Modification d’un outil de facturation

RAPPORT TECHNIQUE

Du 03 avril au 24 juin 2023

Giuliana GODAIL-FABRIZIO

BUT Informatique, 2ème année

Entreprise : **Natural Solutions**

Tuteur en entreprise : **Julien GRAZIANI**

Tuteur IUT : **Gilles PERROT**

Sommaire

Introduction

L'application développée est un outil de facturation. Il s'agit d'un outil interne à l'entreprise qui est destiné à n'être utilisé que par les personnes en charge de la facturation.

Son objectif principal est de simplifier le processus de facturation des clients, en permettant de gagner du temps et en évitant les erreurs fréquentes liées aux fautes de frappe ou aux omissions.

Au-delà de cette fonctionnalité, cet outil offre également la possibilité de générer des rapports détaillant les activités de support[1] effectuées par l'entreprise sur une période donnée. On y gère aussi les informations relatives aux employés : date d'arrivée, date de départ, montant associé à une journée de travail, rôle, etc.

L'objectif de ce rapport technique est de faciliter la prise en main du projet et de fournir des informations détaillées pour d'éventuelles améliorations futures. Pour cela, nous commencerons par examiner les prérequis, puis nous aborderons l'arborescence du projet, et enfin, nous expliquerons certaines parties du code.

[1] Activités de support : tâches effectuées par l'équipe de développement après la livraison d'un projet, similaire à un service après-vente.

1 Prérequis

1.1 Installation préalable

Pour lancer l'application, Docker doit être installé sur votre ordinateur. Il s'agit d'un outil permettant de créer et d'exécuter des conteneurs logiciels légers. Ces conteneurs encapsulent toutes les dépendances nécessaires à l'application, simplifiant ainsi son déploiement et assurant une exécution cohérente sur divers environnements.

Aucune autre installation n'est nécessaire, car les dépendances et autres éléments du projet seront installés directement dans les conteneurs Docker.

1.2 Structure du projet

L'Outil Facturation est composé de quatre parties qui interagissent étroitement :

- La base de données contient toutes les informations nécessaires au bon fonctionnement de l'Outil Facturation. Elle peut être consultée via le frontend et modifiée par le backend.
- Le serveur est l'infrastructure logicielle chargée d'héberger et de gérer l'application. Il s'occupe des requêtes entrantes et sortantes, permettant ainsi l'exécution de l'application et la communication entre le frontend et le backend.
- Le backend récupère les données reçues du frontend et effectue le traitement approprié. Il peut s'agir d'une interaction avec la base de données ou de générer un rapport. Le backend s'exécute sur le serveur.
- Le frontend est l'interface par laquelle l'utilisateur passe pour effectuer une action. Il collecte les données saisies et les transmet au backend pour qu'elles soient traitées de manière appropriée.

1.3 Contraintes techniques

Langages utilisés : NodeJS, JS, ReactJS

Base de données : PostgreSQL, utilisation de Sequelize[2]

Tests : Mocha

Bibliothèques utilisées :

1. **Backend :**
    - Fichiers :
        - **adm-zip** (version 0.5.5) : Une bibliothèque pour la manipulation des fichiers ZIP.
        - **pizzip** (version 3.0.6) : Une bibliothèque pour la manipulation des fichiers ZIP en mémoire.
        - **docxtemplater** (version 3.21.0) : Une bibliothèque pour la génération de fichiers DOCX basés sur des modèles.
        - **xlsx** (version 0.16.9) : Une bibliothèque pour la manipulation des fichiers Excel.
    - Serveur :
        - **express** (version 4.17.1) : Un framework web pour Node.js utilisé pour la création des routes et la gestion des requêtes HTTP.
        - **express-json-validator-middleware** (version 2.1.1) : Un middleware pour Express qui permet de valider les objets JSON des requêtes.
    - Tests :
        - **mocha** (version 8.3.2) : Un framework de test pour Node.js utilisé pour écrire et exécuter des tests unitaires.

[2] Sequelize est un ORM (Object-Relational Mapping) qui permet de simplifier l'interaction avec la base de données.

- **swagger-jsdoc** : Une bibliothèque utilisée pour générer la documentation OpenAPI (Swagger) à partir de commentaires de code.
- **swagger-ui-express** (version 4.1.6) : Un middleware qui permet de visualiser la documentation OpenAPI (Swagger) dans Express.

- Base de données :
  - **pg** (version 8.8.0) : Un pilote Node.js pour la connexion et la communication avec une base de données PostgreSQL.
  - **sequelize** (version 6.27.0) : Un ORM (Object-Relational Mapping) pour la gestion des bases de données relationnelles.
- Utilitaires ou fonctions de manipulation de données :
  - **axios** (version 0.21.1) : Une bibliothèque JavaScript utilisée pour effectuer des requêtes HTTP depuis le backend.
  - **cors** (version 2.8.5) : Un middleware qui permet la gestion des requêtes CORS[3] dans Express.
  - **lodash** (version 4.17.21) : Une bibliothèque utilitaire qui fournit des fonctions de manipulation de données.

2. **Frontend :**
   - Manipulation de dates :
     - @date-io/date-fns (version 1.3.13) : Fournit des utilitaires de gestion de dates pour React, basés sur la bibliothèque date-fns.
     - date-fns (version 2.21.3) : Une bibliothèque JavaScript pour la manipulation de dates et d'heures.
   - Gestion des interfaces utilisateur (UI) :

[3] CORS (Cross-Origin Resource Sharing) est un mécanisme de sécurité utilisé par les navigateurs web pour contrôler les requêtes entre différents domaines.

        - @material-ui/core (version 4.11.3) : Un framework UI qui fournit des composants réutilisables pour la construction de l'interface utilisateur.
        - @material-ui/icons (version 4.11.2) : Contient une collection d'icônes prêtes à l'emploi pour une utilisation dans l'interface utilisateur.
        - @material-ui/lab : Une bibliothèque supplémentaire de composants et de fonctionnalités pour Material-UI.
        - @material-ui/pickers (version 3.3.10) : Fournit des composants de sélection de date et d'heure pour Material-UI.
    - Gestion des requêtes HTTP :
        - axios (version 0.21.1) : Une bibliothèque JavaScript utilisée pour effectuer des requêtes HTTP depuis le navigateur.
    - Bibliothèques de graphiques et de visualisation de données :
        - highcharts (version 9.1.0) : Une bibliothèque de graphiques JavaScript pour la visualisation de données
    - Bibliothèques de développement et de configuration :
        - @babel/core (version 7.13.16) : Utilisé pour la compilation de code JavaScript moderne en une version compatible avec les navigateurs.
        - react (version 17.0.2) : Une bibliothèque JavaScript pour la construction d'interfaces utilisateur réactives.
        - react-dom (version 17.0.2) : Utilisé pour le rendu des composants React dans le navigateur.
        - react-router-dom (version 5.2.0) : Une bibliothèque de routage pour React qui permet de gérer la navigation entre les pages.

2 Arborescences

2.1 Globale

L'Outil Facturation est une application multi-conteneur qui utilise une architecture basée sur un monorepo regroupant à la fois le backend et le frontend.

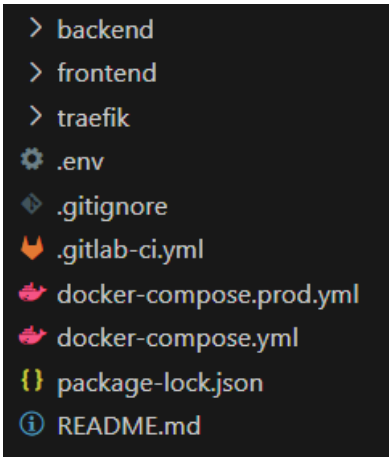


Figure 1 : Arborescence à la racine du projet

Dans le dossier « traefik » on retrouve la configuration du reverse proxy Traefik utilisé pour interroger des API externes.
Cette configuration se met dans un fichier YAML[4]. Ce fichier indique les middlewares à utiliser, les en têtes d'autorisation et les tokens nécessaires. On y trouve aussi les routes à emprunter, cela signifie qu'on y spécifie les préfixes d'URL et les règles de routage qui déterminent comment les requêtes doivent être acheminées vers les API externes.

Les fichiers « docker-compose.yml » et « docker-compose.prod.yml » décrivent et configurent les conteneurs nécessaires à l'exécution de l'application pour les environnements de développement ou de production.

[4] YAML est un format de fichier texte utilisé pour échanger des données entre diverses applications.

Le fichier « .gitlab-ci.yml » est utilisé pour le CI/CD (Continuous Integration and Continuous Deployment) dont je parlerais ci-après.

2.2 Backend

Le backend est structuré en suivant le modèle RCS (Router, Controller, Service).

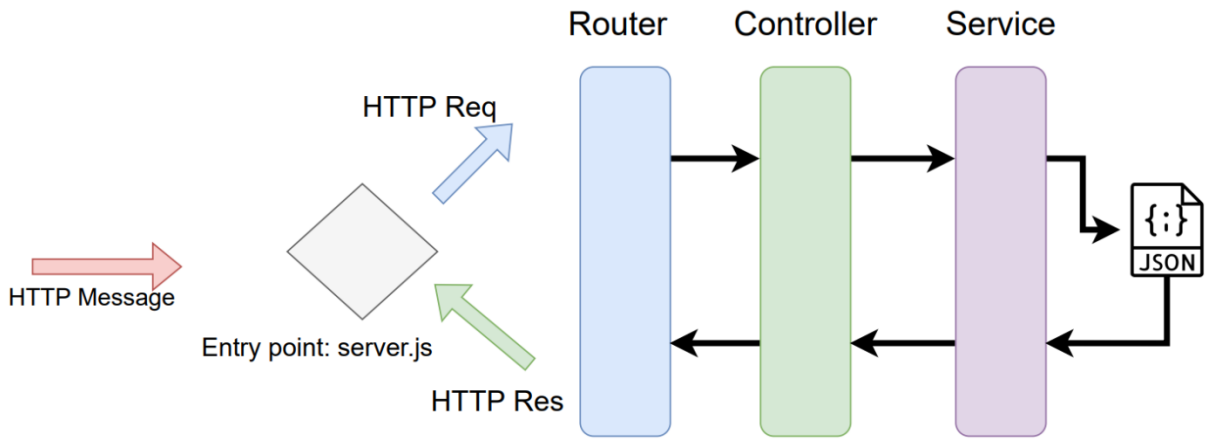


Figure 2 : Schéma du modèle RCS

Dans ce modèle, le « router » (routeur en français) est responsable de la gestion des routes de l'application. Il définit les points d'entrée pour les requêtes HTTP et dirige ces requêtes vers les contrôleurs appropriés. Le routeur détermine quelle action doit être exécutée en fonction de l'URL et des méthodes HTTP (GET, POST, PUT, DELETE, etc.) reçues. Chaque routeur est placé dans le dossier « Routers ».

Le « controller » (contrôleur en français) a pour responsabilité de traiter les requêtes reçues du routeur et d'effectuer les opérations nécessaires pour fournir une réponse. Pour cela, il interagit avec les services. Il se situe dans le dossier « Controllers ».

Le « service » contient des méthodes qui effectuent des opérations complexes et interagissent avec les modèles de données ou d'autres services. Il réalise les manipulations nécessaires et renvoie les résultats ou les données traitées au contrôleur. Les services sont regroupés dans le dossier « Services ».

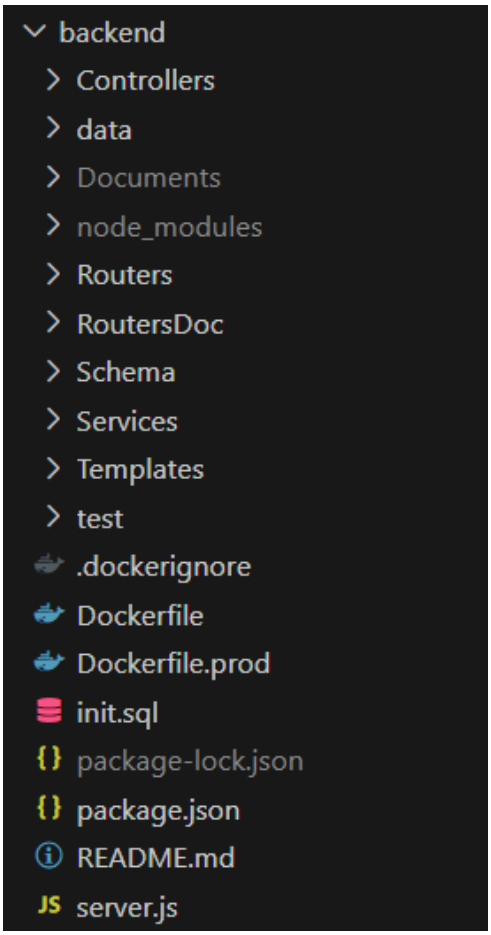


Figure 3 : Arborescence du backend

Dossier « Templates » contient les Templates Word.

Dossier « RoutersDoc » contient la documentation Swagger.

Dossier « data » regroupe les modèles définis par Sequelize et la connexion à la base de données.

Le fichier « server.js » contient la configuration et le démarrage du serveur. C'est le point de départ pour l'exécution du backend.

Les Dockerfile sont utilisés lors de la construction de chaque environnement. On en retrouve aussi dans le frontend.

2.3 Frontend

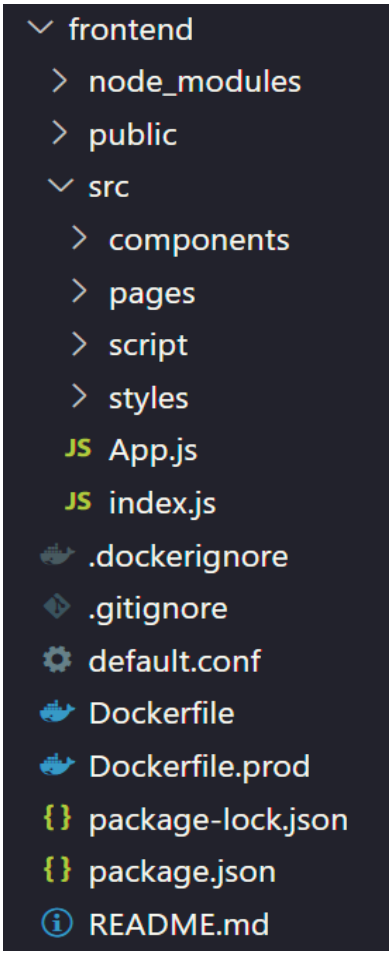


Figure 4 : Arborescence du frontend

Dossier « components » regroupe tous les composants ReactJS utilisés dans les pages de l'application.

Dossiers « pages » contient les pages statiques de l'application.

Dossiers « script » rassemble les fichiers responsables de l'envoi de requêtes vers le backend pour donner suite à des actions effectuées par l'utilisateur.

Le fichier « App.js » joue le rôle de point d'entrée du frontend et s'occupe de la redirection vers la page appropriée.

Le fichier « index.js » configure l'application ReactJS.

3 Explications techniques

3.1 Base de données

Le MCD (Modèle Conceptuel de Données) est constitué de cinq tables qui représentent des entités. Les deux autres tables sont des tables d'association utilisées pour modéliser les relations entre les entités principales.

L'application organise et gère les données en fonction des années. Cela implique que toutes les tables de l'application, telles que la table « configuration » et la table « roles_price », doivent avoir des enregistrements cohérents pour chaque année. Par exemple, il ne peut pas y avoir une année 2025 dans la table « configuration » sans qu'elle soit également présente dans la table « roles_price ».

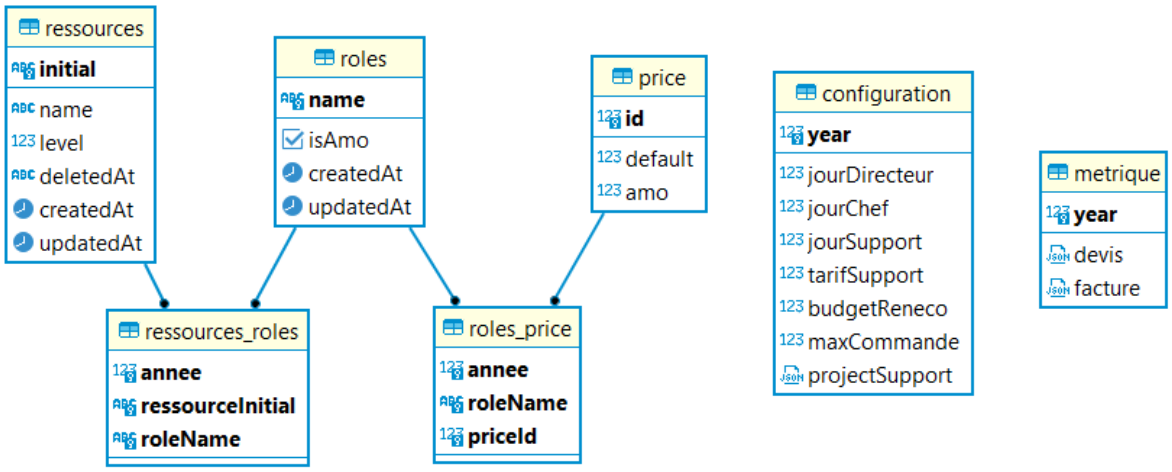


Figure 5 : MCD de l'application

Nous utilisons l'ORM Sequelize pour interagir avec la base de données. Le code ci-dessous permet d'initialiser une instance de Sequelize qui sera utilisée pour effectuer ces interactions.

```
const sequelize = new Sequelize(
  `postgres://${user}:${pwd}@${host}:${port}/${bdd}`,
  { logging: mode === "test" ? false : true }
);
```

Figure 6 : Initialisation d'une instance Sequelize

Pour créer les tables dans la base de données, nous utilisons des modèles Sequelize. Ces modèles servent à informer Sequelize de tous les détails relatifs à chaque table : son nom, ses attributs avec leurs types, ses associations avec d'autres tables, etc.

```
import { DataTypes } from 'sequelize';
import { sequelize } from '../db.js';

const metric = sequelize.define('metrique', {
    year: {
        field: 'year',
        type: DataTypes.INTEGER, // Colonne "year" de type entier
        primaryKey: true // Définit la clé primaire de la table comme étant la colonne "year"
    },
    devis: {
        field: 'devis',
        type: DataTypes.JSON // Colonne "devis" de type JSON
    },
    facture: {
        field: 'facture',
        type: DataTypes.JSON // Colonne "facture" de type JSON
    }
}, {
    tableName: "metrique", // Nom de la table dans la base de données
    timestamps: false // Désactivation des timestamps automatiques (createdAt, updatedAt)
});

export { metric };
```

Figure 7 : Modèle Sequelize de la table « métrique »

3.2 Docker

Quand on lance la commande Shell « docker-compose up » le processus suivi est le suivant pour l'environnement de développement :

- Docker Compose lit le fichier « docker-compose.yml » pour définir la configuration des services.

- Pour chaque service défini, il vérifie si l’image Docker correspondante existe localement et la télécharge si nécessaire.
- Docker Compose crée ensuite des conteneurs distincts pour chaque service et les démarre dans l'ordre spécifié dans le fichier « docker-compose.yml ».
- Les commandes définies dans les fichiers Dockerfile de chaque service sont exécutées pour construire les images des conteneurs.

Lorsqu’on est dans un environnement de production, le même processus est suivi à l’exception près que les fichiers pris en compte sont le « docker-compose.prod.yml » et les « Dockerfile.prod ».

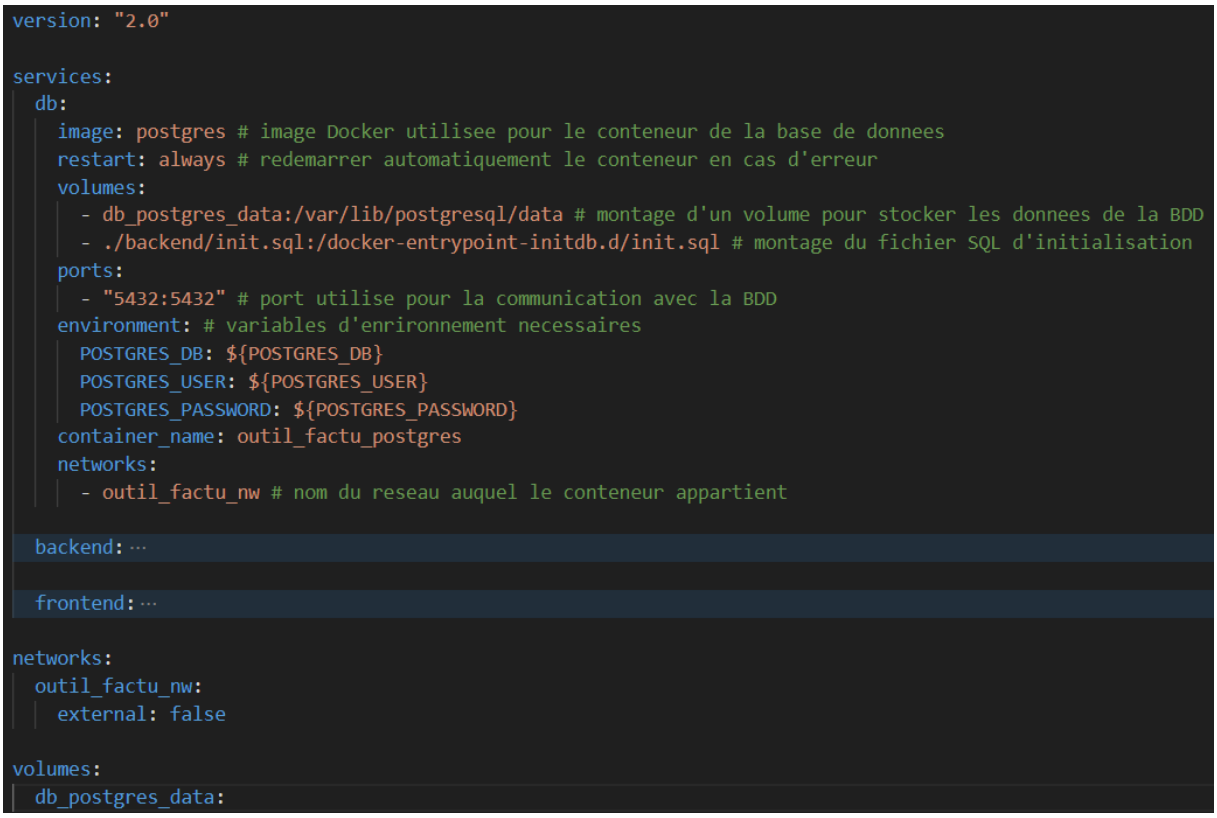
Dans les fichiers « docker-compose.yml » et « docker-compose.prod.yml » on retrouve une section pour chaque service :

- La base de données (db).
- Le backend.
- Le frontend.

Pour chacune de ces sections, on spécifie des informations telles que l'image Docker à utiliser, les variables d'environnement nécessaires, les ports utilisés, les volumes montés pour le stockage des données, et le nom du conteneur. Le nom du réseau auquel chaque conteneur appartient est également indiqué. Cela permet aux différents conteneurs de communiquer entre eux en utilisant ce réseau.

Le fichier « docker-compose.yml » est utilisé pour configurer l’environnement de développement tandis que le fichier « docker-compose.prod.yml » configure les conteneurs de l’environnement de production.

Ces deux fichiers présentent donc plusieurs différences telles que la configuration des volumes, le nom du réseau, etc.



```yaml
version: "2.0"

services:
  db:
    image: postgres # image Docker utilisee pour le conteneur de la base de donnees
    restart: always # redemarrer automatiquement le conteneur en cas d'erreur
    volumes:
      - db_postgres_data:/var/lib/postgresql/data # montage d'un volume pour stocker les donnees de la BDD
      - ./backend/init.sql:/docker-entrypoint-initdb.d/init.sql # montage du fichier SQL d'initialisation
    ports:
      - "5432:5432" # port utilise pour la communication avec la BDD
    environment: # variables d'enrironnement necessaires
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    container_name: outil_factu_postgres
    networks:
      - outil_factu_nw # nom du reseau auquel le conteneur appartient

  backend: ···

  frontend: ···

networks:
  outil_factu_nw:
    external: false

volumes:
  db_postgres_data:
```

Figure 8 : Extrait du fichier docker-compose.yml

Le projet comprend un total de quatre Dockerfile. Chaque Dockerfile est utilisé pour décrire les instructions, les dépendances et les configurations nécessaires à la création d'une image Docker spécifique. Dans le backend comme dans le frontend il y a un Dockerfile pour l'environnement de production et un pour l'environnement de développement.

```
# Utilise l'image de base Node.js version 15
FROM node:15

# Définit le répertoire de travail dans l'image
WORKDIR /app

# Met à jour npm vers la dernière version
RUN npm update -g npm

# Copie le fichier package.json dans le répertoire de travail de l'image
COPY package.json ./

# Installe les dépendances du projet définies dans package.json
RUN npm install

# Installe nodemon pour faciliter le développement
RUN npm install -g nodemon

# Installe mocha pour les tests
RUN npm install -g mocha

# Expose le port 778 pour la communication avec l'extérieur
EXPOSE 778

# Commande par défaut à exécuter lorsque le conteneur démarre
CMD npm run dev
```

Figure 9 : Dockerfile du backend (environnement de développement)

3.3 CI/CD

Le CI/CD est une approche qui permet d'augmenter la fréquence de distribution des applications grâce à l'introduction de l'automatisation au niveau des étapes de développement. L'intégration continue et le déploiement continu sont les principaux concepts du CI/CD. En effet, cette approche résout les problèmes liés à l'intégration de nouveaux segments de code, également connus sous le nom « d'integration hell » (l'enfer de l'intégration), rencontrés par les équipes de développement et de déploiement.

```yaml
stages:
  - build

build_frontend:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:v1.9.0-debug  # Image utilisée pour la construction
    entrypoint: [""]  # Point d'entrée vide pour exécuter la commande personnalisée
  script: # Commande pour construire l'image du frontend ···
  tags:
    - NSCICDK8S  # Balise spécifiant où exécuter ce travail de construction
  only:
    refs:
      - main  # Le travail ne s'exécute que lorsque les références de la branche "main" sont modifiées
      - dev  # Le travail ne s'exécute que lorsque les références de la branche "dev" sont modifiées
    changes:
      - frontend/**/*  # Le travail ne s'exécute que lorsque des changements sont effectués dans le répertoire "frontend"

build_backend: ···
```

Figure 10 : Mise en place du CI/CD

3.4 Génération de documents

Pour calculer le devis, l'utilisateur doit d'abord sélectionner les projets et la commande concernée, puis cliquer sur un bouton pour lancer le processus.

Lorsque l'action a été déclenchée, le programme utilise alors le reverse proxy Traefik pour interroger l'API de Pivotal Tracker. Lors de l'interrogation de l'API, les informations suivantes sont récupérées pour chaque projet sélectionné : le nom du projet, les stories réalisées sur ce projet pour la commande sélectionnée, ainsi que les initiales des personnes qui ont travaillé sur chaque storie et le nombre d'heures qu'elles y ont consacrées.

Une fois ces données récupérées depuis Pivotal Tracker, une requête Axios est envoyée au backend du projet, transmettant les informations précédemment obtenues comme paramètre.

Le routeur du backend appelle alors le contrôleur approprié, qui à son tour fait appel au service adéquat.

Dans ce service, un objet est créé pour remplir le Template Word du devis. Cet objet est renseigné par le montant total devisé et le montant devisé pour chaque projet contenu dans le paramètre.

Le coût d'un projet est calculé en utilisant les initiales des personnes ayant travaillé sur les stories le concernant et le nombre d'heures qu'elles y ont consacrées. Si une personne référencée dans une storie du projet pour la commande sélectionnée n'est pas trouvée, elle est enregistrée dans un tableau avec le lien de la storie où elle apparaît. À ce moment-là, à la fin de la boucle qui parcourt les projets, ce tableau est renvoyé comme une erreur qui est transmise à l'utilisateur. Si toutes les ressources ont été trouvées dans la base de données, les projets sont triés par ordre alphabétique.

Ensuite, le tarif associé au travail du chef de projet et du directeur technique est calculé, ainsi que le tarif associé au travail après la livraison du produit (activités de support).

Enfin, la fonction « createDocx » est appelée pour générer le document Word du devis.

La génération des autres documents s'effectue de façon similaire :

- Interrogation d'une API (Zendesk ou Pivotal Tracker).
- Requête Axios vers le backend.
- Le routeur appelle le contrôleur approprié, qui à son tour fait appel au service adéquat.
- Le service traite les données en fonction du type de fichier demandé.
- Le fichier est téléchargé.

Conclusion

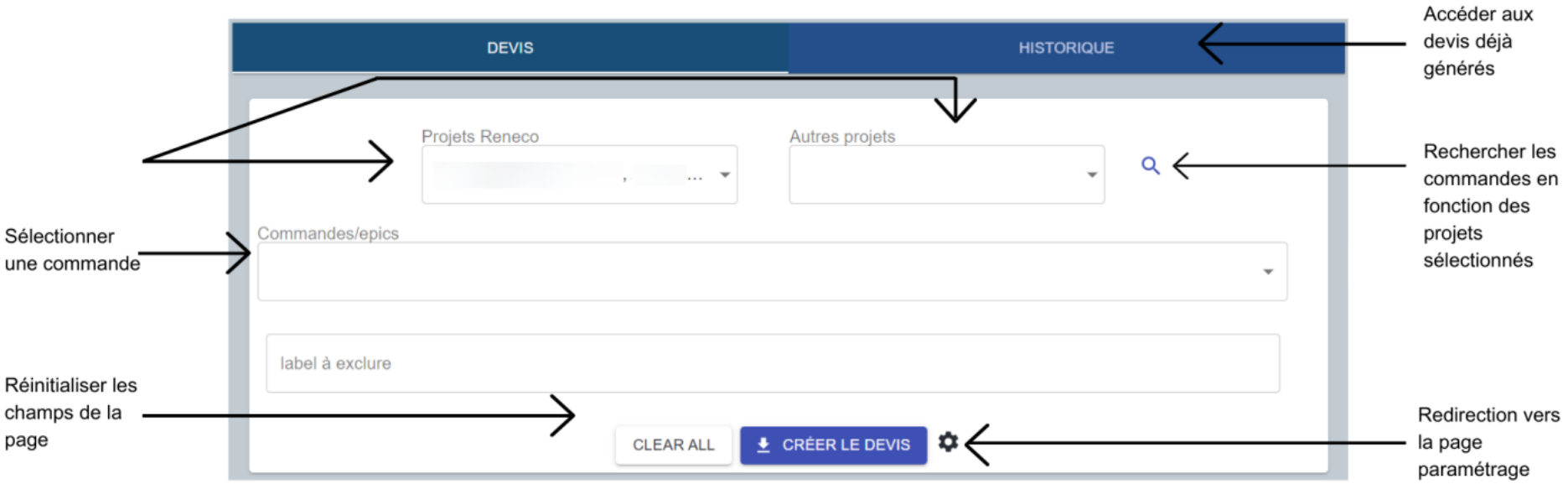
L'application utilise un monorepo qui combine à la fois le backend et le frontend ce qui permet de regrouper les deux parties du code. En utilisant Docker et le CI/CD, l'application est déployée sur la plateforme de l'entreprise, assurant ainsi son accessibilité.

Pour améliorer l'application, il serait utile d'ajouter une fonctionnalité de recherche dans la barre de sélection des projets. Cela permettrait aux utilisateurs de trouver rapidement le projet qu'ils recherchent, évitant ainsi la nécessité de parcourir tous les projets. D’autre part, actuellement, lorsque les utilisateurs génèrent un devis, il arrive qu'ils en simulent plusieurs. Cependant, le problème est que le montant du dernier devis généré est enregistré dans la table des métriques. Par conséquent, lorsque l'utilisateur consulte la page graphique pour voir les montants devisés pour un mois donné, le montant affiché correspond à celui du dernier devis effectué, qui peut différer de celui envoyé au client. Il serait souhaitable de permettre à l'utilisateur de choisir le montant du devis qu'il souhaite afficher dans la page graphique.

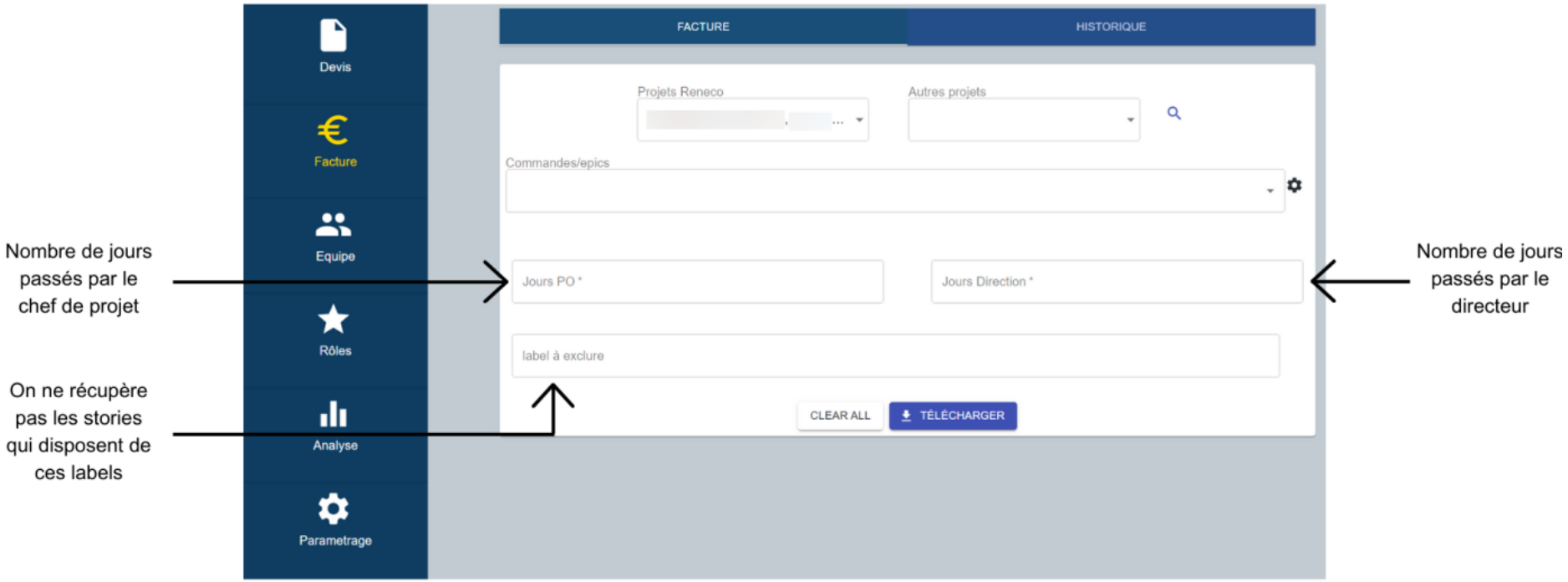
Table des illustrations

Table des annexes

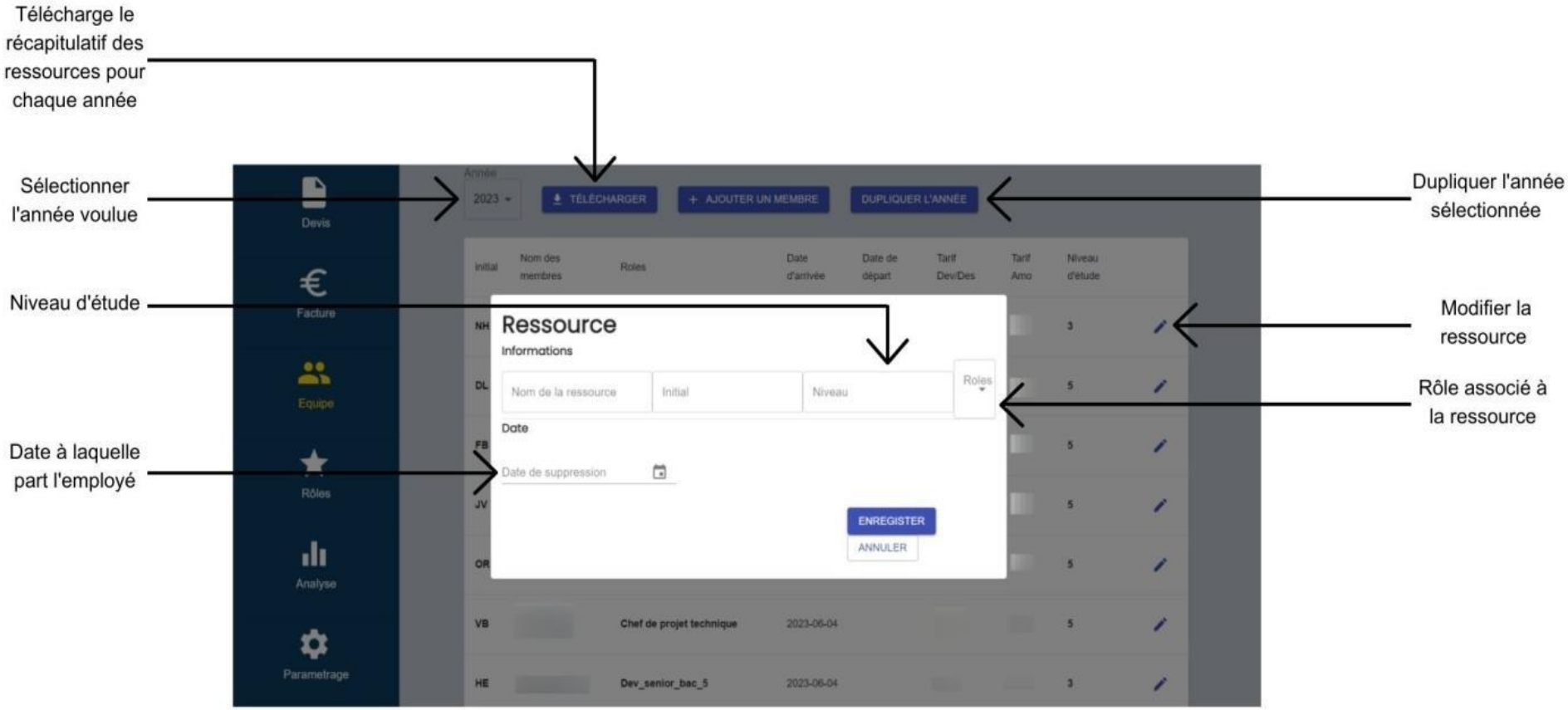
Annexe I : Interface utilisateur - Générer un devis



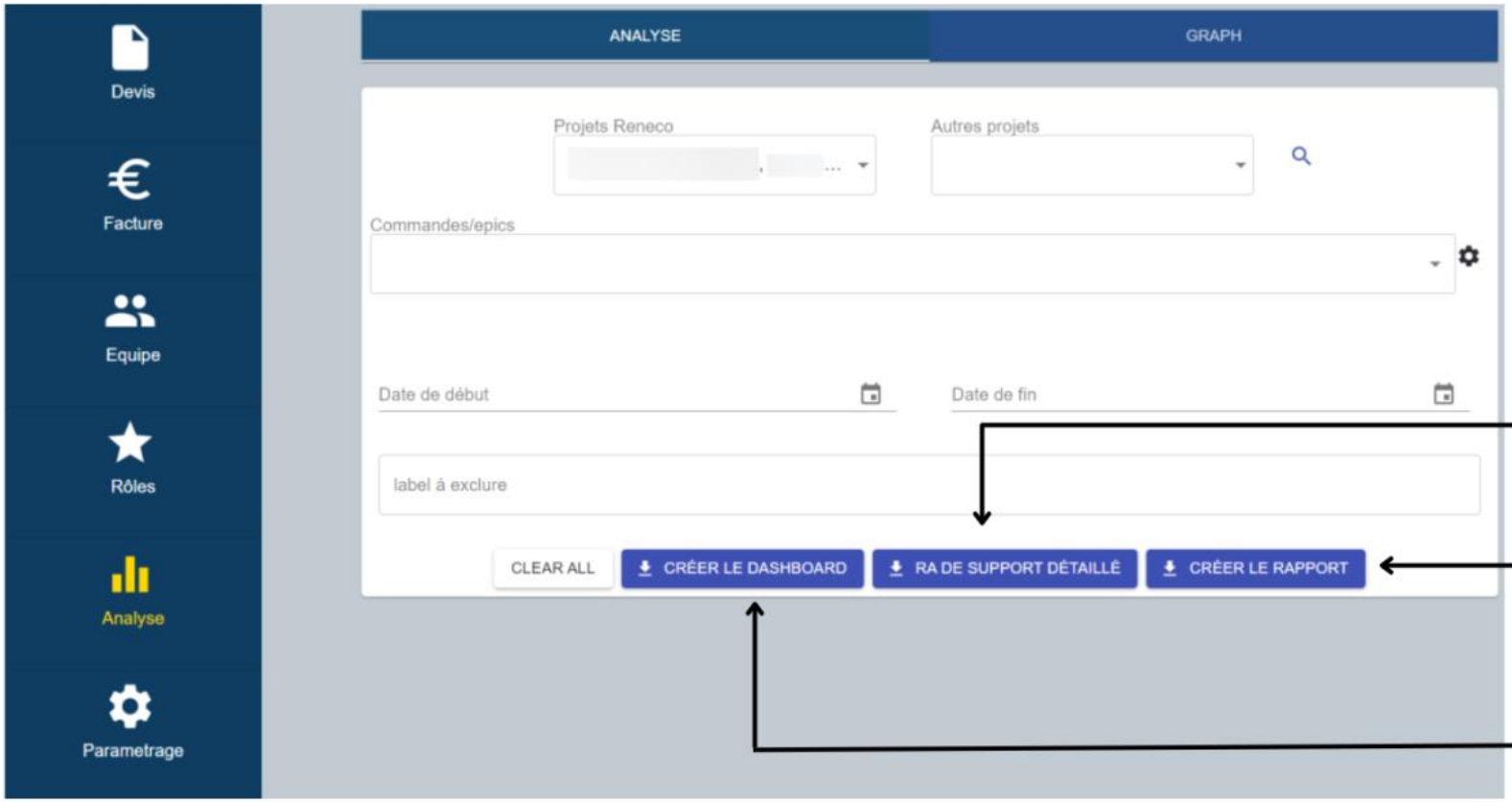
Annexe II : Interface utilisateur - Page facture



Annexe III : Interface utilisateur - Page équipe



Annexe IV : Interface utilisateur - Page analyse



Annexe V : Interface utilisateur - Page graphique