Projet XYZ

Table des matières

1 Analyse préliminaire

1.1 Introduction

L'objectif principal de ce projet est de créer une application de simulation de saison de Formule 1. Elle se présentera sous forme de jeu afin de divertir ses utilisateurs. Cette application sera développée sous forme d'application web en utilisant le langage de programmation Javascript.

Ce projet se fait dans le cadre du TPI. Cela veut dire que le projet sera réalisé en suivant les modalités du TPI. Le dossier de projet ainsi que la présentation faisant suite à ce projet seront aussi évalués en suivant la grille d'évaluation TPI VD.

Plusieurs projets ont été réalisés en prévision de ce TPI. Le premier, un projet de monitoring de ruche d'abeille, n'avait aucun lien avec la technologie qui sera utilisée ni aucun rapport avec le thème (La formule 1) de ce TPI. Cependant, ce projet, ayant été évalué avec la grille d'évaluation du TPI, m'a permis de me familiariser avec cette dite grille ainsi que ce que devrait contenir le dossier de projet. Le second projet la création d'un site web de shopping avec un back-office était, lui, un projet non-officiel. Cela veut dire que le projet ne suivait pas de cahier des charges ni d'évaluation finale mais me servait uniquement de moyen d'assimiler la technologie qui sera utilisée dans ce TPI et d'y gagner en aisance. Plusieurs fonctionnalité ou bout de code seront repris de ce projet.

Ce chapitre décrit brièvement le projet, le cadre dans lequel il est réalisé, les raisons de ce choix et ce qu'il peut apporter à l'élève ou à l'école. Il n'est pas nécessaire de rentrer dans les détails (ceux-ci seront abordés plus loin) mais cela doit être aussi clair et complet que possible (idées de solutions). Ce chapitre contient également l'inventaire et la description des travaux qui auraient déjà été effectués pour ce projet.

Ces éléments peuvent être repris des spécifications de départ.

1.2 Objectifs

1.2.1 Livrable

Une planification initiale contenant un diagramme de Gantt devra être livrée aux 2 experts ainsi qu'au chef de projet par courriel électronique sous le format PDF à la fin du premier jour (2 mai 2025).

Le rapport de projet en l'état devra être livré 3 fois par semaine le lundi, le mercredi et le vendredi en fin de journée. Le rapport de projet sera livré sous forme de PDF dans le repo Git dans un dossier dédié à la documentation.

Le livrable finale devra contenir :

- Le journal de travail avec mention de tout ce qui a été accompli durant le projet devra aussi être livrée.
- Le code source de l'application devra être livré aux 2 experts ainsi qu'au chef de projet. La livraison se fera sous la forme de partage de lien au dépôt Git.
- Les scripts qui serviront la création de base de données ainsi que des données de test
- Un guide de déploiement de l'application sur l'infrastructure locale de l'ETML

A la fin du délai imparti pour la réalisation du TPI, c'est à dire avant le 2 juin 2025 à 11h20, une version électronique du dossier de projet devra être transmise aux 2 experts ainsi qu'au chef de projet. En parallèle, une copie papier du rapport devra être fournie sans délai en trois exemplaires. Elle devra être en tout point identique à la version électronique.

1.2.2 Objectif du produit.

Comme énoncé dans l'introduction, l'objectif du projet et créer une application de simulation de saison de Formule 1. Etant donné que le descriptif peut porter à confusion, je vais décrire, ici, ce à quoi consiste cette application. En effet, il ne s'agit pas de simuler la saison de Formule 1 actuelle, mais de donner la possibilité à un utilisateur de se divertir en choisissant une écurie, choisir les 2 pilotes qui composeront cette écurie et de simuler une saison entière. Une fois la saison lancée, l'utilisateur n'aura aucune influence sur le résultat des courses. Les résultats des courses seront aléatoires en prenant en compte le rating[1] du pilote. Les courses comprendront au moins un accident et un abandon. Cela sera un jeu en solo. Cela veut dire qu'aucun autre joueur pourra prendre part à la session de jeu de l'utilisateur.

Ce chapitre énumère les objectifs du projet. L'atteinte ou non de ceux-ci devra pouvoir être contrôlée à la fin du projet. Les objectifs pourront éventuellement être revus après l'analyse.

Ces éléments peuvent être repris des spécifications de départ.

1.3 Gestion de projet

Une planification initiale sera faite le premier jour de ce projet, directement après la réception du cahier des charges (la planification initiale est trouvable sous le chapitre 1.4).

La méthodologie utilisée lors de ce projet de TPI est celle dite *en cascade*. C'est-à-dire que lors de la planification initiale, le projet a été analysé pour en ressortir toutes les tâches qui seront réalisées. Les tâches ont été mises ensuite dans un ordre logique pour suivre la logique du *Waterfall* qui est qu'une tâche ne peut pas être réalisée avant

[1] Niveau de compétence du pilote

que la précédente ne soit finie. En effet, il sera difficile de s'occuper de la gestion de course avant de s'occuper de la gestion des choix des pilotes et des écuries.
L'entièreté du projet sera disponible sur une repo github (Code source de l'application, planification, journal de travail, rapport, cahier des charges, schémas).

Les personnes impliquées dans ce projet sont :

- Gaël Sonney, chef de projet.
- Yves Bertino, expert n° 1
- Nemanja Pantic, expert n° 2

1.4 Planification initiale

Une planification a donc été faites le premier jour de ce TPI. Les tâches ont d'abord été séparées en plusieurs catégories :

- Entretien, les différents entretiens avec les experts ou le chef de projet.
- Analyse, analyse du projet ou d'une fonctionnalité.
- Réalisation, développement d'une fonctionnalité ou réalisation d'une tâche précise.
- Test, tester une fonctionnalité.
- Documentation, rédiger la partie de la documentation du projet.

Un découpage des tâches plus précises a ensuite été fait. Puis, pour chaque séquence de travail, une estimation des tâches qui seront réalisées ainsi que le temps passé dessus a été faite. Sachant qu'une séquence est définie par une demi-journée de travail.

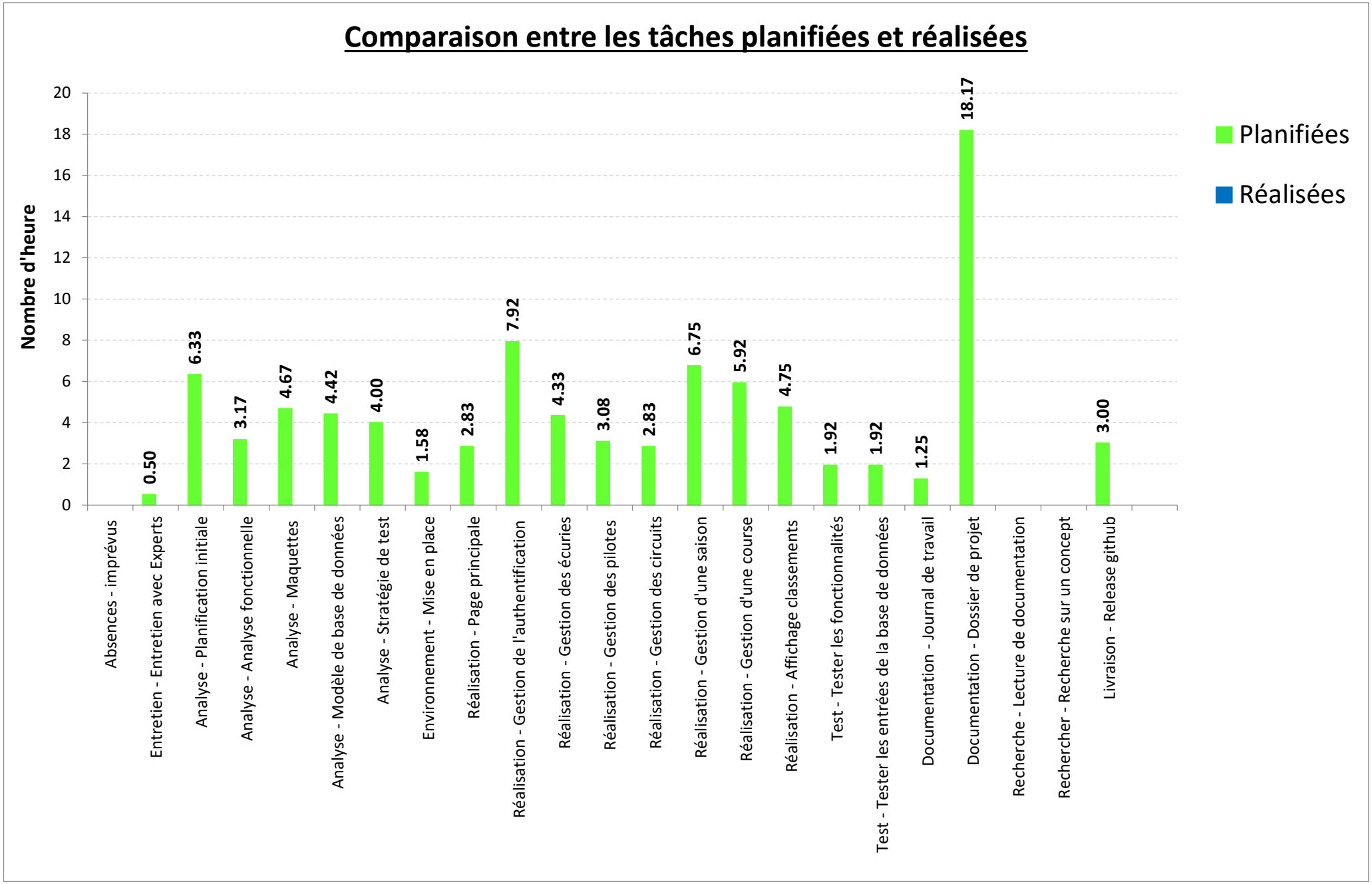


Figure 1: Planification initiale

A Garder à l'esprit que cette planification initiale n'est qu'une première idée de comment pourrait se dérouler le projet et qu'elle n'est là que pour donne une idée de l'ordre des tâches à effectuer ainsi que le temps estimé que prendra chacune d'elle. Il est très peu probable que cette planification initiale soit respectée parfaitement. Des imprévu peuvent survenir, de nouvelles idées, des idées de réalisation différentes, ou une mauvaise estimation du temps peuvent en être la cause.

La période de réalisation de ce projet du TPI est du **vendredi 2 mai 2025 à 8h au lundi 2 juin 2025 à 11h25.**

Les horaires de travail sont :

Lundi	08h00 – 11h25	12h20 – 14h45
Mardi	-	-
Mercredi	08h00 – 12h15	13h10 – 16h35
Jeudi	-	13h10 – 16h35
Vendredi	08h00 – 12h15	13h10 – 16h35

Avec comme jours fériés :

- Lundi 26 mai après-midi, Examen eCG
- Jeudi 29 mai, Ascension
- Vendredi 30 mai, Pont de l'Ascension

Ce chapitre montre la planification du projet. Celui-ci peut être découpé en tâches qui seront planifiées. Il s'agit de la première planification du projet, celle-ci devra être revue après l'analyse. Cette planification sera présentée sous la forme d'un diagramme.

Ces éléments peuvent être repris des spécifications de départ.

2 Analyse / Conception

2.1 Contexte produit

Le Produit sera un accès à un site internet. L'utilisateur devra en premier lieu se connecter afin de pouvoir profiter des fonctionnalités. Le site web sera principalement développé pour les navigateurs web via ordinateur. Si un maximum sera fait pour rendre le site web plus modulable, il ne sera pas pensé pour avoir un responsive design complet. Le site sera tout de même accessible sur un navigateur web depuis un téléphone mobile, mais ne la vue ne sera pas totalement adaptée.

L'application aura 2 types d'utilisateur :

- Les utilisateurs simples, qui pourront jouer et consulter les données
- Les administrateurs, qui pourront modifier les données.

Actions / Rôles	User	Admin
Voir les informations des écuries	✓	✓
Modifier les informations des écuries	✗	✓
Supprimer une écurie	✗	✓
Ajouter une écurie	✗	✓
Voir les informations des pilotes	✓	✓
Modifier les information des pilotes	✗	✓
Supprimer un pilote	✗	✓
Ajouter un pilote	✗	✓
Voir les informations des circuits	✓	✓
Modifier les informations des circuits	✗	✓
Supprimer un circuit	✗	✓

Ajouter un circuit	✗	✓
Lancer une saison	✓	✓

2.1.1 Utilisation

Prenons l'exemple d'utilisation d'un utilisateur simple. Il devra en premier lieu se connecter au site internet avec ses identifiants. S'il n'en a pas, il aura la possibilité de de créer un compte. Ensuite, il aura l'occasion de consulter les données des pilotes, des circuits ou des écuries s'il en a l'envie. Pour commencer une partie, il devra se rendre sur la page principale puis cliquer sur le bouton *Lancer une saison*. Le site web affichera ensuite une liste des écuries, il en choisira une. Puis lui sera affiché la liste des pilotes, il fera de même mais en choisira 2. Ce seront ses 2 pilotes qui courront pour lui dans l'écurie qu'il a choisie. Une fois confirmé, la saison débute. Chaque course se lanceront une à une, un classement des pilotes et des constructeurs sera affiché à la fin de chaque course. L'utilisateur devra, néanmoins, confirmer à chaque fin de course avant de passer à la suivante. A la fin de la saison, un classement final sera affiché. Le joueur se retrouvera donc comme au début de sa session de jeu, sur la page principale avec la possibilité de recommencer une partie.

Placer le produit dans son contexte d'utilisation. Par exemple :

- ***Site Web Internet***
- ***Application, Web, intranet***
- ***Application mobile***
- ***Infrastructure dupliquer des sites multiples***

Présenter les utilisateurs du système ainsi que les rôles qu'ils peuvent avoir

Présenter les données/information que le système prend en charge

Au minimum :

- *Un ou plusieurs schémas de contexte montrant le produit dans son environnement d'utilisation, ainsi que ses utilisateurs. Ce type schéma doit être accompagné d'explications textuelles*

Le concept complet avec toutes ses annexes:

Par exemple :

- *Multimédia: carte de site, maquettes papier, story board préliminaire, …*
- *Bases de données: interfaces graphiques, modèle conceptuel.*
- *Programmation: interfaces graphiques, maquettes, analyse fonctionnelle…*
- *…*

2.2 Contexte technique

2.2.1 Opérationnel

Le produit final est divisé en plusieurs parties :

2.2.1.1 Frontend

La partie frontend[2], c'est-à-dire la partie qui sera visible par l'utilisateur, sera développée en utilisant Vite, React ainsi que TypeScript[3], plus précisément avec le format TSX. Le TSX nous permettra d'intégrer directement du TypeScript dans nos balise JSX[4].

Vite JS est, par définition, un serveur de développement utilisé pour faire tourner localement des applications web. Cela nous permettra aussi de faire un build de notre application afin de le déployer en production.

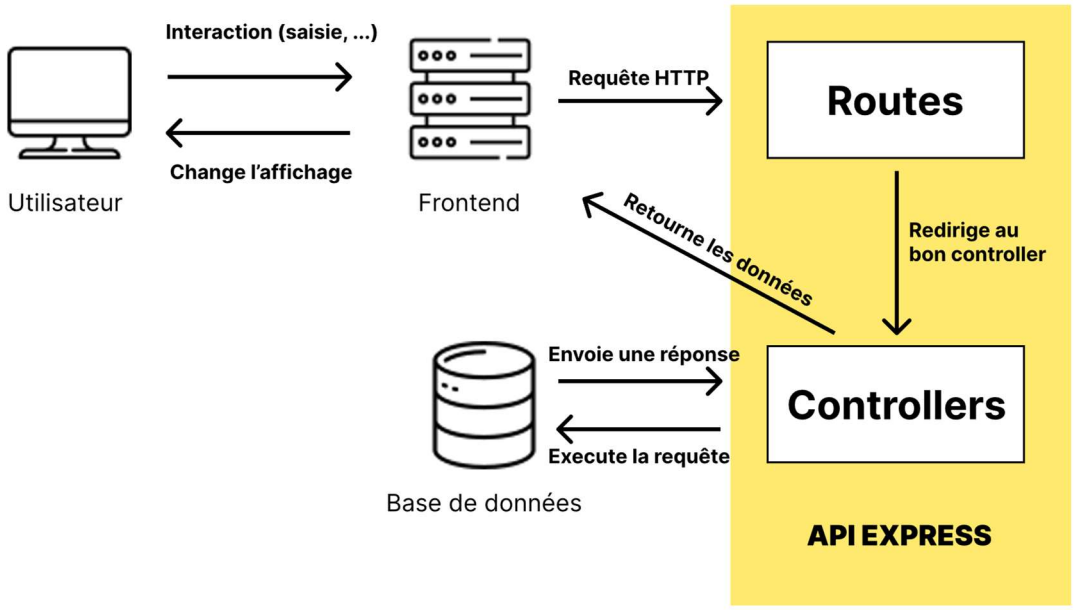
De plus, pour le style de l'application web, je vais utiliser tailwindcss, un framework css qui permettra de gagner du temps lors de la mise en style de notre site.

2.2.1.2 Backend

La partie backend[5], la partie dont l'utilisateur n'a pas directement accès, sera développée avec Express JS. Express JS est un framework backend pour Node.js, qui

[2] Partie visible par l'utilisateur
[3] Langage ressemblant au JS mais typé
[4] Extension de syntaxe pour JS permettant d'y intégrer du HTML
[5] Partie qui interagira avec la base de données

nous permet de créer des API[6], de gérer des requêtes HTTP et faire du routage. Son fonctionnement peut se décrire de cette façon :



Prenons l'exemple ou l'utilisateur se rends sur la page des pilotes. Lors du chargement de la page, la partie frontend effectue une requête HTTP sous forme de *http://localhost:3040/api/drivers* est effectuée (dans le cadre de l'environnement de développement). Dans le fichier *index.js* de notre backend toutes les routes API y sont définies. Parmi elles, la route */api/drivers* est associé au fichier *routes/drivers.js.*
Ce fichier de routes importe les fonctions définies dans notre Controller et lie chaque routes avec une fonction.
Dans notre exemple, comme notre URL est simplement *api/drivers/* (aucun paramètre supplémentaire), c'est la fonction associée à la route « / » qui est exécutée. Dans notre cas c'est la fonction *getAllDrivers*. La fonction avec ce nom présent dans le *DriversController* va ensuite s'exécuter et va effectuer une requête SQL à notre base de données. Le *controller* va ensuite retourner les données reçues à la partie frontend qui va mettre à jour l'affichage et afficher la liste des pilotes

[6] Application Programming Interface : permet la communication entre 2 applications

Placer le produit dans son contexte d'opérations.

Ce chapitre doit impérativement contenir au moins une représentation graphique

Au minimum :

- *Un ou plusieurs schémas de contexte montrant le produit dans son environnement technique. Ce type schéma doit être accompagné d'explications textuelles*

2.2.2 Environnement de développement

L'environnement de développement sera l'application lancée en local sur un PC. Le frontend sera déployé sur le port 5174 tandis que le backend le sera sur le 3040. Il faudra donc démarrer l'environnement via une console, celle de VSC par exemple, via la commande « npm run dev » pour le frontend puis « node index.js » pour le backend. Cela veut dire que cet environnement n'est pas accessible depuis un poste extérieur. Comment son nom l'indique, c'est sur cet environnement que les fonctionnalités seront d'abord développées puis testée avant d'être remise sur l'environnement de production.

2.2.3 Environnement de production

L'environnement de production sera le serveur disponible à l'ETML. Ce serveur contient un service Nginx et Node ce qui nous permet d'héberger mon application web. Le port ouvert et accessible qui m'a été remis est le port 3040. L'application devra donc être modifiée pour que le backend et le frontend utilisent les 2 ce port attribué. L'application, étant hébergée sur le serveur de l'ETML, sera donc accessible par n'importe quelle appareil connecté à internet depuis un navigateur web. Le lien pour y accéder est *aki.w3.pm2etml.ch*. Lorsqu'une fonctionnalité a passé tous les tests sur l'environnement de dev, elle pourra être push sur le serveur. Un nouveau build de l'application devra être fait pour que les changements prennent effets.

2.2.4 Justification des choix

2.3 Dossier de conception

Dans ce chapitre de concept, nous verrons tout ce qui est zonings, maquettes du site web, les différents schémas dont ceux de la base de données et ceux du fonctionnement des fonctionnalités du site web.

2.3.1 Zonings

Les zonings sont les premières esquisses graphiques des pages du sites web permettant de visualiser l'organisation des différents éléments qui seront présents sur les pages. Les zonings ont été réalisé sur Figma.

2.3.1.1 Page principale

Sur la page principale, on trouvera la barre de navigation en haut de notre écran qui contiendra les différents boutons pour ensuite naviguer à travers les différentes pages. Au centre de l'écran se trouvera la *zone de jeu*. C'est ici que se présentera le terrain de jeu pour notre utilisateur. Elle contiendra par exemple le déroulement de la course, ou bien le classement de la course une fois la course terminée.

Figure 2: Zoning page principale

2.3.1.2 Liste des pilotes

Cette page contiendra, comme toutes les autres, la barre de navigation horizontale en haut de l'écran. La liste des pilotes y figurera. Un simple utilisateur pourra ne pourra que consulter ces informations. Seul un utilisateur ayant comme rôle « Administrateur » pourra ajouter, modifier ou supprimer un des pilotes.

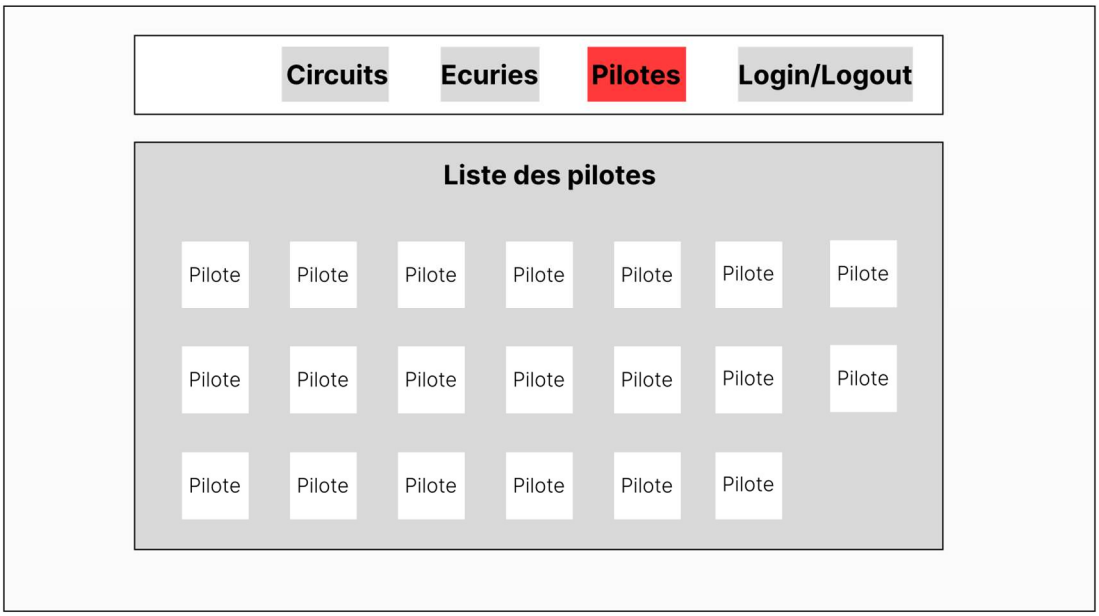


Figure 3: Zoning liste pilotes

2.3.1.3 Liste des écurie et des circuits

Ces pages sont relativement similaires à celle des pilotes dans leur fonctionnement. C'est-à-dire que seul un utilisateur de type « Administrateur » pourra y faire des modifications.

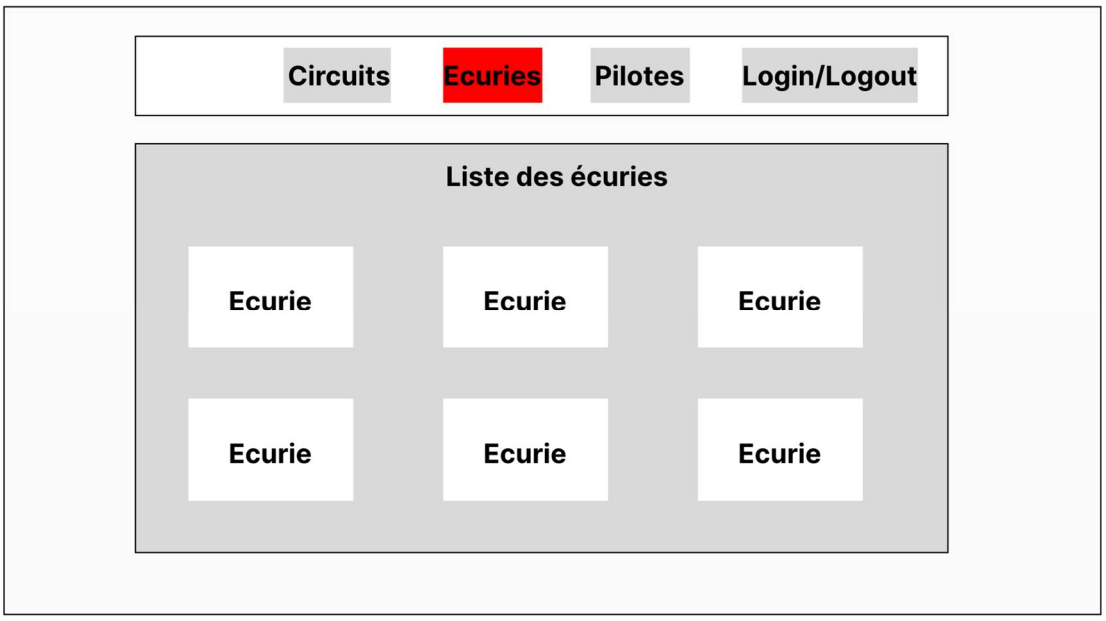


Figure 4: Zoning liste écuries

Etant donné que le zoning de la liste des circuits ressemble énormément à celui des écuries, j'ai trouvé plus pertinent de ne pas le mettre dans le rapport. Cependant il sera bien évidemment mis en annexe.

2.3.2 Maquettes

Les maquettes sont beaucoup plus complètes que les zonings et se rapproche plus du résultat final. Elles ont, comme pour les zonings, été faite sur Figma. Par soucis de pertinence, pas toutes les maquettes sont présentes dans ce chapitre étant donné que plusieurs pages se ressemblent beaucoup (page des pilotes / page des écuries). Vous pourrez trouvez l'ensemble de toutes les maquettes en annexe.

2.3.2.1 Page principale

Figure 5: Maquette page principale

Sur la barre de navigation, la section ou était présent la zone « login/logout » dans le zoning contiendra le nom de l'utilisateur connecté ainsi qu'un bouton de déconnexion qui le reverra sur la page de connexion.

2.3.2.2 *Liste des pilotes*

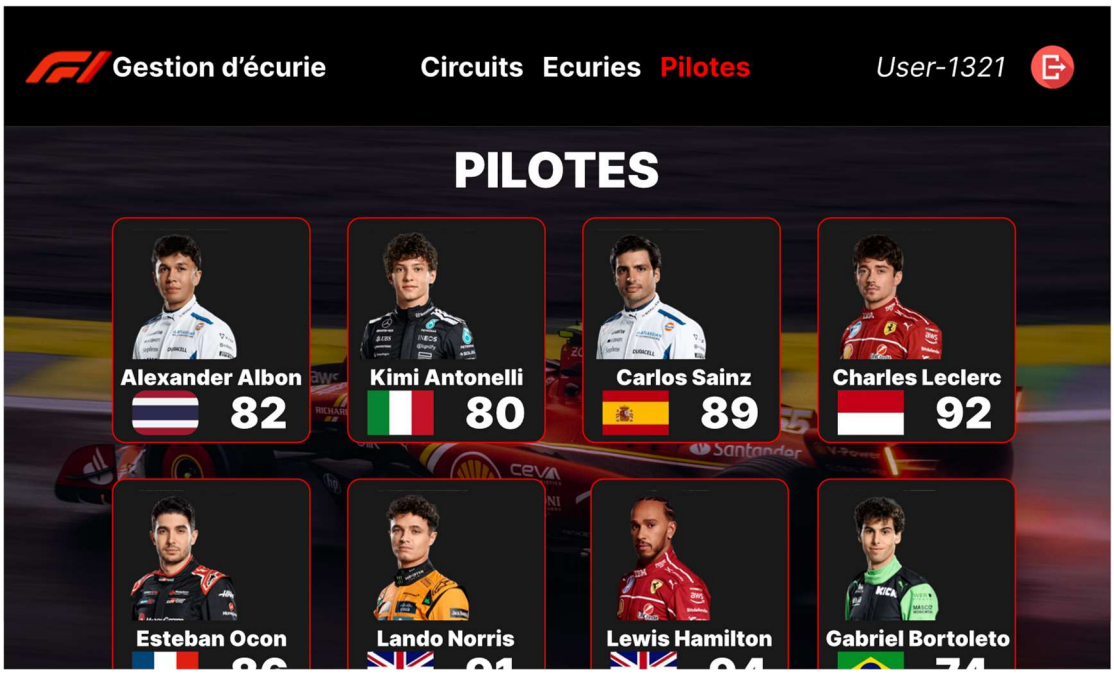


Figure 6: Maquette liste pilotes

Les zones présentent sur la maquette se rapproche assez des zonings mais apportent plus de précision notamment au niveau de container des pilotes. Leurs nom sera affiché ainsi qu'un image, un rating et leur nationalité. Un administrateur aura une vue un peu différente de celle d'un simple utilisateur. Il verra affiché un bouton à côté de chaque pilote afin de pouvoir modifier ses informations. Un bouton redirigeant sur un formulaire de création de pilote sera aussi accessible uniquement aux administrateurs.

2.3.2.3 *Liste des écuries*

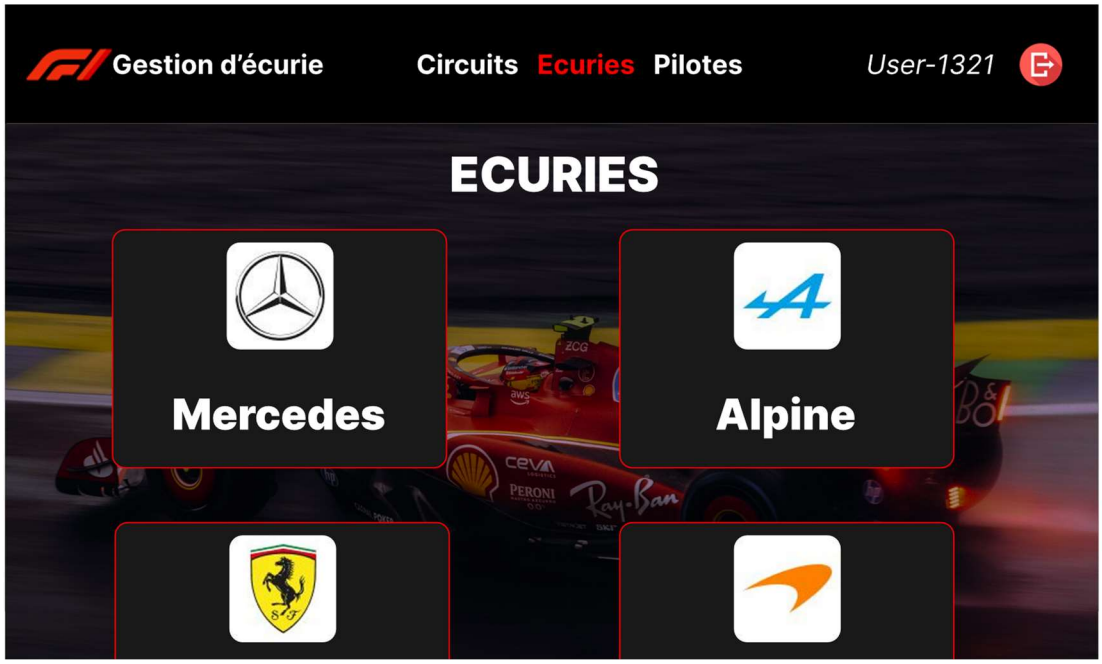


Figure 7: Maquette liste écuries

La liste des écuries reprends aussi les zonings avec la même fonctionnalité de connexion dans la barre de navigation. Un administrateur aura, de même, accès à un bouton pour modifier ou supprimer une écurie ainsi qu'un bouton qui redirige vers un formulaire de création d'écurie.

2.3.2.4 *Classement après course*

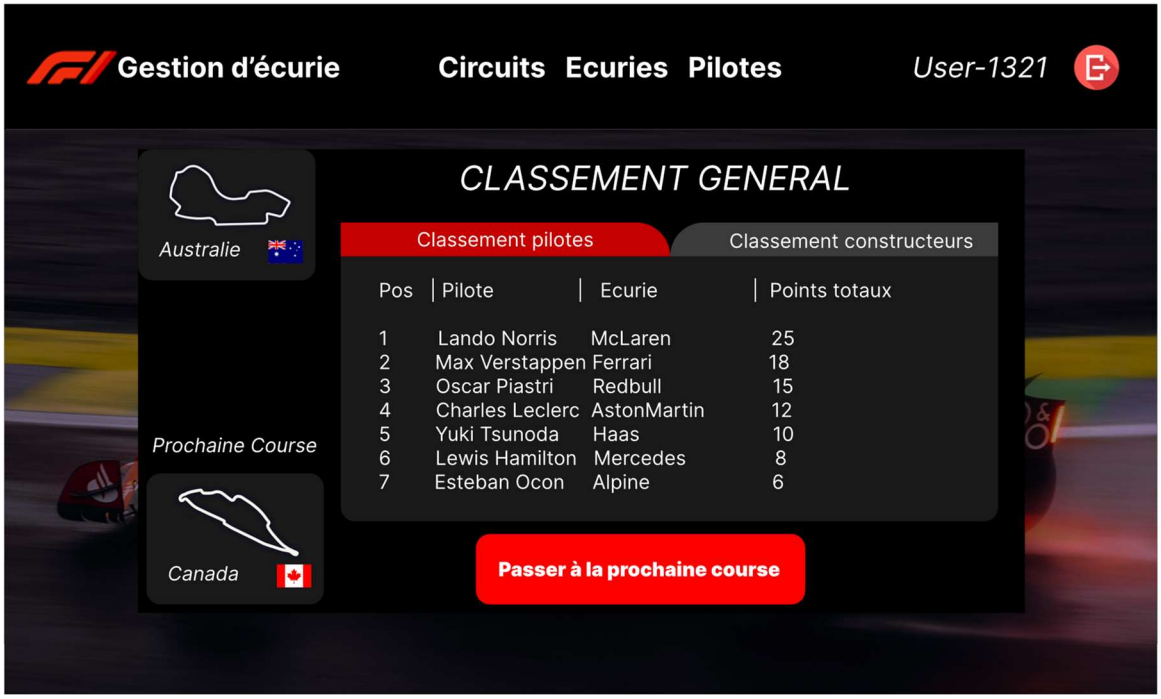


Figure 8: Maquette classement après-course

Après chaque course, un classement sera affiché. L'utilisateur pourra basculer entre le classement des pilotes et le classement des constructeurs. Les pilotes ainsi que l'écurie que l'utilisateur contrôle seront marqué d'une couleur différente afin de pouvoir les différencier plus facilement. Une image avec la course actuelle et la course suivant seront aussi affiché. Un compteur de course pourra aussi être affiché afin de donner plus d'informations utiles à l'utilisateur.

2.3.3 Modèle de base de données

2.3.4 Analyse fonctionnelle

2.3.4.1 *Connexion de l'utilisateur*

L'utilisateur devra être identifié pour pouvoir profiter des fonctionnalités de l'application.

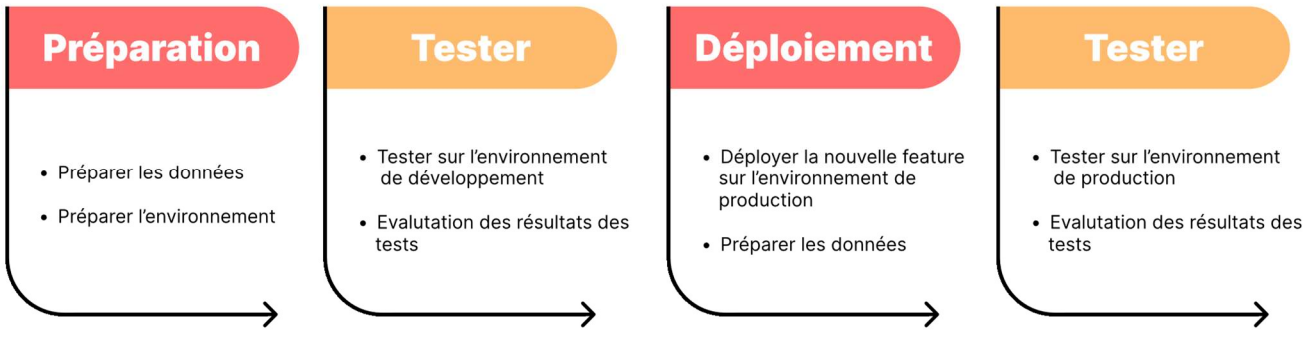
Fournir tous les document de conception:

- *le choix du matériel HW*
- *le choix des systèmes d'exploitation pour la réalisation et l'utilisation*
- *le choix des outils logiciels pour la réalisation et l'utilisation*
- *site web: réaliser les maquettes avec un logiciel, décrire toutes les animations sur papier, définir les mots-clés, choisir une formule d'hébergement, définir la méthode de mise à jour, …*
- *bases de données: décrire le modèle relationnel, le contenu détaillé des tables (caractéristiques de chaque champs) et les requêtes.*
- *programmation et scripts: organigramme, architecture du programme, découpage modulaire, entrées-sorties des modules, pseudo-code / structogramme…*

Le dossier de conception devrait permettre de sous-traiter la réalisation du projet !

2.4 Stratégie de test

Différents tests seront effectués durant ce projet. La stratégie de test adoptée est la suivante :



Lorsqu'une nouvelle fonctionnalité a fini d'être développée, nous allons d'abord être sûr que les données requises sont prêtes. Nous allons aussi nous assurer que notre environnement est aussi prêt (c'est-à-dire que notre environnement est démarré avec la dernière fonctionnalité disponible). Je vais d'abord effectuer la liste des tests pour cette fonctionnalité sur l'environnement de développement. Puis les tests seront évalués selon les résultats attendus. Si tous les tests passent. La fonctionnalité sera déployée sur l'environnement de production. Si elle ne passe pas, une analyse des bugs sera faite afin de connaître ce qu'il faut corriger. Une fois que la *feature* est déployée sur l'environnement de production. Etant donné que cette environnement est différent de celui de développement, je vais réeffectuer la liste de test et réévaluer les résultat. Si tout est bon, la fonctionnalité sera déclarée comme « passée ». Sinon il faudra analyser l'erreur. Il faudra tout de même faire attention a ce qu'une nouvelle *feature* n'en casse pas une autre. Refaire rapidement les tests des autres *features* présentent sur le site sera une chose à faire.

Les tests seront effectué par moi-même au cours du projet

Je suis conscient qu'effectuer des tests sur l'environnement de production n'est pas l'idéal. Il serait préférable d'avoir un environnement de test non accessible aux utilisateurs et similaire à l'environnement de production dédie uniquement aux différents tests.

Décrire la stratégie globale de test:

- *types de des tests et ordre dans lequel ils seront effectués.*
- *les moyens à mettre en œuvre.*

- *couverture des tests (tests exhaustifs ou non, si non, pourquoi ?).*
- *données de test à prévoir (données réelles ?).*
- *les testeurs extérieurs éventuels.*

2.5 ***Risques techniques***

- *risques techniques (complexité, manque de compétences, …).*

Décrire aussi quelles solutions ont été appliquées pour réduire les risques (priorités, formation, actions, …).

2.6 Planification

Révision de la planification initiale du projet :

- *planning indiquant les dates de début et de fin du projet ainsi que le découpage connu des diverses phases.*
- *partage des tâches en cas de travail à plusieurs.*

Il s'agit en principe de la planification ***définitive du projet****. Elle peut être ensuite affinée (découpage des tâches). Si les délais doivent être ensuite modifiés, le responsable de projet doit être avisé, et les raisons doivent être expliquées dans l'historique.*

3 Réalisation

3.1 Point de design spécifique

3.1.1 Gestion de l'authentification

Pour la gestion de l'authentification, j'ai choisi d'utiliser le modèle *RBAC*[7]. C'est-à-dire que chaque utilisateur à un rôle attribué et que les fonctionnalités ainsi sont accessible dépendamment du rôle de l'utilisateur.

L'idée du système d'authentification implémenté est d'avoir l'utilisateur connecté stocké dans le *local storage* du navigateur. Cela permettra à l'utilisateur de toujours être connecté même après avoir fermé l'onglet. L'application stocke donc un objet *currentUser* contenant l'identifiant naturel de l'utilisateur (son nom d'utilisateur) ainsi qui son rôle. Le fait d'avoir l'utilisateur stocké permet d'éviter de devoir faire des requêtes à la base de données à chaque fois qu'il change de page et qu'on doit récupérer son rôle.

La logique d'authentification de l'application peut se représenter sous cette forme :

[7] Role-Based Access Control

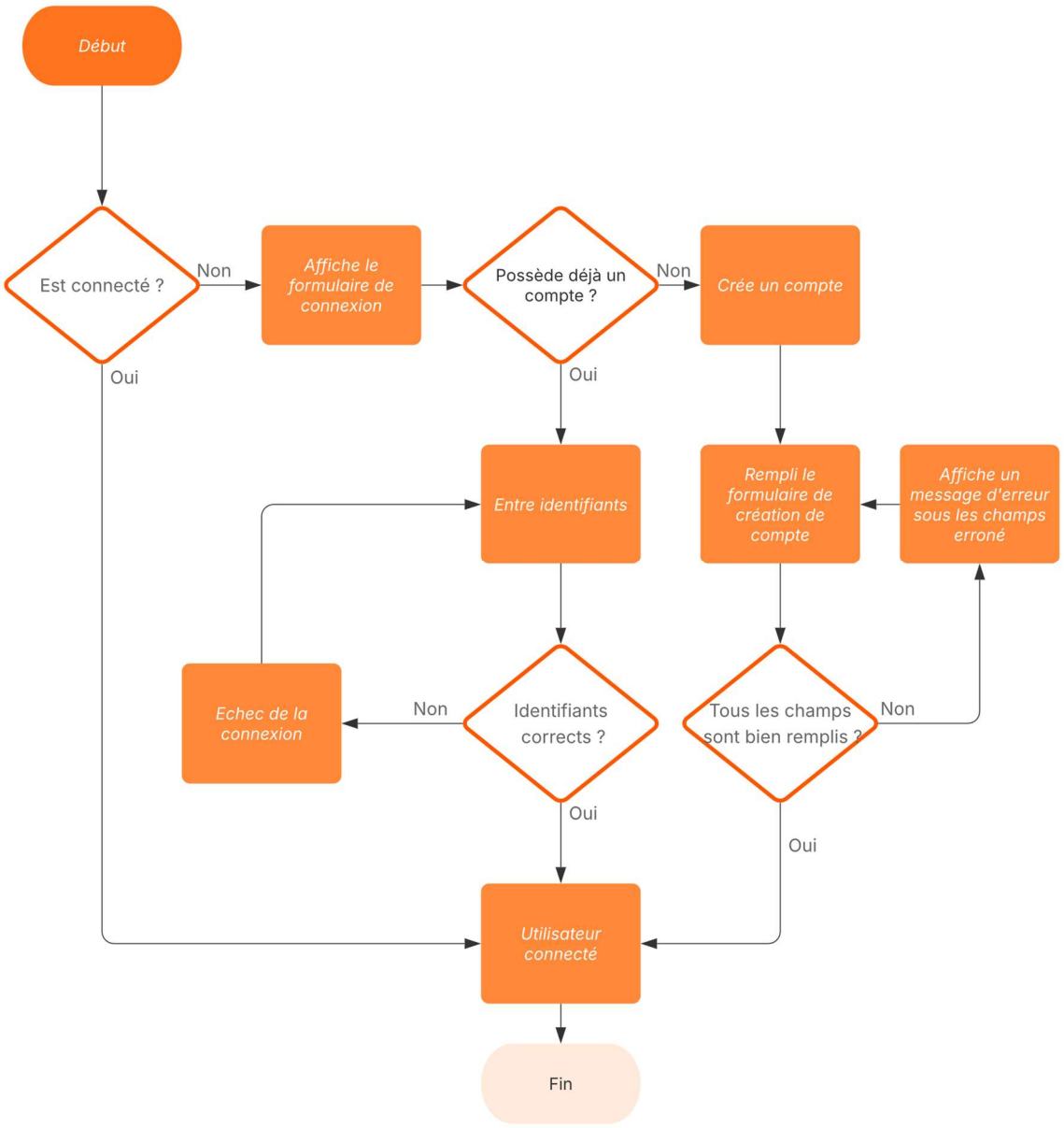


Figure 9: Logigramme authentification

- L'application va d'abord vérifier dans le *local storage* si un objet utilisateur est stocké. Si c'est le cas, alors c'est que l'utilisateur s'était connecté auparavant et donc l'application lui affiche la page principale.
- S'il n'est pas connecté, l'application lui affiche la page de login avec un formulaire de connexion.

- Une fois que l'utilisateur a entré ses identifiants et validé, une requête à la base de données et faite et vérifie si les identifiants sont corrects.
    - Si c'est le cas, l'application stocke le nom d'utilisateur et son rôle dans un objet dans le *local storage* puis lui affiche la page principale.
    - Si ce n'est pas le cas, un message d'erreur est affiché.

Mon cahier des charges ne demande pas de gestion de création de compte. Le schéma montre donc uniquement comment la création d'utilisateur serait gérée mais elle n'a pas été implémentée.

Pour cette gestion d'authentification, j'ai dû mettre en place un contexte d'authentification. Ce contexte possède plusieurs fonctions qui seront utilisées dans la gestion de l'authentification.

3.1.1.1 GetUser

```
4   export async function getUser() {
5
6     const authToken = localStorage.getItem("userToken")
7
8     if (authToken) {
9
10      const user = JSON.parse(atob(authToken.split('.')[1]));
11
12      return [200, { user }] as const;
13    }
14
15    return [401, { authToken: null, user: null }] as const;
16  }
```

La fonction *GetUser* permet de vérifier s'il y a un token d'utilisateur stocké dans le *local storage* et de retourner le contenu de l'objet dans une variable (*currentUser*) qui sera

utilisée lors de la vérification des routes, etc… Si rien n'est trouvé, alors l'application défini l'utilisateur comme déconnecté et met à jour les différentes variables.

3.1.1.2 HandleLogin

La fonction *HandleLogin* va, comme l'indique son nom, gérer la connexion. Elle va d'abord envoyer une requête POST à la base de données pour récupérer les données de l'utilisateur correspondant au nom d'utilisateur entré dans le formulaire. Elle va récupérer le mots de passe hashé stocké dans la base de données puis va comparer avec le mot de passe entré dans le formulaire. Pour hasher le mot de passe et pour la comparaison, j'utilise la dépendance « *becrypt* ».
Si les mots de passes correspondent, les données récupérées (le nom d'utilisateur et son rôle) puis sont stockées dans l'objet *currentUser.*
Si l'utilisateur rafraîchi la page, l'objet n'est pas sauvegardé. Pour régler ce problème, je vais stocker l'objet de l'utilisateur dans le local storage. Au début je pensais simplement laisser l'objet comme ça dans le local storage. Le problème était que n'importe quel utilisateur peut modifier le contenu de cet objet et donc peut se donner les droits administrateur. J'ai donc décidé de chiffrer l'objet en utilisant un JWT[8]. Avec ceci, l'utilisateur ne peut pas modifier son rôle et donc ne peut pas passer d'un utilisateur simple à un administrateur.

8

```
62  async function handleLogin(email: string, password: string) {
63      try {
64          const response = await login(email, password);
65          const authToken = response[1].token;
66
67          setAuthToken(authToken);
68          setIsUserConnected(true);
69          localStorage.setItem('userToken', authToken);
70          try {
71              const decoded = JSON.parse(atob(authToken.split('.')[1]));
72              setCurrentUser({
73                  id: decoded.id,
74                  login: decoded.email,
75                  role: decoded.role,
76              });
77          } catch (err) {
78              console.error("Erreur décodage token :", err);
79          }
80
81          return null;
82      } catch {
83          setAuthToken(null);
84          setCurrentUser(null);
85          setIsUserConnected(false);
86          return "erreur";
87      }
88  }
```

L'utilisation du Token se fait suivant la logique suivante :

Une clé secrète est stockée dans le fichier .env dans le côté serveur. Lorsque le backend fait une requête à la base de données, il transforme l'objet reçu en retour en JWT pour le renvoyer au frontend qui s'occupe de le stocker dans le local storage. Le frontend va ensuite, lorsqu'il en a besoin, déchiffrer le token pour récupérer les informations de l'utilisateur.

```
44    useEffect(() => {
45        if (authToken) {
46            try {
47                const decoded = JSON.parse(atob(authToken.split('.')[1]));
48                setUserRole(decoded.role);
49            } catch (e) {
50                console.error("Erreur lors du décodage du token :", e);
51            }
52        }
53    }, [authToken]);
```

Si le token est manuellement modifié, l'utilisateur est automatiquement déconnecté afin de regénérer un nouveau token valide.
Voici notre objet utilisateur qui sera stocké :

```
{
  "id": 2,
  "email": "admin",
  "role": "admin",
  "iat": 1747833318,
  "exp": 1747840518
}
```

Et voici à quoi cela ressemble au format JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZ
CI6MiwiZW1haWwiOiJhZG1pbiIsInJvbGUiOiJhZG1
pbiIsImlhdCI6MTc0NzgzMzMxOCwiZXhwIjoxNzQ3O
DQwNTE4fQ.XoFc4xsgIJaVF7BhVVF-
FqkItWGPXOg7_jB0z04tUUM
```

3.1.1.3 HandleLogout

La gestion de déconnexion est plutôt simple. Elle va faire appel en premier lieu à la fonction *getUser* dont j'ai décrit le fonctionnement tout à l'heure. Elle va donc récupérer le token de l'utilisateur actuellement connecté puis va le supprimer. Comme la fonction

getUser défini l'état connecté/déconnecté selon s'il trouve ou non un token utilisateur dans le *local storage*, il va donc donner désormais l'état déconnecté.

3.1.2 Gestion des routes frontend

En react, les différentes routes représentent les pages ou les composants affichés à l'écran selon l'URL de l'application. Par exemple si l'URL */mainPage* est actif, la page affichée sera la page principale. Ces distinctions se font grâce à la dépendance « *react-dom-router* ». C'est aussi avec ceci que la gestion des accès aux pages se feront. J'ai créé un composant *ProtectedRoute* qui prend en *props* [9]un tableau des rôles qui seront autorisé à accéder à cette route. Si un utilisateur qui n'a pas le rôle permettant d'accéder à cette route essaie de s'y rendre en rentrant directement le lien de la page dans l'URL, alors il sera tout de suite redirigé à la page de défaut, dans notre cas la page principale. Dans ce projet, actuellement, cette redirection n'a pas grande utilité car il n'y a pas de page interdite à un type d'utilisateur, que des fonctionnalités affichées ou non selon le type. Cependant, avoir ceci d'implémenter permet un meilleur scalabilité de l'application et permettra d'ajouter de nouvelle fonctionnalité et de gérer ses accès beaucoup plus facilement. Cela permettra aussi de rajouter des rôles si besoin comme, par exemple, un utilisateur de type *pilote-reader* qui aura le droit de voir que la page des pilotes. C'est aussi dans ce composant que l'on redirige un utilisateur non-identifié à la page de login. Si l'objet *currentUser* est *null*, alors cela signifie qu'il n'y a pas d'objet user dans le *local storage*. Comme vu précédemment, cela veut donc dire que l'utilisateur est déconnecté est sera donc automatiquement redirigé au formulaire de connexion. Il ne lui sera pas possible d'aller sur d'autre page.

Finalement, voici à quoi ressemble une définition de route :

[9] Argument passé à un composant

```
25    <Route path="/mainPage" element={
26      <ProtectedRoute allowedRoles={['admin', 'user']}>
27        <MainPage />
28      </ProtectedRoute>} />
```

Figure 10: Exemple d'une route

On retrouve bien là l'URL de la route, le composant de protection de route qui a comme *props* le tableau des rôles autorisés à accéder au contenu et en enfant de ce composant le composant de la page principale.

Une route importante à définir est celle par défaut. Si par exemple l'utilisateur entre dans l'URL la route */cettePageNExistePas* alors il sera redirigé sur la page principale. La protection de routes fera aussi effet sur cette redirection.

3.1.3 Gestion des pilotes / écuries / circuits

La gestion des pilotes, des écuries et des circuits sont 3 points techniques relativement similaires. J'ai donc décidé de les regrouper en un seul chapitre et d'expliquer son fonctionnement en prenant la gestion des pilotes comme exemple.

Les seules différences seront au niveau de l'affichage des *cards* et du nombre de champs dans les formulaires de création et de modification. Mais les fonctionnalités d'ajout, de modification et de suppression ont le même fonctionnement.

3.1.3.1 Composition de la page

Le point positif avec React c'est que la composition des pages avec ses composants ressemble beaucoup au zoning de la page. J'ai donc repris globalement le zoning réalisé en début de projet pour gérer la composition de la page. Finalement, la composition de la page ressemble à ça :

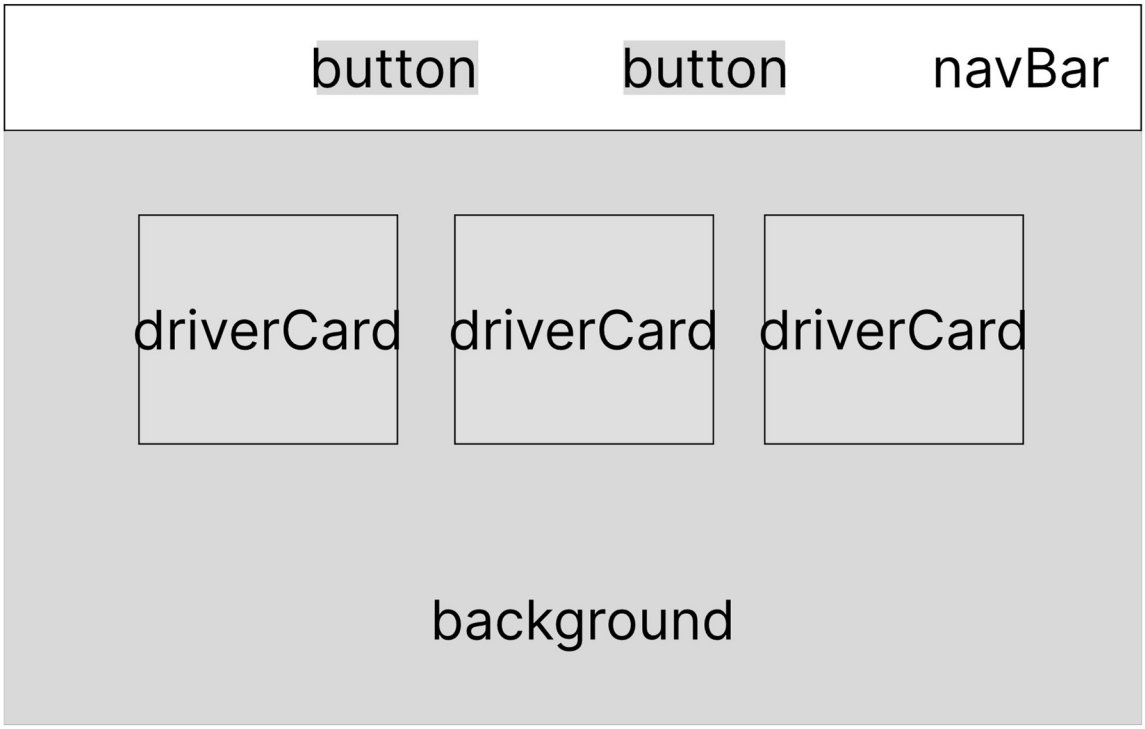


Figure 11: Composition composants liste pilotes

Un autre point positif de React est son architecture basée sur les components. Cela permet de réaliser un seul composant de type *card*, et de le réutiliser autant de fois que nécessaire. Ainsi, qu'on ait 1 ou 45 pilotes dans notre base de données, on a juste à reprendre les données et les parcourir avec une boucle (*.map*) pour afficher tous nos pilotes.

Lorsque l'utilisateur accède à la page */drivers*, l'application envoie une requête http à l'URL */api/drivers* (donc le côté backend). Le *controller* associé à cette route appelle la fonction *getAllDriver*. La fonction va ensuite faire une requête SQL pour récupérer la liste des pilotes. Le *controller* va retourner le résultat de la requête au frontend.
On retrouve ici le fonctionnement *client-server* décrit quelques chapitres auparavant expliquant comment le lien entre *backend* et *frontend* se fait. On a donc maintenant un tableau de pilote qu'on va pouvoir afficher.

Dans le frontend, j'ai ajouté un fichier « interface ». Ce fichier contient la déclaration des types personnalisés qui seront utilisé dans l'application. Par exemple, *driverInterface* décrit la structure d'un objet « pilote » :

```
16  export type driverInterface = {
17      id: number;
18      firstname: string;
19      lastname: string;
20      rating: number;
21      nationality: string;
22      team: string;
23      pictureLink: string;
24      countryFlag: string;
25  }
```

Figure 12: Interface d'un objet pilote

Cela permet d'avoir un développement plus fiable et plus assisté (proposition des paramètres par VSC[10]).

L'application va d'abord afficher le composant *navBar* avec comme onglet actif « Pilotes » donc sera mis en rouge.
Elle va ensuite récupérer le rôle de l'utilisateur actuellement connecté. Si son rôle est « admin » alors l'application va y afficher les boutons de création de pilotes et de modification de pilotes.
Pour l'affichage des *cards*, on va parcourir le contenu du tableau de pilotes et, pour chaque pilote, afficher un composant avec le pilote comme *props* :

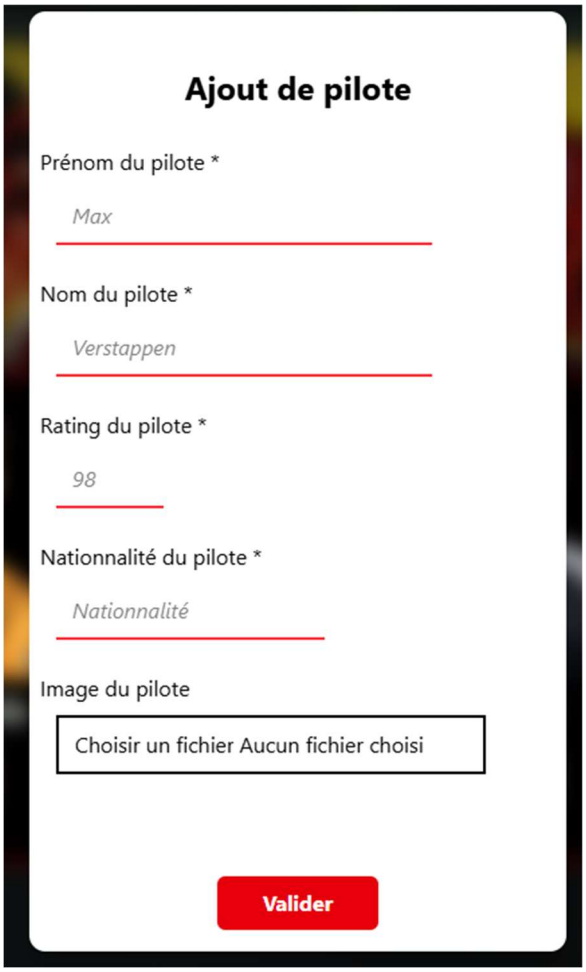
[10] Visual Studio Code

```
90   {driverList.map((driver) => (
91       <>
92           <div className="w-fit h-fit border-2 rounded-2xl border-red-700
93               <DriverCard driver={driver} onEdit={handleEditDriver} />
94           </div>
95       </>
96   ))}
```

Figure 13: Affichage de la liste des pilotes

3.1.3.2 Ajout d'un pilote

Un bouton visible uniquement par les administrateurs permet de faire apparaître un formulaire de création de pilote. Le formulaire contient des inputs pour entrer le nom, le prénom ainsi que le rating du pilote, une liste déroulante pour sélectionner un pays et un input pour sélectionner un photo. Cette dernière partie n'a pas été implémentée. Cependant, pour chaque nouveau pilote, une image de profil par défaut est insérée.



Une fois que l'administrateur valide chacun des champs du formulaire, l'application va vérifier si les champs obligatoires (nom, prénom, rating, nationalité) sont bien remplis, si c'est le cas, alors il envoie une requête au backend qui, lui, va ajouter le pilote dans la base de données. Si tous les champs obligatoires ne sont pas remplis, alors un message d'erreur apparaît et rien ne se passe.

3.1.3.3 Modification d'un pilote

Lorsqu'un administrateur se rend sur la page des pilotes, un logo de modification apparaît à côté de chaque pilote. S'il clique dessus, alors un formulaire similaire à celui de création apparaît à la seul différence que les champs sont déjà préremplis avec les données du pilote que l'administrateur veut modifier.

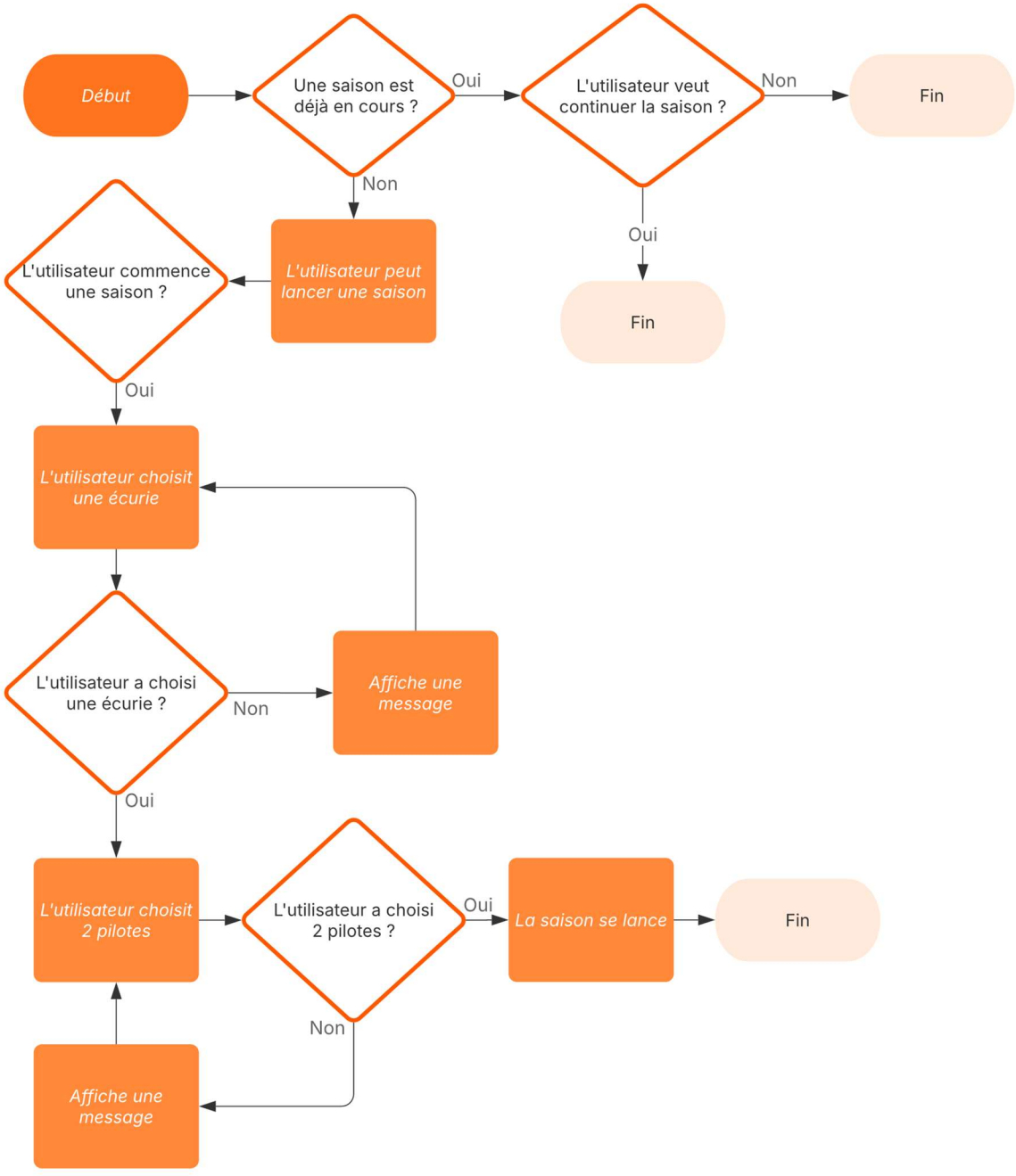
La vérification des données est la même que pour le formulaire d'ajout et la modification des données dans la db par le backend est gérée de la même façon.

3.1.3.4 Suppression d'un pilote

3.1.4 Gestion de jeu

Le jeu est la fonctionnalité principale de l'application. Pour démarrer une partie, il faut d'abord être authentifié pour avoir accès à la page principale où se lance la partie. Pas besoin d'être administrateur, un simple utilisateur peut jouer.
Un bouton permettant de lancer la partie se trouve sur la page principale. Il s'y trouve aussi sur cette page un bouton pour reprendre une partie dans le cas ou l'utilisateur avait déjà lancé une partie auparavant et que les données sont toujours sauvegardées dans le local storage.

Avant d'entrer dans les détails de chaque étape du jeu, voici un schéma décrivant la logique du lancement du jeu :



Le schéma sera expliqué au cours de prochains chapitres.

3.1.4.1 Choix de l'écurie et des pilotes

Lorsqu'un utilisateur choisi de démarrer une nouvelle saison, il accède à la page */game*. La première chose qui est affiché à l'utilisateur est la page de sélection de l'écurie. C'est une page ressemblant à la page des écuries. Le joueur choisi son écurie en cliquant dessus. L'écurie choisie est donc encerclée en rouge. Le joueur est obligé de choisir une et une seule écurie pour continuer.

Lorsque le joueur accède à cette page, l'application stocke une valeur *inGame : true* dans le local storage. Cela permet de savoir si l'utilisateur à déjà lancé une partie ou non.

Une fois l'écurie choisie, le joueur choisi ses 2 pilotes. Exactement 2 pilotes doivent être choisi pour que l'utilisateur puisse continuer. Pour gérer la sélection des pilotes, j'ai mis en place un système de *First In, First Out.* C'est-à-dire :

- Choisi pilote N°1 en cliquant sur un pilote
- Choisi pilote N°2 en cliquant sur un autre pilote
- Si sélectionne un 3ème pilote, le pilote N°1 (qui est le pilote les plus « vieux » dans les pilotes sélectionnés) est remplacé par ce nouveau pilote

```
59  const handleDriverSelection = (driver: driverInterface) => {
60
61      //FIFO
62      setSelectedDrivers((prev) => {
63          if (prev.find(d => d.id === driver.id)) {
64              return prev.filter(d => d.id !== driver.id);
65          }
66          if (prev.length < 2) {
67              return [...prev, driver];
68          }
69          return [prev[1], driver];
70      });
71  };
```

Si l'utilisateur clique sur un pilote déjà sélectionné, alors il est désélectionné.

Lorsque le joueur a choisi ses 2 pilotes, alors la saison peut commencer. Pour cela, l'application appelle la fonction *StartSeason()* qui appartient au contexte du jeu. C'est

cette fonction aussi qui va répartir les pilotes restants dans les écuries de façon aléatoire.
Pour ça, elle va envoyer une requête au backend pour récupérer la liste des pilotes et des écuries. Elle va ensuite filtrer le tableau reçu afin d'enlever les pilotes choisis par le joueur. Ensuite, pour chaque écurie, on va vérifier si c'est l'écurie que le joueur à choisi, si c'est le cas alors on y ajoute les 2 pilotes sélectionnés, sinon, on va chercher 2 pilotes en avec un index aléatoire puis on modifie, dans la base de données, le champs *fkTeam* avec l'id de l'écurie dans laquelle il vient d'être mis.
Sachant qu'un administrateur peut ajouter des pilotes, il se peut qu'il y ait plus de pilotes que de places disponibles. Pour cela, les pilotes restants sont aussi modifiés et leur champ *fkTeam* est modifié avec la valeur *null.* Lorsque la saison commence, les pilotes ayant *null* comme valeur ne seront pas pris en compte.

```
82  const tempTeamArray: teamInterface[] = resTeams.data;
83  // ForEach team
84  tempTeamArray.forEach(team => {
85    // if the id matches the selected team id, edit the driver's team assigned
86    if (team.id === selectedTeam?.id) {
87      for (const driver of selectedDrivers) {
88        axios.post("/api/drivers/updateTeam", {
89          idDriver: driver.id,
90          idTeam: selectedTeam.id,
91        }).catch(err => {
92          console.error(`Erreur lors de l'affectation de ${driver.firstname}`, err);
93        });
94      }
95      return
96    }
97
98    const teamDrivers: driverInterface[] = [];
99    // 2 drivers per team
100   for (let i = 0; i < 2; i++) {
101     if (tempDriverArray.length === 0) break;
102     // select a random number, this will be the index of the driver to assign to the team
103     const randomIndex = Math.floor(Math.random() * tempDriverArray.length);
104     const selectedDriver = tempDriverArray[randomIndex];
105
106     axios.post("/api/drivers/updateTeam", {
107       idDriver: selectedDriver.id,
108       idTeam: team.id,
109     }).catch(err => {
110       console.error(`Erreur lors de l'affectation de ${selectedDriver.firstname}`, err);
111     });
112     teamDrivers.push(selectedDriver);
113     tempDriverArray.splice(randomIndex, 1);
114   }
115 });
116 // drivers that are leftover are set to null
117 axios.post("/api/drivers/updateTeam", {
118   idDriver: tempDriverArray[0].id,
119   idTeam: null,
120 }).catch(err => {
121   console.error(`Erreur lors de l'affectation de ${tempDriverArray[0].firstname}`, err);
122 });
```

Le local storage est aussi mis à jour avec des nouvelles valeurs :

- *gameState : start* Cette variable permet de définir dans quelle état est la partie. Cela permettra de reprendre une partie là où l'utilisateur l'avait laissé.
- *gameSettings* Cet item stock un objet qui contient l'écurie choisie par le joueur et les 2 pilotes choisis ainsi que le numéro de la course.

3.1.4.2 Simulation de course

3.1.4.3 Gestion des classements

3.1.4.4 Fin de la saison

3.2 Dossier de réalisation

Décrire la réalisation "physique" de votre projet

- *les répertoires où le logiciel est installé*
- *la liste de tous les fichiers et une rapide description de leur contenu (des noms qui parlent !)*
- *les versions des systèmes d'exploitation et des outils logiciels*
- *la description exacte du matériel*
- *le numéro de version de votre produit !*
- *programmation et scripts: librairies externes, dictionnaire des données, reconstruction du logiciel - cible à partir des sources.*

NOTE : Evitez d'inclure les listings des sources, à moins que vous ne désiriez en expliquer une partie vous paraissant importante. Dans ce cas n'incluez que cette partie…

3.3 Description des tests effectués

Pour chaque partie testée de votre projet, il faut décrire:

- *les conditions exactes de chaque test*
- *les preuves de test (papier ou fichier)*

- *tests sans preuve: fournir au moins une description*

3.4 Erreurs restantes

S'il reste encore des erreurs:

- *Description détaillée*
- *Conséquences sur l'utilisation du produit*
- *Actions envisagées ou possibles*

3.5 Liste des documents fournis

Lister les documents fournis au client avec votre produit, en indiquant les numéros de versions

- *le rapport de projet*
- *le manuel d'Installation (en annexe)*
- *le manuel d'Utilisation avec des exemples graphiques (en annexe)*
- *autres…*

4 Conclusions

Développez en tous cas les points suivants:

- *Objectifs atteints / non-atteints*
- *Points positifs / négatifs*
- *Difficultés particulières*

- *Suites possibles pour le projet (évolutions & améliorations)*

5 Annexes

5.1 Résumé du rapport du TPI / version succincte de la documentation

5.2 Sources – Bibliographie

Liste des livres utilisés (Titre, auteur, date), des sites Internet (URL) consultés, des articles (Revue, date, titre, auteur)… Et de toutes les aides externes (noms)

5.3 Journal de travail

Date	Durée	Activité	Remarques

5.4 Manuel d'Installation

5.5 Manuel d'Utilisation

5.6 Archives du projet

Media, … dans une fourre en plastique

6 Table des illustrations

7 Glossaire