DOSSIER PROJET

Création d'une application web de gestion pour l'entreprise ECC Rénovation

Estefania Cachada Capitão

Titre Professionnel visé : Développeur Web et Web Mobile (DWWM)

ECC RENOVATION

Lien du Git : https://github.com/Fani1987/ecc-renovation.git

SOMMAIRE

I. Introduction

1.1. Présentation du projet

Le projet "ECC Rénovation" consiste en la création d'une application web complète pour une entreprise (réelle, celle de mon papa) du secteur du bâtiment. L'objectif était de développer une solution "deux-en-un" :

- Une **vitrine publique (front-office)** moderne pour attirer de nouveaux prospects, présenter les réalisations de l'entreprise (portfolio) et permettre une prise de contact facile.
- Une **plateforme de gestion (back-office)** sécurisée permettant à l'administrateur de gérer le contenu du site, les nouveaux clients, et le cycle de vie complet des devis.
- Un **espace client** permettant aux utilisateurs enregistrés de suivre leurs devis, de les valider et de télécharger les documents officiels.

Ce projet a été construit "from scratch" en utilisant une architecture PHP pure, une architecture de **persistance polyglotte** utilisant **MySQL** pour les données relationnelles (clients, devis) et **MongoDB (NoSQL)** pour les données de type document (messages), ainsi que des technologies front-end modernes (Sass, JavaScript).

1.2. Compétences du référentiel couvertes

Conformément aux exigences du Titre Professionnel, ce projet couvre les compétences des deux activités types.

Activité type 1 : Développer la partie front-end d'une application web ou web mobile sécurisée

- **Maquetter des interfaces utilisateur web ou web mobile :** Les maquettes de l'interface publique et des tableaux de bord ont été réalisées.

- **Réaliser des interfaces utilisateur statiques web ou web mobile :** Les maquettes ont été intégrées en HTML5 sémantique et SCSS (Sass).
- **Développer la partie dynamique des interfaces utilisateur web ou web mobile :** Le JavaScript a été utilisé pour le menu responsive, l'affichage dynamique du portfolio (Fetch API) et la soumission asynchrone de formulaires.

Activité type 2 : Développer la partie back-end d'une application web ou web mobile sécurisée

- **Mettre en place une base de données relationnelle :** Une base de données MySQL a été conçue pour gérer les relations entre les clients, les devis, les projets et les administrateurs.
- **Développer des composants d'accès aux données SQL et NoSQL :** L'accès aux données est sécurisé via des requêtes préparées en SQL (MySQLi) pour les données relationnelles, et via la bibliothèque MongoDB PHP pour les données de type document.
- **Développer des composants métier côté serveur :** La logique de connexion, la gestion des sessions, le traitement des formulaires, le hachage des mots de passe et la gestion des rôles (admin/client) ont été implémentés en PHP.

1.3 Contexte et Expression des Besoins

Ce projet simule une commande d'un artisan (M. Capitão, gérant d'ECC Rénovation) souhaitant moderniser sa présence en ligne et optimiser sa gestion commerciale.

1.4 Objectifs du projet

- **Visibilité :** Remplacer un ancien site statique par une plateforme moderne et responsive.

- **Efficacité :** Réduire le temps de gestion des demandes de contact en les centralisant.
- **Professionnalisme :** Offrir aux clients un espace personnel pour suivre leur projet, de la demande de devis au paiement.
- **Autonomie :** Permettre à l'administrateur de mettre à jour son portfolio de réalisations sans faire appel à un développeur.

1.5 Description des utilisateurs (Personas)

A. **Le Visiteur (Prospect)**

- *Profil :* Particulier ou professionnel cherchant à faire des travaux.
- *Besoins :* Se rassurer sur le sérieux de l'entreprise, voir des exemples de travaux, pouvoir contacter l'entreprise facilement.

B. **L'Administrateur (M. Capitao)**

- *Profil :* Gérant de l'entreprise, à l'aise avec les outils numériques de base mais n'est pas technicien.
- *Besoins :* Gérer son site simplement. Consulter les messages, créer des comptes clients, ajouter de nouvelles réalisations, et surtout, créer des devis (PDF) et les assigner à ses clients.

C. **Le Client (M. Dupont)**

- *Profil :* Client existant pour qui un projet est en cours de chiffrage.
- *Besoins :* Accéder à son devis officiel, pouvoir le valider ou le refuser, et garder une trace des échanges.

1.6 Fonctionnalités clés

- **(Visiteur)** Consultation du portfolio dynamique (chargé depuis la BDD).
- **(Visiteur)** Envoi d'un message via le formulaire de contact (NoSQL).

- **(Admin)** Connexion sécurisée à un tableau de bord distinct.
- **(Admin)** Gestion CRUD des Réalisations (portfolio).
- **(Admin)** Consultation des messages (demandes de contact)(NoSQL).
- **(Admin)** Gestion CRUD des Clients (créer un compte client, rechercher).
- **(Admin)** Gestion CRUD des Devis (créer un devis, téléverser le PDF, l'assigner à un client).
- **(Client)** Connexion sécurisée à un espace personnel distinct.
- **(Client)** Consultation de la liste de ses devis et de leur statut.
- **(Client)** Consultation d'un devis détaillé (avec téléchargement du PDF).
- **(Client)** Actions : Accepter, Refuser, ou "Payer" (simulé) un devis.
- **(Client)** Demande de devis supplémentaire depuis son espace.

1.7 Périmètre fonctionnel et exclusions

En accord avec le client (l'entreprise ECC Rénovation), il a été décidé que l'application se concentrerait sur le cycle de vie complet du devis, de sa création jusqu'à sa validation et la simulation de son paiement.

La gestion de la facturation (création, envoi et suivi des factures) a été volontairement exclue du périmètre de ce projet. Conformément à la volonté du chef d'entreprise, ce processus reste géré en externe via son logiciel de comptabilité habituel. Le statut "Payé" dans l'application sert donc de déclencheur pour que l'administrateur initie son processus de facturation manuel.

1.8 Environnement Technique et guide d’utilisation

Pour ce projet, une infrastructure de développement moderne a été mise en place pour simuler un environnement de production fiable et reproductible.

- **Langages** : HTML5, CSS3, SCSS (Sass), JavaScript (ES6), PHP 8.1, SQL.

- **Bases de Données** :
    - MySQL 8.0 (Relationnel) : Pour les données transactionnelles (clients, devis, projets).
    - MongoDB (NoSQL) : Pour les données non structurées (messages de contact).

- **Conteneurisation :** Docker & Docker Compose. L'environnement de développement ne repose pas sur une installation WAMP/XAMPP traditionnelle, mais sur un environnement conteneurisé défini par un fichier `docker-compose.yml`. Cela garantit la portabilité et la cohérence entre les environnements. L'infrastructure est composée de trois services :
    1. Un conteneur **app** (image php:8.1-apache) pour le serveur web et l'interpréteur PHP.
    2. Un conteneur **db-sql** (image mysql:8.0) pour la base de données relationnelle.
    3. Un conteneur **db-nosql** (image mongo:latest) pour la base de données NoSQL.

- **Outils de développement** : Visual Studio Code, Git.
- **Conception (Maquettage)** : Figma.
- **Gestion des dépendances PHP** : Composer (pour l'installation de la bibliothèque mongodb/mongodb).

Guide de Mise en Place de l'Environnement de Développement

La conteneurisation avec Docker garantit qu'un nouvel environnement de développement peut être lancé de manière fiable et identique à la production en suivant ces étapes :

Étape 1 : Prérequis

- Le développeur doit avoir **Git** et **Docker Desktop** installés sur sa machine.

Étape 2 : Récupération et Configuration

1. Cloner le dépôt Git : git clone https://[votre_url_git]/ecc-renovation.git
2. Créer le fichier d'environnement local à partir de l'exemple fourni : cp .env.example .env
3. Modifier le fichier .env pour y ajouter un mot de passe DB_PASSWORD (par exemple : admin123).

Étape 3 : Lancement des Conteneurs

1. Depuis la racine du projet, lancer la commande Docker Compose : docker-compose up -d --build
2. **Ce que fait cette commande :**
    - --build : Construit l'image PHP/Apache personnalisée (docker/php/Dockerfile) en y installant les extensions mysqli et mongodb.
    - Télécharge les images officielles de MySQL 8.0 et MongoDB.
    - Démarre les 3 conteneurs (app, db-sql, db-nosql) et les connecte sur le même réseau privé.

Étape 4 : Installation des Dépendances PHP

1. Une fois les conteneurs lancés, on se connecte au conteneur app pour installer les dépendances PHP (la bibliothèque MongoDB) : docker-compose exec app composer install

Étape 5 : Migration de la Base de Données

1. L'application est en ligne sur http://localhost, mais la base de données est vide. Il faut importer le fichier .sql (préalablement exporté) :
2. **Copie :** docker cp ecc_renovation.sql ecc_db_sql:/tmp/ecc_renovation.sql
3. **Import :** docker exec -i ecc_db_sql bash -c "mysql -u root -p[MDP_DU_ENV] ecc_renovation < /tmp/ecc_renovation.sql"
4. La base NoSQL (MongoDB) se crée automatiquement lors de la première soumission de formulaire.

Étape 6 : Finalisation

- L'application est maintenant pleinement fonctionnelle en local sur http://localhost.

Stratégie de Déploiement en Production

L'utilisation de Docker et Docker Compose ne sert pas uniquement à l'environnement de développement ; elle est la clé du déploiement en production.

Pour déployer cette application sur un serveur public, la méthode serait la suivante :

1. **Provisionner un VPS (Serveur Privé Virtuel)** (ex: chez OVH, DigitalOcean) fonctionnant sous Linux (Ubuntu).
2. **Installer les outils** essentiels sur ce serveur : git, docker, et docker-compose.
3. **Cloner le dépôt Git** du projet sur le serveur.
4. **Créer le fichier .env** de production (avec les vrais mots de passe de la base de données) à partir du modèle `.env.example`.
5. **Lancer l'application** avec la même commande qu'en local :

```
docker-compose up -d --build.
```

6. **Importer les données** de la base de données SQL et configurer un nom de domaine pour qu'il pointe vers l'adresse IP du serveur.

Cette approche garantit que l'environnement de production est **strictement identique** à l'environnement de développement, éliminant ainsi les conflits de version et les problèmes de « ça marche sur ma machine ».

Méthodologie de Gestion de Version (Git)

Le simple fait d'utiliser Git n'est pas suffisant ; il est crucial d'adopter un flux de travail (workflow) professionnel pour garantir la stabilité du projet et la lisibilité de l'historique.

Conformément aux bonnes pratiques de l'industrie, le projet n'a pas été développé sur une seule branche (main). L'utilisation d'une branche unique est risquée : elle ne fournit aucun filet de sécurité en cas d'erreur et pollue l'historique de la version stable.

Pour ce projet, un flux de travail basé sur les branches de fonctionnalités (Feature Branch Workflow) a été appliqué :

1. **La branche `main` est protégée**. Elle contient la version 1.0 stable et dockerisée du projet.
2. **Développement en branches** : Tout développement ultérieur est effectué dans une branche de fonctionnalité dédiée.
3. **Démonstration Pratique** : Pour illustrer ce flux de travail, la page "Politique de confidentialité" a été ajoutée. Ce travail n'a pas été effectué directement sur `main`, mais sur une branche dédiée nommée `feature/politique-confidentialite`.
4. **Fusion (Merge)** : Une fois la fonctionnalité terminée et testée, cette branche a été proprement fusionnée (`merge`) dans la branche `main`.

L'historique Git de ce projet reflète cette méthodologie, démontrant une gestion de version propre et professionnelle où la branche `main` est mise à jour uniquement par des fusions de branches de fonctionnalités terminées.

NB : Mon historique Git reflète fidèlement le cycle de vie réel du projet.

1. ***Au début**, sur la branche main, on peut voir l'historique de la phase de **construction et de prototypage**, avec des commits fréquents pour sauvegarder les avancées (l'ajout de MongoDB, la correction du CSS, etc.).*
2. *Une fois l'application stabilisée et "dockerisée" (ce que j'ai considéré comme la **Version 1.0**), j'ai adopté un **workflow de production** plus strict pour toute nouvelle modification.*
3. *J'ai d'ailleurs appliqué ce workflow pour la dernière fonctionnalité : l'ajout de la page de politique de confidentialité. Comme vous pouvez le voir, ce travail n'a pas été fait sur main, mais sur une branche dédiée **feature/politique-confidentialite**, qui a ensuite été testée et **fusionnée (merge)** proprement dans main. (ainsi que pour une correction de bug : **fix/bug-main-js**).*

Je me suis adapté aux bonnes pratiques et je suis passé d'un mode de développement initial rapide à un mode de maintenance propre et professionnel qui protège la branche de production.

Exemple du "push" d'une branche :

- **Objectif :** Ajouter une nouvelle page de politique de confidentialité.
- **Branche Principale :** main (elle est stable et on n'y touche pas).
- **Dépôt Distant (GitHub) :** origin

Étape 1 : Créer une nouvelle branche de fonctionnalité (en local)

(Les commandes sont envoyées sur le terminal ouvert a la racine de mon projet)

Je commence par me baser sur la version stable (`main`) et je crée ma "copie de travail" isolée, que j'appelle `feature/politique-confidentialite`.

```
Bash
# Je vérifie que je suis bien sur la branche principale
git checkout main

# Je crée ma nouvelle branche et je bascule dessus
git checkout -b feature/politique-confidentialite
```

(Je suis maintenant sur la branche feature/politique-confidentialite. main est en sécurité.)

Étape 2 : Faire le travail (en local)

Je fais toutes les modifications nécessaires pour cette fonctionnalité :

1. Je crée le fichier `politique-confidentialite.php`.
2. Je modifie `partials/_footer.php` pour ajouter le lien.

Étape 3 : Sauvegarder le travail (en local)

Une fois le travail terminé et testé, je l'enregistre dans l'historique de **ma** branche locale avec un "commit".

```
Bash
# J'ajoute tous mes fichiers modifiés
git add .

# Je crée un "instantané" (commit) avec un message clair
git commit -m "feat: Ajout page Politique de confidentialité"
```

*(À cet instant, le commit n'existe **que sur mon ordinateur**. GitHub ne le voit pas encore.)*

Étape 4 : Le "Push" de la branche (L'action que tu demandes)

C'est l'étape où je publie ma branche de fonctionnalité sur le dépôt distant (GitHub) pour la première fois.

```
Bash

# C'est la commande clé :

git push origin feature/politique-confidentialite
```

Ce que fait cette commande :

- `git push` : C'est l'ordre "d'envoyer".
- `origin` : C'est la destination (le nom de mon dépôt distant sur GitHub).
- `feature/politique-confidentialite` : C'est le nom de la branche que je veux envoyer.

II. Réalisations Front-End

2.1. Maquettes des interfaces utilisateur

Le maquettage a permis de définir l'ergonomie du site avant l'écriture du code. Une approche "mobile-first" a été privilégiée pour le design.

- Maquette Figma de l'accueil (Desktop)

Cf : Annexe 1

- Maquette Figma de l'accueil (Mobile)

Cf : Annexe 2

- Maquette Figma du Tableau de Bord Admin (Desktop)

Cf : Annexe 3

- Maquette Figma du tableau de bord Admin (Mobile)

Cf : Annexe 4

- Maquette Figma de l’Espace Client (Desktop)

Cf : Annexe 5

- Maquette Figma de l’Espace Client (Mobile)

Cf : Annexe 6

- Maquette Figma de la page Espace Client/ Détail d’un devis (Desktop)

Cf : Annexe 7

- Maquette Figma de la page Espace Client/ Détail d’un devis (Mobile)

Cf : Annexe 8

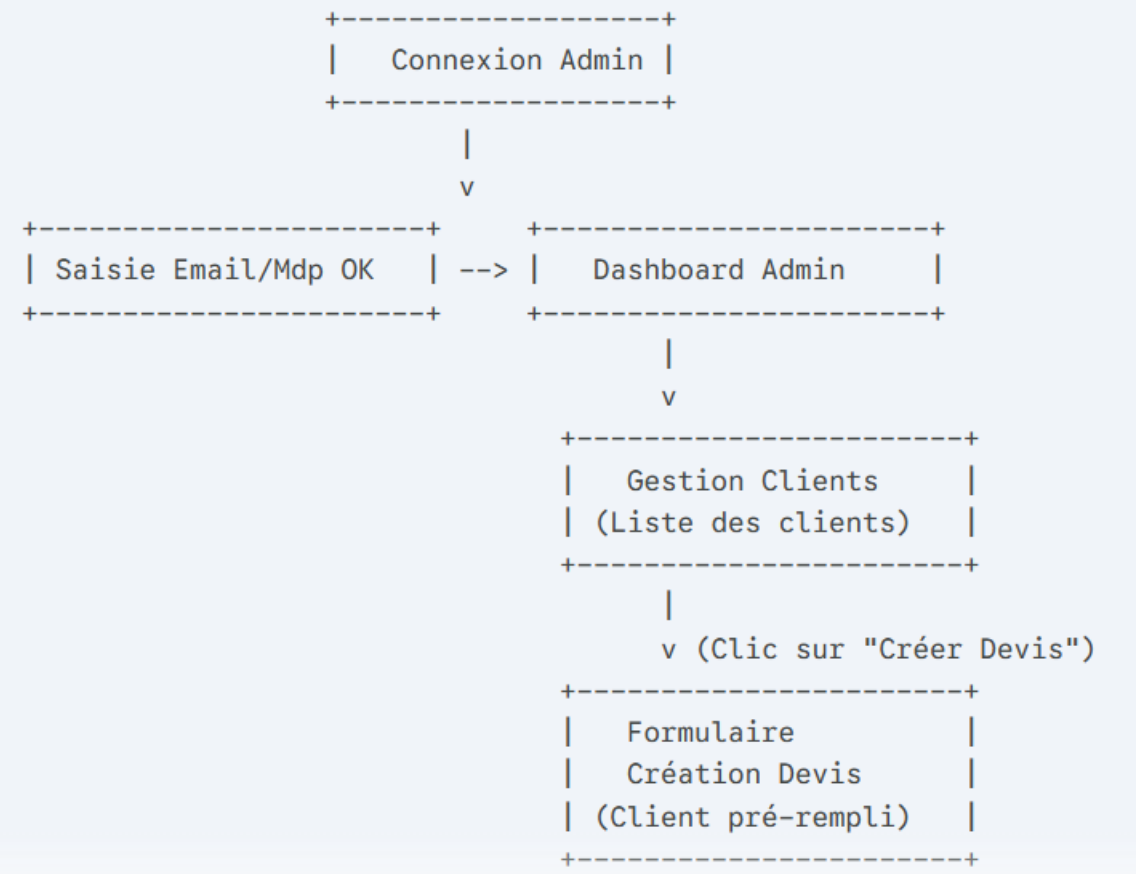
4.2. Schémas de l'enchaînement des écrans

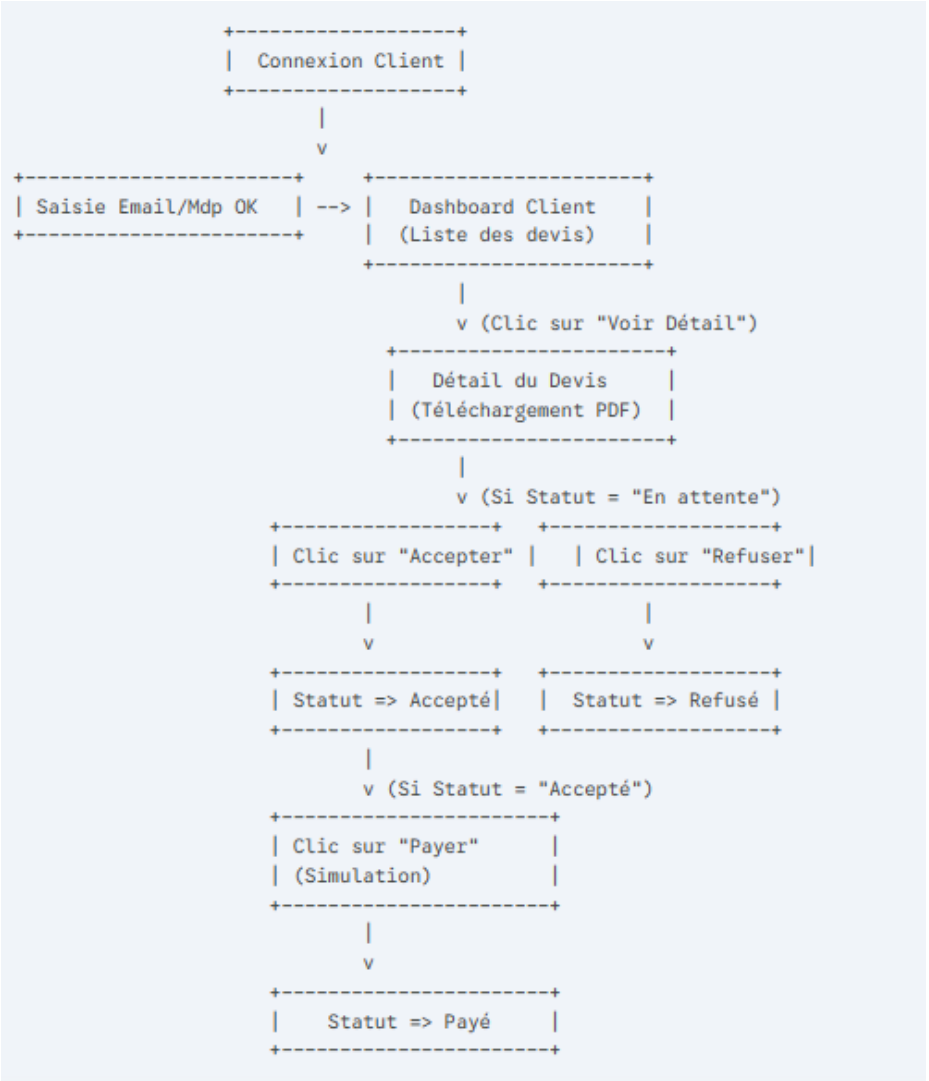
Les diagrammes suivants illustrent les parcours de navigation (ou "user journeys") pour les deux principaux types d'utilisateurs de l'application : l'administrateur et le client.

Cette séparation est fondamentale pour l'architecture du projet, car chaque rôle a des permissions et des objectifs distincts :

- **Le Parcours Administrateur** montre le flux de travail interne : de la connexion au tableau de bord, jusqu'à l'exécution d'une tâche de gestion clé, comme la création d'un devis à partir de la fiche d'un client.
- **Le Parcours Client** illustre le cycle de vie d'un devis du point de vue du client : de sa connexion à son tableau de bord personnel, la consultation d'un devis spécifique, et les actions conditionnelles qui en découlent (Accepter, Refuser ou Payer).

Ces schémas ont servi de guide lors du développement pour assurer une expérience utilisateur logique et intuitive pour chaque rôle.




Connexion Client
Saisie Email/Mdp OK
Dashboard Client
(Liste des devis)
Détail du Devis
(Téléchargement PDF)
(Clic sur "Voir Détail")
(Si Statut = "En attente")
Clic sur "Accepter"
Clic sur "Refuser"
Statut => Accepté
Statut => Refusé
(Si Statut = "Accepté")
Clic sur "Payer"
(Simulation)
Statut => Payé

4.3. Interfaces utilisateur statiques (HTML/SCSS)

L'intégration des maquettes a été une étape fondamentale pour traduire la vision du design en une application web fonctionnelle. Pour garantir la qualité et la maintenabilité du projet, des choix technologiques et méthodologiques précis ont été faits.

L'intégration a été réalisée en **HTML5 sémantique** et **SCSS** pour un code modulaire et maintenable. L'architecture **BEM** (Block, Element, Modifier) a été utilisée pour structurer le CSS. Pour la mise en page, des techniques modernes comme **CSS Grid** et **Flexbox** ont été employées.

Voici le détail de ces choix :

1. HTML5 Sémantique : Donner du Sens à la Structure

Plutôt que d'utiliser des balises génériques pour l'ensemble de la structure, le HTML5 sémantique a été privilégié.

- **Pourquoi ?** Utiliser des balises comme <header>, <nav>, <main>, <section>, ou <footer> ne sert pas qu'à la structure. Cela donne un **contexte** au contenu, ce qui est essentiel pour **l'accessibilité** (les lecteurs d'écran pour les malvoyants) et le **référencement (SEO)** (les moteurs de recherche comprennent mieux la hiérarchie de la page).
- **Exemple :** La section des services est définie par <section id="services">, et la navigation principale par une balise <nav>.

2. SCSS (Sass) : Rendre le CSS Modulaire et Maintenable

Le CSS natif peut devenir chaotique sur un projet de cette taille. L'utilisation du préprocesseur SCSS (Sass) a résolu ce problème de trois façons :

1. **Les Variables :** Toutes les valeurs de la charte graphique (couleurs primary-blue, polices, ombres $shadow) sont définies dans un seul fichier (_variables.scss). Si le client veut changer le bleu principal du site, je n'ai qu'une seule ligne à modifier pour que le changement s'applique partout.

2. **L'Imbrication (Nesting) :** Le code SCSS est écrit en "emboîtant" les sélecteurs, ce qui reflète la structure du HTML. C'est plus propre et évite les longs sélecteurs CSS.
3. **Les Partials (Modularité) :** Le code est divisé en "partials" (fichiers commençant par _). Chaque composant (_header.scss, _cards.scss, _tables.scss) et chaque page (_home.scss, _login.scss) a son propre fichier. Le fichier main.scss se contente de les importer. Cette architecture rend le débogage et l'ajout de fonctionnalités infiniment plus rapides.

3. BEM (Block, Element, Modifier) : Une Architecture CSS Robuste

Pour éviter les conflits de style et les problèmes de spécificité (les !important), une convention de nommage stricte, BEM, a été utilisée.

- **Block :** Un composant autonome. (Ex: .service-card ou .card-action).
- **Element :** Une partie d'un bloc. (Ex: .card__title ou .card__icon).
- **Modifier :** Une variation d'un bloc. (Ex: .card--action pour une carte avec un fond différent).
- **Résultat :** Le code est prévisible et chaque composant est indépendant. Je peux déplacer un .service-card n'importe où sur le site et être certain qu'il conservera son style.

4. CSS Grid et Flexbox : La Mise en Page Moderne

Le projet n'utilise aucune technique de "hack" ancienne (comme les float ou les tables) pour la mise en page.

- **Flexbox (Mise en page 1D) :** A été utilisé pour tous les besoins d'alignement sur un seul axe. Par exemple, pour centrer les éléments dans la barre de navigation (<nav>) ou pour aligner le texte et l'image côte à côte dans la section "Qui sommes-nous ?".

Exemple : La section "Qui sommes-nous ?" (#about) avec Flexbox

Ce code montre une structure sémantique simple (`<section>`) contenant un conteneur Flexbox (`.about-content`) qui englobe les twox colonnes (`.about-text` et `.about-image`).

Extrait de code HTML extrait de `index.php` :

```
<section id="about">
            <div class="container">
                <h2>Qui sommes-nous ?</h2>
                <div class="about-content">
                    <div class="about-text">
                        <h3>ECC Rénovation : [..]
                        <p>Fondée par Jorge CAPITAO, ECC
Rénovation […].</p>
                        <p>Notre engagement :[…].</p>
                        <a href="#portfolio"
class="btn">Voir nos projets</a>
                    </div>
                    <div class="about-image">
                        <img
src="https://images.pexels.com/photos/5691535/pexels-
photo-
```

```
5691535.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1"
alt="Équipe ECC Rénovation au travail">
                    </div>
                </div>
            </div>
        </section>
```

Extrait de code CSS (`pages/_home.scss` et `main.scss`) correspondant :

1. **Styles Desktop (dans pages/_home.scss)**

```
.about-content {
        display: flex;
   align-items: center; // Centre les colonnes
verticalement
   gap: 40px; // Ajoute un espace entre texte et image
     }

.about-text, .about-image {
   flex: 1; // Dit au texte de prendre 50 % de l'espace
     }
```

2. **Ajustement Responsive (dans `main.scss`)**

```
@media (max-width: 768px) {

.about-content { flex-direction: column; }// Change l'axe
de Flexbox : les colonnes deviennet des lignes
```

Explication du choix technique (Flexbox) :

Pour la section 'Qui sommes-nous ?', j'ai utilisé **Flexbox** car c'est l'outil idéal pour gérer l'alignement sur **un seul axe** (horizontal sur desktop).

La propriété `display: flex` transforme `.about-content` en conteneur flexible, et `flex: 1` sur les deux enfants (`.about-text` et `.about-image`) leur dit d'occuper chacun la moitié de l'espace.

L'efficacité de cette méthode est prouvée sur mobile : une seule ligne, flex-direction: column, suffit à réorganiser les éléments en une pile verticale, créant un affichage parfaitement lisible sur téléphone.

- **CSS Grid (Mise en page 2D) :** A été utilisé pour la mise en page complexe en deux dimensions. L'exemple le plus parlant est la grille des services (.services-grid) qui, grâce à grid-template-columns: repeat(auto-fit, minmax(280px, 1fr)) , crée une grille parfaitement responsive qui adapte le nombre de colonnes à l'espace disponible **sans nécessiter de media queries** .

Exemple : La grille des services (#services) avec utilisation de CSS Grid :

Code HTML correspondant (Extrait de `index.php`) :

```
<section id="services">
            <div class="container">
                <h2>Nos Services</h2>
                <div class="services-grid">

                    <div class="service-card">
                        <i class="fas fa-paint-roller"></i>
                        <h3>Rénovation Intérieure</h3>
                        <p>Peinture, plâtrerie, […]</p>
                    </div>

                    <div class="service-card">
                        <i class="fas fa-hard-hat"></i>
                        <h3>Maçonnerie Générale</h3>
                        <p>Travaux de gros œuvre, murs […]</p>
                    </div>

                    <div class="service-card">
                        <i class="fas fa-bath"></i>
                        <h3>Rénovation de Salle de Bain</h3>
                        <p>Création et modernisation […]</p>
                    </div>

                    <div class="service-card">
                        <i class="fas fa-kitchen-set"></i>
                        <h3>Aménagement de Cuisine</h3>
                        <p>Installation et conception […]</p>
                    </div>

                    <div class="service-card">
                        <i class="fas fa-ruler-combined"></i>
                        <h3>Pose de revêtements</h3>
                        <p>Parquet, carrelage, lino... […]</p>
                    </div>

                    <div class="service-card">
                        <i class="fas fa-house-chimney-window"></i>
                        <h3>Rénovation Extérieure</h3>
                        <p>Ravalement de façade, isolation […]</p>
                    </div>

                </div>
            </div>
        </section>
```

Code SCSS (pages/_home.scss) correspondant :

```scss
#services {
    background-color: $light-blue-bg;
}
.services-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(280px,
1fr));
    gap: 30px;
}

.service-card {
    background-color: $white;
    padding: 30px;
    text-align: center;
    border-radius: 8px;
    box-shadow: $shadow;
    transition: transform 0.3s ease;
    &:hover {
        transform: translateY(-10px);
    }
// ... styles de l'icône, titre, etc. ...

}
```

Explication du choix technique (CSS Grid) :

Pour organiser la section des services, j'ai utilisé **CSS Grid Layout**. La propriété `grid-template-columns: repeat(auto-fit, minmax(280px, 1fr))` est particulièrement efficace : elle crée une grille entièrement responsive sans nécessiter de media queries complexes. Les cartes s'ajustent automatiquement pour remplir l'espace disponible, garantissant une présentation optimale sur toutes les tailles d'écran.

2.4. Interfaces utilisateur dynamiques (JavaScript)

Le JavaScript moderne (ES6+) a été utilisé pour rendre les interfaces interactives afin de créer une expérience utilisateur (UX) fluide, en évitant les rechargements de page complets à chaque action.

Plutôt que d'utiliser des recharges de page traditionnelles, l'application fonctionne sur un modèle plus proche d'une "Single Page Application" (SPA) pour ses sections interactives. Le back-end (PHP) est utilisé comme une **API** qui fournit des données brutes au format **JSON**, et c'est le JavaScript (côté client) qui se charge de construire et de mettre à jour le HTML.

Cette approche est illustrée par trois fonctionnalités clés du projet :

1. Chargement Asynchrone des Données (AJAX avec fetch)

La galerie "Nos Réalisations" n'est pas codée en dur en HTML. Au chargement de la page, le JavaScript prend le relais :

- **Appel Asynchrone :** Une requête fetch est envoyée au script PHP /api/get_projets.php. Le mot-clé **asynchrone** est essentiel : le navigateur ne se bloque pas en attendant la réponse ; il continue d'afficher le reste de la page.
- **Réception de JSON :** Le script PHP se connecte à la base de données (MySQL) et renvoie la liste des projets au format JSON.
- **Manipulation du DOM :** Le JavaScript reçoit ce JSON , boucle dessus (forEach) , et construit dynamiquement les blocs HTML du portfolio .
- **Résultat :** Le portfolio s'affiche sans que la page n'ait jamais rechargé.

2. Soumission de Formulaire sans Rechargement

Le formulaire de contact public utilise la même logique pour une expérience utilisateur améliorée :

- **Prévention de l'envoi classique :** Le script intercepte l'événement submit du formulaire et exécute event.preventDefault(). Cette

commande est cruciale, car elle empêche le navigateur d'effectuer l'action par défaut (recharger la page en envoyant les données).

- **Envoi Asynchrone :** Il utilise fetch en méthode POST pour envoyer les données du formulaire au script api/submit_contact.php.
- **Retour d'information (Feedback) :** Le script PHP (connecté à MongoDB) renvoie une réponse JSON (ex: { "success": true, "message": "Message envoyé !" }). Le JavaScript lit cette réponse et affiche le message de succès ou d'échec dans une div, le tout sans quitter la page d'accueil.

3. Utilisation de la Syntaxe Moderne ES6+

Pour rendre ce code lisible et maintenable, la syntaxe ES6+ a été privilégiée :

- **const (et let) :** Utilisés à la place de var pour une gestion de la portée (scope) des variables plus stricte, ce qui évite les bugs.
- **Template Literals (Backticks ``) :** La construction du HTML pour le portfolio utilise des "template literals" (les accents graves) . Cela permet d'injecter des variables (ex: ${projet.titre}) directement dans une chaîne de caractères sur plusieurs lignes, ce qui est infiniment plus propre que l'ancienne concaténation avec des +.
- **Manipulation Moderne du DOM :** Des méthodes comme querySelector (pour cibler les éléments) et classList.toggle() (utilisée pour le menu burger) sont utilisées pour interagir avec le HTML de manière efficace.

Exemple : Chargement dynamique du portfolio via fetch()

Le code suivant (**extrait de js/main.js**) contacte l'API PHP, récupère la liste des projets en JSON, et construit le HTML de la galerie côté client :

```
document.addEventListener('DOMContentLoaded', function() {

// Le script s'exécute uniquement lorsque le document HTML
// est entièrement chargé et analysé par le navigateur.
```

```
// C'est une sécurité pour garantir que 'portfolioGrid'
existe.
document.addEventListener('DOMContentLoaded', function() {

  // --- 1. GESTION DU PORTFOLIO ---

  // On sélectionne l'élément conteneur dans le HTML
  // où nous allons injecter nos projets.
  const portfolioGrid
document.querySelector('.portfolio-grid');

  // ↓ CECI EST UNE PRATIQUE DE PROGRAMMATION DÉFENSIVE ↓
  // On vérifie si cet élément existe sur la page
actuelle.
  // Cela empêche le script de planter (avec une erreur
"null")
  // sur les pages qui n'ont pas de galerie (ex: admin,
espace client).
  if (portfolioGrid) {

      // --- 2. L'APPEL ASYNCHRONE (AJAX) ---
      // On utilise l'API Fetch, la méthode moderne pour
faire des requêtes HTTP.
      // L'appel est ASYNCHRONE : le navigateur n'est pas
bloqué
      // pendant qu'il attend la réponse du serveur.
      fetch('api/get_projets.php')

          // --- 3. GESTION DE LA PROMESSE (Partie 1) ---
          // .then() se déclenche quand le serveur a
répondu.
          // 'response' contient la réponse HTTP (statut
200, 404, 500, etc.)
          .then(response => {

              // On vérifie si la réponse est un succès
(statut 200-299)
              if (!response.ok) {
                  // Si c'est une erreur (ex: 404, script
PHP non trouvé),
                  // on lève une erreur pour sauter au
bloc .catch()
```

```
                throw new Error('La requête a échoué');
            }
            // Si tout va bien, on décode le corps de la
réponse,
            // qui est au format JSON, en un objet
JavaScript utilisable.
            return response.json();
        })


        // --- 4. GESTION DE LA PROMESSE (Partie 2) ---
        // .then() se déclenche après que le JSON a été
décodé.
        // 'data' est maintenant notre tableau d'objets
(nos projets).
        .then(data => {

            // Cas où le script PHP fonctionne mais ne
renvoie rien
            if (data.length === 0) {
                portfolioGrid.innerHTML = "<p>Aucune
réalisation à afficher pour le moment.</p>";
                return; // On arrête le script ici.
            }

            // On vide la grille pour éviter d'ajouter
des doublons
            // si le script était appelé plusieurs fois.
            portfolioGrid.innerHTML = '';

            // --- 5. CONSTRUCTION DYNAMIQUE DU HTML ---
            // On boucle sur chaque objet 'projet' dans
notre tableau 'data'.
            data.forEach(projet => {

                // On utilise les "template literals"
(les backticks ``)
                // pour construire une chaîne HTML
propre en y injectant
                // les variables ${projet.titre}, etc.
                const portfolioItemHTML = `
                    <div class="portfolio-item">
```

```
                            <img
src="${projet.chemin_image}" alt="${projet.titre}">
                            <div class="portfolio-overlay">
                                <h3>${projet.titre}</h3>
                            </div>
                        </div>
                    `;

                    // On ajoute le nouveau bloc HTML à
l'intérieur de notre grille.
                    portfolioGrid.innerHTML +=
portfolioItemHTML;
                });

            })

            // --- 6. GESTION DES ERREURS ---
            // Ce bloc .catch() intercepte TOUTES les
erreurs :
            // - L'erreur 'La requête a échoué' qu'on a
levée (ex: 404)
            // - Une erreur réseau (pas d'internet)
            // - Une erreur de décodage (si le PHP renvoie
autre chose que du JSON)

            .catch(error => {
                // On log l'erreur technique pour le
développeur
                console.error('Erreur lors de la
récupération des projets:', error);
                // On affiche un message propre pour
l'utilisateur
                portfolioGrid.innerHTML = "<p>Impossible de
charger les réalisations. Veuillez réessayer plus
tard.</p>";
            });
    }
});
```

III. Réalisations Back-End

3.1. Conception des Bases de Données (Persistance Polyglotte)

Le référentiel du Titre Professionnel demande de "Développer des composants d'accès aux données **SQL et NoSQL**". Plutôt que de traiter cette exigence de manière superficielle, ce projet a été l'occasion de mettre en œuvre une architecture de **persistance polyglotte**, c'est-à-dire l'utilisation de différents types de bases de données pour différents types de données, au sein de la même application.

Ce choix d'architecture n'est pas anodin ; il répond à des besoins métiers distincts :

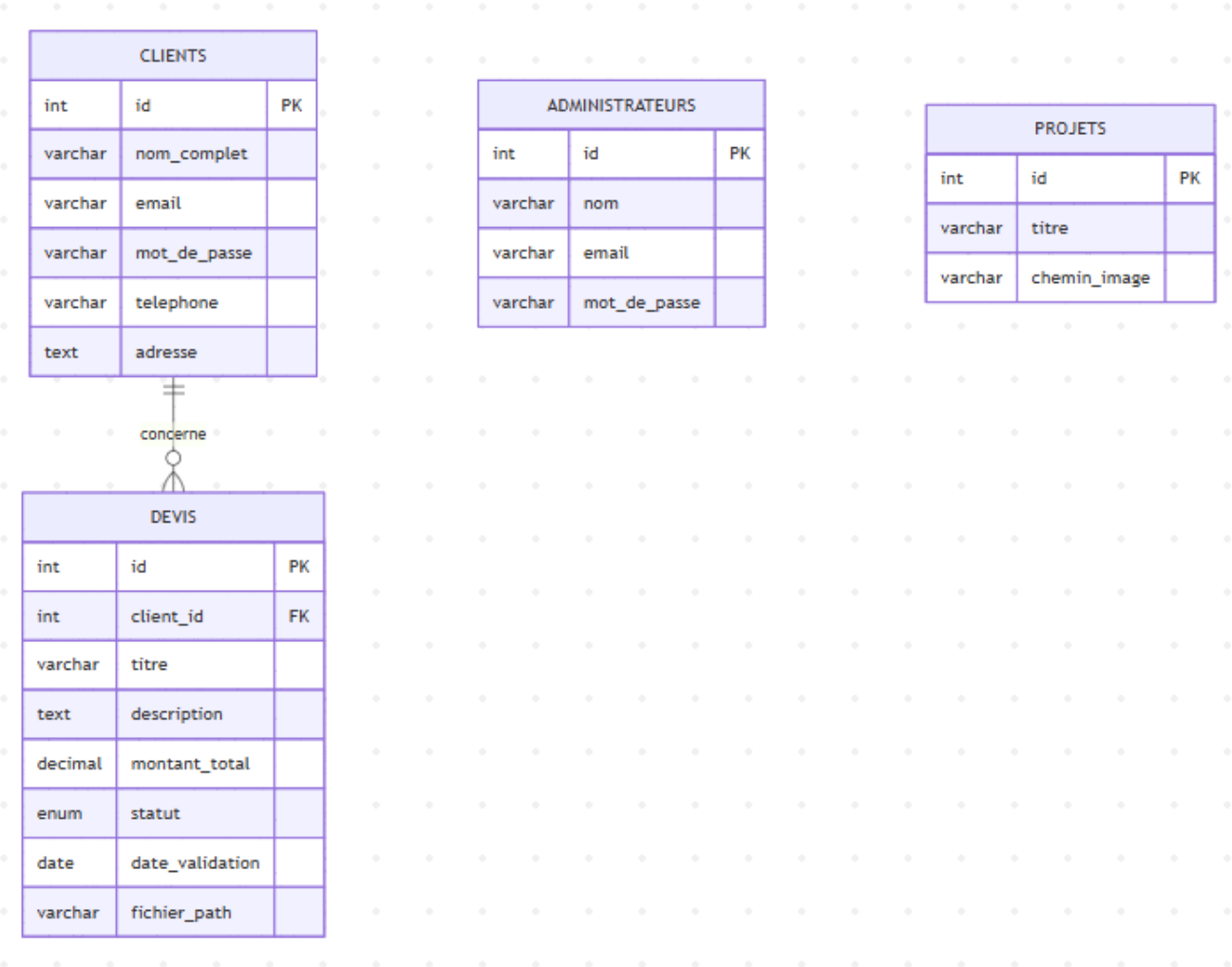
1. **Pourquoi MySQL (SQL) a été conservé :** Le cœur de l'application (la gestion des clients, des devis, des administrateurs et des projets) repose sur des données **hautement relationnelles et transactionnelles** . L'intégrité des données est cruciale : un devis *doit* appartenir à un client, et une facture *doit* être liée à un devis. Le modèle relationnel SQL, avec ses clés étrangères et ses schémas stricts, est la solution la plus robuste pour garantir cette cohérence.
2. **Pourquoi MongoDB (NoSQL) a été ajouté :** La fonctionnalité de "demandes de contact" (demandes_contact) représente un cas d'usage radicalement différent. Un message est un **document autonome**:
    - **Flexibilité (Schema-Less) :** Si demain, le formulaire de contact ajoute un champ "Numéro de téléphone" ou "Entreprise", la base NoSQL (MongoDB) acceptera ces nouveaux documents sans qu'il soit nécessaire de modifier la structure de la base (contrairement à SQL qui exigerait un ALTER TABLE).
    - **Nature des données :** Les messages sont des "logs" ; ils n'ont pas de relations complexes avec d'autres tables, ce qui les rend parfaits pour une base de données NoSQL de type document.

En résumé, j'ai utilisé **MySQL** pour sa **robustesse** transactionnelle et **MongoDB** pour sa **flexibilité** documentaire. La section suivante présente

les composants d'accès aux données pour ces twox systèmes, démontrant la capacité à interagir avec les deux types de bases de données en fonction du besoin.

3.1.1. Base de Données Relationnelle (MySQL)

Schéma de la base de données :



Script de création des tables clés :

Script pour la table clients :

```
CREATE TABLE clients (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nom_complet VARCHAR(255) NOT NULL,
    email VARCHAR(191) NOT NULL UNIQUE,
    mot_de_passe VARCHAR(255) NOT NULL,
    telephone VARCHAR(20),
    adresse TEXT,
    date_creation TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Script pour la table devis (montrant la clé étrangère) :

```
CREATE TABLE devis (
    id INT AUTO_INCREMENT PRIMARY KEY,
    client_id INT NOT NULL,
    titre VARCHAR(255) NOT NULL,
    montant_total DECIMAL(10, 2) NOT NULL,
    statut ENUM('En attente', 'Accepté', 'Refusé', 'Payé')
NOT NULL DEFAULT 'En attente',
    fichier_path VARCHAR(255) NULL,
    date_creation TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (client_id) REFERENCES clients(id) ON
DELETE CASCADE
) ;
```

3.1.2. Base de Données NoSQL (MongoDB)

Pour les demandes de contact, une base de données NoSQL de type document (MongoDB) a été choisie. Un message est un **document autonome** : il n'a pas de relations complexes avec d'autres tables. Cette approche offre plus de flexibilité (schema-less) si les champs du formulaire venaient à changer.

Structure d'un document de la collection `demandes_contact` :

```
{
  "_id": "ObjectId('...')"
  "nom": "Jean Dupont (Client existant)",
  "email": "jean.dupont@email.com",
  "message": "Je souhaiterais rénover ma cuisine...",
  "date_soumission": "ISODate('...')"
  "statut": "nouveau"
}
```

3.2. Composants d'accès aux données (SQL et NoSQL)

3.2.1 Démonstration des Opérations CRUD (Create, Read, Update, Delete) pour la partie SQL

L'acronyme **CRUD** (Create, Read, Update, Delete), ou **"Créer, Lire, Mettre à jour et Supprimer"**, désigne les quatre opérations fondamentales de la persistance des données. Elles constituent le socle de toute application web dynamique qui interagit avec une base de données. Démontrer la maîtrise de ces quatre opérations est essentiel pour valider les compétences en développement back-end.

Choix de l'exemple : La Gestion des Clients

Pour ce projet, le module de **Gestion des Clients** a été choisi comme l'exemple de référence pour démontrer la mise en œuvre complète du cycle CRUD .

Ce choix est stratégique pour plusieurs raisons :

1. **Centralité Métier :** Le client est l'entité au cœur de la logique de l'application. C'est à lui qu'on associe les devis , et c'est souvent à partir d'une demande de contact (NoSQL) qu'il est créé .
2. **Couverture Complète :** Ce module implémente naturellement les quatre opérations :
    - **C**reate : L'administrateur crée un nouveau client via un formulaire .
    - **R**ead : L'administrateur recherche et lit la liste des clients .
    - **U**pdate : L'administrateur modifie les informations d'un client existant.
    - **D**elete : L'administrateur supprime un client .
3. **Complexité Démontrée :** Au-delà du simple CRUD, ce module intègre une logique avancée :
    - Le **"Create"** peut être pré-rempli depuis un message NoSQL (page messages.php).
    - Le **"Read"** n'est pas une simple liste, mais un système de recherche (LIKE ?) .
    - L'**"Update"** contient une logique conditionnelle (ne pas changer le mot de passe s'il est laissé vide).

La section suivante décompose ce module en présentant chaque opération de CRUD de manière distincte, en illustrant le parcours de l'administrateur avec les captures d'écran des interfaces et les extraits de code PHP/SQL correspondant qui s'exécutent en arrière-plan.

Initialisation : Le prérequis à toute opération CRUD

Avant qu'une quelconque opération CRUD (Create, Read, Update, Delete) puisse être exécutée, deux étapes d'initialisation sont fondamentales pour garantir la **sécurité** et la **fonctionnalité** de l'application.

Ces étapes sont systématiquement exécutées en tête de chaque script d'action (comme client_actions.php ou devis_actions.php) ou de page protégée (comme dashboard.php).

- **Démarrage de la Session (_session_start.php) :** C'est la première étape, indispensable pour la **sécurité**. Démarrer la session permet au script d'accéder aux variables $_SESSION. C'est ce qui permet au "gardien de sécurité" de vérifier si un utilisateur est bien authentifié (par exemple, if (!isset($_SESSION['admin_id'])) et s'il a les droits nécessaires pour effectuer une modification dans la base de données.

- **Connexion à la Base de Données (database.php) :** La deuxième étape est de se connecter à la base de données. Les opérations CRUD lisent ou écrivent des données. Inclure ce fichier établit la connexion au serveur MySQL et fournit l'objet de connexion ($conn) qui est essentiel pour préparer (prepare()) et exécuter (execute()) les requêtes SQL .

Sans ces deux lignes, toute tentative d'action protégée échouerait : soit par manque d'autorisation (la session), soit par incapacité à communiquer avec les données (la base de données).

Extrait de code type (Exemple : admin/client_actions.php)

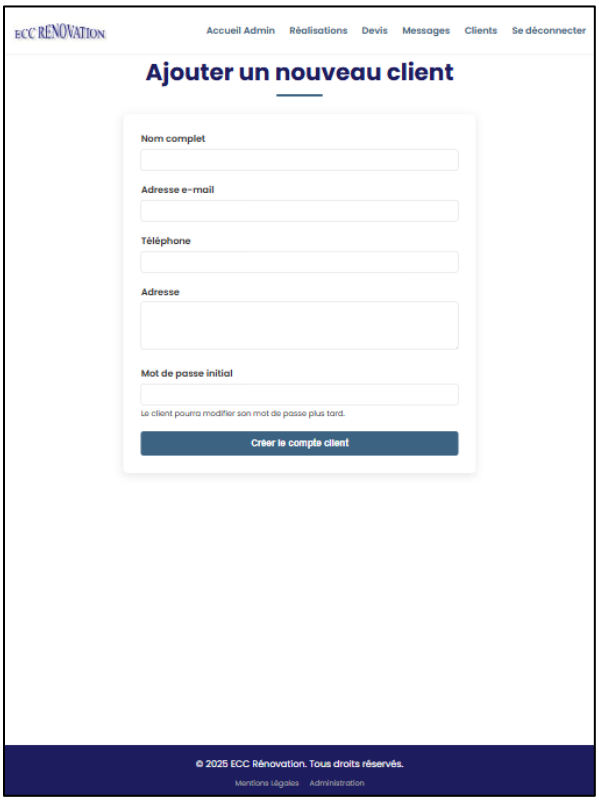
Voici à quoi ressemble le début d'un script d'action type, illustrant cette initialisation :

```php
<?php

// 1. DÉMARRER LA SESSION (Authentification)
require_once '../partials/_session_start.php';
```

```php
// 2. CONNEXION À LA BASE DE DONNÉES (Pour les opérations CRUD  Create, Read, Update, Delete)
require_once '../config/database.php';



// 3. GARDIEN DE SÉCURITÉ
// On vérifie que la variable de session 'admin_id' existe.
// Cette variable n'a pu être créée que par le script 'login_process.php'.
if (!isset($_SESSION['admin_id'])) {
    // Si la session n'existe pas, l'utilisateur n'est pas un admin connecté.
    // On lui envoie un code "403 Forbidden" (Accès Interdit).
    header('HTTP/1.1 403 Forbidden');
    // On arrête le script immédiatement.
    exit();
}

// 4. LOGIQUE CRUD
// Le script peut maintenant exécuter des requêtes sécurisées... par exemple :
if (isset($_POST['action']) && $_POST['action'] == 'add')
{
    // ...
    // $stmt = $conn->prepare(...);
    // ...
}
```

A. CREATE : Créer un nouveau client

Capture d'écran de la page `client_form.php` :



Extrait de code fichier : admin/client_action.php

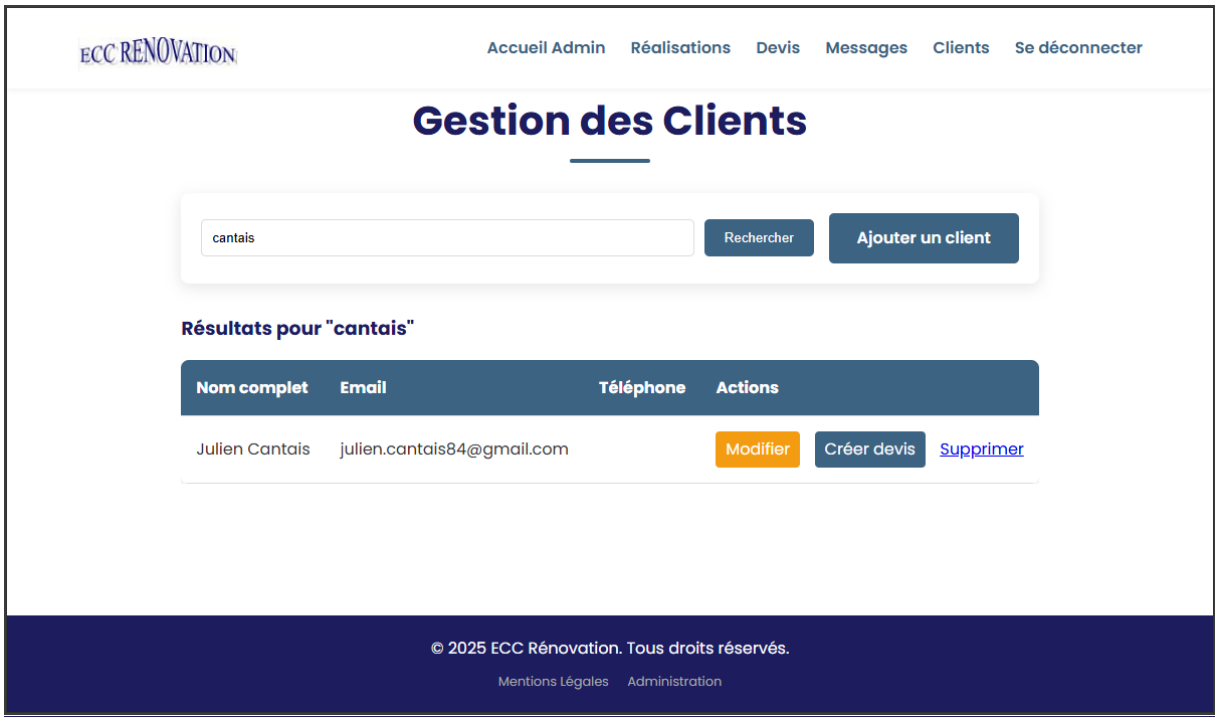
```php
// --- GESTION DE LA CRÉATION (CREATE) ---

// ROUTEUR D'ACTION (Logique "Create")
// On vérifie que le formulaire a été envoyé (POST) et
qu'il contient
// un champ 'action' avec la valeur 'add'.
if (isset($_POST['action']) && $_POST['action'] == 'add')
{

    // 3. RÉCUPÉRATION ET NETTOYAGE DES DONNÉES
    // On récupère les données du formulaire et on
supprime les espaces
    // superflus au début ou à la fin avec trim().
    $nom_complet = trim($_POST['nom_complet']);
```

```php
    $email = trim($_POST['email']);
    $telephone = trim($_POST['telephone']);
    $adresse = trim($_POST['adresse']);
    $mot_de_passe = $_POST['mot_de_passe'];

    // 4. VALIDATION DES DONNÉES
    // On vérifie que les champs obligatoires ne sont pas
vides.
    if (empty($nom_complet) || empty($email) ||
empty($mot_de_passe)) {
        // die() arrête le script et affiche un message.
C'est une validation simple.
        die("Erreur : Le nom, l'email et le mot de passe
sont obligatoires.");
    }
    // On utilise une fonction PHP robuste pour valider le
format de l'email.
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        die("Erreur : L'adresse email n'est pas valide.");
    }

    // 5. SÉCURITÉ : HACHAGE DU MOT DE PASSE
    // On ne stocke JAMAIS un mot de passe en clair.
    // On utilise password_hash() qui applique
l'algorithme BCRYPT (sécurisé).
    $mot_de_passe_hache = password_hash($mot_de_passe,
PASSWORD_DEFAULT);

    // 6. INSERTION SÉCURISÉE (Requête Préparée contre
l'injection SQL)
    // On définit le "modèle" de la requête avec des '?'
comme marqueurs.
    $sql = "INSERT INTO clients (nom_complet, email,
telephone, adresse, mot_de_passe) VALUES (?, ?, ?, ?, ?)";

    // On prépare la requête (on envoie le modèle au
serveur MySQL).
    $stmt = $conn->prepare($sql);

    // On lie les variables aux marqueurs '?' (on envoie
les données séparément).
```

```php
    // "sssss" signifie que les 5 variables sont des
chaînes de caractères (strings).
    // La BDD traite ces données comme du texte, jamais
comme du code SQL.
    $stmt->bind_param("sssss", $nom_complet, $email,
$telephone, $adresse, $mot_de_passe_hache);

    // On exécute la requête finale.
    $stmt->execute();
    // On ferme la requête préparée.
    $stmt->close();

    // 7. REDIRECTION
    // Une fois le client créé, on redirige
l'administrateur vers la liste des clients.
    header('Location: clients.php');
    exit(); // On arrête le script.
}
```

B. READ : Rechercher et lire les clients

Capture d'écran de la page `client.php` :



Extrait de code fichier : `admin/clients.php`

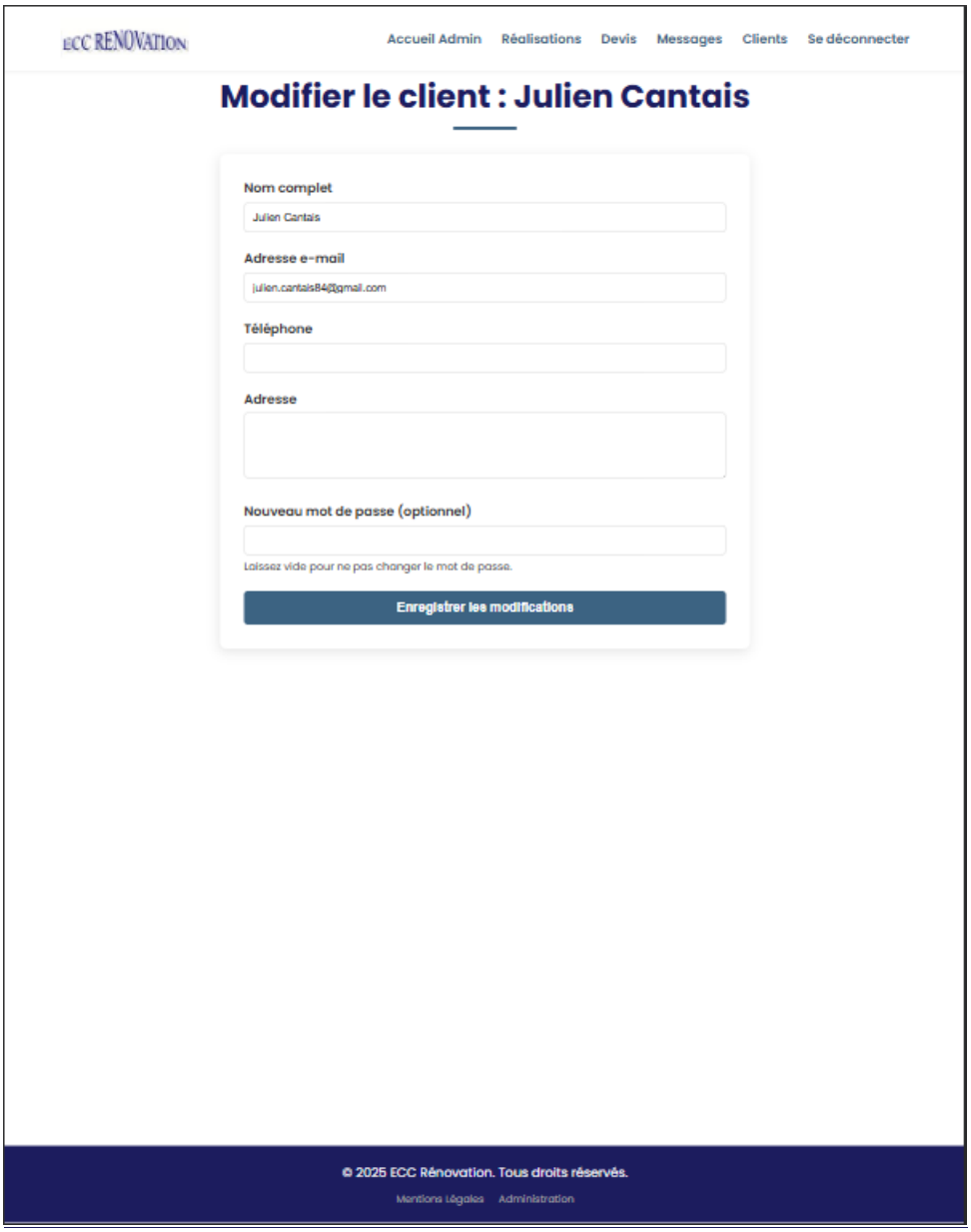
```php
// Initialisation d'un tableau vide pour les clients et du terme de recherche
$clients = [];
$search_term = '';

// Vérifie si le formulaire de recherche a été soumis (via la méthode GET)
if (isset($_GET['search'])) {

    // Nettoie le terme de recherche (supprime les espaces superflus au début et à la fin)
    $search_term = trim($_GET['search']);

    // N'exécute la recherche que si le terme n'est pas vide
    if (!empty($search_term)) {
```

```
        // 1. DÉFINITION DE LA REQUÊTE
        // Requête SQL utilisant "LIKE ?" pour trouver des
correspondances partielles
        // L'opérateur '?' est un marqueur de position
pour la requête préparée.
        $sql = "SELECT id, nom_complet, email, telephone
FROM clients WHERE nom_complet LIKE ? ORDER BY nom_complet
ASC";

        // 2. PRÉPARATION (Sécurité)
        // Prépare la requête SQL, la sépare des données
pour éviter les injections SQL.
        $stmt = $conn->prepare($sql);

        // 3. LIAISON (Binding)
        // Ajoute les wildcards '%' au terme de recherche
pour chercher "contient"
        // (ex: "dup" devient "%dup%")
        $search_pattern = "%" . $search_term . "%";
        // Lie la variable $search_pattern au premier '?'
de la requête ($stmt).
        // Le "s" indique que la variable est une chaîne
de caractères (string).
        $stmt->bind_param("s", $search_pattern);

        // 4. EXÉCUTION
        // Exécute la requête sur le serveur de base de
données.
        $stmt->execute();

        // 5. RÉCUPÉRATION
        // Récupère l'ensemble des résultats de la
requête.
        $result = $stmt->get_result();
        // Transforme tous les résultats en un tableau
associatif PHP.
        $clients = $result->fetch_all(MYSQLI_ASSOC);

        // 6. NETTOYAGE
        // Ferme la requête préparée pour libérer les
ressources.
        $stmt->close();
```

```
    }
}
// Ferme la connexion à la base de données.
$conn->close();
```

C. UPDATE : Modifier un client

Capture d'écran de la page formulaire `client_edit_form.php`:



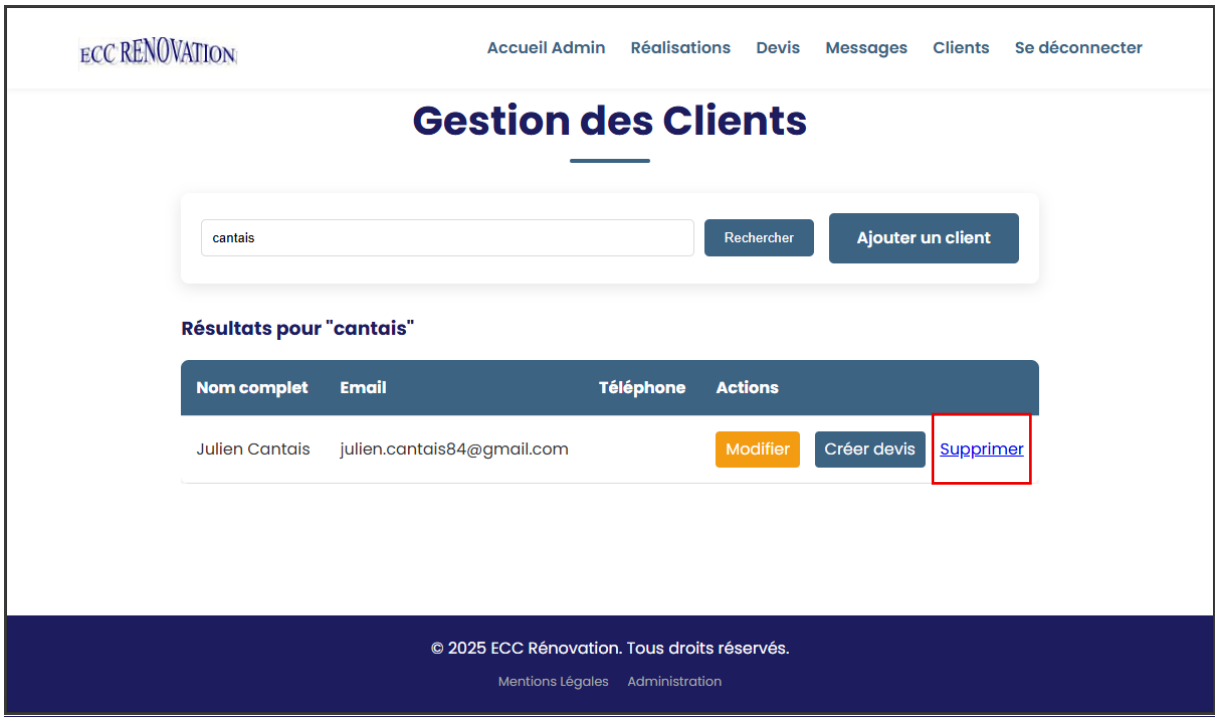
Extrait de code fichier : admin/client_action.php

```
// --- GESTION DE LA MODIFICATION (UPDATE) ---

// Vérifie que le formulaire a été envoyé avec l'action
"edit"
if (isset($_POST['action']) && $_POST['action'] == 'edit')
{

    // 1. RÉCUPÉRATION ET NETTOYAGE DES DONNÉES
    // Récupère l'ID du client, forcé en entier (intval)
pour la sécurité
    $client_id = intval($_POST['client_id']);
    // Récupère les données texte en supprimant les
espaces blancs au début/fin
    $nom_complet = trim($_POST['nom_complet']);
    $email = trim($_POST['email']);
    $telephone = trim($_POST['telephone']);
    $adresse = trim($_POST['adresse']);
    $mot_de_passe = $_POST['mot_de_passe']; // Récupère le
champ (peut être vide)

    // 2. VALIDATION DES DONNÉES
    // S'assure que les champs critiques ne sont pas vides
    if (empty($nom_complet) || empty($email) || $client_id
== 0) {
        die("Erreur : Le nom, l'email et l'ID client sont
obligatoires.");
    }

    // 3. LOGIQUE MÉTIER CONDITIONNELLE (Mot de passe)
    // C'est le cœur de la logique "Update" :
    // On vérifie si un NOUVEAU mot de passe a été saisi.
    if (!empty($mot_de_passe)) {

        // CAS 1 : L'admin a fourni un nouveau mot de
passe
        // On hache le nouveau mot de passe pour le
stocker en toute sécurité
        $mot_de_passe_hache = password_hash($mot_de_passe,
PASSWORD_DEFAULT);
```

```
        // On prépare une requête SQL qui met à jour TOUS
les champs, y compris le mot de passe
        $sql = "UPDATE clients SET nom_complet = ?, email
= ?, telephone = ?, adresse = ?, mot_de_passe = ? WHERE id
= ?";
        $stmt = $conn->prepare($sql);
        // On lie 6 paramètres ("ssssi" -> 5 strings, 1
integer)
        $stmt->bind_param("ssssi", $nom_complet, $email,
$telephone, $adresse, $mot_de_passe_hache, $client_id);
    } else {

        // CAS 2 : Le champ mot de passe a été laissé vide
        // On prépare une requête SQL qui met à jour tous
les champs SAUF le mot de passe
        $sql = "UPDATE clients SET nom_complet = ?, email
= ?, telephone = ?, adresse = ? WHERE id = ?";
        $stmt = $conn->prepare($sql);
        // On ne lie que 5 paramètres ("ssssi" -> 4
strings, 1 integer)
        $stmt->bind_param("ssssi", $nom_complet, $email,
$telephone, $adresse, $client_id);
    }

    // 4. EXÉCUTION
    // Exécute la requête préparée (soit celle avec mdp,
soit celle sans)
    $stmt->execute();
    $stmt->close();

    // 5. REDIRECTION
    // Redirige l'admin vers la page de recherche, en
affichant directement le client modifié
    header('Location: clients.php?search=' .
urlencode($nom_complet));
    exit();
}
```

D. DELETE : Supprimer un client

Capture d'écran de la page formulaire `client.php`:



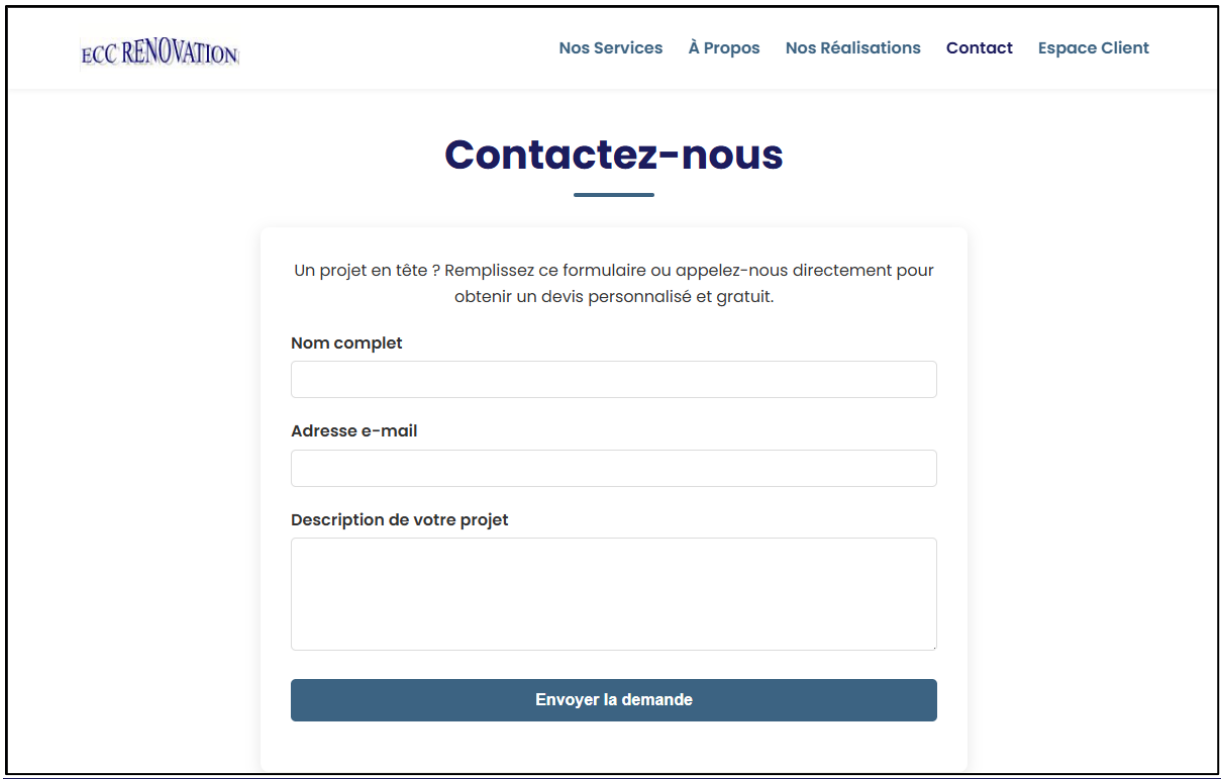
Extrait de code fichier : `admin/client_action.php`

```php
// --- GESTION DE LA SUPPRESSION D'UN CLIENT ---

// 1. ROUTEUR D'ACTION (Logique "Delete")
// Vérifie si l'URL contient les paramètres (ex:
...?action=delete&id=12)
if (isset($_GET['action']) && $_GET['action'] == 'delete')
{

    // 2. SÉCURISATION DE L'ENTRÉE
    // Récupère l'ID depuis l'URL et le force à être un
nombre entier.
    // C'est une mesure de sécurité cruciale
(assainissement)
    // pour empêcher les injections SQL via le paramètre
'id'.
    $id = intval($_GET['id']);

    // 3. PRÉPARATION DE LA REQUÊTE (Sécurité)
```

```php
    // Définit le "modèle" SQL avec un marqueur de
position '?'.
    $sql = "DELETE FROM clients WHERE id = ?";
    // Prépare la requête, la séparant des données.
    $stmt = $conn->prepare($sql);

    // Lie la variable $id (qui est un entier) au marqueur
'?'.
    // Le "i" signifie que la variable est de type
Integer.
    $stmt->bind_param("i", $id);

    // 4. EXÉCUTION
    // Exécute la suppression dans la base de données.
    $stmt->execute();
    // Ferme la requête préparée.
    $stmt->close();

    // 5. REDIRECTION
    // Redirige l'administrateur vers la page de gestion
des clients.
    header('Location: clients.php');
    // Arrête le script pour s'assurer qu'aucun autre code
n'est exécuté.
    exit();
}
```

3.2.2 Composant d'accès NoSQL : Enregistrement d'un message (MongoDB)

Capture d'écran de la page formulaire présent au bas de la page d'accueil (`index.php`)



Pour l'accès NoSQL, la bibliothèque PHP MongoDB est utilisée via Composer. L'insertion se fait via la méthode insertOne, qui enregistre un document BSON, démontrant ainsi la maîtrise des deux types d'accès.

Extrait de code fichier : `api/submit_contact.php`

```php
<?php
// On charge la connexion MongoDB et l'autoloader de Composer
require_once '../config/mongodb.php';

// On indique que la réponse sera du JSON
header('Content-Type: application/json');

// On vérifie que la méthode de requête est bien POST
```

```php
if ($_SERVER['REQUEST_METHOD'] == 'POST') {

    // --- SÉCURITÉ : RÉCUPÉRATION ET NETTOYAGE DES
DONNÉES ---
    $nom = trim($_POST['name']);
    $email = trim($_POST['email']);
    $message = trim($_POST['message']);

    // Validation simple : on vérifie que les champs ne
sont pas vides
    if (empty($nom) || empty($email) || empty($message)) {
        echo json_encode(['success' => false, 'message' =>
'Tous les champs sont obligatoires.']);
        exit();
    }
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        echo json_encode(['success' => false, 'message' =>
'L\'adresse e-mail n\'est pas valide.']);
        exit();
    }

    // --- INSERTION DANS MONGODB ---
    try {
        // On insère un "document" dans la collection
        $insertResult = $collectionMessages->insertOne([
            'nom' => $nom,
            'email' => $email,
            'message' => $message,
            'date_soumission' => new
MongoDB\BSON\UTCDateTime(), // Format de date NoSQL
            'statut' => 'nouveau'
        ]);

        // On vérifie si l'insertion a réussi
        if ($insertResult->getInsertedCount() === 1) {
            echo json_encode(['success' => true, 'message'
=> 'Votre message a bien été envoyé. Nous vous répondrons
rapidement !']);
        } else {
            echo json_encode(['success' => false,
'message' => 'Une erreur est survenue lors de
l\'enregistrement.']);
```

```
        }
    } catch (Exception $e) {
        // Gère les erreurs de connexion ou d'insertion
        echo json_encode(['success' => false, 'message' =>
'Erreur de base de données NoSQL : ' . $e->getMessage()]);
    }
} else {
    // Si la méthode n'est pas POST, on renvoie une erreur
    echo json_encode(['success' => false, 'message' =>
'Méthode non autorisée.']);
}
```

3.3. Composants métier (PHP)

La logique métier gère les règles de l'application, comme l'authentification et la gestion des sessions.

Exemple : Logique métier hybride (espace-client/demande_devis.php)

Ce script illustre parfaitement la persistance polyglotte. Il **lit** les informations du client dans la base **SQL (MySQL)** pour récupérer son nom, puis **écrit** la nouvelle demande de devis dans la base **NoSQL (MongoDB)**.

```php
<?php
// 1. Démarrer la session
require_once '../partials/_session_start.php';
// 2. Connexion à MySQL (pour LIRE les infos du client)
require_once '../config/database.php';
// 3. Connexion à MongoDB (pour ÉCRIRE le message)
require_once '../config/mongodb.php';

// Gardien de sécurité : le client doit être connecté
if (!isset($_SESSION['client_id'])) {
    header('HTTP/1.1 403 Forbidden');
    exit();
}

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $message_texte = trim($_POST['message']);
```

```
    // On récupère les infos du client depuis la session
    $client_id = $_SESSION['client_id'];

    // --- ÉTAPE 1 : LECTURE DEPUIS MYSQL ---
    $stmt_client = $conn->prepare("SELECT nom_complet,
email FROM clients WHERE id = ?");
    $stmt_client->bind_param("i", $client_id);
    $stmt_client->execute();
    $result_client = $stmt_client->get_result();
    $client_data = $result_client->fetch_assoc();
    $stmt_client->close();
    $conn->close(); // On ferme la connexion MySQL

    $nom = $client_data['nom_complet'] . " (Client
existant)";
    $email = $client_data['email'];

    if (empty($message_texte)) {
        die("Le message ne peut pas être vide.");
    }

    // --- ÉTAPE 2 : ÉCRITURE DANS MONGODB ---
    try {
        $insertResult = $collectionMessages->insertOne([
            'nom' => $nom,
            'email' => $email,
            'message' => $message_texte,
            'date_soumission' => new
MongoDB\BSON\UTCDateTime(),
            'statut' => 'nouveau'
        ]);

        if ($insertResult->getInsertedCount() === 1) {
            header('Location: dashboard.php?success=1');
            exit();
        } else {
            die("Une erreur est survenue lors de
l'enregistrement dans MongoDB.");
        }
    } catch (Exception $e) {
```

```
        die("Erreur de base de données NoSQL : " . $e-
>getMessage());
    }
}
```

IV. Sécurité, Tests et Veille

4.1. Présentation des éléments de sécurité

La sécurité a été une priorité à chaque étape du développement :

- **Contre les injections SQL :** Utilisation systématique des requêtes préparées (MySQLi) pour toutes les requêtes SQL, comme démontré ci-dessus.
- **Contre les failles XSS (Cross-Site Scripting) :** Utilisation de la fonction htmlspecialchars() avant tout affichage de données provenant de la base de données (noms de clients, messages, titres de devis, etc.).
- **Sécurité des mots de passe :** Les mots de passe ne sont jamais stockés en clair. Ils sont hachés avec l'algorithme BCRYPT via password_hash() et vérifiés avec password_verify().

  Gestion des accès : Le site utilise les sessions PHP ($_SESSION) pour séparer les droits. Des "gardiens de sécurité" sont placés au début de chaque page admin et client pour vérifier si l'utilisateur est connecté et a le bon rôle.
- **Sécurisation des identifiants :** Les identifiants de la base de données ne sont pas versionnés sur Git. Ils sont stockés dans un fichier .env local, qui est lu par PHP mais ignoré par Git grâce au fichier .gitignore.

Contrôle d'accès basé sur les Rôles (Gardiens de Sécurité) :
L'application gère des rôles distincts (Administrateur, Client). L'accès à chaque page sensible est protégé par un "gardien" placé

au tout début du script. Ce gardien vérifie si la variable de session ($_SESSION['admin_id'] ou $_SESSION['client_id']) existe. Si elle n'existe pas, l'utilisateur est immédiatement redirigé vers la page de connexion, empêchant toute exécution de code ou accès non autorisé.

Extrait de code (`admin/dashboard.php`) :

```php
<?php
// 1. DÉMARRER LA SESSION
require_once '../partials/_session_start.php';

// 2. CONNEXION AUX BASES DE DONNÉES
require_once '../config/database.php'; // Pour les projets et clients (SQL)
require_once '../config/mongodb.php'; // Pour les messages (NoSQL)

// Le gardien de sécurité
if (!isset($_SESSION['admin_id'])) {
    header('Location: index.php');
    exit();
}
```

De plus, les scripts d'action (comme `client_actions.php`) renvoient un code `HTTP 403 Forbidden` pour bloquer les requêtes directes non authentifiées, ce qui est une pratique de sécurité robuste pour les API.

4.2. Jeu d'essai

Un jeu d'essai a été réalisé pour la fonctionnalité la plus représentative : **l'acceptation d'un devis par un client.**

Scénario	Données en entrée	Résultat Attendu	Résultat Obtenu	Analyse
1. Acceptation	Client connecté, clique sur "Accepter" (Devis ID 1, Statut = 'En attente')	Le statut du devis 1 passe à 'Accepté' dans la BDD.	Le statut est mis à jour à 'Accepté'.	Conforme
2. Refus	Client connecté, clique sur "Refuser" (Devis ID 2, Statut = 'En attente')	Le statut du devis 2 passe à 'Refusé'.	Le statut est mis à jour à 'Refusé'.	Conforme
3. Paiement (simulé)	Client connecté, clique sur "Payer" (Devis ID 1, Statut = 'Accepté')	Le statut du devis 1 passe à 'Payé'.	Le statut est mis à jour à 'Payé'.	Conforme
4. Accès non autorisé	Client B tente d'accéder à voir_devis.php?id=1 (Devis du Client A)	Redirection vers le dashboard.php du Client B.	Redirection vers le dashboard.php.	Conforme

4.3. Veille technique et de sécurité

Une veille active a été maintenue tout au long du projet pour garantir l'utilisation des bonnes pratiques et la compréhension des risques. Cette veille s'est articulée autour de trois axes :

1. Veille de Sécurité (OWASP)

- **Source :** Consultation régulière du **Top 10 de l'OWASP** (Open Web Application Security Project).

- **Application Concrète :** L'étude de l'OWASP a directement influencé l'architecture du projet :
    - **A01:2021 (Broken Access Control) :** M'a conduit à implémenter les "Gardiens de Sécurité" (vérification de $_SESSION) au début de chaque script sensible .
    - **A03:2021 (Injection) :** A confirmé la nécessité absolue d'utiliser des **requêtes préparées** pour MySQL (contre l'injection SQL) et des bibliothèques officielles pour MongoDB (contre l'injection NoSQL).
    - **A07:2021 (Identification and Authentication Failures) :** M'a poussé à utiliser password_hash() et password_verify() (BCRYPT) pour les mots de passe, plutôt que des algorithmes obsolètes comme MD5 .

2. Veille Technologique (Documentation et Bonnes Pratiques)

- **Sources :** Documentation officielle de PHP, documentation de MongoDB, ressources de STUDI.
- **Application Concrète :**
    - **Architecture :** C'est cette veille qui m'a orienté vers l'architecture de **persistance polyglotte** (SQL + NoSQL), une pratique moderne pour gérer différents types de données de manière optimale.
    - **Environnement :** La compréhension des problèmes de "ça marche sur ma machine" m'a conduit à adopter **Docker** et docker-compose pour créer un environnement de développement reproductible et prêt pour la production.

3. Outils de Veille et d'Assistance (IA)

- **Source :** Utilisation d'assistants IA (type Gemini) comme outil de "pair programming".
- **Application Concrète :** L'IA n'a pas écrit le code, mais a servi de partenaire de réflexion pour :
    - **Débogage :** Accélérer la résolution de problèmes complexes (ex: erreurs de "headers already sent" ou conflits de port Docker).
    - **Validation d'architecture :** Confirmer la pertinence d'utiliser le script `espace-client/demande_devis.php` comme exemple de composant métier hybride (lecture SQL / écriture NoSQL) .
    - **Syntaxe :** Servir de documentation interactive pour des points précis (ex: la syntaxe de `grid-template-columns`).

Cette veille m'a permis de ne pas seulement exécuter les tâches, mais aussi de comprendre *pourquoi* je les exécutais d'une certaine manière, en accord avec les standards actuels de l'industrie.

V. Conclusion

Le projet "ECC Rénovation" répond non seulement à tous les objectifs fonctionnels fixés par le cahier des charges, mais il m'a surtout permis de construire une application web complète en intégrant des architectures et des méthodologies professionnelles modernes.

Au-delà de la réalisation d'un CRUD complet, ce projet démontre la maîtrise de concepts avancés :

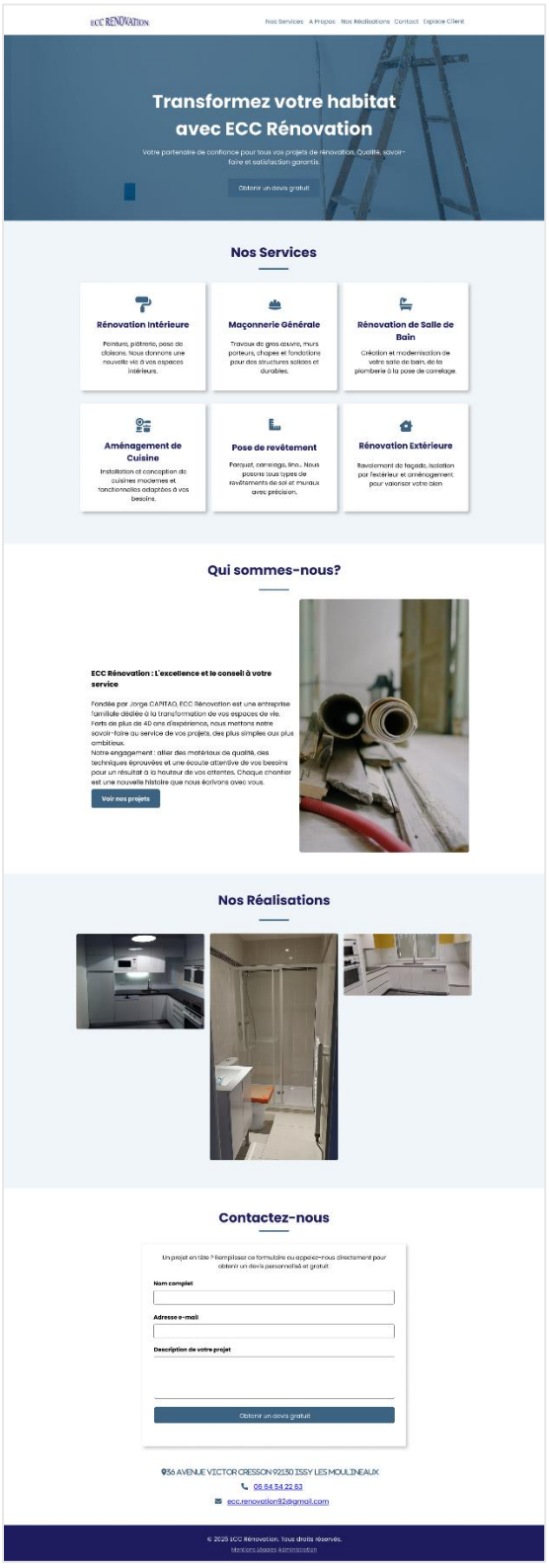
1. **Une architecture de persistance polyglotte (SQL + NoSQL)** a été mise en place, en utilisant MySQL pour sa robustesse transactionnelle (clients, devis) et MongoDB pour sa flexibilité documentaire (messages).
2. L'ensemble de l'application a été **conteneurisé avec Docker** , garantissant un environnement de développement stable, portable et identique à celui de la production.
3. La **sécurité** a été un pilier central, avec une protection rigoureuse contre les failles (XSS, injections SQL, CSRF) et une gestion des accès basée sur les rôles et les sessions .
4. Le code est **organisé et maintenable** , grâce à l'architecture SCSS modulaire (BEM/Partials) et à l'application d'un workflow Git professionnel (Feature Branch Workflow) .

Sur le plan personnel, ce projet a été un défi formateur, me permettant de passer de la théorie (maîtriser la syntaxe PHP) à la pratique (architecturer et orchestrer un système multi-services complexe).

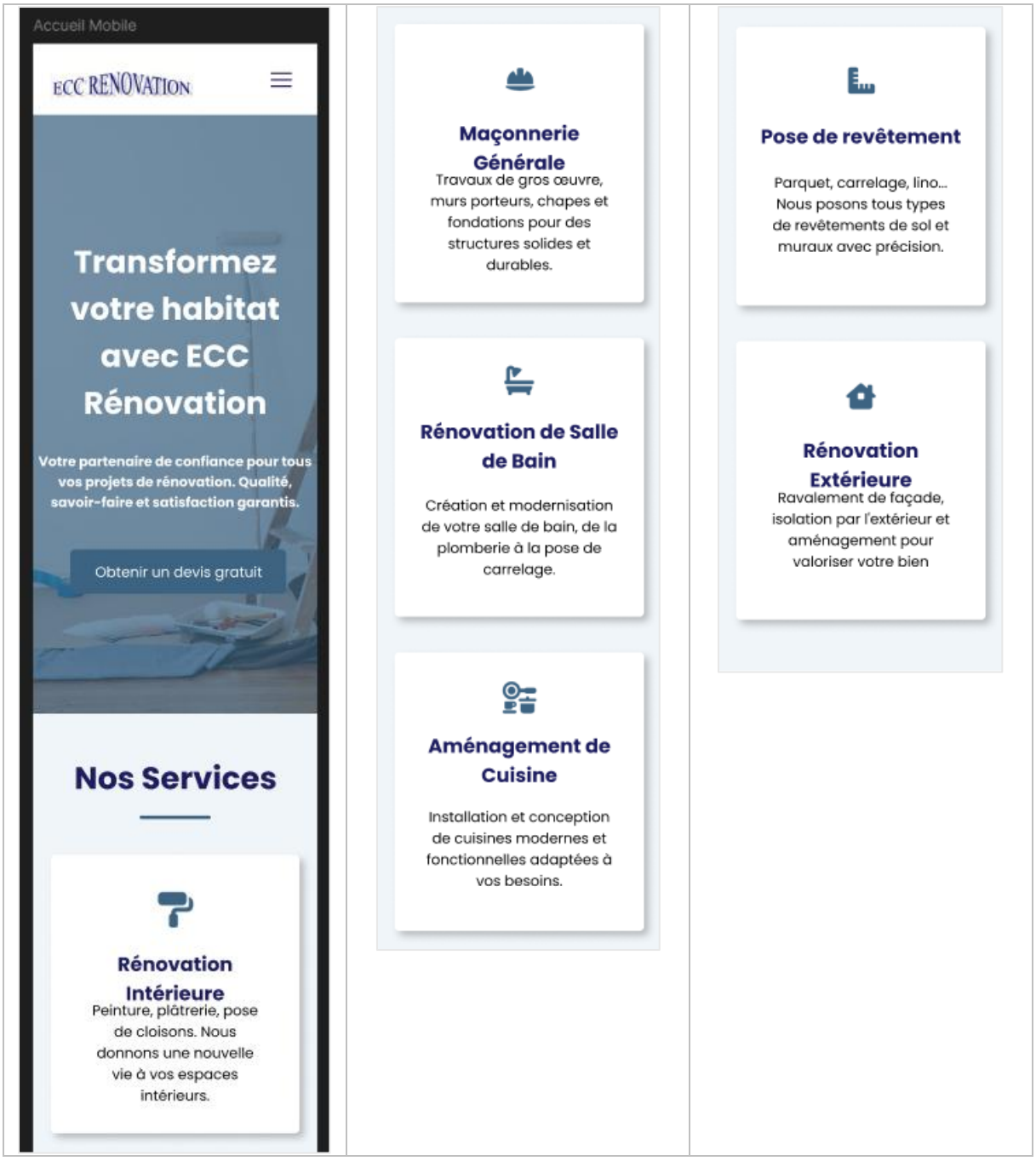
L'application est aujourd'hui fonctionnelle, sécurisée, et prête à évoluer.

VI. Annexes

Annexe 1 : Maquette Figma Accueil (Desktop)



Annexe 2 : Maquette Figma Accueil (Mobile)

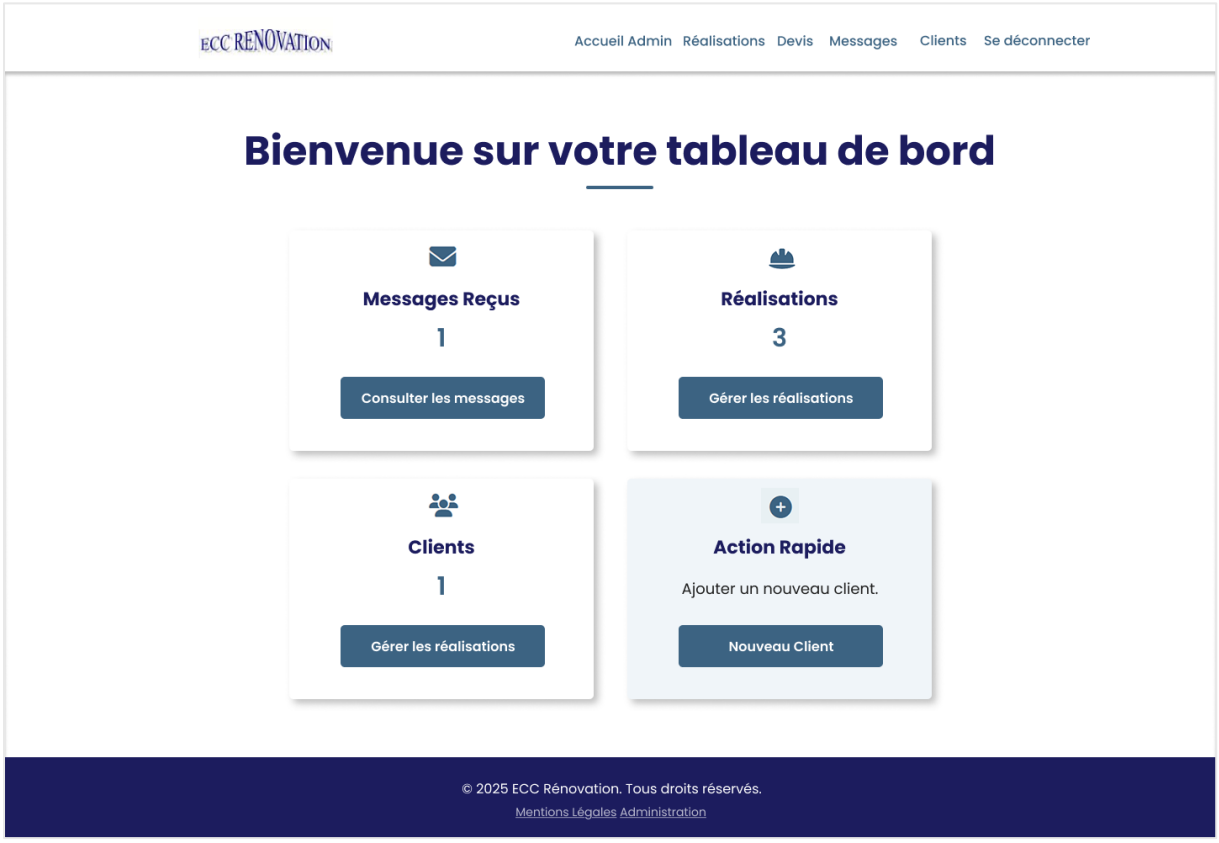


1	2	3

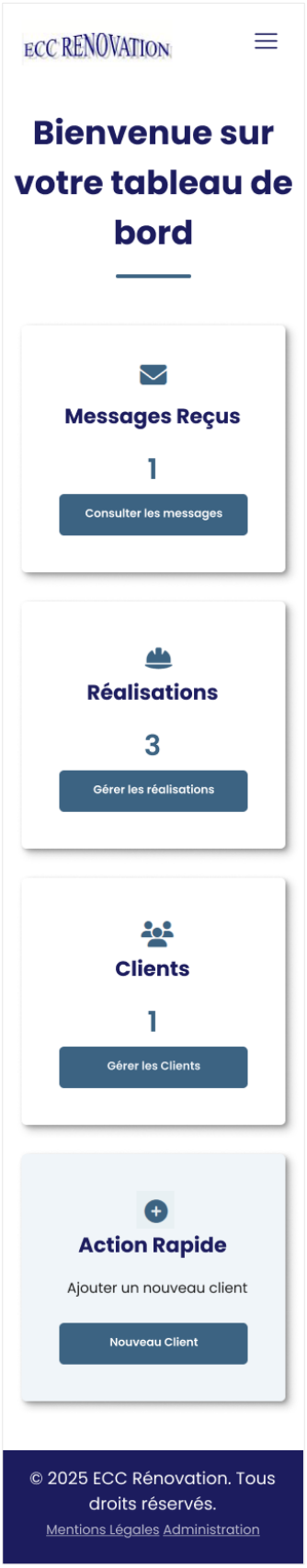
Suite de Maquette Figma Accueil (Mobile)

		
4	5	6

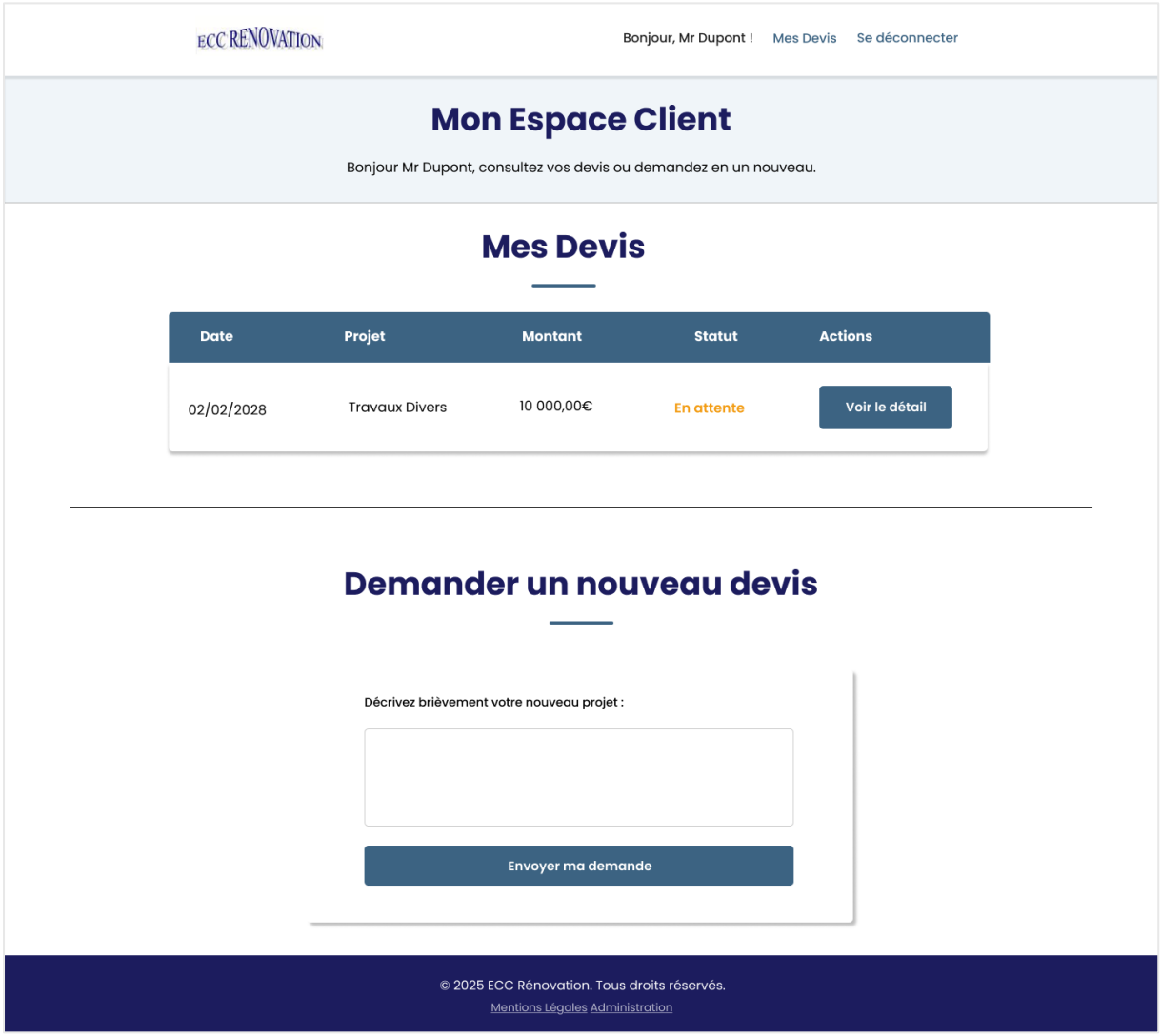
Annexe 3 : Maquette Figma Tableau de bord Admin (Mobile)



Annexe 4 : Maquette Figma Tableau de bord Admin (Mobile)

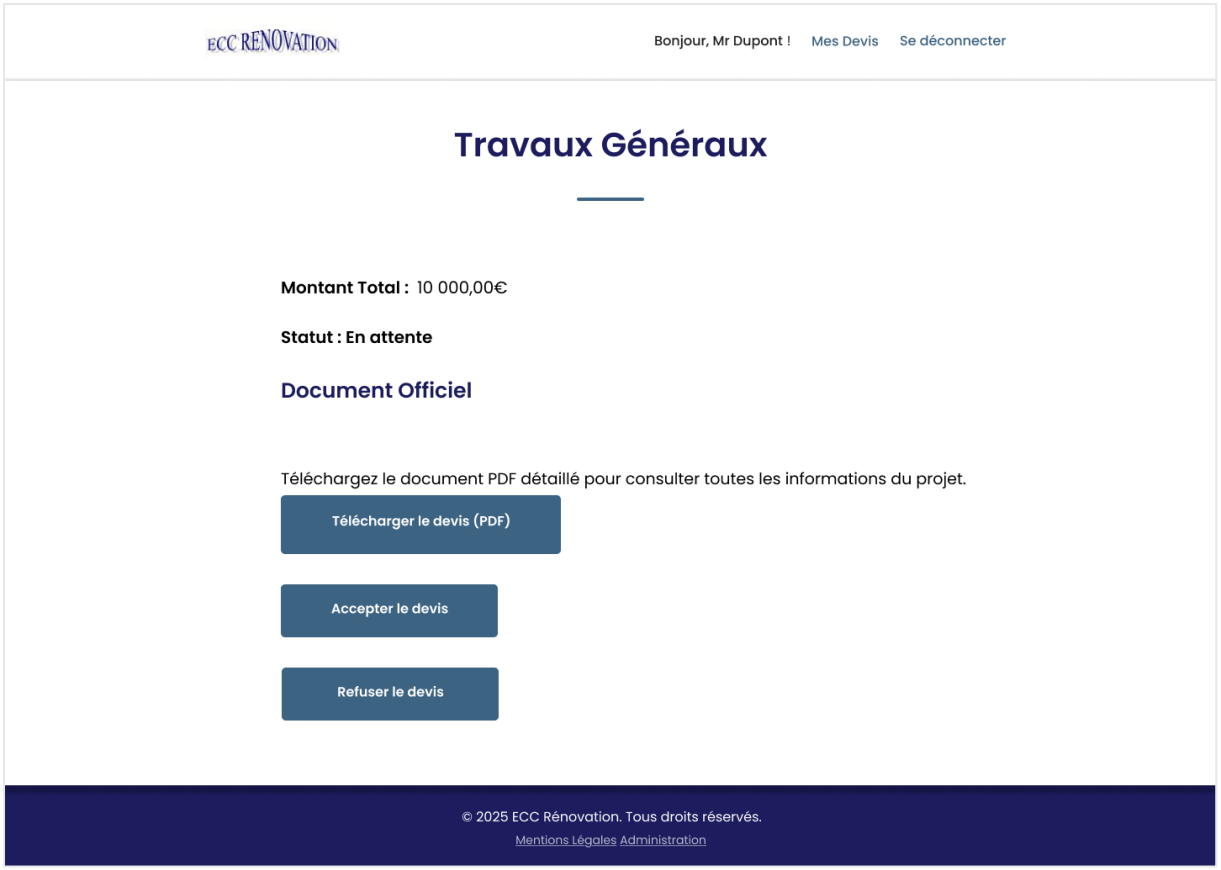


Annexe 5 : Maquette Figma Espace Client (Desktop)



Annexe 6 : Maquette Figma Espace Client (Mobile)

Annexe 7 : Maquette Figma de la page Espace Client/ Détail d'un devis (Desktop)



Annexe 8 : Maquette Figma de la page Espace Client/ Détail d’un devis (Mobile)