir des sous composants ce qui permet de créer une hiérarchie de composants. Un exemple concret est le composant "reservation-list" qui englobe un sous-composant appelé "reservation-card".



Architecture du code liée aux réservations

C'est le composant Reservation-list qui va contenir l'ensemble des réservations effectuées, et renvoyer vers un reservation-card quand il est souhaité par l'utilisateur

## Composant Reservation-list

Le composant "reservation-list", quant à lui, est chargé d'afficher toutes les réservations en utilisant des instances du composant "reservation-card". En d'autres termes, il utilise une boucle pour créer plusieurs instances du composant "reservation-card", chacune représentant une réservation différente, afin de les afficher toutes dans une liste.

Reservation Type	Location	Area	Arrival Date	Departure Date	Status	Your rating	Actions
House	Paris	100 m <sup>2</sup>	11/04/2023	13/04/2023	Cancelled	No rating	<a href="#">Review</a> <a href="#">Details</a>
Apartment	Toronto	80 m <sup>2</sup>	12/04/2023	19/04/2023	Pending	3.5	<a href="#">Cancel</a> <a href="#">Details</a>

Aperçu de la page "Mes réservations" globale

## Composant Reservation-card

Le composant Reservation-card est spécifiquement conçu pour afficher une carte contenant des informations précises en utilisant les fichiers HTML et CSS qui lui sont associés. Sa fonction principale consiste à présenter les données relatives à une réservation de manière claire et lisible, en suivant les normes de présentation et de design définies pour le composant (html, css).

En voici un aperçu :



Aperçu d'une réservation

## Routage

Le routage permet de naviguer entre différentes vues ou pages en fonction de l'URL cette étape est essentielle pour créer des applications web à pages multiples en utilisant une architecture de type SPA (Single Page Application).

Le système de routage d'Angular utilise un module spécifique appelé RouterModule, qui permet de définir les différentes routes de l'application. Celle-ci sont définies dans un tableau de configurations, où chaque route est définie avec un chemin d'URL et un composant associé.

```

1 import { NgModule } from '@angular/core';
2 import { RouterModule, Routes } from '@angular/router';
3 import { AuthenticationGuard, ContextGuard } from './app.guard';
4 import { HostPropertyListComponent } from './host-property-list/host-property-list.component';
5 import { PropertiesComponent } from './properties/properties.component';
6 import { PropertyListComponent } from './property-list/property-list.component';
7 import { ReservationListComponent } from './reservation-list/reservation-list.component';
8
9 const routes: Routes = [
10   { path: '', component: PropertyListComponent },
11   { path: 'properties/:id', component: PropertiesComponent },
12   { path: 'myproperties',
13     component: HostPropertyListComponent,
14     canActivate: [ AuthenticationGuard, ContextGuard ],
15     runGuardsAndResolvers: 'always',
16   },
17   { path: 'myreservations',
18     component: ReservationListComponent,
19     canActivate: [ AuthenticationGuard ],
20     runGuardsAndResolvers: 'always' },
21   { path: '**', redirectTo: '' }
22 ];
23
24 @NgModule({
25   imports: [RouterModule.forRoot(routes, { onSameUrlNavigation: 'reload' })],
26   exports: [RouterModule]
27 })
28 export class AppRoutingModule { }
29

```

Code illustrant l'implémentation des routes

Une fois les routes définies, nous pouvons utiliser les liens de navigation d'Angular pour naviguer entre les différentes pages.

→ Par exemple pour naviguer vers la page « All properties » on utilise :

routerLink= « /properties » qui spécifie la route et routerLinkActive qui permet de mettre en actif le bouton associé.

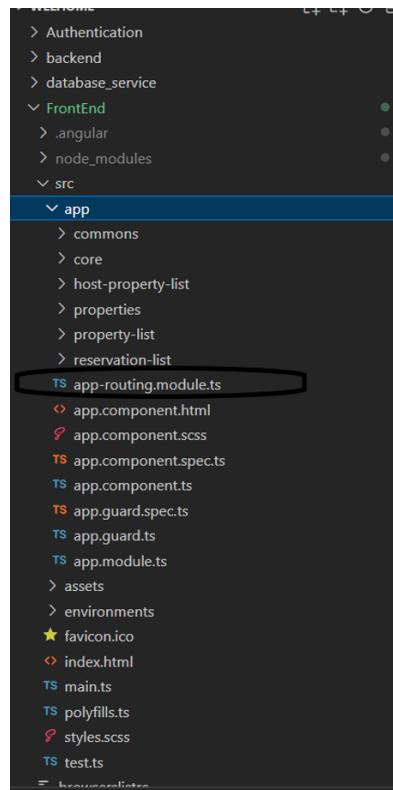
```

<div>
  <button type="button" class="btn btn-primary" routerLink="/properties" routerLinkActive="active" ariaCurrentWhenActive="page">All Properties</button>
  <button type="button" class="btn btn-primary" routerLink="/myreservations" routerLinkActive="active" ariaCurrentWhenActive="page" *ngIf="this.isLoggedIn">My Reservations</button>
  <button type="button" class="btn btn-primary" routerLink="/myproperties" routerLinkActive="active" ariaCurrentWhenActive="page" *ngIf="this.isLoggedIn">My Properties</button>
  <button type="button" class="btn btn-primary" (click)="this.login()" *ngIf="!this.isLoggedIn">Login</button>
  <button type="button" class="btn btn-danger" (click)="this.logout()" *ngIf="this.isLoggedIn">Logout</button>
</div>

```

Partie de code exposant les routages implémentés

⇒ La configuration des routes se trouve dans le fichier `app-routing.module.ts` ici présent :

Emplacement du fichier `app-routing.module.ts`

## Guards

Les guards permettent de protéger les routes de l'application en fonction de certains critères.

```

import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';
import { AuthService } from './core/auth/auth.service'

@Injectable({
  providedIn: 'root'
})
export class AppGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree | Promise<boolean | UrlTree> | boolean | UrlTree {
    return this.authService.isLoggedIn;
  }

  constructor(
    private authService: AuthService,
  ) {}

}
  
```

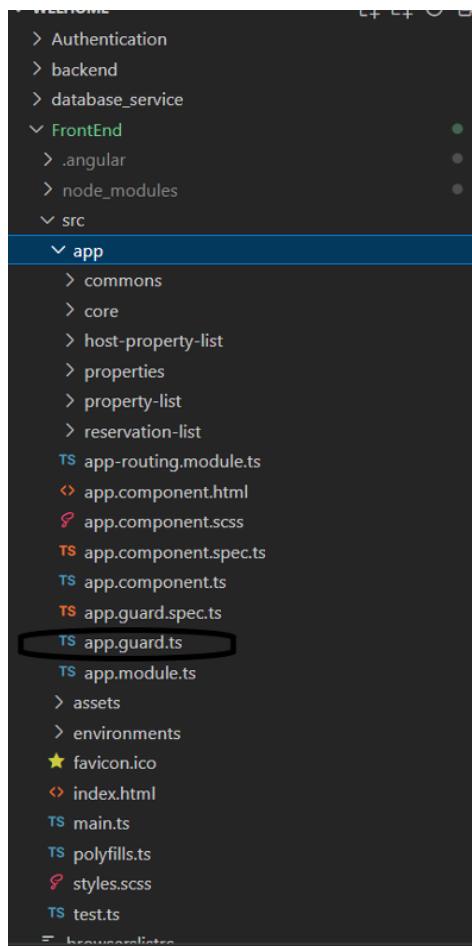
Illustration des Guards

Pour notre site, nous avons utilisés le guard `canActivate` qui permet de déterminer si un utilisateur est autorisé à accéder à une route ou non.

→ Dans notre cas c'est pour vérifier si l'utilisateur est connecté, et s'il est connecté, de vérifier si c'est uniquement un locataire ou aussi un propriétaire.

L'utilisation des guards nous permet de mettre en place une sécurité granulaire dans l'application, en accordant différents niveaux d'accès en fonction des permissions de chaque utilisateur.

⇒ Le voici entouré au sein des fichiers de notre projet :



Emplacement du fichier "app.guard.ts"

De plus, le fichier `environnement.ts` contient les variables d'environnement spécifiques à l'environnement de développement de notre application. Nous l'avons utilisé pour stockées les URL d'API backend et de l'authentification.

```
export const environment = {
  production: false,
  authUrl: 'https://backend.zain.ovh', // 'http://localhost:3001',
  backEndUrl: 'http://zain.ovh:9092/api' // 'http://localhost:9092/api',
};
```

```
export const environment = {
  production: false,
  authUrl: 'https://backend.zain.ovh', // 'http://localhost:3001',
  backEndUrl: 'http://zain.ovh:9092/api' // 'http://localhost:9092/api',
};
```

```

export const environment = {
  production: false,
  authUrl: 'https://backend.zain.ovh', //http://localhost:3001',
  backEndUrl: 'http://zain.ovh:9092/api' //'http://localhost:9092/api',
};

```

Variables d'environnement dans le fichier "environnement.ts"

## Intégration avec d'autres services de l'application

Dans le cadre de ce projet, le frontend n'a eu à intéragir qu'avec le service backend et le service d'authentification. Le répertoire « core » contient tous les fichiers relatifs à la communication avec des services.

### Intégration des fonctions backend pour une application web stable et performante

Les informations utilisées par notre application sont extraites d'une base de données (cf. partie qui parle de la base des données). Pour des raisons de sécurité et de performances, il est recommandé de ne pas permettre au frontend d'avoir un accès direct à la base des données. C'est la raison pour laquelle nous avons utilisés une couche supplémentaire, le backend.

Le backend est responsable de la gestion des opérations techniques, tandis que le frontend est chargé d'afficher les résultats à l'utilisateur.

Nous avons fait appels aux fonctions du backend pour diverses raisons :

- Les résultats de recherche et du filtre
- L'affichage des propriétés, des réservations, des notes/commentaires
- L'ajout des propriétés, des réservations, des notes/ commentaires
- La mise à jour des propriétés et des réservations

Pour exploiter les fonctionnalités implémentées par le backend, nous avons dans un premier temps fait appels aux services utiles au frontend (ces appels se trouvent dans des fichiers services.ts). Puis nous avons utilisés des interfaces afin des déserialiser les données. En effet les données envoyées par le back est au format JSON et pour pouvoir correctement les utiliser, il a fallu qu'on les transforme en objet.

Pour clarifier davantage, nous allons illustrer notre explication avec un exemple concret : la gestion des réservations.

Dans un premier temps nous définissons les propriétés d'une réservation dans une interface. Ces propriétés doivent être en cohérences avec ceux de la base des données.

```
src/app/core/reservation/reservation.model.ts
1  export interface IReservation {
2      id: number,
3      propertyId : number,
4      renterEmail : string,
5      startDate : Date,
6      endDate : Date,
7      confirmedOwner : boolean,
8      confirmedRenter : boolean,
9      totalPrice : number
10 }
```

```
src/app/core/reservation/reservation.model.ts
1  export interface IReservation {
2      id: number,
3      propertyId : number,
4      renterEmail : string,
5      startDate : Date,
6      endDate : Date,
7      confirmedOwner : boolean,
8      confirmedRenter : boolean,
9      totalPrice : number
10 }
```

Interface contenant les propriétés d'une réservation dans "reservation.model.ts"

Nous avons implémenté plusieurs méthodes, cependant nous ne présenterons dans ce rapport que celles liées aux opérations CRUD.

## CREATE

La méthode addReservation utilise la méthode http POST pour envoyer la nouvelle reservation au serveur, en utilisant l'API backend spécifiée dans la variable «backEndUrl» (definie dans le fichier environnement.ts). Puis met à jour la liste des reservations avec l'appel à la méthode getReservation.

```
src/app/core/reservation/reservation.service.ts
1  addReservation(reservation: IReservation): Observable<IReservation> {
2      return this.http.post<IReservation>(`${environment.backEndUrl}/reservations`, reservation)
3          .pipe(
4              map((reservation) => {
5                  this.getReservations();
6                  this.toastService.showSuccess('Reservation added successfully');
7                  return reservation;
8              })
9          );
10 }
```

```

    addReservation(reservation: IReservation): Observable<IReservation> {
      return this.http.post<IReservation>(`${environment.backEndUrl}/reservations`, reservation)
        .pipe(
          map((reservation) => {
            this.getReservations();
            this.toastService.showSuccess('Reservation added successfully');
            return reservation;
          })
        );
    }
  
```

Aperçu de la méthode getReservation dans *reservation.service.ts*

## READ

La méthode getReservation() effectue une requête http GET vers l'url qui contient la liste des réservation en fonction de l'adresse mail du locataire (qui est connecté).

```

getReservations(): ReservationService {
  if (this.reservationSubscription) {
    this.reservationSubscription.unsubscribe();
  }
  this.reservationLoadingSubject.next(true);
  this.reservationSubscription = this.http.get<IReservation[]>(`${environment.
    backEndUrl}/reservations/renter_email/${this.authService.profile!.email}`)
    .subscribe({
      next: (reservations) => {
        this.reservationSubject.next(reservations);
      },
      error: (error) => {
        this.reservationSubject.next([ ]);
      },
      complete: () => {
        this.reservationLoadingSubject.next(false);
      }
    });
  return this;
}
  
```

```

getReservations(): ReservationService {
  if (this.reservationSubscription) {
    this.reservationSubscription.unsubscribe();
  }
  this.reservationLoadingSubject.next(true);
  this.reservationSubscription = this.http.get<IReservation[]>(`${environment.
    backEndUrl}/reservations/renter_email/${this.authService.profile!.email}`)
    .subscribe({
      next: (reservations) => {
        this.reservationSubject.next(reservations);
      },
      error: (error) => {
        this.reservationSubject.next([ ]);
      },
      complete: () => {
        this.reservationLoadingSubject.next(false);
      }
    });
  return this;
}
  
```

Implémentation de getReservation dans "reservation.service.ts"

## UPDATE

Nous avons mis en place la fonction "updateReservation" en utilisant la méthode http PUT de l'API

```
updateReservation(reservation: IReservation): Observable<IReservation> {
  return this.http.put<IReservation>(`${environment.backEndUrl}/reservations/${reservation.id}`, reservation)
    .pipe(
      map((reservation) => {
        this.getReservations();
        this.getOwnerReservations();
        this.toastService.showSuccess('Reservation updated successfully');
        return reservation;
      })
    );
}
```

```
updateReservation(reservation: IReservation): Observable<IReservation> {
  return this.http.put<IReservation>(`${environment.backEndUrl}/reservations/${reservation.id}`, reservation)
    .pipe(
      map((reservation) => {
        this.getReservations();
        this.getOwnerReservations();
        this.toastService.showSuccess('Reservation updated successfully');
        return reservation;
      })
    );
}
```

Développement de la méthode updateReservation dans "reservation.service.ts"

## DELETE

Comme son nom l'indique, la fonction "deleteReservation" permet de supprimer une réservation en utilisant la méthode http DELETE et en fournissant l'identifiant de la réservation à supprimer.

```
deleteReservation(reservation: IReservation): Observable<IReservation> {
  return this.http.delete<IReservation>(`${environment.backEndUrl}/reservations/${reservation.id}`)
    .pipe(
      map((reservation) => {
        this.getReservations();
        this.getOwnerReservations();
        this.toastService.showSuccess('Reservation deleted successfully');
        return reservation;
      })
    );
}
```

```
deleteReservation(reservation: IReservation): Observable<IReservation> {
  return this.http.delete<IReservation>(`${environment.backEndUrl}/reservations/${reservation.id}`)
    .pipe(
      map((reservation) => {
        this.getReservations();
        this.getOwnerReservations();
        this.toastService.showSuccess('Reservation deleted successfully');
        return reservation;
      })
    );
}
```

Implémentation de la méthode deleteReservation dans "reservation.service.ts"

⇒ Peu importe la méthode utilisée, nous pouvons constater que nous suivons ces étapes :

- On effectue une HTPP vers l'API backend en utilisant la méthode http appropriée (POST, GET, PUT, DELETE)

- On génère la réponse http en utilisant des opérateurs d'observables tels que « map », "subscribe", "catchError", etc
- On met à jour les données de l'application en conséquence, en utilisant:
  - Des variables de sujet qui contiennent les données en entrée (par exemple le prix de la location)
  - Des variables de comportement qui représente l'action que l'application doit effectuer (par exemple mettre à jour un logement)
  - ou d'autres mécanismes de gestion de l'état qui sont là pour gérer l'état actuel de l'application. Par exemple lorsque l'application affiche à temps réel une notification, suite à une suppression.
  - On affiche un message de feedback visuel à l'utilisateur en utilisant le service de notification tel que *toastService*.

## Intégration des fonctionnalités du service d'authentification

Nous avons dû créer une interface et un fichier service.ts qui contient tous les appels au service d'authentification, tout comme pour l'intégration avec le service backend.

De plus, nous avons utilisé l'intercepteur pour ce service. Un intercepteur est un middleware qui permet de modifier les requêtes http entrantes ou sortantes.

Dans notre cas, il est utilisé pour ajouter un entête d'autorisation à toutes les requêtes sortantes.

```
intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
  const authToken = this.authService.token;
  const authRequest = request.clone({
    headers: request.headers.set('Authorization', 'Bearer ' + authToken).set('Content-Type', 'application/json'),
  });
  return next.handle(authRequest);
}
```

Implémentation dans "auth.interceptor.ts"

Pour utiliser le service d'authentification, l'application suit plusieurs étapes:

- La récupération du jeton d'accès
- Le clonage de la requête. Cette copie est nécessaire pour ajouter l'entête d'autorisation sans modifier la requête d'origine
- L'ajout de l'entête d'autorisation
- La transmission de la requête modifiée

Outre la mise en place de l'intercepteur, nous avons également développé des méthodes permettant une communication directe avec le service d'authentification

### Démarrer une session

La fonction login() est utilisée pour initier le processus d'authentification via l'API de Google. Voici les étapes de l'implémentation

- Ouverture d'une fenêtre de navigateur en utilisant la méthode window.open() pour accéder à l'URL d'authentification de Google.
- Attente d'une réponse de la fenêtre ouverte en ajoutant un écouteur d'événement window.addEventListener().
- Extraction des données reçues de l'événement et vérification de la présence d'un jeton d'accès (token) et d'une adresse e-mail valide.
- Stockage des informations d'authentification localement dans la mémoire du navigateur en utilisant la méthode localStorage.setItem().
- Affichage d'un message de succès à l'aide de la bibliothèque toast pour indiquer que l'utilisateur est connecté.
- Fermeture de la fenêtre d'authentification en appelant la méthode close() sur l'objet loginWindow.

```
login(): void {
  const loginWindow = window.open(`${environment.authUrl}/auth/google`, 'Authentication', 'height=800,width=600');
  if (loginWindow !== null) {
    //loginWindow.focus();
    window.addEventListener('message', event => {
      if (event.source === loginWindow) {
        return;
      }
      const data = event.data.data;
      if (data.access_token !== undefined && data.email !== undefined) {
        localStorage.setItem('token', data.access_token);
        localStorage.setItem('email', data.email);
        this.toast.showSuccess('Logged in');
      } else {
        this.toast.showError('Login failed');
      }
      loginWindow.close();
    }, { once: false });
  }
}
```

Développement dans "Auth.service.ts"

## Arrêter une session

La méthode logout() est utilisée pour déconnecter l'utilisateur et supprimer les informations d'authentification stockées localement dans le navigateur. Puis l'application redirige l'utilisateur vers la page principale.

```
logout(): void {
  this.toast.showInfo('Logged out');
  this.contextService.setContext('RENTER');
  localStorage.removeItem('token');
  localStorage.removeItem('email');
  this.router.navigate(['/']);
}
```

Aperçu de la partie dans "Auth.service.ts"

En dehors de celles mentionnées précédemment, nous avons également implémenté d'autres fonctions importantes utile à l'authentification:

- `startupToken()` : récupère le token d'authentification stocké en localStorage au démarrage de l'application.
- `getProfile()` : renvoie un objet contenant l'adresse e-mail de l'utilisateur si elle est stockée en localStorage, sinon elle renvoie null
- `getToken()` : renvoie le token d'authentification stocké en localStorage si présente, sinon elle renvoie null
- `getIsLoggedIn()` : renvoie un booléen indiquant si l'utilisateur est connecté ou non, en vérifiant si un token d'authentification est stocké en localStorage

## Bilan & axes d'améliorations

En somme, on peut en conclure que nous pensons avoir réussi à implémenter notre IHM correctement. L'ensemble des fonctionnalités que l'on souhaitait implémenter ont trouvé vie.

Evidemment, certaines améliorations sont toujours possibles. Par exemple, il pourrait être possible d'ajouter un choix de langues, autre que l'anglais qui est l'unique langue de notre site actuellement.

En même temps que notre site commençait à prendre forme, on a développé d'autres microservices essentielles à la bonne utilisation de notre site : à savoir l'authentification et le mailing.

## IV. Gérer les accès et autorisations à un site : le micro-service d'authentification

Le micro-service d'authentification que nous avons développé utilise l'authentification OAuth2 de Google pour permettre d'une part, aux utilisateurs de se connecter à notre application et d'autre part, de sécuriser les différents endpoints qui requièrent une connexion de la part de l'utilisateur. Avant de vous proposer une explication détaillée du fonctionnement de notre micro-service d'authentification, nous tenons à expliquer brièvement OAuth2.

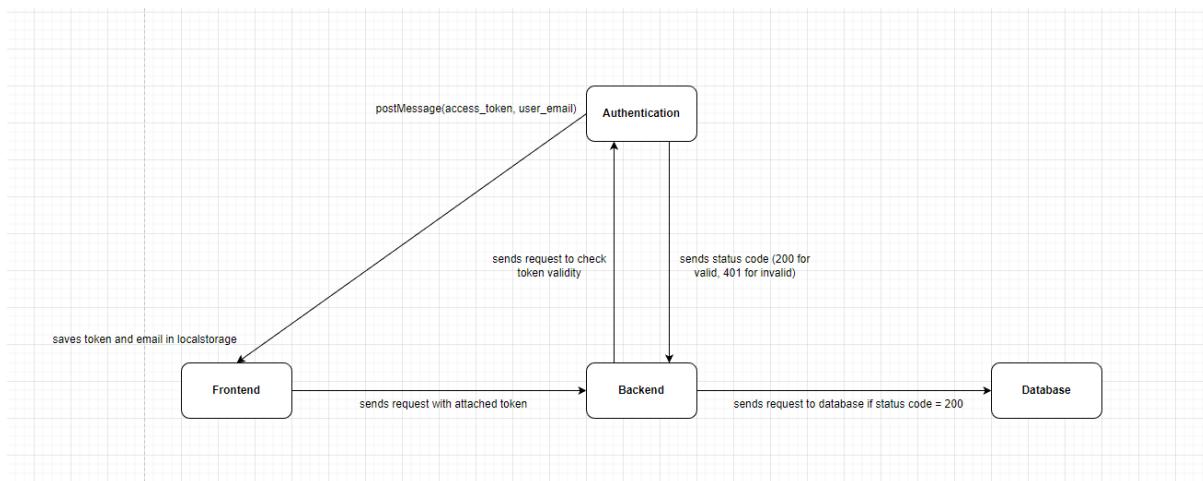
### Initiation à l'authentification avec OAuth2

#### OAuth2

OAuth2 est un système qui permet aux applications d'accéder aux informations d'un utilisateur sur des sites comme Google, Facebook ou Twitter sans révéler le mot de passe de l'utilisateur. Il donne à l'application un "jeton" (access token) temporaire pour accéder aux données de l'utilisateur avec des autorisations spécifiques. Ainsi, les utilisateurs contrôlent les données auxquelles les applications peuvent accéder et pour combien de temps, tout en protégeant leur mot de passe. OAuth2 est couramment utilisé pour la connexion et les permissions dans les applications modernes, car il offre une méthode sécurisée et uniforme pour gérer l'accès aux ressources en ligne.

Dans le cadre de notre projet, nous avons opté pour une authentification auprès du provider Google, c'est pourquoi nous parlerons seulement de ce dernier.

#### Diagramme fonctionnel high-level



Aperçu simplifié de la logique à atteindre

## Technologies employées

Comme nous avions décidé que le microservice d'authentification aurait la charge de proposer un formulaire d'inscription (que nous verrons par la suite) à un utilisateur n'ayant pas encore de compte dans notre base de données, il a donc fallu utiliser une technologie de frontend pour l'interface du formulaire, et une technologie de backend pour les opérations.

Voici les technologies employées :

- Frontend : React
- Backend : Node

## Implémentation

### Authentification auprès de Google

**Etape préliminaire** : enregistrer notre application dans Google Cloud Console pour récupérer un **ClientID** et un **ClientSecret** représentant les identifiants de notre application auprès de Google. C'est notamment via cette console que nous pouvons autoriser les différents endpoints à utiliser l'api Google pour l'OAuth2, en spécifiant les **URI de redirection** autorisés.

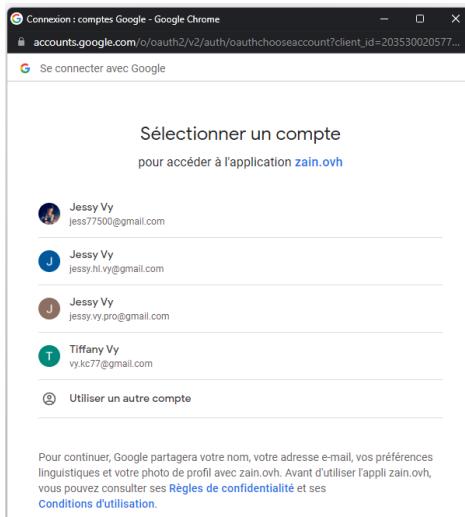
### URI de redirection autorisés ?

À utiliser avec les requêtes provenant d'un serveur Web

URI 1 *	<input type="text" value="https://backend.zain.ovh/auth/google/callback"/>
---------	--

Capture d'écran de l'interface pour les autorisations

Lorsque l'utilisateur clique sur le bouton "Login" dans l'interface utilisateur produite par le frontend, cela appelle l'endpoint **/auth/google** dans notre backend sur une nouvelle fenêtre (pop up), qui génère l'URL d'authentification de Google en utilisant l'**ID client** et l'**URI de redirection** que nous avons définis dans la console Google Cloud.



Redirection vers l'URL d'authentification de Google

L'utilisateur est invité à autoriser notre application à accéder à certaines informations de son compte Google, telles que son adresse e-mail et son profil. Après l'approbation de l'utilisateur, Google génère un code d'autorisation unique et le renvoie à notre application en redirigeant l'utilisateur vers l'URI de redirection que nous avons fournie.

## Récupération et vérification de l'access token

Une fois que notre application reçoit le code d'autorisation, elle envoie une requête POST à l'endpoint **/auth/google/callback** de notre backend. Notre backend échange ensuite le code d'autorisation contre un access token en envoyant une requête POST à l'endpoint <https://accounts.google.com/o/oauth2/token> de Google.

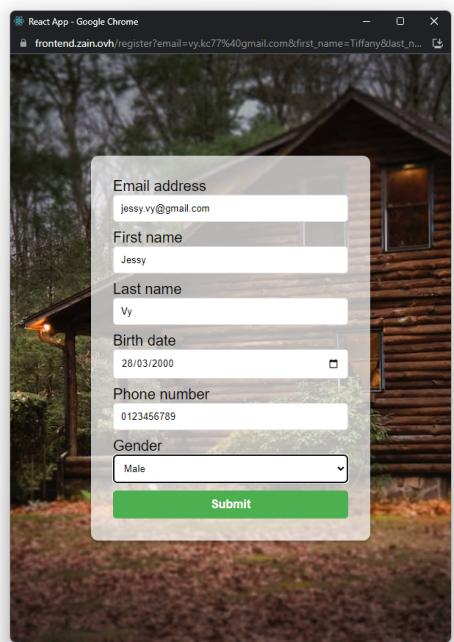
Notre backend vérifie également la validité de l'access token en envoyant une requête GET à l'endpoint <https://www.googleapis.com/oauth2/v1/userinfo> de Google, avec l'access token en tant qu'Authorization header.

## Vérification de l'existence de l'utilisateur dans la base de données

Après avoir vérifié la validité de l'access token, notre backend vérifie si l'utilisateur existe déjà dans notre base de données en envoyant une requête GET sur un endpoint de notre base de données permettant de récupérer un profil d'utilisateur grâce à son adresse mail. Si un utilisateur correspondant est trouvé, notre API REST renvoie un code de réponse ainsi que les informations de l'utilisateur sous forme de JSON.

## Gestion des utilisateurs existants et nouveaux

- Si l'utilisateur existe déjà dans la base de données, notre backend envoie un message au frontend contenant l'access token et l'adresse e-mail de l'utilisateur à l'aide de la méthode postMessage. Cela permettra au frontend de stocker ces deux variables dans le **localStorage** du navigateur.
- Si l'utilisateur n'existe pas encore dans la base de données, notre backend redirige l'utilisateur vers l'endpoint **/register** de l'interface frontend en y ajoutant en header les différentes informations basiques de l'utilisateur. Cela ouvre un formulaire pré-rempli avec les informations de base de l'utilisateur, telles que son adresse e-mail, son prénom et son nom. L'utilisateur peut ensuite saisir des informations supplémentaires, telles que sa date de naissance, son numéro de téléphone et son sexe, avant de soumettre le formulaire.



Aperçu du formulaire d'inscription

## Enregistrement des nouveaux utilisateurs

Lorsque l'utilisateur soumet le formulaire d'inscription, le frontend envoie une requête POST à l'endpoint **/register** de notre backend, avec les données du formulaire en tant que body de la requête et l'access token en tant que authorization header (Bearer token).

Le backend de notre microservice traite ensuite cette requête et envoie une requête POST à l'endpoint <http://zain.ovh:9090/api/profiles> de notre database avec les données rentrées dans le formulaire en tant que body de la requête pour créer un nouveau profil utilisateur dans la base de données.

De la même manière que lorsque l'utilisateur a déjà un compte dans notre base de données, nous envoyons l'access token et l'adresse email de l'utilisateur au frontend par un postMessage afin que ce dernier puisse stocker ces variables dans le localStorage.

## Finalisation du processus d'authentification

Une fois que le nouveau profil utilisateur a été créé avec succès, notre backend envoie un message au frontend contenant l'access token et l'adresse e-mail de l'utilisateur à l'aide de la méthode

postMessage. Le frontend peut alors stocker ces deux valeurs pour récupérer les informations de l'utilisateur et envoyer des requêtes au backend avec l'access token attaché.

## Rafraîchissement de l'access token

Nous avons vu comment procéder pour obtenir un access token, toutefois, celui-ci est soumis à une date d'expiration (généralement une heure). Ainsi, dans l'endpoint **/auth/google/callback** nous avons défini une méthode `refreshAccessToken` permettant d'envoyer une requête post à <https://accounts.google.com/o/oauth2/token> en fournissant entre autres, un **refresh token**.

Ce refresh token est utilisé par google pour vérifier qu'un utilisateur a bien le droit d'avoir un nouvel access token. La réponse de cette requête post vers google est ensuite récupérée dans une variable "tokens" dans laquelle nous mettons le champ "access\_token" présente dans le corps de la réponse.

Cette méthode "refreshAccessToken" est appelée toutes les 5 minutes une fois qu'un utilisateur a tenté de se connecter, qu'il ait un compte dans notre base de données ou non.

Nous comprenons donc que la valeur de l'access token dans `googleClient` est mise à jour toutes les 5 minutes, cependant, tant que cette valeur n'est pas stockée dans le "localStorage", alors elle est inutilisable. C'est pourquoi nous avons créé un endpoint **/auth/refresh-token** sur lequel le frontend peut récupérer par une requête get un access token mis à jour.

Le frontend récupère le token sur cet endpoint toutes les 5 minutes.

→ Par ailleurs, il est important de mentionner que nous avons tenu à protéger cet endpoint en lui faisant accepter un access token en tant que header d'autorisation pour éviter toute tentative de récupération d'un token depuis cet endpoint par une requête sur Postman par exemple.

## Sécurité et protection des données

Notre micro-service d'authentification prend en compte la sécurité et la protection des données des utilisateurs. L'utilisation d'OAuth2 garantit que les mots de passe des utilisateurs ne sont jamais partagés avec notre application, minimisant ainsi les risques de fuites de données.

## Collaboration avec le Frontend, le Backend et le Déploiement

Ce micro-service d'authentification étant étroitement lié au frontend et au backend, il a été revu et modifié pendant la phase d'intégration entre ces 3 blocs. Les difficultés rencontrées sont surtout apparues au moment de cette phase d'intégration et de déploiement. Principalement, nous avons dû adapter le code de ces 3 blocs afin d'assurer leur interconnexion notamment en traitant des erreurs de "cross origin" (CORS), mais nous avons aussi dû travailler en étroite collaboration avec les membres participant au déploiement des différents services.

## Bilan & axes d'amélioration

En conclusion, le microservice d'authentification que nous avons développé est une solution efficace pour permettre à des utilisateurs de se connecter à notre application tout en garantissant la sécurité et la protection de leurs données personnelles.

En utilisant l'authentification OAuth2 de Google, nous avons mis en place une méthode uniforme et sécurisée pour gérer l'accès aux ressources en ligne.

Nous avons également mis en place un formulaire d'inscription pour les nouveaux utilisateurs qui n'ont pas encore de compte dans notre base de données.

Le processus de récupération et de vérification de l'access token ainsi que la vérification de l'existence de l'utilisateur dans notre base de données ont été traités de manière approfondie dans notre microservice.

Bien que notre microservice d'authentification soit fonctionnel, il existe toujours des axes d'amélioration. Nous pourrions notamment envisager une intégration avec d'autres fournisseurs d'authentification, tels que Facebook ou Twitter.

Nous pourrions également ajouter des fonctionnalités de gestion de compte utilisateur, telles que la modification de mot de passe ou la récupération de compte. Enfin, nous pourrions améliorer la gestion des erreurs et des exceptions pour une meilleure expérience utilisateur.

## V. Toujours rester informé : le mailing

Ce sous-projet visait à créer un microservice d'envoi permettant à d'autres microservices d'envoyer des notifications par courrier électronique aux utilisateurs de l'application.

Le microservice a été créé pour simplifier le processus d'envoi d'e-mails en fournissant une API facile à utiliser pour que d'autres équipes puissent l'intégrer dans leur code.

### Implémentation

On commence par importer l'ensemble des modules allant être utiles lors de l'implémentation : en commençant par "nodemailer" : module allant nous permettre de configurer un service de messagerie, au "body-parser" : module allant permettre de mettre en forme le message allant être envoyé en se basant sur une structure JSON.

Il faut par la suite mettre en place un "transporteur" pour le service de messagerie, pour qu'il transmettre chacune des notification directement à l'utilisateur dès que la demande est reçue.

On fait appel également à l'application Express pour créer un module CORS qui sera utilisé pour autoriser l'accès à l'ensemble des ressources.

En se basant sur ce dernier, un routeur Express est aussi configuré pour vérifier l'authentification de l'utilisateur connecté en se basant sur le token récupérer depuis le micro-service d'authentification (décris précédemment).

On utilise également Swagger, un outil de conception, de construction et de documentation des API. NPM dispose d'un module "swagger-autogen" qui peut générer le document JSON et héberger la page swagger.

Pour de plus amples informations à propos des différentes fonctionnalités du micro-service, voici le lien vers l'API du micro-service : <http://zain.ovh:9095/api/>

Servers  
http://zain.ovh:9095/ Authorize

**Mail**

**POST** /mail Send a mail to some recipients

**Schemas**

- Mail >
- Message >
- Error >

Capture d'écran de l'API présente dans : <http://zain.ovh:9095/api/>

Comme pour les autres micro-services Node.js, nous utilisons également Docker pour conteneuriser ce micro-service.

## Utilité du token d'authentification

Le token d'authentification est nécessaire, car il permettra d'activer le service de mailing ou non, pour envoyer un mail de réservation. 2 cas se présentent alors :

- Un token d'authentification valide est présent : le mail est envoyé à l'utilisateur connecté

**Responses**

**Curl**

```
curl -X 'POST' \
'http://zain.ovh:9095/mail' \
-H 'Accept: application/json' \
-H 'Authorization: Bearer ya29.adMe19s' \
-H 'Content-Type: application/json' \
-d '{
  "subject": "Hello, world!",
  "htmlBody": "This is the body of the email<br>You can use <b></b> tags here",
  "recipients": [
    "john.doe@gmail.com",
    "jane.doe@gmail.com"
  ]
}'
```

**Request URL**  
<http://zain.ovh:9095/mail>

**Server response**

Code	Details
200	<b>Response body</b> <pre>{   "message": "Email sent" }</pre> <div style="text-align: right;"> <span style="border: 1px solid #ccc; padding: 2px;">Copy</span> <span style="border: 1px solid #ccc; padding: 2px;">Download</span> </div> <b>Response headers</b> <pre>access-control-allow-origin: * connection: keep-alive content-length: 24 content-type: application/json; charset=utf-8 date: Sat, 15 Apr 2023 14:10:06 GMT etag: W/"18-qk09V+skJfAI1Ns+VBIHkpCyCpk" keep-alive: timeout=5 x-powered-by: Express</pre>

Un token d'authentification valide

- Le token n'est pas présent : le mail n'est pas envoyé et, un message indique à l'utilisateur qu'il n'est pas connecté

```
Curl
curl -X 'POST' \
  'http://zain.ovh:9095/mail' \
  -H 'Accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "subject": "Hello, world!",
    "htmlbody": "This is the body of the email<br>You can use <b></b> tags here",
    "recipients": [
      "john.doe@gmail.com",
      "jane.doe@gmail.com"
    ]
}'
```

Request URL  
`http://zain.ovh:9095/mail`

Server response

Code	Details
401	Error: Unauthorized
Response body	
<pre>{   "error": "Unauthorized request" }</pre>	
<a href="#">Copy</a> <a href="#">Download</a>	
Response headers	
<pre>access-control-allow-origin: * connection: keep-alive content-length: 32 content-type: application/json; charset=utf-8 date: Sat, 25 Mar 2023 14:11:29 GMT etag: W/20-XTXILCqV1k8K5Kan/yQGreErgM" keep-alive: timeout=5 x-powered-by: Express</pre>	

Un token d'authentification invalide

## Finalité

Ce microservice est un exemple exposant qu'il est tout à fait possible d'implémenter un service de messagerie indépendant du reste en se basant sur NodeJs et une API REST protégé par une authentification basé sur un token.

## VI. Finalisation : le déploiement

Dans le monde numérique d'aujourd'hui, le déploiement rapide et fiable des applications logicielles est essentiel pour répondre aux demandes des clients et rester en avance sur la concurrence. Dans cette partie, nous discuterons du déploiement de notre projet, qui a impliqué le déploiement de différents composants sur un serveur Oracle OCI Ubuntu lié au domaine zain.ovh. Plus précisément, notre frontend, service d'authentification et d'envoi d'email ont été déployés dans Docker, tandis que notre service de base de données et notre backend ont été déployés en Java. Pour assurer un processus de déploiement fluide, nous avons utilisé des pipelines Jenkins pour orchestrer le déploiement de ces composants.

### Repartition des tâches

Tâches à effectuer	Euphraïm	Quang Viet	Zain
Mise en place/Administration du Serveur			✓
Installations/Configurations Nécessaires			✓
Ecriture des pipelines	✓	✓	✓
Administration Artifactory	✓		
Configuration Docker		✓	

Tâches à effectuer	Euphraïm	Quang Viet	Zain
Administration Jenkins			<input checked="" type="checkbox"/>

## Architecture finale

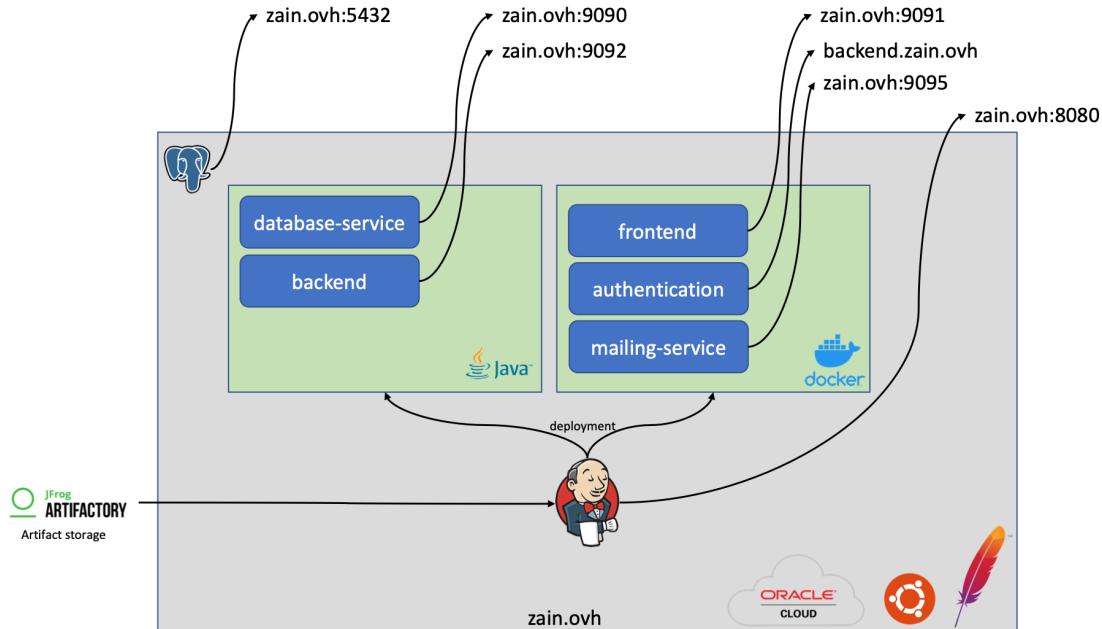


Schéma récapitulatif de l'ensemble des domaine et ports impliqués

Les swagger de nos api springboot :

- database-service : <http://zain.ovh:9090/swagger-ui/index.html>
- backend : <http://zain.ovh:9092/swagger-ui/index.html>

## Plan de déploiement

- Préparation de l'environnement de déploiement
  - Configuration du serveur Oracle OCI Ubuntu
  - Création du domaine `zain.ovh`
  - Configuration des connexions réseaux et des pare-feux
  - Installation des outils nécessaires pour le déploiement (Docker, Java, Jenkins)
- Déploiement des services Docker
  - Configuration de l'image Docker pour le frontend, le service d'authentification et le service de mailing
  - Utilisation de Docker Compose pour déployer les services Docker
  - Test de l'intégration entre les différents services Docker

- Déploiement du service de base de données et du backend Java
  - Configuration du service de base de données Oracle
  - Configuration de l'environnement Java pour le backend
  - Compilation du code source Java et création du package d'exécution
  - Déploiement du package sur le serveur
  - Configuration des variables d'environnement pour le service Java
  - Test de l'intégration entre le service de base de données et le backend Java
- Configuration de Jenkins pour l'orchestration du déploiement
  - Création d'un pipeline Jenkins pour chaque service
  - Configuration des étapes de déploiement pour chaque pipeline
  - Test de l'exécution de chaque pipeline
- Déploiement de la solution complète
  - Déploiement de tous les services en utilisant les pipelines Jenkins
  - Vérification de l'intégration entre tous les services déployés

## Résultats de déploiement

Le déploiement du projet a été un succès ! Le processus de déploiement orchestré via Jenkins a permis une mise à jour facile et rapide de chaque composant, avec des erreurs de déploiement corrigées rapidement grâce aux notifications configurées dans Jenkins.

## VII. Conclusion générale

En conclusion, la mise en place de notre application de location de logement basée sur l'architecture de micro-services a été une très bonne expérience.

Elle a permis à notre équipe de développer des compétences en collaboration efficace, en organisation méthodique de projet et en communication fluide pour la réussite du projet. Nous avons exploré de nombreuses technologies, notamment *AngularJS* pour l'interface utilisateur personnalisable, *OAuth2* pour la sécurité, et *Spring Boot* pour le backend & la persistance des données.

Nous pouvons tous être fiers de ce que nous avons accompli ensemble, et convaincus que cette expérience nous a permis d'apprendre énormément tant au niveau technique que social.

## Liens utiles

- Github du projet : <https://github.com/za1nzafar/welhome>
- Application web : <http://zain.ovh:9091/>
- Swagger mailing-service : <http://zain.ovh:9095/api/>
- Swagger database-service : <http://zain.ovh:9090/swagger-ui/index.html>

- Swagger backend-service : <http://zain.ovh:9092/swagger-ui/index.html>
- Vidéo de démo : <https://www.youtube.com/watch?v=4OCWRfscZe0>

## Comment lancer le projet en localhost ?

⇒ Etapes à suivre :

### Pour les services utilisant Docker notamment :

- authentication-service
- frontend-service
- mailing service

#### Via le terminal :

- Accéder au répertoire qui contient le fichier docker-compose.yml
- Lancer la commande

```
docker-compose up -d
```

### Pour les services utilisant java, notamment :

- database-service
- backend-service

#### Via le terminal :

- Accéder au répertoire qui contient le fichier pom.xml
- Lancer la commande

```
mvn clean install
```

- Lancer la commande

```
java -jar target_dir/*.jar &
```