Rapport d'Analyse et de Déploiement - TP Webservices

Réalisé par :

Dimeck Raphaël
Salaün Kerrian

Table des matières

Introduction générale

Ce projet s'inscrit dans une série de trois travaux pratiques dédiés à la conception, au déploiement et à la sécurisation de Web Services modernes. À travers ces TP, nous explorons des thématiques clés du développement backend et DevOps : création d'API REST, authentification déléguée (OAuth2, Keycloak), documentation Swagger, containerisation (Docker), orchestration (k3s/Kubernetes) et sécurité en environnement cloud-native.

- TP1 : Mise en place de l'application OpomlyTravel et des Webservices REST sécurisés.
- TP2 : Déploiement dans un cluster Kubernetes léger (k3s) avec monitoring et montée en charge.
- TP3 : Analyse des vulnérabilités, compromission et sécurisation des environnements Kubernetes.

Note explicative

1. Structure du Projet

```
TP-Webservices/
├── client/        # Frontend React (SPA)
├── server/        # Backend Express/Node.js (API REST)
├── docker-compose.yml
└── docs/           # Spécifications Swagger/OpenAPI, fichiers techniques
```

---

2. Prérequis

Pour le développement local :

- **Node.js** ≥ 16.x
- **npm**
- **Docker** et **Docker Compose**
- Navigateur web récent (Chrome, Firefox...)

Pour le déploiement Kubernetes :

- **WSL 2** (Windows Subsystem for Linux)
- **k3s**
- **kubectl**
- Accès à un registre Docker privé (`registry.infres.fr`)

---

3. Installation locale

Étapes

1. **Cloner le dépôt**
   ```
   git clone <url-du-repo>
   ```

2. Aller dans le dossier
   ```
   cd TP-Webservices
   ```

3. Installer les dépendances

   - Backend :
     ```
     cd server
     npm install
     ```

   - Frontend :
     ```
     cd ../client
     npm install
     ```

---

4. Lancement local

4.1. Lancement via NPM (développement)

Backend :
```
npm run server
# API accessible sur http://localhost:3001
```

Frontend :
```
npm run client
# Application accessible sur http://localhost:3000
```

4.2. Lancement via Docker Compose

```
docker-compose up --build
```

Accès

- Frontend : http://localhost:3000
- Backend : http://localhost:3001
- Keycloak : http://localhost:8080
- Swagger : http://localhost:3001/api-docs

5. Fonctionnalités principales

Authentification

- Authentification par **Google OAuth 2.1**
- Authentification par **Keycloak (OpenID Connect)**
- Stockage sécurisé des tokens
- Déconnexion, rafraîchissement de session

Page d'accueil

- Hero section immersive
- Grille de **destinations de rêve** (images Unsplash)
- Bouton d'appel à l'action : redirection vers les voyages réservés si authentifié

Voyages réservés

- Route `GET /api/trips`
- Affichage sous forme de cartes (image, destination, prix, date)
- Données mockées mais structurées pour un backend réel

API REST

- `/api/google/*` → Gestion des connexions Google
- `/api/keycloak/*` → Gestion des sessions Keycloak
- `/api/trips` → Récupération des voyages disponibles

6. Documentation API (Swagger/OpenAPI)

Accessible via : http://localhost:3001/api-docs

- Visualisation interactive des routes
- Possibilité de tester les endpoints avec token JWT
- Contrat rédigé manuellement avec Swagger Editor

---

7. Sécurité et Middleware

- Middleware de protection des routes (JWT obligatoire)
- Intégration Keycloak (client "bearer-only" pour le backend)
- Configuration CORS entre client et serveur
- Validation manuelle du JWT (middleware personnalisé)

---

8. Tests

Tests manuels

- Lancement de l'app via navigateur : accès à `/`, login, `/mytrip`
- Vérification de la protection des routes via appels sans token
- Inspection du token JWT avec jwt.io

Tests API

- Swagger UI
- Postman avec token dans l'en-tête Authorization

---

9. Déploiement avec Docker

Build local

```
docker-compose build
```

Push vers registre privé

```
docker tag opomly-server registry.infres.fr/opomly/server
docker push registry.infres.fr/opomly/server
```

---

10. Déploiement Kubernetes (k3s)

Étapes générales

Installation de k3s

```
curl -sfL https://get.k3s.io | K3S_KUBECONFIG_MODE="644" sh -
```

1. **Déploiement**

```
kubectl apply -f k8s/opomly-server.yaml
kubectl apply -f k8s/opomly-client.yaml
```

2. **Accès**

Ajouter dans `/etc/hosts` :

```
127.0.0.1 opomly.infres.fr
```

Accéder à : http://opomly.infres.fr

Analyse de la solution

1. Mise en place de l'application REST

Solution mise en place

Nous avons développé un **backend Node.js avec Express** situé dans le dossier `server/`. Ce backend expose des **Web Services REST** permettant la gestion de vols. Les endpoints REST permettent d'obtenir des ressources statiques représentant les caractéristiques d'un vol.

L'ensemble des routes est défini dans le fichier `server/routes/trips.routes.js`.

Difficultés rencontrées

- Structurer les ressources et concevoir des routes cohérentes afin de refléter une hiérarchie RESTful correcte et intuitive.

Améliorations possibles

- Connecter le backend à une base de données pour rendre les ressources dynamiques et persistantes.
- Implémenter des fonctionnalités avancées : **pagination, recherche filtrée, tri**.
- Ajouter des statuts HTTP plus précis et une gestion détaillée des erreurs.

---

2. Mise en place de l'application JavaScript (Frontend)

Solution mise en place

Le frontend est développé sous la forme d'une **Single Page Application (SPA)** avec **React**, situé dans le dossier `client/`.
Il consomme les Web Services REST exposés par le backend afin d'afficher dynamiquement la liste des vols disponibles et de permettre la réservation.

Les appels API sont réalisés via la méthode `fetch` dans les composants React, avec une gestion asynchrone des réponses.

Difficultés rencontrées

- Gestion du **CORS (Cross-Origin Resource Sharing)** entre le frontend et le backend lors des appels API.

- Synchronisation de l'authentification et du passage des tokens JWT entre le frontend et le backend.

Améliorations possibles

- Améliorer l'expérience utilisateur (UX/UI) : loader, notifications, affichage des erreurs API.

- Ajouter une gestion robuste des erreurs réseau et de l'expiration des sessions.

---

3. Délégation d'autorisation OAuth 2.1 (Google)

Solution mise en place

Le backend intègre l'**authentification Google via OAuth 2.1**, mise en œuvre dans `server/services/google.auth.js`.
L'application a été préalablement enregistrée sur la Google Developer Console pour obtenir les credentials nécessaires.

Le backend gère l'intégralité du **flux d'autorisation OAuth**, permettant de récupérer les informations du profil utilisateur authentifié.

Difficultés rencontrées

- Configuration des **credentials OAuth Google** et gestion correcte des URI de redirection.

- Compréhension et mise en œuvre du flux d'autorisation OAuth côté serveur (code d'autorisation, échange de tokens).

Améliorations possibles

- Ajouter la prise en charge d'autres fournisseurs OAuth (Facebook, GitHub, etc.).

- Sécuriser davantage le stockage des tokens d'accès et des refresh tokens.

- Implémenter un système de rafraîchissement automatique des tokens.

---

4. Protection du Web Service REST avec OpenID Connect (Keycloak)

Solution mise en place

Keycloak a été déployé via Docker grâce au fichier `docker-compose.yml`.
Le backend REST a été configuré comme **client "bearer-only" dans Keycloak** et protégé à l'aide d'un middleware personnalisé (`server/middleware/keycloak.middleware.js`).

Cette configuration assure que seules les requêtes authentifiées et munies d'un token valide peuvent accéder aux ressources protégées.

Difficultés rencontrées

- Configuration fine de Keycloak et adaptation du middleware pour gérer l'autorisation.
- Appropriation des concepts du protocole **OpenID Connect** et des rôles associés.

Améliorations possibles

Aucunes

---

5. Authentification du frontend via Keycloak

Solution mise en place

Le frontend React utilise le package `keycloak-js` pour déléguer l'authentification à Keycloak.
La configuration du client **public** Keycloak est intégrée au frontend via un contexte React (`src/contexts/AuthContext.tsx`).

Le token JWT obtenu est ensuite utilisé pour sécuriser les appels API vers le backend.

Difficultés rencontrées

- Intégration technique de Keycloak avec React (initialisation, redirection, gestion du token).
- Synchronisation du token JWT entre le frontend et le backend dans les headers des requêtes.

Améliorations possibles

- Implémenter le **rafraîchissement automatique du token**.
- Améliorer la gestion des erreurs d'authentification (redirections, affichage clair des statuts d'erreur).

---

6. Analyse du token JWT

Solution mise en place

Le token JWT généré par Keycloak est récupéré via les outils de développement du navigateur. Son contenu est analysé via le site jwt.io pour vérifier :

- Les **claims** (informations sur l'utilisateur, l'émetteur, la durée de validité, etc.)
- Les permissions et les rôles attribués.

Difficultés rencontrées

- Compréhension de la structure détaillée d'un JWT : header, payload, signature.

Améliorations possibles

- Intégrer un outil d'analyse de JWT directement dans le frontend pour afficher les claims utilisateur de manière transparente.

---

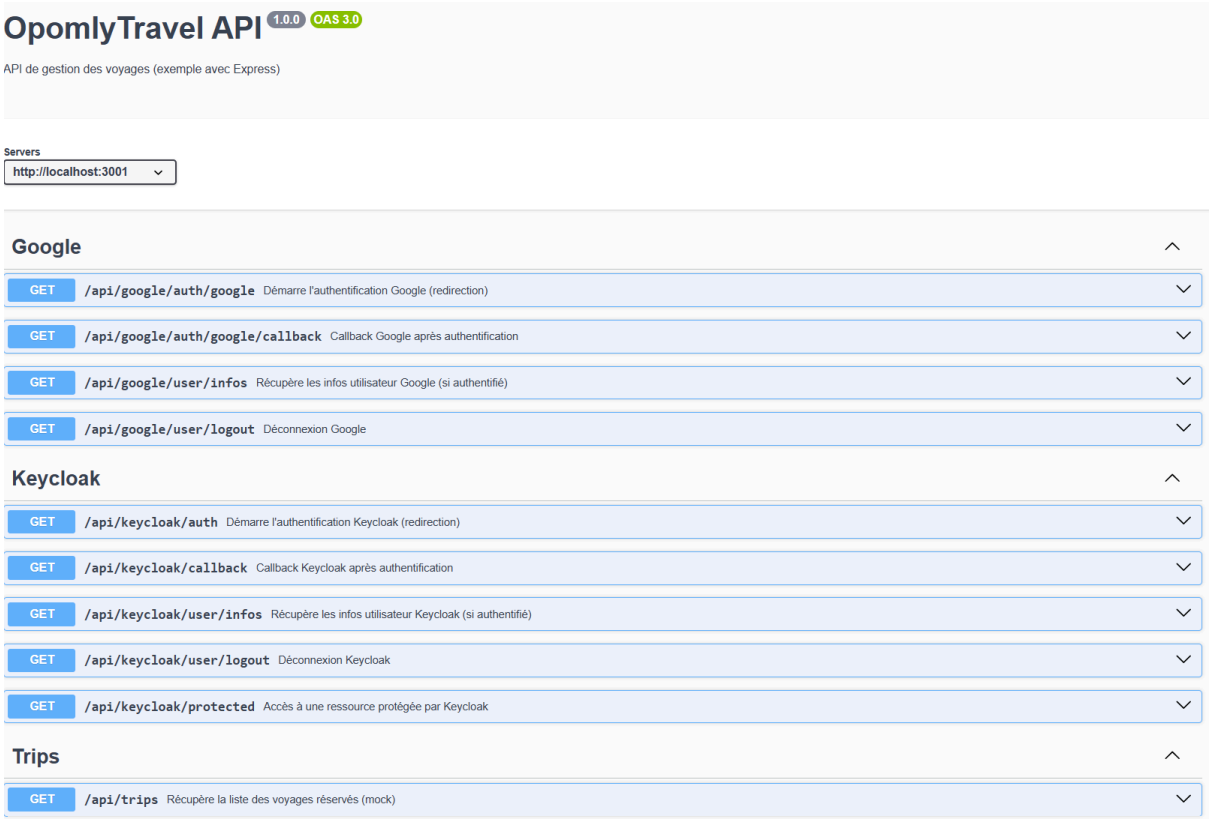
7. Définition du contrat d'API avec OpenAPI

Solution mise en place

Le contrat OpenAPI a été rédigé avec Swagger Editor afin de :

- Documenter de manière structurée l'ensemble des endpoints REST.

- Générer une documentation lisible et interactive en HTML.
- Faciliter la génération de librairies clientes.



Difficultés rencontrées

- Prise en main des spécifications OpenAPI.
- Structuration correcte des ressources et des réponses HTTP.

Améliorations possibles

- Automatiser la génération du contrat OpenAPI à partir des routes Express.
- Ajouter des exemples de requêtes/réponses et des descriptions détaillées pour chaque champ.

---

8. Management d'API avec Gravitee

Solution mise en place

Gravitee a été installé pour **gérer l'exposition et la sécurité des API**.
Le contrat OpenAPI a été importé dans Gravitee, permettant :

- La configuration d'une API Gateway.
- La mise en place de politiques d'accès telles que le **rate limiting**.
- Le test et la surveillance des API via la gateway.

Difficultés rencontrées

- Configuration initiale de Gravitee et compréhension des différents composants (API Gateway, Management API, Portal).
- Apprentissage des politiques d'accès et de transformation des requêtes.

Améliorations possibles

- Automatiser le déploiement de Gravitee et des API via des scripts ou des pipelines CI/CD.
- Implémenter des politiques de sécurité supplémentaires (IP filtering, quotas).

---

9. Containerisation et orchestration

Solution mise en place

Chaque composant (backend, frontend, Keycloak) dispose de son propre **Dockerfile** à la racine de son dossier.
Un fichier `docker-compose.yml` global orchestre l'exécution des services localement.

Les images sont construites et nommées selon la convention :

```
registry.infres.fr/MyService
```

Difficultés rencontrées

- Problèmes de permissions avec WSL, notamment sur les fichiers `node_modules/.bin`.
- Exécution des scripts React pendant le build Docker, résolus en ajoutant `RUN chmod +x` dans les Dockerfile.

Améliorations possibles

- Optimiser les Dockerfile avec une approche **multi-stage build**.
- Ajouter des **probes de santé** pour assurer la disponibilité des services.
- Sécuriser les variables d'environnement sensibles via Docker Secrets.

TP Kubernetes - Sécurité et Déploiement

1. Containerisation des Services

Mise en place

Chaque service dispose d'un Dockerfile spécifique :

- `client/` pour le frontend React
- `server/` pour le backend Express
- Déploiement orchestré avec docker-compose

Les images sont buildées, taguées et poussées vers le registre local `registry.infres.fr`.

Difficultés rencontrées

- Gestion des droits d'accès aux fichiers sous WSL.
- Configuration correcte des chemins et des scripts dans les Dockerfile.

Améliorations possibles

Aucune

---

2. Installation et Configuration de k3s

Mise en place

- Installation de k3s sous WSL2 après mise à jour de l'environnement.
- Désactivation du démarrage automatique de k3s pour un meilleur contrôle manuel.
- Configuration de l'environnement kubectl via la variable `KUBECONFIG`.

Difficultés rencontrées

- Configuration réseau entre WSL2 et le cluster Kubernetes.
- Gestion des ports et du DNS sous WSL.

Améliorations possibles

- Création d'un **script d'installation automatisé** de k3s et de ses dépendances.
- Ajout de probes de santé sur les services déployés.

---

3. Mise en place d'un registre Docker local

Mise en place

- Configuration de l'accès local au registre `registry.infres.fr`.
- Déploiement d'un registre Docker via des fichiers YAML spécifiques.
- Configuration des fichiers :
    - `/etc/rancher/k3s/registries.yaml`

- `/etc/docker/daemon.json` avec `insecure-registries`

Difficultés rencontrées

- Redémarrage coordonné de Docker et de k3s.

---

4. Déploiement sur k3s

Mise en place

- Déploiement des services avec des manifests Kubernetes :
    - Deployments
    - Services ClusterIP
    - Ingress avec mapping de noms DNS
- Exemple d'accès :

`http://MyService.infres.fr`

Difficultés rencontrées

- Configuration DNS et résolution des noms sous WSL2.
- Ajustement des modes réseau (bridge vs host).

Améliorations possibles

- Intégration d'un certificat SSL (Let's Encrypt) avec gestion automatique via cert-manager.

---

5. Mise en place du Horizontal Pod Autoscaler (HPA)

Mise en place

- Déploiement du HPA avec un fichier YAML dédié.
- Génération de charge via ApacheBench (`ab`) pour simuler des pics d'utilisation.

Difficultés rencontrées

- Configuration des métriques CPU pour le déclenchement du scaling.
- Observations parfois lentes des variations de réplicas.

---

6. TP Kubernetes - Sécurité avancée

Déploiement d'une charge vulnérable

- Utilisation d'une application volontairement vulnérable (vulnnode).
- Déploiement via Docker Compose et Kubernetes.
- Vérification de l'accès public via le DNS local.

Exploitation via Reverse Shell

- Injection d'un payload Perl généré via revshells.com dans un formulaire.
- Établissement d'un reverse shell fonctionnel avec la machine attaquante.

Mouvement latéral

- Utilisation des tokens Kubernetes montés dans les pods pour accéder à d'autres ressources.
- Téléchargement de kubectl dans le pod vulnérable pour tester les permissions.

Escalade de privilèges

- Création d'un pod privilégié avec accès au système hôte via `chroot`.
- Contrôle complet de la machine hôte possible depuis le pod compromis.

Conclusion Générale

Ce TP nous a permis de mettre en œuvre :

- Un **écosystème complet** basé sur Kubernetes avec des services conteneurisés.
- La gestion d'un **registre Docker local** et l'orchestration via k3s.
- La mise en place d'une **sécurité progressive** des services REST.
- L'analyse et l'exploitation de **vulnérabilités courantes** dans les clusters Kubernetes.