ChatGPT

Rapport de Stage CDA – Projet MABOTE

Auteur : Mustapha Mansouri

Sommaire

Introduction

Le projet **MABOTE** est une application web de prise de rendez-vous et de gestion de contenu, conçue dans le cadre d'un stage de fin de formation *Concepteur Développeur d'Applications* (CDA). L'objectif était de fournir à l'institut de beauté MABOTÉ un outil numérique permettant aux clients de planifier des soins en ligne (7j/7, 24h/24) et à l'administrateur de gérer simplement ses services et son contenu. Concrètement, le système comprend :
- une interface **client** (front-office) pour réserver des soins (massages, soins du visage, etc.) et consulter les informations du salon ;
- une interface **administrateur** (back-office) sécurisée pour gérer les rendez-vous (validation, annulation) et mettre à jour le contenu éditorial (catalogue de services, actualités, horaires).

Le développement a utilisé une architecture web moderne : une application front-end en **Vue.js** consommant une API **RESTful** développée avec **Node.js/Express**, connectée à une base de données **MySQL** via l'ORM **Sequelize**. Ce découplage clair entre client et serveur facilite la maintenance et l'évolutivité de l'application.
Ce rapport détaille les aspects fonctionnels, techniques et organisationnels du projet MABOTE, ainsi que les compétences mises en œuvre lors de sa réalisation.

Contexte et enjeux

L'institut de beauté **MABOTÉ** (implanté à HEM) souhaitait moderniser la gestion de son planning et améliorer son service client. Jusque-là, les prises de rendez-vous s'effectuaient uniquement par

téléphone, ce qui limitait les créneaux disponibles et augmentait le risque de doubles réservations. La gérante souhaitait également pouvoir mettre à jour elle-même le contenu du site web (descriptions des soins, tarifs, promotions en cours, etc.) sans recours à un prestataire technique extérieur. Ce besoin de digitalisation a donné naissance au projet **MABOTE**.

L'entreprise imposait en outre des contraintes fortes : le stage de courte durée (un mois) limitait le périmètre fonctionnel, et aucun budget n'était prévu pour des outils payants. Les solutions retenues devaient être open-source et légères. De plus, l'absence d'équipe technique chez le client nécessitait une documentation claire du déploiement. Enfin, d'un point de vue légal, l'application devait se conformer au RGPD pour la gestion des données personnelles des clients.

Analyse des besoins

Après avoir défini le contexte, un **cahier des charges** structuré a été rédigé. Les principaux objectifs et fonctionnalités du projet sont les suivants :

- **Réservation en ligne** : permettre aux clients de consulter la liste des services offerts et de prendre rendez-vous en ligne en quelques clics, avec confirmation immédiate. Les créneaux disponibles sont affichés dynamiquement.
- **Gestion du planning** : pour l'administrateur, fournir un espace sécurisé (login/mot de passe) permettant de visualiser l'agenda des rendez-vous, de confirmer ou d'annuler des demandes, et d'ajouter/modifier/supprimer des rendez-vous.
- **Administration du contenu** : offrir des outils d'édition pour gérer les services proposés (nom, description, durée, prix) ainsi que le contenu du site (actualités, textes de présentation, horaires, galerie d'images).
- **Sécurité et confidentialité** : protéger les données personnelles des utilisateurs (comptes clients, mots de passe, etc.) via un chiffrement adapté, sécuriser les endpoints de l'API et se conformer au RGPD (consentement pour le stockage des données).
- **Interface responsive et conviviale** : concevoir une interface utilisateur moderne, intuitive et adaptée aux ordinateurs comme aux mobiles.
- **Extensibilité** : prévoir la possibilité de futures évolutions (par exemple, envoi de rappels par email/SMS ou prise en charge de plusieurs employés).

Ces besoins ont été validés avec l'équipe pédagogique et l'entreprise, puis traduits en maquettes fonctionnelles (réalisées sur papier et sous Figma) avant le développement.

Architecture technique

L'architecture du projet est de type **client-serveur** en couches. Le front-end en Vue.js est totalement indépendant du back-end Express/Node : il consomme l'API REST en JSON. Le serveur back-end est structuré en modules : routes, contrôleurs et modèles (ORM Sequelize). La base de données MySQL constitue la couche de persistance. Ce modèle découplé permet de séparer l'application en services distincts (gestion des utilisateurs, des services, des rendez-vous, du contenu éditorial) et facilite la maintenance. Par exemple, les requêtes CRUD sont exposées via des routes `/api/services`, `/api/rendezvous`, etc. Côté front-end, Vue Router gère la navigation entre les vues (page d'accueil, formulaire de réservation, interface admin, etc.), et Axios effectue les appels API.

Cette architecture RESTful assure une séparation claire des responsabilités : l'interface web se concentre sur l'ergonomie et l'affichage réactif, tandis que l'API gère la logique métier. La structure du code back-end (fichiers `app.js`, dossiers `routes/`, `controllers/`, `models/`) suit les bonnes pratiques d'un projet Express. Les configurations (accès à la base, clés secrètes, URL, etc.) sont gérées via un fichier `.env`. À terme, cette organisation permettra d'ajouter de nouvelles fonctionnalités ou services sans impacter les autres couches.

Conception de la base de données

La base de données relationnelle a été conçue pour stocker les entités principales : **Utilisateur**, **Service**, **RendezVous** et **Contenu** (actualités). Le Modèle Conceptuel identifie d'abord ces entités et leurs relations. Un **Utilisateur** représente soit un client, soit l'administrateur (via le champ `role`). Un **RendezVous** correspond à une réservation, avec les attributs date/heure et statut (EN_ATTENTE, CONFIRME, ANNULE). Un **Service** est une prestation (nom, description, durée, prix). L'entité **Contenu** stocke les articles/actualités (titre, texte, date). Les relations sont : un utilisateur peut prendre plusieurs rendez-vous (1,N), chaque rendez-vous est lié à un seul utilisateur (client) et à un seul service (relation N-1 vers Service).

Le Modèle Logique (MLD) traduit ce MCD en tables SQL. Par exemple, la table `utilisateur` a `id_utilisateur` (clé primaire), `nom`, `email` (unique), `mot_de_passe` (hashé), `role`. La table `rendezvous` a `id_rdv`, `date_heure`, `statut`, ainsi que les clés étrangères `id_utilisateur` et `id_service` pointant vers les tables `utilisateur` et `service`. La table `service` contient `id_service`, `nom_service`, `description`, `duree`, `prix`. La table `contenu` contient `id_contenu`, `titre`, `texte`, `date_publication`.

Le script SQL (MPD) a été généré via Sequelize. Par exemple, la création de la table `utilisateur` est :

```
CREATE TABLE utilisateur (
  id_utilisateur INT AUTO_INCREMENT PRIMARY KEY,
  nom VARCHAR(100) NOT NULL,
  email VARCHAR(100) NOT NULL UNIQUE,
  mot_de_passe VARCHAR(255) NOT NULL,
  role VARCHAR(20) NOT NULL DEFAULT 'CLIENT'
);
```

Les contraintes d'intégrité (FOREIGN KEY) et les index (par exemple, index unique sur `email`, index sur `date_heure` dans `rendezvous`) ont été définis pour garantir la cohérence des données. Le code Sequelize `sync()` crée automatiquement les tables au démarrage. Par exemple, le modèle Sequelize suivant définit l'entité **Utilisateur** (fichier `models/Utilisateur.js`) :

```
const { Model, DataTypes } = require('sequelize');
class Utilisateur extends Model {}
Utilisateur.init({
  nom: { type: DataTypes.STRING(100), allowNull: false },
  email: { type: DataTypes.STRING(100), allowNull: false, unique: true },
  mot_de_passe: { type: DataTypes.STRING(255), allowNull: false },
```

```
  role: { type: DataTypes.STRING(20), allowNull: false, defaultValue:
'CLIENT' }
}, {
  sequelize,
  modelName: 'Utilisateur',
  tableName: 'utilisateur',
  timestamps: false
});
```

Les autres modèles (`Service`, `RendezVous`, `Contenu`) sont définis de manière similaire, avec leurs associations (par ex. `RendezVous.belongsTo(Utilisateur)` pour lier un RDV à son client).

Développement Back-End

Le back-end est implémenté en Node.js avec le framework Express. L'application est configurée dans un fichier `app.js` principal, qui charge les middlewares globaux (`express.json()` pour parser le JSON, `helmet()` pour la sécurité HTTP, CORS pour autoriser le front-end, etc.) et importe les routes. Le dossier `routes/` définit les endpoints de l'API : par exemple, `routes/rendezvous.js` définit les routes `/api/rendezvous` (GET, POST, PUT, DELETE) et les associe aux fonctions du contrôleur correspondant (`rendezvousController`). Le dossier `controllers/` contient les fonctions de logique métier pour chaque entité (par ex. `controllers/rendezvousController.js` exporte les méthodes `create`, `getAll`, `update`, `delete`). Les appels à la base de données sont effectués via ces contrôleurs utilisant les méthodes Sequelize définies dans `models/`.

La structure du code back-end comprend :
- **app.js** : configuration d'Express et des middlewares (parsing JSON, sécurité, CORS), initialisation de Sequelize.
- **routes/** : définition des routes REST pour chaque entité.
- **controllers/** : implémentation de la logique métier par entité.
- **models/** : définitions Sequelize des entités (Utilisateur, Service, RendezVous, Contenu).
- **config/** (ou fichier `.env`) : paramètres de connexion (hôte, base, utilisateur MySQL), clés secrètes (JWT), etc.

Les opérations CRUD sont codées en suivant le modèle : vérification des données, appel du modèle Sequelize, puis renvoi de la réponse. Par exemple, le contrôleur de création de rendez-vous (`createRdv`) peut ressembler à :

```
async function createRdv(req, res) {
  const { nomClient, idService, date_heure } = req.body;
  // Vérifier qu'aucun rendez-vous n'existe déjà à cette date/heure
  const existant = await RendezVous.findOne({ where: { date_heure } });
  if (existant) {
    return res.status(400).json({ message: 'Créneau déjà réservé' });
  }
  // Enregistrer le nouveau rendez-vous
  const rdv = await RendezVous.create({ nomClient, idService, date_heure,
statut: 'EN_ATTENTE' });
```

```
  res.status(201).json(rdv);
}
```

La route Express correspondante (`routes/rendezvous.js`) serait :

```
const express = require('express');
const router = express.Router();
const rdvController = require('../controllers/rendezvousController');

router.get('/', rdvController.getAll);
router.post('/', rdvController.create);
router.put('/:id', rdvController.update);
router.delete('/:id', rdvController.delete);

module.exports = router;
```

L'API gère également la création et la gestion des utilisateurs et des services avec la même approche CRUD. Les validations de format (avec `express-validator`) et la gestion des erreurs (statuts HTTP appropriés) rendent l'API robuste et conforme aux standards RESTful.

Développement Front-End

L'interface utilisateur est développée avec **Vue.js** (version 3). Le code est organisé en composants et vues : par exemple `PageReservation.vue` et `FormulaireRdv.vue` pour la réservation client, ainsi que `DashboardAdmin.vue`, `ListeServices.vue`, `GestionContenu.vue` pour l'interface administrateur. Vue Router gère la navigation entre les vues (accueil, réservation, espace admin, etc.), et Axios est utilisé pour effectuer les appels HTTP vers l'API back-end.

Le front-end est responsive (utilisation de CSS flexible ou d'un framework CSS) et propose des validations de base sur les formulaires (champs obligatoires, formats). Par exemple, le composant de formulaire de réservation (`FormulaireRdv.vue`) contient des champs liés en `v-model`. Lors de la soumission, une méthode `reserve()` envoie les données à l'API. Un exemple simplifié :

```
<template>
  <div>
    <h2>Prendre Rendez-vous</h2>
    <form @submit.prevent="reserve">
      <input v-model="nomClient" type="text" placeholder="Votre nom"
required />

<!-- Autres champs : email, téléphone, sélection du service, date/heure -->
      <button type="submit">Réserver</button>
    </form>
  </div>
</template>
```

```
<script>
import axios from 'axios';
export default {
  data() {
    return {
      nomClient: '',
      // autres champs...
    };
  },
  methods: {
    async reserve() {
      try {
        await axios.post('/api/rendezvous', {
          nomClient: this.nomClient,
          // autres données...
        });
        alert('Rendez-vous réservé avec succès !');
      } catch (error) {
        console.error(error);
        alert('Erreur lors de la réservation.');
      }
    }
  }
};
</script>
```

Côté administrateur, des composants similaires permettent de lister les rendez-vous (tableau ou calendrier), de valider/annuler, et de gérer les services et contenus via des formulaires. Les données affichées sont automatiquement actualisées grâce à la réactivité de Vue.js, offrant une expérience utilisateur fluide.

Sécurité de l'application

La sécurité a été intégrée dès la conception. Les mots de passe utilisateur sont hachés avec **bcrypt** avant stockage (par exemple `bcrypt.hash(password, saltRounds)`). L'accès aux zones protégées (interface admin et API sensibles) est contrôlé par des **JSON Web Tokens (JWT)**. Lorsqu'un administrateur se connecte via l'endpoint `/api/auth/login`, le serveur vérifie son identifiant et son mot de passe, puis génère un JWT signé contenant l'ID utilisateur et son rôle.

Un middleware d'authentification vérifie la présence et la validité du JWT sur chaque requête protégée. Exemple :

```
const jwt = require('jsonwebtoken');

function authMiddleware(req, res, next) {
  const token = req.header('Authorization')?.replace('Bearer ', '');
  if (!token) return res.status(401).json({ message: 'Token manquant' });
```

```
  try {
    const payload = jwt.verify(token, process.env.JWT_SECRET);
    req.user = { id: payload.id, role: payload.role };
    next();
  } catch (err) {
    return res.status(401).json({ message: 'Token invalide' });
  }
}
```

Un middleware d'autorisation supplémentaire vérifie `req.user.role` pour restreindre certaines routes aux administrateurs (renvoyer 403 Forbidden sinon). Ce schéma stateless (sans session serveur) assure la sécurité et la scalabilité.

D'autres protections ont été mises en place : le middleware helmet renforce les en-têtes HTTP, et les requêtes SQL sont paramétrées via Sequelize pour éviter les injections. Les entrées utilisateur sont validées (avec `express-validator` côté serveur et contrôles côté client) pour garantir l'intégrité des données. Les configurations CORS limitent les requêtes au domaine du front-end.

Tests et assurance qualité

La qualité de l'application a été vérifiée par des tests automatisés et manuels. Des **tests unitaires** et d'intégration ont été écrits avec **Jest** et **supertest** pour vérifier les endpoints critiques du back-end. Par exemple, on s'assure qu'un nouveau rendez-vous peut être créé via l'API :

```
const request = require('supertest');
const app = require('../app');

describe('API RendezVous', () => {
  test('devrait créer un nouveau rendez-vous', async () => {
    const res = await request(app)
      .post('/api/rendezvous')
      .send({
        nomClient: 'Dupont',
        telephoneClient: '0612345678',
        idService: 3,
        date_heure: '2025-06-15T10:30:00'
      });
    expect(res.statusCode).toBe(201);
    expect(res.body).toHaveProperty('id_rdv');
  });
});
```

Des tests similaires existent pour les endpoints utilisateurs, services, etc. La commande `npm test` exécute l'ensemble des tests pour s'assurer que le code reste correct à chaque modification.

En parallèle, des tests manuels ont été réalisés via **Postman** pour chaque endpoint (GET, POST, PUT, DELETE). Par exemple, voici une requête POST pour créer un rendez-vous :

```
POST /api/rendezvous
Content-Type: application/json

{
  "nomClient": "Dupont",
  "telephoneClient": "0612345678",
  "idService": 3,
  "date_heure": "2025-06-15T10:30:00"
}
```

Ces tests ont permis de détecter et corriger plusieurs bugs (gestion des erreurs, validations), garantissant une application fiable.

Intégration Continue (CI/CD) et déploiement

Un pipeline **CI/CD** a été mis en place avec GitHub Actions pour automatiser la validation du code. À chaque `push` sur la branche principale, le workflow exécute les étapes suivantes :
1. Cloner le dépôt et installer les dépendances (`npm install`).
2. Lancer les tests (`npm test`).
3. (Optionnel) Déployer automatiquement sur un serveur de staging ou de production.

Exemple de configuration `.github/workflows/ci.yml` :

```
name: CI

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test
```

Ce pipeline garantit qu'aucun code défectueux n'est fusionné. En cas de réussite, un déploiement automatisé (par exemple sur un service cloud) peut être déclenché pour mettre à jour l'application en production.

Conclusion

Le projet MABOTE a atteint ses objectifs : l'institut de beauté dispose désormais d'une application web fonctionnelle pour la gestion des rendez-vous et du contenu. Les clients peuvent réserver en ligne 24/7, et l'administrateur peut gérer le planning et le contenu du site de manière autonome. Les choix technologiques (Vue.js, Node.js/Express, Sequelize) ont permis de respecter les contraintes (coût nul, temps limité) tout en assurant une solution évolutive et maintenable.

Ce rapport a présenté l'ensemble du cycle de développement : de l'analyse du besoin à la conception, en passant par l'implémentation, la sécurité et les tests. Les compétences du référentiel CDA (analyse, conception, développement, tests, déploiement) ont été mobilisées avec succès.

Pour l'avenir, plusieurs évolutions sont envisageables : notifications par email/SMS pour les rappels de rendez-vous, prise en charge multi-employés, et déploiement sur un environnement cloud sécurisé.

Je remercie mon tuteur pédagogique, l'équipe de l'institut MABOTÉ et l'ensemble de mon établissement pour leur soutien et leurs conseils durant ce projet.

Annexes

- Maquettes et captures d'écran (wireframes et interface réalisée).
- Extraits de code complémentaires (modèles Sequelize `RendezVous`, contrôleurs de validation, etc.).
- Documentation technique (instructions d'installation, fichier `.env.example`, collection Postman ou documentation Swagger de l'API).
- Relevé des heures et bilan des compétences acquises.
