

# Rapport WEB avancé

Chad Estoup—Streiff, Elouan Lahellec

|  |          |
|--|----------|
| <b>SETUP.....</b>                      | <b>1</b> |
| <b>CREATION DE LA BASE. ....</b>       | <b>2</b> |
| Containerisation .....                 | 2        |
| Création de GRPC .....                 | 2        |
| Création du backend en python .....    | 4        |
| Création du front en HTML JS CSS ..... | 4        |
| <b>AUTHENTIFICATION .....</b>          | <b>4</b> |
| Google auth .....                      | 4        |
| Keycloak auth .....                    | 6        |
| <b>KUBERNETESS.....</b>                | <b>8</b> |

## Setup

Pour suivre l'organisation du projet nous avons choisi ces technologies :

- Un front en HTML et JavaScript
- Un back-end en Python en FastAPI
- Un service en JAVA avec GRPC.
- Google comme délégation d'authentification OAuth 2.1

Comme outils nous utilisons :

- IntelliJ pour éditer le service grpc et gérer maven
- Visual Studio Code pour gérer le front et le backend

# Création de la base.

## Containerisation

Nous avons dès le début organisé le projet sous forme de containers docker afin d'éviter les problèmes de déploiement, compilation et génération du aux différents OS que nous utilisons (Windows et MacOS).

Ainsi nous créons un docker-compose permettant de construire et lancer tous les containers.

Nous faisons attention à mettre les informations sensibles dans un fichier de configuration externe pour éviter les variables en dur dans le code.

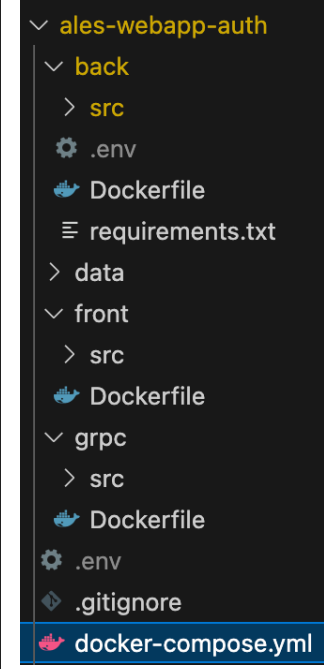
```
version: '3'

services:
  resa_front:
    build:
      context: front
    env_file:
      - .env
    container_name: resa_front
    restart: always
    depends_on:
      - resa_backend
    ports:
      - ${APP_PORT}:80
    networks:
      - resa-network

  resa_backend:
    build:
      context: back
    env_file:
      - .env
    container_name: resa_backend
    restart: always
    ports:
      - ${BACK_PORT}:80
    networks:
      - resa-network

  resa_grpc:
    build:
      context: grpc
    env_file:
      - .env
    container_name: resa_grpc
    restart: always
    ports:
      - ${GRPC_PORT}:8080
    networks:
      - resa-network

networks:
  resa-network:
    name: resa-network
    driver: bridge
```



## Création de GRPC

Nous décidons de faire 4 endpoints :

- Réservation de vol pour un utilisateur
- Lister les vols d'un utilisateur
- Réservation d'hôtel pour un utilisateur
- Lister les hôtels d'un utilisateur

Nous utilisons un bête Manager singleton pour sauvegarder et générer les différents vols, hôtels et réservations.

Nous avons qu'une difficulté rencontrée pour cette partie qui est le packaging en fat jar qui a bloqué quelques minutes.

Voici notre .proto final:

```
syntax = "proto3";

package reservation;

service HotelReservation {
  rpc ReserveHotel (HotelRequest) returns (HotelResponse);
  rpc ListReservedHotels (ListHotelsRequest) returns (ListHotelsResponse);
}

service FlightReservation {
  rpc ReserveFlight (FlightRequest) returns (FlightResponse);
  rpc ListReservedFlights (ListFlightsRequest) returns (ListFlightsResponse);
}

message HotelRequest {
  string hotel_id = 1;
  string user_id = 2;
}

message HotelResponse {
  string reservation_id = 1;
  bool success = 2;
}

message ListHotelsRequest {
  string user_id = 1;
}

message ListHotelsResponse {
  repeated HotelReservationInfo hotels = 1;
}

message HotelReservationInfo {
  string hotel_id = 1;
  string user_id = 2;
  string check_in_date = 3;
  string check_out_date = 4;
  int32 number_of_rooms = 5;
}

message FlightRequest {
  string flight_id = 1;
  string user_id = 2;
}

message FlightResponse {
  string reservation_id = 1;
  bool success = 2;
}

message ListFlightsRequest {
  string user_id = 1;
}

message ListFlightsResponse {
  repeated FlightReservationInfo flights = 1;
}
```

```

}

message FlightReservationInfo {
    string flight_id = 1;
    string departure_date = 3;
    string return_date = 4;
    int32 number_of_tickets = 5;
}

```

## Création du backend en python

Nous avons l'habitude de faire ce genre de backend, donc aucun problème.

Après avoir généré le code python pour GRPC pour communiquer avec le service, nous avons implémenté les 4 endpoints.

Tout fonctionne correctement.

## Création du front en HTML JS CSS

Nous avons fait un site tout simple qui appelle l'API et récupère les informations.

Aucune difficulté.

## Authentification

### Google auth

Nous avons implémenté les endpoints côté backend pour envoyer au client l'autorisation de demande d'authentification et l'endpoint pour valider le code google et retourner un token JWT.

```

#####
#           Google Auth           #
#####
GOOGLE_CLIENT_ID = config["GOOGLE_CLIENT_ID"]
GOOGLE_CLIENT_SECRET = config["GOOGLE_CLIENT_SECRET"]
GOOGLE_REDIRECT_URI = "http://localhost:8901/auth"
JWT_SECRET = config["JWT_SECRET"]
JWT_ALGORITHM = "HS256"

@app.get("/login/google")
async def login_google():
    return {
        "url":
f"https://accounts.google.com/o/oauth2/auth?response_type=code&client_id={GOOGLE_CLIENT_ID}&redirect_uri={GOOGLE_REDIRECT_URI}&scope=openid%20profile%20email&access_type=offline"
    }

@app.get("/auth/google")
async def auth_google(code: str):
    token_url = "https://accounts.google.com/o/oauth2/token"
    data = {
        "code": code,
        "client_id": GOOGLE_CLIENT_ID,
        "client_secret": GOOGLE_CLIENT_SECRET,
        "redirect_uri": GOOGLE_REDIRECT_URI,

```

```

        "grant_type": "authorization_code",
    }
    response = requests.post(token_url, data=data)
    access_token = response.json().get("access_token")
    user_info = requests.get(
        "https://www.googleapis.com/oauth2/v1/userinfo",
        headers={"Authorization": f"Bearer {access_token}"},
    )
    user_info = user_info.json()

    token = jwt.encode(
        {
            "sub": user_info["id"],
            "name": user_info["name"],
            "email": user_info["email"],
            "exp": datetime.datetime.utcnow() + datetime.timedelta(hours=1),
        },
        JWT_SECRET,
        algorithm=JWT_ALGORITHM,
    )
    return {"access_token": token, "token_type": "bearer"}

@app.get("/token")
async def get_token(token: str = Depends(oauth2_scheme)):
    try:
        payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM])
        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED, detail="Token has expired"
        )
    except jwt.InvalidTokenError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token"
        )

def get_current_user(token: str):
    try:
        payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM])
        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED, detail="Token has expired"
        )
    except jwt.InvalidTokenError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token"
        )

```

Nous avons sécurisé les endpoints en demandant le token et en vérifiant sa validité:

```

@app.post("/flights/reserve", tags=["Flight"])
async def add_flights_reservation(
    flight: str,
    token: str = Depends(oauth2_scheme),
):
    user = get_current_user(token)

    with grpc.insecure_channel("resa_grpc:8080") as channel:
        flight_stub = reservation_pb2_grpc.FlightReservationStub(channel)

        flight_request = reservation_pb2.FlightRequest(
            flight_id=flight,
            user_id=user["sub"],
        )

```

```

    flight_stub.ReserveFlight(flight_request)
    return True

```

Du côté JS nous avons fait un bouton qui va récupérer l'URL google pour s'authentifier.

```

document.getElementById('loginBtn').addEventListener('click', () => {
    fetch('http://localhost:8902/login/google')
        .then(response => response.json())
        .then(data => {
            window.location.href = data.url;
        });
});

```

Puis nous avons fait une page HTML de redirection avec un code JS qui va demander au serveur de valider le code et récupérer le token d'authentification.

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Google Login</title>
    <link rel="stylesheet" href="../assets/css/loading.css">
    <script>
        window.addEventListener('load', () => {
            const urlParams = new URLSearchParams(window.location.search);
            const code = urlParams.get('code');

            if (code) {
                fetch(`http://localhost:8902/auth/google?code=${code}`)
                    .then(response => response.json())
                    .then(userInfo => {
                        saveToken(userInfo.access_token);
                        window.location.href = "/";
                    });
            } else {
                const token = loadToken();
                if (token) {
                    fetchUserInfo(token);
                }
            }
        });

        function saveToken(token) {
            document.cookie = `token=${token}; path=/; max-age=3600;`;
        }
    </script>
</head>
<body>
    <div style="display: flex; justify-content: center; align-items: center; width: 100vw; height: 100vh;">
        <span class="loader"></span>
    </div>
</body>
</html>

```

## Keycloak auth

Nous avons créé dans notre docker les services nécessaires pour lancer un serveur keycloak

```

postgres:
    image: postgres:15.7-alpine
    restart: unless-stopped
    healthcheck:

```

```

    test: ["CMD", "pg_isready", "-U", "keycloak"]
environment:
  POSTGRES_DB: keycloak
  POSTGRES_USER: ${KC_DB_USERNAME}
  POSTGRES_PASSWORD: ${KC_DB_PASSWORD}
volumes:
  - ./data/db:/var/lib/postgresql/data
hostname: kc_postgres
networks:
  - resa-network

keycloak:
  image: quay.io/keycloak/keycloak:25.0.1
  command: ["start-dev", "--import-realm"]
  restart: unless-stopped
  environment:
    KC_DB: postgres
    KC_DB_USERNAME: ${KC_DB_USERNAME}
    KC_DB_PASSWORD: ${KC_DB_PASSWORD}
    KC_DB_URL: "jdbc:postgresql://kc_postgres:5432/keycloak"
    KC_METRICS_ENABLED: true
    KC_LOG_LEVEL: ${KC_LOG_LEVEL}
    KEYCLOAK_ADMIN: ${KEYCLOAK_ADMIN}
    KEYCLOAK_ADMIN_PASSWORD: ${KEYCLOAK_ADMIN_PASSWORD}
  ports:
    - ${KC_PORT}:8080
  hostname: keycloak
  networks:
    - resa-network

```

Sur l'UI de keycloak nous avons créé un royaume "Ales-Resa" et un utilisateur "back".

Comme pour google on ajoute le code nécessaire coté backend

```

KEYCLOAK_SERVER_URL = config["KEYCLOAK_SERVER_URL"]
KEYCLOAK_REALM = config["KEYCLOAK_REALM"]
KEYCLOAK_CLIENT_ID = config["KEYCLOAK_CLIENT_ID"]
KEYCLOAK_CLIENT_SECRET = config["KEYCLOAK_CLIENT_SECRET"]
KEYCLOAK_REDIRECT_URI = "http://localhost:8901/auth/keycloak"

@app.get("/login/keycloak", tags=["Keycloak"])
async def login_keycloak():
    return {
        "url": f"http://localhost:8900/realms/{KEYCLOAK_REALM}/protocol/openid-
connect/auth?response_type=code&client_id={KEYCLOAK_CLIENT_ID}&redirect_uri={KEYCLOAK_REDIRECT_URI}&scope=openid%20profile%20email"
    }

@app.get("/auth/keycloak", tags=["Keycloak"])
async def auth_keycloak(code: str):
    token_url = f"{KEYCLOAK_SERVER_URL}/realms/{KEYCLOAK_REALM}/protocol/openid-connect/token"
    data = {
        "code": code,
        "client_id": KEYCLOAK_CLIENT_ID,
        "client_secret": KEYCLOAK_CLIENT_SECRET,
        "redirect_uri": KEYCLOAK_REDIRECT_URI,
        "grant_type": "authorization_code",
    }
    response = requests.post(token_url, data=data)
    response_data = response.json()
    access_token = response_data.get("access_token")
    if not access_token:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token")

    user_info = requests.get(
        f"{KEYCLOAK_SERVER_URL}/realms/{KEYCLOAK_REALM}/protocol/openid-connect/userinfo",
        headers={"Authorization": f"Bearer {access_token}"},
    )

```

```
user_info = user_info.json()

token = jwt.encode(
    {
        "sub": user_info["sub"],
        "name": user_info["name"],
        "email": user_info["email"],
        "exp": datetime.datetime.utcnow() + datetime.timedelta(hours=1),
    },
    JWT_SECRET,
    algorithm=JWT_ALGORITHM,
)

return {"access_token": token, "token_type": "bearer"}
```

Nous dupliqué le code JS et HTML utilisé pour Google en remplaçant les endpoints google vers ceux de keycloak.

Nous avons une difficulté avec les hostname mais en réglant les hosts de la machine hôte on a résolue le prblm.

## Kubernetess

Nous n'avons pas reussis a push les images sur le repo infres.

La commande suivante retourne une erreur 404.

```
wget http://registry.infres.fr/v2/MyService/tags/list
```