RUNEO DRIVE

Clément Sartoni - 2023

Résumé

Projet de préparation au TPI consistant en la continuation du projet Runeo Drive, déjà débuté au P_Appro_1 et permettant aux chauffeurs du Paléo de recevoir les informations des courses à effectuer

Clément Sartoni

Table des matières

1. Analyse préliminaire

1.1. Introduction

Ce projet a été initié par M. Carrel il y a 4 ans pour répondre aux besoins d'organisation des chauffeurs du Paléo. Il a été repris par la classe min-mid 4 de l'etml dans le cadre de la préparation au TPI, et les différentes tâches à réaliser ont été réparties entre les différents membres de la classe grâce à une organisation agile et une séparation en user stories.

L'application a déjà fait ses preuves et est déjà utilisée au paléo depuis plusieurs années. Elle est séparée en deux parties : l'application de bureau utilisée par les coordinateurs pour planifier les trajets, et l'application mobile utilisée par les chauffeurs pour recevoir les informations et prendre en charge les trajets.

Ce rapport détaillera l'analyse et la réalisation des tâches à la charge de Clément Sartoni, concernant toutes l'application mobile, Runeo-Drive. Chaque partie sera donc divisée pour parler de chaque tâche séparément.

1.2. Objectifs

Améliorations de l'application Runeo-Drive existante selon 3 axes :

1.2.1. Restreindre le choix de véhicule

Souvent (mais pas toujours), le type de véhicule est défini lors de la publication d'un run. Si c'est le cas, dans l'application actuelle, l'application propose de choisir un véhicule parmi tous les véhicules existants.

Il faut que ce choix soit restreint aux seuls véhicules du type fixé.

1.2.2. Adresse du backend

Il faut une liste déroulante qui permet de choisir avec le nom du festival et laisser une option "Autre..." qui permet d'introduire une valeur ('localhost:8000' par exemple).

1.2.3. Autres runs de mon artiste

Il arrive fréquemment qu'à la fin d'un run les personnes transportées posent des questions au chauffeur par rapport au retour à l'hôtel - par exemple - ou par rapport aux transports du lendemain.

Le chauffeur ne dispose pas (actuellement) d'un moyen efficace de répondre, il doit souvent dire à la personne de s'adresser au bureau des chauffeurs.

On veut donc qu'il puisse avoir un accès facile et rapide à l'information grâce à son app

1.2.4. Planification initiale

Le projet a débuté le lundi 20 mars et se terminera le vendredi 28 avril 2023.

Il y a deux semaines de vacances prévues du lundi 10 au vendredi 21 avril 2023, ainsi que deux jours d'absence supplémentaires : le lundi 27 mars, où je suis absent pour des raisons personnelles, et le vendredi 7 avril qui est un jour férié.

Dans une semaine de travail normale, le temps disponible pour travailler est organisé comme indiqué dans ce tableau :

Jour	Lundi	Mardi	Mercredi	Jeudi	Vendredi	Total
Périodes	8	0	9	4	9	30
Heures	6h	0h	6h45	3h	6h45	22h30

En prenant donc en compte les vacances et les jours de congé, il y a à disposition pour ce projet 103 périodes ou 77 heures et 15 minutes.

1.2.5. Découpe en sprints

Vu le peu de temps à disposition pour ce projet, il a été choisi d'effectuer des sprints d'une semaine afin d'avoir un suivi plus pertinent et efficace. Cela a le désavantage de passer plus de temps sur des aspects organisationnels mais cela est nécessaire pour ne pas manquer de retours sur le travail qui est en cours, et si le travail de gestion des sprints est fait efficacement le coût se limite à quelques heures par semaine. Les sprints reviews se feront le jeudi à 15h00, et les sprints commenceront donc le vendredi matin de chaque semaine. Le dernier vendredi sera consacré à l'impression du rapport et aux dernières retouches qui seront potentiellement discutées lors de la dernière sprint review.

1.2.5.1. Sprint 1

Les objectifs de ce sprint sont les suivants :

- L'analyse préliminaire est terminée.
- Les sprints sont définis sur IceScrum ainsi que dans la documentation, et les dates des sprint reviews sont définies.
- Le journal de travail reflète bien la réalité des activités et les bonnes habitudes sont instaurées pour rester durant les sprints suivants.
- Le product backlog contient suffisamment de stories pour pouvoir procéder au planning des sprints suivants. Ceci inclut pour chaque user story une description détaillée, les tests d'acceptance ainsi qu'une première ébauche des tâches à réaliser.

La sprint review se fera le jeudi 23 mars 2023 à 15h00.

1.2.5.2. Sprint 2

L'objectif de ce sprint sera principalement de réaliser la user story concernant l'implémentation du choix de l'adresse du backend lors de l'ouverture de l'application. Celle-ci a été choisie comme première tâche car c'est potentiellement la plus complexe à réaliser et que la fonctionnalité qu'elle apporte est essentielle.

Il faudra aussi prévoir de documenter la réalisation et les points de design spécifiques de cette user story.

La sprint review se fera le jeudi 30 mars 2023 à 15h00.

1.2.5.3. Sprint 3

L'objectif de ce sprint sera de réaliser la user story consistant à créer la fonctionnalité « Autres runs de mon artiste » et de commencer la dernière, ayant pour but de restreindre le choix du véhicule. Ces user stories sont un peu moins complexes car il est probable que certaines pages déjà existantes puissent être adaptées aux nouveaux besoins assez simplement. Tout comme pour la fonctionnalité précédente, il faudrait aussi avoir déjà documenté la réalisation ainsi que les points de design spécifiques.

La sprint review se fera le jeudi 6 avril 2023 à 15h00.

1.2.5.4. Sprint 4

Pour ce dernier sprint, l'objectif sera de finaliser la user story de restriction du choix du véhicule ainsi que de terminer la documentation.

La sprint review se fera le jeudi 27 avril 2023 à 15h00, afin d'avoir encore une journée de travail pour effectuer des retouches avant le TPI si nécessaire.

2. Analyse / Conception

2.1. Analyse fonctionnelle

2.1.1. Autres runs de mon artiste

En tant que chauffeur, je veux accéder rapidement à la liste des runs de l'artiste que je transporte pour répondre à questions.

Tests d'acceptance :

Présence du bouton	Quand on est sur la page d'un run, juste après les informations sur les runners, se trouve un bouton intitulé "Autres runs de l'artiste", comme sur la maquette de la page d'un run (Figure 1).
Clic sur le bouton	Sur la page d'un run, quand on clique sur le bouton "Autres runs de l'artistes", on est redirigé vers une page contenant tous les runs de l'artiste, comme sur la maquette "Autres runs de l'artiste" (Figure 2).
Clic sur un run	Sur la page "Autres runs de l'artiste", quand on clique sur un run, la page du run en question s'ouvre correctement et le bouton "autres runs de l'artiste" n'est plus visible.
Retour depuis un run	Sur la page du run ouvert après qu'on ait cliqué dessus sur la page "autres runs de l'artiste", quand on clique sur la petite flèche de retour en haut à droite, on retourne bien en arrière sur la page "autres runs de l'artiste".

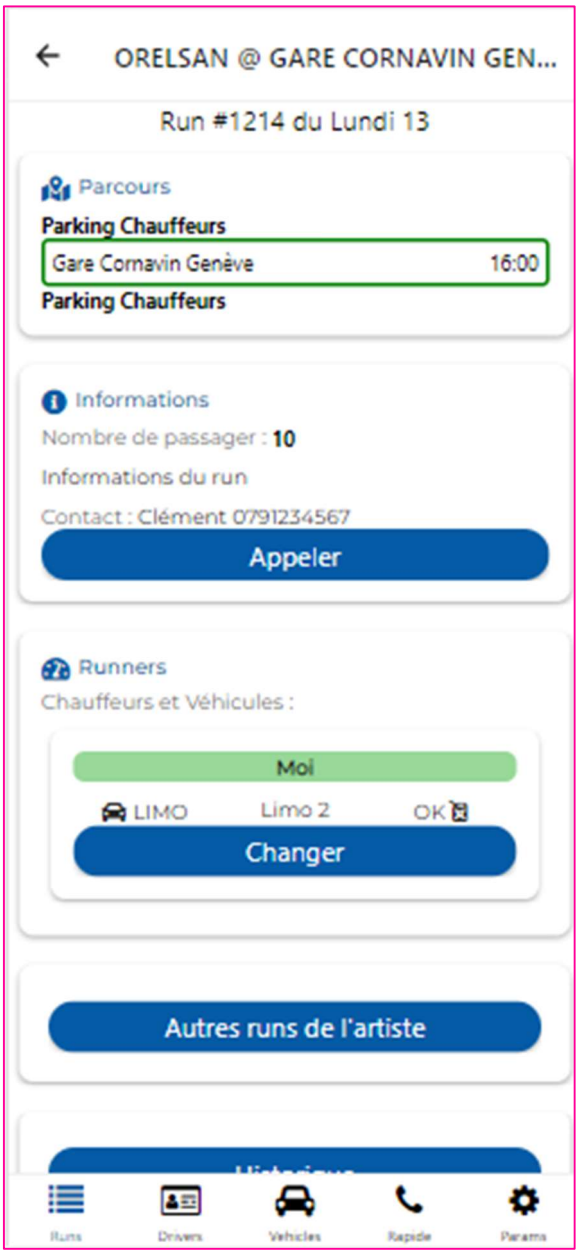

Figure 1 : Maquette de la page d'un run

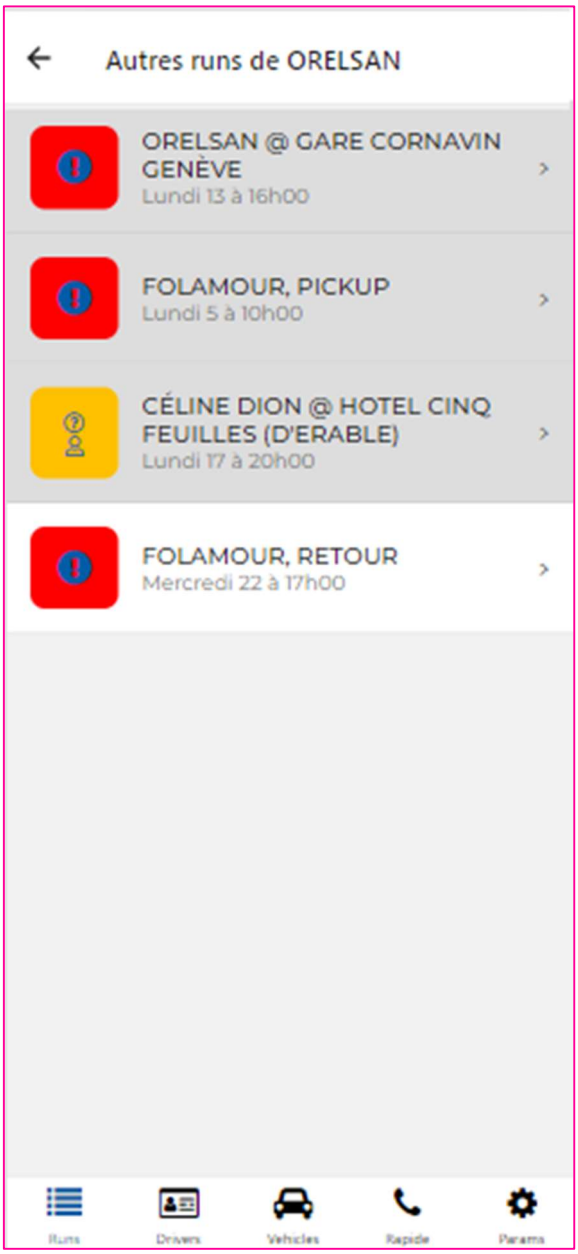

Figure 2 : Maquette de la page "Autres runs de l'artiste"

2.1.2. Adresse du backend

En tant que chauffeur, je veux pouvoir choisir à quel backend mon application se connecte pour pouvoir travailler à Belfort et à Paléo sans changer d'app.

Tests d'acceptance :

Présence de la dropdown	Sur la première page de l'application permettant de se connecter, en dessus du champ permettant d'entrer le token, une liste déroulante est présente et accepte les valeurs "Paléo", "Belfort" ou "autre". Paléo est sélectionné par défaut
Champ autre	Sur la première page, quand la valeur "autre" est sélectionnée dans la liste déroulante, un champ texte apparait au-dessous et permet de rentrer un URL.
Connexion à la bonne API	Sur la première page, quand le festival ou l'URL et un token valide sont renseignés et que l'on clique sur le bouton "connexion", les runs du festival sélectionné apparaissent.
Erreur réseau	Sur la première page, alors que le smartphone ne peut pas atteindre le backend Quand on clique sur le bouton "connexion", une erreur avertit l'utilisateur qu'il y a un problème avec son réseau ou le site du festival sélectionné.

2.1.3. Restreindre le choix de véhicule

En tant que chauffeur, je veux que l'app me propose de choisir un véhicule du type fixé pour me simplifier la manipulation et éviter des erreurs.

Test d'acceptance :

Sélection des véhicules	Dans la page d'un run, quand le type de véhicule est fixé, lorsque je clique sur le bouton pour sélectionner le ou changer de véhicule, seuls les véhicules du type fixé apparaissent.

2.2. Concept

Le site web gère la base de données mysql et met à disposition une API pour que l'application mobile puisse récupérer les informations.

Pour mieux comprendre le contexte du travail présenté dans ce document, le fonctionnement de l'application sera détaillé ici au moyen de différents schémas déjà existants. Ils ont été créés lors des étapes antérieures du projet par d'autres personnes.

Le but premier de l'application est de gérer les runs, ou courses, qui sont concrètement les demandes ou besoins de transport des artistes ou des personnes les accompagnant. Ces runs sont planifiés par les coordinateurs soit avant le festival soit pendant, avec toutes les informations nécessaires comme les heures de rendez-vous et le type de véhicule par exemple. Ensuite, ils apparaissent sur la page principale de l'application mobile et les chauffeurs peuvent se désigner pour effectuer les runs tout en sélectionnant leur véhicule.

2.2.1. Diagrammes

À la page suivante se trouve le MCD de la base de données. Les deux tables les plus importantes sont les utilisateurs et les runs.

Le processus complet de gestion d'un run est détaillé en page 8 grâce à un schéma expliquant les différents états par lesquels un run doit passer.

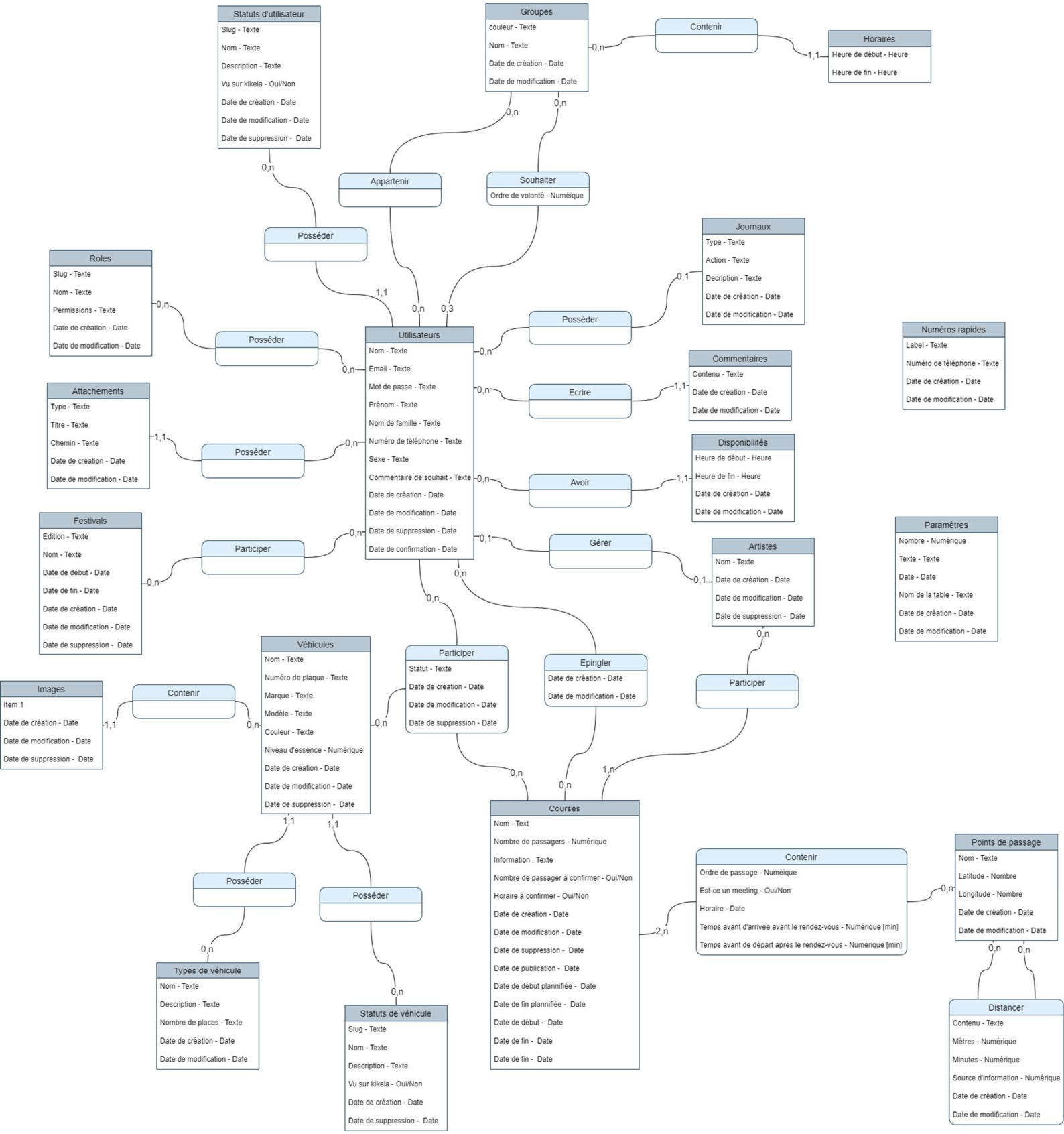

Figure 3 : MCD de la base de données

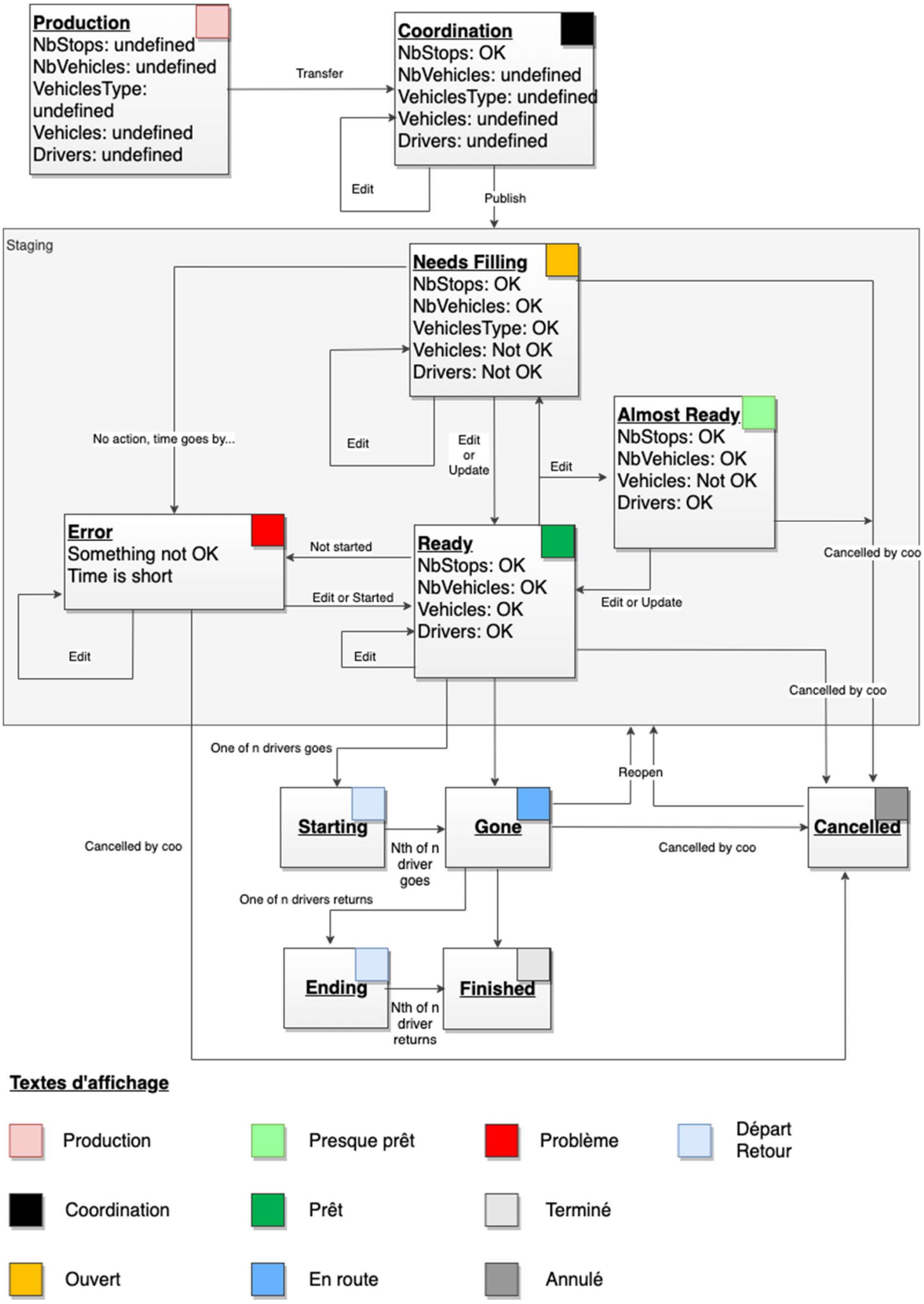

Figure 4: Schéma détaillant les différents états par lesquels passe un run

2.2.2. Fonctionnement de l'application mobile

Les chauffeurs téléchargent l'application depuis le site web ou se font envoyer un lien de téléchargement via WhatsApp. Ils reçoivent aussi de la part des coordinateurs un token de connexion leur permettant de s'identifier sur la première page de l'application et d'avoir accès aux runs.

2.2.3. Arborescence des fichiers de l'application mobile

Voici l'arborescence de tous les fichiers du dossier « src », situé à la racine du projet. C'est ici que sont situés tous les fichiers de code.

```
G:\RUNEO-DRIVE\SRC
    App.component.tsx
    Loader.component.tsx
    Provider.component.tsx
    RootNavigator.ts
    Router.component.tsx

├──auth
        Auth.component.tsx
        TokenAuth.component.tsx

├──common
    ├──component
        │   ButtonComponent.tsx
        │   InfoComponent.tsx
        │   ListCommonResource.component.tsx
        │   StatusCircle.component.tsx
        │   TextInput.component.tsx
        │   UpdatedInfoCard.component.tsx
        │   Card.component.tsx
        │   IconButtonComponent.tsx
        │   Dropdown.tsx
        │
        └──text
                ImportantText.component.tsx
                InlineText.component.tsx
                MeetingText.component.tsx

    ├──container
            CacheStatus.container.ts
            DataContainer.interface.ts
            FastDials.container.ts
            Network.container.ts
            Runs.container.ts
            Users.container.ts
            Vehicles.container.ts
            Auth.container.ts

    ├──hook
            Loader.hook.ts
            Run.hook.ts
```

Cela vous permettra de vous orienter dans les fichiers si vous souhaitez voir l'origines des captures d'écran présentes dans ce rapport.

```
    ├──resources
            Comment.resource.ts
            Common.resource.ts
            FastDial.resource.ts
            Log.resource.ts
            Run.resource.ts
            Runner.resource.ts
            User.resource.ts
            Vehicle.resource.ts
            VehicleCategory.resource.ts

    └──utils
            Cache.utils.ts
            CacheHelper.utils.ts
            Color.utils.ts
            Date.utils.ts
            LastUpdatedRun.utils.ts
            List.utils.ts
            Phone.utils.ts
            Run.utils.tsx
            User.utils.tsx
            Jerikan.utils.ts
            Vehicle.utils.tsx

├──fastDials
        FastDials.component.tsx

├──notifications
        ToastNotification.ts

├──params
        Params.component.tsx

├──runs
    │   Runs.component.tsx
    │   RunsEnd.component.tsx
    │   RunsEndPopUpFuil.component.tsx
└──vehicles
    │   Vehicles.components.tsx
    │
    ├──detail
            CommentDetailVehicles.component.tsx
            CreateCommentDetailVehicles.component.tsx
            DetailVehicles.component.tsx
            GasLevelDetailVehicles.component.tsx
            PlateDetailVehicles.component.tsx

    └──list
            ListVehicles.component.tsx
            ListVehiclesView.component.tsx
```

2.3. Stratégie de test

En raison d'un problème concernant le build de l'application, il ne sera pas possible pour ce projet d'appliquer la procédure de test qui serait idéale et devrait normalement être utilisée. Ce chapitre sera donc séparé en deux.

2.3.1. Stratégie de test théorique

Il faudrait normalement installer complètement l'application sur un smartphone Android et sur un iPhone à chaque session de test pour vérifier que les modifications fonctionnent sur un environnement semblable à celui qui sera utilisé en production.

Pour ce faire, il est théoriquement possible de build l'application puis de la distribuer sur un canal de test sur le Play Store (Android) ainsi que sur l'App Store (iOS). Cependant, il est pour cela nécessaire d'avoir auparavant configuré des comptes permettant de publier des applications sur les deux systèmes. Dans l'état actuel, le compte développeur sur le Play Store est presque configuré et prêt à publier l'application, mais il n'y a rien de prévu pour iOS.

2.3.2. Stratégie de test utilisée

À chaque sprint review, M. Carrel effectuera un pull de la branche « develop » de GitHub sur son poste. Ensuite, l'application sera lancée en utilisant l'application « Expo Go » sur son smartphone. Les tests mentionnés dans l'analyse des user stories qui auront été terminées pourront ensuite être effectués.

Changer ainsi d'environnement permet de vérifier que les modifications effectuées dans l'environnement de développement fonctionnent une fois déployées ailleurs.

Concernant les données de test, tant que personne n'utilise la base de données de production de paléo, donc tant que les données de l'année passée n'ont pas été effacées, il est possible de créer des runs de tests depuis le site web afin d'utiliser le même backend qu'en production.

Il est possible que dans le futur la production ne soit plus disponible pour des tests, ce qui impliquerait de devoir lancer en plus de l'application mobile l'application Runeo-Desk localement et d'utiliser une base de données locale dans laquelle des runs de test pourront être créés.

2.4. Risques techniques

Les risques techniques sont assez légers pour ce projet. Le principal est le manque de formation concernant le React et son fonctionnement dans une application en React Native, qui est le framework utilisé dans l'application mobile. Cependant, cela reste un langage connu et relativement maitrisé à la suite d'un projet précédent sur la même application.

Un autre risque potentiel est celui de reprendre du code déjà existant et codé par d'autres personnes. Cela rend plus complexe de prédire combien de temps la réalisation des fonctionnalités va prendre, étant donné que selon la manière dont l'application a été réalisée, il sera possible de plus ou moins réutiliser des éléments ou alors il pourrait ne pas être possible en l'état d'ajouter la fonctionnalité ce qui demanderait de remodeler des fonctionnalités déjà existantes.

Choix techniques

Ces choix techniques ont été effectués lors de la création du projet et aucune modification n'a donc été appliquée à ces derniers.

Outil de gestion des versions	Github
Framework du site web Runeo Desk	Laravel 9.16.0
Framework JS de l'application mobile	React Native version 14.4
Kit de développement logiciel de l'application mobile	Expo SDK version 43

2.5. Points de design spécifiques

2.5.1. Changement de l'adresse du backend

2.5.1.1. Champ URL caché

Pour réaliser la fonctionnalité, utiliser un champ « dropdown » était nécessaire, et il devait accepter une valeur « Autre », qui serait donc entrée via un champ texte.

Plutôt que de gérer les deux champs dans le formulaire, il a été choisi de reporter les valeurs de la dropdown dans le champ texte à chaque changement de sélection. Grâce à une fonction appelée en même temps, le champ texte peut être caché lorsqu'il contient une valeur entrée depuis la dropdown, et affiché uniquement quand l'utilisateur doit lui-même renseigner son URL après avoir sélectionné « autre.

```
//#region dropdown config
const data = urlConfigData;

const [selected, setSelected] = useState(data[0]);

const [urlVisible, setUrlVisible] = useState(false);

function onPress(item: { label: string; value: string }) : void {
    setUrlVisible(item.value == '')
}
//#endregion

const authContainer = AuthContainer.useContainer();
const initialValues = {
    token: "",
    url: selected.value
};
```

Figure 5 : début de la déclaration des variables et fonctions nécessaires au fonctionnement du formulaire (fichier src\auth\TokenAuth.component.tsx)

Ici, on peut observer la déclaration des variables « selected » et « urlVisible », qui utilisent la méthode « useState » de react, permettant de recharger tout ce qui dépend des variables à chaque fois qu'elles sont modifiées en utilisant leur fonction « set ».

Ce fonctionnement est utilisé pour les deux fonctionnalités requises du système : la variable « selected » permet de reporter les données étant sélectionnées dans la dropdown dans le formulaire, en l'utilisant dans le tableau « initialValues ». Et la variable urlVisible définit l'affichage du champ URL en étant définie à chaque fois qu'une nouvelle valeur est sélectionnée (fonction « onPress »).

```jsx
<Formik
    onSubmit={onSubmit}
    validationSchema={
        yup.object().shape({
            token: yup.string().min(5).required(),
            url: yup.string().min(1).required()
        })}
    initialValues={initialValues}
    enableReinitialize={true}>
    {(formik) => (
        <View>
            <Text style={{fontFamily: 'Montserrat-ExtraBold', marginLeft: 10}}>FESTIVAL</Text>
            <Dropdown label={selected.label} data={data} onSelect={setSelected} onPress={onPress}/>

            <View style={urlVisible ? {} : {display: "none"}}>
                <Text style={{fontFamily: 'Montserrat-ExtraBold', marginLeft: 10}}>URL</Text>
                <TextInputComponent
                name={"url"}
                formik={formik}
                inputProps={{
                    placeholder: "Ex: https://192.168.241.121:8000/api"
                }}/>
            </View>
```

Figure 6 : composants JSX du formulaire, de la dropdown et du champ URL (fichier src\auth\TokenAuth.component.tsx)

On peut voir dans cette capture d'écran comment les variables influencent sur les composants.

- Le paramètre « enableReinitialize » renseigné dans le formulaire Formik permet de réinitialiser les valeurs à chaque fois que les « initialValues » sont modifiées.
- Les paramètres « onSelect » et « onPress » de la dropdown permettent à la dropdown, qui a été importée et adaptée depuis un projet tiers récupéré sur internet, de mettre à jour la valeur sélectionnée et de déclencher la fonction onPress.
- Le champ de l'URL et son label sont regroupés dans une View pour pouvoir gérer leur affichage avec un opérateur ternaire dépendant de la valeur « urlVisible ».

Figure 7 : Rendu graphique de la fonctionnalité

2.5.1.2. Données d'url

Les données concernant les différentes options des festivals et des adresses qui y sont associées sont pour l'instant renseignées « en dur » en haut du fichier App.tsx, au même endroit où était auparavant indiqué l'URL auquel l'application se connectait. Les données sont ensuite rapatriées en faisant un « import » du tableau dans le fichier « TokenAuth.component.tsx ».

```
// Le paramétrage concernant les URLs par défaut utilisés par l'application a été transformé sous la forme suivante pour
// pouvoir accepter plusieurs festivals ou pouvoir rentrer une IP personnalisée pour le débug.
// Laisser la valeur de l'option "Autre" vide pour que le champ "url" aparaisse.
export const urlConfigData = [
  { label: 'Paléo', value: 'http://runeo.paleo.ch/api' },
  { label: 'Belfort', value: 'http://belfort.festival.temp/api' },
  { label: 'Autre', value: '' }
];
```

Figure 8 : nouveau paramétrage dans le fichier "App.tsx"

Il suffit donc de modifier les informations du tableau « urlConfigData » pour inclure de nouveaux festivals ou modifier leurs adresses.

2.5.1.3. Gestion des erreurs

Les tests d'acceptance spécifient le besoin de faire une distinction entre les erreurs d'accès à l'url spécifiée et entre celles liées à un token invalide. Auparavant, le code gérait déjà les exceptions mais n'exploitait pas les informations contenues dans les messages liés à celles-ci. Il a donc du fallu faire quelques tests pour analyser les informations à disposition puis modifier légèrement le fonctionnement pour gérer cette donnée.

```
try {
  Axios.defaults.baseURL = url;
  const user = await getAuthenticatedUserApi();
  await AsyncStorage.setItem(USER_STORAGE_KEY, JSON.stringify(user));
  setAuthenticatedUser(user);
} catch (e) {
  console.log(e);
  await logout();
  throw new Error(e.message);
}
```

Figure 9 : gestion d'erreur lors de la tentative de récupération des données (fichier src\common\container\Auth.container.ts)

Cette partie du fichier gère les erreurs pouvant provenir de la tentative de connexion à l'API :

- Il a fallu ajouter ici l'assignation de l'URL à Axios (le client de requêtes HTTP/HTTPS), car il essaie de s'y connecter directement après que l'url soit renseignée.
- Auparavant, le message de l'erreur n'était pas transmis lors de la création de la nouvelle erreur, qui est transmise à la fonction supérieure.

```
try {
    await authContainer.authenticate(values);
} catch (e) {
    if(e.message == "Network Error")
    {
        if(urlVisible)
        {
            setFieldError("url", "Erreur de connexion, vérifie ton accès à internet et l'URL que tu as entré.");
        }
        else
        {
            setFieldError("token", "Erreur de connexion, vérifie ton accès à internet.");
        }
    }
    else
    {
        setFieldError("token", "Erreur de token, vérifie que le token que tu as entré est bien valide pour le festival sélectionné.")
    }
    setSubmitting(false);
}
```

Figure 10 : Transformation de l'erreur en information « user-friendly »
(fichier src\auth\TokenAuth.component.tsx)

Ici, l'erreur envoyée depuis le fichier Auth.container.ts est récupérée et transformée en un message « user-friendly » (compréhensible par l'utilisateur), puis affichée en dessous des bons champs du formulaire.

Pour ce faire, le message de l'erreur a d'abord été analysé en affichant directement « e.message » dans le formulaire et en testant plusieurs scénarios d'erreur. Il a été remarqué que le message « Network Error » était retourné quand le site ne pouvait être atteint et que le message « 401- unauthorized » apparaissait quand uniquement le token était invalide. Il était ensuite facile d'utiliser ces informations pour configurer la condition et personnaliser le message.

En bonus, une distinction supplémentaire a été faite entre deux scénarios : si l'utilisateur a entré lui-même son URL, il est plus probable que le problème vienne de là, et donc il faut indiquer le message d'erreur sous le champ URL. Mais s'il a sélectionné un des festivals par défaut, il a plus probablement un problème avec sa connexion internet ou alors le site du festival sélectionné est down (moins probable). Étant donné que le champ URL est masqué il faut alors indiquer l'erreur sous le champ token comme normalement.

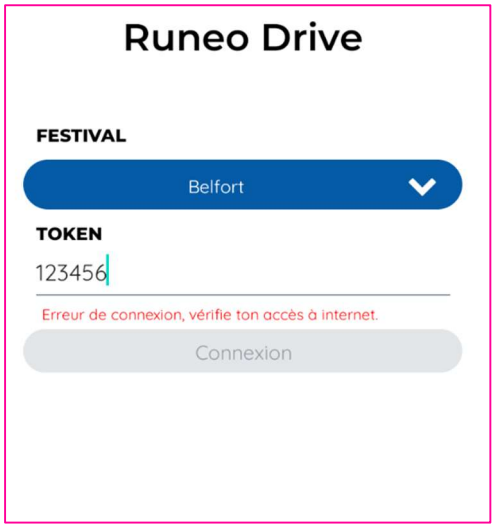

Figure 12 : Pas d'accès internet

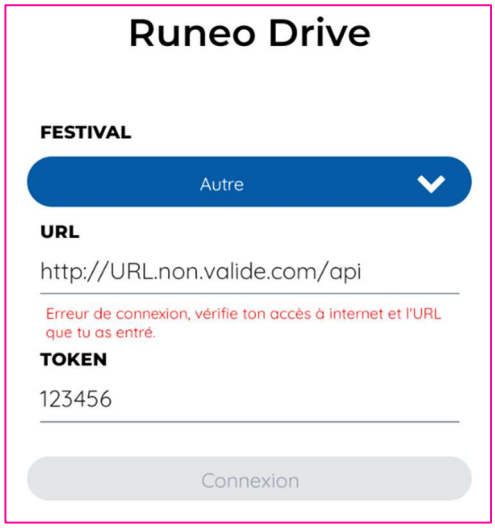

Figure 13 : URL invalide

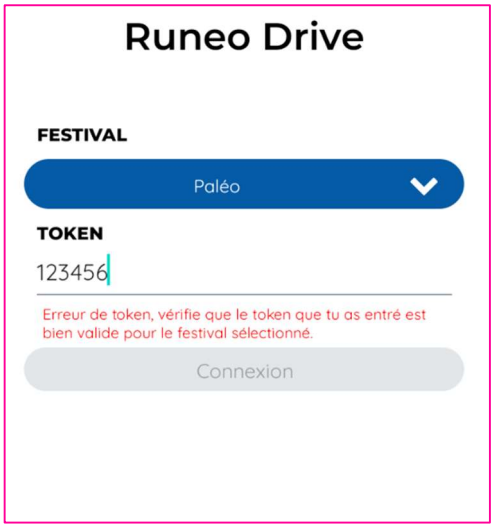

Figure 11 : Token invalide

2.5.2. Autres runs de mon artiste

Concrètement, l'objectif de cette fonctionnalité est de pouvoir, pour chaque run, afficher une liste des runs programmés ou déjà terminés pour le même artiste. Le but pour la réalisation était de reprendre autant des composant et fonctionnements déjà présents dans l'application que possible afin de ne pas refaire le travail à double et de conserver une certaine unité dans le code.

2.5.2.1. Modification de l'API

Avant ce projet, l'application mobile ne recevait aucune information concernant l'artiste du run, seulement le titre du run qui contenait toujours le nom de l'artiste. L'API a donc dû être modifiée pour transmettre les données nécessaires.

Deux options étaient envisageables :

- Envoyer toutes les informations nécessaires liées aux runs et faire le tri sur l'application mobile
- Créer un nouveau lien permettant de récupérer tous les runs du même artiste.

La première option posait un problème tout d'abord parce que seuls les runs futurs étaient actuellement envoyés à l'application mobile, donc il serait impossible de récupérer les runs passés. Étant donné que pour la deuxième option il fallait tout de même spécifier ce que l'on voulait récupérer, il a été finalement choisi d'ajouter l'information de l'id de l'artiste à tous les runs et de créer le lien « runs/artist/{artist} » qui permettrait de récupérer une liste de runs.

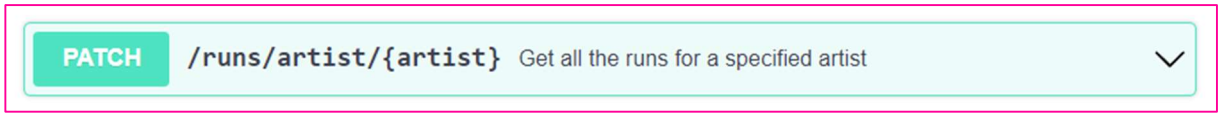

Figure 14 : Définition Swagger du lien de l'API

En utilisant l'application Swagger Editor, il a été possible de définir les besoins de l'API avant de la coder. Cela permet à la fois de documenter et potentiellement de faire réaliser les modifications du code PHP par une autre personne si le temps venait à manquer.

Comme ce n'était pas le cas pour ce projet, le code PHP du site web a été modifié comme on peut le voir sur les captures d'écran suivantes :

```php
/** ****************************
 * Runs resource
 */
Route::prefix('runs')->group(function () {
    Route::patch('{run}/start', [RunController::class, 'start']);
    Route::patch('{run}/stop', [RunController::class, 'stop']);
    Route::patch('{run}/acknowledge', [RunController::class, 'acknowledge']);
    Route::patch('artist/{artist}', [RunController::class, 'runsOfArtist']);
});
```

Figure 15 : Fichier api.php, le dernier lien est celui qui a été ajouté.

```php
/**
 * Get all the runs of the specified artist
 *
 * @param \Illuminate\Http\Request $request
 * @param \App\Models\Artist       $artist
 * @return \Illuminate\Http\Response
 */
public function runsOfArtist(Request $request, Artist $artist)
{
    return new RunCollection($artist->runsForSummary());
}
```

Figure 16 : Méthode retournant les données dans RunControler.php

Techniquement la réalisation n'était pas très complexe car grâce à Laravel et aux méthodes déjà existantes il a suffi de relier la nouvelle route à la bonne méthode.

2.5.2.2. Adaptation de la liste de runs

Pour commencer, il faut récupérer les données. Pour ce faire, la fonction suivante est appelée en asynchrone :

```typescript
function getRunsFromSameArtistApi(run: RunResource): Promise<RunResource[]> {
  return Axios.patch(`/runs/artist/${run.artist_id}`)
    .then((res) => res.data.map(parseRunResource))
    .catch((error) => error.text);
}
```

Figure 17 : fonction exécutant la requête depuis le fichier Runs.container.ts

La syntaxe et le fonctionnement de cette fonction est grandement inspiré par toutes les autres requêtes exécutées à l'API dans ce fichier, la seule modification notable de fonctionnement est la passation du run en paramètres. Grâce au package Axios, une requête « patch » est ensuite envoyée à l'API, et les résultats sont ensuite convertis en tableau de runs grâce au mapping de chaque donnée dans la fonction « parseRunResource », qui est la même que celle utilisée pour la liste de runs normale.

```typescript
useEffect(() => {
    runContainer.getRunsFromSameArtist(currentRun).then((data) => {
        if (items.length == 0){
            data = data.sort((runA,runB) => {return (runA.begin_at.diff(runB.begin_at).toMillis() > 0) ? 1 : -1});
            setItems(data);
        }
        setIsLoading(false)
    })
});
```

Figure 18 : Récupération des données depuis le fichier ListRunsFromArtist.component.tsx

La récupération des données a besoin d'être encapsulée dans un « useEffect » afin de pouvoir attendre le résultat de la fonction asynchrone sans perturber le reste du code de la page. L'idée de ce fonctionnement a été trouvée en examinant celui utilisé pour récupérer l'historique des logs des runs (dans le fichier « CommentRuns.component.tsx »), étant donné que la situation est à peu près similaire. En effet, il faut aussi récupérer une liste de données qui provient d'une requête patch et qui n'est pas assez centrale pour

que cela soit pertinent d'utiliser un container pour gérer les données. Pour gérer la liste de runs centrale par exemple, aucun run n'est transmis sans passer par un objet « cacheHelper », permettant de transmettre les données facilement grâce à un composant de liste créé exprès pour ce fonctionnement. Ce système rend aussi disponible la liste de runs actuellement chargés dans le reste de l'application simplement en utilisant une méthode du container au début du fichier, « useContainer() ».

```
return (
    <SafeAreaView style={styles.fill}>
        { isLoading ?
            <Text style={styles.loading}>Chargement...</Text>
            :
            <FlatList
                keyExtractor={((item: RunResource) => String(item.id))}
                data={items}
                renderItem={renderItem}
                refreshing={isLoading}
                style={{
                    height: "100%"
                }}
            />
        }
    </SafeAreaView>
)
```

Figure 19 : Affichage de la liste des runs, fichier ListRunsFromArtist.component.tsx

On peut voir dans ce code que seule le composant natif « FlatList » est utilisé pour afficher la liste des runs. Les propriétés les plus importantes sont « data », contenant la valeur du tableau items rempli ou non avec les données récupérées dans la partie de code au-dessus, et « renderItem », une fonction permettant de gérer l'affichage de chaque run. Cette dernière a pu être reprise entièrement de la liste de runs déjà existantes pour conserver une unité visuelle.

On peut aussi observer le système utilisé pour gérer le temps de chargement des données. La variable « isLoading » est aussi un « state » de React (tout comme « items), ce qui fait que chaque fois que sa valeur change, tous les appels à cette variable sont recalculés. Donc ici, tant que isLoading est true, seul un message d'information « Chargement… » sera affiché au milieu de la page, en attendant que les données soient prêtes à être affichées. La valeur de isLoading passe alors à false (comme on peut le voir dans la figure 18 de la page précédente) et c'est la liste des runs qui sera alors affichée.

Pour finir, voici le rendu visuel de la fonctionnalité :

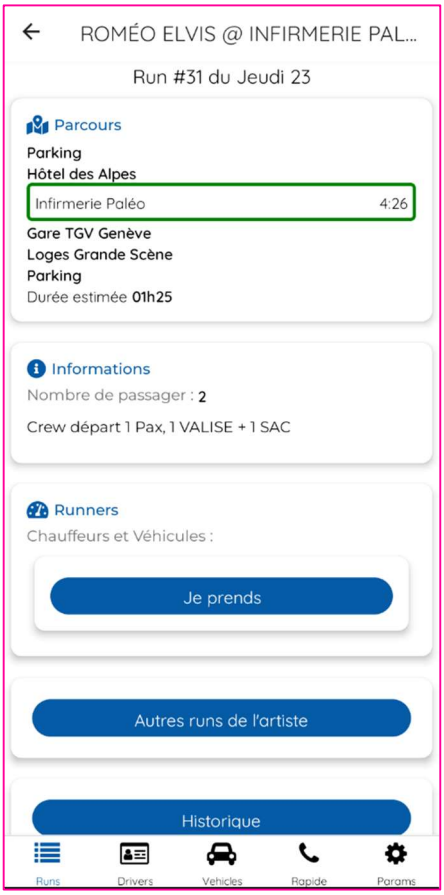

Figure 22 : Page des détails d'un run

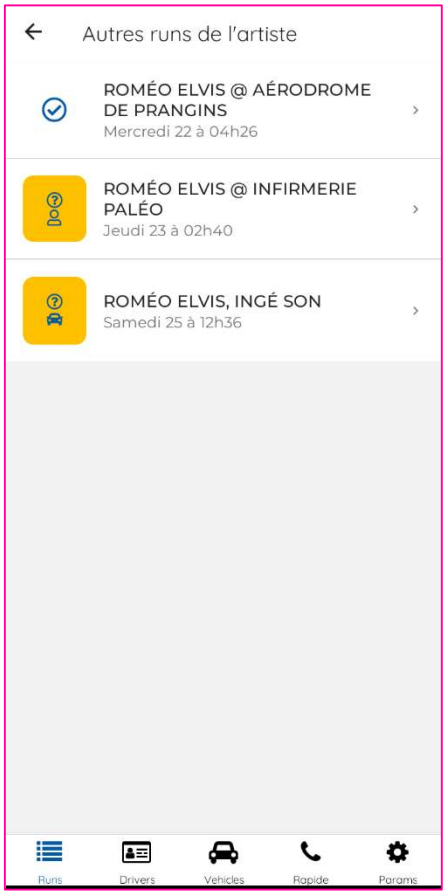

Figure 21 : Page des autres runs de l'artiste

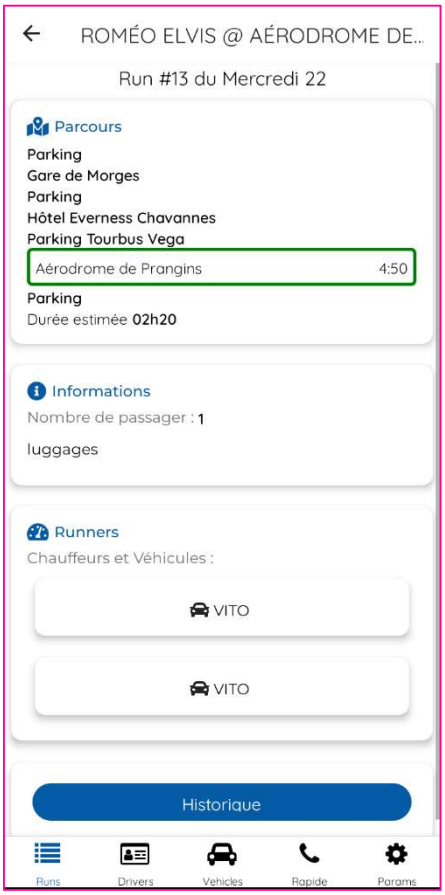

Figure 20 : Page des détails d'un run affichée après le clic

2.5.2.3. Fonctionnement de la navigation

En react native, la navigation au sein de l'application est gérée par un objet global pouvant être appelé grâce à la méthode « useNavigation() ».

Voyons comment ce système fonctionne en se penchant d'abord sur le composant du bouton, présent sur la page d'un run :

```
export function DetailRunsOtherFromArtistComponent({currentRun}: InfoDetailRunsComponentProps) {
    const navigation = useNavigation();

    const onChangePress = async () => {
        navigation.navigate("listFromArtist", {runId: currentRun.id});
    }

    //if the page is called after the runs from other artist page the index is tree
    if(navigation.getState().index >= 3)
    {
        return null
    }

    return (
        <CardContainerComponent>
            <View style={styles.buttonContainer}>
                <View style={styles.buttonWrapper}>
                    <ButtonComponent titleStyle={styles.buttonTitle} title="Autres runs de l'artiste" onPress={onChangePress}/>
                </View>
            </View>
        </CardContainerComponent>
    )
}
```

Figure 23 : Composant du bouton dans la page de détail d'un run, fichier DetailRunsOtherFromArtist.component.tsx

On peut ici voir la méthode principale de cette navigation, « navigate ». Elle est appelée après que le bouton ait été cliqué, et prend en paramètre le nom de la page qui doit être appelée, et des propriété optionnelles pouvant être utilisées par cette prochaine page. On y insère assez logiquement l'id du run depuis lequel le bouton est cliqué afin de pouvoir récupérer le run dans la page suivante.

On peut aussi observer la condition en dessous, permettant de ne pas afficher le composant si l'index de navigation est égal à trois. En effet, le problème suivant avait été relevé depuis l'analyse : le bouton pour afficher la page des autres runs de l'artiste ne doit pas pouvoir être recliqué depuis un run dont les détails ont été ouverts depuis cette même page. L'objet de navigation possède donc heureusement cette propriété « index », permettant de savoir combien de pages ont été ouvertes les unes par-dessus les autres. Car à chaque fois que la méthode « navigate » est appelée pour ouvrir une nouvelle page, la navigation l'ouvre par-dessus la page actuelle afin qu'il soit possible pour l'utilisateur de retourner en arrière. La valeur trois provient simplement de l'observation de cet index une fois la page réouverte grâce au système que l'on verra plus loin.

Pour que la navigation puisse trouver la page « listFromArtist » renseignée en paramètre, il a fallu définir la route dans le fichier « Runs.component.tsx » :

```
return (
    <Stack.Navigator initialRouteName={"list"}>
        <Stack.Screen name={"list"} component={ListRunsComponent} options={{headerShown: false}}/>
        <Stack.Screen
            name={"detail"}
            component={DetailRunsComponent}
            options={generateStackOptionWithRunTitle}
        />
        <Stack.Screen name={"select_vehicle"}
                      component={RunsSelectVehicleComponent}
                      options={{title: "Choisissez un véhicule", headerBackTitle: "Annuler"}}/>
        <Stack.Screen name={"listFromArtist"}
                      component={ListRunsFromArtistComponent}
                      options={{title : "Autres runs de l'artiste"}}/>
```

Figure 24 : Composant "stack" permettant de définir les routes de la navigation

On peut voir ici les routes déjà paramétrées auparavant ainsi que celle qui a été ajoutée, avec le nom « listFromArtist ». Le fonctionnement est assez simple, il suffit de définir le composant devant être appelé dans la page et optionnellement le titre du bandeau de navigation.

Le titre « Autres runs de l'artiste » a été choisi à la place de l'objectif de départ qui était d'afficher « Autres runs de [nom de l'artiste] ». En effet, il aurait fallu faire un appel de plus à l'API pour récupérer cette donnée, et il a été décidé que la valeur ajoutée ne valait pas la peine de potentiellement ralentir encore le chargement et de perdre du temps de travail.

Voyons maintenant comment la nouvelle page de détail d'un run est ouverte depuis le composant « ListRunsFromArtistComponent » :

```
const gotoRun = (run: RunResource) => navigation.push("detail", {run: run})

const renderItem = ({item}: { item: RunResource }) => {
    return <ListRunsItemComponent onSelectRun={gotoRun} run={item}/>
}
```

Figure 25 : fonctions permettant d'afficher chaque élément de la liste et de rediriger vers la bonne page

La fonction « renderItem » retourne simplement le même composant qui est déjà utilisé pour la liste de runs normale.

Les changements ont lieu surtout lors de l'appel à la navigation : on peut voir qu'à la place de « navigate », la méthode « push » est utilisée. En effet, si l'on appelle la méthode « navigate » en renseignant une route qui est déjà ouverte dans la pile, la navigation va retourner sur cette page et non en ouvrir une nouvelle. La méthode « push » permet de résoudre ce problème en forçant l'ouverture de la page spécifiée par-dessus la page actuelle, peu importe les pages déjà ouvertes auparavant. Cela permet de pouvoir retrouver la page des autres runs de l'artiste lors du retour en arrière, et d'avoir le bon index pour le fonctionnement de la condition détaillée sur la page précédente.

Un autre détail qui a son importance est le passage de l'objet du run directement dans les paramètres de la navigation, à la place de simplement indiquer son ID. Dans le reste de l'application, seul l'id du run est

passé en paramètre, puis le run est récupéré grâce à une méthode « useRunFromRouteParam() » qui va effectuer un « find », donc une recherche, dans le container des runs. Cependant, les runs récupérés de l'API pour la page des autres runs de l'artiste ne sont pas disponibles dans un container (comme indiqué au point précédent 2.5.2.2), et encore moins dans le container des runs principaux, car certains peuvent être des runs déjà terminés qui ne doivent dont pas apparaitre dans la liste principale. Il a donc fallu modifier la méthode « useRunFromRouteParam() », ainsi que les valeurs acceptées en tant que paramètres :

```
export interface RunDetailParams {
    runId: number,
    run: RunResource | null
}
```

Figure 26 : changement des valeurs acceptées en tant que paramètres dans le fichier Run.component.tsx

```
export function useRunFromRouteParam(): RunResource | null {
    const runsContainer = RunsContainer.useContainer();

    const route = useRoute();
    const params = route.params as RunDetailParams;

    if(params.run == null)
    {
        return runsContainer.items.find(run => run.id === params.runId) || null;
    }
    else
    {
        return params.run;
    }
}
```

Figure 27 : fonction useRunFromRouteParam du fichier Run.hook.ts

Cette méthode étant utilisée dans le reste de l'application, le changement ne devait pas impacter l'utilisation normale de la fonction mais seulement ajouter une possibilité. C'est pourquoi le paramètre « run » est nullable, et que s'il n'est pas spécifié, la fonction retourne le même résultat qu'avant. En revanche, s'il est spécifié, alors il suffit de retourner le run entré en paramètre. Le fait que la fonction retourne exactement la même chose qu'avant et que le contexte appelant la fonction n'ait pas besoin d'en savoir plus permet d'utiliser exactement la même page de détails d'un run que celle appelée la première fois.

2.5.3. Restreindre le choix de véhicule

La majorité de cette fonctionnalité était déjà présent dans le code, mais n'avait pas été finalisé pour des raisons inconnues. Les modifications tiennent donc littéralement en deux lignes de code. Pour que vous puissiez comprendre le contexte de ces minimes modifications, il tout de même nécessaire d'expliquer le fonctionnement du système entier.

```
const selectVehicle = (runnerId: number, type: string) => {
    const params: RunsSelectVehicleParams = {runnerId, type}
    navigation.navigate("select_vehicle", params);
}
```

Figure 28 : Fonction selectVehicle du fichier DetailRunsRunners.component.tsx

```
<ButtonComponent
    title="Choisir"
    disabled={ !isInternetReachable }
    color="#f194ff"
    onPress={() => selectVehicle(runner.id, runner.vehicle_category?.type as string)}
/>
```

Figure 29 : Bouton permettant de choisir le véhicule, même fichier que la fig. 28

On peut voir ici la provenance du type de véhicule devant être filtré : il est passé en tant que paramètre de navigation en même temps que l'ID du runner.

```
const {runnerId, type} = route.params as RunsSelectVehicleParams;

return(
    <ListVehiclesViewComponent
        filter={(vehicle: VehicleResource) => vehicle.type.type === type}
        hideStatusColor={true}
        onItemPress={(vehicle: VehicleResource) =>{
            updateVehicle(runnerId, vehicle.id)
                .then(() => {
                    navigation.goBack()
                })
                .catch(() => {
                    Alert.alert("Erreur", "Le véhicule n'a pas pu être sélectionné")
                })
            }
        }
    />
)
```

Figure 30 : élément retourné du composant du fichier RunsSelectVehicle.component.tsx

Le composant appelé depuis la route, RunsSelectVehicleComponent, retourne le composant ListVehiclesViewComponent. Il spécifie dans les propriétés la fonction devant être exécutée après un clic sur un élément, mais aussi une fonction permettant de filtrer les éléments selon le bon type de véhicule, récupéré depuis les paramètres de navigation. Cependant, avant les modifications effectuées durant ce projet, la fonction passée n'était jamais utilisée.

```
export interface ListVehiclesViewComponentProps {
    onItemPress: (vehicle: VehicleResource) => void,
    hideStatusColor?: boolean,
    filter?: (item: VehicleResource) => boolean
}
```

Figure 31 : Propriétés du composant ListVehicleViewComponent, dans le fichier ListVehicleView.component.tsx

Le paramètre « filter » a été ajouté afin de pouvoir récupérer la fonction. Étant donné que le code est en TypeScript, il est obligatoire de spécifier le type du paramètre et de retour de la fonction.

```
return (
    <SafeAreaView>
        <ListCommonResourceComponent
            filter={props.filter}
            sort={(vehicleA: VehicleResource, vehicleB: VehicleResource) => {
                return vehicleA.name > vehicleB.name ? 1 : -1
            }}
            dataContainer={VehiclesContainer}
            renderItem={renderItem}/>
    </SafeAreaView>
)
```

Figure 32 : Retour du composant ListVehicleViewComponent, dans le même fichier

Le paramètre « filter » est ensuite transmis au composant « ListCommonResourceComponent » afin de filtrer les données. Grâce à tout le travail déjà effectué dans le passé, ces deux modifications étaient les seules nécessaires, le composant « ListCommonResourceComponent » étant déjà prévu afin d'accepter le paramètre de filtrage.

```
if (props.filter) {
    data = data.filter(props.filter)
}
```

Figure 33 : Filtrage final des données dans le fichier ListCommonResource.component.tsx