Modernisation des formulaires PHP sur de nouvelles technologies

RAPPORT TECHNIQUE

Du 16 janvier au 22 avril 2024

Entreprise : Grand Belfort

Tuteur en entreprise :
Laurent STOCKER

Architecte Coordinateur systèmes d'information – Directeur adjoint

Antoine LACHAT

3e année du BUT informatique
Option donnée, structuration et analyse

Tutrice référente

Corinne PATERLINI

Informatique

1) Il manque des sous-titres qui auraient facilité la recherche d'information.
D'ailleurs vos sous-titres sont mal présentés.
Bref, c'est assez confus.

2) Légendez toutes vos illustrations !
La plupart sont mal présentées (pb d'ancrage et/ou d'adaptation du texte).

3) S'il est utile de citer les outils utilisés, il est inutile d'expliquer leur rôle à un professionnel...

4) Certains passages sont mal construits et donc peu clairs (cf. 1.1 ou début de 1.1.2).

Table des matières

Trop vague

Trop flou : il faut des sous-parties !

Présentation de l’application

Ce rapport technique documente le processus de réalisation de plusieurs formulaires pour le Grand Belfort, qu'ils soient destinés au public ou à un usage interne. Parmi ces formulaires figurent les réservations de vélos électriques, les demandes de chèques culture, les réservations de salles de l'hôtel de ville et des formulaires de gestion communale.

Réalisé sur une période s'étendant du 16 janvier 2024 au 22 avril 2024 au sein du Grand Belfort à Belfort, ce travail vise à faciliter la reprise du projet pour d'éventuelles améliorations futures. Ce rapport offre une vue d'ensemble du fonctionnement global du projet en mettant en lumière les prérequis et enfin dans les détails spécifiques de certaines parties clés de formulaire.

1. Prérequis

Les sous-titres doivent être mis en évidence !

1.1 Structure des projets

Il faut un sous-titre

Expliquez ce que c'est

Pour commencer, je vais parler de la structure des projet suivants : Check culture, dépôt sauvage, mon vélo électrique, signalétique et réservation de salle de l'hôtel de ville.

DE QUOI PARLEZ-VOUS ????
On ne le comprend que quelques lignes après !

Ces formulaires ont été développés avec un front-end différent des deux autres. Ils ont été construits en utilisant EJS (Embedded JavaScript), une technologie permettant de générer dynamiquement du HTML côté serveur. Chaque projet est organisé dans un seul répertoire, connectant le front-end au back-end. Ils ont été conçus selon le modèle MVC (Modèle, Vue, Contrôleur).

Pour les formulaires SMGPAP et GBCA : 1) traduisez
2) ce doit être un sous-titre !

Les front-ends de ces formulaires ont été développés avec le framework Vue.js, version 3. Chaque projet est divisé en un front-end et un back-end distincts. Le back-end suit une structure MVC (Modèle, Vue, Contrôleur), tandis que le front-end adopte une structure standard propre à Vue.js.

Points Communs : à ne pas souligner

Environnement d'exécution : Tous les projets utilisent Node.js comme environnement d'exécution.

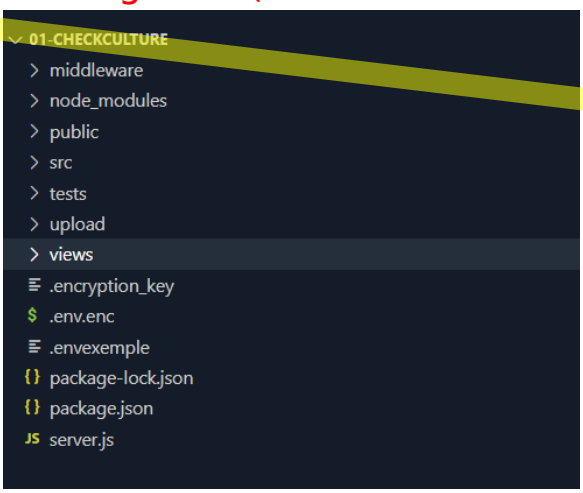
Communication avec la base de données PostgreSQL 16 : Sequelize est utilisé comme ORM pour établir les communications sécurisées et efficaces avec la base de données. Les modèles Sequelize dans chaque formulaire créer directement les tables dans PostgreSQL.

1.1 Les arborescences

1.1.1 L'arborescence pour les projets EJS

L'arborescence pour les projets réaliser avec EJS se compose de cette façon :

Il manque une légende (et inutile de l'ancrer comme caractère)



Dans cette arborescence, nous retrouvons les composants essentiels tant du côté back-end que front-end de nos projets. Voici une analyse détaillée des dossiers et fichiers clés pour assurer le bon fonctionnement de l'application :

Middleware : Le dossier middleware contient les middlewares utilisés dans l'application. Ces composants jouent un rôle crucial dans le traitement des requêtes HTTP avant qu'elles n'atteignent les routes. Ils permettent la gestion des sessions, la vérification des autorisations, ou encore la manipulation des données en entrée avant leur traitement par le serveur.

node_modules : Le dossier node_modules regroupe toutes les dépendances installées via npm (Node Package Manager). Ces dépendances sont des bibliothèques utilisées dans le projet pour des fonctionnalités telles que la gestion des routes, la validation des données, la création de test, etc.

Public : Le dossier public contient les ressources statiques accessibles au client, telles que les fichiers HTML, les fichiers CSS, les images.

src : Le dossier abrite l'ensemble du code source côté back-end. Il est organisé en plusieurs sous-dossiers et fichiers clés, chacun jouant un rôle spécifique dans le fonctionnement de l'application :

laissez moins d'espace, idem après

- **Routes** : Ce dossier contient les définitions des routes, déterminant les points d'entrée pour les requêtes HTTP. Chaque fichier de ce dossier expose les différentes routes et leurs gestionnaires associés.

- **Controller** : Le dossier Controller regroupe les contrôleurs. Ils sont chargés de traiter les requêtes HTTP reçues par les routes, en effectuant les opérations nécessaires sur les données et en renvoyant les réponses appropriées. C'est dans cette partie que sont écrites les requête Sequelize.

- **Modele (modèle Sequelize)** : Les modèles de données sont définis en utilisant Sequelize. Chaque fichier de ce dossier correspond à un modèle de base de données, décrivant sa structure et ses relations avec d'autres modèles.

- **Database** : Le dossier database contient les configurations et les scripts nécessaires à la connexion et à la gestion de la base de données.

- **sql** : Ce dossier abrite les fichiers d'insertions SQL des différentes tables permettant le bon fonctionnement.

Utils : Le dossier utils regroupe les utilitaires et les fonctions réutilisables. Il contient des fonctions pour crypter, décrypter, convertir des données, envoyé des mails, etc.

Test : Le dossier test contient les fichiers de test unitaires et d'intégration.

server.js : C'est le point d'entrée de l'application côté serveur. Il contient la logique nécessaire au démarrage et à la configuration du serveur web, ainsi que la gestion des routes et des requêtes HTTP.

package.json : Il contient la liste des bibliothèques nécessaires au bon fonctionnement du projet. Il est également utilisé pour définir des scripts personnalisés, tels que les commandes de démarrage de l'application ou de lancement des tests. Une simple installation du projet à l'aide de « npm install » les installera toutes.

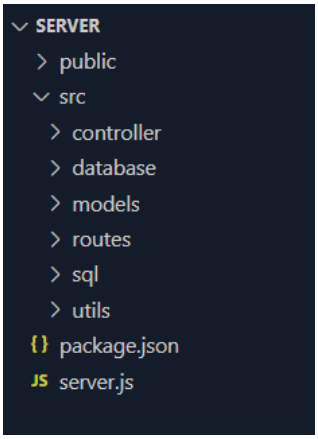
Views : Les vues correspondent à une page ou à une section de l'application, regroupant les composants nécessaires à son affichage et à son fonctionnement.

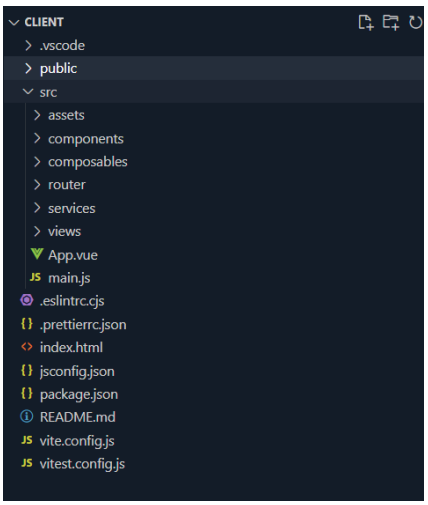
. env.enc : Ce fichier stocke les variables d'environnement chiffrées.

.encryption_key : Il contient la clé nécessaire pour déchiffrer ces données

1.1.2 L'arborescence pour les projets Vue.js

Pour les projets avec un front-end réaliser en vue.js. Ils se composent comme ceci :





Légende !!

pas pratique : vous ne les avez pas légendées ni numérotées !

Pour le projet côté back-end (illustration 3), il ne sera pas nécessaire de la décrire, pluriel tous les concepts que j'ai expliqué plus haut dans « L'arborescence pour les projets EJS » reprennent la partie serveur.

Que décrivez-vous ?

Public : Le dossier public contient les ressources statiques de l'application, telles que les fichiers HTML, les images, les polices de caractères, etc.

src : Le dossier src est le cœur de l'application Vue.js, regroupant le code source côté front-end. Il est subdivisé en plusieurs sous-dossiers et fichiers clés :

Assets : Ce dossier contient les ressources statiques utilisées par l'application, telles que les images, les icônes, les polices de caractères, etc.

Components : Les composants réutilisables de l'application sont regroupés dans ce dossier. Chaque composant encapsule une partie spécifique de l'interface utilisateur.

Composables : Les composables sont des fonctions Vue.js qui encapsulent la logique et les fonctionnalités réutilisables dans toute l'application.

Router : Ce dossier contient la configuration du router Vue Router. Il définit les différentes routes de l'application, associant chaque route à un composant Vue spécifique.

Services : Les services de l'application, tels que les appels API ou les gestionnaires d'état globaux, sont regroupés dans ce dossier.

Views : Les vues de l'application sont définies dans ce dossier. Chaque vue correspond à une page ou à une section de l'application.

App.vue : Le fichier App.vue est le composant racine de l'application Vue.js. Il sert de conteneur principal pour tous les autres composants et vues de l'application. Ce

fichier définit la structure de base de l'application, y compris la mise en page globale, les en-têtes, les pieds de page, etc.

Index.html : Il est utilisé pour définir la structure de base de la page web et pour inclure les balises <script> nécessaires au chargement de l'application Vue.js.

Vite.config.js : Le fichier vite.config.js est le fichier de configuration de l'outil de build Vite. Il est utilisé pour personnaliser et configurer le comportement de Vite lors de la construction de l'application. Ce fichier peut contenir des options de configuration telles que la spécification des chemins d'entrée et de sortie, la configuration des plugins.

oh !

1.2 Présentation du outils techniques utilisés

pluriel

Les différents formulaires utilisent plusieurs outils techniques et sont détaillé à la suite :

S'il est utile de citer les outils utilisés, il est inutile de les expliquer : vous vous adressez à un professionnel

Framework Vue.js (version 3) : C'est est un framework JavaScript moderne et réactif, offrant une syntaxe simple et intuitive, des performances élevées, une excellente documentation, et une communauté active. Il permet de construire des interfaces utilisateur interactives et dynamiques de manière efficace, grâce à sa flexibilité et à sa modularité.

Sequelize : C'est un ORM (Object-Relational Mapping) pour Node.js, qui facilite l'interaction avec une base de données relationnelle. Il permet d'écrire des requêtes en utilisant des objets JavaScript plutôt que du SQL brut, ce qui rend le code plus expressif, sécurisé et facile à lire.

Node.js : C'est un environnement d'exécution JavaScript côté serveur, construit sur le moteur JavaScript V8 de Chrome. Il permet d'écrire des applications JavaScript côté serveur, en utilisant le même langage des deux côtés du développement (côté client et côté serveur). Il offre de grande performance en termes de rapidité et de sa scalabilité. Il propose une large bibliothèque de modules via npm.

PostgreSQL (version 16) : C'est un système de gestion de base de données relationnelle open-source, robuste et performant. Connu pour sa fiabilité, sa conformité aux standards, et ses fonctionnalités avancées, PostgreSQL est largement utilisé dans le développement d'applications web ce qui en fait un bon choix.

EJS : C'est un moteur de templates JavaScript qui permet d'incorporer du code JavaScript dans des fichiers HTML pour générer des pages web dynamiques côté serveur. Il offre une syntaxe simple et familière, ainsi que la création de modèles réutilisables et la manipulation de données dynamiques.

1.3 Explication générale

Il faut des titres précis !

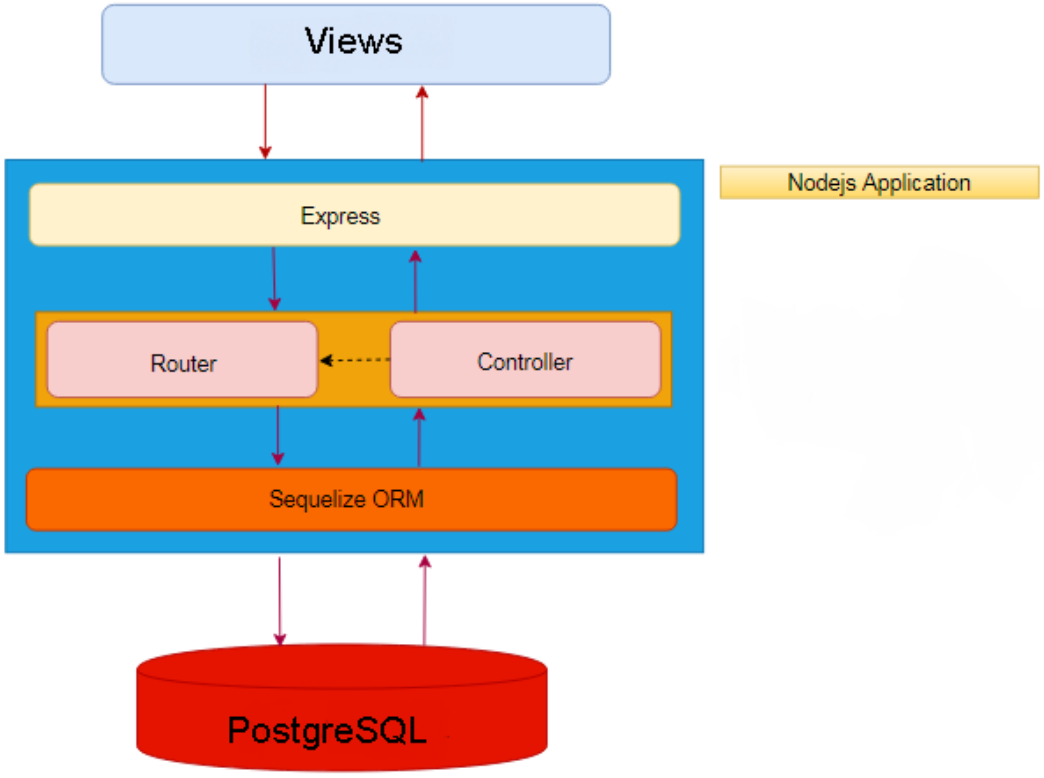


Figure 1- Schéma des différentes interactions entres les modules

Dans une application web avec Node.js, Express, Sequelize, PostgreSQL, le processus débute lorsque le navigateur web envoie une requête HTTP vers le serveur

Node.js. Express, qui agit comme un gestionnaire de routes, reçoit cette requête et la dirige vers la route appropriée. Cette route est associée à un contrôleur spécifique, qui contient la logique des actions.

Le contrôleur peut alors interagir avec la base de données PostgreSQL via Sequelize pour récupérer, mettre à jour ou supprimer des données selon les besoins. Une fois que les données sont manipulées, le contrôleur renvoie une réponse appropriée à Express, qui l'envoie ensuite au navigateur sous la forme d'une vue HTML. Le navigateur affiche ensuite cette vue à l'utilisateur. Ce processus se répète à chaque fois qu'une requête est effectuée.

2. Explication de codes

Sous-titre ?

Pour chaque formulaire, il y a un fichier nommé « decryptENV.js » dans le répertoire « utils » qui permet de crypter les données importantes contenu dans le fichier .env contenant les variables d'environnements nécessaire à la connexion à la base de données :

inutile d'ancrer au caractère cette figure !

```
const forge = require('node-forge');
const fs = require('fs');

// Fonction pour chiffrer un fichier
function encrypt() {
    generateKey()
    const input = fs.readFileSync('../../.env', 'utf8');
    const key = loadKey()
    const iv = forge.random.getBytesSync(16);

    const cipher = forge.cipher.createCipher('AES-CBC', key);
    cipher.start({ iv: iv });
    cipher.update(forge.util.createBuffer(input, 'utf8'));
    cipher.finish();

    const encrypted = forge.util.encode64(cipher.output.getBytes());
    fs.writeFileSync('../../.env.enc', iv + encrypted, 'utf8');

    // Suppression du .env
    fs.unlinkSync('../../.env');
}

// Charger la clé stockée
function loadKey() {
    return fs.readFileSync('../../.encryption_key', 'binary');
}

function generateKey() {
    const key = forge.random.getBytesSync(32);
    fs.writeFileSync('../../.encryption_key', key, 'binary');
}

encrypt()
```

Figure 2- Algorithme de cryptage

Ce code utilise les modules « node-forge » et « fs » pour chiffrer un fichier.

- « fs » permet de lire, d'écrire, modifier ou supprimé des fichiers.
- « node-forge » permet de générer des clés, de chiffrer et de déchiffrer des données.

Deux fonctions sont définies : « generateKey() » génère une clé de chiffrement, alors que « loadKey() » charge la clé depuis un fichier. Dans la fonction « encrypt() », une clé est générée et le contenu du fichier « .env » est lu.

Ensuite, un vecteur d'initialisation aléatoire est généré, et le contenu du fichier est chiffré à l'aide de la clé et du vecteur d'initialisation avec l'algorithme AES-CBC. Le contenu chiffré est ensuite enregistré dans un nouveau fichier « .env.enc », puis le fichier original « .env » est supprimé.

Pour le déchiffrement : Sous-titre

```
const forge = require('node-forge');
const fs = require('fs');
const password = 'votreMotDePasse'; // Utilisez le mot de passe utilisé pour encrypter

/**
 * Dechiffrage du fichier .env.enc
 * Parmétrage des variables d'environnement sans .env brut
 */
function decrypt() {
    const encrypted = fs.readFileSync('.env.enc', 'utf8');
    const iv = encrypted.slice(0, 16);
    const encryptedData = encrypted.slice(16);
    const key = loadKey();

    const decipher = forge.cipher.createDecipher('AES-CBC', key);
    decipher.start({ iv: iv });
    decipher.update(forge.util.createBuffer(forge.util.decode64(encryptedData)));
    decipher.finish();

    const decrypted = decipher.output.toString('utf8');
    const envVariables = decrypted.split('\n');
    envVariables.forEach(line => {
        let [key, value] = line.split('=');
        if(key && value) {
            if(value.includes('\r')) value = value.replace('\r', '');
            process.env[key] = value;
        }
    });
}

// Charger la clé stockée
function loadKey() {
    return fs.readFileSync('.encryption_key', 'binary');
}

module.exports = { decrypt };
```

Figure 3- algorithme de décryptage

présentation à revoir !

Tout d'abord, il charge le contenu chiffré du fichier '.env.enc' et extrait le vecteur d'initialisation (IV) et les données chiffrées.

Ensuite, il charge la clé de chiffrement à partir du fichier '.encryption_key'. Utilisant la bibliothèque 'node-forge', il crée un déchiffreur avec l'algorithme AES-CBC et la clé chargée, puis il procède au déchiffrement des données.

Une fois les données déchiffrées, elles sont transformées en variables d'environnement et configurées dans le processus en cours. Cela permet de charger les variables d'environnement sans avoir besoin du fichier brut '.env', ce qui renforce la sécurité en évitant d'exposer des informations sensibles en texte clair.

Sous-titre ?

Pour certains formulaires qui possèdent une partie administration comme pour « check culture » et « Réservation d'un vélo électrique ». Il y a un fichier nommé « gestionAdmin.Js » dans le répertoire « utils » qui permet de vérifier que le mot de passe rentré par l'admin est correct à celui crypté dans la base de données. Si tout se passe bien, un jeton d'authentification lui sera approprié.

md

```
async function authAdmin(req, res, next) {
    const providedPassword = req.body.password;

    try {
        const hashedProvidedPassword = crypto.createHash('sha256').update(providedPassword).digest('hex');
        console.log("hashedProvidedPassword", hashedProvidedPassword);

        const admin = await Admin.findOne({ where: { username: 'admin' } });
        console.log("admin", admin);

        if (!admin) {
            return res.status(401).send({ message: 'Unauthorized: Admin not found' });
        }
        const hashedPasswordFromDatabase = admin.password_hash.replace('\\x', '');
        console.log("admin", hashedPasswordFromDatabase);
        console.log("admin", hashedProvidedPassword );

        if (hashedPasswordFromDatabase === hashedProvidedPassword) {
            console.log("Administrateur authentifié");
            const token = jwt.sign({ admin: true }, jwtSecretKey, { expiresIn: '1h' });
            res.cookie('adminToken', token, { httpOnly: true });
            next();
        } else {
            console.log("Mot de passe incorrect");
            return res.status(401).send({ message: 'Unauthorized: Incorrect password' });
        }

    } catch (error) {
        console.error('Erreur lors de la vérification du mot de passe :', error);
        return res.status(500).send({ message: 'Internal Server Error' });
    }
}
```

Figure 4 - algorithme d'authentification d'un administrateur

Lorsqu'une requête d’authentification est reçue, la fonction récupère le mot de passe fourni par l'utilisateur à partir du corps de la requête. Ensuite, elle utilise l'algorithme de hachage SHA-256 pour hasher ce mot de passe fourni.

Traduisez mieux : cryptage / crypter

Elle cherche dans la base de données un administrateur avec le nom d'utilisateur « admin ». Si aucun administrateur n'est trouvé, elle renvoie une réponse avec le statut 401 indiquant une erreur d'authentification.

Sinon, elle compare le mot de passe haché trouvé dans la base de données avec le mot de passe haché fourni par l'utilisateur. Si les deux correspondent, l'administrateur est authentifié avec succès. Dans ce cas, un jeton d'authentification est généré à l'aide de JWT (JSON Web Token), signé avec une clé secrète, et envoyé dans un cookie HTTP avec la réponse. Sinon, une réponse avec le statut 401 est renvoyée, indiquant un mot de passe incorrect.

En cas d'erreur lors de l'exécution de la fonction, une réponse avec le statut 500 est renvoyée, indiquant une erreur interne du serveur.

Table des illustrations

Il en manque !