

# Créer une application mobile avec React Native

*Support de cours pour la formation*

*React Native de Mai 2023*

*Formateur : Dufrène Valérian*

*“Déclaration d’activité enregistrée sous le numéro 32800232680 auprès du préfet de  
région HAUTS-DE-FRANCE”*

## Chapitre 1 - Découverte de React Native

## Déroulé et objectif

Cette formation est découpée en 3 chapitres, qui serviront de fil rouge à la formation.

À la fin de cette formation, vous saurez :

- Expliquer l'utilité de React Native.
- Déterminer si React Native est adapté à la création de votre projet.
- Mettre en place un projet React Native en ligne de commande avec Expo.
- Utiliser les commandes basiques de la CLI React Native.
- Utiliser les commandes CLI de Expo utiles au développement de votre projet.
- Faire la différence entre les composants **cores**, les composants **natifs** et les composants **communities**.
- Déterminer les points qui servent au référencement d'une application React Native.
- Créer un composant personnalisé.
- Structurer votre projet pour ajouter des composants personnalisés.
- Passer des props d'un composant parent à un composant enfant.
- Comprendre comment fonctionne le composant StyleSheet.
- Utiliser les vues pour découper votre application.
- Importer de nouveaux composants cores.
- Découper les différentes pages de votre application en "screens".
- Créer un APK pour exporter l'application sur un téléphone Android.
- Stocker des données sur le stockage local d'un appareil.
- Structurer un projet complexe, en respectant les bonnes pratiques.
- Gérer des interactions avec l'utilisateur.
- Traiter des données fournies par un utilisateur.
- Créer un CRUD pour votre application.
- Concevoir une UI améliorant l'UX de votre application.

## Objectifs de production (fil rouge)

1. Application de calcul d'IMC.
2. Application vitrine pour un restaurant.

## Historique de React Native

React native est un framework open-source destiné à créer des applications natives (mobile, tablette, windows, console, casque VR, smart TV, ...), rendu publique le 26 Mars 2015.

Ce framework est principalement utilisé pour développer des applications **Android**, **IOS** et **UWP** (Universal Windows Platform) :

[https://fr.wikipedia.org/wiki/Universal\\_Windows\\_Platform](https://fr.wikipedia.org/wiki/Universal_Windows_Platform)).

Il permet de développer une application native, en utilisant du **Javascript** ou du **Typescript** et en utilisant le langage de balisage **JSX**.

À la compilation, le langage utilisé est converti en langage natif (**Java** pour Android et **Swift** pour IOS).

Il est donc plus simple pour un développeur web de se convertir au développement d'applications natives, grâce aux langages requis par React Native.

Avoir une expérience en **ReactJs** peut faciliter la compréhension de la structure des projets et du processus de développement d'une application **React Native**.

Pour la création d'une application sous React Native, il faut garder une phrase en tête : *"Tout est composant"*.

Pour la création d'une application dédiée au web, il est préférable d'utiliser uniquement **ReactJs**, qui est plus optimisé.

**Date de lancement de React Native : 26 Mars 2015.**

### Prérequis (obligatoires)

Pour commencer la mise en place d'un projet React Native, avec expo, vous aurez besoin de :

- Installer **NodeJS** sur son ordinateur (version LTS conseillée) : <https://nodejs.org/en/download>.
- Un terminal de commande (Bash, Powershell, ...).
- Un **IDE** (Visual Studio Code, JetBrains, ...) : Nous utiliserons Visual Studio Code lors de cette formation.
- Mettre à jour **NPM** : `npm install npm@latest -g`.
- Avoir des connaissances avancées **Javascript** ou **Typescript** / **JSX** / **NPM**.
- Avoir des connaissances basiques **HTML** / **CSS**.
- Installation sur son mobile de l'application "**Expo Go**" (**Android**) ou "**Expo**" (**IOS**).
- (**Optionnel mais fortement recommandé**) Installation de **Android Studio** sur son ordinateur, pour tester l'application sous différentes versions d'Android.
- Savoir utiliser **GIT**.

## Prérequis (optionnels)

Ces points peuvent être bénéfiques pour le suivi de la formation, mais sont optionnels :

- Avoir des connaissances **ReactJS / API**.
- Utiliser **Yarn**.
- Utiliser les extensions suivantes sur Visual Studio Code : **ES7+ React/Redux/React-Native snippets / React Native Tools / React Native Snippets**.
- Utiliser **Github**, si vous n'êtes pas à l'aise avec **GIT CLI**.

## Premiers pas

Nous allons commencer par initialiser notre premier projet React Native.

**Ouvrez votre terminal de commandes** et placez-vous dans le dossier où vous souhaitez créer votre application (l'initialisation du projet créera votre application dans un sous-dossier, au nom spécifié lors du lancement de la commande).

***Assurez-vous d'avoir les droits d'administration lors du lancement des commandes liées à React Native !***

Pour initialiser un projet, entrez la commande suivante :

***`npx create-expo-app@latest react-native-imc-calculator`***

Cette commande aura pour effet de lancer la création d'une application qui portera le nom de "react-native-imc-calculator".

Une fois l'initialisation terminée, entrez la commande suivante pour vous déplacer dans le dossier de votre nouvelle application :

***cd react-native-imc-calculator***

Pour vous assurer que l'installation se soit bien déroulée, entrez la commande :

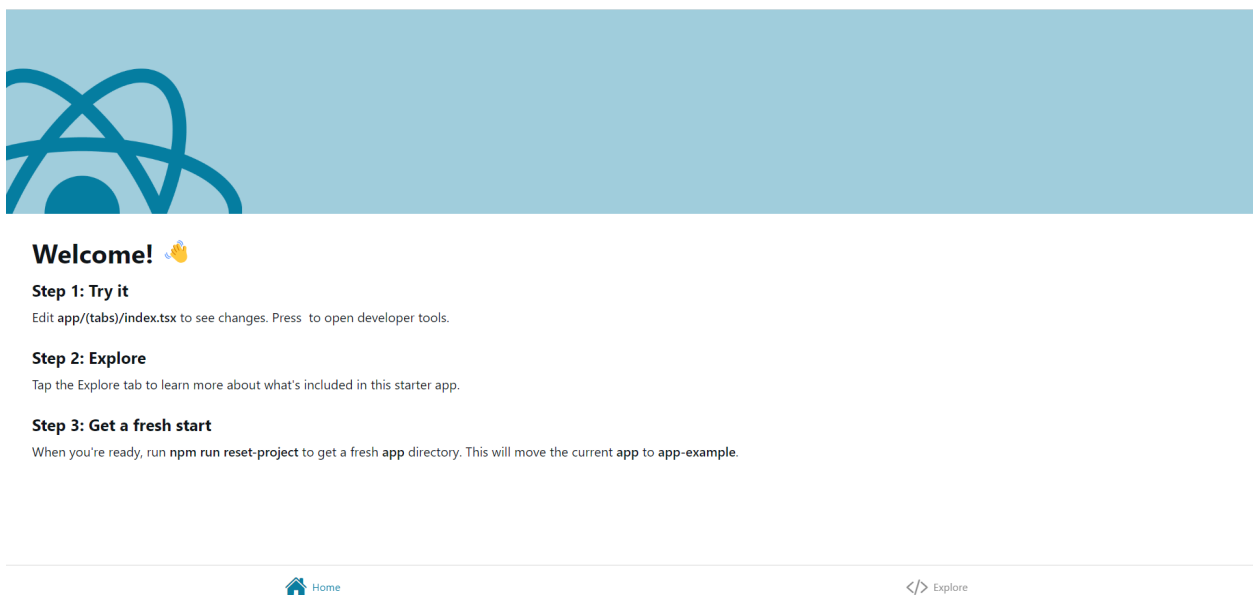
***npm run web***

ou

***npm run start***

En mode web, votre application s'ouvrira sous votre navigateur par défaut.

En mode classique (start) vous pourrez choisir le mode de lancement de l'application, ou scanner le QR Code avec **Expo Go** (ou l'appareil photo sur IOS).

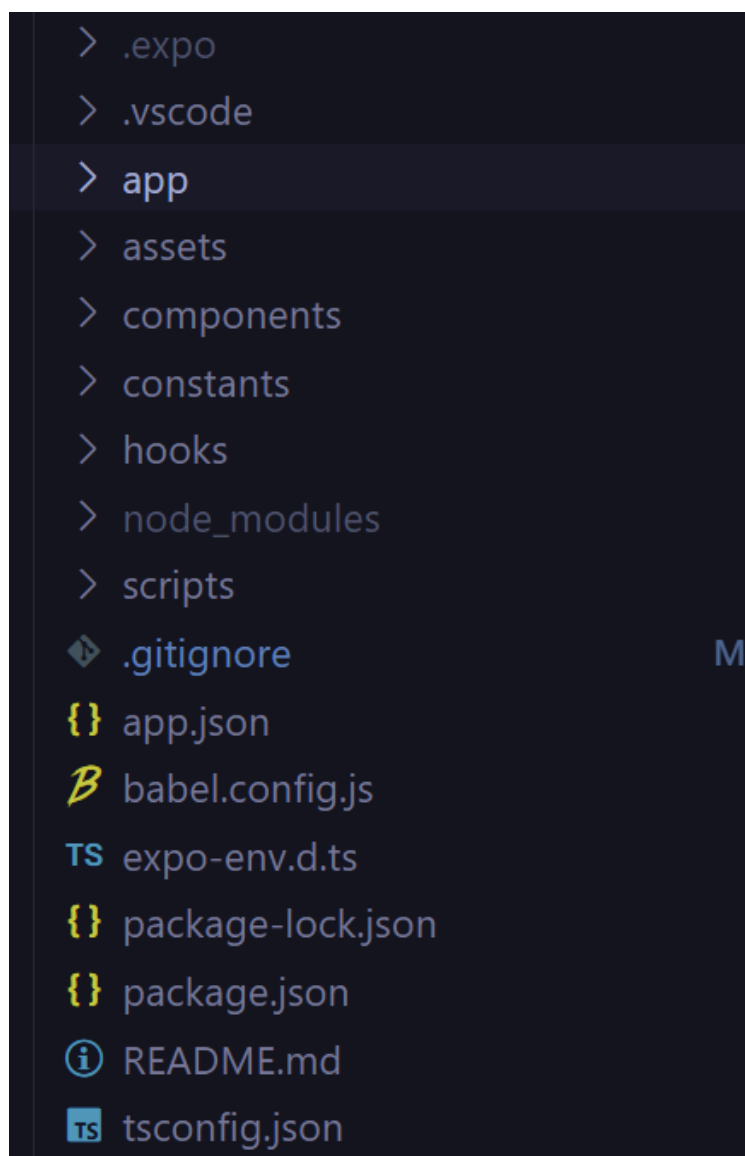


## Analyse de la structure initiale

Lors de la rédaction de ce support, React Native était en version **0.74**.

Sous cette version, vous devrez créer les éléments de votre application dans le dossier “**app**” et la structure de vos fichiers devront respecter la logique de “**file routing**” (navigation basée sur les noms de dossiers et de fichiers).

Voici la structure initiale du projet :



Nous allons analyser les fichiers initiaux pour comprendre leur utilité.

## App.json :

A screenshot of a code editor window showing the 'App.json' file for a project named 'dice-launcher'. The file contains a JSON configuration for an Expo application. The configuration includes details for the application's name, slug, version, orientation, icons, splash screen, and platform-specific settings for iOS and Android. The JSON is formatted with syntax highlighting and line numbers on the left side of the editor.

```
1 {
2   "expo": {
3     "name": "dice-launcher",
4     "slug": "dice-launcher",
5     "version": "1.0.0",
6     "orientation": "portrait",
7     "icon": "./assets/images/icon.png",
8     "scheme": "myapp",
9     "userInterfaceStyle": "automatic",
10    "splash": {
11      "image": "./assets/images/splash.png",
12      "resizeMode": "contain",
13      "backgroundColor": "#ffffff"
14    },
15    "ios": {
16      "supportsTablet": true
17    },
18    "android": {
19      "adaptiveIcon": {
20        "foregroundImage": "./assets/images/adaptive-icon.png",
21        "backgroundColor": "#ffffff"
22      }
23    }
24  }
25 }
```

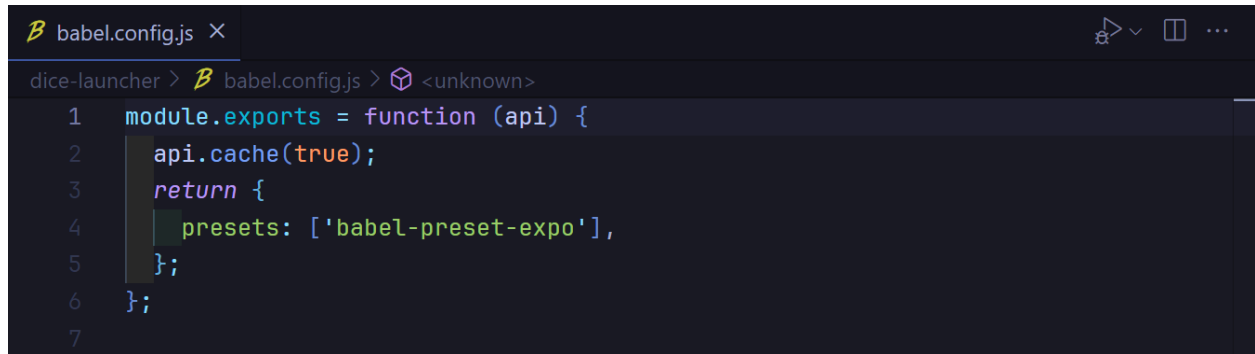
Ce fichier contient toute la configuration de votre application.

Il sera utilisé lors de la création de votre **APK**, pour définir certaines données, comme le **splashScreen** (écran d'accueil s'affichant lors du chargement de l'application), l'orientation par défaut de l'application (portrait ou paysage) ou encore les icônes utilisées pour l'affichage sous Android et IOS.

Vous trouverez également un paramètre "**plugins**", qui nous permettra d'ajouter une dépendance à "**expo-font**", lors de l'import de polices personnalisées.



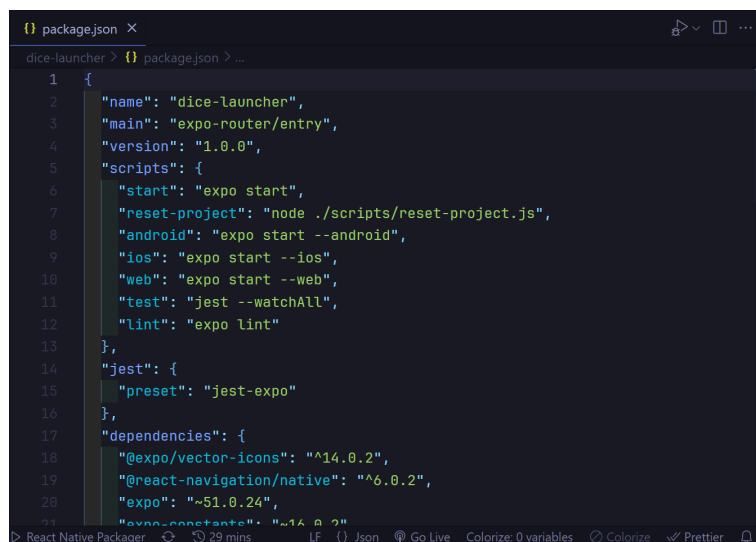
## babel.config.js :



```
babel.config.js X
dice-launcher > babel.config.js > <unknown>
1 module.exports = function (api) {
2   api.cache(true);
3   return {
4     presets: ['babel-preset-expo'],
5   };
6 };
7
```

Ce fichier permet d'ajouter une configuration pour le moteur Javascript nommé Babel.

## package.json :

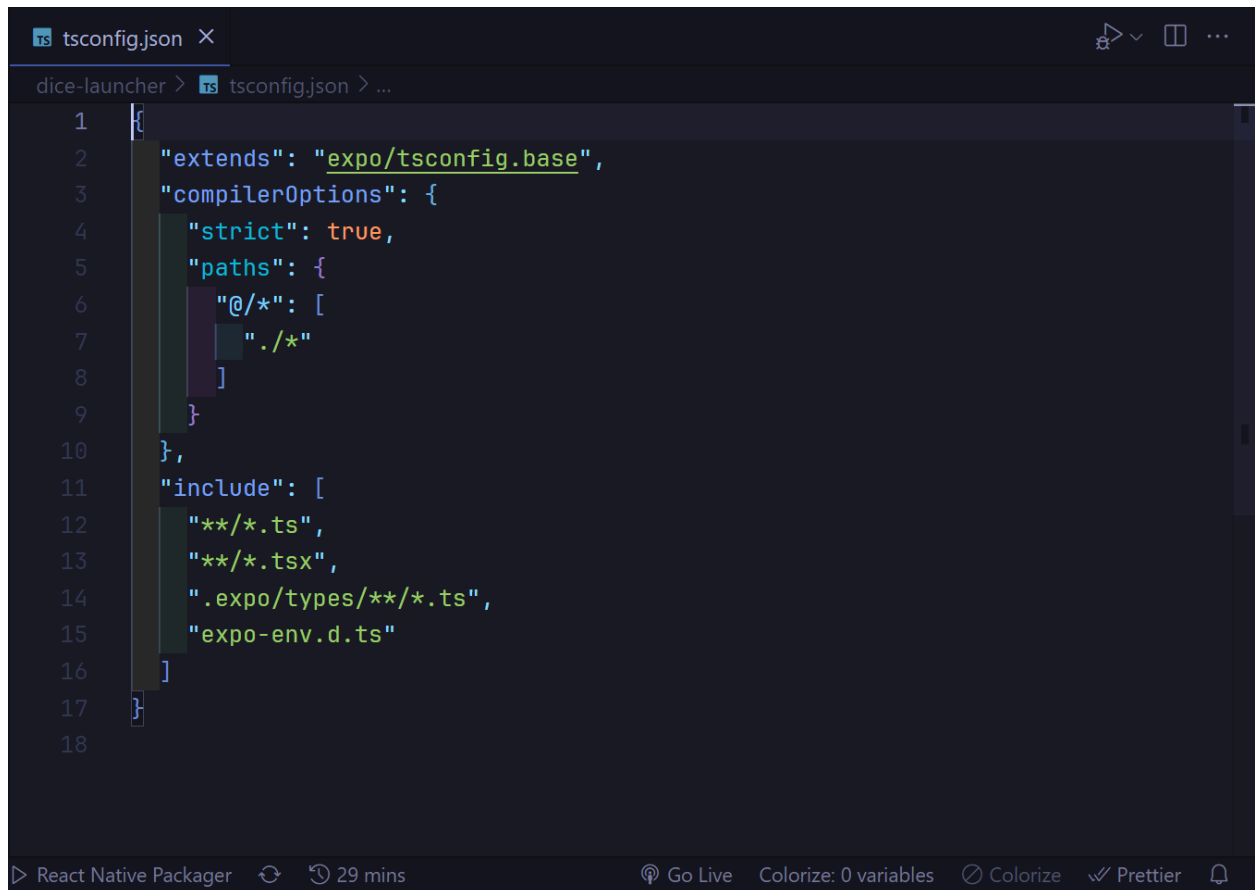


```
package.json X
dice-launcher > package.json > ...
1 {
2   "name": "dice-launcher",
3   "main": "expo-router/entry",
4   "version": "1.0.0",
5   "scripts": {
6     "start": "expo start",
7     "reset-project": "node ./scripts/reset-project.js",
8     "android": "expo start --android",
9     "ios": "expo start --ios",
10    "web": "expo start --web",
11    "test": "jest --watchAll",
12    "lint": "expo lint"
13  },
14  "jest": {
15    "preset": "jest-expo"
16  },
17  "dependencies": {
18    "@expo/vector-icons": "^14.0.2",
19    "@react-navigation/native": "^6.0.2",
20    "expo": "~51.0.24",
21    "expo-constants": "~16.0.2"
22  }
23 }
```

Ce fichier contient tous les paramètres de vos dépendances et scripts.

En utilisant la commande “**npm i**”, vous installerez les dépendances inscrites dans ce fichier.

ts.config.json :

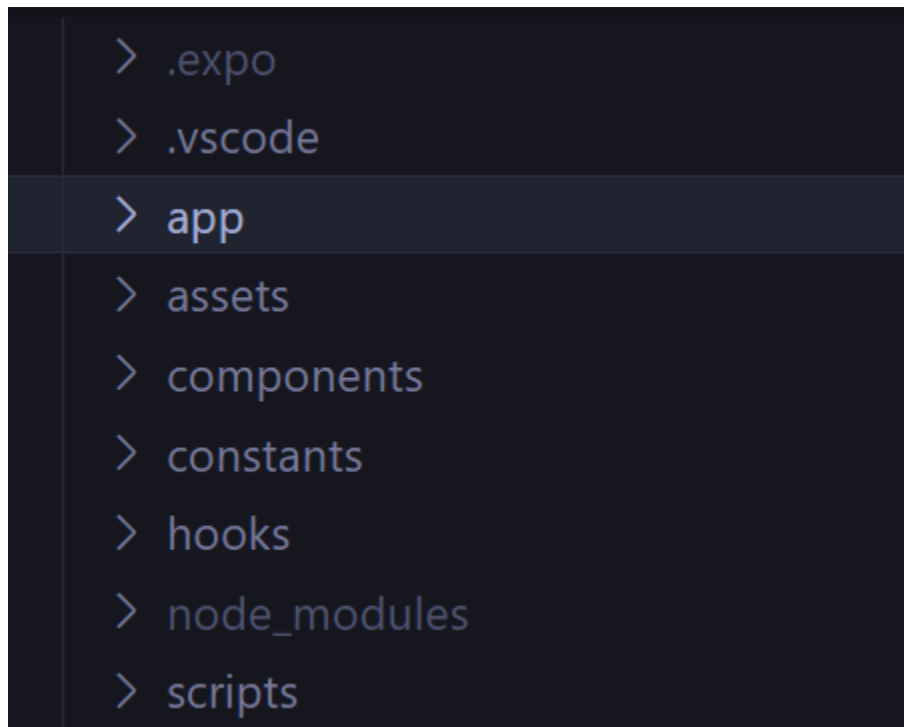


```
1 {
2   "extends": "expo/tsconfig.base",
3   "compilerOptions": {
4     "strict": true,
5     "paths": {
6       "@/*": [
7         "./*"
8       ]
9     }
10  },
11  "include": [
12    "**/*.ts",
13    "**/*.tsx",
14    ".expo/types/**/*.ts",
15    "expo-env.d.ts"
16  ]
17 }
18
```

Ce fichier vous permet d'ajouter une configuration pour Typescript.

Le paramètre “**paths**” vous permettra d'utiliser la notation  
“@/dossier/fichier\_a\_importer”.

Analysons désormais la structure des dossiers initiaux.



#### **.expo :**

Ce dossier contient une configuration pour les commandes du terminal fournies par Expo.

#### **.vscode :**

Ce dossier contient des configurations supplémentaires pour Visual Studio Code.

#### **app :**

Ce dossier a pour rôle de contenir tous les fichiers de l'application et utilise une structure de **file routing**.

### assets :

Ce dossier a pour rôle de contenir les ressources de l'application, comme les images, les audios, les vidéos, les polices, etc...

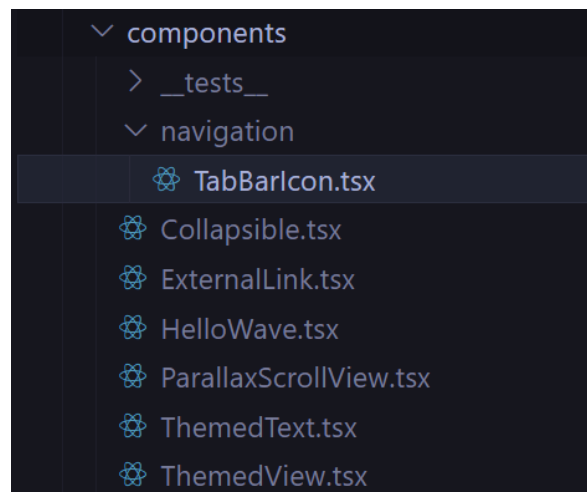
### components :

Ce dossier a pour rôle de contenir tous les composants personnalisés créés pour l'application.

Lors de l'initialisation du projet, vous aurez quelques composants qui vous serviront d'exemple.

Dans "**components > navigation**", vous pourrez trouver le composant **TabBarIcon**.

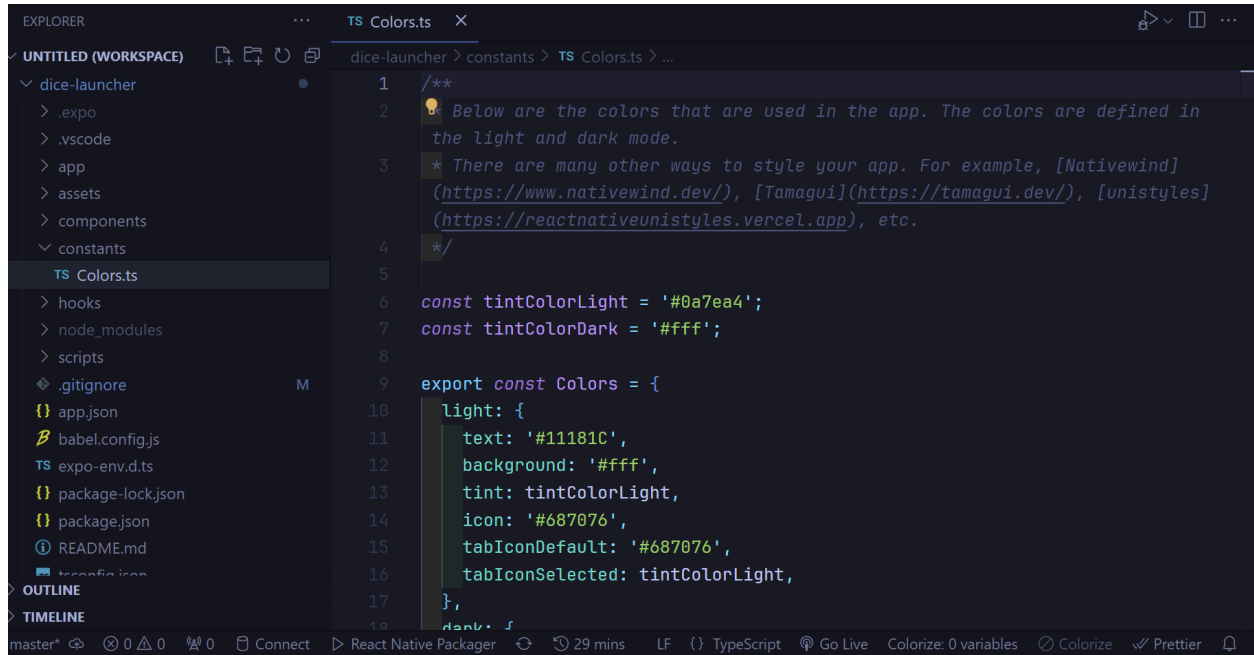
Je vous recommande de conserver ce composant, car il vous permet d'économiser du temps de création pour un composant gérant l'affichage des icônes de la tabBar.



## constants :

Ce dossier a pour rôle de contenir vos constantes d'application.

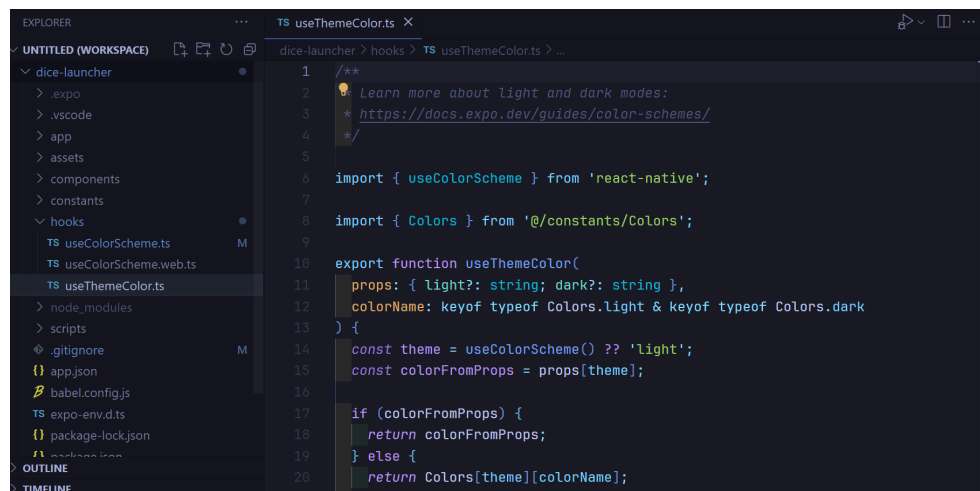
Vous avez un exemple lors de l'initialisation du projet, avec le fichier **"Colors.js"**, qui contient des constantes de thème.



The screenshot shows the VS Code interface with the 'dice-launcher' workspace. The Explorer sidebar on the left shows the project structure, with the 'constants' folder expanded and 'TS Colors.ts' selected. The main editor displays the content of 'Colors.ts'.

```
1 /**
2  * Below are the colors that are used in the app. The colors are defined in
3  * the light and dark mode.
4  * There are many other ways to style your app. For example, [Nativewind]
5  * (https://www.nativewind.dev/), [Tamagui] (https://tamagui.dev/), [unistyles]
6  * (https://reactnativeunistyles.vercel.app), etc.
7  */
8
9 const tintColorLight = '#0a7ea4';
10 const tintColorDark = '#fff';
11
12 export const Colors = {
13   light: {
14     text: '#11181C',
15     background: '#fff',
16     tint: tintColorLight,
17     icon: '#687076',
18     tabIconDefault: '#687076',
19     tabIconSelected: tintColorLight,
20   },
21   dark: {
22     text: '#fff',
23     background: '#0a0a0a',
24     tint: tintColorDark,
25     icon: '#fff',
26     tabIconDefault: '#fff',
27     tabIconSelected: tintColorDark,
28   },
29 }
```

## hooks :



The screenshot shows the VS Code interface with the 'dice-launcher' workspace. The Explorer sidebar on the left shows the project structure, with the 'hooks' folder expanded and 'TS useThemeColor.ts' selected. The main editor displays the content of 'useThemeColor.ts'.

```
1 /**
2  * Learn more about light and dark modes:
3  * https://docs.expo.dev/guides/color-schemes/
4  */
5
6 import { useColorScheme } from 'react-native';
7
8 import { Colors } from '@constants/Colors';
9
10 export function useThemeColor(
11   props: { light?: string; dark?: string },
12   colorName: keyof typeof Colors.light & keyof typeof Colors.dark
13 ) {
14   const theme = useColorScheme() ?? 'light';
15   const colorFromProps = props[theme];
16
17   if (colorFromProps) {
18     return colorFromProps;
19   } else {
20     return Colors[theme][colorName];
21   }
22 }
```

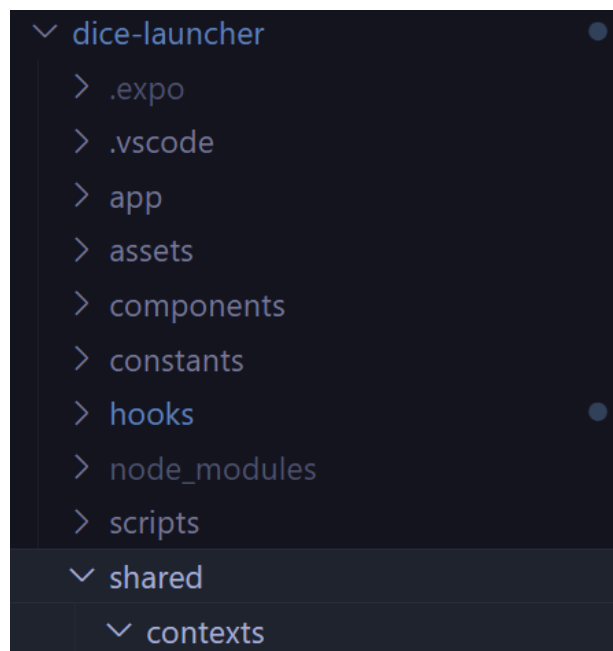
Ce dossier a pour but de contenir les **hooks** personnalisés.

## Création de nos premiers dossiers

Afin d'améliorer la structure de notre application, pour la création des applications fil rouge, nous allons créer de nouveaux dossiers, qui auront un rôle spécifique dans la suite de cette formation.

À la racine du dossier “**react-native-imc-calculator**” (dossier d'application), créez un dossier nommé “**shared**”.

Ensuite, créez à l'intérieur du dossier “**shared**”, un dossier nommé “**contexts**”.



Le dossier “**shared**” a pour rôle de contenir les éléments partagés par les différents éléments de votre application.

Le dossier “**contexts**” a pour rôle de contenir les **contextes** de notre application.

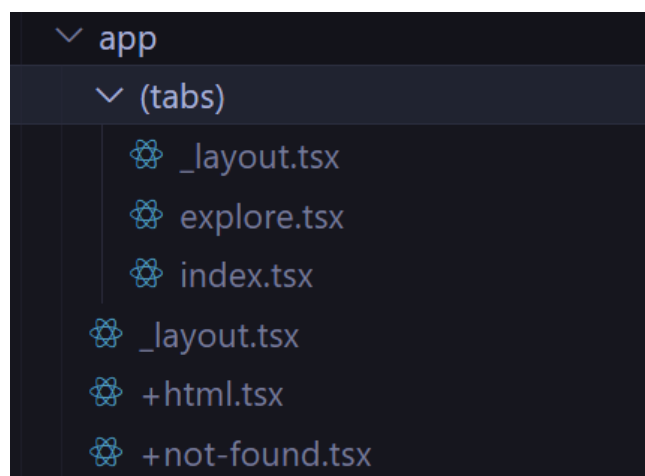
Les contextes nous permettront de centraliser des données et de les fournir à notre application via des **providers**.

## Refonte des fichiers initiaux

Pour commencer la création de notre application, nous allons modifier les fichiers créés par défaut, afin de récupérer une structure de base.

Nous allons conserver les styles par défaut de l'application et nous améliorerons le style de l'application au fil de sa conception.

Rendez-vous dans le dossier “app”.



**Supprimez** le fichier “+html.tsx”, qui représente uniquement la structure de page statique pour la version web.

Ce fichier ne nous intéresse pas pour la création d'application mobile.

**Supprimez** également le fichier “+not-found.tsx”, qui représente la page d'erreur personnalisée.

Si vous essayez de charger une route qui n'existe pas, cette page d'erreur sera affichée.

En version application mobile, elle n'a pas d'intérêt pour notre projet.

**Supprimez** le dossier “(tabs)” et son contenu.

Le dossier “(tabs)” représente un **groupe de routes**, qui permet de sectionner son application par **domaine**.

Nous créerons notre propre structure de routes pour les projets fil rouge.

**Supprimez tout le code** contenu dans le fichier “\_layout.tsx” du dossier “app”.

Nous allons commencer par la refonte de ce fichier pour commencer la création de notre application.

### Création du routage et du premier écran

Dans une application React Native, nous allons parler d'**écrans** pour définir les composants qui affichent des données à l'utilisateur.

Quand on parle d'écran, on parle bien sûr du composant **squelette** de l'écran, qui pourra lui-même contenir des **composants personnalisés**.

Pour commencer l'application, nous allons créer notre premier fichier “\_layout.tsx” (celui du dossier “app”), qui contiendra un composant de structuration.

Dans ce fichier, nous allons définir la structure de notre premier routage.

Afin de voir deux méthodes de routage différents, utilisables dans une application React Native (**Stack** et **Tabs**), nous allons utiliser le composant **Stack** fourni par “expo-router” dans ce premier fichier de structure.



Commencez par créer le composant de structure, nommé **“RootLayout”**, qui devra être exporté par défaut et ne prendra pas de **props**.

```
export default function RootLayout(){  
  
}
```

Ce composant devra retourner un composant `<Stack></Stack>`, fourni par la dépendance “**expo-router**” (n’oubliez pas l’import).

Ce composant `<Stack>` englobera un composant `<Stack.Screen />`, qui représentera un écran de l'application.

Le composant `<Stack.Screen/>` aura besoin de deux propriétés :

- **name**, qui prendra la chaîne de caractère “**(global)**” comme valeur et qui représente le chemin contenant l’écran (ici un groupe de routes).
- **options**, qui prendra comme valeur une **interpolation d’objet**, contenant les paramètres :
  - **title**, qui prendra comme valeur la chaîne de caractères “**Groupe global**”, représentant le texte inclus dans l’entête, en haut de l’écran.
  - **headerShown**, qui prendra une valeur booléenne de type “**false**”, ce qui nous permet de masquer l’affichage de l’entête.

```
import { Stack } from "expo-router";

export default function RootLayout(){
```

```

return(

  <Stack>

    <Stack.Screen

      name="(global)"

      options={{

        title:"Groupe global",

        headerShown: false,

      }}

    />

  </Stack>

);
}

```

Concernant les groupes de routes, dans une application plus complexe, vous pourriez en avoir plusieurs, séparant les différentes sections de votre application.

Voici un exemple de structure de routes utilisant les groupes de routes :

- app
  - (global)
    - HomeScreen
  - (user)
    - LoginScreen
    - ProfileScreen
  - (admin)

- LoginScreen
- ProfileScreen
- AddArticleScreen
- ModifyArticleScreen
- DeleteArticleScreen

Avec l'implémentation de ce composant `<Stack.Screen/>`, nous allons avoir besoin de créer le chemin spécifié.

Créez un dossier, nommé “(global)”, dans le dossier “app”.

Dans le dossier “(global)”, créez un nouveau fichier nommé “\_layout.tsx”.

Dans ce fichier, créez un nouveau composant nommé “GlobalLayout”, qui devra être exporté par défaut et ne prendra pas de **props**.

Ce composant devra retourner un composant `<Tabs></Tabs>`, fourni par “expo-router”.

```
import { Tabs } from "expo-router";

export default function GlobalLayout(){

  return(

    <Tabs>

    </Tabs>

  );

}
```

Le composant `<Tabs></Tabs>` devra englober un composant `<Tabs.Screen/>`.

Le composant `<Tabs.Screen/>` représente une route de notre application, accessible via un onglet dans une **tabBar** (barre de navigation en bas d'écran, comportant différents onglets).

Nous allons lui fournir une propriété **"name"**, qui contiendra la chaîne de caractères **"index"**.

La valeur de la propriété **"name"** représente le chemin à cibler pour afficher l'écran, soit le composant **"HomeScreen"**, que l'on créera dans le fichier **"app > (global) > index.tsx"**.

```
<Tabs.Screen
  name="index"
/>
```

Nous allons également fournir une propriété **"options"** à **Tab.Screen**, qui contiendra une interpolation d'objet avec les paramètres suivants :

- **title**, qui contiendra la chaîne de caractère **"Accueil"**.

```
title: "Accueil",
```

- **headerTitleStyle**, qui contiendra la chaîne de caractères **"Titillium Web Bold"** (une police de caractère que l'on va implémenter).

```
headerTitleStyle: {
  fontFamily:"Titillium Web Bold"
},
```

- **tabBarLabel**, qui prendra comme valeur une **fonction anonyme**, qui retournera un composant `<Text></Text>` (fourni par “react-native”). Le composant **Text** contiendra le texte “Accueil”.

```
tabBarLabel: () => (<Text>
    Accueil
  </Text>),
```

- **tabBarIcon**, qui prendra comme valeur une **fonction anonyme**, qui retournera un composant `<TabBarIcon/>` (créé à l’initialisation du projet et à importer via le chemin “@/components/navigation/TabBarIcon”). Le composant **TabBarIcon** prendra une propriété “name”, qui aura pour valeur “home-outline” (icône de maison, style outline, fournie par Ionicons) et une propriété “size”, qui aura pour valeur “{14}”.

```
tabBarIcon: () => (<TabBarIcon name="home-outline" size={14}/>),
```

- **tabBarItemStyle**, qui prendra un objet comme valeur, contenant une décomposition de “styles.tabBarItemCustom” (instruction de style que l’on créera dans le feuille de style du fichier “(global) > \_layout.tsx”) et un paramètre “backgroundColor”, qui aura pour valeur “#FFFFFF”.

```
tabBarItemStyle: {
    ...styles.tabBarItemCustom,
    backgroundColor: "#FFFFFF",
  },
```

- **tabBarIconStyle**, qui prendra un objet comme valeur, contenant une décomposition de “styles.tabBarIconCustom” (instruction de style que

l'on créera dans le feuille de style du fichier “(global) > \_layout.tsx”).

```
tabBarIconStyle: {  
  
    ...styles.tabBarIconCustom  
  
},
```

- **tabBarLabelStyle**, qui prendra comme valeur la chaîne de caractères “**Titillium Web Light**” (une police que nous allons implémenter).

```
tabBarLabelStyle:{  
  
    fontFamily: "Titillium Web Light"  
  
},
```

### [Astuce]:

Si vous souhaitez masquer un onglet de la barre de navigation, vous pouvez ajouter un paramètre “**href**”, qui prendra une valeur “**null**”, dans l’objet fourni à la propriété “**options**”.

```
href: null,
```

Sortez de votre composant **GlobalLayout** et placez-vous en dessous pour créer une constante “**styles**”, qui contiendra une feuille de style créée avec la méthode “**create**” de la classe “**StyleSheet**”, fournie par “**react-native**”.

La méthode **create** prendra un objet en paramètre et cet objet contiendra nos instructions de style.

```
const styles = StyleSheet.create({  
  
});
```

Dans cette feuille de style, nous allons ajouter les instructions suivantes :

- **tabBarItemCustom**, qui prendra un objet comme valeur et contiendra :
  - **flex**, de valeur “1”.
  - **flexDirection**, de valeur “column”.
  - **justifyContent**, de valeur “center”.
  - **alignItems**, de valeur “center”.
- **tabBarIconCustom**, qui prendra un objet comme valeur et contiendra :
  - **height**, de valeur “18”.

```
const styles = StyleSheet.create({  
  
  tabBarItemCustom: {  
  
    flex:1,  
  
    flexDirection: "column",  
  
    justifyContent:"center",  
  
    alignItems:"center",  
  
  },  
  
  tabBarIconCustom: {  
  
    height:18,  
  
  }  
  
});
```

Votre fichier “app > (global) > \_layout.tsx” devrait désormais contenir ce code :

```
import { TabBarIcon } from "@components/navigation/TabBarIcon";

import { Tabs } from "expo-router";

import { StyleSheet, Text } from "react-native";

export default function GlobalLayout() {

  return (

    <Tabs>

      <Tabs.Screen

        name="index"

        options={{

          title: "Accueil",

          headerTitleStyle: {

            fontFamily:"Titillium Web Bold"

          },

          tabBarLabel: () => (<Text>

            Accueil

          </Text>),

          tabBarIcon: () => (<TabBarIcon name="home-outline" size={14}/>),

          tabBarItemStyle: {
```



```

        ...styles.tabBarItemCustom,

        backgroundColor: "#FFFFFF",

    },

    tabBarIconStyle: {

        ...styles.tabBarIconCustom

    },

    tabBarLabelStyle: {

        fontFamily: "Titillium Web Light"

    },

    href: null,

    }}

    />

</Tabs>

);
}

const styles = StyleSheet.create({

    tabBarItemCustom: {

        flex: 1,

        flexDirection: "column",

        justifyContent: "center",

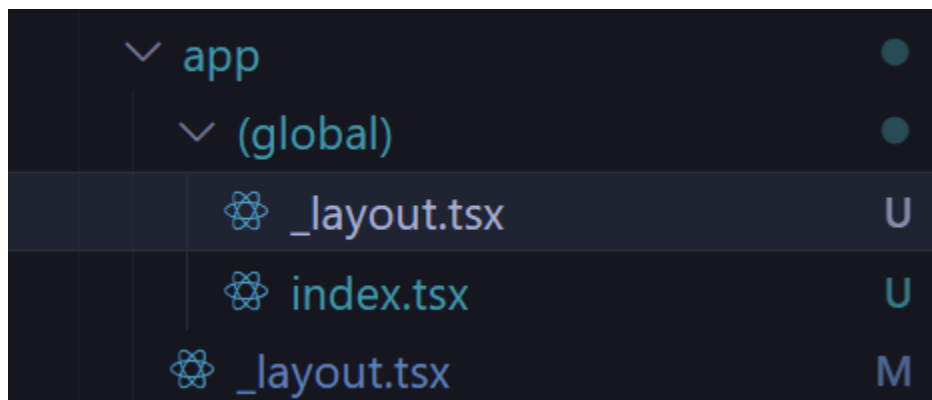
```

```
    alignItems:"center",  
  },  
  tabBarIconCustom: {  
    height:18,  
  }  
});
```

## Création du premier composant d'écran

Créez un dossier nommé “home”, dans le dossier “app > (global)”.

Dans ce dossier, créez un fichier nommé “index.tsx”.



Chaque dossier comportant un fichier “index.tsx” (ou [index.jsx/index.js](#)) ouvert par React Native sera considéré comme la racine du dossier et s'exécutera.

Nous verrons dans la création de cette application comment utiliser cette exécution automatique pour déclencher certaines actions.

Dans ce fichier, créez le composant “**HomeScreen**”, qui devra être exporté par défaut et ne prendra pas de **props**.

**HomeScreen** devra retourner un composant `<View></View>` (fourni par “react-native”), qui englobera un composant `<Text></Text>` (fourni par “react-native”).

Le composant **Text** contiendra le texte “**Accueil**”.

```
import { Text, View } from "react-native";

export default function HomeScreen(){

  return(

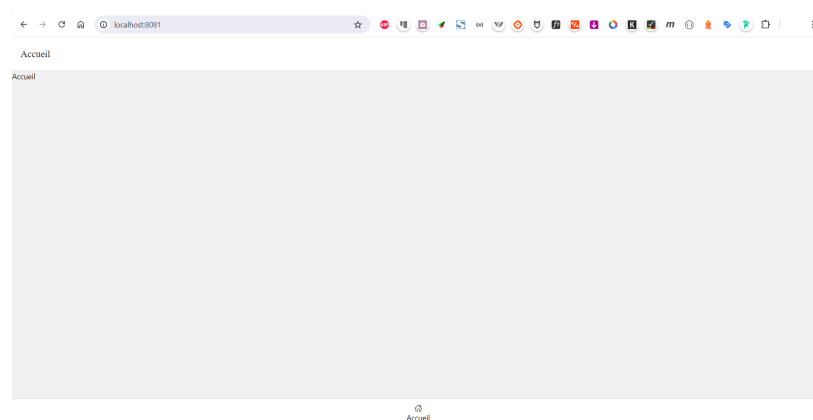
    <View>

      <Text>Accueil</Text>

    </View>

  );
}
```

Ouvrez votre application, vous devriez voir apparaître votre premier composant d’écran !



(commande **npm run web**, si vous n’avez pas lancé votre serveur local).

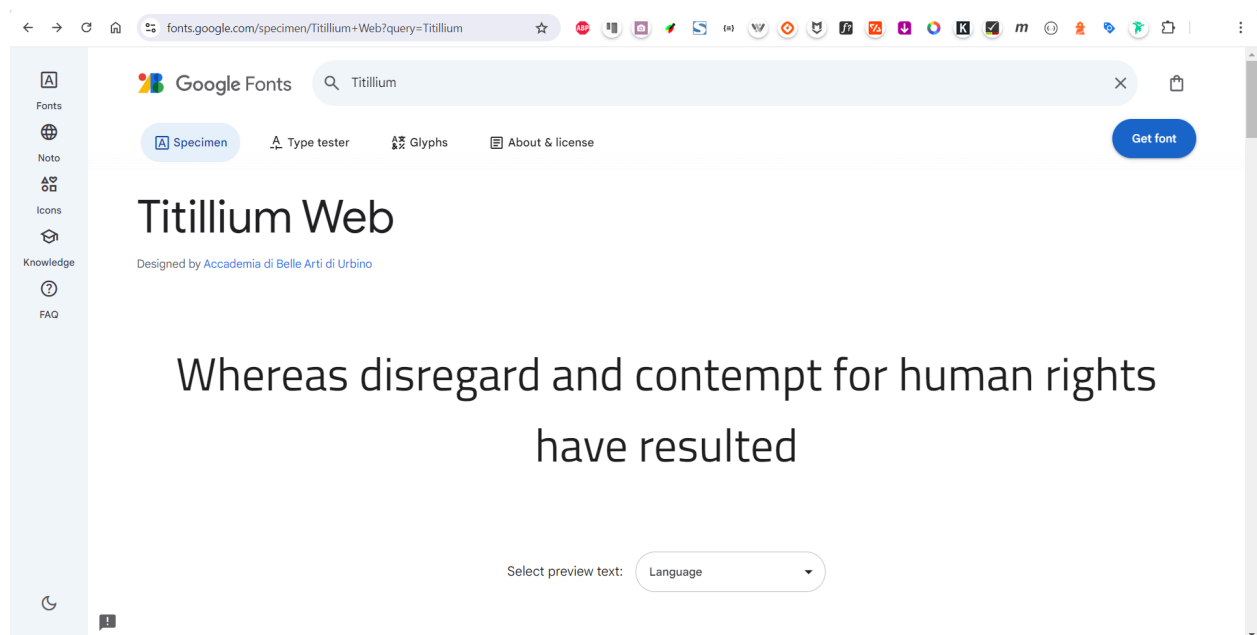
## Implémentation des polices personnalisées

Dans ce projet, nous utiliserons une police de caractères qui se nomme “**Titillium Web**”, et qui comporte plusieurs formats d’affichage.

Avant d’implanter les polices personnalisées dans l’application, il faudra les télécharger pour les placer dans le dossier “**assets > fonts**”.

Rendez-vous sur Google Fonts, à l’adresse suivante pour trouver la police **Titillium Web** :

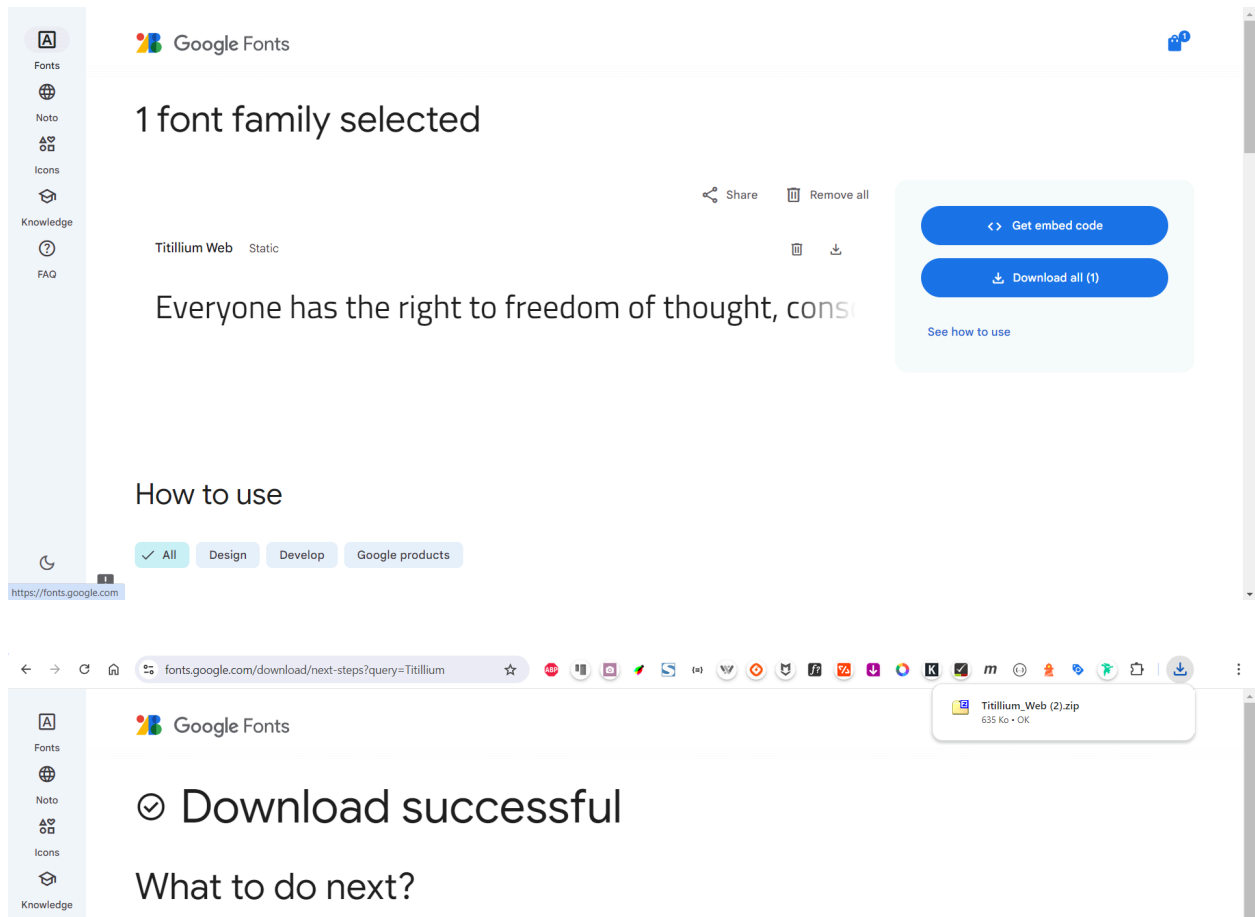
<https://fonts.google.com/specimen/Titillium+Web?query=Titillium>.



Cliquez sur le bouton “**Get font**”, en haut à droite de la page pour ajouter la police à votre panier.

Vous serez alors redirigé vers votre page de sélection de polices.

Cliquez sur “**Download all**” pour télécharger un fichier **zip** contenant les polices **Titillium Web**.



Ouvrez l'archive téléchargée et extrayez les polices suivantes, dans le dossier “assets > fonts” de votre projet.

- **Titillium Web Light**
- **Titillium Web Regular**
- **Titillium Web SemiBold**
- **Titillium Web Bold**

C:\Users\valen\Downloads\Titillium_Web (2).zip\										
Fichier Édition Affichage Favoris Outils Aide										
Ajouter Extraire Tester Copier Déplacer Supprimer Informations										
C:\Users\valen\Downloads\Titillium_Web (2).zip\										
Nom	Taille	Compressé	Modifié le	Créé le	Accédé le	Attributs	Chiffrer	Commentai...	CRC	Method
OFI Lot	4 524	4 524	2024-08-06 2...				-		ABAFB72B	Store
TitilliumWeb-Black.ttf	43 664	43 664	2023-08-25 1...				-		75F07871	Store
TitilliumWeb-Bold.ttf	53 896	53 896	2023-08-25 1...				-		65AD6510	Store
TitilliumWeb-BoldItalic.ttf	62 924	62 924	2023-08-25 1...				-		E0CFC524	Store
TitilliumWeb-ExtraLight.ttf	56 724	56 724	2023-08-25 1...				-		308A3D7B	Store
TitilliumWeb-ExtraLightItalic.ttf	60 848	60 848	2023-08-25 1...				-		E1107D48	Store
TitilliumWeb-Italic.ttf	65 284	65 284	2023-08-25 1...				-		6EFD231	Store
TitilliumWeb-Light.ttf	57 600	57 600	2023-08-25 1...				-		86D4A8FE	Store
TitilliumWeb-LightItalic.ttf	64 560	64 560	2023-08-25 1...				-		20E9DAD4	Store
TitilliumWeb-Regular.ttf	57 392	57 392	2023-08-25 1...				-		835571D2	Store
TitilliumWeb-SemiBold.ttf	56 752	56 752	2023-08-25 1...				-		030A3BE5	Store
TitilliumWeb-SemiBoldItalic.ttf	64 816	64 816	2023-08-25 1...				-		C0EE2008	Store

4 / 12 objet(s) sélectionné(s) 225 640 56 752 2023-08-25 19:51:24

Nom	Modifié le	Type	Taille
SpaceMono-Regular.ttf	26/10/1985 10:15	Fichier de police T...	92 Ko
TitilliumWeb-SemiBold.ttf	25/08/2023 19:51	Fichier de police T...	56 Ko
TitilliumWeb-Bold.ttf	25/08/2023 19:51	Fichier de police T...	53 Ko
TitilliumWeb-Light.ttf	25/08/2023 19:51	Fichier de police T...	57 Ko
TitilliumWeb-Regular.ttf	25/08/2023 19:51	Fichier de police T...	57 Ko

Pour vous assurer de voir la police, si vous utilisez le serveur web pour tester l'application, pensez à installer les polices sur votre machine.

Titillium Web SemiBold (OpenType)
Imprimer
Installer

Nom de la police : Titillium Web SemiBold  
Version : Version 1.002;PS 57.000;hotconv 1.0.70;makeotf.lib2.5.55311  
Disposition OpenType, TrueType Contours

abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ  
1234567890.,:;' " (!?) +-\* / = é è à ç ù ê ù â

12 Voix ambiguë d'un cœur qui au zéphyr préfère les jattes de kiwis. 1234567890  
18 Voix ambiguë d'un cœur qui au zéphyr préfère les jattes de kiwis. 1234567890  
24 Voix ambiguë d'un cœur qui au zéphyr préfère les jattes de kiwis. 1234567890  
36 Voix ambiguë d'un cœur qui au zéphyr préfère les jattes de kiwis. 1234567890  
48 Voix ambiguë d'un cœur qui au zéphyr préfère les jattes de kiwis. 1234567890  
60 Voix ambiguë d'un cœur qui au zéphyr préfère les jattes de kiwis. 1234567890

Ensuite, coupez le serveur local et dans le chemin de votre projet (attention à être placé à la racine de votre projet dans le terminal), entrez la commande :

```
npx expo install expo-font
```

Ouvrez ensuite le fichier “**app.json**” et vérifiez la présence de “**expo-font**”, dans la partie “**plugins**”.

```
"plugins": [
  "expo-router",
  "expo-font"
],
```

Ouvrez le fichier “**\_layout.tsx**”, du dossier “(**global**)”.

Ajoutez la ligne d’import suivante :

```
import { useFonts } from "expo-font";
```

Ensuite, placez-vous entre l’ouverture du composant et le retour du composant pour déclarer les polices à utiliser dans le fichier.

```
export default function GlobalLayout() {

  /**
   * Implémentation des polices ici
   */

  return (
```

L'initialisation des polices au sein du fichiers utilisent une structure similaire à l'initialisation d'un **useState** de React.

On aura besoin de déstructurer, dans un tableau, une variable **"loaded"**, qui stockera le retour de l'état de chargement des polices et une variable **"error"**, qui stockera les erreurs capturées lors de la tentative d'initialisation des polices.

Pour initialiser ces variables, on utilisera le hook **useFonts**, fourni par **expo-font**.

En valeur par défaut, il faudra fournir un objet, contenant comme paramètre le nom de la police, formatée pour l'utilisation dans l'application (exemple : **"Titillium Web Bold"**) et en valeur, on utilisera la méthode **require("")** pour importer la police associée (exemple : **require("@/assets/fonts/TitilliumWeb-Bold.ttf")**).

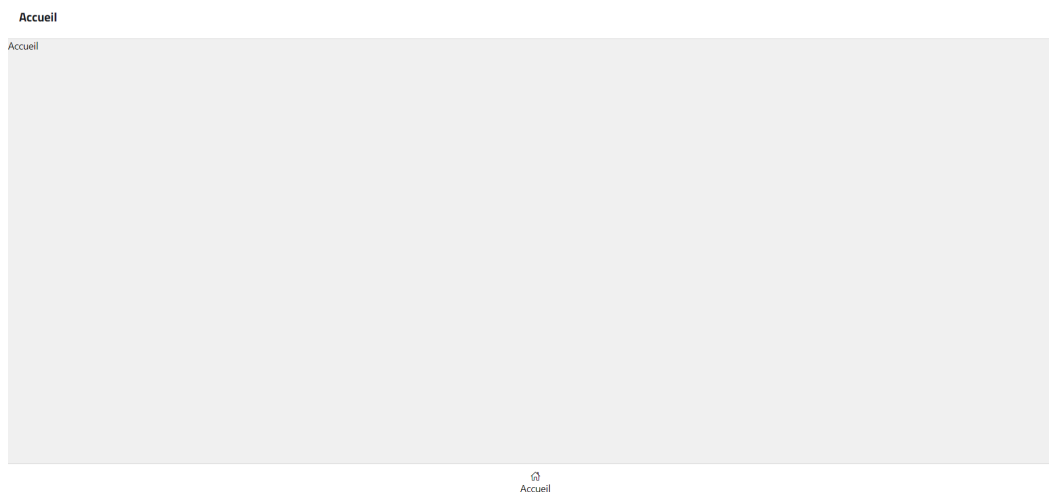
Dans le fichier **\_layout.tsx**, du dossier (**global**), on aura besoin de :

- **Titillium Web Light**
- **Titillium Web Bold**

```
const [loaded, error] = useFonts({  
  "Titillium Web Light": require("@/assets/fonts/TitilliumWeb-Light.ttf"),  
  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),  
});
```

Si vous rechargez votre serveur local, vous devriez voir la police du titre de la page et du label de la tabBar se modifier.





Désormais, vous utiliserez la même méthode d'implémentation pour les polices, sur chaque fichier nécessitant une police personnalisée.

### Création du premier composant personnalisé

Actuellement, notre fichier **index.tsx**, du fichier (**global**) (représentant l'écran **HomeScreen**) contient uniquement un composant `<View></View>` et un composant `<Text></Text>`.

Nous allons donc créer notre premier composant personnalisé pour étoffer le contenu de cet écran.

Pour la version finale du composant **HomeScreen**, nous souhaitons arriver à ce résultat :



Ce composant retournera un composant **View**, qui prendra une propriété **style**, de valeur “`{{ ...styles.view }}`”, car nous lui fournirons du **style inline** par la suite.

```
return(  
    <View style={{  
        ...styles.view  
    }}>  
  
    </View>  
  
);
```

Ce composant **View** englobera un composant **Text**, qui englobera une interpolation de “**props.text**” et aura une propriété **style**, de valeur “**{{...styles.text}}**”.

N'oubliez pas de vérifier les imports des composants **View** et **Text**, depuis "react-native".

```
import { Text, View } from "react-native";

export default function WelcomeText({props}){

  return(

    <View style={{
```

```

        ...styles.view
    }}>

    <Text

        style={{

            ...styles.text

        }}

    >

        {props.text}

    </Text>

</View>

);

};

```

En dessous du composant, créez une constante **styles**, qui prendra pour valeur “**StyleSheet.create({})**”.

**StyleSheet** est importé également depuis “**react-native**”.

```

const styles = StyleSheet.create({

});

```

Dans cette nouvelle feuille de style, ajoutez les instructions suivantes :

```
const styles = StyleSheet.create({  
  
  view: {  
  
    width: "auto",  
  
    maxWidth: "100%",  
  
    backgroundColor: "transparent",  
  
    color: "#1D3124",  
  
    paddingVertical: 8,  
  
    paddingHorizontal: 12,  
  
    marginHorizontal: "auto",  
  
    borderRadius: 256,  
  
  },  
  
  text: {  
  
    fontSize: 16,  
  
    color: "#1D3124",  
  
    fontFamily: "Titillium Web Regular",  
  
  },  
  
});
```

Nous allons désormais créer la logique du composant **WelcomeText**, entre l'ouverture du composant et son retour.

```
export default function WelcomeText({props}){

  /**
   * Logique du composant
   */

  return(
```

Pour commencer, nous allons préparer les propriétés qui pourront être fournies au composant **WelcomeText**.

Nous aurons besoin des propriétés :

- **darkMode**, pour gérer le passage manuel en mode sombre du composant.
- **fontFamilyLight**, qui nous permettra de spécifier que nous souhaitons utiliser la police **Titillium Web Light**.
- **fontFamilyBold**, qui nous permettra de spécifier que nous souhaitons utiliser la police **Titillium Web Bold**.
- **addStyles**, qui nous permettra de fournir des instructions de styles supplémentaires au composant, en fonction des besoins.

```
const {darkMode, fontFamilyLight, fontFamilyBold, addStyles} = props;
```

Nous allons également importer **useFonts**, qui est fournie par **expo-font**.

```
import { useFonts } from "expo-font";
```

Et initialiser 3 polices de caractères :

- Titillium Web Light
- Titillium Web Regular
- Titillium Web Bold

```
const [loaded, error] = useFonts({  
  
  "Titillium Web Light": require("@/assets/fonts/TitilliumWeb-Light.ttf"),  
  
  "Titillium Web Regular": require("@/assets/fonts/TitilliumWeb-Regular.ttf"),  
  
  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),  
  
});
```

Ensuite, dans la propriété **style** du composant **View**, nous allons ajouter deux instructions de styles conditionnelles, en prenant en compte la valeur de la props **darkMode**.

- **backgroundColor**, qui passera en couleur “**#1D3124**”, si la propriété **darkMode** existe et est égale à **true**.
- **color**, qui passera en couleur “**#FFFFFF**”, si la propriété **darkMode** existe et est égale à **true**.

```
<View style={{  
  
  ...styles.view,  
  
  backgroundColor: darkMode && "#1D3124",  
  
  color: darkMode && "#FFFFFF"  
  
}}>
```

Pour le composant **Text**, nous allons suivre la même logique et ajouter les instructions suivantes :

- **color**, qui passera en couleur “#FFFFFF”, si la props **darkMode** est égale à **true**.
- **fontFamily**, qui transformera la police de caractère en :
  - “**Titillium Web Bold**”, si la props **darkMode** ou la props **fontFamilyBold** est égale à **true**.
  - “**Titillium Web Light**”, si la props **fontFamilyLight** est égale à **true**.
- **addStyles**, que l’on devra apporter en utilisant le **spread operator** de Javascript (notation : “...”) et qui permettra une application des styles complémentaires, qui écraseront les styles par défaut, car placés en dernier dans l’ordre des styles.

```
color: darkMode && "#FFFFFF",

fontFamily: ((darkMode || fontFamilyBold) && "Titillium Web Bold") ||
(fontFamilyLight && "Titillium Web Light"),
...addStyles
```

Le composant **WelcomeText** est enfin prêt à être utilisé.

Vous devriez avoir ce code dans le fichier **WelcomeText.jsx** :

```
import { StyleSheet, Text, View } from "react-native";

import { useFonts } from "expo-font";

export default function WelcomeText({props}){
```



```

const {darkMode, fontFamilyLight, fontFamilyBold, addStyles} = props;

const [loaded, error] = useFonts({
  "Titillium Web Light": require("@/assets/fonts/TitilliumWeb-Light.ttf"),
  "Titillium Web Regular": require("@/assets/fonts/TitilliumWeb-Regular.ttf"),
  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),
});

return(
  <View style={{
    ...styles.view,
    backgroundColor: darkMode && "#1D3124",
    color: darkMode && "FFFFFF"
  }}>

    <Text

      style={{
        ...styles.text,
        color: darkMode && "FFFFFF",
        fontFamily: ((darkMode || fontFamilyBold) && "Titillium Web Bold") ||
(fontFamilyLight && "Titillium Web Light"),
        ...addStyles

```

```

    }}

    >

    {props.text}

  </Text>

</View>

);

};

const styles = StyleSheet.create({

});

```

## Création du composant CtaButton

Continuons sur notre lancée et créons un deuxième composant personnalisé, nommé “**CtaButton**”, dans un nouveau fichier nommé “**CtaButton.jsx**”, placé dans le dossier “**components > globals**”.

Créez le composant **CtaButton**, qui sera exporté par défaut et prendra un paramètre **props**.

Aujourd’hui, le composant **Button** de React Native est déprécié et la documentation officielle nous informe qu’il faut le remplacer par le composant **Pressable**.

Notre composant **CtaButton** retournera donc un composant **Pressable**, qui englobera un composant **Text**, fourni par “**react-native**”.

```
export default function CtaButton({props}){  
  
  return(  
  
    <Pressable>  
  
      <Text>  
  
        </Text>  
  
    </Pressable>  
  
  );  
}
```

Créez directement une constante **styles**, qui contiendra la feuille de style du composant, avec les instructions suivantes :

```
const styles = StyleSheet.create({  
  
  customButton: {  
  
    width: "auto",  
  
    backgroundColor: "#E59560",  
  
    marginVertical: 8,  
  
    padding: 12,  
  
  },  
  
  customButtonText: {
```

```

    fontFamily: "Titillium Web Bold",

    fontWeight: "bold",

    fontSize: 16,

    color: "#FFFFFF",

  }
});

```

Et pensez à vérifier les premiers imports de votre composant.

```
import { Pressable, StyleSheet, Text } from "react-native";
```

Ensuite, nous allons commencer par la création de la logique de notre bouton (entre l'ouverture du composant et le retour du composant).

```

export default function CtaButton({props}){

  /**

   * Logique du composant

  */

  return(

```

Nous utiliserons la police de caractères **Titillium Web Bold** par défaut, il faudra donc importer **useFonts** depuis **expo-font** et initialiser notre police.

```
import { useFonts } from "expo-font";
```

```
const [loaded, error] = useFonts({
  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf")
});
```

Ensuite, nous allons préparer la déstructuration des props utilisables par notre composant.

- **text**, qui sera utilisé pour fournir le texte affiché dans la zone **Pressable**.
- **borderRadius**, pour modifier en fonction de la situation, le rayon de la bordure du bouton.
- **withIcon**, pour spécifier si notre bouton utilisera une icône et l'afficher conditionnellement.
- **iconType**, qui sera utilisée de paire avec **withIcon**, pour spécifier quel type d'icône utiliser.
- **actionOnPress**, pour fournir une action (méthode) à déclencher lors de la pression sur la zone **Pressable** (clic sur le bouton).
- **disabled**, qui nous permettra de modifier conditionnellement l'état d'activation du bouton.
- **addStyles**, qui nous servira à fournir du style supplémentaire.
- **addTextStyles**, qui, comme **addStyles**, permettra de fournir du style supplémentaire, pour le texte du bouton.

```
const { text, borderRadius, withIcon, iconType, actionOnPress, disabled,
addStyles, addTextStyles } = props;
```

Nous allons également stocker un état, qui conservera une valeur booléenne, pour savoir si le bouton est pressé ou non, avec **useState** fourni par "react".

```
const [buttonIsPressed, setButtonIsPressed] = useState(false);
```

Nous utiliserons l'état **buttonIsPressed** pour appliquer un style conditionnel sur la zone **Pressable**.

Par défaut, son état sera initialisé à **false**.

Pour prévoir l'utilisation d'une icône à l'intérieur du bouton, nous allons créer une méthode, nommée "**handleShowIcon**", qui prendra un paramètre nommé "**type**".

Les icônes seront affichées en utilisant le composant **<Entypo />**, fourni par "**@expo/vector-icons**".

Vous pouvez consulter la liste d'icônes à l'adresse <https://icons.expo.fyi/Index> et filtrer les icônes par la librairie **Entypo**.

```
import { Entypo } from "@expo/vector-icons";
```

Dans cette méthode, on utilisera un **switch** (de Javascript), pour vérifier la valeur du paramètre **type** et définir une icône spécifique en fonction des cas suivants :

- **edit** : on retournera un composant **<Entypo name="edit" size={18} color="#FFFFFF"/>**.
- **history**: on retournera un composant **<Entypo name="clock" size={18} color="#FFFFFF"/>**.
- **profile**: on retournera un composant **<Entypo name="user" size={18} color="#FFFFFF"/>**.
- **setting** ou **default** : on retournera un composant **<Entypo name="cog" size={18} color="#FFFFFF"/>**.

```
const handleShowIcon = (type) => {

  switch (type) {

    case "edit":

      return (<Entypo name="edit" size={18} color="#ffffff" />);

    case "history":

      return (<Entypo name="clock" size={18} color="#ffffff" />);

    case "profile":

      return (<Entypo name="user" size={18} color="#ffffff" />);

    case "settings":

    default:

      return (<Entypo name="cog" size={18} color="#ffffff" />);

  }

};
```

Cette méthode nous permettra d'afficher une icône à l'intérieur d'un bouton, par exemple :



Nous allons désormais travailler sur le retour du composant **CtaButton**, pour ajouter les propriétés, conditions et interpolations nécessaires au fonctionnement du composant.

### Composant Pressable :

Ajoutez une propriété **style**, qui contiendra une **interpolation d'objet**.

Dans cet objet, ajoutez les instructions suivantes :

- **...styles.customButton** : pour fournir les styles de notre feuille de style, liés au paramètre **customButton**.
- **borderRadius** : qui utilisera soit la valeur de la props **borderRadius**, ou qui aura une valeur par défaut de **8**.
- **borderWidth** : qui prendra une valeur de **1**.
- **borderColor** : qui prendra la couleur “**#1D3124**” si la props **buttonIsPressed** est égale à **true**. Dans le cas contraire, on appliquera la couleur “**transparent**”.
- **backgroundColor**: qui appliquera la couleur “**#79797A**”, si la props **disabled** est égale à **true**. Dans le cas contraire, on lui appliquera “**styles.customButton.backgroundColor**”.
- **...addStyles** : pour utiliser les styles complémentaires fournis via la props **addStyles**.

Ensuite, notre zone pressable prendra les propriétés suivantes :

- **onPress**, qui prendra comme valeur “**{() => actionOnPress && actionOnPress()}**”, pour déclencher l'action envoyée via la props **actionOnPress** (uniquement si **actionOnPress** est définie, avec la notation “**actionOnPress &&**”).
- **onPressIn**, qui prendra la valeur “**{() => setButtonIsPressed(true)}**”, pour modifier l'état **buttonIsPressed** et lui fournir la valeur **true**.



- **onPressOut**, qui prendra la valeur “`() => setButtonIsPressed(false)`”, pour modifier l’état **buttonIsPressed** et lui fournir la valeur **false**.
- **disabled**, qui prendra la valeur “`{disabled}`”, pour activer ou désactiver le bouton via la props **disabled**.

```
<Pressable

  style={{

    ...styles.customButton,

    borderRadius: borderRadius ?? 8,

    borderWidth: 1,

    borderColor: buttonIsPressed ? "#1D3124" : "transparent",

    backgroundColor: disabled ? "#79797A" :
styles.customButton.backgroundColor,

    ...addStyles

  }}

  onPress={() => actionOnPress()}

  onPressIn={() => setButtonIsPressed(() => true)}

  onPressOut={() => setButtonIsPressed(() => false)}

  disabled={disabled}

/>
```

## Composant Text :

Ajoutez lui une propriété **style**, qui prendra une interpolation d'objet comme valeur.

Cet objet contiendra les paramètres suivants :

- ...styles.customButtonText
- ...addTextStyles

```
<Text style={{
  ...styles.customButtonText,
  ...addTextStyles
}}>
```

Le composant **Text** englobera son contenu, qui sera représenté par une interpolation conditionnelle de la props **"withIcon"**, qui déclenchera la méthode **"handleShowIcon(iconType)"**, si **withIcon** est égale à **true**.

Cette interpolation conditionnelle sera suivie par l'interpolation de la props **"text"**.

```
{withIcon && handleShowIcon(iconType)} {text}
```

Le composant **CtaButton** est désormais prêt pour l'utilisation et devrait ressembler à :

```
import { Entypo } from "@expo/vector-icons";
import { useFonts } from "expo-font";
```

```

import { useState } from "react";

import { Pressable, StyleSheet, Text } from "react-native";

export default function CtaButton({ props }) {

  const [loaded, error] = useFonts({

    "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf")

  });

  const { text, borderRadius, withIcon, iconType, actionOnPress, disabled,
addStyles, addTextStyles } = props;

  const [buttonIsPressed, setButtonIsPressed] = useState(false);

  const handleShowIcon = (type) => {

    switch (type) {

      case "edit":

        return (<Entypo name="edit" size={18} color="#ffffff" />);

      case "history":

        return (<Entypo name="clock" size={18} color="#ffffff" />);

      case "profile":

```

```

        return (<Entypo name="user" size={18} color="#ffffff" />);

    case "settings":

    default:

        return (<Entypo name="cog" size={18} color="#ffffff" />);

    }

};

return (

    <Pressable

        style={{

            ...styles.customButton,

            borderRadius: borderRadius ?? 8,

            borderWidth: 1,

            borderColor: buttonIsPressed ? "#1D3124" : "transparent",

            backgroundColor: disabled ? "#79797A" :
styles.customButton.backgroundColor,

            ...addStyles

        }}

        onPress={() => actionOnPress && actionOnPress()}

        onPressIn={() => setButtonIsPressed(true)}

        onPressOut={() => setButtonIsPressed(false)}

```

```

        disabled={disabled}

    >

    <Text style={{

        ...styles.customButtonText,

        ...addTextStyles

    }}>

        {withIcon && handleShowIcon(iconType)} {text}

    </Text>

</Pressable>

);
}

const styles = StyleSheet.create({

    customButton: {

        width: "auto",

        backgroundColor: "#E59560",

        marginVertical: 8,

        padding: 12,

    },

    customButtonText: {

        fontFamily: "Titillium Web Bold",

```

```
    fontWeight: "bold",

    fontSize: 16,

    color: "#FFFFFF",
  }
});
```

### Création d'une liste d'importation dans le dossier components > globals

Pour permettre l'importation des composants personnalisés du dossier **globals**, en ciblant uniquement le chemin “@/components/globals”, dans nos autres fichiers, nous pouvons créer une **liste d'exportation** des composants du dossier.

Créez un nouveau fichier, nommé “**index.js**”, dans le dossier “**globals**”.

Dans ce fichier, importez vos composants **WelcomeText** et **CtaButton**, puis créez un “**export**” classique, qui contiendra les deux composants.

```
import WelcomeText from "./WelcomeText";

import CtaButton from "./CtaButton";


export {

  WelcomeText,

  CtaButton,

}
```

Désormais, pour utiliser vos nouveaux composants, il vous suffira de les importer sous cette forme : `import {WelcomeText, CtaButton} from "@components/globals"`; (le fichier `index.js` sera automatiquement exécuté par React).

### Première utilisation des composants personnalisés

Nous avons désormais deux nouveaux composants à utiliser dans notre application.

Rendez-vous dans le fichier `"index.jsx"`, du dossier `"app > (global)"`.

Importez les composants **WelcomeText** et **CtaButton**.

```
import { WelcomeText, CtaButton } from "@components/globals";
```

Dans le composant **HomeScreen**, ajoutez deux **WelcomeText**.

Le premier aura la chaîne de caractères **"Bonjour"**, en propriété **text**.

Le second aura la chaîne de caractères **"Quel est votre prénom?"**, en propriété **text**.

(Attention, les propriétés doivent être définies dans la propriété **props**, qui prendra une interpolation d'objet comme valeur, contenant les propriétés attendues sous forme de paramètre de cet objet)

Ajoutez également un composant **CtaButton**, qui prendra la chaîne de caractères **"Commencer"** en propriété **text**.

```

export default function HomeScreen() {

  return (

    <View>

      <WelcomeText

        props={{

          text: "Bonjour.",

        }}

      />

      <WelcomeText

        props={{

          text: "Quel est votre prénom ?",

        }}

      />

      <CtaButton

        props={{

          text: "Commencer",

        }}

      />

    </View>

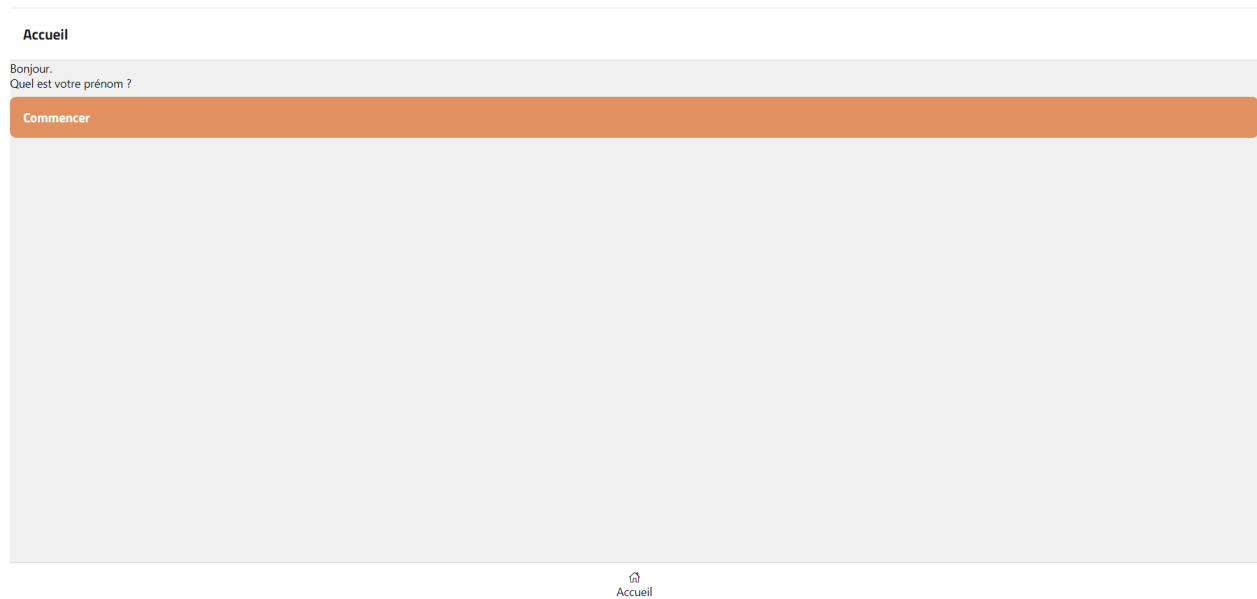
  );

}

```



Vous devriez désormais avoir cet affichage, sur l'application :



Ajoutons un peu de style à la page d'accueil.

Sur le composant **View**, ajoutez une propriété **style**, prenant comme valeur “`{styles.container}`”.

```
<View style={styles.container}>
```

L'objectif de la session étant d'apprendre à créer une application mobile, je vous fournirais désormais un style par défaut pour les écrans et les éléments.

Je vous invite à créer la propre identité graphique de votre application, pour qu'elle vous ressemble.

Ajoutez cette feuille de style au composant **HomeScreen** (style définitif).

```
const styles = StyleSheet.create({

  container: {

    flex: 1,

    justifyContent: "center",

    alignItems: "center",

    backgroundColor: "#F6F4E8",

  },

  firstnameInput: {

    width: 240,

    height: 40,

    borderRadius: 8,

    backgroundColor: "ffffff",

    textAlign: "center",

    paddingHorizontal: 16,

    paddingVertical: 12,

    borderWidth: 1,

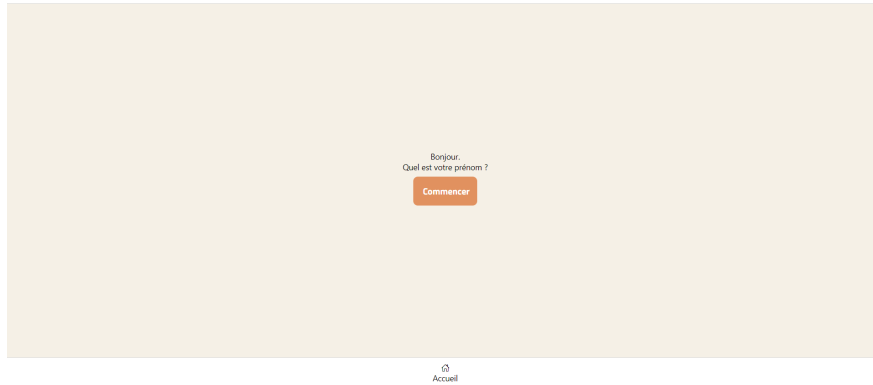
    borderColor: "#BACEC1",

    fontFamily: "Titillium Web Regular",

  },

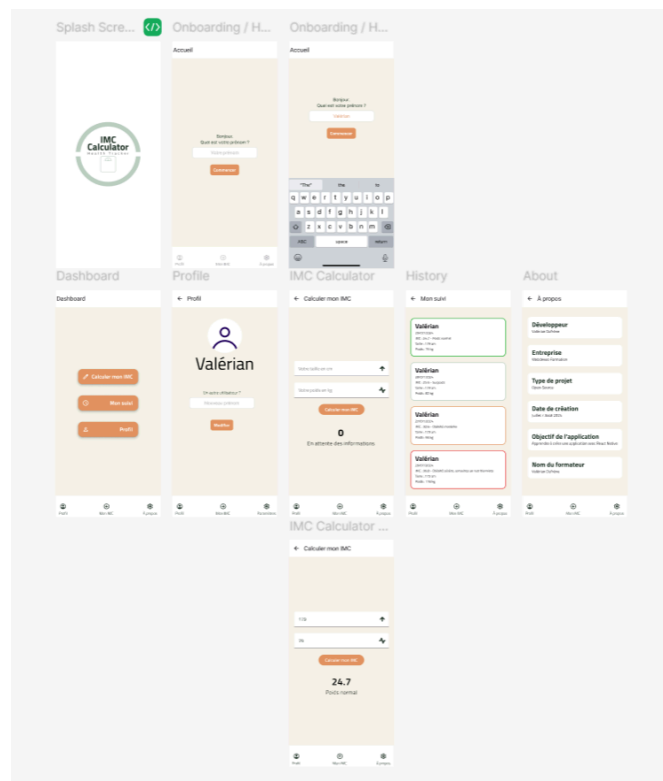
});
```

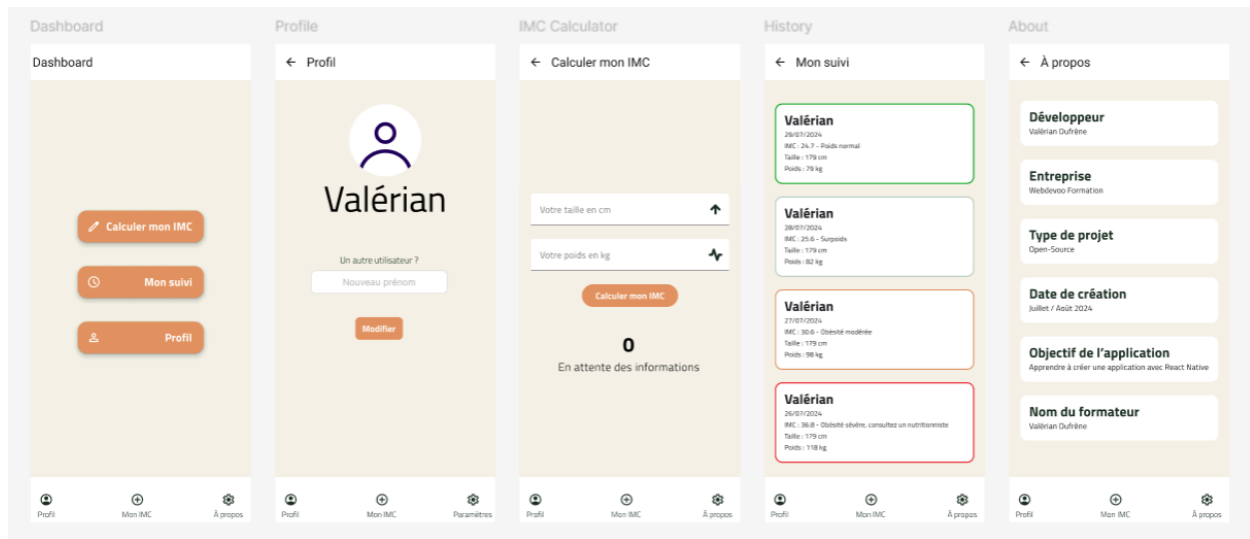
Accueil



Vous devriez désormais avoir ce résultat à l'écran, en vérifiant votre application.

Voici le style par défaut qui sera appliqué, avec les instructions de style fournies lors de la lecture de ce support (certains éléments peuvent différer avec la maquette) :





Nous allons désormais créer un champ d'entrée utilisateur, avec le composant **TextInput**, pour permettre à l'utilisateur de renseigner son prénom, sur la page d'accueil de l'application.

Quand nous avancerons sur la suite de l'application, nous ferons en sorte que la page d'accueil ne soit affichée au lancement de l'application que lorsque cette donnée n'aura pas été renseignée et stockée sur l'appareil.

Quand l'utilisateur aura fourni son prénom, nous le stockerons dans le **stockage local** de l'application, avec une clé nommée "**@profile**".

Cette clé "**@profile**" nous permettra d'exécuter notre affichage conditionnel de la page d'accueil.

Pour suivre cette logique, nous allons suivre ce cheminement :

- Création du champ d'entrée utilisateur sur la page d'accueil.
- Création d'un script permettant d'utiliser le stockage local de l'application, pour stocker différentes données.

- Création d'un contexte global pour notre application, qui centralisera les données et interagira avec le stockage local de l'application pour initialiser les données.
- Implémentation d'un **indicateur d'activité**, qui sera affiché le temps de la vérification des données stockées.

### Création d'un champ d'entrée utilisateur

Dans le composant d'écran **HomeScreen**, nous allons désormais préparer l'implémentation d'un composant **TextInput**, qui représente un champ d'entrée de texte et qui est fourni par "react-native".

Commencez par ajouter l'import de **TextInput**, depuis "react-native".

```
import { StyleSheet, Text, View, TextInput } from "react-native";
```

Ensuite, nous allons créer un état, avec **useState**, qui se nommera "newFirstname" et aura pour mutateur "setNewFirstname".

```
import { useState } from "react";

export default function HomeScreen() {

  const [newFirstname, setNewFirstname] = useState("");

  return (
```

Entre les composants **WelcomeText** et **CtaButton**, ajoutez un composant **TextInput**.

```
<TextInput />
```

Donnez lui les propriétés suivantes :

- **style**, qui prendra comme valeur “**{styles.firstnameInput}**”.
- **placeholder**, qui prendra comme valeur la chaîne de caractères “**Votre prénom**”.
- **placeholderTextColor**, qui prendra la valeur “**#BACEC1**”.
- **value**, qui prendra la valeur “**{newFirstname}**”.
- **onChangeText**, qui prendra la valeur “**{setNewFirstname}**” et qui aura pour effet de modifier l’état de **newFirstname** à chaque changement dans le champ d’entrée utilisateur.

```
<TextInput  
  
  style={styles.firstnameInput}  
  
  placeholder="Votre prénom"  
  
  placeholderTextColor="#BACEC1"  
  
  value={newFirstname}  
  
  onChangeText={setNewFirstname}  
  
/>
```

Ensuite, nous allons désactiver le **CtaButton**, grâce à sa propriété **disabled**, si la longueur de l’état **newFirstname** est inférieure ou égale à 0.

```
disabled: newFirstname.length <= 0,
```

Et nous allons fournir une propriété **actionOnPress** au **CtaButton**, qui prendra comme valeur `() => handleConfirmFirstname(newFirstname)`.

La méthode **handleConfirmFirstname** devra être créée dans la logique du composant **HomeScreen** et nous permettra d'appliquer un formatage sur le prénom fourni ainsi que d'exécuter certaines actions en lien avec le contexte, le stockage et le routeur de l'application.

```
<CtaButton  
  
  props={{  
  
    text: "Commencer",  
  
    actionOnPress: () =>  
  
      handleConfirmFirstname(newFirstname),  
  
    disabled: newFirstname.length <= 0,  
  
  }}  
  
</>
```

Dans la logique du composant **HomeScreen**, créez la méthode **handleConfirmFirstname**, qui prendra un paramètre nommé “**name**” ayant pour valeur par défaut une **chaîne de caractères vide**.

Cette méthode devra être asynchrone.

Dans cette méthode, on commencera par vérifier si la longueur de **name** est inférieure ou égale à 0.

Si cette condition est vraie, on retournera la fonction, pour stopper son exécution.

Dans le cas contraire, on appliquera un formatage sur les données contenues dans le paramètre **name**.

- Suppression des espaces inutiles, avant et après le texte contenu, grâce à la méthode **trim()** de Javascript | **name.trim()**.
- Suppression des espaces internes du texte, grâce à la méthode **replace()** de Javascript | **name.replace(" ", "")**.
- Modification de la casse, pour passer le texte en minuscule, avec la méthode **toLowerCase()** de Javascript | **name.toLowerCase()**.
- Modification de la première lettre du prénom pour la passer en majuscule, avec la méthode **slice()** de Javascript | **name.slice(0, 1).toUpperCase() + name.slice(1, name.length)**.

```
const handleConfirmFirstname = async (name = "") => {  
  
  if (name.length <= 0) {  
  
    return;  
  
  }  
  
  name = name.trim();
```



```

name = name.replace(" ", "");

name = name.toLowerCase();

name = name.slice(0, 1).toUpperCase() + name.slice(1, name.length);

};

```

Avant d’aller plus loin dans la création de la logique de notre **HomeScreen** et de la méthode **handleConfirmFirstname()**, nous aurons besoin de mettre en place le script permettant l’utilisation du stockage local de l’application et le contexte global qui centralisera les données de l’application.

La méthode **handleConfirmFirstname()** attends encore la logique suivante :

- Attente de l’initialisation de la clé “**@profile**” dans le stockage local, permettant de stocker le prénom fourni par l’utilisateur, si la clé “**@profile**” est inexistante.
- Réinitialisation de l’état **newFirstname** sur une chaîne de caractère vide (action permettant de supprimer la valeur contenue dans le champ d’entrée utilisateur).
- Attente du retour de la méthode **findFirstname()**, fournie par le contexte de l’application (méthode permettant de récupérer la valeur de la clé “**@profile**” dans le stockage local pour centraliser cette donnée dans le contexte actuel).
- Utilisation du routeur pour rediriger sur la route “**dashboard**”, qui affichera l’écran “**DashboardScreen**”.

```

// await initFirstname(name);

// setNewFirstname("");

// await findFirstname();

// router.push("dashboard");

```

## Utilisation du stockage local

Nous allons désormais devoir mettre en place un script partagé pour notre application (qui sera contenu dans le dossier “**shared**”), regroupant des méthodes pour utiliser le stockage local de l’application.

Créez un fichier nommé “**AsyncFunctions.js**” dans le dossier “**shared**”.

Ouvrez votre terminal et installez cette dépendance :

```
npm i @react-native-async-storage/async-storage
```

Lors de la création d’un site internet, nous pouvons utiliser le **stockage local** du navigateur et le **stockage de session**.

En React Native, nous avons la possibilité d’utiliser un **stockage local**, similaire au stockage local d’un navigateur web.

La gestion de ce stockage sera accessible en utilisant la méthode **AsyncStorage**, fournie par “**@react-native-async-storage/async-storage**”.

Importez **AsyncStorage**, dans le fichier **AsyncFunctions**.

```
import AsyncStorage from "@react-native-async-storage/async-storage";
```

Dans ce fichier, nous allons créer toutes les méthodes qui nous permettront d’interagir avec le stockage local de l’application.

Créez une méthode nommée “**setData**”, qui devra être **asynchrone** et ne devra pas être exportée.

**setData** prendra deux paramètres :

- “**key**”, ayant comme valeur par défaut “**@key**”.
- **value**, n’ayant pas de valeur par défaut.

Ouvrez une instruction **try/catch**, pour contenir la logique de la méthode et faciliter l'analyse des erreurs sur le serveur web lors du développement de l'application.

```
try{  
  
    // Logique de la méthode setData  
  
}catch(err){  
  
    console.table(err);  
  
}
```

Dans le bloc **try**, créez une constante nommée “**jsonValue**”, qui stockera la valeur “**JSON.stringify(value)**”.

Ensuite, **attendez** le retour de la méthode **setItem**, d’**AsyncStorage**, en lui fournissant comme paramètres “**key**” et “**jsonValue**” | **await AsyncStorage.setItem(key, value)**.

```
const setData = async (key = "@key", value) => {  
  
    try{  
  
        const jsonValue = JSON.stringify(value);  
  
        await AsyncStorage.setItem(key, jsonValue);  
  
    }catch(err){  
  
        console.table(err);  
  
    }  
  
}
```

La méthode **setData** pourra désormais être utilisée pour stocker une **paire clé/valeur** dans le stockage local.

On utilise la méthode **JSON.stringify(value)** pour aplatir les données fournies sous la forme d'une chaîne de caractère.

De cette manière, on pourra stocker un objet JSON en l'associant simplement à une clé, dans le stockage local.

Attention, cette méthode n'est pas magique et comporte quelques limitations, dont la taille maximale de stockage qui est limitée à **6 MB**, par défaut et la taille de tampon mémoire pour la lecture des données qui est limitée à **2 MB** (<https://react-native-async-storage.github.io/async-storage/docs/limits>).

Créez une méthode nommée “**getData**”, qui sera **asynchrone** et prendra un paramètre “**key**”, ayant comme valeur par défaut “**@key**”.

Ajoutez un bloc **try/catch** pour la gestion des erreurs lors du développement et dans le bloc **try** :

- Créez une constante “**jsonValue**”, qui attendra le retour de la méthode “**getItem**” d'**AsyncStorage**. Cette méthode prendra une **clé** en paramètre.
- Retournez **jsonValue**.

Pour valider le format de données à retourner, nous créerons une méthode nommée “**verifyJsonValue**”, qui aura pour rôle de vérifier si les données retournées sont au format JSON ou contiennent un autre type de données.

Cette méthode prendra le contenu de **jsonValue** en paramètre.

Vous pouvez anticiper la création de cette méthode et l'utiliser pour retourner les données à un format valide.

```
const getData = async (key = "@key") => {
  try{
    const jsonValue = await AsyncStorage.getItem(key);
    return verifyJsonValue(jsonValue);
  }catch(err){
    console.table(err);
  }
}
```

Créez une méthode nommée “**deleteData**”, qui devra être **asynchrone** et prendra un paramètre “**key**”, qui aura la valeur par défaut “**@key**”.

Ajoutez un bloc **try/catch** pour gérer les erreurs lors du développement.

Dans le bloc **try**, nous appellerons simplement la méthode “**removeItem**” d’**AsyncStorage**, qui prendra une **clé** en paramètre.

```
const deleteData = async (key = "@key") => {
  try{
    AsyncStorage.removeItem(key);
  }catch(err){
    console.table(err);
  }
}
```

Créez une méthode nommée “**deleteAllDatas**”, qui sera **asynchrone** et ne prendra pas de paramètres.

Cette méthode permettra de **vider le stockage**, en supprimant l’intégralité des **paires clé/valeur** contenues dans le **stockage local**.

Ajoutez votre bloc **try/catch** pour la gestion des erreurs lors du développement.

Dans le bloc **try**, appelez la méthode “**clear**” d’**AsyncStorage**.

```
const deleteAllDatas = async () => {  
  
  try{  
  
    AsyncStorage.clear();  
  
  }catch(err){  
  
    console.table(err);  
  
  }  
  
}
```

Créez une méthode nommée “**verifyJsonValue**”, qui prendra un paramètre nommé “**jsonValue**”.

Cette méthode aura pour rôle de vérifier le format des données récupérées dans le stockage local et permettra d’appliquer un formatage sur les données retournées.

Si le paramètre “**jsonValue**” est de valeur “**undefined**”, stoppez l’exécution de la méthode en la retournant.

Créez une constante nommée “**isJsonStringified**”, qui prendra comme valeur “**jsonValue.includes(“:”)**”.

Avec la méthode **includes()** de Javascript, on vérifie la présence du symbole “:” dans les données.

Effectuez un retour conditionnel en fonction de la valeur (booléenne) de la constante “**isJsonStringified**” :

- **true** : retourner “ **JSON.parse(jsonValue)** ”, pour pouvoir utiliser les données au format objet.
- **false** : retourner “ **jsonValue.replaceAll(“\”, “”)** ”, pour supprimer tous les guillemets doubles présents dans les données.

```
const verifyJsonValue = (jsonValue) => {  
  
  if(!jsonValue){  
  
    return;  
  
  }  
  
  const isJsonStringified = jsonValue.includes(":");  
  
  return isJsonStringified ? JSON.parse(jsonValue) : jsonValue.replaceAll("\",  
  "");  
  
}
```

Nous avons désormais les méthodes utilitaires d'**AsyncStorage** nous permettant d'utiliser le stockage local de l'application.

Nous allons désormais avoir besoin de créer des méthodes utilitaires, qui utiliseront ces méthodes d'interaction avec le stockage, que l'on exportera pour pouvoir les utiliser dans les différents fichiers de notre application.

Commençons par créer une méthode nous permettant d'initialiser le prénom de l'utilisateur (donc, notre clé "**@profile**"), dans le stockage.

Créez la méthode "**initFirstname**", qui doit être **exportée** de manière classique et soit être **asynchrone**.

Cette méthode prendra un paramètre "**firstname**", qui aura une chaîne de caractères vide comme valeur par défaut.

Ajoutez un bloc **try/catch** pour la gestion des erreurs lors du développement, et dans le bloc **try** :

- Stoppez l'exécution de la fonction, en la retournant, si "**firstname.length <= 0**".
- Créez une constante "**jsonValue**", qui aura pour valeur "**await getData("@profile")**". Si la clé "**@profile**" n'existe pas, "**jsonValue**" aura une valeur "**undefined**".
- Si "**jsonValue**" est "**undefined**" :
  - Attendez le retour de la méthode "**setData("@profile", firstname)**", pour initialiser la clé "**@profile**" avec la valeur de "**firstname**".



- Retournez **“firstname”**.
- Retournez **“verifyJsonValue(jsonValue)”**. Ce retour ne sera accessible que si la clé **“@profile”** est existante.

```
export async function initFirstname(firstname = ""){  
  
  try{  
  
    if(firstname.length <= 0){  
  
      return;  
  
    }  
  
    const jsonValue = await getData("@profile");  
  
    if(!jsonValue){  
  
      await setData("@profile", firstname);  
  
      return firstname;  
  
    }  
  
    return verifyJsonValue(jsonValue);  
  
  }catch(err){  
  
    console.table(err);  
  
  }  
  
}
```

Créez la méthode “**modifyFirstname**”, qui devra être **exportée** de manière classique et être **asynchrone**.

Elle prendra un paramètre “**firstname**”, qui aura une chaîne de caractères vide comme valeur par défaut.

Ajoutez votre bloc **try/catch**.

**Stoppez** l’exécution de la fonction, en la **retournant**, si “**firstname.length <= 0**”.

Sinon, attendez le retour de la méthode “**setData(“@profile”, firstname)**”, puis retournez “**firstname**”.

Cette méthode nous permettra de modifier la clé “**@profile**”.

```
export async function modifyFirstname(firstname = ""){

  try{

    if(firstname.length <= 0){

      return;

    }

    await setData("@profile", firstname);

    return firstname;

  }catch(err){

    console.table(err);

  }

}
```

Créez la méthode “**getSavedFirstname**”, qui devra être **exportée** de manière classique et être **asynchrone**.

Ajoutez votre bloc **try/catch**.

Créez une constante “**jsonValue**”, qui stockera la valeur “**await getData("@profile")**”.

Retournez “**verifyJsonValue(jsonValue)**”.

Cette méthode nous permettra de récupérer la valeur de la clé “**@profile**”.

```
export async function getSavedFirstname(){  
  
  try{  
  
    const jsonValue = await getData("@profile");  
  
    return verifyJsonValue(jsonValue);  
  
  }catch(err){  
  
    console.table(err);  
  
  }  
  
}
```

Comme nous travaillons dans le fichier “**AsyncStorage**”, nous allons prévoir l’intégralité des méthodes en lien avec le stockage nécessaire à notre application.

L’application finale permettra à l’utilisateur :

- D’ajouter son prénom pour identifier son suivi de données dans l’application.
- Calculer son IMC (indice de masse corporelle).
- Consulter l’historique de ses calculs d’IMC (fonctionnalité de suivi, utilisant le prénom stocké lors du calcul, affichant les données calculées et affichant la date ou le calcul a été effectué).

Nous aurons donc besoin d’interagir avec le stockage local pour :

- Stocker le prénom de l’utilisateur, sur la clé “**@profile**”, et lui permettre de le modifier (par exemple, pour une famille qui souhaiterait conserver son historique sur le même appareil).
- Stocker les données d’un calcul d’IMC, sur la clé “**@history**”, dans un objet servant d’**historique**, qui contiendra le suivi des calculs d’IMC.

Concernant l’historique des calculs, nous allons agrémenter les données avec des informations complémentaires, pour afficher un résultat plus clair à l’utilisateur (le calcul d’IMC nous permettra uniquement de récupérer un score).

Voici un aperçu des données à stocker dans l’historique.

```
firstname, date (format : `${day}/${month}/${year}`), timestamp (format:
Date.parse(date)), imc, imcHint, size, sizeUnit (en cm), weight, weightUnit (en
kg)
```

Nous allons donc commencer par créer la méthode la plus simple, pour la gestion de l'historique, qui est la méthode de récupération de données.

Créez la méthode “**getSavedHistory**”, qui devra être **exportée** de manière classique et être **asynchrone**.

Ajoutez votre bloc **try/catch**.

Créez une constante “**jsonValue**”, qui stockera la valeur “**await getData("@history")**”.

Retournez “**jsonValue**”.

Cette méthode nous permettra de récupérer les données stockées sur la clé “**@history**”.

```
export async function getSavedHistory(){  
  
  try{  
  
    const jsonValue = await getData("@history");  
  
    return jsonValue;  
  
  }catch(err){  
  
    console.table(err);  
  
  }  
  
}
```

Créez la méthode “**saveImcResultToHistory**”, qui devra être **exportée** de manière classique et être **asynchrone**.

Cette méthode prendra les paramètres suivants :

- **size**
- **weight**
- **imc**
- **imcHint**
- **sizeUnit = “cm”**
- **weightUnit = “kg”**

Le paramètre **imcHint** a pour rôle d’accueillir une chaîne de caractère explicative représentant le résultat du score d’IMC.

Par exemple : “ Maigreur, consultez un nutritionniste ”.

```
export async function saveImcResultToHistory(size, weight, imc, imcHint, sizeUnit = "cm", weightUnit = "kg"){};
```

Ajoutez votre bloc **try/catch**.

```
export async function saveImcResultToHistory(size, weight, imc, imcHint, sizeUnit = "cm", weightUnit = "kg"){
  try{
    //Logique de la méthode
  }catch(err){
    console.table(err);
  }
}
```

Dans le bloc **try** :

- Créez une constante “**firstname**”, contenant la valeur “**await getSavedFirstname()**”.

```
const firstname = await getSavedFirstname();
```

- Créez une constante “**actualDate**”, contenant la valeur “**new Date()**”.

```
const actualDate = new Date();
```

- Créez une variable “**day**”, contenant la valeur “**actualDate.getDate()**”.

```
let day = actualDate.getDate();
```

- Modifiez conditionnellement la valeur de la variable “**day**” :
  - Si “**day < 10**”, stockez la valeur “**`0\${day}`**”.
  - Sinon, stockez la valeur “**day**”.

```
day = day < 10 ? `0${day}` : day;
```

- Créez une variable “**month**”, contenant la valeur “**actualDate.getMonth() + 1**” (car les mois commencent à 0, pour Javascript).

```
let month = actualDate.getMonth() + 1;
```

- Modifiez conditionnellement la valeur de “**month**” :
  - Si “**month < 10**”, stockez la valeur “**`0\${month}`**”.
  - Sinon, stockez la valeur “**month**”.

```
month = month < 10 ? `0${month}` : month;
```

- Créez une variable “**year**”, contenant la valeur “**actualDate.getFullYear()**”.

```
let year = actualDate.getFullYear();
```

- Créez une constante “**historyResult**”, qui contiendra un **objet** (représentant une valeur de l'historique), avec les paramètres suivants :
  - **firstname** ayant pour valeur le paramètre **firstname**.
  - **date** ayant pour valeur la date formatée en date française.
  - **timestamp** ayant pour valeur le timestamp basé sur **actualDate** (utilisez la méthode “**Date.parse(actualDate)**”).
  - **imc** ayant pour valeur le paramètre **imc**, en s’assurant que **imc** retourne bien une **valeur numérique** et **1 décimale** (utilisez “**Number()**” et “**.toFixed(1)**”).
  - **imcHint** ayant pour valeur le paramètre **imcHint**.
  - **size** ayant pour valeur le paramètre **size**, en s’assurant que la valeur est bien numérique.
  - **sizeUnit** ayant pour valeur le paramètre **sizeUnit**.
  - **weight** ayant pour valeur le paramètre **weight**, en s’assurant que la valeur est bien numérique.
  - **weightUnit** ayant pour valeur le paramètre **weightUnit**.

```
const historyResult = {  
  firstname: firstname,  
  date: `${day}/${month}/${year}`,  
  timestamp: Date.parse(actualDate),  
  imc: Number(imc.toFixed(1)),  
  imcHint: imcHint,  
  size: Number(size),  
  sizeUnit: sizeUnit,  
  weight: Number(weight),  
}
```



```
weightUnit: weightUnit,  
  
};
```

- Créez une variable “**history**”, qui contiendra un **tableau vide**.

```
let history = [];
```

- Créez une variable “**oldHistory**”, qui aura pour valeur “**await getData("@history")**”.

```
const oldHistory = await getData("@history");
```

- Si “**oldHistory**” contient une valeur :
  - Fixez la valeur de “**history**” à “[ ...oldhistory, historyResult ]”.
- Sinon :
  - Fixez la valeur de “**history**” à “[ historyResult ]”.

```
if(oldHistory){  
  
    history = [  
  
        ...oldHistory,  
  
        historyResult  
  
    ];  
}else{  
  
    history = [  
  
        historyResult  
  
    ];  
  
}
```

- Attendez le retour de la méthode “setData(“@history”, history)”.

```
await setData("@history", history);
```

Cette méthode étant assez lourde, nous allons la décomposer en 4 étapes :

1. Initialisation des données complémentaires (non fournies en paramètres).
2. Création d'une valeur de l'historique et formatage des données.
3. Définition des données de l'historique.
4. Stockage des nouvelles données dans l'historique.

```
export async function saveImcResultToHistory(size, weight, imc, imcHint, sizeUnit  
= "cm", weightUnit = "kg"){  
  
  try{  
  
    // Étape 1 - Initialisation des données complémentaires  
  
    const firstname = await getSavedFirstname();  
  
    const actualDate = new Date();  
  
    let day = actualDate.getDate();  
  
    day = day < 10 ? `0${day}` : day;  
  
    let month = actualDate.getMonth() + 1;  
  
    month = month < 10 ? `0${month}` : month;  
  
    let year = actualDate.getFullYear();  
  
    // Étape 2 - Création d'une valeur de l'historique et formatage des données  
  
    const historyResult = {
```

```

    firstname: firstname,

    date: `${day}/${month}/${year}`,

    timestamp: Date.parse(actualDate),

    imc: Number(imc.toFixed(1)),

    imcHint: imcHint,

    size: Number(size),

    sizeUnit: sizeUnit,

    weight: Number(weight),

    weightUnit: weightUnit,

  };

  // Étape 3 - Définition des données de l'historique

  let history = [];

  const oldHistory = await getData("@history");

  if(oldHistory){

    history = [

      ...oldHistory,

      historyResult

    ];

  }else{

    history = [

      historyResult

```

```
    ];  
  
  }  
  
  // Étape 4 - Stockage des nouvelles données de l'historique  
  
  await setData("@history", history);  
  
}catch(err){  
  
  console.table(err);  
  
}  
  
}
```

Nous avons désormais terminé la création du fichier “**AsyncFunctions.js**”.

Ce fichier est en version définitive et nous n'aurons plus de modifications à lui apporter pour ce projet fil rouge.

Nous allons désormais avoir besoin de créer un **contexte** pour notre application.

## Création d'un contexte et de son provider / Apprendre à centraliser des données

Dans le dossier “shared > contexts”, créez un nouveau fichier nommé “ImcCalculatorProvider.jsx”.

*Pourquoi “ImcCalculatorProvider” au lieu de “ImcCalculatorContext” ?*

*Avec React, nous pouvons créer des **contextes**, pour donner un contexte à une partie de notre application.*

*Ce contexte nous permettra de centraliser des données, pour toujours travailler sur des données synchronisées au sein de l'application (ou certaines sections).*

*Hors, ce contexte doit être créé mais ce n'est pas lui qui fournit directement les données à notre application.*

*Pour diffuser ses données, il aura besoin d'être accompagné d'un **Provider**.*

*Le provider est le **fournisseur de données** du contexte et dispose d'une logique spécifique permettant la fluctuation des données et l'héritage des données centralisées.*

Importez “createContext” et “useContext”, depuis “react”.

```
import {createContext, useContext} from "react";
```

Sous les imports, créez une constante “ImcCalculatorContext” et stockez y la valeur “createContext()”.

```
const ImcCalculatorContext = createContext();
```

Sous cette constante, créez la méthode “**useImcCalculatorContext**”, qui ne prendra pas de paramètres et devra être **exportée** de manière classique.

Cette méthode retournera une **exécution** de la méthode “**useContext(ImcCalculatorContext)**”.

```
export const useImcCalculatorContext = () => useContext(ImcCalculatorContext);
```

C’est cette méthode “**useImcCalculatorContext**” que l’on fournira aux composants qui souhaitent utiliser des données centralisées dans ce contexte.

Sous cette constante, créez le composant “**ImcCalculatorProvider**”, qui devra être **exportée par défaut** et qui prendra une propriété “**children**”.

Cette méthode retournera un composant `<ImcCalculatorContext.Provider></ImcCalculatorContext.Provider>`, qui prendra une propriété “**value**” (on lui donnera par défaut une interpolation d’objet, qui contiendront les variables/méthodes/états à fournir aux composants enfant).

Ce composant englobera une interpolation de la propriété “**children**”.

```
export default function ImcCalculatorProvider({children}){  
  
  return(  
  
    <ImcCalculatorContext.Provider  
  
      value={{}}  
  
    >
```

```

    {children}

    </ImcCalculatorContext.Provider>

  );
}

```

Dans la logique du composant “ImcCalculatorProvider” :

- Créez un état “**firstname**” et son mutateur “**setFirstname**”, en utilisant `useState(“”)`.

```
const [firstname, setFirstname] = useState(“”);
```

- Créez un état “**history**” et son mutateur “**setHistory**”, puis initialisez l’état sous la forme d’un **tableau vide**.

```
const [history, setHistory] = useState([]);
```

- Créez la méthode “**findFirstname**”, qui devra être **asynchrone** et ne prendra pas de paramètres.
  - Créez une constante “**result**”, qui stockera le retour (de manière asynchrone) de la méthode “**getSavedFirstname()**”.
  - Si “**result**” contient une valeur, **mutez** l’état de “**firstname**” en lui définissant la valeur de “**result**”.

```

const findFirstname = async () => {

  const result = await getSavedFirstname();

  if(result){

    setFirstname(result);

  }

}

```

- Créez la méthode “**verifyIfProfileExists**”, qui devra être **asynchrone** et ne prendra pas de paramètres.
  - Appelez de manière asynchrone la méthode “**findFirstname()**”.
  - Si le state (l’état) “**firstname**” contient une valeur, utilisez la méthode “**push()**” de “**router**” (fourni par “**expo-router**”) pour rediriger vers la route “**dashboard**” | **router.push(“dashboard”)**.

```
const verifyIfProfileExists = async () => {  
  
  await findFirstname();  
  
  if(firstname){  
  
    // Pensez à ajouter l'import suivant : import {router} from "expo-router"  
  
    router.push("dashboard");  
  
  }  
  
}
```

- Créez la méthode “**findHistory**”, qui devra être **asynchrone** et ne prendra pas de paramètres.
  - Créez une constante “**result**”, qui stockera le retour asynchrone de la méthode “**getSavedHistory()**”.
  - Si “**result**” contient une valeur, **mutez** le state “**history**” en lui fournissant la valeur de “**result**”.

```
const findHistory = async () => {  
  
  const result = await getSavedHistory();  
  
  if(result){  
  
    setHistory(result);  
  
  }  
  
}
```



- Pour initialiser les données des états **“firstname”** et **“history”** dès le premier chargement du **contexte**, nous allons utiliser la méthode **“useEffect”** de React.
  - Déclenchez les méthodes **“findFirstname()”** et **“findHistory()”**, dans la logique de la méthode **“useEffect()”**.

```
useEffect(() => {  
  
  findFirstname();  
  
  findHistory();  
  
}, []);
```

- Ajoutez dans la propriété **“value”** du composant **ImcCalculatorContext.Provider** :
  - **firstname**
  - **setFirstname**
  - **history**
  - **setHistory**
  - **findFirstname**
  - **findHistory**
  - **verifyIfProfileExists**

```
return(  
  
  <ImcCalculatorContext.Provider  
  
    value={{  
  
      firstname,  
  
      setFirstname,
```

```

        history,

        setHistory,

        findFirstname,

        findHistory,

        verifyIfProfileExists,
    }}

    >

    {children}

</ImcCalculatorContext.Provider>

);

```

C'est de cette manière que le **Provider** transmet les données aux composants enfants qui utilisent un appel à la méthode “**useImcCalculatorContext**”.

La seule condition pour récupérer ces données, c'est que le composant enfant soit englobé par le composant **ImcCalculatorProvider**.

**Vous avez du mal à comprendre comment ?**

Nous allons voir ça rapidement, en intégrant notre contexte à l'application et en terminant la création de notre page d'accueil, qui utilisera les données “**firstname**”, “**findFirstname**” et “**verifyIfProfileExists**” pour son affichage conditionnel (et permettre le stockage du prénom de l'utilisateur lors du premier lancement).

## Intégrer le contexte dans l'application

Rendez-vous dans le fichier “**app** > **\_layout.tsx**” et importez “**ImcCalculatorProvider**” depuis “**@/shared/contexts/ImcCalculatorProvider**”.

```
import ImcCalculatorProvider from "@shared/contexts/ImcCalculatorProvider";
```

Dans le retour du composant **RootLayout**, englobez le composant **Stack** avec le composant **ImcCalculatorProvider**.

```
<ImcCalculatorProvider>

  <Stack>

    <Stack.Screen

      name="(global)"

      options={{

        title: "Groupe global",

        headerShown: false,

      }}

    />

  </Stack>

</ImcCalculatorProvider>
```

Ce fichier “**\_layout.jsx**” étant la structure de notre dossier “**app**” et étant le fichier le plus haut dans l’arborescence de l’application, toute notre application pourra faire appel au contexte “**ImcCalculatorContext**” et récupérer les données spécifiées dans la propriété “**value**” du **Provider**.

## Utiliser le contexte dans l'application

Rendez-vous dans le fichier “**app > (global) > index.tsx**” et importez “**ImcCalculatorProvider**” depuis “**@/shared/contexts/ImcCalculatorProvider**”.

```
import { useImcCalculatorContext } from
"@/shared/contexts/ImcCalculatorProvider";
```

Dans la logique du composant **HomeScreen** :

- Décomposez les données de “**useImcCalculatorContext()**” dans les variables “**firstname**”, “**findFirstname**” et “**verifyIfProfileExists**”.

```
const {
  firstname,
  findFirstname,
  verifyIfProfileExists,
} = useImcCalculatorContext();
```

- Créez un état “**loading**” et son mutateur “**setLoading**”, qui auront la valeur par défaut “**true**”.

```
const [loading, setLoading] = useState(true);
```

- Utilisez “**useEffect**” (fourni par “**react**”) :
  - Exécutez à l'intérieur la méthode “**verifyIfProfileExists()**”.
  - Si l'état “**firstname**” ne contient pas de valeur, mutez l'état de “**loading**” et définissez sa valeur à “**false**”.
  - Ajoutez “**firstname**” au tableau de dépendance de “**useEffect**”.

```
useEffect(() => {

    verifyIfProfileExists();

    if (!firstname) {

        setLoading(false);

    }

}, [firstname]);
```

- Dans la méthode “**handleConfirmFirstname()**”, ajoutez la logique suivante :
  - Exécutez la méthode “**initFirstname(name)**”, de manière asynchrone.
  - **Mutez** l’état de “**newFirstname**” et donnez lui une **chaîne de caractères vide** comme valeur.
  - Exécutez de manière asynchrone la méthode “**findFirstname()**”.
  - Utilisez la méthode “**router.push(“dashboard”)**”.

```
await initFirstname(name);

setNewFirstname("");

await findFirstname();

router.push("dashboard");
```

Avec cette nouvelle logique, nous pourrions modifier l’affichage du composant **HomeScreen**, en nous basant sur l’état “**loading**”.

Si l’état “**loading**” est égal à “**true**”, on retournera un composant **View**, qui englobera le composant **ActivityIndicator** (fourni par “**react-native**”) pour signaler à l’utilisateur qu’un traitement est en cours.

## Dans le retour du composant HomeScreen :

- Englobez le composant **View** actuel avec un **React Fragment** (balise vide : `<></>`).

```
<>

<View style={styles.container}>

//Contenu de votre composant actuel

</View>

</>
```

- Englobez le composant **View** actuel dans une interpolation conditionnelle vérifiant la valeur de l'état **"loading"**.
  - Si la valeur **"loading"** est égale à **"true"** :
    - Retournez un composant **View**, qui aura une propriété **style** ayant pour valeur **"{styles.container}"**.
    - Englobez un composant **ActivityIndicator**, qui aura une propriété **"size"** ayant pour valeur **"large"** et une propriété **"color"** ayant pour valeur **"#E59560"**.

```
(<View style={styles.container}>

  <ActivityIndicator

    size="large"

    color="#E59560"

  />

</View>)
```

- Sinon, retournez votre composant **View** actuel et son contenu.

```
(  
  
  <View style={styles.container}>  
  
    <WelcomeText  
  
      props={{  
  
        text: "Bonjour.",  
  
      }}  
  
    />  
  
    <WelcomeText  
  
      props={{  
  
        text: "Quel est votre prénom ?",  
  
      }}  
  
    />  
  
    <TextInput  
  
      style={styles.firstnameInput}  
  
      placeholder="Votre prénom"  
  
      placeholderTextColor="#BACEC1"  
  
      value={newFirstname}  
  
      onChangeText={setNewFirstname}  
  
    />  
  
    <CtaButton
```

```

      props={{
        text: "Commencer",
        actionOnPress: () => handleConfirmFirstname(newFirstname),
        disabled: newFirstname.length <= 0,
      }}
    />
  </View>
)

```

Notre page d'accueil est désormais terminée.

Voici le code complet du fichier “**app > (global) > index.tsx**” :

```

import { router } from "expo-router";

import { initFirstname } from "@/shared/AsyncFunctions";

import {
  ActivityIndicator,
  StyleSheet,
  TextInput,
  View,
} from "react-native";

import {
  WelcomeText,

```



```

    CtaButton,
  } from "@components/globals";

import { useEffect, useState } from "react";

import { useFonts } from "expo-font";

import { useImcCalculatorContext } from
"@shared/contexts/ImcCalculatorProvider";

export default function HomeScreen() {

  const {

    firstname,

    findFirstname,

    verifyIfProfileExists,

  } = useImcCalculatorContext();

  const [loading, setLoading] = useState(true);

  const [newFirstname, setNewFirstname] = useState("");

  const [loaded, error] = useFonts({

    "Titillium Web Regular": require("@assets/fonts/TitilliumWeb-Regular.ttf"),

  });

  useEffect(() => {

    verifyIfProfileExists();

    if (!firstname) {

      setLoading(false);
    }
  });

```

```

    }

    }, [firstname]);

    const handleConfirmFirstname = async (name = "") => {

        if (name.length <= 0) {

            return;

        }

        name = name.trim();

        name = name.replace(" ", "");

        name = name.toLowerCase();

        name =

            name.slice(0, 1).toUpperCase() +

            name.slice(1, name.length);

        await initFirstname(name);

        setNewFirstname("");

        await findFirstname();

        router.push("dashboard");

    };

    return (

        <>

        {loading ? (

            <View style={styles.container}>

```

```

        <ActivityIndicator
            size="large"
            color="#E59560"
        />
    </View>
) : (
    <View style={styles.container}>
        <WelcomeText
            props={{
                text: "Bonjour.",
            }}
        />
        <WelcomeText
            props={{
                text: "Quel est votre prénom ?",
            }}
        />
        <TextInput
            style={styles.firstnameInput}
            placeholder="Votre prénom"
            placeholderTextColor="#BACEC1"

```

```

        value={newFirstname}

        onChangeText={setNewFirstname}

    />

    <CtaButton

        props={{

            text: "Commencer",

            actionOnPress: () => handleConfirmFirstname(newFirstname),

            disabled: newFirstname.length <= 0,

        }}

    />

</View>

    )}

</>

);
}

const styles = StyleSheet.create({

    container: {

        flex: 1,

        justifyContent: "center",

        alignItems: "center",

        backgroundColor: "#F6F4E8",

```

```
},  
  
firstnameInput: {  
  
    width: 240,  
  
    height: 40,  
  
    borderRadius: 8,  
  
    backgroundColor: "#ffffff",  
  
    textAlign: "center",  
  
    paddingHorizontal: 16,  
  
    paddingVertical: 12,  
  
    borderWidth: 1,  
  
    borderColor: "#BACEC1",  
  
    fontFamily: "Titillium Web Regular",  
  
},  
  
});
```

## Préparer la navigation vers un deuxième écran

Rendez-vous dans le fichier “**app > (global) > \_layout.tsx**”.

Dans le retour du composant **GlobalLayout**, dupliquez votre composant **Tabs.Screen** pour ajouter un deuxième onglet de navigation dans votre application.

Modifiez les propriétés suivantes à l’intérieur de ce nouvel onglet :

- **name:** “dashboard/index”.
- Dans la propriété **options** :
  - **title:** “Dashboard”.
  - **tabBarLabel:** () => (<Text>Dashboard</Text>)
  - **tabBarIcon :** () => (<TabBarIcon name=“stats-chat-outline” size={14}/>)
  - **href=null**

```
<Tabs.Screen
  name="dashboard/index"
  options={{
    title: "Dashboard",
    headerTitleStyle: {
      fontFamily:"Titillium Web Bold"
    },
    tabBarLabel: () => (<Text>Dashboard</Text>),
    tabBarIcon: () => (<TabBarIcon name="stats-chart-outline" size={14}/>),
    tabBarItemStyle: [styles.tabBarItemCustom, {
```

```
        backgroundColor: "#fff"

    }],

    tabBarIconStyle: {

        ...styles.tabBarIconCustom

    },

    tabBarLabelStyle: {

        fontFamily: "Titillium Web Light"

    },

    href: null

  }}

/>
```

L'option **"name"** doit représenter la cible à charger, ici **"dashboard/index"**, car nous allons créer notre deuxième écran dans un dossier dédié et lui fournir un fichier **"index.tsx"**.

L'option **"href"**, ayant pour valeur **"null"** permet de **masquer l'onglet** dans la **tabBar**.

Nous souhaitons uniquement utiliser la route vers **"dashboard"** comme redirection automatique pour présenter à l'utilisateur la liste des fonctionnalités disponibles dans l'application (qui seront également disponibles dans la **tabBar**, pour la navigation rapide).

## Création du deuxième écran - DashboardScreen

Dans le dossier “(global)”, créez un nouveau dossier nommé “dashboard”, dans ce dossier, créez un nouveau fichier nommé “index.tsx”

Ajoutez les imports suivants :

```
import { CtaButton } from "@components/globals";  
  
import { router } from "expo-router";  
  
import { StyleSheet, View } from "react-native";
```

Créez la feuille de style du composant et ajoutez lui les instructions suivantes :

```
const styles = StyleSheet.create({  
  
  view: {  
  
    flex: 1,  
  
    justifyContent: "center",  
  
    alignItems: "center",  
  
    backgroundColor: "#F6F4E8",  
  
  },  
  
  menuButtonsContainer:{  
  
    width:"auto",  
  
    gap: 42,  
  
  },  
  
});
```



```

menuButtonText: {
  display:"flex",
  flexDirection:"row",
  justifyContent: "space-between",
  alignItems:"center",
  gap:12,
  fontSize: 20,
}
});

```

Créez le composant “**DashboardScreen**”, qui devra être **exporté par défaut** et ne prendra pas de propriétés.

Ce composant retournera :

- Un composant **View**, ayant une propriété “**style={styles.view}**”.
- Un composant **View**, ayant une propriété “**style={styles.menuButtonsContainer}**”.
- Un composant **CtaButton**, ayant une propriété “**props**”, qui aura pour valeur un objet contenant les paramètres suivants :
  - **text**: “Calculer mon IMC”
  - **withIcon**: true
  - **iconType**: “edit”
  - **actionOnPress**: () => router.push(“imc-calculator”)
  - **borderRadius**: 16
  - **addTextStyles**: styles.menuButtonText
- Un composant **CtaButton**, ayant une propriété “**props**”, qui aura pour valeur un objet contenant les paramètres suivants :

- text: "Mon suivi"
- withIcon: true
- iconType: "history"
- actionOnPress: () => router.push("history")
- borderRadius: 16
- addTextStyles: styles.menuButtonText
- Un composant **CtaButton**, ayant une propriété "**props**", qui aura pour valeur un objet contenant les paramètres suivants :
  - text: "Mon profil"
  - withIcon: true
  - iconType: "profile"
  - actionOnPress: () => router.push("profile")
  - borderRadius: 16
  - addTextStyles: styles.menuButtonText

```
export default function DashboardScreen() {

  return (

    <View style={styles.view}>

      <View style={styles.menuButtonsContainer}>

        <CtaButton

          props={{

            text: "Calculer mon IMC",

            withIcon: true,

            iconType: "edit",

            actionOnPress: () => router.push("imc-calculator"),

            borderRadius: 16,
```

```

        addTextStyles: styles.menuButtonText
    }}

/>

<CtaButton

    props={{

        text: "Mon suivi",

        withIcon: true,

        iconType: "history",

        actionOnPress: () => router.push("history"),

        borderRadius: 16,

        addTextStyles: styles.menuButtonText
    }}

/>

<CtaButton

    props={{

        text: "Mon profil",

        withIcon: true,

        iconType: "profile",

        actionOnPress: () => router.push("profile"),

        borderRadius: 16,

        addTextStyles: styles.menuButtonText
    }}

```

```

    }}

  />

</View>

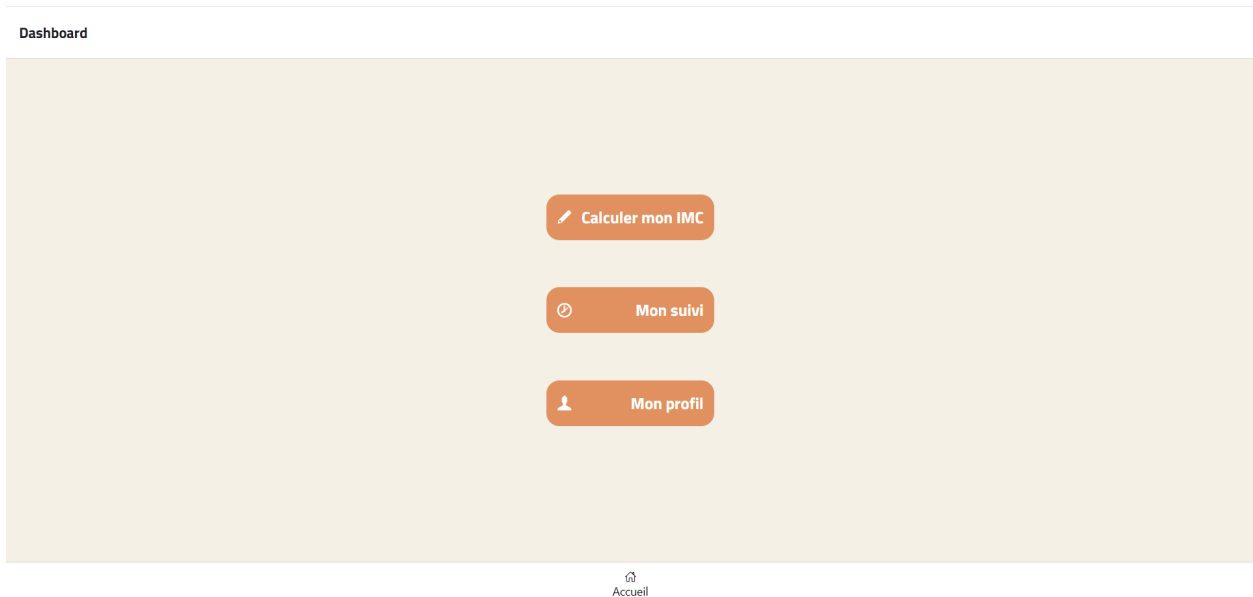
</View>

);
}

```

Si votre application affiche une erreur concernant les routes et que vous souhaitez tester votre application et voir son affichage, vous devrez commenter les propriétés utilisant “**router.push()**”, car les routes “**imc-calculator**”, “**history**” et “**profile**” n’existent pas encore (pas encore de **Tabs.Screen** associés).

Notre deuxième écran est désormais terminé :



Si vous essayez de cibler l’url de la page d’accueil après avoir stocké votre prénom, vous serez désormais redirigés automatiquement vers cette page.

L’onglet “**Accueil**” n’a pas cet effet car il ne recharge pas l’état de “**firstname**” lors de la navigation.

## Préparation de la navigation finale

Dans le fichier “app > (global) > \_layout.tsx”.

Ajoutez un paramètre “href”, de valeur “null” sur l’onglet ciblant la page d’accueil.

```
<Tabs.Screen
  name="index"
  options={{
    title: "Index",
    headerTitleStyle: {
      fontFamily:"Titillium Web Bold"
    },
    tabBarLabel: () => (<Text>
      Accueil
    </Text>),
    tabBarIcon: () => (<TabBarIcon name="home-outline" size={14}/>),
    tabBarItemStyle: {
      ...styles.tabBarItemCustom,
      backgroundColor:"#fff",
    },
    tabBarIconStyle: {
```

```

        ...styles.tabBarIconCustom

    },

    tabBarLabelStyle:{

        fontFamily: "Titillium Web Light"

    },

    href: null,

  }}

/>

```

**Dupliquez** un des onglets de navigation **quatre fois** et retirez l’option “**href**” sur ces quatre nouveaux composants **Tabs.Screen**.

Modifiez les valeurs de chaque onglet, pour obtenir cette navigation :

- Onglet “**Profil**” :
  - **name**: “**profile/index**”
  - Paramètres de la propriété “**options**” :
    - **title**: “**Profil**”
    - **tabBarLabel**: () => (<Text>Profil</Text>)
    - **tabBarIcon**: () => (<TabBarIcon  
name=“person-circle-outline” size={14}/>)
- Onglet “**Mon IMC**” :
  - **name**: “**imc-calculator/index**”
  - Paramètres de la propriété “**options**” :
    - **title**: “**Calculer mon IMC**”
    - **tabBarLabel**: () => (<Text>Mon IMC</Text>)

- `tabBarIcon : () => (<TabBarIcon name="add-circle-outline" size={14}/>)`
- Onglet "Mon suivi" :
  - `name: "history/index"`
  - Paramètres de la propriété "options" :
    - `title: "Mon suivi"`
    - `tabBarLabel: () => (<Text>Mon suivi</Text>)`
    - `tabBarIcon : () => (<TabBarIcon name="time-outline" size={14}/>)`
- Onglet "À propos" :
  - `name: "about/index"`
  - Paramètres de la propriété "options" :
    - `title: "À propos"`
    - `tabBarLabel: () => (<Text>À propos</Text>)`
    - `tabBarIcon : () => (<TabBarIcon name="settings-outline" size={14}/>)`

Le composant **GlobalLayout** est désormais en version finale :

```
import { TabBarIcon } from "@components/navigation/TabBarIcon";

import { Tabs } from "expo-router";

import { StyleSheet, Text } from "react-native";

import { useFonts } from "expo-font";

export default function GlobalLayout() {

  const [loaded, error] = useFonts({

    "Titillium Web Light": require("@assets/fonts/TitilliumWeb-Light.ttf"),
```

```

    "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),
  });

  return (
    <Tabs>

      <Tabs.Screen
        name="index"

        options={{
          title: "Index",

          headerTitleStyle: {
            fontFamily:"Titillium Web Bold"
          },

          tabBarLabel: () => (<Text>

            Accueil

          </Text>),

          tabBarIcon: () => (<TabBarIcon name="home-outline" size={14}/>),

          tabBarItemStyle: {
            ...styles.tabBarItemCustom,

            backgroundColor:"#fff",
          },

          tabBarIconStyle: {
            ...styles.tabBarIconCustom

```



```

    },

    tabBarLabelStyle: {

        fontFamily: "Titillium Web Light"

    },

    href: null,

  }}

/>

<Tabs.Screen

  name="dashboard/index"

  options={{

    title: "Dashboard",

    headerTitleStyle: {

      fontFamily: "Titillium Web Bold"

    },

    tabBarLabel: () => (<Text>Dashboard</Text>),

    tabBarIcon: () => (<TabBarIcon name="stats-chart-outline" size={14}/>),

    tabBarItemStyle: [styles.tabBarItemCustom, {

      backgroundColor: "#fff"

    }],

    tabBarIconStyle: {

      ...styles.tabBarIconCustom

```

```

    },

    tabBarLabelStyle: {

        fontFamily: "Titillium Web Light"

    },

    href: null

  }}

/>

<Tabs.Screen

  name="profile/index"

  options={{

    title: "Profil",

    headerTitleStyle: {

      fontFamily: "Titillium Web Bold"

    },

    tabBarLabel: () => (<Text>Profil</Text>),

    tabBarIcon: () => (<TabBarIcon name="person-circle-outline"
size={14}/>),

    tabBarItemStyle: [styles.tabBarItemCustom, {

      backgroundColor: "#fff"

    }],

    tabBarIconStyle: {

```

```

        ...styles.tabBarIconCustom

    },

    tabBarLabelStyle:{

        fontFamily: "Titillium Web Light"

    },

    }}

/>

<Tabs.Screen

    name="imc-calculator/index"

    options={{

        title: "Calculer mon IMC",

        headerTitleStyle: {

            fontFamily:"Titillium Web Bold"

        },

        tabBarLabel: () => (<Text>Mon IMC</Text>),

        tabBarIcon: () => (<TabBarIcon name="add-circle-outline" size={14}/>),

        tabBarItemStyle: [styles.tabBarItemCustom, {

            backgroundColor:"#fff"

        }],

        tabBarIconStyle: {

            ...styles.tabBarIconCustom

```

```

    },

    tabBarLabelStyle:{

      fontFamily: "Titillium Web Light"

    },

  }}

/>

<Tabs.Screen

  name="history/index"

  options={{

    title: "Mon suivi",

    headerTitleStyle: {

      fontFamily:"Titillium Web Bold"

    },

    tabBarLabel: () => (<Text>Mon suivi</Text>),

    tabBarIcon: () => (<TabBarIcon name="time-outline" size={14}/>),

    tabBarItemStyle: [styles.tabBarItemCustom, {

      backgroundColor:"#fff"

    }],

    tabBarIconStyle: {

      ...styles.tabBarIconCustom

    },

```

```

        tabBarLabelStyle:{

            fontFamily: "Titillium Web Light"

        },

    }}

/>

<Tabs.Screen

    name="about/index"

    options={{

        title: "À propos",

        headerTitleStyle: {

            fontFamily:"Titillium Web Bold"

        },

        tabBarLabel: () => (<Text>À propos</Text>),

        tabBarIcon: () => (<TabBarIcon name="settings-outline" size={14}/>),

        tabBarItemStyle: [styles.tabBarItemCustom, {

            backgroundColor:"#fff"

        }],

        tabBarIconStyle: {

            ...styles.tabBarIconCustom

        },

        tabBarLabelStyle:{

```

```

        fontFamily: "Titillium Web Light"

      },

    }}

  />

</Tabs>

);
}

const styles = StyleSheet.create({

  tabBarItemCustom: {

    flex:1,

    flexDirection: "column",

    justifyContent:"center",

    alignItems:"center",

  },

  tabBarIconCustom: {

    height:18,

  }

});

```

## Préparation des écrans supplémentaires

Pour simplifier la création des prochains écrans, nous allons créer les fichiers contenant les **composants d'écran**, en leur donnant un retour minimal par défaut.

De cette manière, vous pourrez tester votre nouvelle navigation et vérifier si votre application a les réactions attendues.

### Écran Calculer mon IMC :

Créez un nouveau dossier nommé “**imc-calculator**”, dans le dossier “(global)”, et créez un fichier “**index.tsx**” à l'intérieur.

Donnez-lui ce code par défaut :

```
import { StyleSheet, Text, View } from "react-native";

export default function ImcCalculatorScreen(){

  return(

    <View>

      <Text>IMC Calculator Screen</Text>

    </View>

  );
}

const styles = StyleSheet.create({});
```

Ajoutez ces instructions de styles :

```
const styles = StyleSheet.create({

  view: {

    flex: 1,

    backgroundColor: "#F6F4E8",

  },

  scrollView: {

    width: "100%",

  },

  contentView: {

    flex: 1,

    justifyContent: "center",

    alignItems: "center",

    gap: 48,

    padding: 20,

  },

  calculateIMCFormView: {

    justifyContent: "flex-start",

    alignItems: "stretch",

    gap: 24,

    padding: 0,
```



```

    },

    calculateIMCResultView: {

        justifyContent: "center",

        alignItems: "center",

        gap: 12,

    },

    textInputs: {

        width: 240,

        height: 40,

        borderRadius: 8,

        fontSize: 16,

        backgroundColor: "#ffffff",

        textAlign: "left",

        paddingHorizontal: 16,

        paddingVertical: 12,

        borderWidth: 1,

        borderColor: "#BACEC1",

        fontFamily: "Titillium Web Regular",

    },

});

```

## Écran Mon suivi:

Créez un nouveau dossier nommé **“history”**, dans le dossier **“(global)”**, et créez un fichier **“index.tsx”** à l’intérieur.

Donnez-lui ce code par défaut :

```
import { StyleSheet, Text, View } from "react-native";

export default function HistoryScreen(){

  return(

    <View>

      <Text>History Screen</Text>

    </View>

  );

}

const styles = StyleSheet.create({});
```

Donnez lui ce style par défaut :

```
const styles = StyleSheet.create({

  view: {

    flex: 1,

    justifyContent: "flex-start",

    alignItems: "stretch",

    gap: 0,
```

```
    backgroundColor: "#F6F4E8",

    paddingVertical: 42,

    paddingHorizontal: 20,

  },

});
```

### Écran Mon profil:

Créez un nouveau dossier nommé “**profile**”, dans le dossier “(global)”, et créez un fichier “**index.tsx**” à l’intérieur.

Donnez-lui ce code par défaut :

```
import { StyleSheet, Text, View } from "react-native";

export default function ProfileScreen() {

  return (

    <View>

      <Text>Profile Screen</Text>

    </View>

  );

}

const styles = StyleSheet.create({});
```

Donnez lui le style suivant :

```
const styles = StyleSheet.create({

  view: {

    flex: 1,

    justifyContent: "flex-start",

    alignItems: "center",

    backgroundColor: "#F6F4E8",

  },

  scrollView: {

    width:"100%",

    gap: 84,

    paddingVertical: 24,

  },

  userInfoView: {

    width:"auto",

    justifyContent:"center",

    alignItems:"center",

    gap: 6,

    marginBottom: 84,

  },

  modifyUserView: {
```

```
width:"auto",

justifyContent:"center",

alignItems:"center",

gap: 8,

},

avatar: {

width: 128,

height: 128,

borderRadius: 128,

marginHorizontal: "auto",

},

firstname: {

fontSize: 64,

textAlign: "center",

fontFamily: "Titillium Web Bold",

},

firstnameInput: {

width: 240,

height: 40,

borderRadius: 8,

backgroundColor: "#ffffff",
```

```

    textAlign: "center",

    paddingHorizontal: 16,

    paddingVertical: 12,

    borderWidth: 1,

    borderColor: "#BACEC1",

    fontFamily: "Titillium Web Regular",
  },
  cta: {

    marginHorizontal: "auto",

  },
  ctaText: {

    fontSize: 16,

  },
});

```

## Écran À propos:

Créez un nouveau dossier nommé “**profile**”, dans le dossier “(global)”, et créez un fichier “**index.tsx**” à l’intérieur.

Donnez-lui ce code par défaut :

```

import { StyleSheet, Text, View } from "react-native";

export default function AboutScreen() {

```

```
return (  
  <View>  
    <Text>About Screen</Text>  
  </View>  
)  
);  
}  
  
const styles = StyleSheet.create({});
```

Donnez lui ce style par défaut :

```
const styles = StyleSheet.create({  
  container: {  
    flex: 1,  
    backgroundColor: "#F6F4E8",  
  },  
  scrollView: {  
    width: "100%",  
  },  
  contentView: {  
    justifyContent: "flex-start",  
    alignItems: "center",  
    gap: 24,  
  },  
});
```

```
paddingVertical: 36,

},

customFont: {

  fontFamily: "Titillium Web Regular",

}

});
```

Lors de la création de vos applications, pensez à prévoir les écrans nécessaires en avance, pour créer leur fichier d’affichage minimal, afin de fluidifier l’implémentation des vos éléments.

## Création d’un écran - About

Rendez-vous dans le fichier “**app > (global) > about > index.tsx**”.

Commencez par importer “**useFonts**”, depuis “**expo-font**”.

```
import { useFonts } from "expo-font";
```

Initialisez la police “**Titillium Web Regular**”, pour la feuille de styles.

```
const [loaded, error] = useFonts({

  "Titillium Web Regular": require("@/assets/fonts/TitilliumWeb-Regular.ttf"),

});
```



## Retour du composant AboutScreen :

- Ajoutez un composant **SafeAreaView** (fourni par “react-native”) et donnez-lui la propriété “**style={styles.container}**”.

**SafeAreaView** contiendra la vue de l'écran à l'intérieur de la “zone d'affichage sûre” des téléphones (pour éviter de perdre un bout d'affichage avec les bords arrondis ou les composants, comme la caméra).

- Ajoutez un composant **ScrollView** (fourni par “react-native”) et donnez-lui la propriété “**style={styles.scrollView}**”.

**ScrollView**\* nous permet d'afficher une vue scrollable, qui permet de faire défiler le contenu qui déborde de l'écran.

\* Pour afficher un défilement horizontal, vous pouvez lui fournir la propriété “**horizontal**”, à la valeur “**true**”.

- Ajoutez un composant **View** et donnez-lui la propriété “**style={styles.contentView}**”.

**View** contiendra une section d'affichage (qui englobe certains composants) de la page.

Vous pouvez utiliser autant de composants **View** que nécessaire pour structurer votre page.

- Dans ce composant **View**, ajoutez :
  - Un composant “**Card**”, qui prendra une propriété “**props**”, contenant un paramètre “**title**” (titre à implémenter en fonction de l'affichage souhaité). Il pourra englober toutes sortes de composants et gérer leur affichage, mais nous utiliserons principalement des composants **Text** avec pour le contenu de la page **À propos**.  
Ce composant “**Card**” sera importé depuis le chemin “**@/components/globals**” (oui, c'est notre prochain composant personnalisé).

-

```

return (

  <SafeAreaView style={styles.container}>

    <ScrollView style={styles.scrollView}>

      <View style={styles.contentView}>

        <Card

          props={{

            title: "Développeur",

          }}

        >

          <Text style={styles.customFont}>Dufrène Valérien</Text>

        </Card>

        <Card

          props={{

            title: "Entreprise",

          }}

        >

          <Text style={styles.customFont}>Webdevoo Formation</Text>

        </Card>

        <Card

          props={{

            title: "Type de projet",

```

```

    }}

    >

    <Text style={styles.customFont}>Open-source</Text>

  </Card>

</View>

</ScrollView>

</SafeAreaView>

);

```

Votre éditeur de code devrait vous **afficher une erreur** pour le composant **Card** et la **ligne d'importation** de ce composant.

C'est normal, le composant “**Card**” ne fait pas encore partie de notre boîte à composants, nous allons donc le créer.

## Création d'un composant personnalisé - Card

Créez le fichier “**components > globals > Card.jsx**”.

- Créez le composant **Card**, qui devra être **exporté par défaut** et qui prendra une propriété “**props**” et une propriété “**children**”.

```
export default function Card({props, children}){}
```

- Dans la logique du composant, déstructurez “**props**” dans les variables “**title**” et “**borderColor**”.

```
const {title, borderColor} = props;
```

- Initialisez la police **Titillium Web Bold**.

```
const [loaded, error] = useFonts({  
  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),  
});
```

Retour du composant Card :

- Ajoutez un composant **View** et donnez-lui la propriété “**style={{ ...styles.view, borderColor: borderColor ?? “transparent” }}**”.
- Donnez-lui un composant **Text**, en enfant, avec la propriété “**style={styles.title}**”.
  - Englobez l’interpolation de “**title**”, dans le composant **Text**.
- Sous le composant **Text**, interpolez la props “**children**”.

```
return(  
  
  <View style={{  
  
    ...styles.view,  
  
    borderColor: borderColor ?? "transparent"  
  
  }}>  
  
    <Text style={styles.title}>  
  
      {title}
```

```
    </Text>

    {children}

  </View>

);
```

Ajoutez la feuille de style suivante :

```
const styles = StyleSheet.create({

  view:{

    width:350,

    backgroundColor:"#FFFFFF",

    borderWidth:4,

    borderColor:"transparent",

    gap:4,

    borderRadius:12,

    padding: 16,

    marginHorizontal: "auto",

  },

  title: {

    fontSize:24,

    fontWeight:"bold",

    letterSpacing: .15,
```

```

    fontFamily: "Titillium Web Bold",
  }
});

```

**Card** est le dernier composant personnalisé de notre projet fil rouge, mais je vous incite à créer vos propres composants pour agrémenter l'application.

Son code final est :

```

import { StyleSheet, Text, View } from "react-native";

import { useFonts } from "expo-font";

export default function Card({props, children}){

  const {title, borderColor} = props;

  const [loaded, error] = useFonts({

    "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),

  });

  return(

    <View style={{

      ...styles.view,

      borderColor: borderColor ?? "transparent"

    }}>

      <Text style={styles.title}>

        {title}

      </Text>

    </View>

  );
}

```

```

        </Text>

        {children}

    </View>

);
}

const styles = StyleSheet.create({
  view:{
    width:350,

    backgroundColor:"#FFFFFF",

    borderWidth:4,

    borderColor:"transparent",

    gap:4,

    borderRadius:12,

    padding: 16,

    marginHorizontal: "auto",
  },
  title: {
    fontSize:24,

    fontWeight:"bold",

    letterSpacing: .15,

    fontFamily: "Titillium Web Bold",
  },
});

```

```
}  
});
```

Ouvrez désormais le fichier “**components > globals > index.js**” et ajoutez le composant “**Card**” à la liste d’exportation.

```
import WelcomeText from "./WelcomeText";  
  
import CtaButton from "./CtaButton";  
  
import Card from "./Card";  
  
export {  
  
  WelcomeText,  
  
  CtaButton,  
  
  Card,  
  
}
```

## Amélioration de l’écran About

Vous pouvez désormais ajouter les informations que vous souhaitez, en plus de vos informations de contact, sur la page **À propos**.

Dans chaque application que vous créerez, je vous conseille de prévoir un espace de présentation pour donner vos informations, vos coordonnées et donner des détails spécifiques au développement de l’application.



## Création d'un écran - Profile

Rendez-vous dans le fichier “app > (global) > profile > index.tsx”.

Dans la logique du composant ProfileScreen :

Décomposez “useImCalculatorContext()” et récupérez éléments “firstname”, “setFirstname”, “findFirstname”.

```
const {firstname, setFirstname, findFirstname} = useImCalculatorContext();
```

Créez un état “newFirstname” et son mutateur “setNewFirstname”, avec pour valeur par défaut (“”).

```
const [newFirstname, setNewFirstname] = useState("");
```

Initialisez les polices Titillium Web Regular et Titillium Web Bold.

```
const [loaded, error] = useFonts({  
  "Titillium Web Regular": require("@/assets/fonts/TitilliumWeb-Regular.ttf"),  
  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),  
});
```

Créez la méthode “handleModifyFirstname”, qui devra être asynchrone et ne prendra pas de paramètres.

- Si “newFirstname.length <= 0”, stoppez son exécution en la retournant.

```
if (newFirstname.length <= 0) {

    return;

}
```

- Créez une variable “formattedNewFirstName”, qui aura pour valeur “newFirstname.toLowerCase().trim().replaceAll(“ ”, “”)”.
- Modifiez la variable “formattedNewFirstName” et donnez lui la valeur “formattedNewFirstName.slice(0,1).toUpperCase() + formattedNewFirstName.slice(1, formattedNewFirstName.length)”.
- Mutez l’état de “newFirstName” pour lui fournir “formattedNewFirstName” comme valeur.

```
let formattedNewFirstName = newFirstname

    .toLowerCase()

    .trim()

    .replaceAll(" ", "");

formattedNewFirstName = formattedNewFirstName.slice(0, 1).toUpperCase() +
formattedNewFirstName.slice(1, formattedNewFirstName.length);

setNewFirstname(() =>

    formattedNewFirstName

);
```

- Exécutez de manière asynchrone la méthode “modifyFirstname(newFirstname)”.
- Gérez son retour avec la méthode “.then()”, qui prendra un paramètre “data”.
  - Si “data” est “undefined”, “null” ou “false”, effectuez un

**retour** pour stopper l'exécution de la méthode **".then()"**.

- Sinon, **mutez "newFirstname"** et définissez-lui une **chaîne de caractères vide** comme valeur.

```
await modifyFirstname(newFirstname).then((data) => {  
  
    if (!data) {  
  
        return;  
  
    }  
  
    setNewFirstname("");  
  
});  
  
findFirstname();
```

Code complet de la méthode **"handleModifyFirstname"** :

```
const handleModifyFirstname = async () => {  
  
    if (newFirstname.length <= 0) {  
  
        return;  
  
    }  
  
    let formattedNewFirstName = newFirstname  
  
    .toLowerCase()  
  
    .trim()  
  
    .replaceAll(" ", "");  
  
    formattedNewFirstName = formattedNewFirstName.slice(0, 1).toUpperCase() +  
  
    formattedNewFirstName.slice(1, formattedNewFirstName.length);
```

```

setNewFirstname(() =>

    formattedNewFirstName

);

await modifyFirstname(newFirstname).then((data) => {

    if (!data) {

        return;

    }

    setNewFirstname("");

});

findFirstname();

};

```

## Retour du composant ProfileScreen :

- Ajoutez un composant **SafeAreaView**, ayant une propriété “**style={styles.view}**”.
- Ajoutez un composant **ScrollView**, ayant une propriété “**styles.scrollView**”.
- Ajouter un composant **View**, ayant une propriété “**styles.userInfoView**”, qui englobera :
  - Un composant **Image** (fourni par “**react-native**”), qui aura les propriétés suivantes :

- style={styles.avatar}
- source={{
 uri:
 "https://plus.unsplash.com/premium\_photo-1678865310627-129c0980ab6e?q=80&w=1974&auto=format&fit=crop&ixlib=rb-4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D"
 }}
- Un composant **Text**, qui aura la propriété "style={styles.firstname}" et qui englobera :
  - L'interpolation de "firstname".

```
<View style={styles.userInfoView}>

  <Image

    style={styles.avatar}

    source={{

      uri:
      "https://plus.unsplash.com/premium_photo-1678865310627-129c0980ab6e?q=80&w=1974&auto=format&fit=crop&ixlib=rb-4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D",

    }}

  />

  <Text style={styles.firstname}>{firstname}</Text>

</View>
```

- Un composant **View**, ayant la propriété `“style={styles.modifyUserView}”`, qui englobera :
  - Un composant **WelcomeText**, qui aura comme propriété `“props={{`  
     `text: “Un autre utilisateur ?”,`  
     `fontFamilyLight: true,`  
     `}}”`.
- Un composant **TextInput**, qui aura les propriétés suivantes :
  - `style={styles.firstnameInput}`
  - `placeholder=“Nouveau prénom”`
  - `placeholderTextColor=“#BACEC1”`
  - `onChangeText={setNewFirstname}`
  - `value={newFirstname}`
- Un composant **CtaButton**, ayant les propriétés suivantes :
  - `props={{`  
     `text: “Modifier”,`  
     `actionOnPress: () => handleModifyFirstname(),`  
     `disabled: newFirstname.length <= 0,`  
     `addStyles: styles.cta,`  
     `addTextStyles: styles.ctaText`  
     `}}`

```
<View style={styles.modifyUserView}>

  <WelcomeText

    props={{

      text: "Un autre utilisateur ?",

      fontFamilyLight: true,

    }}

  <TextInput

    style={styles.firstnameInput}
    placeholder="Nouveau prénom"
    placeholderTextColor="#BACEC1"
    onChangeText={setNewFirstname}
    value={newFirstname}

  <CtaButton

    props={{
      text: "Modifier",
      actionOnPress: () => handleModifyFirstname(),
      disabled: newFirstname.length <= 0,
      addStyles: styles.cta,
      addTextStyles: styles.ctaText
    }}

  </CtaButton>
</View>
```

```

/>

<TextInput

  style={styles.firstnameInput}

  placeholder="Nouveau prénom"

  placeholderTextColor="#BACEC1"

  onChangeText={setNewFirstname}

  value={newFirstname}

/>

<CtaButton

  props={{

    text: "Modifier",

    actionOnPress: () => handleModifyFirstname(),

    disabled: newFirstname.length <= 0,

    addStyles: styles.cta,

    addTextStyles: styles.ctaText,

  }}

/>

</View>

```

## Code complet du retour du composant ProfileScreen :

```
return (  
  <SafeAreaView style={styles.view}>  
    <ScrollView style={styles.scrollView}>  
      <View style={styles.userInfosView}>  
        <Image  
          style={styles.avatar}  
          source={{  
            uri:  
"https://plus.unsplash.com/premium_photo-1678865310627-129c0980ab6e?q=80&w=1974&auto=format&fit=crop&ixlib=rb-4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D",  
          }}  
        />  
        <Text style={styles.firstname}>{firstname}</Text>  
      </View>  
  
      <View style={styles.modifyUserView}>  
        <WelcomeText  
          props={{  
            text: "Un autre utilisateur ?",  
            fontFamilyLight: true,  

```



```

    }}

  />

  <TextInput

    style={styles.firstnameInput}

    placeholder="Nouveau prénom"

    placeholderTextColor="#BACEC1"

    onChangeText={setNewFirstname}

    value={newFirstname}

  />

  <CtaButton

    props={{

      text: "Modifier",

      actionOnPress: () => handleModifyFirstname(),

      disabled: newFirstname.length <= 0,

      addStyles: styles.cta,

      addTextStyles: styles.ctaText,

    }}

  />

</View>

</ScrollView>

</SafeAreaView>

```

```
);
```

Ajoutez cette feuille de style :

```
const styles = StyleSheet.create({  
  view: {  
    flex: 1,  
    justifyContent: "flex-start",  
    alignItems: "center",  
    backgroundColor: "#F6F4E8",  
  },  
  scrollView: {  
    width: "100%",  
    gap: 84,  
    paddingVertical: 24,  
  },  
  userInfoView: {  
    width: "auto",  
    justifyContent: "center",  
    alignItems: "center",  
    gap: 6,  
    marginBottom: 84,  
  },  
});
```

```
},

modifyUserView: {

    width:"auto",

    justifyContent:"center",

    alignItems:"center",

    gap: 8,

},

avatar: {

    width: 128,

    height: 128,

    borderRadius: 128,

    marginHorizontal: "auto",

},

firstname: {

    fontSize: 64,

    textAlign: "center",

    fontFamily: "Titillium Web Bold",

},

firstnameInput: {

    width: 240,

    height: 40,
```

```
    borderRadius: 8,

    backgroundColor: "#ffffff",

    textAlign: "center",

    paddingHorizontal: 16,

    paddingVertical: 12,

    borderWidth: 1,

    borderColor: "#BACEC1",

    fontFamily: "Titillium Web Regular",
  },

  cta: {

    marginHorizontal: "auto",

  },

  ctaText: {

    fontSize: 16,

  },

});
```

Notre écran Profile est enfin prêt, voici son code complet :

```
import { useState } from "react";

import {
  Image,
  SafeAreaView,
  ScrollView,
  StyleSheet,
  Text,
  TextInput,
  View,
} from "react-native";

import {
  modifyFirstname,
} from "@shared/AsyncFunctions";

import {
  CtaButton,
  WelcomeText,
} from "@components/globals";

import { useFonts } from "expo-font";

import { useImcCalculatorContext } from
"@shared/contexts/ImcCalculatorProvider";
```

```

export default function ProfileScreen() {

  /**
   * TODO : Régler le problème de l'avatar qui ne charge pas sur mobile !
   * Solution: L'image utilisée par défaut était au format SVG, donc incompatible
   avec les appareils mobiles de manière native.
   */

  const {firstname, setFirstname, findFirstname} = useImcCalculatorContext();

  const [newFirstname, setNewFirstname] = useState("");

  const [loaded, error] = useFonts({
    "Titillium Web Regular": require("@/assets/fonts/TitilliumWeb-Regular.ttf"),
    "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),
  });

  const handleModifyFirstname = async () => {

    if (newFirstname.length <= 0) {

      return;

    }

    let formattedNewFirstName = newFirstname

    .toLowerCase()

    .trim()

    .replaceAll(" ", "");

    formattedNewFirstName = formattedNewFirstName.slice(0, 1).toUpperCase() +

```

```

formattedNewFirstName.slice(1, formattedNewFirstName.length);

setNewFirstname(() =>

    formattedNewFirstName

);

await modifyFirstname(newFirstname).then((data) => {

    if (!data) {

        return;

    }

    setNewFirstname("");

});

findFirstname();

};

return (

    <SafeAreaView style={styles.view}>

        <ScrollView style={styles.scrollView}>

            <View style={styles.userInfosView}>

                <Image

                    style={styles.avatar}

                    source={{

                        uri:

```

```

"https://plus.unsplash.com/premium_photo-1678865310627-129c0980ab6e?q=80&w=1974&auto=format&fit=crop&ixlib=rb-4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D",

    }}

  />

  <Text style={styles.firstname}>{firstname}</Text>

</View>

<View style={styles.modifyUserView}>

  <WelcomeText

    props={{

      text: "Un autre utilisateur ?",

      fontFamilyLight: true,

    }}

  />

  <TextInput

    style={styles.firstnameInput}

    placeholder="Nouveau prénom"

    placeholderTextColor="#BACEC1"

    onChangeText={setNewFirstname}

    value={newFirstname}

  />

  <CtaButton

```



```

        props={{
            text: "Modifier",
            actionOnPress: () => handleModifyFirstname(),
            disabled: newFirstname.length <= 0,
            addStyles: styles.cta,
            addTextStyles: styles.ctaText,
        }}

    />

</View>

</ScrollView>

</SafeAreaView>

);
}

const styles = StyleSheet.create({
    view: {
        flex: 1,
        justifyContent: "flex-start",
        alignItems: "center",
        backgroundColor: "#F6F4E8",
    },
    scrollView: {

```

```
width:"100%",

gap: 84,

paddingVertical: 24,
},
userInfoView: {

width:"auto",

justifyContent:"center",

alignItems:"center",

gap: 6,

marginBottom: 84,
},
modifyUserView: {

width:"auto",

justifyContent:"center",

alignItems:"center",

gap: 8,
},
avatar: {

width: 128,

height: 128,

borderRadius: 128,
```

```
    marginHorizontal: "auto",
  },
  firstname: {
    fontSize: 64,
    textAlign: "center",
    fontFamily: "Titillium Web Bold",
  },
  firstnameInput: {
    width: 240,
    height: 40,
    borderRadius: 8,
    backgroundColor: "#ffffff",
    textAlign: "center",
    paddingHorizontal: 16,
    paddingVertical: 12,
    borderWidth: 1,
    borderColor: "#BACEC1",
    fontFamily: "Titillium Web Regular",
  },
  cta: {
    marginHorizontal: "auto",
```

```

    },
    ctaText: {
      fontSize: 16,
    },
  },
});

```

## Création d'un écran - ImcCalculator

Rendez-vous dans le fichier “**app > (global) > imc-calculator > index.tsx**”.

Ajoutez cette feuille de style au composant **ImcCalculatorScreen** :

```

const styles = StyleSheet.create({
  view: {
    flex: 1,
    backgroundColor: "#F6F4E8",
  },
  scrollView: {
    width: "100%",
  },
  contentView: {

```

```
    flex: 1,

    justifyContent: "center",

    alignItems: "center",

    gap: 48,

    padding: 20,
  },

  calculateIMCFormView: {

    justifyContent: "flex-start",

    alignItems: "stretch",

    gap: 24,

    padding: 0,
  },

  calculateIMCResultView: {

    justifyContent: "center",

    alignItems: "center",

    gap: 12,
  },

  textInputs: {

    width: 240,

    height: 40,

    borderRadius: 8,
```

```

    fontSize: 16,

    backgroundColor: "#ffffff",

    textAlign: "left",

    paddingHorizontal: 16,

    paddingVertical: 12,

    borderWidth: 1,

    borderColor: "#BACEC1",

    fontFamily: "Titillium Web Regular",

  },
});

```

### Logique du composant ImcCalculatorScreen :

- Décomposez la méthode “`useImcCalculatorContext()`” pour récupérer la méthode “`findHistory`”.

```
const {findHistory} = useImcCalculatorContext();
```

- Créez les états suivants :
  - “`size`” et son mutateur “`setSize`”, ayant pour valeur initiale une **chaîne de caractères vide**.
  - “`weight`” et son mutateur “`setWeight`”, ayant pour valeur initiale une **chaîne de caractères vide**.
  - “`imc`” et son mutateur “`setImc`”, ayant pour valeur initiale de **0**.
  - “`result`” et son mutateur “`setResult`”, ayant pour valeur initiale une **chaîne de caractères vide**.

```
const [size, setSize] = useState("");

const [weight, setWeight] = useState("");

const [imc, setImc] = useState(0);

const [result, setResult] = useState("");
```

- Initialisez les polices “Titillium Web Light”, “Titillium Web Regular” et “Titillium Web Bold”.

```
const [loaded, error] = useFonts({

  "Titillium Web Light": require("@/assets/fonts/TitilliumWeb-Light.ttf"),

  "Titillium Web Regular": require("@/assets/fonts/TitilliumWeb-Regular.ttf"),

  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),

});
```

- Créez la méthode “**determineResultHint**”, qui devra être **asynchrone** et prendra le paramètre “**imc: number**” (paramètre **imc**, de type **number**).
  - Créez une variable “**imcHint**”, ayant pour valeur par défaut une **chaîne de caractères vide**.
  - Ajoutez un **switch()**, prenant la valeur “**true**” en paramètre, et créez les cas suivants :
    - Si “(imc <= 18.5)” :

```
case (imc <= 18.5):

  imcHint = `Maigreur, consultez un nutritionniste`;

  setResult(imcHint);

  return imcHint;
```

- Si "(imc <= 25 && imc > 18.5)" :

```
case (imc <= 25 && imc > 18.5):  
  
    imcHint = `Poids normal`;  
  
    setResult(imcHint);  
  
    return imcHint;
```

- Si "(imc <= 30 && imc > 25)" :

```
case (imc <= 30 && imc > 25):  
  
    imcHint = `Surpoids`;  
  
    setResult(imcHint);  
  
    return imcHint;
```

- Si "(imc <= 35 && imc > 30)" :

```
case (imc <= 35 && imc > 30):  
  
    imcHint = `Obésité modérée`;  
  
    setResult(imcHint);  
  
    return imcHint;
```

- Si "(imc <= 40 && imc > 35)" :

```
case (imc <= 40 && imc > 35):  
  
    imcHint = `Obésité sévère, consultez un  
nutritionniste`;  
  
    setResult(imcHint);
```



```
return imcHint;
```

- Si “(imc > 40)” :

```
case (imc > 40):  
  
    imcHint = `Obésité morbide, consultez un  
nutritionniste rapidement`;  
  
    setResult(imcHint);  
  
    return imcHint;
```

- Sinon, on gère le cas par défaut :

```
default:  
  
    imcHint = `Les valeurs fournies sont incorrectes`;  
  
    setResult(imcHint);  
  
    return imcHint;
```

- Créez la méthode “handleCalculateIMC”, qui devra être **asynchrone** et prendra les paramètres “(size:number, weight:number)”.
- On utilisera la méthode de calcul “ $IMC = ((poids * 10000) / (taille * taille))$ ” pour calculer le score d’IMC.
  - Si “size <= 0” :

```
if(size <= 0){  
  
    setResult(`Division par zéro impossible`);  
  
    setSize("");  
  
    setImc(0);  
  
    return;
```

```
}
```

- Créez une variable “**imc**” et stockez la formule de calcul (adaptée au code actuel, avec les paramètres **size** et **weight**) : “((Number(weight) \* 10000) / Number(size) \* Number(size))”.
- Transformez le contenu de la variable “**imc**” pour forcer l’affichage d’une décimale avec la méthode “**.toFixed(1)**”.
- **Mutez** l’état “**imc**”, en lui fournissant la variable “**imc**” comme valeur.

```
let imc = (Number(weight) * 10000) / (Number(size) *  
Number(size));  
  
imc = Number(imc.toFixed(1));  
  
setImc(imc);
```

- Créez une constante “**imcHint**”, qui stockera de manière **asynchrone** le retour de la méthode “**determineResultHint(Number(imc))**”.
- **Mutez** l’état “**size**” et donnez-lui une **chaîne de caractères vide** comme valeur.
- **Mutez** l’état “**weight**” et donnez-lui une **chaîne de caractères vide** comme valeur.

```
const imcHint = await determineResultHint(Number(imc));  
  
setSize("");  
  
setWeight("");
```

- Exécutez de manière **asynchrone** la méthode “**saveImcResultToHistory()**”, en lui fournissant les

paramètres suivants :

- `Number(size)`,
- `Number(weight)`,
- `Number(imc.toFixed(1))`,
- `imcHint`

```
await saveImcResultToHistory(  
  
    Number(size),  
  
    Number(weight),  
  
    Number(imc.toFixed(1)),  
  
    imcHint  
  
);
```

- Exécutez la méthode “`findHistory`”.

```
findHistory();
```

Voici le code complet de la méthode “`handleCalculateIMC`” :

```
const handleCalculateIMC = async (size: number, weight: number) => {  
  
    /**  
  
     * Formule de calcul de l'IMC :  
  
     *  $IMC = ((poids * 10000) / (taille * taille))$   
  
     */  
  
    if(size <= 0){  
  
        setResult(`Division par zéro impossible`);
```

```

        setSize("");

        setImc(0);

        return;
    }

    let imc = (Number(weight) * 10000) / (Number(size) * Number(size));

    imc = Number(imc.toFixed(1));

    setImc(imc);

    const imcHint = await determineResultHint(Number(imc));

    setSize("");

    setWeight("");

    await saveImcResultToHistory(

        Number(size),

        Number(weight),

        Number(imc.toFixed(1)),

        imcHint

    );

    findHistory();
}

```

## Retour du composant `ImcCalculatorScreen` :

- Ajoutez un composant **`SafeAreaView`**, ayant pour propriété `“style={styles.view}”`.
- Ajoutez un composant **`ScrollView`**, ayant pour propriété `“style={styles.scrollView}”`.
- Ajoutez un composant **`View`**, ayant pour propriété `“style={styles.contentView}”` et qui englobera :
  - Un composant **`View`**, ayant pour propriété `“styles.calculateIMCFormView”`, qui englobera :
    - Un composant **`TextInput`**, ayant les propriétés suivantes (`inputMode` permet de définir le type d’input attendu, `keyboardType` permet de modifier le type de clavier affiché sur mobile) :

```
<TextInput  
  
  placeholder="Votre taille en cm"  
  
  placeholderTextColor="#79797A"  
  
  inputMode="numeric"  
  
  keyboardType="numeric"  
  
  onChangeText={setSize}  
  
  value={size}  
  
  style={styles.textInputs}  
  
>
```

- Un composant **TextInput**, ayant les propriétés suivantes :

```
<TextInput  
  
  placeholder="Votre poids en kg"  
  
  placeholderTextColor="#79797A"  
  
  inputMode="numeric"  
  
  keyboardType="numeric"  
  
  onChangeText={setWeight}  
  
  value={weight}  
  
  style={styles.textInputs}  
  
>
```

- Un composant **CtaButton**, ayant les propriétés suivantes :

```
<CtaButton  
  
  props={{  
  
    text:"Calculer mon IMC",  
  
    borderRadius:100,  
  
    addStyles: {  
  
      width:"auto",  
  
      marginHorizontal:"auto",  
  
      marginVertical:0,  
  
      paddingVertical: 10,  
  
      paddingHorizontal: 24,  
  

```

```

    },

    addTextStyles: {

        fontSize: 16,

        fontWeight: "bold",

        letterSpacing: .1,

    },

    actionOnPress: () => handleCalculateIMC(Number(size),
Number(weight)),

    /**

    * Autre notation possible :

    * disabled: (weight.length <= 0) || (size.length <= 0)

    */

    disabled: ((size.length && weight.length) <= 0)

    }}

/>

```

- Un composant **View**, ayant pour propriété “style={styles.calculateIMCResultView}” et englobant :
  - Un composant **WelcomeText**, ayant les propriétés suivantes :

```

<WelcomeText

    props={{

        text: imc > 0 ? imc.toFixed(1) : imc,

```

```

        fontFamilyBold: true,

        addStyles: {

            fontSize:36,

            letterSpacing:.63,

            lineHeight:48,

            padding:0,

        }

    }}

/>

```

- Un composant **WelcomeText**, ayant les propriétés suivantes :

```

<WelcomeText

    props={{

        text: result.length > 0 ? result : "En attente des
informations",

        fontFamilyLight: true,

        addStyles:{

            padding:0,

        }

    }}

/>

```



Le composant **ImcCalculatorScreen** est terminé, voici son code complet :

```
import {  
  SafeAreaView,  
  ScrollView,  
  StyleSheet,  
  TextInput,  
  View,  
} from "react-native";  
  
import { useFonts } from "expo-font";  
  
import { useState } from "react";  
  
import { CtaButton, WelcomeText } from "@components/globals";  
  
import { saveImcResultToHistory } from "@shared/AsyncFunctions";  
  
import { useImcCalculatorContext } from  
"@shared/contexts/ImcCalculatorProvider";  
  
export default function ImcCalculatorScreen() {  
  
  const {findHistory} = useImcCalculatorContext();  
  
  const [size, setSize] = useState("");  
  
  const [weight, setWeight] = useState("");  

```

```

const [imc, setImc] = useState(0);

const [result, setResult] = useState("");

const [loaded, error] = useFonts({
  "Titillium Web Light": require("@/assets/fonts/TitilliumWeb-Light.ttf"),
  "Titillium Web Regular": require("@/assets/fonts/TitilliumWeb-Regular.ttf"),
  "Titillium Web Bold": require("@/assets/fonts/TitilliumWeb-Bold.ttf"),
});

const determineResultHint = async (imc: number) => {
  /**
   * TODO : transformer en gestion par état du contexte ImcCalculatorContext.
   */

  let imcHint = "";

  switch(true){
    case (imc <= 18.5):
      imcHint = `Maigreur, consultez un nutritionniste`;
      setResult(imcHint);
      return imcHint;

    case (imc <= 25 && imc > 18.5):
      imcHint = `Poids normal`;

```

```

    setResult(imcHint);

    return imcHint;

case (imc <= 30 && imc > 25):

    imcHint = `Surpoids`;

    setResult(imcHint);

    return imcHint;

case (imc <= 35 && imc > 30):

    imcHint = `Obésité modérée`;

    setResult(imcHint);

    return imcHint;

case (imc <= 40 && imc > 35):

    imcHint = `Obésité sévère, consultez un nutritionniste`;

    setResult(imcHint);

    return imcHint;

case (imc > 40):

    imcHint = `Obésité morbide, consultez un nutritionniste rapidement`;

    setResult(imcHint);

    return imcHint;

default:

    imcHint = `Les valeurs fournies sont incorrectes`;

    setResult(imcHint);

```

```

        return imcHint;
    }
}

const handleCalculateIMC = async (size: number, weight: number) => {

    /**
     * Formule de calcul de l'IMC :
     *  $IMC = ((poids * 10000) / (taille * taille))$ 
     */

    if(size <= 0){

        setResult(`Division par zéro impossible`);

        setSize("");

        setImc(0);

        return;
    }

    let imc = (Number(weight) * 10000) / (Number(size) * Number(size));

    imc = Number(imc.toFixed(1));

    setImc(imc);

    const imcHint = await determineResultHint(Number(imc));

```

```

setSize("");

setWeight("");

await saveImcResultToHistory(

    Number(size),

    Number(weight),

    Number(imc.toFixed(1)),

    imcHint

);

findHistory();

}

return (

    <SafeAreaView style={styles.view}>

        <ScrollView style={styles.scrollView}>

            <View style={styles.contentView}>

                <View style={styles.calculateIMCFormView}>

                    <TextInput

                        placeholder="Votre taille en cm"

                        placeholderTextColor="#79797A"

                        inputMode="numeric"

```

```

        keyboardType="numeric"

        onChangeText={setSize}

        value={size}

        style={styles.textInputs}

    />

```

```

<TextInput

    placeholder="Votre poids en kg"

    placeholderTextColor="#79797A"

    inputMode="numeric"

    keyboardType="numeric"

    onChangeText={setWeight}

    value={weight}

    style={styles.textInputs}

/>

```

```

<CtaButton

    props={{

        text:"Calculer mon IMC",

        borderRadius:100,

        addStyles: {

```

```

        width:"auto",

        marginHorizontal:"auto",

        marginVertical:0,

        paddingVertical: 10,

        paddingHorizontal: 24,

    },

    addTextStyles: {

        fontSize: 16,

        fontWeight: "bold",

        letterSpacing: .1,

    },

    actionOnPress: () => handleCalculateIMC(Number(size),
Number(weight)),

    /**
     * Autre notation possible :
     * disabled: (weight.length <= 0) || (size.length <= 0)
     */

    disabled: ((size.length && weight.length) <= 0)

  }}

/>

</View>

```

```

<View
  style={styles.calculateIMCResultView}
>

  <WelcomeText
    props={{
      text: imc > 0 ? imc.toFixed(1) : imc,
      fontFamilyBold: true,
      addStyles: {
        fontSize: 36,
        letterSpacing: .63,
        lineHeight: 48,
        padding: 0,
      }
    }}
  />

  <WelcomeText
    props={{
      text: result.length > 0 ? result : "En attente des informations",
      fontFamilyLight: true,
      addStyles:{

```



```

        padding:0,
      }
    }}

  />

</View>

</View>

</ScrollView>

</SafeAreaView>

);
}

const styles = StyleSheet.create({
  view: {
    flex: 1,
    backgroundColor: "#F6F4E8",
  },
  scrollView: {
    width: "100%",
  },
  contentView: {
    flex: 1,

```

```
    justifyContent: "center",

    alignItems: "center",

    gap: 48,

    padding: 20,
  },

  calculateIMCFormView: {

    justifyContent: "flex-start",

    alignItems: "stretch",

    gap: 24,

    padding: 0,
  },

  calculateIMCResultView: {

    justifyContent: "center",

    alignItems: "center",

    gap: 12,
  },

  textInputs: {

    width: 240,

    height: 40,

    borderRadius: 8,

    fontSize: 16,
```

```

    backgroundColor: "#ffffff",

    textAlign: "left",

    paddingHorizontal: 16,

    paddingVertical: 12,

    borderWidth: 1,

    borderColor: "#BACEC1",

    fontFamily: "Titillium Web Regular",

  },
});

```

## Création d'un écran - History

Rendez-vous dans le fichier “app > (global) > history > index.tsx”.

Ajoutez la feuille de style suivante au composant **HistoryScreen** :

```

const styles = StyleSheet.create({

  view: {

    flex: 1,

    justifyContent: "flex-start",

    alignItems: "stretch",

    gap: 0,

```

```

        backgroundColor: "#F6F4E8",

        paddingVertical: 42,

        paddingHorizontal: 20,

    },
});

```

### Logique du composant HistoryScreen :

- Déstructurez la méthode “useImcCalculatorContext()” et récupérez les valeurs “history”, “setHistory” et “findHistory”.

```
const {history, setHistory, findHistory} = useImcCalculatorContext();
```

- Créez un état “loading” et son mutateur “setLoading”, en donnant une valeur initiale à “true”.

```
const [loading, setLoading] = useState(true);
```

- Créez une méthode “initDatas”, qui est **asynchrone** et ne prend pas de paramètres.
  - Créez une constante “datas”, qui stockera le **retour asynchrone** de la méthode “findHistory()”.
  - **Mutez** l’état “loading” et donnez-lui la valeur “false”.

```

const initDatas = async () => {

    const datas = await findHistory();

    setLoading(false);

};

```

- Créez une méthode “**setCardBorderColor**”, qui prendra le paramètre “(imc: number)”.
  - Ajoutez un **switch()**, qui prendra la valeur “true” en paramètre et traitera les cas suivants :

- Si “imc > 40” :

```
case imc > 40:  
  
    return "#1D3124";
```

- Si “imc <= 18.5” ou “imc <= 40 && imc > 35” :

```
case imc <= 18.5:  
  
case imc <= 40 && imc > 35:  
  
    return "#ED1C24";
```

- Si “imc <= 35 && imc > 30” :

```
case imc <= 35 && imc > 30:  
  
    return "#E59560";
```

- Si “imc <= 30 && imc > 25” :

```
case imc <= 30 && imc > 25:  
  
    return "#BACEC1";
```

- Si “imc <= 25 && imc > 18.5” ou dans le cas par “défaut” :

```
case imc <= 25 && imc > 18.5:  
  
default:  
  
    return "#00A511";
```

- Utilisez la méthode “**useEffect()**” et exécutez la méthode “**initDatas()**” si “**loading**” est égal à “**true**”, et ajoutez-lui un **tableau de dépendances vide**.

```
useEffect(() => {  
  
    if(loading){  
  
        initDatas();  
  
    }  
  
}, []);
```

### Retour du composant HistoryScreen :

- Ajoutez un composant **SafeAreaView**, ayant pour propriété “**style={styles.view}**”.
- Englobez à l’intérieur une interpolation conditionnelle, vérifiant si “**(history.length > 0 && !loading)**”.

```
return (  
  
    <SafeAreaView style={styles.view}>  
  
        {(history.length > 0 && !loading) ? () : ()}  
  
    </SafeAreaView>  
  
);
```

- Si la valeur est “**true**” :
- Retournez un composant **FlatList** (fourni par “**react-native**”), contenant les propriétés suivantes :

```

<FlatList
  data={history.sort((a: any, b:any) => {

    const elementA = Number(a.timestamp);

    const elementB = Number(b.timestamp);

    // Tri par ordre décroissant

    // Pour inverser l'ordre de tri, il suffit de modifier les
retours des deux conditions suivantes (-1 / 1).

    if(elementA < elementB){

      return 1;

    }

    if(elementA > elementB){

      return -1;

    }

    return 0;

    // Fin du tri par ordre décroissant

  })}

  renderItem={({data: any}) => (

    <Card

      props={{

        title: data.item.firstname,

```

```

        borderColor: setCardBorderColor(
            Number(data.item.imc)
        ),
    }}
>
<Text>{data.item.date}</Text>

<Text>

    IMC : {data.item.imc} -{" "}

    {data.item.imcHint}

</Text>

<Text>

    Taille : {data.item.size}{" "}

    {data.item.sizeUnit}

</Text>

<Text>

    Poids : {data.item.weight}{" "}

    {data.item.weightUnit}

</Text>

</Card>

)}}

keyExtractor={({item: any}) => item.timestamp}

```



```

ItemSeparatorComponent={() => (
  <View
    style={{
      width: "100%",
      height: 24,
    }}
  ></View>
)}
/>

```

- Sinon, retournez un **React Fragment**, qui contiendra un affichage conditionnel sur l'état **"loading"** et renverra :
  - Si **"loading"** est égal à **"true"** :
    - Retournez un composant **ActivityIndicator** (fourni par **"react-native"**), ayant les propriétés suivantes :
      - **size = "large"**
      - **color = "#E59560"**
  - Sinon :
    - Retournez un composant **Card**, qui aura une propriété **"props={{ title: "Aucunes données" }}"** et englobera :
      - Un composant **Text**, contenant le texte **"Vous n'avez jamais calculé votre IMC"**.
      - Un composant **CtaButton**, ayant une propriété

```

    "props = {{
      text: "Calculer mon IMC",
      actionOnPress: () =>
        router.push("imc-calculator"),
      addStyles: {
        marginLeft: "auto",
        marginTop: 24,
        padding: 6,
      }
    }}".

```

```

<>

{loading ? (

  <ActivityIndicator

    size="large"

    color="#E59560"

  />

) : (

  <Card

    props={{

      title: "Aucunes données"

    }}

  >

    <Text>Vous n'avez jamais calculé votre IMC.</Text>

    <CtaButton

```

```
        props={{
            text:"Calculer mon IMC",
            actionOnPress: () =>
router.push("imc-calculator"),
            addStyles:{
                marginLeft: "auto",
                marginTop:24,
                padding:6,
            }
        }}
    />
</Card>
    )}
</>
```

Le composant **HistoryScreen** est désormais terminé.

Voici son code complet :

```
import { ActivityIndicator, FlatList, SafeAreaView, StyleSheet, Text, View } from
"react-native";

import { getSavedHistory } from "@shared/AsyncFunctions";

import { useEffect, useState } from "react";

import { Card, CtaButton } from "@components/globals";

import { router } from "expo-router";

import {useImcCalculatorContext} from "@shared/contexts/ImcCalculatorProvider";

export default function HistoryScreen() {

  const {history, setHistory, findHistory} = useImcCalculatorContext();

  const [loading, setLoading] = useState(true);

  const initDatas = async () => {

    const datas = await findHistory();

    setLoading(false);

  };

  const setCardBorderColor = (imc: number) => {

    switch (true) {

      case imc > 40:

        return "#1D3124";

      case imc <= 18.5:
```

```

        case imc <= 40 && imc > 35:

            return "#ED1C24";

        case imc <= 35 && imc > 30:

            return "#E59560";

        case imc <= 30 && imc > 25:

            return "#BACEC1";

        case imc <= 25 && imc > 18.5:

        default:

            return "#00A511";

    }

};

useEffect(() => {

    if(loading){

        initDatas();

    }

}, []);

return (

    <SafeAreaView style={styles.view}>

        {(history.length > 0 && !loading) ? (

            <FlatList

                data={history.sort((a: any, b:any) => {

```

```

    const elementA = Number(a.timestamp);

    const elementB = Number(b.timestamp);

    // Tri par ordre décroissant

    // Pour inverser l'ordre de tri, il suffit de modifier les retours
des deux conditions suivantes (-1 / 1).

    if(elementA < elementB){

        return 1;

    }

    if(elementA > elementB){

        return -1;

    }

    return 0;

    // Fin du tri par ordre décroissant
  })}

renderItem=(({data: any}) => (

  <Card

    props={{

      title: data.item.firstname,

      borderColor: setCardBorderColor(

        Number(data.item.imc)

```

```

        ),
    }}

>

<Text>{data.item.date}</Text>

<Text>

    IMC : {data.item.imc} -{" "}

    {data.item.imcHint}

</Text>

<Text>

    Taille : {data.item.size}{" "}

    {data.item.sizeUnit}

</Text>

<Text>

    Poids : {data.item.weight}{" "}

    {data.item.weightUnit}

</Text>

</Card>

)}}

keyExtractor=({item: any}) => item.timestamp}

ItemSeparatorComponent={() => (

    <View

```

```

        style={{
            width: "100%",
            height: 24,
        }}
    </View>

)}

/>

) : (

<>

{loading ? (

    <ActivityIndicator

        size="large"

        color="#E59560"

    />

) : (

    <Card

        props={{

            title: "Aucunes données"

        }}

    >

        <Text>Vous n'avez jamais calculé votre IMC.</Text>

```



```

        <CtaButton
            props={{
                text:"Calculer mon IMC",
                actionOnPress: () => router.push("imc-calculator"),
                addStyles:{
                    marginLeft: "auto",
                    marginTop:24,
                    padding:6,
                }
            }}
        />
    </Card>

    )}

</>

)}

</SafeAreaView>

);
}

const styles = StyleSheet.create({
    view: {
        flex: 1,

```

```
justifyContent: "flex-start",  
  
alignItems: "stretch",  
  
gap: 0,  
  
backgroundColor: "#F6F4E8",  
  
paddingVertical: 42,  
  
paddingHorizontal: 20,  
  
},  
  
});
```

## Fin de la création de l'application IMC Calculator

Votre application est fonctionnelle.

Nous allons désormais voir les points personnalisables avant de parler de la compilation d'une application avec **Expo**, pour générer un SDK.

Rendez-vous dans le fichier "**app.json**".

Vous pouvez créer des ressources personnalisées pour modifier les paramètres suivants :

- **icon**: vous permet de modifier l'icône de l'application.
- **image** (splash) : Vous permet de modifier l'image affichée lors du chargement de l'application.
- **adaptativeIcon** (android) : Vous permet de modifier l'icône à taille variable pour les appareils Android.

Attention, si vous modifiez une ressource graphique dans ce fichier, pensez à modifier le nom pour ne pas garder le nom par défaut, sinon votre application risque de conserver ses informations de cache.

```
"icon": "./assets/images/icon.png",  
"scheme": "myapp",  
"userInterfaceStyle": "automatic",  
"splash": {  
  "image": "./assets/images/splash.png",  
  "resizeMode": "contain",  
  "backgroundColor": "#ffffff"  
},
```

```
"android": {  
  "adaptiveIcon": {  
    "foregroundImage": "./assets/images/adaptive-icon.png",  
    "backgroundColor": "#ffffff"  
  }  
},
```

Nous allons désormais voir comment préparer la compilation de l'application.

## Préparation de la compilation d'une application - Prebuild

Avant de compiler l'application, on peut effectuer une action de “**prebuild**”, pour pré-compiler l'application et apporter des modifications sur les fichiers spécifiques à android (création d'un dossier android lors de l'opération).

Par exemple, si vous souhaitez modifier la taille par défaut du stockage local, avec AsyncStorage, vous devrez ajouter une configuration dans le fichier “**android/gradle.properties**”.

```
AsyncStorage_db_size_in_MB=10
```

Ouvrez votre terminal, placez-vous à la racine de votre projet et lancez la commande :

```
npx expo prebuild
```

Pour créer un pre-build IOS, entrez la commande :

```
npx expo prebuild --platform ios
```

Étapes de prebuild :

**What would you like your Android package name to be?** : Vous permet de définir l'adresse d'accès à l'application. Validez la proposition par défaut.

Si vous n'avez pas d'erreur dans l'application, votre prebuild sera validé et le dossier “**android**” sera créé.

## Créer un APK en compilant son application - Expo build

Lien de la documentation officielle pour un build Android et une publication sur Google Play :

<https://docs.expo.dev/tutorial/eas/android-production-build/>.

La mise en ligne sur Google Play store nécessite un compte développeur, qui est facturé **\$25**, en **paiement unique**.

Lien de la documentation officielle pour un build IOS et une publication sur Apple Store : <https://docs.expo.dev/tutorial/eas/ios-production-build/>.

Un compte développeur chez Apple coûte **99\$ par an**. Si vous voulez diffuser une application dans votre entreprise, il faut adhérer à l'Apple Developer Enterprise Program, qui coûte **299\$/an**.