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

Modèle de rapport v1.4

Candidat No 151464

Nom du candidat : Max Langenegger

Elevate – Projet TPI

### Sommaire

Elevate – Projet TPI 0

1 Résumé du rapport du TPI 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 Envoie des requêtes en attentes 31

6.7 Gestion des erreurs 34

6.8 Spécifications détaillées 34

6.9 Design du système 34

7 Test 36

7.1 Procédure de test 36

7.2 Protocol de test 36

7.3 Signature du protocole de test 36

7.4 PWA mode hors-ligne 36

7.5 Vérification connexion réseau PWA 37

7.6 Vérification connexion réseau PWA 37

8 Tests Techno 39

8.1 Test PWA 39

8.2 PWA mode hors-ligne 41

8.3 PWA statuts réseau 43

8.4 Envoi et réception d’image 45

8.5 Requêtes en attentes et base de données locale 47

9 Conclusion 51

9.1 Améliorations possibles 51

9.2 Auto-évaluation 51

1 Bibliographie: liste des sources et références 52

2 Glossaire 53

3 Signatures 54

4 Annexes 55

# Résumé du rapport du TPI

Le Résumé du rapport du TPI est une présentation conceptuelle du travail effectué et du résultat attendu d’au maximum une page A4 qui permettent au lecteur une rapide compréhension du rapport de travail. . Il contient trois paragraphes : Situation de départ, mise en œuvre, résultats.

Pour plus de détails, veuillez vous référez au Manuel ICT - partie B : Documentation / rapport du TPI - Question 14 - page 81.

Dans de nombreuses courses comme la Millets Cup, qui se déroule sur les hauteurs de Lessoc, la gestion des résultats repose encore sur un système manuel. Chaque poste (sommet, slalom, touch, cible) est géré par un commissaire, qui enregistre les performances des concurrents sur papier avant de les transmettre à l’organisation. Ce processus est long et sujet aux erreurs : mauvaise retranscription des résultats, oublis, retards dans la mise à jour du classement, et difficulté à rectifier les erreurs.

Mon projet Elevate vise à moderniser et automatiser cette gestion des résultats. Grâce à une saisie numérique en temps réel, chaque commissaire pourra enregistrer directement les performances des concurrents via une interface simple et intuitive, réduisant ainsi les risques d’erreurs et les délais de traitement. Les résultats permettent d’afficher un classement en temps réel accessible à tous. De plus, un mode offline garantit la continuité de la saisie même en l’absence de connexion, avec une synchronisation une fois le réseau rétabli.

Elevate est un nouveau projet que je dois construire de A à Z, brique par brique pour finaliser mon TIP. Le seul fichier actuel lié à ce TPI est le cahier des charges que j’ai reçu.

# 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 présent sur ce projet pour

Etablir la liste des participants au projet avec leur rôle respectif.

Décrire comment sont gérées les sauvegardes de la documentation et du code (fréquence, supports utilisés, utilisation de logiciel de gestion de version, …)

# 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.

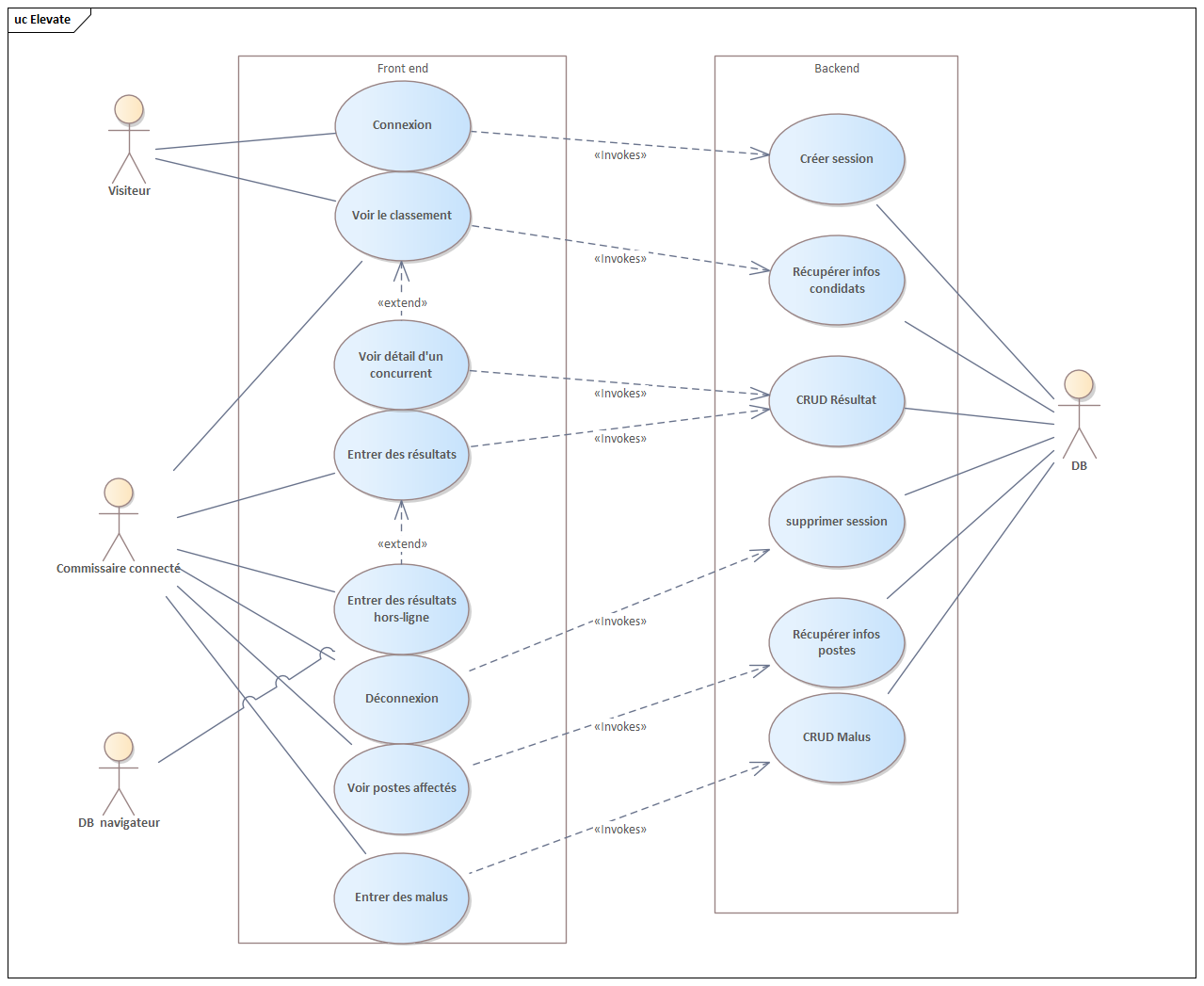
J’ai aussi créé un fichier sessionController.php qui centralise la gestion des sessions (comme la création, la vérification de connexion, et la déconnexion). Ça me permet de garder un code plus propre et plus facile à maintenir.

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'];

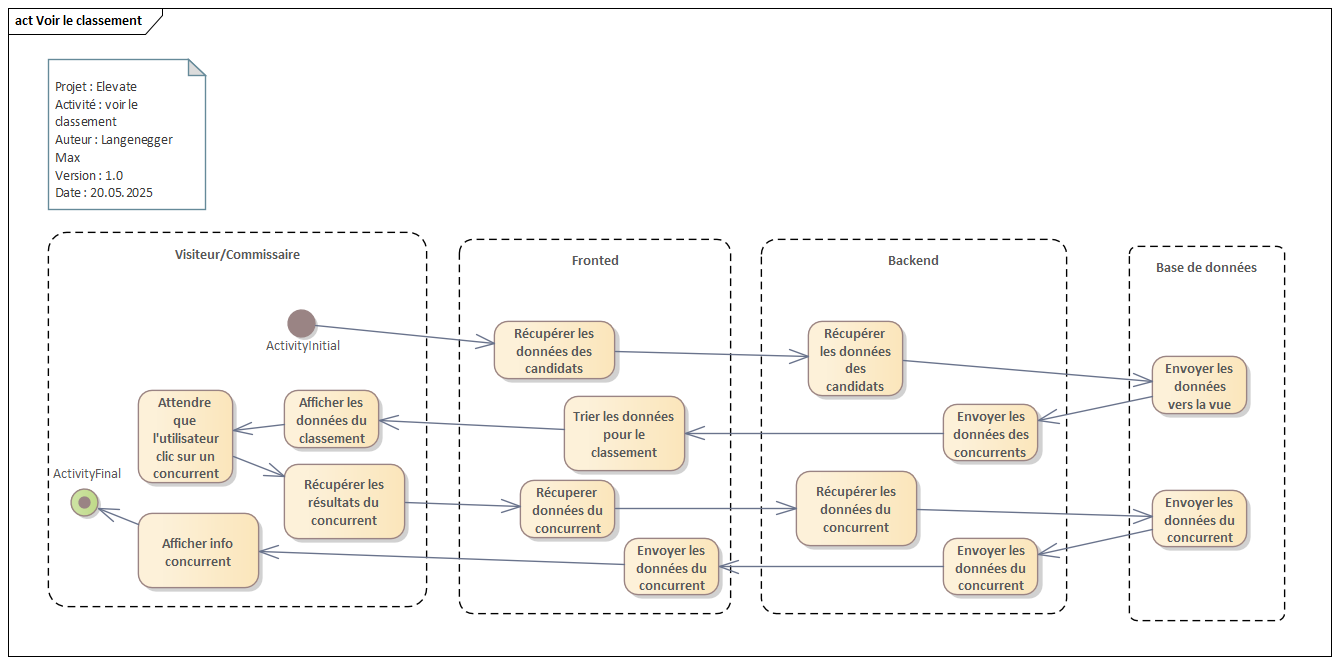
## Diagramme des cas d’utilisation

Les différents acteurs de mon application, côté utilisateur, seront soit des visiteurs ou soit des commissaires qui se sont connecter au préalable. Côté serveur, celui-ci englobe la base de données est les fichiers PHP qui constitueront le serveur en entier.

Les visiteurs pourront consulter le classement actuel avec diverses informations comme le nom et prénom des concurrents, leurs numéros de dossard, leurs nombres de points ainsi que leur lieu d’habitation. Quand un utilisateur va cliquer sur un concurrent, il pourra alors avoir accès à des informations plus détaillé sur le concurrent et ces résultats. Il y aura une liste avec l’intégralité des résultats du concurrent choisi ainsi que d’autre informations sur celui-ci.

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 :**

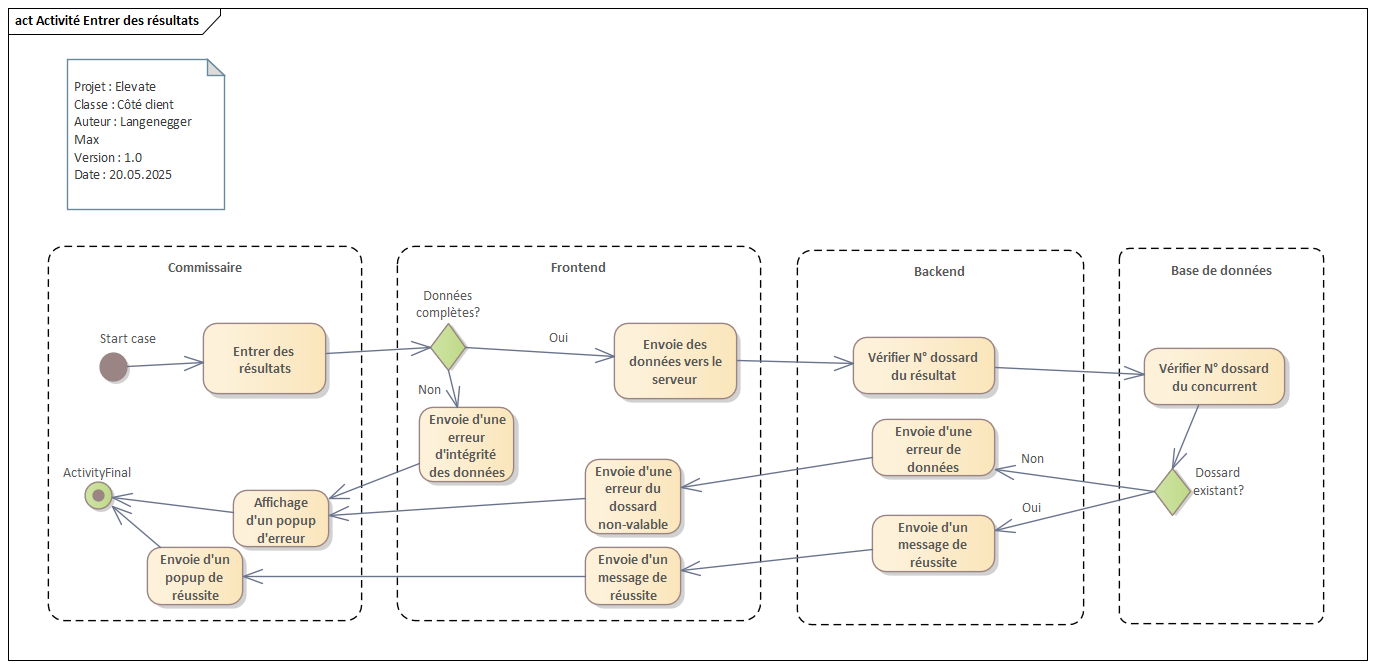
1. **Démarrage**
   * L’activité débute automatiquement lorsqu’un visiteur ou commissaire accède à la page de classement.
2. **Récupération des données du classement**
   * Le frontend envoie une requête au backend pour **récupérer les données de tous les concurrents**.
   * Le backend interroge la base 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 :**

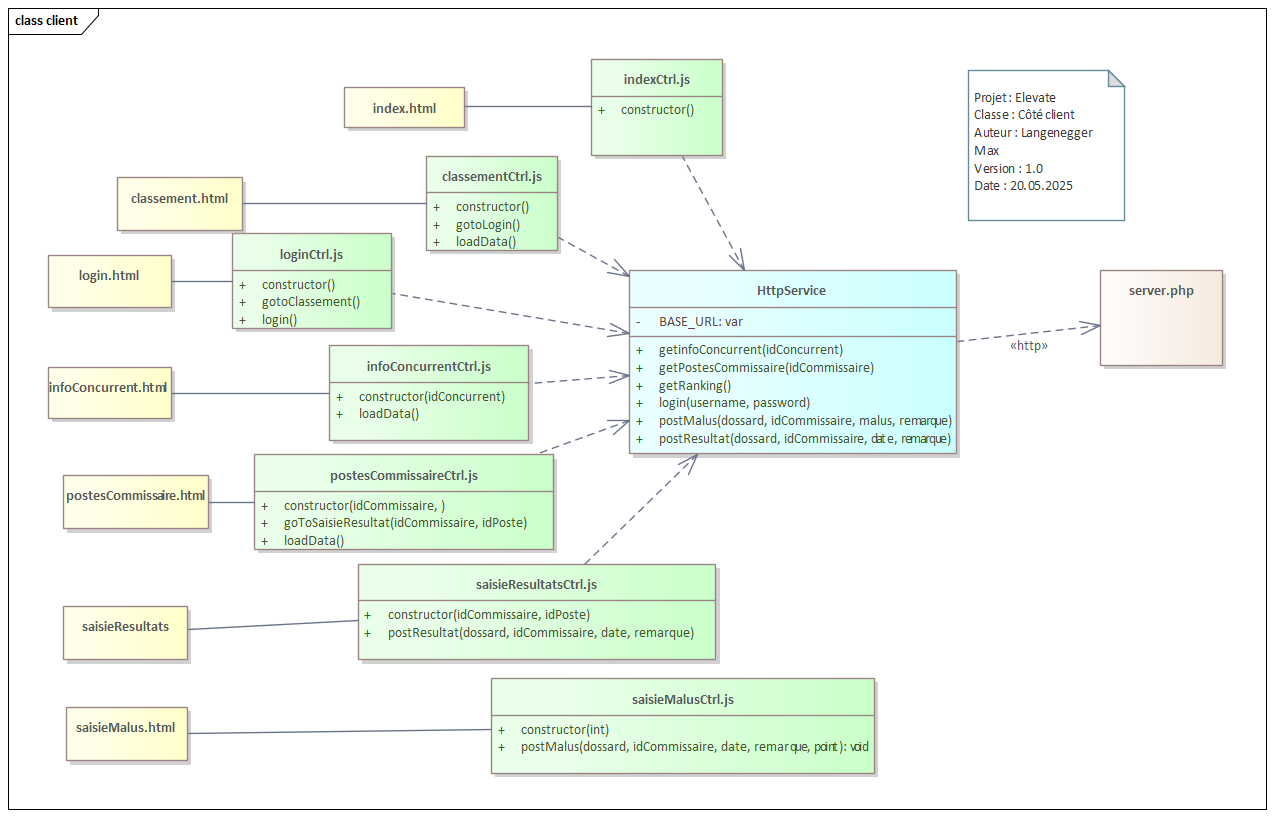
* Vérification précoce côté client pour éviter des appels inutiles au serveur.
* Validation serveur pour garantir l'intégrité des données.
* Gestion claire des cas d’erreur avec retours visuels pour l’utilisateur.
* Utilisation de messages différenciés selon le type d’erreur (intégrité, dossard inconnu,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

## Diagramme de classe

### Client

Le diagramme client présente la structure des différents contrôleurs JavaScript utilisés pour piloter l’interface de l’application PWA.

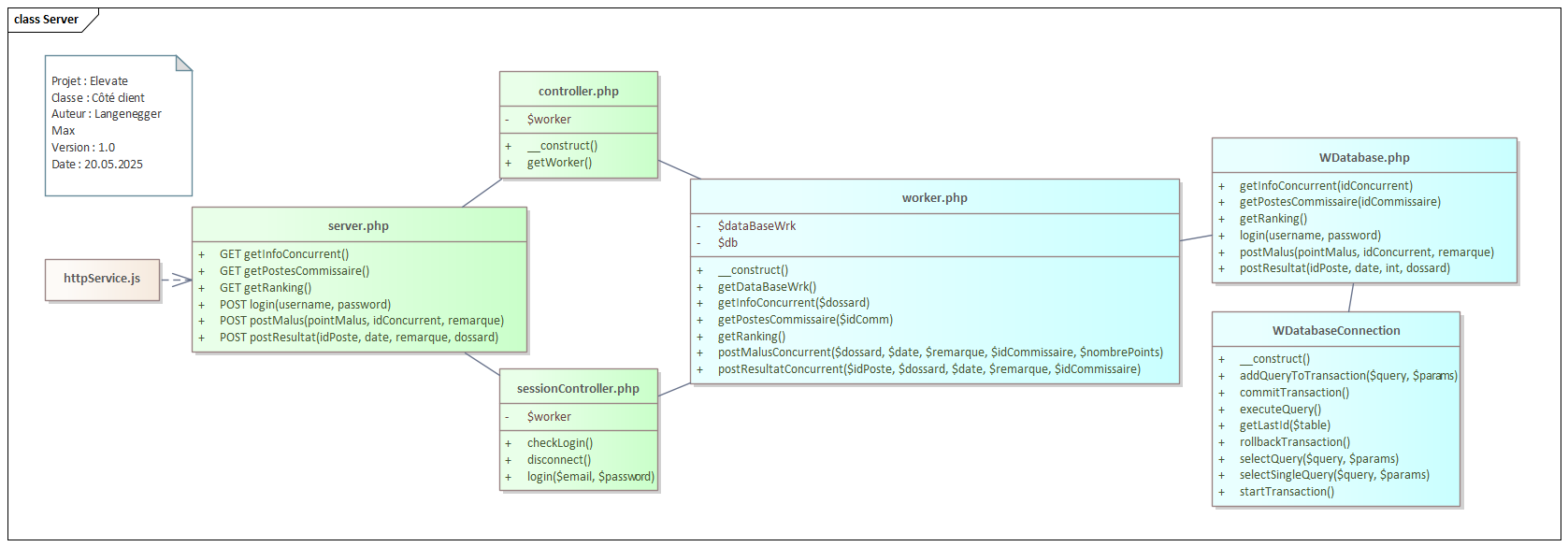
* Chaque fichier \*.js correspond à un contrôleur de page, qui initialise la page (constructor()), charge les données (loadData()), ou permet la navigation entre les vues (gotoClassement() par exemple).
* Tous les contrôleurs font appel à une classe centrale httpService, qui gère les communications avec le backend via fetch().

1. **httpService.js**

* Contient l’URL de base de l’API et les fonctions de communication avec le serveur :
  + getInfoConcurrent()
  + getPostesCommissaire(id)
  + login(email, password)
  + postMalus(...)
  + postResultat(...)

Cette structure permet de séparer les responsabilités : les contrôleurs se concentrent sur la logique de vue, tandis que httpService centralise les appels réseau.

### Serveur

Le diagramme côté serveur illustre le fonctionnement de l’API PHP, organisée de manière modulaire et claire.

**server.php**

* Point d’entrée unique pour toutes les requêtes HTTP. Il redirige les appels vers le bon contrôleur en fonction du type de requête (GET ou POST).

**controller.php & sessionController.php**

* Ces deux fichiers font office de contrôleurs métiers.
* controller.php traite les données générales (résultats, malus…), tandis que sessionController.php gère l’authentification et la session.

**worker.php**

* Sert d’intermédiaire entre les contrôleurs et la base de données.
* Il appelle les fonctions du fichier WDatabase.php.

**WDatabase.php**

* Contient les méthodes métiers liées à la base de données :
  + getRanking()
  + getInfoConcurrent()
  + postResultat(...)
  + postMalus(...)

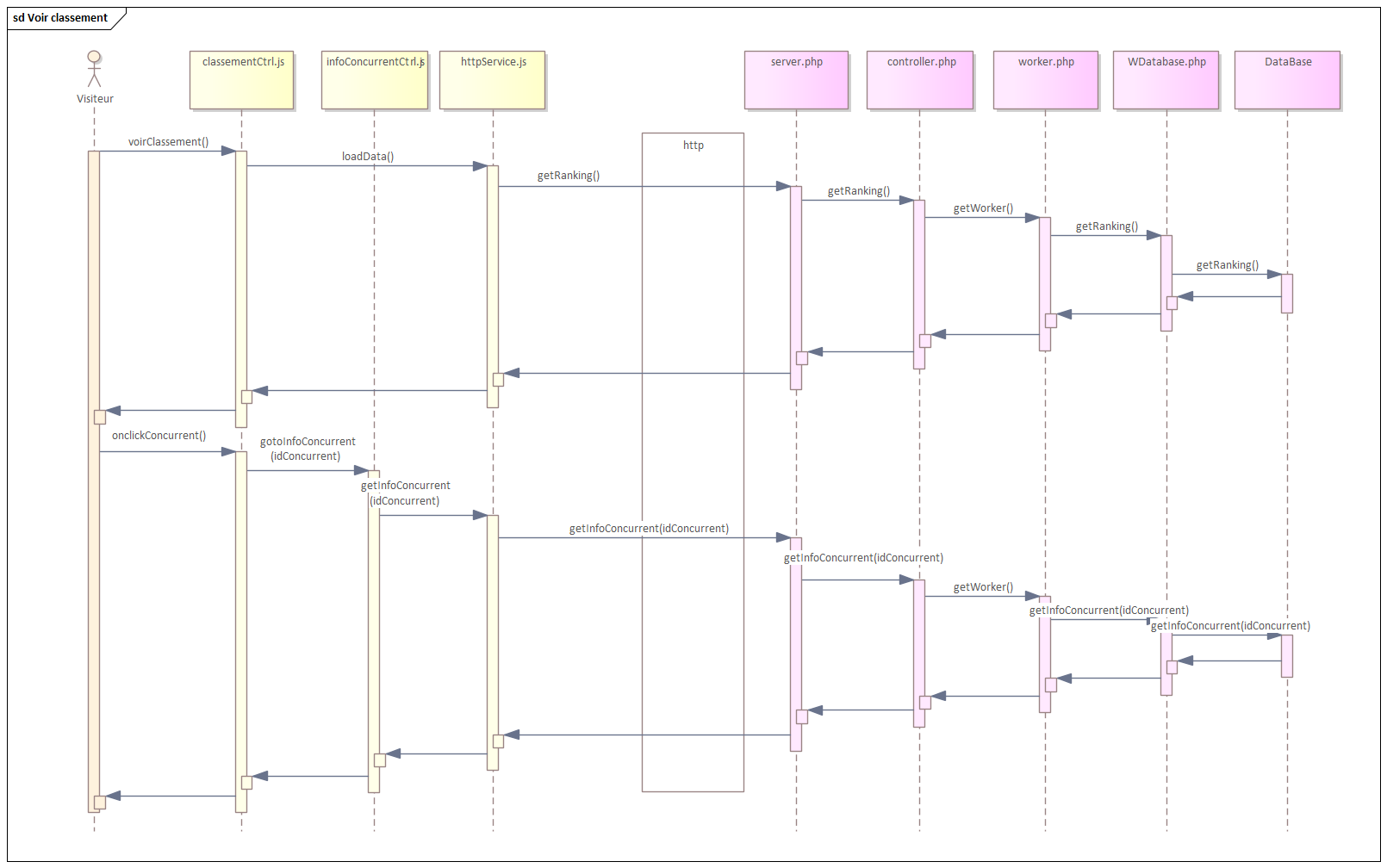
**WDatabaseConnection.php**

* Fournit les fonctions génériques de manipulation de base de données :
  + prepare, executeQuery, startTransaction, selectQuery, etc.

Cette architecture respecte le principe de séparation des couches :

* server.php joue le rôle de routeur
* controller.php/sessionController.php : logique métier
* worker.php : logique de traitement
* WDatabase.php & WDatabaseConnection.php : accès aux données

## Diagramme de séquence

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

## Architecture du système

Pour ce projet, j’ai organisé mon application en deux grandes parties : le client et le serveur. L’idée était de bien séparer ce qui est visible par l’utilisateur (le front-end) de ce qui gère les données et la logique métier (le back-end).

La partie client contient tout ce qui s’affiche dans le navigateur :

* views/ : Ce dossier contient les différentes pages HTML de l’application, comme la page de connexion (login.html), la page du classement, ou encore les pages de saisie pour les commissaires.
* js/controller/ : Tous les fichiers JavaScript qui contrôlent les différentes pages (par exemple loginCtrl.js gère la logique de la page de connexion).
* js/worker/ : Ici, on retrouve par exemple httpService.js qui gère les requêtes vers le serveur.
* sw.js et manifest.json : Ces fichiers permettent d’ajouter des fonctionnalités PWA (comme l’installation sur le bureau ou le cache offline).

La partie server s’occupe de toute la logique côté back-end :

* controllers/ : On y retrouve le contrôleur principal et le sessionController.php qui gère les connexions, la création et la destruction des sessions utilisateur.
* workers/ : C’est là que sont gérés tous les traitements plus techniques.
  + database/ : Contient la classe de connexion à la base de données (WDatabaseConnection.php) et les fonctions de requêtes (WDatabase.php).
  + worker.php : Fait le lien entre les contrôleurs et la base de données.
* exceptions/ : Pour gérer proprement les erreurs (ex : DatabaseException).
* config.php : Contient les informations de configuration (comme les identifiants de la base).
* server.php : Le point d’entrée de l’API, qui route les requêtes vers le bon contrôleur.

Cette organisation me permet d’avoir un code clair, bien structuré et facilement maintenable. Elle facilite aussi la réutilisation des fonctions, la sécurisation du code, et la gestion des différentes responsabilités dans le projet.

## Concept de tests

|  |  |  |
| --- | --- | --- |
| **Fonctionnalité testée** | **Méthode de test** | **Résultat attendu** |
| 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. |
| Afficher les résultats et les malus d’un concurrent | Cliquer sur un concurrent et voir tout les résultats | Les résultats et les malus liés au dossard sélectionné s’affichent correctement. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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 |
| 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. |
| 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. |
| 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. |
| 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. |
| 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 |

## Moyens nécessaires

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

# 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

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

Le contenu généré par l’IA peut être incorrect.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.

**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’object 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.

## Gestion des erreurs

## Spécifications détaillées

## Design du système

# Test

## Procédure de test

## Protocol de test

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nr. | Objet testé | Description du test | Attente | Résultat | Visa |
| 1.0 |  |  |  |  |  |

## Signature du protocole de test

|  |  |  |
| --- | --- | --- |
| Date | Nom | Signature |
| 30.05.2025 |  |  |

# 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

Conclusion sur le projet en général. Les objectifs sont-ils atteints ? Si non, pourquoi ?

## Améliorations possibles

Il y a encore pas mal d’axes d’amélioration disponible pour compléter au mieux le projet.

La plus grande partie qui peut être améliorer est quand on est offline, pour le moment on récupère et stock les données que nous avons déjà vu lors de la visite de l’application. On ne pourra pas voir une information que l’on n’a pas déjà vu quand on était online. L’amélioration possible pour palier à ce problème on pourrait charger toutes les requêtes nécessaires au bon fonctionnement de l’affichage de toutes les données. Le deuxième point important en mode offline concerne les résultats qu’envoient les commissaires quand ils sont offline. Actuellement, les requêtes sont envoyées à la base de données quand le commissaire retrouve la connexion et appuie sur le bouton destiné à l’envoie des requêtes. Le problème est que le numéro de dossard n’est pas vérifié quand on renvoie les requêtes qui sont en attentes. Cet problématique rejoint le point précédent, il faudrait récupérer les données des concurrents pour ensuite vérifier en local si le dossard existe ou non.

## Auto-évaluation

Quels sont mes sentiments vis-à-vis du travail effectué. Enoncé des facteurs qui expliquent le succès ou l’échec de la réalisation des objectifs du projet.

# Bibliographie: liste des sources et références

Conseil: Cette partie peut contenir une liste des sources et éléments littéraires utilisés.

# Glossaire trier par ordre alphabetique

|  |  |
| --- | --- |
| Terme | Signification |
|  |  |
|  |  |
|  |  |

# Signatures

Je soussigné déclare que les informations contenues dans ce rapport de travail pratique individuel rendu ce jour le XX.XX.20XX dans le cadre de la procédure de qualification de mon CFC d’informaticien/-ne, 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.

L’apprenti/-e doit signer la documentation avant la remise afin de témoigner de la justesse des informations qui y figurent.

|  |  |  |
| --- | --- | --- |
| Date | Nom | Signature |
|  |  |  |

# Annexes

Code imprimé, protocole de séances, etc.: spécifier complètement et annexer à la documentation.

Listings de scripts et programmes. Le travail fourni par la candidate ou le candidat doit être entièrement documenté. Omettre le code généré automatiquement s’il n’est pas nécessaire à la compréhension.

Dans le cas où des manuels ont été créés, vous pouvez les ajouter ici en tant qu’annexe.