|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **Informaticien/-ne CFC**  Travail pratique individuel 2025 (TPI) |  | |
|  | |

Modèle de rapport v1.4

Candidat No 151464

Nom du candidat : Max Langenegger

Elevate – Projet TPI

Elevate – Projet TPI 0

1 Résumé du rapport du TPI 2

1.1 Situation de départ 2

1.2 Mise en œuvre 2

1.3 Résultats 2

2 Les grandes lignes du projet 3

2.1 Cahier des charges / exigences du système 3

2.2 Organisation du projet 4

3 Analyse 6

3.1 Objectifs du système 6

3.2 Choix de variante 10

3.3 Sécurité de l’information et protection des données 10

3.4 Diagramme des cas d’utilisation 12

3.5 Diagramme d’activité 13

4 Conception 17

4.1 Diagramme de classe 17

5.2 Diagramme de séquence 20

5.3 Architecture du système 20

5.4 Concept de tests 21

5.5 Moyens nécessaires 22

6 Réalisation 23

6.1 Déploiement 23

6.2 Schéma de la base de données 23

6.3 Fonctionnement du mode offline (PWA) 25

6.4 Chargement des données hors-ligne 26

6.5 Chargement du classement 28

6.6 Récupérer les informations du concurrent 31

6.7 Envoie des requêtes en attentes 32

6.8 Gestion des erreurs 36

7 Test 37

7.1 Procédure de test 37

7.2 Protocol de test 38

7.3 Signature du protocole de test 39

8 Tests Techno 40

8.1 Test PWA 40

8.2 PWA mode hors-ligne 42

8.3 PWA statuts réseau 44

8.4 Envoi et réception d’image 46

8.5 Requêtes en attentes et base de données locale 48

9 Conclusion 51

9.1 Améliorations possibles 51

9.2 Auto-évaluation 51

10 Bibliographie: liste des sources et références 53

11 Signatures 54

12 Annexes 55

# Résumé du rapport du TPI

## Situation de départ

Le système de gestion des résultats des compétitions de marche et vol était jusqu’à présent entièrement manuel. Les commissaires de course notaient les résultats sur papier, ce qui entraînait des erreurs de saisie, des retards dans le traitement des données et une absence de mise à jour en temps réel du classement. Cette situation a motivé la conception d’une solution numérique permettant d’automatiser la saisie et la gestion des résultats de manière fiable et sécurisée.

## Mise en œuvre

Le projet Elevate a été développé sous forme d’une application web Progressive Web App (PWA), accessible sur smartphone, tablette et PC. L’architecture du projet est divisée en deux parties : un client (frontend) en HTML, CSS et JavaScript, et un serveur (backend) en PHP et MySQL. Les fonctionnalités principales incluent l’affichage du classement en temps réel, la saisie des résultats et des malus par les commissaires, et la possibilité de synchroniser les résultats hors ligne grâce à IndexedDB et un service worker. La sécurité des données est assurée par l’utilisation de requêtes préparées pour prévenir les injections SQL et par le hachage des mots de passe pour protéger les accès.

## Résultats

L’application Elevate permet désormais aux commissaires d’enregistrer rapidement et efficacement les résultats des concurrents, même en l’absence de connexion Internet, et de les synchroniser dès que la connexion est rétablie. Les visiteurs peuvent consulter le classement général à jour depuis leur navigateur. L’interface est ergonomique et adaptée aux écrans tactiles pour une meilleure utilisation sur le terrain. Cette solution améliore significativement la fiabilité et la rapidité du processus de gestion des résultats, tout en répondant aux besoins identifiés dans le cahier des charges.

# Les grandes lignes du projet

## Cahier des charges / exigences du système

Ce cahier des charges décrit de manière précise et mesurable les objectifs à atteindre pour la réalisation de l’application **Elevate**, destinée à la gestion des compétitions de marche et vol. Le système doit permettre la saisie, la consultation et la synchronisation des performances des concurrents par les commissaires, via une application mobile Progressive Web App (PWA) sécurisée, rapide et intuitive.

**A. Objectifs généraux**

* Développer une application mobile de type PWA fonctionnant sur tous les smartphones modernes, sans installation via un store.
* Offrir une expérience utilisateur fluide, même dans des zones avec une mauvaise couverture réseau.
* Garantir la sécurité des données, notamment par le hachage des mots de passe et la protection contre les injections SQL.
* Permettre une synchronisation automatique ou manuelle des données saisies hors ligne dès que la connexion est rétablie.
* Mettre à disposition un classement consultable en temps réel, sans authentification.

**B. Exigences fonctionnelles**

|  |  |  |
| --- | --- | --- |
| ID | Exigence fonctionnelle | Mesure de succès |
| EF1 | Affichage en temps réel du classement général | Classement trié automatiquement par points décroissants |
| EF2 | Détail complet des performances d’un concurrent | Affichage des types de postes, horaires, points |
| EF3 | Authentification des commissaires | Connexion via e-mail + mot de passe ou code PIN |
| EF4 | Mémorisation du login précédent pour simplifier la reconnexion | Champ prérempli après une première connexion |
| EF5 | Affichage des postes affectés à chaque commissaire | Liste filtrée automatiquement selon l’identifiant utilisateur |
| EF6 | Saisie d’un résultat avec formulaire interactif | Enregistrement réussi avec réinitialisation du formulaire |
| EF7 | Enregistrement de malus avec remarque obligatoire | Malus ajouté au concurrent, consultable dans ses résultats |
| EF8 | Fonctionnement offline avec synchronisation des résultats en attente | Données stockées localement et envoyées une fois connecter |
| EF9 | Indicateur graphique en cas de résultats non synchronisés | Icône ou message visible jusqu’à synchronisation complète |
| EF10 | Affichage d’un message d’erreur si le dossard est invalide | Notification claire sans plantage du système |

**C. Exigences techniques**

|  |  |  |
| --- | --- | --- |
| ID | Exigence technique | Mesure de succès |
| ET1 | Application développée en HTML, CSS, JS (client) et PHP/MySQL (serveur) | Fonctionnalités réparties selon architecture client/serveur |
| ET2 | Backend développé en PHP sous forme d’API REST sécurisée | Appels AJAX fonctionnels avec retours JSON |
| ET3 | Mot de passe des commissaires haché avec password\_hash() | Aucun mot de passe en clair dans la base |
| ET4 | Protection contre les injections SQL via requêtes préparées PDO | Aucune concaténation de chaînes dans les requêtes SQL |
| ET5 | Fichiers manifest.json et sw.js présents pour la gestion PWA | Fonctionnement hors ligne et possibilité d’installation |
| ET6 | Base de données MySQL avec structure claire (résultats, utilisateurs, postes, malus, etc.) | Schéma relationnel complet respecté |
| ET7 | Affichage responsive adapté aux mobiles | Navigation fluide sur petit écran, boutons accessibles |

**D. Exclusions du projet**

Conformément au mandat, certaines fonctionnalités sont hors périmètre :

* La gestion (création, modification, suppression) des participants.
* La gestion des comptes utilisateurs des commissaires.
* La gestion dynamique des postes et affectations.  
  Ces données seront saisies manuellement en base de données avant la course.

## Organisation du projet

Pour l’organisation de mon projet, j’ai choisi d’utiliser un planning en mode cascade qui se trouvera dans un fichier Excel disponible dans les annexes. Une feuille *Journal de bord* sera aussi présente dans ce classeur Excel qui servira à lister les heures passés avec la description des tâches effectuées.

Plusieurs sauvegardes du projet sont faîtes quotidiennement, une se trouve en local dans les fichiers de mon téléphone, une version est disponible sur mon [github](https://github.com/langeneggerm/TPI_Elevate) et une autre version se trouve sur une clé USB personnel. Ces trois supports permettent d’avoir plusieurs issues de secours si mon environnement de travail qui est OneDrive lâche.

Mon chef de projet actuel pour mon TPI est M. Galley Philippe qui supervisera ce projet durant les deux semaines consacrées au TPI. Deux experts sont aussi présents sur ce projet,   
M. Gérald Schaller qui est l’expert principal et M. Jean-Pierre Gauch qui est l’expert secondaire.

# Analyse

## Objectifs du système

### Analyse de l’état actuel

Actuellement, la gestion des compétitions de marche et vol repose sur un système manuel et papier, comme c’est le cas lors d’événements tels que la Millets Cup. Chaque commissaire de course est chargé d’enregistrer les résultats des concurrents à la main sur des fiches ou formulaires. Ces documents sont ensuite transmis physiquement à l’organisation centrale pour traitement.

Ce mode de fonctionnement présente plusieurs **limitations majeures** :

* **Risque d’erreurs humaines** : les erreurs de saisie, les oublis ou les mauvaises retranscriptions sont fréquents et difficiles à corriger.
* **Temps de traitement long** : le processus de collecte, transmission et encodage des résultats est lent, ce qui retarde la mise à jour du classement.
* **Aucune visualisation en temps réel** : les participants, spectateurs ou autres commissaires n'ont pas accès à un classement à jour pendant la course.
* **Traçabilité limitée** : en cas de litige ou d’erreur, il est difficile de savoir qui a saisi quoi et à quel moment.
* **Dépendance au support papier** : les documents peuvent être perdus, abîmés ou mal interprétés, surtout dans un environnement extérieur et parfois difficile d’accès.

Ces limitations montrent que le système actuel n’est ni fiable, ni efficace, et qu’il ne répond plus aux besoins d’une compétition moderne. Il nuit à la réactivité des organisateurs et à l’expérience globale des participants.  
C’est sur la base de ce constat que le projet Elevate a été imaginé, avec comme but de numériser et automatiser l’ensemble du processus, tout en tenant compte des contraintes spécifiques liées au terrain (problèmes de réseau, simplicité d’utilisation).

### Analyse de l’état désiré

L’application désirée à la fin de mon mandat sera une application PWA avec plusieurs vues pour les différents besoins utilisable avec connexion ou sans connexion au réseau.

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.

Dans un premier temps, quand nous accèderont au site, nous verront le classement actuel basé sur les informations présentes dans la base de données.

On pourra y voir notamment les diverses informations concernant les concurrents des courses comme leurs prénoms et noms de famille, leurs numéros de dossard, leurs nombre de points ainsi que lieu d’habitation.

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.Quand on va cliquer sur un concurrent on va pouvoir avoir accès à ses l’entièreté des résultats que le concurrent à effectuer tout au long des courses que ce soient ses postes terminés ou les malus accumulés. Les résultats et malus seront triés du résultat le plus récent au plus ancien.

Une image contenant texte, capture d’écran, Police, conception

Le contenu généré par l’IA peut être incorrect.La page de login aura pour seul et unique but de permettre au commissaire de s’authentifier pour pouvoir pour ensuite avoir accès aux postes liés au commissaire authentifié.

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.Cette page est disponible quand le commissaire sera alors authentifié. Les postes qui seront liés au commissaire vont alors être affiché sur l’interface. Plusieurs autres informations sont disponibles sur cette page :

Le nombre de requête en attente, cela définit le nombre de résultats et malus que le commissaire a entré en étant hors connexion. Les requêtes sont donc en attente jusqu’à ce que l’utilisateur décide de les envoyer  
  
Il y a aussi une partie malus qui permettra au commissaire de donner des malus aux différents concurrents.

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.

La partie saisie des résultats sera visible une fois qu’un commissaire aura sélectionner un poste qui lui a été attribué. Il arrive alors sur cette page dans laquelle il pourra entrer un résultat pour un concurrent. La page demande un numéro de dossard, une date, ainsi qu’une remarque concernant le résultat.  
La page dédié aux malus est pratiquement la même hors-mis le fait que l’ajout d’un nombre de points est requis lors de la saisie de celui-ci.

### Objectifs

Voici la liste détaillée des objectifs mesurables à atteindre dans le cadre du projet **Elevate**, tels que définis dans le cahier des charges :

**Objectifs fonctionnels**

1. **Afficher le classement en temps réel**
   * Classement trié par total de points décroissants
   * Affichage de la photo, dossard, nom, prénom, code postal, localité et score
   * Mise à jour dynamique sans rechargement de la page
2. **Afficher les résultats détaillés d’un concurrent**
   * Liste de tous les résultats enregistrés (type de poste, nom, date, heure, points)
3. **Afficher les malus d’un concurrent**
   * Remarque visible et liée à la bonne personne avec heure d’enregistrement
4. **Authentifier les commissaires de course**
   * Connexion par email + code PIN
   * Mémoire du login précédent pour simplifier les connexions suivantes
   * Système de session sécurisé
5. **Afficher les postes assignés à chaque commissaire**
   * Filtrage automatique selon l’utilisateur connecté
   * Visualisation simple des postes disponibles
6. **Permettre la saisie d’un résultat par un commissaire**
   * Formulaire de saisie avec dossard, heure, remarque
   * Saisie rapide et efficace, avec formulaire réinitialisé après enregistrement
   * Message d’erreur si le dossard est invalide
7. **Permettre la saisie d’un malus**
   * Champ de remarque obligatoire
   * Enregistrement lié au bon concurrent et au bon poste
8. **Gérer les saisies hors connexion (mode offline)**
   * Stockage local des résultats et malus si pas de réseau
   * Synchronisation automatique ou manuelle dès que le réseau est disponible
   * Indicateur graphique des résultats en attente
9. **Sécuriser les échanges de données**
   * Utilisation de requêtes préparées (protection contre les injections SQL)
   * Hachage des mots de passe avec password\_hash et vérification via password\_verify

**Objectifs techniques**

1. **Développer une PWA (Progressive Web App)**
   * Compatible avec tous les smartphones
   * Fonctionne sans installation
   * Inclut manifest.json et service worker (sw.js)
2. **Créer une API backend sécurisée en PHP**
   * Interface REST sans interface graphique
   * Fonctionne avec une base de données MySQL
   * Hébergement sur serveur local (WAMP) puis sur AlpHosting
3. **Garantir l’accessibilité et la rapidité de l’interface**
   * Interface simple et épurée
   * Grands boutons, navigation rapide, sans scrolling inutile
   * Chargements asynchrones pour optimiser la fluidité

## Choix de variante

Le premier choix que j’ai dû faire concerne le système de connexion que les commissaires vont utiliser pour se connecter à leurs comptes pour qu’ils puissent entrer des résultats dans la base de données.

Le dilemme était de choisir entre un mot de passe ou un code PIN. J’ai choisi le mode de connexion avec le code PIN, qui sera plus utile à entrer lors de la connexion pour chaque personne sur le terrain grâce à sa rapidité d’exécution.

Le deuxième choix concerne l’envoie des résultats qui sont en attentes. La première option était d’envoyer les requêtes directement quand l’utilisateur retrouve la connexion internet sans pour autant qu’il soit l’auteur de cette envoie. La deuxième était de faire en sorte que l’utilisateur puisse les envoyer à tout moment quand il retrouvera la connexion.  
Un popup informera alors l’utilisateur en fonction de l’avancer de sa demande.

Le choix que

## Sécurité de l’information et protection des données

Les données présentes dans la base de données seront protégées de trois façons distinctes :

### Injections SQL

Pour ce projet, j’ai fait attention à la sécurité de ma base de données, notamment pour éviter les injections SQL. C’est un type d’attaque où quelqu’un essaie d’envoyer du code malveillant dans une requête SQL, par exemple via un formulaire ou une URL, pour accéder ou modifier des données sans autorisation.

Pour me protéger contre ça, j’utilise des requêtes préparées avec PDO. Ça permet de séparer le code SQL des données envoyées par l’utilisateur. Du coup, même si quelqu’un essaie d’injecter du code SQL dans un champ, ça ne fonctionnera pas.

Voici un exemple avec une des fonctions que j’ai utilisées :

            $queryPrepared = *$this*->pdo->prepare($query);

            $queryPrepared->execute($params);

Grâce à ça, les paramètres sont automatiquement sécurisés, et il n’y a pas de concaténation de texte dans les requêtes, ce qui évite les failles.

J’ai aussi mis des blocs try/catch pour gérer les erreurs proprement, sans afficher des messages trop techniques à l’utilisateur. En cas de problème, je peux aussi faire un rollback pour annuler une transaction, ce qui aide à garder la base de données dans un état cohérent.

### Attaques XSS

Dans mon projet, j’ai aussi pris en compte les attaques XSS (Cross-Site Scripting). C’est un type d’attaque où un utilisateur malveillant essaie d’injecter du code JavaScript ou HTML dans une page web, souvent via un formulaire ou une URL. Le but, c’est que ce code soit exécuté dans le navigateur des autres utilisateurs, ce qui peut par exemple voler des informations ou modifier le contenu de la page.

Pour éviter ça, j’ai créé une fonction sanitizeInput qui nettoie toutes les données envoyées par les utilisateurs avant de les afficher. Elle fait deux choses importantes :

1. Elle enlève les espaces inutiles au début et à la fin du texte avec trim.
2. Elle transforme tous les caractères spéciaux (comme <, >, " ou ') en entités HTML avec htmlspecialchars.

Du coup, même si quelqu’un essaie d’envoyer un script comme ***<script>alert('XSS') </script>***, il sera transformé en texte inoffensif :  
***&lt;script&gt; alert(&#039;XSS&#039;)&lt;/script&gt;***

Voici la fonction utilisée :

function sanitizeInput($input)

{

    return htmlspecialchars(trim($input), ENT\_QUOTES, 'UTF-8');

}

### Hachage de mot de passe et gestion de session

Pour sécuriser l’authentification dans mon projet, j’ai mis en place deux éléments essentiels : le **hachage des mots de passe** et la **gestion des sessions utilisateur**.

Quand un utilisateur se connecte, je ne compare jamais directement le mot de passe en clair. À la place, tous les mots de passe sont hachés lors de l’enregistrement (grâce à password\_hash) et je les vérifie ensuite avec password\_verify.  
Ça permet de ne jamais stocker de mots de passe lisible dans la base de données. Même si quelqu’un accède à la base, il ne pourra pas voir les vrais mots de passe.

Dans ma fonction de connexion (login), je fais par exemple :

        if (password\_verify($password, $user['PIN'])) {

            //connexion réussi

        }

Une fois le mot de passe vérifié, j’utilise les **sessions PHP** pour garder l’utilisateur connecté de manière sécurisée. Je stocke par exemple son email et son ID dans $\_SESSION, ce qui me permet de l’identifier sur les autres pages sans avoir à redemander ses infos à chaque fois.

J’ai aussi créé un fichier sessionController.php qui centralise la gestion des sessions (comme la création, la vérification de connexion, et la déconnexion). Ça me permet de garder un code plus propre et plus facile à maintenir.

Grâce à cette approche, je protège à la fois les mots de passe des utilisateurs et leur session de navigation. C’est une étape importante pour éviter les vols de données ou les accès non autorisés.

                $\_SESSION['email'] = $email;

                $\_SESSION['id'] = $user['id'];

## Diagramme des cas d’utilisation

### Description du Cas d'Utilisation

Cette application gère le classement de concurrents et permet la consultation et la saisie de résultats.

**Acteurs Principaux (en vert) :**

* Visiteur : Un utilisateur non connecté qui peut consulter les informations publiques de l'application.
* Commissaire connecté : Un utilisateur authentifié ayant des privilèges pour gérer les résultats et les malus.

**Acteurs Secondaires (en jaune) :**

* Serveur : Représente l'ensemble des fichiers PHP et de la base de données qui gèrent la logique métier et le stockage des informations.
* DB (Base de Données) : Le système de stockage persistant des données de l'application (classement, concurrents, résultats, malus, etc.).
* DB navigateur : Représente le stockage local qui pourrait être utilisé pour des fonctionnalités hors ligne.

**Cas d'Utilisation :**

1. Visiteur :
   * Consulter le classement : Le visiteur peut voir le classement actuel des concurrents. Ce classement inclut des informations telles que le nom et prénom des concurrents, leurs numéros de dossard, leurs nombres de points, et leur lieu d'habitation.
   * Voir le détail d'un concurrent : En cliquant sur un concurrent dans le classement, le visiteur accède à une page détaillée affichant l'intégralité des résultats du concurrent choisi, ainsi que d'autres informations spécifiques sur celui-ci.
2. Commissaire connecté :
   * Connexion : Le commissaire doit se connecter à l'application pour accéder à ses fonctionnalités. Cette action implique la création d'une session côté serveur.
   * Entrer des résultats : Le commissaire peut saisir les résultats des concurrents. Cette opération interagit avec le serveur pour effectuer des opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) sur les résultats dans la base de données.
   * Entrer des résultats hors-ligne : Le commissaire a la possibilité de saisir des résultats même sans connexion internet, probablement en utilisant le stockage local du navigateur, qui sera synchronisé avec le serveur une fois la connexion rétablie.
   * Voir postes affectés : Le commissaire peut consulter les postes qui lui sont affectés. Cette fonctionnalité récupère les informations des postes depuis le serveur.
   * Entrer des malus : Le commissaire peut appliquer des malus aux concurrents. Cette action interagit avec le serveur pour effectuer des opérations CRUD sur les malus dans la base de données.
   * Déconnexion : Le commissaire peut se déconnecter de l'application, ce qui entraîne la suppression de sa session côté serveur.

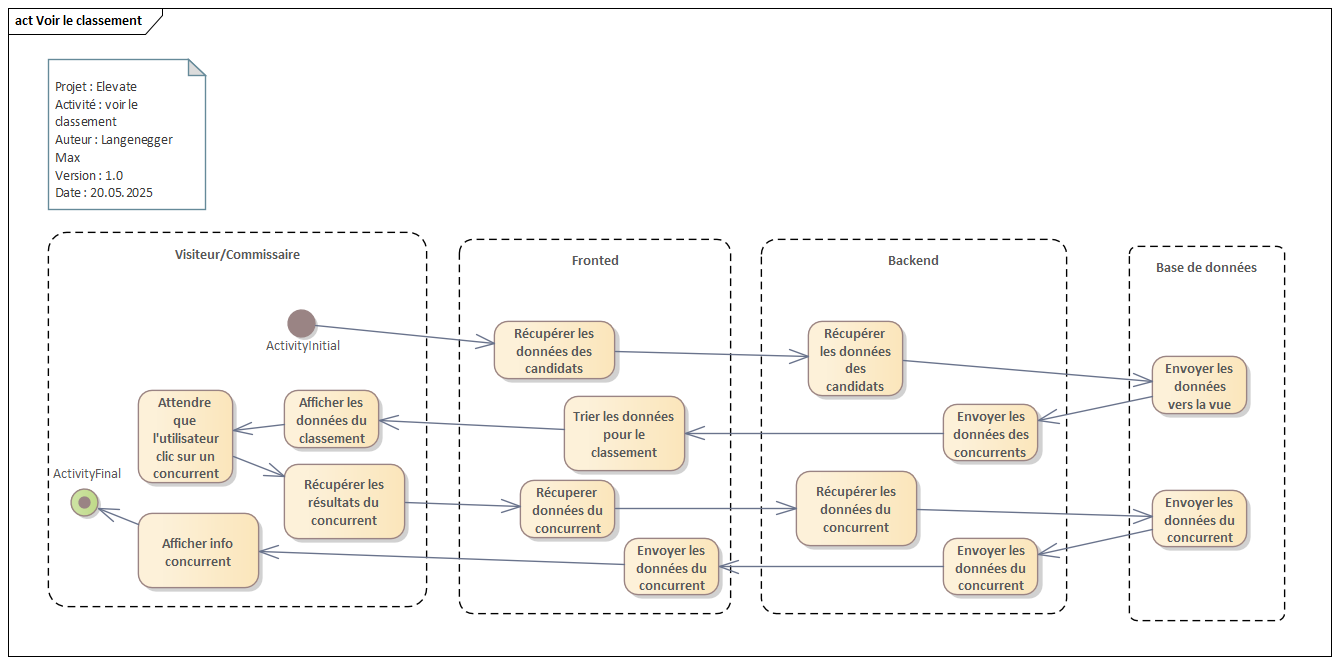
**Interactions Côté Serveur :**

Le Serveur gère les interactions suivantes avec la DB :

* Créer session : Gère l'authentification et la création de sessions utilisateur.
* Récupérer infos candidats : Fournit les données des concurrents pour l'affichage du classement et des détails.
* CRUD Résultat : Effectue les opérations de gestion des résultats des concurrents.
* Supprimer session : Gère la déconnexion des utilisateurs.
* Récupérer infos postes : Fournit les informations sur les postes affectés.
* CRUD Malus : Effectue les opérations de gestion des malus.

Pour qu’un visiteur passe d’un simple utilisateur à un commissaire, il devra se connecter à l’application à l’aide d’un code PIN pour vérifier son identité. Il aura alors accès à d’autre fonctionnalité de l’application comme voir les postes auquel il a été affecté ou encore entrer des résultats des différents concurrents. Le commissaire pourra toujours consulter le classement actuel. Si l’envie le prend, il peut sans autre se déconnecter pour redevenir un visiteur.

## Diagramme d’activité

Ce diagramme décrit le processus de visualisation des résultats d’un concurrent dans l’application Elevate. Il modélise l’interaction entre les différents composants du système : visiteur ou commissaire, frontend, backend et base de données.

Il met en évidence le cheminement des données depuis la base jusqu’à l’affichage dans l’interface utilisateur, que ce soit pour consulter le classement général ou consulter les détails d’un concurrent.

**Déroulement de l’activité :**

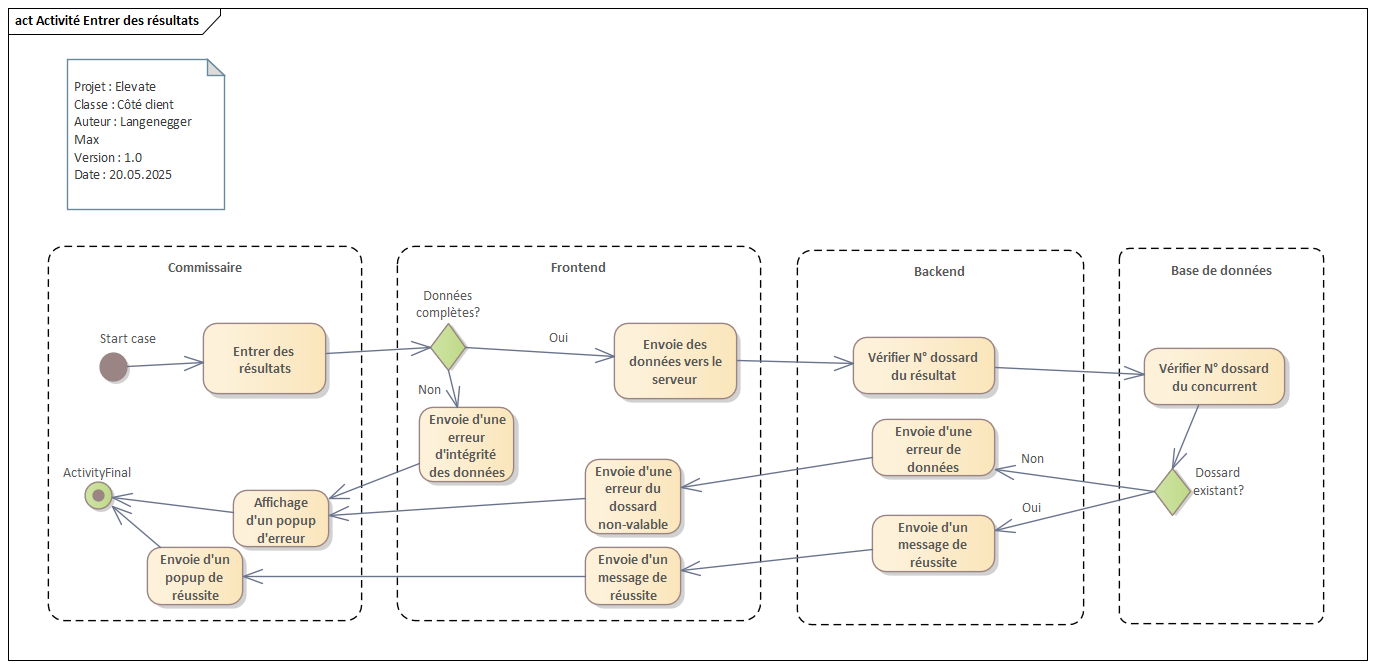
1. **Démarrage**
   * L’activité débute automatiquement lorsqu’un visiteur ou commissaire accède à la page de classement.
2. **Récupération des données du classement**
   * Le frontend envoie une requête au backend pour **récupérer les données de tous les concurrents**.
   * Le backend interroge la base de données et renvoie la liste complète des concurrents avec leurs points.
   * Le frontend trie ces données pour afficher le classement dans l’ordre décroissant.
3. **Affichage du classement**
   * Une fois les données reçues et triées, la liste des concurrents est affichée à l’utilisateur.
4. **Interaction utilisateur**
   * L’utilisateur clique sur un concurrent pour consulter ses résultats détaillés.
   * Cette action déclenche une nouvelle requête du frontend vers le backend pour **récupérer les résultats individuels** de ce concurrent.
5. **Affichage des résultats détaillés**
   * Le backend envoie les données du concurrent, récupérées depuis la base.
   * L’interface frontend les affiche dans une nouvelle vue, dédiée aux résultats détaillés.
6. **Fin du processus**
   * Le processus se termine après l’affichage des informations du concurrent sélectionné.

**Points forts du processus :**

* Interface fluide permettant d’accéder rapidement aux détails de chaque concurrent.
* Séparation claire entre la logique de présentation (frontend) et la gestion des données (backend).

Ce diagramme confirme que le système répond aux attentes d’un accès rapide, intuitif et structuré aux résultats, aussi bien pour les visiteurs que pour les commissaires.

### Entrer des résultats

Ce diagramme modélise le processus de saisie d’un résultat par un commissaire dans l’application Elevate. Il décrit le flux d’exécution entre les différents acteurs du système : le commissaire (utilisateur), l’interface frontend, le backend et la base de données. Il permet de visualiser les différentes étapes, vérifications et cas d’erreur possibles.

**Déroulement de l’activité :**

1. **Départ du processus**
   * Le processus démarre dès que le commissaire accède à la page de saisie et remplit les informations nécessaires.
2. **Étape : Entrer des résultats**
   * Le commissaire saisit le numéro de dossard, la date, l’heure et éventuellement une remarque.
   * L’application vérifie que toutes les données requises ont été fournies.
3. **Vérification côté frontend**
   * Si les données sont incomplètes ou incohérentes, un message d’erreur d’intégrité est affiché.
   * Si les données sont complètes, elles sont envoyées au serveur.
4. **Traitement côté backend**
   * Le backend commence par vérifier le numéro de dossard fourni.
   * Il interroge la base de données pour vérifier si le dossard du concurrent existe.
5. **Réponse de la base de données**
   * Si le dossard n’existe pas, une erreur est renvoyée au frontend, qui affiche une erreur spécifique.
   * Si le dossard est reconnu, les données sont considérées comme valides.
6. **Fin du processus**
   * En cas de succès, un message de confirmation est renvoyé et un popup de réussite est affiché au commissaire.
   * En cas d’échec (erreur de saisie ou dossard invalide), un popup d’erreur est affiché à l’utilisateur.

**Points forts du processus :**

* Vérification précoce côté client pour éviter des appels inutiles au serveur.
* Validation serveur pour garantir l'intégrité des données.
* Gestion claire des cas d’erreur avec retours visuels pour l’utilisateur.
* Utilisation de messages différenciés selon le type d’erreur (intégrité, dossard inconnu, etc.).

Ce diagramme permet d'assurer que le processus de saisie est robuste, user-friendly et conforme aux exigences du système (vérifications, messages, stockage sécurisé).

# Conception

## Architecture

Le projet Elevate a été conçu selon une architecture client-serveur classique, répartie en deux parties principales : le frontend (client) et le backend (serveur). Cette architecture garantit une bonne séparation des responsabilités et facilite la maintenance et l’évolution du système.

**Client**

Le côté client est développé en JavaScript (ES6) et structuré de manière modulaire grâce à des contrôleurs, chacun associé à une vue spécifique. La structure des fichiers est la suivante :

* **js/controllers** : contient les contrôleurs pour chaque fonctionnalité de l’application.
  + classementCtrl.js : gère l’affichage du classement général.
  + indexCtrl.js : gère la page d’accueil et l’enregistrement du service worker.
  + infoConcurrentCtrl.js : affiche les résultats détaillés d’un concurrent.
  + loginCtrl.js : gère la connexion des commissaires.
  + postesCommissaireCtrl.js : affiche les postes associés à un commissaire connecté.
  + saisieMalusCtrl.js : gère la saisie des malus pour un concurrent.
  + saisieResultatsCtrl.js : gère la saisie des résultats pour un concurrent.
* **js/worker** :
  + httpService.js : centralise les appels AJAX au backend et permet de communiquer avec le serveur via des requêtes asynchrones.
* **views** : contient les fichiers HTML correspondant à chaque fonctionnalité ou contrôleur. Chaque vue est associée à un contrôleur pour gérer les interactions.
* **index.html** : page d’accueil principale qui permet d’accéder aux différentes fonctionnalités.
* **sw.js** : fichier du service worker permettant la gestion du mode hors ligne grâce à la mise en cache des fichiers et à IndexedDB.

**Serveur**

Le côté serveur est développé en PHP et organisé selon une architecture MVC simplifiée. La structure des fichiers est la suivante :

* **server.php** : point d’entrée unique du backend, qui reçoit toutes les requêtes entrantes du client et redirige les actions vers les bons contrôleurs.
* **config.php** : contient les constantes de configuration (connexion à la base de données, etc.).
* **controllers** :
  + controller.php : centralise la gestion des actions côté serveur et redirige les requêtes vers les workers appropriés.
  + sessionController.php : gère les sessions utilisateurs, notamment la connexion et la déconnexion des commissaires.
* **workers/database** :
  + WDatabase.php : exécute les requêtes liées aux fonctionnalités métier (ex. : classement, résultats, malus).
  + WDatabaseConnection.php : gère la connexion PDO à la base de données de manière sécurisée et réutilisable.
* **workers** :
  + worker.php : fait le lien entre les requêtes du contrôleur et les classes métiers pour exécuter les opérations demandées.

Cette architecture modulaire et découpée en responsabilités distinctes facilite la compréhension et la maintenance du projet, tout en permettant d’ajouter de nouvelles fonctionnalités de manière simple et efficace. Elle répond parfaitement aux besoins de l’application Elevate, notamment en matière de gestion sécurisée des résultats et des malus, de mode offline et de performance.

## Diagramme de classe

### Client

Ce diagramme représente l'architecture côté client de l'application "Elevate", en séparant les composants en Interface Homme-Machine (IHM), Contrôleurs et Workers.

**Code Couleur :**

* **Jaune (IHM)** : Représente les fichiers HTML, qui sont les interfaces utilisateur directes.
* **Vert (Contrôleur)** : Représente les fichiers JavaScript qui gèrent la logique de présentation et l'interaction avec l'utilisateur. Ils agissent comme des intermédiaires entre l'IHM et les Workers.
* **Bleu (Worker)** : Représente les services ou modules qui gèrent la communication avec le serveur (API) et effectuent des tâches de fond.

**Composants du Diagramme :**

1. **IHM (Interface Homme-Machine)** :
   * index.html : La page d'accueil ou d'entrée de l'application.
   * classement.html : Affiche le classement des concurrents.
   * login.html : La page de connexion.
   * infoConcurrent.html : Affiche les informations détaillées d'un concurrent.
   * postesCommissaire.html : Affiche les postes affectés à un commissaire.
   * saisieResultats.html : Page pour la saisie des résultats.
   * saisieMalus.html : Page pour la saisie des malus.
2. **Contrôleurs** :
   * indexCtrl.js : Contrôleur principal, gère la navigation initiale.
   * classementCtrl.js : Gère la logique d'affichage et d'interaction de la page de classement. Il contient des méthodes pour aller à la page de connexion (gotoLogin()) et charger les données (loadData()).
   * loginCtrl.js : Gère la logique de connexion, incluant la navigation vers le classement (gotoClassement()) et l'appel à la fonction de connexion (login()).
   * infoConcurrentCtrl.js : Gère l'affichage des informations détaillées d'un concurrent et le chargement de ses données (loadData()).
   * postesCommissaireCtrl.js : Gère l'affichage des postes du commissaire, permet de naviguer vers la saisie des résultats (goToSaisieResultat()) et charge les données (loadData()).
   * saisieResultatsCtrl.js : Gère la logique de saisie des résultats, incluant l'envoi des données (postResultat()).
   * saisieMalusCtrl.js : Gère la logique de saisie des malus, incluant l'envoi des données (postMalus()).
3. **Workers** :
   * HttpService : Ce module central agit comme un worker, gérant toutes les communications HTTP avec le serveur (server.php). Il contient des méthodes pour :
     + getInfoConcurrent(idConcurrent) : Récupérer les informations d'un concurrent.
     + getPostesCommissaire(idCommissaire) : Récupérer les postes affectés à un commissaire.
     + getRanking() : Récupérer le classement.
     + login(username, password) : Envoyer les identifiants de connexion.
     + postMalus(dossard, idCommissaire, malus, remarque) : Envoyer les données de malus.
     + postResultat(dossard, idCommissaire, date, remarque) : Envoyer les données de résultats.
4. **Backend (Hors du client)** :
   * server.php : Représente le serveur PHP qui reçoit les requêtes de HttpService et interagit avec la base de données.

**Flux d'Interaction (Exemples) :**

* Un utilisateur accède à classement.html (IHM).
* classementCtrl.js (Contrôleur) est instancié et appelle HttpService.getRanking() (Worker).
* HttpService envoie une requête HTTP à server.php (Backend).
* server.php interagit avec la base de données et renvoie les données à HttpService.
* HttpService transmet les données à classementCtrl.js, qui les affiche dans classement.html.

De même, pour la connexion, login.html (IHM) utilise loginCtrl.js (Contrôleur) qui appelle HttpService.login() (Worker) pour communiquer avec server.php.

### Serveur

Ce diagramme représente l'architecture côté serveur de l'application "Elevate", en séparant les composants en Interface Homme-Machine (IHM), Contrôleurs et Workers, selon le code couleur défini.

**Code Couleur :**

* **Jaune (IHM)** : Représente le point d'entrée principal du serveur, recevant les requêtes du client.
* **Vert (Contrôleur)** : Représente les fichiers PHP qui gèrent la logique de traitement des requêtes, la coordination entre les différents modules et la gestion des sessions.
* **Bleu (Worker)** : Représente les modules qui effectuent les tâches spécifiques, comme l'exécution de la logique métier ou l'interaction directe avec la base de données.

**Composants du Diagramme :**

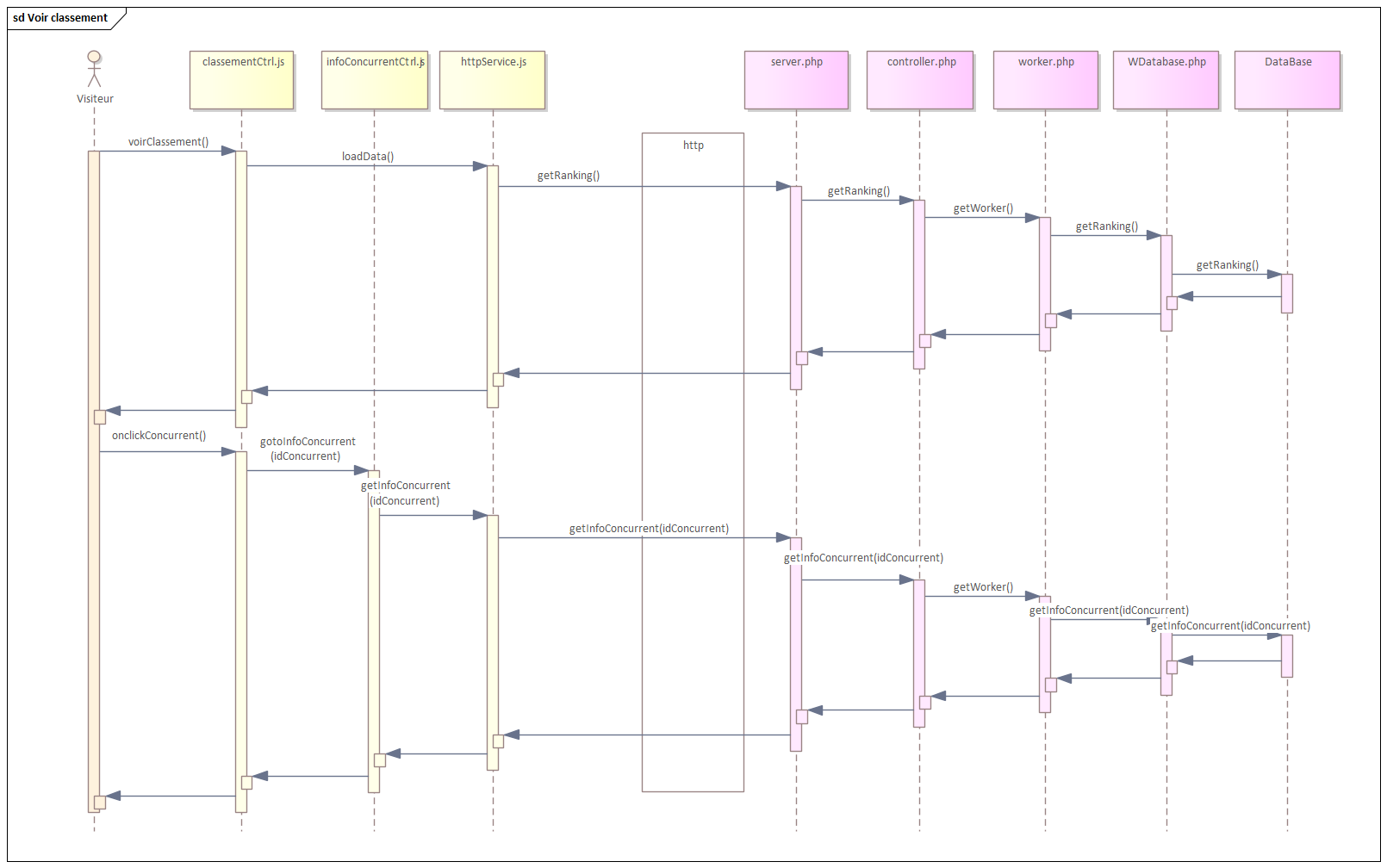
1. **IHM (Point d'entrée Serveur)** :
   * server.php : C'est le point d'entrée principal du serveur. Il reçoit toutes les requêtes HTTP (GET et POST) envoyées par le client (HttpService.js). Il agit comme un routeur initial qui distribue les requêtes aux contrôleurs appropriés.
2. **Contrôleurs** :
   * controller.php : Ce contrôleur semble être le gestionnaire principal des opérations. Il contient un constructeur et une méthode getWorker(), suggérant qu'il est responsable d'instancier et de fournir l'accès aux modules Worker.
   * sessionController.php : Ce contrôleur est spécifiquement dédié à la gestion des sessions utilisateur. Il contient des méthodes pour vérifier l'état de connexion (checkLogin()), déconnecter un utilisateur (disconnect()), et gérer le processus de connexion (login(Semail, $PIN)). Il interagit avec un Worker pour ses opérations.
3. **Workers** :
   * worker.php : Ce module est un worker de haut niveau qui encapsule la logique métier. Il contient des méthodes pour récupérer des informations sur les concurrents (getInfoConcurrent($dossard)), les postes des commissaires (getPostesCommissaire($idComm)), le classement (getRanking()), ainsi que pour poster des malus (postMalusConcurrent()) et des résultats (postResultatConcurrent()). Il dépend de WDatabaseWrk pour les interactions avec la base de données.
   * WDatabase.php : Ce worker est responsable des opérations de base de données de plus haut niveau. Il fournit des méthodes pour interroger et manipuler les données spécifiques à l'application, telles que getInfoConcurrent(idConcurrent), getPostesCommissaire(idCommissaire), getRanking(), login(username, password), postMalus(pointMalus, idConcurrent, remarque), et postResultat(idPoste, date, int, dossard). Il s'appuie sur WDatabaseConnection pour l'exécution réelle des requêtes.
   * WDatabaseConnection.php : Ce worker de bas niveau gère la connexion physique à la base de données et les fonctionnalités transactionnelles. Il offre des méthodes pour ajouter des requêtes à une transaction (addQueryToTransaction()), exécuter des requêtes (executeQuery()), obtenir le dernier ID inséré (getLastId()), gérer les transactions (commitTransaction(), rollbackTransaction(), startTransaction()), et exécuter des requêtes de sélection (selectQuery(), selectSingleQuery()).

**Flux d'Interaction (Exemples) :**

* Une requête GET /getRanking arrive sur server.php (IHM).
* server.php délègue la requête au controller.php (Contrôleur).
* controller.php utilise son getWorker() pour accéder à worker.php (Worker).
* worker.php appelle getRanking() qui, à son tour, utilise WDatabaseWrk (une instance de WDatabase.php) pour récupérer les données.
* WDatabase.php interagit avec WDatabaseConnection.php (Worker) pour exécuter la requête selectQuery() sur la base de données.
* Les données remontent ensuite la chaîne (WDatabaseConnection -> WDatabase -> worker -> controller -> server) pour être renvoyées au client.

De même, pour une requête de connexion, server.php (IHM) redirige vers sessionController.php (Contrôleur), qui utilise un Worker

## Diagramme de séquence

Cette séquence représente le cheminement de l’action pour récupérer le classement depuis le frontend qui commence par classementCtrl.js

## Concept de tests

Tous les tests seront effectués sur plusieurs plateforme pour garantir le bon fonctionnement de l’application et de sa compatibilité avec le plus de navigateur possible.

L’entièreté des tests se fera sur Windows 11 avec Edge et Chrome, sur IPhone avec Safari et sur Android avec Chrome

|  |  |  |  |
| --- | --- | --- | --- |
| **N°** | **Fonctionnalité testée** | **Méthode de test** | **Résultat attendu** |
| 1 | Afficher le classement | Se rendre sur le site [Elevate](https://elevate.emf-infopro-tpi.ch/) et vérifier si les concurrents sont triés par nombre de points | Le classement s'affiche en ordre décroissant de points avec les infos des concurrents. |
| 2 | Afficher les résultats et les malus d’un concurrent | Cliquer sur un concurrent et voir tous les résultats | Les résultats et les malus liés au dossard sélectionné s’affichent correctement. |
| 3 | Rafraîchir le classement | Le classement se rafraichit correctement avec le bouton et dynamiquement toutes les 10 secondes | Le classement se met à jour avec les dernières données sans recharger toute la page. |
| 4 | Se rendre à la page de login | Cliquer sur le bouton de connexion pour arriver à la page de login | L’utilisateur est redirigé vers l’écran de connexion. |
| 5 | Se connecter en tant que commissaire | Entrer les identifiants du commissaire pour arriver à la page suivantes | L’utilisateur est authentifié et redirigé vers la sélection de postes. |
| 6 | Afficher les postes selon le commissaire connecté | Après s’être connecté, l’utilisateur est redirigé vers la page avec tous ces postes | Seuls les postes affectés au commissaire connecté sont visibles. |
| 7 | Entrer un résultat pour un concurrent | Cliquer sur un poste et entrer un numéro de dossard valide, une date et une remarque. | Le résultat est enregistré, et une confirmation s’affiche. Le formulaire est réinitialisé et sélectionne le champ n° de dossard |
| 8 | Entrer un malus pour un concurrent | Cliquer sur Malus et entrer un numéro de dossard valide, une date, une remarque et un nombre de points. | Le malus est enregistré avec la remarque et l’heure, puis visible dans les détails. |
| 9 | Entrer un résultat avec un dossard non-valide | Entrer un résultat avec un n° de dossard non-valide. | L’utilisateur est informé de sa mauvaise saisie du dossard. |
| 10 | Entrer un malus avec un dossard non-valide | Entrer un malus avec un n° de dossard non-valide. | L’utilisateur est informé de sa mauvaise saisie du dossard. |
| 11 | Entrer un email ou code PIN incorrect | Se rendre sur la page de login et saisir un email et code PIN aléatoire. | L’utilisateur est informé de sa mauvaise saisie des informations de login. |
| 12 | Entrer un résultat en étant hors-ligne | Couper la connexion internet et entrer un résultat APRÈS s’être connecté en tant que commissaire | L’utilisateur est informé sur son état lié au réseau et du stockage des résultats hors-ligne |
| 13 | Télécharger l’application Web depuis le navigateur | Sur navigateur, cliquer sur le bouton pour installer l’application Elevate L’application est disponible. Installer Elevate. Sur mobile se rendre sur les paramètres du site et ajouter l’application sur l’écran d’accueil. | L’application est visible et accessible sur le bureau ou sur l’écran d’accueil pour mobile. |

# Réalisation

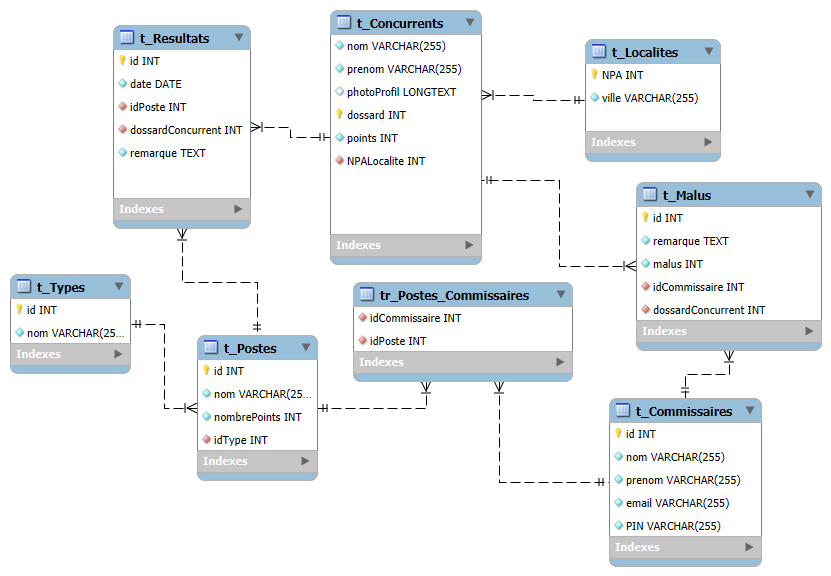
## Déploiement

Le code de mon projet est déployé sur le site <https://elevate.emf-infopro-tpi.ch/>. La partie client est directement accessible via la racine avec tout les fichiers et dossier disponible comme les js ou encore les différentes vues de l’application. Le côté client se trouve sur le même hébergeur mais dans le dossier */serveur/.*

Les logins de connexion de l’application se trouvent en annexes pour se connecter en tant que commissaire.

## Schéma de la base de données

Le schéma relationnel de la base de données a été conçu pour répondre aux besoins fonctionnels du projet **Elevate**, en permettant la gestion des résultats, des malus, des commissaires, des postes et des concurrents. La base est normalisée et respecte les bonnes pratiques de modélisation (clés primaires, relations, types adaptés). Voici les explications des différentes tables :



**t\_Concurrents**

Contient les informations personnelles et sportives de chaque participant :

* nom, prenom, photoProfil : informations de base
* dossard : numéro unique de dossard utilisé pour l'identification
* points : total des points obtenus
* NPALocalite : lien vers la table t\_Localites

**t\_Localites**

Contient les localités liées aux concurrents :

* NPA : numéro postal (clé primaire)
* ville : nom de la localité

**t\_Postes**

Décrit les postes de contrôle utilisés dans la course :

* nom : nom du poste (ex. "Sommet Brâ")
* nombrePoints : nombre de points attribués à ce poste
* idType : type de poste, lien vers t\_Types

**t\_Types**

Liste les types de postes possibles (ex. "Sommet", "Slalom", "Touch", etc.)

**t\_Resultats**

Stocke chaque résultat saisi par les commissaires :

* date : date et heure du résultat
* idPoste : poste concerné
* dossardConcurrent : lien vers le concurrent

**t\_Malus**

Stocke les sanctions (malus) attribuées :

* description : raison du malus
* malus : nombre de points à retirer
* idCommissaire : commissaire ayant infligé le malus
* dossardConcurrent : concurrent concerné

**t\_Commissaires**

Contient les informations des commissaires :

* nom, prenom, email : informations personnelles
* PIN : mot de passe (haché) pour l’authentification

**tr\_Postes\_Commissaires**

Table de liaison entre les commissaires et leurs postes affectés :

* idCommissaire et idPoste forment une clé composée

## Fonctionnement du mode offline (PWA)

Pour garantir le fonctionnement de l’application Elevate même dans des zones sans connexion Internet, j’ai mis en place un mode offline grâce à un service worker. Cette technologie permet de mettre en cache les fichiers essentiels à l’application et d’intercepter les requêtes réseau pour fournir une expérience fluide aux utilisateurs.

Concrètement, deux scripts principaux assurent cette fonctionnalité :

  <script>

    if ('serviceWorker' in navigator) {

      window.addEventListener('load', () => {

        navigator.serviceWorker.register('/sw.js')

          .then(*registration* => {

            console.log('Service Worker enregistré avec succès:', *registration*);

          })

          .catch(*error* => {

            console.log('Erreur lors de l’enregistrement du Service Worker:', *error*);

          });

      });

    }

  </script>

Ce script est chargé au démarrage de l’application. Il vérifie que le navigateur prend en charge les service workers et enregistre le fichier sw.js. Cette étape est indispensable pour activer le cache et permettre l’utilisation hors ligne.

const CACHE\_NAME = 'elevate\_cache';

const FILES\_TO\_CACHE = [

  '/index.html'

//autre fichiers mise en cache

];

//  Installation : cache des fichiers statiques

self.addEventListener('install', *event* => {

*event*.waitUntil(

    caches.open(CACHE\_NAME)

      .then(*cache* => {

        console.log('Mise en cache des fichiers statiques');

        return *cache*.addAll(FILES\_TO\_CACHE);

      })

  );

  self.skipWaiting();

});

Le service worker se charge de mettre en cache les fichiers nécessaires pour faire fonctionner l’application même hors ligne (HTML, CSS, JS, icônes…). Grâce à caches.open(), un cache nommé elevate\_cache est créé et rempli avec la liste des fichiers statiques à mettre en cache. Lors de l’installation (install), ces fichiers sont téléchargés et stockés pour être accessibles ultérieurement.

Lorsqu’un utilisateur accède à l’application et que le service worker est enregistré :

* Tous les fichiers précisés dans FILES\_TO\_CACHE sont enregistrés dans le cache du navigateur.
* Si l’utilisateur perd la connexion réseau, le service worker intercepte les requêtes et fournit les fichiers à partir du cache.
* L’application reste ainsi accessible et fonctionnelle même en mode hors ligne.

Ce mécanisme garantit une expérience utilisateur fluide, même pour les commissaires de course se trouvant dans des zones mal couvertes par le réseau.

## Chargement des données hors-ligne

Pour compléter la gestion du mode offline, j’ai mis en place une stratégie de mise en cache dynamique des requêtes réseau côté client. Cette stratégie est gérée via le service worker, qui intercepte les requêtes HTTP et répond avec les données stockées dans le cache lorsque la connexion réseau n’est pas disponible. Cela permet à l’application de rester partiellement fonctionnelle même en cas de coupure Internet.

Le script suivant, placé dans le fichier sw.js, illustre ce mécanisme :

// Interception des requêtes GET uniquement

self.addEventListener('fetch', *event* => {

  const { request } = *event*;

  // Ne traiter que les requêtes GET

  if (request.method !== 'GET') return;

*event*.respondWith(

    caches.match(request).then(*cached* => {

      if (*cached*) return *cached*;

      return fetch(request).then(*networkResponse* => {

        return caches.open(CACHE\_NAME).then(*cache* => {

*cache*.put(request, *networkResponse*.clone());

          return *networkResponse*;

        });

      }).catch(() => {

        // Optionnel : retourner une page de secours en cas d'échec

        return **new** *Response*("Contenu non disponible hors ligne.", {

          status: 503,

          statusText: "Offline"

        });

      });

    })

  );

});

1. **Interception des requêtes GET**
   * Le service worker écoute l’événement fetch et filtre uniquement les requêtes HTTP de type GET (afin de ne pas interférer avec les requêtes POST, comme l’envoi des résultats ou des malus).
2. **Vérification dans le cache**
   * Si la ressource demandée est déjà présente dans le cache (caches.match(request)), elle est directement renvoyée, assurant ainsi un affichage rapide et une économie de bande passante.
3. **Téléchargement réseau et mise à jour du cache**
   * Si la ressource n’est pas trouvée dans le cache, le service worker essaie de la récupérer sur le réseau (fetch(request)).
   * Une fois la réponse reçue, elle est stockée dans le cache (cache.put(request, networkResponse.clone())) pour les prochaines utilisations hors ligne.
4. **Gestion des erreurs**
   * En cas d’échec du téléchargement réseau (par exemple si la connexion est coupée), une réponse personnalisée est renvoyée avec un message indiquant que le contenu n’est pas disponible hors ligne.

Grâce à cette approche, l’application Elevate propose une expérience utilisateur plus fluide, même en mode offline, et garantit que les commissaires ou visiteurs peuvent continuer à naviguer sur les pages déjà visitées ou consulter les données précédemment chargées.

## Chargement du classement

L’affichage du classement général des concurrents est une fonctionnalité clé de l’application Elevate. Elle permet aux visiteurs et aux commissaires de consulter en temps réel les résultats et les performances des concurrents. Cette fonctionnalité est basée sur un enchaînement clair entre le client et le serveur, garantissant la fluidité et la rapidité de l’affichage.

**Côté client (classementCtrl.js)**

    getRanking(

      async (*data*) => {

*this*.renderClassement(*data*);

      },

      async (*xhr*, *status*, *error*) => {

        console.warn("Erreur AJAX :", *status*, *error*);

        console.warn("Détail complet :", *xhr*.responseText);

      }

    );

 renderClassement(*data*) {

    const listElement = document.querySelector("ul.space-y-2");

    listElement.innerHTML = "";

*data*.sort((*a*, *b*) => *b*.points - *a*.points);

*data*.forEach(*concurrent* => {

      const listItem = document.createElement("li");

      listItem.className = "flex items-center justify-between p-3 bg-white rounded-xl shadow";

      listItem.innerHTML = `

        <div class="flex items-center space-x-3 cursor-pointer concurrent-item" data-dossard="${*concurrent*.dossard}">

          <img src="data:image/png;base64,${*concurrent*.photoProfil}" alt="Photo concurrent" class="w-16 h-16 rounded-full bg-gray-200" />

          <div class="text-base">

            <p class="font-semibold">N° ${*concurrent*.dossard} - ${*concurrent*.prenom} ${*concurrent*.nom}</p>

            <p class="text-gray-500 text-sm">${*concurrent*.NPA} ${*concurrent*.ville}</p>

          </div>

        </div>

        <span class="text-right font-bold text-base">${*concurrent*.points}</span>

      `;

      const clickableDiv = listItem.querySelector(".concurrent-item");

      clickableDiv.addEventListener("click", () => {

*this*.infoConcurrent = **new** InfoConcurrent(clickableDiv.dataset.dossard)

      });

      listElement.appendChild(listItem);

    });

  }

Dans la partie client, la méthode getRanking() est appelée pour récupérer les données du classement. Elle utilise la fonction getRanking() du module httpService.js, qui effectue une requête AJAX de type GET vers l’API du serveur. En cas de succès, la méthode renderClassement() est appelée pour afficher les données reçues ; en cas d’erreur, des messages de debug sont affichés dans la console pour faciliter le diagnostic.

La méthode renderClassement(data) trie les concurrents par ordre décroissant de points et crée dynamiquement des éléments HTML pour chacun d’eux. Chaque concurrent est affiché avec sa photo, son numéro de dossard, son nom, son prénom, son NPA et sa ville, ainsi que son total de points. Une fois cliqué, chaque concurrent ouvre un composant InfoConcurrent qui permet de consulter ses détails.

Le classement est ensuite actualisé toutes les 10 secondes automatiquement grâce à cette ligne de code :

    setInterval(*this*.loadData(), 10000);

**Appel au backend (httpService.js)**

function getRanking(*successCallback*, *errorCallback*) {

  $.ajax({

    type: "GET",

    dataType: "json",

    url: BASE\_URL + "server.php",

    data: "action=getRanking",

    xhrFields: {

      withCredentials: true

    },

    success: *successCallback*,

    error: *errorCallback*,

  });

}

La fonction getRanking(successCallback, errorCallback) utilise jQuery pour envoyer une requête AJAX en GET à l’URL du serveur PHP. Elle ajoute action=getRanking en paramètre pour indiquer au serveur l’action demandée. La fonction gère la réponse JSON côté client et transmet les données au callback successCallback, ou affiche une erreur avec errorCallback.

**Traitement côté serveur (server.php → controller.php → worker.php → WDatabase.php)**

1. **server.php** : point d’entrée unique du serveur qui reçoit l’action demandée (action=getRanking) et la transmet au contrôleur approprié.
2. **controller.php** : récupère l’action et délègue le traitement au worker correspondant.
3. **worker.php** : centralise les appels aux fonctions métiers et fait appel à la méthode getRanking() du module WDatabase.php.
4. **WDatabase.php** : exécute la requête SQL pour récupérer la liste complète des concurrents en les joignant avec leurs localités. La requête sélectionne les noms, prénoms, dossards, emails, photos de profil, points et localités. Les résultats sont transformés en tableau associatif et renvoyés au contrôleur.

public function getRanking(): array

    {

        try {

            $QUERY = "SELECT

            c.nom,

            c.prenom,

            c.dossard,

            c.email,

            c.photoProfil,

            c.points,

            l.NPA,

            l.ville

        FROM

            t\_Concurrents AS c

        JOIN

            t\_Localites AS l ON c.idLocalite = l.id;

        ";

            $rows = *WDatabaseConnection*::getInstance()->selectQuery($QUERY, []);

            $result = [];

            foreach ($rows as $row) {

                $result[] = [

                    'nom' => $row['nom'],

                    'prenom' => $row['prenom'],

                    'dossard' => $row['dossard'],

                    'email' => $row['email'],

                    'photoProfil' => $row['photoProfil'],

                    'points' => $row['points'],

                    'NPA' => $row['NPA'],

                    'ville' => $row['ville'],

                ];

            }

            return $result;

        } catch (*Exception* $e) {

            throw new *Exception*("Erreur lors de la récupération du classement : " . $e->getMessage());

        }

    }

La méthode getRanking() du module WDatabase.php utilise une requête SQL optimisée avec une jointure entre la table t\_Concurrents et la table t\_Localites. Elle trie et prépare les données au format JSON pour les renvoyer au client.

## Récupérer les informations du concurrent

Cette fonctionnalité permet aux commissaires de consulter les résultats et malus associés à un concurrent donné en cliquant sur son numéro de dossard. L’application récupère les informations depuis le serveur et les affiche de manière lisible et triée par date.

Lorsqu’un utilisateur clique sur un concurrent, la méthode getInfoConcurrent() est appelée avec le numéro de dossard. Cette méthode récupère les résultats et les malus via une requête AJAX et fusionne les données dans un tableau unique nommé lignes. Chaque élément contient le type (résultat ou malus), la date, la remarque éventuelle, le label (nom du poste ou "Malus") et le nombre de points (positif ou négatif). Les données sont ensuite triées par date décroissante.

Enfin, le tableau est transformé en code HTML et inséré dynamiquement dans la page, dans un conteneur dédié. Si aucun résultat ou malus n’est disponible, un message d’information est affiché. Cette méthode assure une visualisation claire et lisible pour l’utilisateur.

    getInfoConcurrent(*dossard*, (*data*) => {

        if (!*data* || (!*data*.resultat && !*data*.malus)) return;

*const* lignes = [];

        // Fusionner résultats et malus dans un seul tableau

        if (*data*.resultat) {

*data*.resultat.forEach(*item* => {

                lignes.push({

                    type: 'resultat',

                    date: *item*.date,

                    remarque: *item*.remarque || '',

                    label: `${*item*.nomTypePoste} - ${*item*.nomPoste}`,

                    points: parseInt(*item*.nombrePoints, 10) || 0

                });

            });

        }

        if (*data*.malus) {

*data*.malus.forEach(*item* => {

                lignes.push({

                    type: 'malus',

                    date: *item*.date,

                    remarque: *item*.descriptionMalus || '',

                    label: 'Malus',

                    points: -Math.abs(parseInt(*item*.pointsMalus, 10)) || 0

                });

            });

        }

        // Tri par date décroissante

*lignes*.*sort*((*a*, *b*) => *new* *Date*(*b*.*date*) - *new* *Date*(*a*.*date*));

        // Génération du HTML

*const* resultatsHTML = lignes.length === 0

            ? `<li class="text-center text-gray-500 italic">Aucun résultat disponible.</li>`

            : *lignes*.*map*(*item* => {

                const dateObj = **new** *Date*(*item*.date);

                const dateStr = dateObj.toLocaleDateString('fr-CH');

                const heureStr = dateObj.toLocaleTimeString('fr-CH', { hour: '2-digit', minute: '2-digit' });

                const pointsColor = *item*.points > 0 ? 'text-blue-600' :

*item*.points < 0 ? 'text-red-600' :

                                    'text-gray-500';

                const pointsStr = `${*item*.points > 0 ? '+' : ''}${*item*.points} pts`;

                const remarqueHTML = *item*.remarque

                    ? `<p class="italic text-gray-600">Remarque : ${*item*.remarque}</p>`

                    : '';

                return `

                    <li class="mb-3 border-b pb-2">

                        <p><strong>${*item*.label}</strong></p>

                        <p>${dateStr} - ${heureStr}</p>

                        ${remarqueHTML}

                        <p class="${pointsColor} font-semibold">${pointsStr}</p>

                    </li>

                `;

            }).*join*('');

        // Affichage des résultats

        document.*querySelector*('#resultContainer').innerHTML = `

            <section class="bg-white rounded-lg shadow mt-4 p-4 mx-2 overflow-y-auto">

                <ul class="list-none p-0 m-0">

                    ${resultatsHTML}

                </ul>

            </section>

        `;

    });

## Envoie des requêtes en attentes

Pour garantir la robustesse de l’application Elevate même en cas de perte de connexion réseau, j’ai implémenté un système de stockage des requêtes en attente. Lorsqu’un commissaire se trouve hors ligne et souhaite enregistrer un résultat, la requête est stockée localement grâce à IndexedDB. Cette requête sera ensuite envoyée automatiquement ou manuellement une fois la connexion rétablie.

Lorsqu’un commissaire clique sur le bouton "postResultat", le script suivant est exécuté :

            $("#postResultat").on("click", async () => {

                const numeroDossard = $("#dossard").val();

                const dateHeure = $("#datetime").val();

                const estValide = !isNaN(*Date*.parse(dateHeure.replace(" ", "T")));

                if (!estValide) {

                    alert("La date et heure est invalide.");

                    return;

                }

                const remarques = $("#remarques").val();

                const idCommissaire = localStorage.getItem("id");

                const data = {

                    idPoste,

                    numeroDossard,

                    dateHeure,

                    remarques,

                    idCommissaire

                };

                if (navigator.onLine) {

                    postResultatConcurrent(*idPoste*, numeroDossard, dateHeure, remarques, idCommissaire,

                        () => {

                            Swal.fire({

                                title: "Résultat enregistré !",

                                text: "Le concurrent N° " +numeroDossard+ " a recu un résultat",

                                icon: "success"

                              });

                            $('input[type="text"]').val('');

                            $('#dossard').val('');

                            $('#remarques').val('');

                            $('#dossard').focus();

                        },

                        (*error*) => {

                            alert("Erreur lors de l'ajout du résultat: " + *error*.responseText);

                        }

                    );

                } else {

                    console.log("Hors ligne : stockage local du résultat");

                    await *this*.storeOfflineResult(data);

                    Swal.fire({

                        title: "Vous êtes hors connexion !",

                        text: "Les requêtes sont mise en attente",

                        icon: "warning"

                      });

                    $('input[type="text"]').val('');

                    $('#dossard').val('');

                    $('#remarques').val('');

                    $('#dossard').focus();

                }

            });

        });

    }

En mode hors ligne, les requêtes sont stockées localement grâce à la fonction storeOfflineResult(data) :

    async storeOfflineResult(*data*) {

        const db = await **new** PostesCommissaireController().openPostDB();

        const tx = db.transaction('post-requests', 'readwrite');

        const store = tx.objectStore('post-requests');

        await store.add({

            functionName: "postResultatConcurrent",

            args: [

*data*.idPoste,

*data*.numeroDossard,

*data*.dateHeure,

*data*.remarques,

*data*.idCommissaire

            ]

        });

    }

Cette fonction utilise IndexedDB pour stocker la requête (avec son nom de fonction et ses arguments) dans une object store appelée post-requests. Ainsi, même en l’absence de connexion Internet, l’application conserve les informations nécessaires à l’envoi ultérieur des résultats.

            $("#sync-posts-btn").on("click", async () => {

                await *this*.flushPostRequests();

            });

Ensuite, pour gérer l’envoi des requêtes stockées localement lorsque l’utilisateur est hors ligne, j’ai implémenté une méthode de synchronisation manuelle qui permet aux commissaires de transmettre les résultats dès qu’une connexion Internet est disponible. Cette fonctionnalité est accessible via un bouton « Synchroniser les résultats », qui déclenche la méthode flushPostRequests().

 async flushPostRequests() {

        if (!navigator.onLine) {

            Swal.fire({

                title: "Vous êtes hors ligne.",

                text: "Veuillez vous reconnecter pour envoyer les requêtes.",

                icon: "warning"

              });

            return;

        }

        const db = await *this*.openPostDB();

        const tx = db.transaction('post-requests', 'readwrite');

        const store = tx.objectStore('post-requests');

        const allRequests = await **new** *Promise*((*resolve*, *reject*) => {

            const req = store.getAll();

            req.onsuccess = () => resolve(req.result);

            req.onerror = () => reject(req.error);

        });

        for (const req of allRequests) {

            try {

                const { functionName, args } = req;

                // Vérifie si la fonction existe

                if (typeof window[functionName] === "function") {

                    await **new** *Promise*((*resolve*, *reject*) => {

                        window[functionName](...args,

                            (*data*) => resolve(*data*),

                            (*error*) => reject(*error*)

                        );

                    });

                } else {

                    console.warn(`Fonction ${functionName} inconnue`);

                }

            } catch (err) {

                console.error('Erreur envoi requête différée:', err);

                continue;

            }

        }

        // Nettoyage

        const clearTx = db.transaction('post-requests', 'readwrite');

        await **new** *Promise*((*resolve*, *reject*) => {

            const clearReq = clearTx.objectStore('post-requests').clear();

            clearReq.onsuccess = () => resolve();

            clearReq.onerror = () => reject(clearReq.error);

        });

        await *this*.updateButtonText();

        alert("Toutes les requêtes en attente ont été envoyées.");

    }

Lors de la synchronisation, l’application vérifie d’abord si l’utilisateur est connecté à Internet. Si ce n’est pas le cas, un message d’alerte est affiché pour indiquer qu’il faut rétablir la connexion avant de poursuivre. Si la connexion est active, l’application ouvre la base IndexedDB et récupère toutes les requêtes précédemment mises en attente dans l’object store post-requests. Chaque requête contient le nom de la fonction à appeler (par exemple postResultatConcurrent) et ses arguments (comme l’ID du poste, le numéro de dossard, la date et l’heure, les remarques et l’ID du commissaire).

Ensuite, pour chaque requête récupérée, l’application vérifie si la fonction correspondante existe dans l’espace global (window). Si c’est le cas, elle appelle dynamiquement cette fonction en lui transmettant les arguments et en gérant les réponses asynchrones via des promesses. Cette étape permet de relancer automatiquement l’envoi des résultats différés au serveur.

Une fois toutes les requêtes traitées, l’objet store post-requests est vidé afin d’éviter les doublons lors de la prochaine synchronisation. Enfin, un message de confirmation est affiché à l’utilisateur pour indiquer que toutes les requêtes en attente ont bien été envoyées.

Grâce à cette synchronisation manuelle, les commissaires peuvent saisir les résultats même hors ligne, puis les envoyer plus tard en un seul clic dès qu’une connexion est disponible. Cette fonctionnalité renforce la fiabilité et la robustesse de l’application Elevate, en permettant de gérer les coupures réseau de manière transparente pour l’utilisateur.

## Connexion utilisateur

La fonctionnalité de connexion utilisateur a été implémentée pour authentifier les commissaires et leur permettre d’accéder aux fonctionnalités sécurisées de l’application. Cette étape est essentielle pour garantir que seuls les utilisateurs autorisés peuvent saisir des résultats ou des malus.

**Fonctionnement côté client**

L’utilisateur saisit son email et son mot de passe dans le formulaire de connexion. Lorsque l’utilisateur clique sur le bouton de connexion, la méthode loginCommissaire() est appelée côté client. Cette méthode récupère les valeurs des champs de saisie et appelle la fonction login() en transmettant l’email et le mot de passe, ainsi que les callbacks de succès et d’erreur.

    loginCommissaire() {

        const username = $("#email").val();

        const password = $("#password").val();

        login(username, password,

*gestion des résultats*

        );

    }

En cas de succès, les informations de session (email et id) sont enregistrées localement via localStorage pour simplifier l’accès aux données utilisateur durant la session. Un popup SweetAlert affiche un message de confirmation, et l’utilisateur est redirigé vers la sélection des postes. En cas d’erreur (mauvais identifiants ou problème réseau), un message d’alerte est affiché à l’utilisateur.

                    localStorage.setItem("email", *data*.email);

                    localStorage.setItem("id", *data*.id);

                    localStorage.setItem("isLogged", true);

**Appel AJAX**

La fonction login() utilise jQuery pour effectuer un appel AJAX de type POST vers le serveur (server.php), en précisant l’action à effectuer (action=login), ainsi que les identifiants saisis par l’utilisateur. Les callbacks success et error permettent de gérer la réponse du serveur ou les éventuelles erreurs réseau.

function login(*login*, *mdp*, *successCallback*, *errorCallback*) {

  $.ajax({

    type: "POST",

    dataType: "json",

    url: BASE\_URL + "server.php",

    data: {

      action: "login",

      email: *login*,

      password: *mdp*

    },

    xhrFields: {

      withCredentials: true

    },

    success: *successCallback*,

    error: *errorCallback*

  });

}

**Traitement côté serveur**

Dans server.php, l’action login est traitée par la vérification des paramètres attendus (email et password). Si les paramètres sont présents, ils sont d’abord nettoyés via la fonction sanitizeInput(), puis transmis à la méthode login() du sessionController.

                    case 'login':

                        if (isset($\_POST['email'], $\_POST['password'])) {

                            $username = sanitizeInput($\_POST['email']);

                            $password = sanitizeInput($\_POST['password']);

                            echo $session->login($username, $password);

                            http\_response\_code(200);

                        } else {

                            handleErrorResponse("Paramètres manquants pour la connexion.");

                        }

                        break;

La méthode login() vérifie la présence de l’email dans la base de données des commissaires. Si l’utilisateur existe, le mot de passe est vérifié grâce à password\_verify(). Si la vérification est réussie, une session est ouverte avec les variables $\_SESSION['email'] et $\_SESSION['id']. Le serveur renvoie ensuite une réponse JSON avec un attribut result: true pour indiquer le succès de la connexion, accompagné de l’email et de l’id du commissaire.

    public function login($email, $password)

    {

        $request = "SELECT \* FROM t\_Commissaires WHERE email = :email";

        $params = array('email' => $email);

        $user = *$this*->worker->getDataBaseWrk()->selectSingleQuery($request, $params);

        if (isset($user['email'])) {

            if (password\_verify($password, $user['PIN'])) {

                $\_SESSION['email'] = $email;

                $\_SESSION['id'] = $user['id'];

                return '{"result": true, "email": "' . $user['email'] . '"' . ',"id":"' . $user['id'] . '"}';

            }

        }

        return '{"result": false}';

    }

Si la vérification échoue ou si l’utilisateur n’existe pas, le serveur renvoie un objet JSON avec result: false pour indiquer une erreur de connexion.

## Gestion des erreurs

Dans l’application Elevate, la gestion des erreurs est un aspect essentiel pour garantir la fiabilité du système et offrir une expérience utilisateur fluide. Elle est implémentée à différents niveaux du projet, aussi bien côté serveur (PHP) que côté client (JavaScript).

**Côté serveur**

Au niveau du serveur, les erreurs sont gérées principalement dans les fichiers **server.php**, **controller.php** et **worker.php**. Par exemple, lors de la réception des requêtes POST pour enregistrer un résultat (postResultatConcurrent) ou un malus (postMalusConcurrent), le serveur vérifie d’abord si la session utilisateur est active. Si ce n’est pas le cas, une réponse JSON avec un message d’erreur approprié est renvoyée et le code de réponse HTTP est défini à 401 (non autorisé).

                        if (!isset($\_SESSION['email'])) {

                            handleErrorResponse("Utilisateur non authentifié.");

                            http\_response\_code(401);

                            break;

                        }

De plus, avant d’exécuter une opération critique, le serveur s’assure de la présence de tous les paramètres attendus. En cas de paramètres manquants, un message d’erreur clair est renvoyé au client :

                        if (isset($\_POST['dossard'], $\_POST['date'], $\_POST['remarque'], $\_POST['idCommissaire'], $\_POST['nombrePoints'])) {

...

                        } else {

                            handleErrorResponse("Paramètres manquants pour la connexion.");

                        }

                        break;

Enfin, lors des opérations sur la base de données (ex. : récupération du classement via getRanking()), un bloc try/catch est utilisé pour capturer les erreurs et renvoyer un message explicite :

    public function getRanking(): array

    {

        try {

...

        } catch (*Exception* $e) {

            throw new *Exception*("Erreur lors de la récupération du classement : " . $e->getMessage());

        }

    }

Sur le client, la gestion des erreurs est assurée principalement avec **AJAX** et les callbacks de réponse. Lors d’un appel réseau (comme pour la connexion d’un utilisateur), un callback d’erreur est utilisé pour afficher une alerte ou un message dans la console :

        login(username, password,

            (*data*) => {

                if (*data*.result === true) {

...

                } else {

                    Swal.fire({

                        icon: "error",

                        title: "Oops...",

                        text: "Code PIN ou email incorrect !",

                      });

                }

            },

            (*error*) => {

                console.error("Erreur AJAX :", *error*);

                if (*error*.responseText) {

                    console.error("Réponse brute :", *error*.responseText);

                }

                alert("Erreur réseau ou problème lors de la connexion.");

            }

        );

Cela permet à l’utilisateur d’être informé rapidement d’un problème (ex. : réseau ou mauvaise authentification). Dans certains cas, des popups SweetAlert sont utilisés pour afficher des messages d’erreur plus conviviaux (par exemple en cas de mauvais code PIN).

**Conclusion**

La gestion des erreurs est ainsi centralisée et standardisée dans l’ensemble du projet. Les erreurs sont gérées à la fois côté serveur (avec des vérifications de session, des vérifications des paramètres, et des messages explicites) et côté client (avec des callbacks AJAX et des messages d’alerte). Cette approche garantit une meilleure robustesse de l’application et permet à l’utilisateur d’identifier rapidement la cause d’un problème éventuel.

# Test

## Procédure de test

|  |  |  |  |
| --- | --- | --- | --- |
| **N°** | **Fonctionnalité testée** | **Description du test** | **Résultat attendu** |
| 1 | Afficher le classement | Se rendre sur le site [Elevate](https://elevate.emf-infopro-tpi.ch/) et vérifier si les concurrents sont triés par nombre de points | Le classement s'affiche en ordre décroissant de points avec les infos des concurrents. |
| 2 | Afficher les résultats et les malus d’un concurrent | Cliquer sur un concurrent et voir tous les résultats | Les résultats et les malus liés au dossard sélectionné s’affichent correctement. |
| 3 | Rafraîchir le classement | Le classement se rafraichit correctement avec le bouton et dynamiquement toutes les 10 secondes | Le classement se met à jour avec les dernières données sans recharger toute la page. |
| 4 | Se rendre à la page de login | Cliquer sur le bouton de connexion pour arriver à la page de login | L’utilisateur est redirigé vers l’écran de connexion. |
| 5 | Se connecter en tant que commissaire | Entrer les identifiants du commissaire pour arriver à la page suivantes | L’utilisateur est authentifié et redirigé vers la sélection de postes. |
| 6 | Afficher les postes selon le commissaire connecté | Après s’être connecté, l’utilisateur est redirigé vers la page avec tous ces postes | Seuls les postes affectés au commissaire connecté sont visibles. |
| 7 | Entrer un résultat pour un concurrent | Cliquer sur un poste et entrer un numéro de dossard valide, une date et une remarque. | Le résultat est enregistré, et une confirmation s’affiche. Le formulaire est réinitialisé et sélectionne le champ n° de dossard |
| 8 | Entrer un malus pour un concurrent | Cliquer sur Malus et entrer un numéro de dossard valide, une date, une remarque et un nombre de points. | Le malus est enregistré avec la remarque et l’heure, puis visible dans les détails. |
| 9 | Entrer un résultat avec un dossard non-valide | Entrer un résultat avec un n° de dossard non-valide. | L’utilisateur est informé de sa mauvaise saisie du dossard. |
| 10 | Entrer un malus avec un dossard non-valide | Entrer un malus avec un n° de dossard non-valide. | L’utilisateur est informé de sa mauvaise saisie du dossard. |
| 11 | Entrer un email ou code PIN incorrect | Se rendre sur la page de login et saisir un email et code PIN aléatoire. | L’utilisateur est informé de sa mauvaise saisie des informations de login. |
| 12 | Entrer un résultat en étant hors-ligne | Couper la connexion internet et entrer un résultat APRÈS s’être connecté en tant que commissaire | L’utilisateur est informé sur son état lié au réseau et du stockage des résultats hors-ligne |
| 13 | Télécharger l’application Web depuis le navigateur | Sur navigateur, cliquer sur le bouton pour installer l’application Elevate L’application est disponible. Installer Elevate. Sur mobile se rendre sur les paramètres du site et ajouter l’application sur l’écran d’accueil. | L’application est visible et accessible sur le bureau ou sur l’écran d’accueil pour mobile. |

La procédure de test a été élaborée pour vérifier que toutes les fonctionnalités principales de l’application Elevate répondent aux exigences spécifiées dans le cahier des charges. Chaque fonctionnalité a été testée de manière manuelle à l’aide d’un navigateur web, en simulant les actions d’un commissaire ou d’un visiteur. Les tests ont été effectués sur la plateforme Elevate en ligne, ainsi qu’en mode hors-ligne pour valider la gestion des résultats différés. Pour chaque fonctionnalité, la méthode de test, le résultat attendu et le résultat obtenu ont été consignés dans un tableau récapitulatif afin de garantir la traçabilité et la validation des objectifs du projet. Tous les tests ont été effectués sur PC (Chrome et Edge) ainsi que sur IOS et Android.

## Protocol de test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr. | Fonctionnalité testée | Résultat attendu | Résultat final | Visa |
| 1 | Afficher le classement | Le classement s'affiche en ordre décroissant de points avec les infos des concurrents. | Conforme | OK |
| 2 | Afficher les résultats et les malus d’un concurrent | Les résultats et les malus liés au dossard sélectionné s’affichent correctement. | Conforme | OK |
| 3 | Rafraîchir le classement | Le classement se met à jour avec les dernières données sans recharger toute la page. | Conforme | OK |
| 4 | Se rendre à la page de login | L’utilisateur est redirigé vers l’écran de connexion. | Conforme | OK |
| 5 | Se connecter en tant que commissaire | L’utilisateur est authentifié et redirigé vers la sélection de postes. | Conforme | OK |
| 6 | Afficher les postes selon le commissaire connecté | Seuls les postes affectés au commissaire connecté sont visibles. | Conforme | OK |
| 7 | Entrer un résultat pour un concurrent | Le résultat est enregistré, et une confirmation s’affiche. Le formulaire est réinitialisé et sélectionne le champ n° de dossard. | Conforme | OK |
| 8 | Entrer un malus pour un concurrent | Le malus est enregistré avec la remarque et l’heure, puis visible dans les détails. | Conforme | OK |
| 9 | Entrer un résultat avec un dossard non-valide | L’utilisateur est informé de sa mauvaise saisie du dossard. | Conforme | OK |
| 10 | Entrer un malus avec un dossard non-valide | L’utilisateur est informé de sa mauvaise saisie du dossard. | Conforme | OK |
| 11 | Entrer un email ou code PIN incorrect | L’utilisateur est informé de sa mauvaise saisie des informations de login. | Conforme | OK |
| 12 | Entrer un résultat en étant hors-ligne | L’utilisateur est informé sur son état lié au réseau et du stockage des résultats hors-ligne. | Conforme | OK |
| 13 | Télécharger l’application Web depuis le navigateur | L’application est visible et accessible sur le bureau ou sur l’écran d’accueil pour mobile. | Conforme | OK |

## Signature du protocole de test

|  |  |  |
| --- | --- | --- |
| Date | Nom | Signature |
| 30.05.2025 | Langenegger Max |  |

# Tests Techno

## Test PWA

Tests technologiques pour la mise en place d'une application PWA de base. Ce test permettra de créer un site avec un manifest permettant l’installation du site en tant qu’application PWA basique sans aucune autre fonctionnalité.  
  
J’ai commencé par créer un simple fichier html qui possède un lien vers un manifest ainsi qu’un texte *Hello World PWA*.

<html>

<head>

    <link *rel*="manifest" *href*="manifest.json" />

</head>

<body>

Hello World PWA

</body>

</html>

J’ai aussi créé le manifest permettant l’installation de l’application avec différents paramètres tels que le nom de l’application, l’url de l’application et les icones qui seront affiché sur l’application

{

    "name": "Test PWA",

    "short\_name": "PWA",

    "description": "Ce site est un site de test pour une PWA",

    "start\_url": "/testTechno/PWASimple",

    "display": "standalone",

    "background\_color": "#ffffff",

    "theme\_color": "#a70d0d",

    "icons": [

        {

            "src": "bob\_192x192.png",

            "sizes": "192x192",

            "type": "image/png"

        },

        {

            "src": "bob\_512x512.png",

            "sizes": "512x512",

            "type": "image/png"

        }

    ]

}

Lors de l‘accès au [site](https://elevate.emf-infopro-tpi.ch/testTechno/PWASimple/), celui-ci nous demande bien si l’on veut installer le site en tant qu’application.

Une image contenant texte, capture d’écran, Police, logo

Le contenu généré par l’IA peut être incorrect.Une image contenant texte, capture d’écran, Système d’exploitation, Page web

Le contenu généré par l’IA peut être incorrect.

L’application est fonctionnelle et téléchargeable sur navigateur ET aussi téléphone mobile.

Une image contenant texte, capture d’écran, dessin humoristique, graphisme

Le contenu généré par l’IA peut être incorrect.

## PWA mode hors-ligne

Tests technologiques pour faire fonctionner une application PWA en mode hors ligne (cache des fichiers). Il faut ajouter un serviceWorker qui va gérer les fichiers que le navigateur va mettre en cache. Cela permet de rendre l’application disponible même hors réseau.

Pour permettre l’ajout et l’utilisation d’un serviceWorker, j’ai d’abord créé un lien entre le fichier html et le fichier sw.js. J’ai ajouté un bout de JavaScript dans le html qui va vérifier que le navigateur supporte les serviceWorker et enregistre sw.js si le navigateur est correct.

    <script *type*="text/javascript" *src*="sw.js"></script>

    <script>

        if ('serviceWorker' in navigator) {

      window.addEventListener('load', () => {

        navigator.serviceWorker.register('/testTechno/PWAHorsLigne/sw.js')

          .then(*registration* => {

            console.log('Service Worker enregistré avec succès:', *registration*);

          })

          .catch(*error* => {

            console.log('Erreur lors de l’enregistrement du Service Worker:', *error*);

          });

      });

    }

    </script>

*sw.js*

const CACHE\_NAME = 'v1';

const FILES\_TO\_CACHE = [

  '/favicon.ico',

  '/testTechno/PWAHorsLigne/',

  '/testTechno/PWAHorsLigne/index.html',

  '/testTechno/PWAHorsLigne/sw.js',

  '/testTechno/PWAHorsLigne/logo/bob\_192x192.png',

  '/testTechno/PWAHorsLigne/logo/bob\_512x512.png',

  '/testTechno/PWAHorsLigne/parapente.jpg',

  '/testTechno/PWAHorsLigne/manifest.json'

];

self.addEventListener('install', *event* => {

*event*.waitUntil(

    caches.open(CACHE\_NAME)

      .then(*cache* => {

        console.log('Mise en cache des fichiers');

        return *cache*.addAll(FILES\_TO\_CACHE);

      })

  );

});

self.addEventListener('activate', *event* => {

*event*.waitUntil(

    caches.keys().then(*keys* =>

*Promise*.all(

*keys*.filter(*key* => *key* !== CACHE\_NAME)

            .map(*key* => caches.delete(*key*))

      )

    )

  );

});

self.addEventListener('fetch', *event* => {

*event*.respondWith(

    caches.match(*event*.request)

      .then(*response* => *response* || fetch(*event*.request))

  );

});

Maintenant l’application est disponible et visible même lorsque la connexion est interrompue.

Dans un premier temps sur navigateur, la page affiche que nous sommes hors connexion.

Une image contenant dessin humoristique, smiley, sourire, émoticône

Le contenu généré par l’IA peut être incorrect.

Puis la page et les fichiers présent dans le cache s’affiche correctement et tous les fichiers sont bien présents dans le cache du navigateur :

Une image contenant capture d’écran, texte, Logiciel multimédia, Logiciel de graphisme

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, nombre, Police, ligne

Le contenu généré par l’IA peut être incorrect.

## PWA statuts réseau

Tests technologiques pour établir si une connexion réseau est active ou non dans une application PWA. Analyser les moyens possibles pour savoir si une connexion est perdue ou rétablie.

Pour effectuer ce test j’ai ajouté un fichier *main.js* qui permet de vérifier l’état actuel du réseau pour ensuite afficher un message différent en fonction de celui-ci

$(document).ready(function () {

    document.getElementById('status').innerHTML = navigator.onLine ? 'online' : 'offline';

    var target = document.getElementById('target');

});

function handleStateChange() {

    var timeBadge = **new** *Date*().toTimeString().split(' ')[0];

    var newState = document.createElement('p');

    var state = navigator.onLine ? 'online' : 'offline';

    newState.innerHTML = '' + timeBadge + ' Connection state changed to ' + state + '.';

    console.log(newState.innerHTML);

    target.appendChild(newState);

}

window.addEventListener('online', handleStateChange);

window.addEventListener('offline', handleStateChange);

Une image contenant texte, capture d’écran, oiseau

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran

Le contenu généré par l’IA peut être incorrect.

## Envoi et réception d’image

Tests technologiques pour l'enregistrement d'une image dans une base de données MySQL et la récupération d'une image depuis une base de données afin de l'afficher dans une page HTML.

Pour effectuer ce test j’ai utilisé deux fichiers php, un qui va s’occuper d’envoyer les images et d’afficher du HTML et un autre qui va récupérer les images dans la base de données.

*Upload\_base64.php*

<?php

$host = 'localhost';

$dbname = 'db\_tt';

$username = 'root';

$password = '';

try {

    $pdo = new *PDO*("mysql:host=$host;dbname=$dbname", $username, $password);

} catch (*PDOException* $e) {

    die("Erreur de connexion : " . $e->getMessage());

}

if (isset($\_FILES['image'])) {

    $imageData = file\_get\_contents($\_FILES['image']['tmp\_name']);

    $base64Image = base64\_encode($imageData);

    $stmt = $pdo->prepare("INSERT INTO t\_img (img) VALUES (?)");

    $stmt->bindParam(1, $base64Image);

    $stmt->execute();

    echo "Image encodée en base64 enregistrée avec succès dans `t\_img` !";

}

?>

<!DOCTYPE *html*>

<html>

<head>

    <meta *charset*="UTF-8">

    <title>Envoyer une image (base64)</title>

</head>

<body>

    <h1>Uploader une image en base64</h1>

    <form *action*="upload\_base64.php" *method*="post" *enctype*="multipart/form-data">

        <input *type*="file" *name*="image" *required*>

        <input *type*="submit" *value*="Envoyer">

    </form>

    <p><a *href*="view\_base64.php">Voir les images enregistrées</a></p>

</body>

</html>

Le fichier upload va d’abord créer une connexion à la base de données et ensuite vérifier si il y a bien une image a envoyer pour ensuite l’encoder en base64 et l’ajouter à la base de données.  
Les deux fichiers php vont envoyer et recevoir les images sur la base de données stockée en local grâce à WAMP

*View\_base64.php*

<?php

$host = 'localhost';

$dbname = 'db\_tt';

$username = 'root';

$password = '';

try {

    $pdo = new *PDO*("mysql:host=$host;dbname=$dbname", $username, $password);

} catch (*PDOException* $e) {

    die("Erreur de connexion : " . $e->getMessage());

}

$stmt = $pdo->query("SELECT img FROM t\_img");

$images = $stmt->fetchAll(*PDO*::FETCH\_ASSOC);

?>

<!DOCTYPE *html*>

<html>

<head>

    <meta *charset*="UTF-8">

    <title>Images enregistrées (base64)</title>

</head>

<body>

    <h1>Images dans la base (base64)</h1>

    <?php foreach ($images as $img): ?>

        <div *style*="margin-bottom:20px;">

            <img *src*="data:image/jpeg;base64,<?= htmlspecialchars($img['img']) ?>" *style*="max-width:300px;">

        </div>

    <?php endforeach; ?>

</body>

</html>

View va s’occuper de récupérer les images et de les afficher sur la page actuelle.

Une image contenant texte, capture d’écran, Police, ligne

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, dessin humoristique, art

Le contenu généré par l’IA peut être incorrect.

## Requêtes en attentes et base de données locale

Tests technologiques sur les moyens d'enregistrer des données (les résultats en attente) en local et de gérer ces données : cache, base de données locale, fichier, autre.

*index.html*

<!DOCTYPE *html*>

<html *lang*="fr">

<head>

  <meta *charset*="UTF-8">

  <title>PWA Images</title>

  <link *rel*="manifest" *href*="manifest.json" />

  <script *type*="text/javascript" *src*="sw.js"></script>

  <script *src*="js/db.js"></script>

  <script *src*="js/main.js"></script>

  <script>

    if ('serviceWorker' in navigator) {

      window.addEventListener('load', () => {

        navigator.serviceWorker.register('/testTechno/PWARequeteDonneesCache/sw.js')

          .then(*registration* => {

            console.log('Service Worker enregistré avec succès:', *registration*);

          })

          .catch(*error* => {

            console.log('Erreur lors de l’enregistrement du Service Worker:', *error*);

          });

      });

    }

  </script>

</head>

<body>

  <h1>PWA - Gestion des images</h1>

  <div *id*="etatConnexion" *style*="padding: 10px; font-weight: bold;"></div>

  <input *type*="file" *id*="imageFile" *accept*="image/\*">

  <button *id*="uploadBtn">Envoyer l’image</button>

  <button *id*="loadBtn">Charger les images</button>

  <div *id*="resultat"></div>

</body>

</html>

Le fichier html contient les divers liens vers les fichiers JavaScript ainsi qu’un input, deux boutons et un div qui va accueillir le résultat des fichiers php.

*main.js*

  document.addEventListener('DOMContentLoaded', () => {

async function envoyerImage() {

  const fileInput = document.getElementById('imageFile');

  const file = fileInput.files[0];

  if (!file) return;

  const formData = **new** *FormData*();

  formData.append('image', file);

  if (navigator.onLine) {

    try {

      const res = await fetch('upload\_base64.php', {

        method: 'POST',

        body: formData

      });

      if (res.ok) {

        alert(await res.text());

      } else {

        console.error('Réponse non OK, stockage local.');

        await saveImageOffline(file);

      }

    } catch (err) {

      console.error('Erreur réseau, stockage local.');

      await saveImageOffline(file);

    }

  } else {

    console.log('Hors ligne, image stockée localement.');

    alert("Les images seront envoyé lorsque vous serez connecté à internet");

    await saveImageOffline(file);

  }

}

  async function chargerImages() {

    if (navigator.onLine) {

      console.log("chargement en ligne")

      fetch('view\_base64.php')

        .then(*res* => *res*.json())

        .then(*data* => {

          const container = document.getElementById('resultat');

          container.innerHTML = '';

*data*.forEach(*img* => {

            const image = document.createElement('img');

            image.src = `data:image/jpeg;base64,${*img*}`;

            image.style.width = '200px';

            image.style.margin = '10px';

            container.appendChild(image);

          });

        });

    } else {

      try {

        console.log("chargement hors ligne")

        const images = await getAllImagesOffline();

        const container = document.getElementById('resultat');

        container.innerHTML = '';

        images.forEach(*img* => {

          const image = document.createElement('img');

          const reader = **new** *FileReader*();

          reader.onloadend = function() {

            image.src = reader.result;

            image.style.width = '200px';

            image.style.margin = '10px';

            container.appendChild(image);

          };

          reader.readAsDataURL(*img*);

        });

      } catch (err) {

        console.log('Erreur lors du chargement des images locales:', err);

      }

    }

  }

  // Boutons JS

  document.getElementById('uploadBtn').addEventListener('click', envoyerImage);

  document.getElementById('loadBtn').addEventListener('click', chargerImages);

  function mettreAJourEtatConnexion() {

const etat = document.getElementById('etatConnexion');

if (navigator.onLine) {

  etat.textContent = '🟢 Connecté à Internet';

  etat.style.color = 'green';

} else {

  etat.textContent = '🔴 Hors ligne';

  etat.style.color = 'red';

}

}

mettreAJourEtatConnexion();

window.addEventListener('online', () => {

mettreAJourEtatConnexion();

console.log('Connexion retrouvée. Envoi des images locales.');

syncImages();

});

window.addEventListener('offline', () => {

mettreAJourEtatConnexion();

console.log('Perte de connexion');

});

});

Ce fichier va principalement vérifier l’état du réseau et sélectionner les différentes méthodes à utiliser en fonction de l’état de celui-ci. Ce que l’on veut avec ce test c’est stocker les requêtes que l’on fait lorsque l’on est sans connexion pour ensuite les envoyer quand on récupère

# Conclusion

L’ensemble des objectifs définis dans le cahier des charges ont été atteints de manière satisfaisante dans le cadre du projet Elevate. Le développement de l’application a permis de répondre à toutes les fonctionnalités attendues, notamment l’affichage du classement en temps réel, la gestion sécurisée des résultats et des malus, la saisie hors ligne des résultats, ainsi que la synchronisation manuelle des données.

L’application est stable, ergonomique et adaptée à une utilisation sur le terrain, notamment grâce à sa compatibilité PWA et à sa capacité à fonctionner sans connexion Internet. La sécurité des données a été renforcée par l’utilisation de mots de passe hachés et de requêtes préparées pour prévenir les injections SQL.

En conclusion, le projet a permis d’atteindre tous les objectifs fixés, tout en respectant les contraintes techniques et fonctionnelles initiales. Cela démontre une bonne maîtrise de la conception et de la mise en œuvre d’une application web complète, en accord avec les attentes du mandant et du contexte de la compétition de marche et vol.

## Améliorations possibles

Il reste encore plusieurs axes d’amélioration possibles pour compléter et optimiser le projet. La principale amélioration à envisager concerne le fonctionnement hors ligne. Actuellement, lorsqu’un utilisateur est hors ligne, seules les données déjà consultées lors des sessions précédentes sont disponibles. Cela signifie qu’aucune information nouvelle (comme de nouveaux concurrents ou de nouveaux postes) ne peut être affichée si elle n’a pas été préalablement chargée en ligne. Pour pallier cette limitation, il serait intéressant de prévoir un préchargement complet des données nécessaires à l’affichage de toutes les sections de l’application, afin d’assurer un fonctionnement optimal même en mode hors ligne.

Un autre point à améliorer concerne l’envoi des résultats par les commissaires lorsqu’ils sont hors ligne. Actuellement, les requêtes en attente sont stockées localement et envoyées au serveur une fois la connexion rétablie et l’utilisateur ayant cliqué sur le bouton de synchronisation. Cependant, lors de cette synchronisation, le numéro de dossard n’est pas vérifié localement avant l’envoi. Cela pourrait entraîner des erreurs si le numéro saisi est invalide. Une solution serait d’intégrer une vérification locale des numéros de dossard avant la synchronisation, en récupérant les données des concurrents stockées en local ou préchargées pour valider la cohérence des informations.

Enfin, trois fonctionnalités supplémentaires pourraient encore enrichir le projet :

* La détection d’un laps de temps trop court entre deux résultats saisis pour un même concurrent. Cette vérification permettrait d’éviter les doublons et d’assurer la qualité des données.
* L’ajout d’une option permettant de modifier la date et l’heure des résultats, afin de corriger facilement d’éventuelles erreurs de saisie.
* La mise en cache complète de toutes les requêtes nécessaires au fonctionnement de l’application, permettant ainsi une expérience utilisateur optimale même en mode hors ligne.

## Auto-évaluation

Dans l’ensemble, je suis très satisfait du travail effectué et du résultat obtenu. J’ai pu atteindre tous les objectifs définis dans le cahier des charges et répondre aux besoins exprimés lors de la phase de conception. La planification initiale et l’organisation du projet m’ont permis de travailler efficacement et de livrer une application complète et fonctionnelle.

J’ai particulièrement apprécié le développement de la fonctionnalité hors ligne avec IndexedDB et Service Worker, qui m’a permis de découvrir et de maîtriser des technologies modernes et utiles dans des contextes réels. La séparation claire entre le frontend et le backend m’a aidé à structurer le code et à rendre l’application évolutive.

Parmi les facteurs qui expliquent le succès du projet, je citerais :

* une bonne anticipation des besoins fonctionnels et techniques grâce à une analyse approfondie en amont ;
* une organisation rigoureuse du code et des tests réguliers ;
* une motivation constante pour résoudre les problèmes rencontrés (notamment pour la synchronisation hors ligne et la sécurité des données).

En conclusion, ce projet m’a permis de renforcer mes compétences en développement web full stack et en conception d’architectures modulaires. Je suis fier d’avoir pu livrer une application qui correspond aux attentes et qui est prête à être utilisée en conditions réelles.

# Bibliographie: liste des sources et références

Documentation tailwind ->[Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.](https://tailwindcss.com/)

Documentation sweetAlert ->[SweetAlert2 - a beautiful, responsive, customizable and accessible (WAI-ARIA) replacement for JavaScript's popup boxes](https://sweetalert2.github.io/)

Documentation serviceWorker ->[Cache - Web APIs | MDN](https://developer.mozilla.org/en-US/docs/Web/API/Cache)

ChatGPT diverses fonctions

# Signatures

Je soussigné déclare que les informations contenues dans ce rapport de travail pratique individuel rendu ce jour le 02.06.2025 dans le cadre de la procédure de qualification de mon CFC d’informaticien, ne sont pas plagiées. Toutes les informations de sources extérieures ainsi que les informations fournies par des tiers durant le déroulement du travail sont consignées.

|  |  |  |
| --- | --- | --- |
| Date | Nom | Signature |
| 02.06.2025 | Langenegger Max |  |

# Annexes

