Dev mobile 102 : Components, Navigation et API

Prérequis

Pour suivre cette formation, il faut évidemment avoir suivi la Dev mobile 101. Et aussi, vraiment, la Dev web 104: le JavaScript.

Le projet

Pour avoir un fil conducteur tout au long de ces formations, nous allons créer une application qui pourrait être utile au GInfo : elle permettra aux utilisateurs de laisser un feedback anonyme sur les formations.

Les Components

C'est quoi un component ?

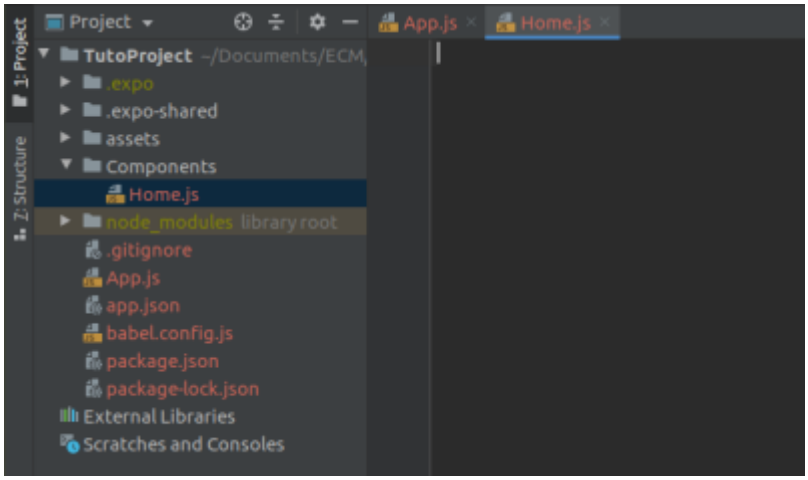
Avec cette formation, vous allez découvrir ce qui fait la puissance de React (et que React Native utilise aussi) : les components. Lors de la précédente formation, vous avez probablement entendu ce mot, pour parler des 'balises' JSX, comme `<View>` ou `<Text>` par exemple. Ce sont des components.

Et la grande force de React, c'est que maintenant, nous allons créer nos propres Component ! Vous pourrez ainsi définir, par exemple, un component `Utilisateur` qui affiche joliment un utilisateur, et dans toutes vos pages qui ont besoin d'afficher un utiliasteur, vous mettrez simplement `<Utilisateur/>` dans le JSX, au milieu des autres components de React Native.

Je suis désolé mais en vrai c'est trop stylé.

Créer un component

Nous allons créer un premier component. Tout d'abord, créons un dossier `Components` dans notre projet, et un fichier `Home.js` dedans.



Ensuite, nous devons importer `React`, afin d'utiliser la classe `React.component`. En effet, nous allons ensuite définir une nouvelle classe, héritant de cette classe générique. À la fin du fichier, on exporte cette classe. Cela permettra à d'autres fichiers de l'utiliser.

```
import React from 'react';

class Home extends React.Component {

}

export default Home;
```

Les components doivent forcément disposer d'une méthode `render`, qui retourne une vue JSX. Nous allons faire le minimum pour notre component `Home` :

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

class Home extends React.Component {
     render() {
        return (
            <View style={styles.redView}/>
        )
    }
}

const styles = StyleSheet.create({
    redView: {
        flex:1,
        backgroundColor: '#f00',
    }
});

export default Home;
```

On a simplement crée la méthode `render`, qui retourne une `View`, avec une `StyleSheet` pour accueillir nos styles. N'oubliez pas les import !

Utilisons notre component

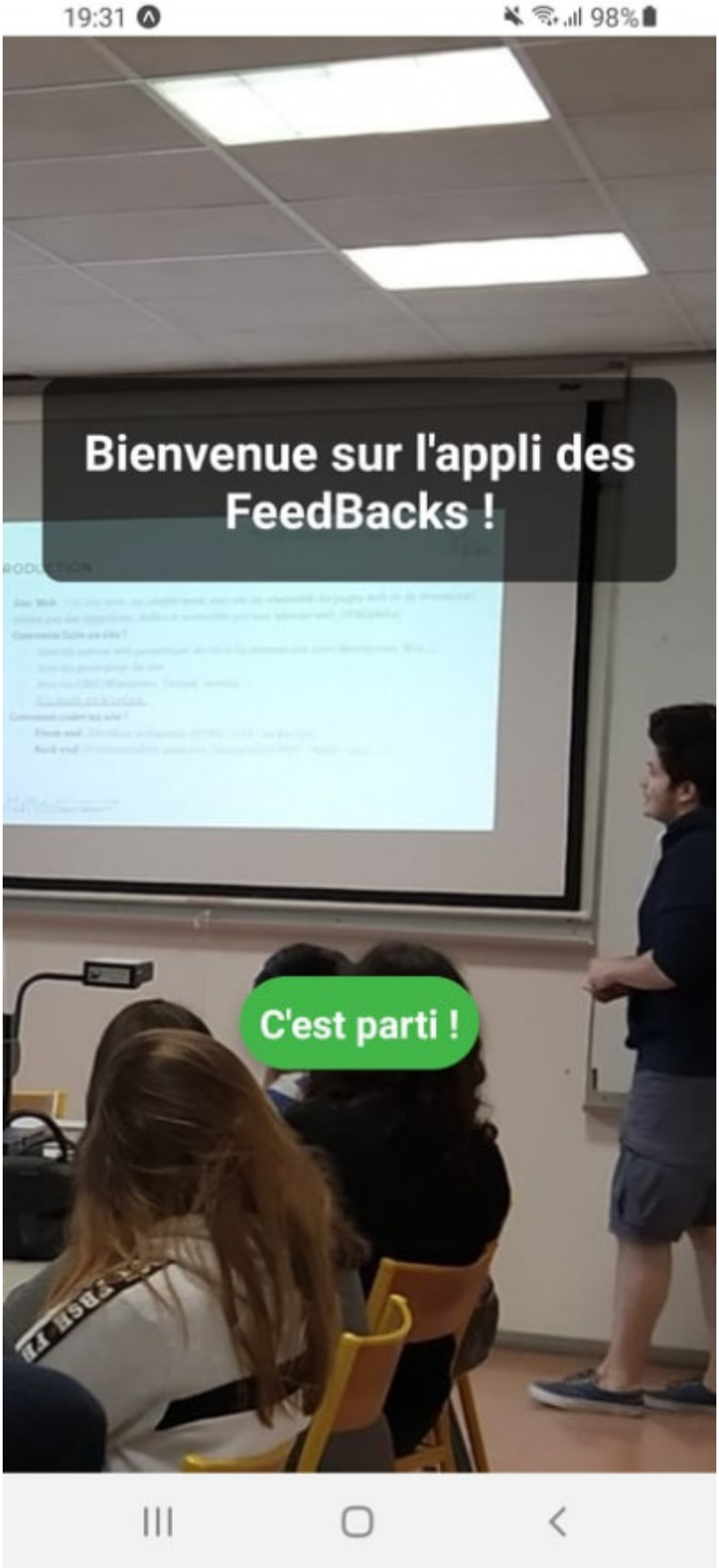
Pour utiliser notre component, nous allons modifier le `App.js`. Il faut importer le component `Home`, et l'appeler dans le JSX. Ici, le component est seul.

```
import Home from "./Components/Home";

export default function App() {
  return (
    <View style={styles.container}>
        <Home/>
    </View>
  );
}
```

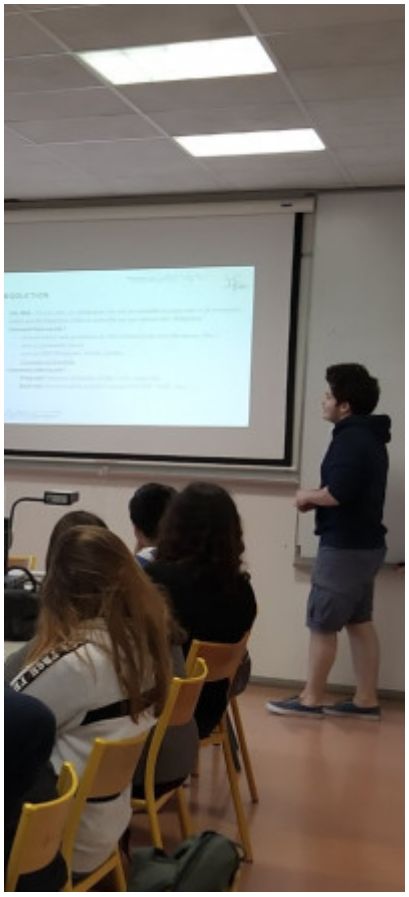
Et maintenant, quand on lance l'application, on voit une page toute rouge ! On a bien créé et utilisé un component.

Un peu de pratique

Maintenant, nous allons utiliser ce component `Home` pour créer la page d'accueil de noter application. Modifiez le fichier `Home.js` pour obtenir une page d'accueil de ce type :



vous pouvez utiliser cette image de fond :



Correction :

```
import React from 'react';
import {View, StyleSheet, Image, Text, TouchableOpacity} from "react-
native";

class Home extends React.Component {

    goBtn = () => {
        console.log('OUIIIIIII')
    }

    render() {
        return (
            <View style={styles.redView}>
                <Image source={require('../assets/forma_bkg.jpg')}
style={styles.imgBackG}/>
                <View style={styles.titleContainer}>
                    <View style={styles.title}>
                        <Text style={styles.titleText}>Bienvenue sur
l'appli des FeedBacks !</Text>
                    </View>
                </View>
                <View style={styles.goBtnContainer}>
                    <TouchableOpacity style={styles.goBtn}
onPress={this.goBtn}>
                        <Text style={styles.goBtnText}>C'est parti !</Text>
```

```
                    </TouchableOpacity>
                </View>
            </View>
        )
    }
}

const styles = StyleSheet.create({
    redView: {
        flex:1,
    },
    imgBackG: {
        position: 'absolute',
        top: 0,
        left:0,
        bottom:0,
        right: 0
    },
    titleContainer: {
        flex: 2,
        alignItems: 'center',
        justifyContent: 'center',
    },
    title: {
        backgroundColor: '#000000A0',
        padding: 20,
        margin: 20,
        borderRadius: 10,
    },
    titleText: {
        color: '#fff',
        fontSize: 25,
        fontWeight: 'bold',
        textAlign: 'center',
    },
    goBtnContainer: {
        flex: 1,
        alignItems: 'center',
        justifyContent: 'flex-start',
    },
    goBtn: {
        backgroundColor: '#42b649',
        padding: 10,
        borderRadius: 100,
        elevation: 10,
    },
    goBtnText: {
        color: '#fff',
        fontSize: 20,
        fontWeight: 'bold'
    }
```

```
});

export default Home;
```

La Navigation

De quoi on parle ?

On vient de créer une page d'accueil pour notre application, avec un joli bouton “C'est parti !”. Mais pour l'instant, ce bouton n'a aucun effet. Mais que doit faire ce bouton ?

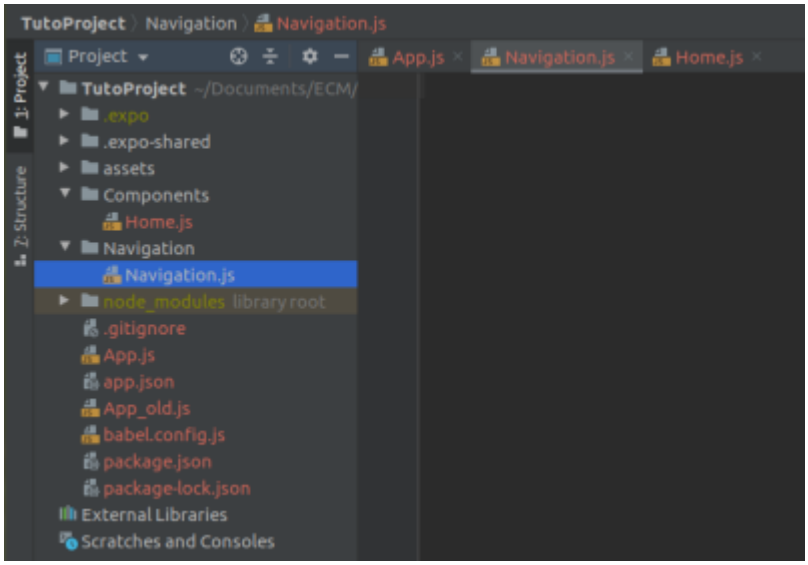
Su un site web, vous vous attendriez à ce que ce bouton vous renvoie vers une autre page. C'est exactement ce que nous voulons faire. Pour cela nous allons utiliser la `Navigation`, pour naviguer entre nos vues. La navigation n'est pas incluse directement dans react-native, il faut l'installer :

```
npm install react-navigation react-navigation-stack
```

Créer un Navigator

Pour implémenter une navigation, nous allons créer un `StackNavigator`. Il existe plusieurs styles de `Navigator`, vous pourrez regarder leurs particularités sur la documentation. Le `StackNavigator` est le plus simple : les vues (ou 'pages') sont conceptuellement empilées les unes sur les autres (visuellement, on n'a que la vue du dessus de la pile qui est affichée); ce qui permets par exemple une gestion du bouton de retour très naturelle.

Commençons par créer un dossier `Navigation`, et dedans, un fichier `Navigation.js`.



Dedans, nous allons importer les fonctions permettant de créer un `StackNavigator` et de l'exporter, pour le mettre à disposition des autres fichiers. Voilà le code :

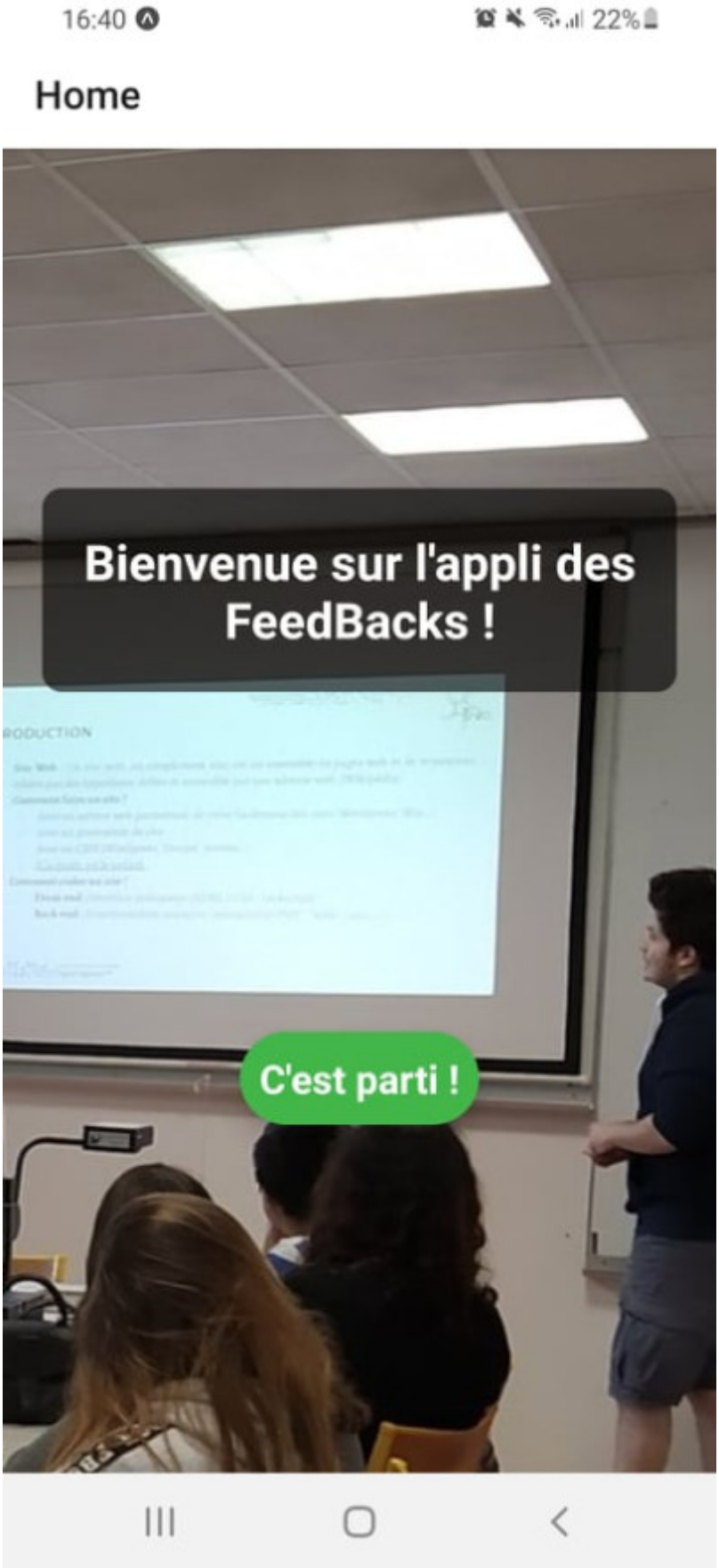
```
import {createStackNavigator} from 'react-navigation-stack';
import {createAppContainer} from 'react-navigation';
```

```
const AppNavigator = createStackNavigator({})

export default createAppContainer(AppNavigator);
```

Maintenant, tout va se passer dans le `createStackNavigator`. On ajoute le component `Home` en premier (sans oublier de l'importer). Pour ne pas revenir dessus, on va aussi créer un component `FormaList`, qu'on modifiera plus tard (pour l'instant une page rouge suffit). Et on l'ajoute aussi au `StackNavigator` :

```
import Home from '../Components/Home'
import FormaList from "../Components/FormaList";

const AppNavigator = createStackNavigator({
    Home: {
        screen: Home
    },
    FormaList: {
        screen: FormaList
    }
})
```

Et pour utiliser cette navigation dans l'appli, on va simplement modifier le `App.js` : remplacer `import Home from './Components/Home';` par `import Navigation from './Navigation/Navigation';`; et <Home/> par <Navigation/> dans le JSX. On lance l'appli... et on voit que seul le component `Home` est affiché (car c'est le premier dans la liste).



Naviguons

Jusqu'à maintenant, on a mis en place une navigation, c'est cool... mais rien n'a changé dans notre application. Rappelons que l'objectif était de pouvoir cliquer sur le bouton 'C'est parti !' et arriver sur

une autre page.

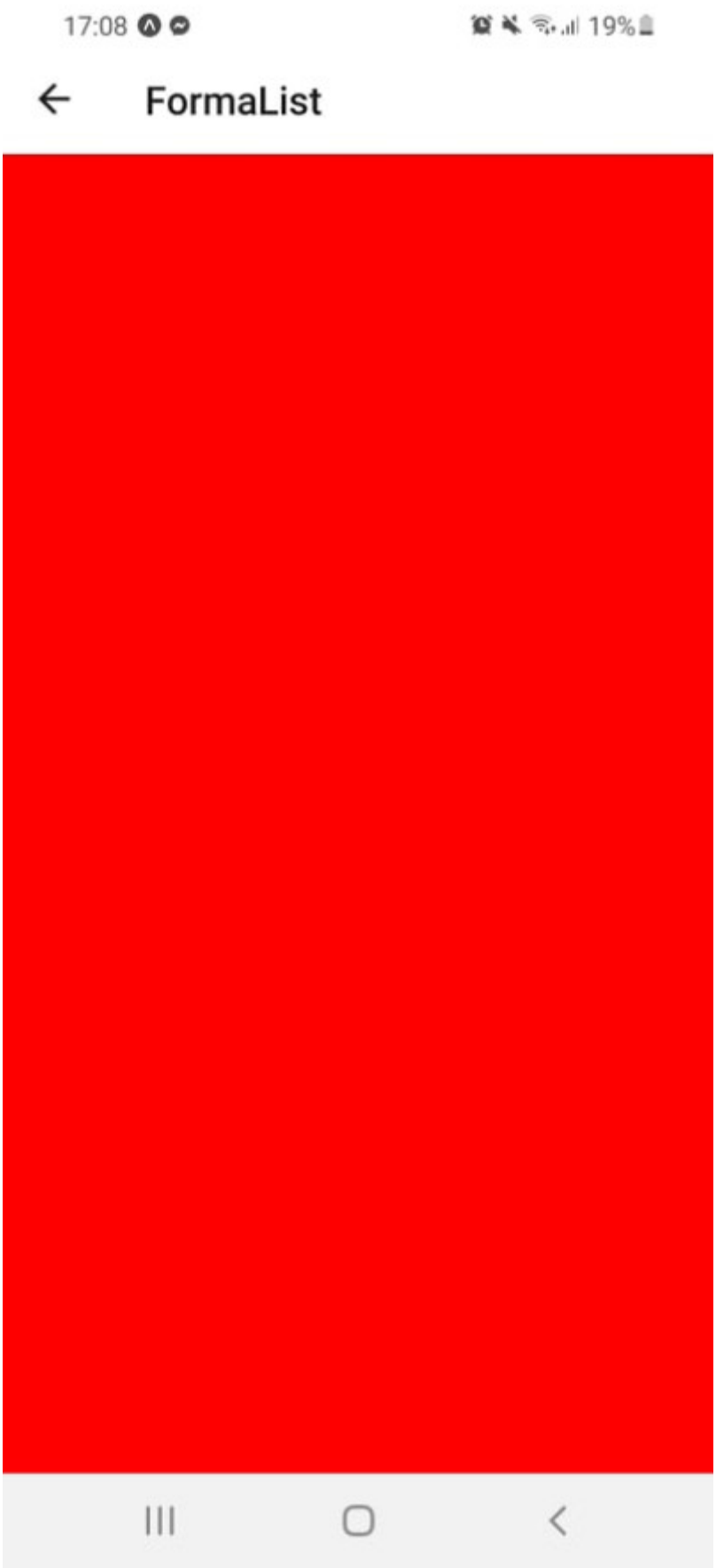
Tout d'abord, il faut gérer le clic sur le bouton. Les `TouchableOpacity` ont une prop `onPress`, qui prends la fonction à appeler lorsqu'on appuie dessus. On va donc ajouter `onPress={this.goBtn}`, et créer la méthode `goBtn` dans notre component :

```
    goBtn() {
        console.log('OUIIIIIII')
    }
```

Comme ça, lorsqu'on clique sur le bouton, la console affiche 'OUIIIIIII'. Donc on sait déclencher une action par un clic. Maintenant, on va modifier `goBtn` pour ajouter la navigation (on la passe en fonction fléchée pour pouvoir utiliser `this`) :

```
    goBtn = () => {
        console.log('OUIIIIIII')
        this.props.navigation.navigate('FormaList')
    }
```

Et quand on clique... TADAAMMM ! On arrive sur une vue toute rouge, avec 'FormaList' en titre ! On a bien navigué.



Se connecter à une API Web

Rappel des objectifs

Le projet est de faire une application permettant aux utilisateurs d'accéder à la liste des formations du GInfo et à la liste des sessions de chaque formation, afin qu'ils puissent laisser des commentaires sur chaque session.

Cela nécessite de partager un certain nombre de données entre les différent utilisateurs : les formations, les sessions, et les commentaires. Autrement dit, nous avons besoin d'un back-end. Étant donné que ce n'est pas le sujet de cette formation, nous avons développé le back-end, qui est acessible à l'adresse https://feedback-forma.ginfo.centrale-marseille.fr. Elle est toute simple, et réponds aux routes suivantes :

- `/list` - JSON La liste des formations (id, nom, image, description)
- `/img/{nom_de_l'image}` - lien pour accéder à l'image
- `/forma/{id_de_la_formation}` - JSON La formation portant l'id correspondant (id, nom, image, description)
- `/forma/{id_de_la_formation}/sessions` - JSON La liste des sessions de la formation concernée (id, formaId, dateTime, formateurs)
- `/session/{id_de_la_session}` - JSON La session portant l'id correspondant (id, formaId, dateTime, formateurs)
- `/session/{id}/coms` - JSON La liste des commentaires de la session (id, sessionId, dateTime, contenu)

Et une autre route pour l'envoi de commentaires (ce sera pour une autre fois héhé).

fetch

La prochaine étape pour l'application est de récupérer des données de cette API. Pour cela, on utilise `fetch`, la fonction qui permets de faire des requêtes HTTP en JS. Dans le component `FormaList`, on va ajouter une méthode `loadFormas`, qui récupère la liste des formations.

```
    loadFormas() {
fetch('https://feedback-forma.ginfo.centrale-marseille.fr/list').then((respo
nse) => {
            return response.json()
        }).then((data) => {
            console.log(data[0]);
        })
    }
```

Cette fonction va simplement récupérer la liste des formations, et afficher la première dans la console. Avec `fetch`, on utilise `then`, car ici on veut faire de la programmation asynchrone (on récupère la réponse, **PUIS** on utilise les données), ce qui n'est pas le cas du JS en général.


```
Object {
  "description": "Dans cette formation, tu apprendra les bases du devellopement web, notamment le HTML et le CSS !",
  "id": 1,
  "image": "logo.png",
  "nom": "Dev Web 1",
}
```

Cette liste de formation qu'on récupère, il va aussi falloir la stocker. On va donc ajouter des attributs à notre component : `isLoading`, pour pouvoir afficher une écran de chargement, et `formas`, pour stocker la liste des formations. On les crée dans le `constructor` (faites de la POO merde). Et on va les modifier dans la méthode `loadFormas` :

```
    constructor(props) {
        super(props);
        this.formas = [];
        this.isLoading = true;
    }

    loadFormas() {
        if(this.isLoading){
fetch('https://feedback-forma.ginfo.centrale-marseille.fr/list').then((respo
nse) => {
                return response.json()
            }).then((data) => {
                this.formas = data
                this.isLoading = false
                setTimeout(() => {this.forceUpdate()}, 1000)
            })
        }
    }
```

À quoi sert l'attribut `isLoading`, et que fait la méthode `forceUpdate` (et pourquoi utilise-t-on ce `setTimeout`) ? On modifie le `render`, en ajoutant aussi les méthodes `displayLoading` et `displayFormas` (pour faire plus propre) :

```
    displayLoading() {
        return (
            <View style={styles.redView}>
                <ActivityIndicator size="large" color="#00ff00" />
            </View>
        )
    }

    displayFormas() {
        return (
            <View style={styles.blueView} />
        )
    }

    render() {
        this.loadFormas()
        return (
            <View style={styles.mainContainer}>
                {this.isLoading ? this.displayLoading() :
this.displayFormas()}
            </View>
        )
```

```
    }
```

Avec tout ce code, voilà ce qu'il se passe lorsqu'on clique sur le bouton pour passer sur la page `FormaList` :

1. Le `constructor` est exécuté : `formas = []` et `isLoading = true`
2. La méthode `render` s'exécute une première fois : elle appelle la méthode `loadFormas`, mais retourne quand même une valeur. Vu que `isLoading` est vrai, on prends l'expression avant les deux points, donc `displayLoading`, qui rends une page rouge avec un `ActivityIndicator` vert.
3. La méthode `loadFormas` s'exécute. On entre dans le `if`, on envoie une requête pour avoir la liste des formations. Lorsqu'on la reçoit, on mets cette liste dans `formas` et `isLoading = false` (on a finis de charger). On appelle le `setTimeout`
4. `setTimeout`, c'est simple : ça va permettre d'exécuter un code (ici on appelle `forceUpdate`) après un délai (ici 1000ms = 1 seconde). Si on ne mets pas ça, la page de chargement va s'afficher pendant moins de 20ms, donc c'est moche
5. Après 1 seconde donc, le `forceUpdate` est appelé. Il permets simplement de recharger le component, c'est à dire qu'il appelle une nouvelle fois `render`.
6. `render` est appelé, mais cette fois-ci `isLoading` est faux, donc on passe dans la deuxième option : `displayForma`. C'est ici qu'on va faire notre vraie page !

Vous avez tout compris ? Cool. Maintenant, on va afficher ces formations !

Une FlatList

Les `FlatList`, c'est cool. En gros c'est comme si tu avais une boucle `for` sur les éléments d'une liste, qui à chaque fois ajoute un certain component pour représenter l'élément, sauf que ça le fait tout seul. Voilà la syntaxe :

```
    displayFormas() {
        return (
            <ScrollView style={styles.mainContainer}>
                <FlatList
                    data={this.formas}
                    keyExtractor={(item) => item.id.toString()}
                    renderItem={({item}) => <FormaItem forma={item}/>}
                />
            </ScrollView>
        )
    }
```

On a mis une `ScrollView` pour pouvoir scroller dans la liste si elle est trop longue. Et il va falloir créer ce component `FormaItem`, et l'importer ici ! Mais comment récupérer la formation concernées ? On a écris `<FormaItem forma={item}/>`, c'est à dire qu'on a ajouté une `prop` ayant pour nom `forma` et pour valeur l'élément concerné. Dans le component `FormaItem`, on va par exemple pouvoir récupérer le nom de la formation :

```
<Text style={styles.titleText}>{this.props.forma.nom}</Text>
```

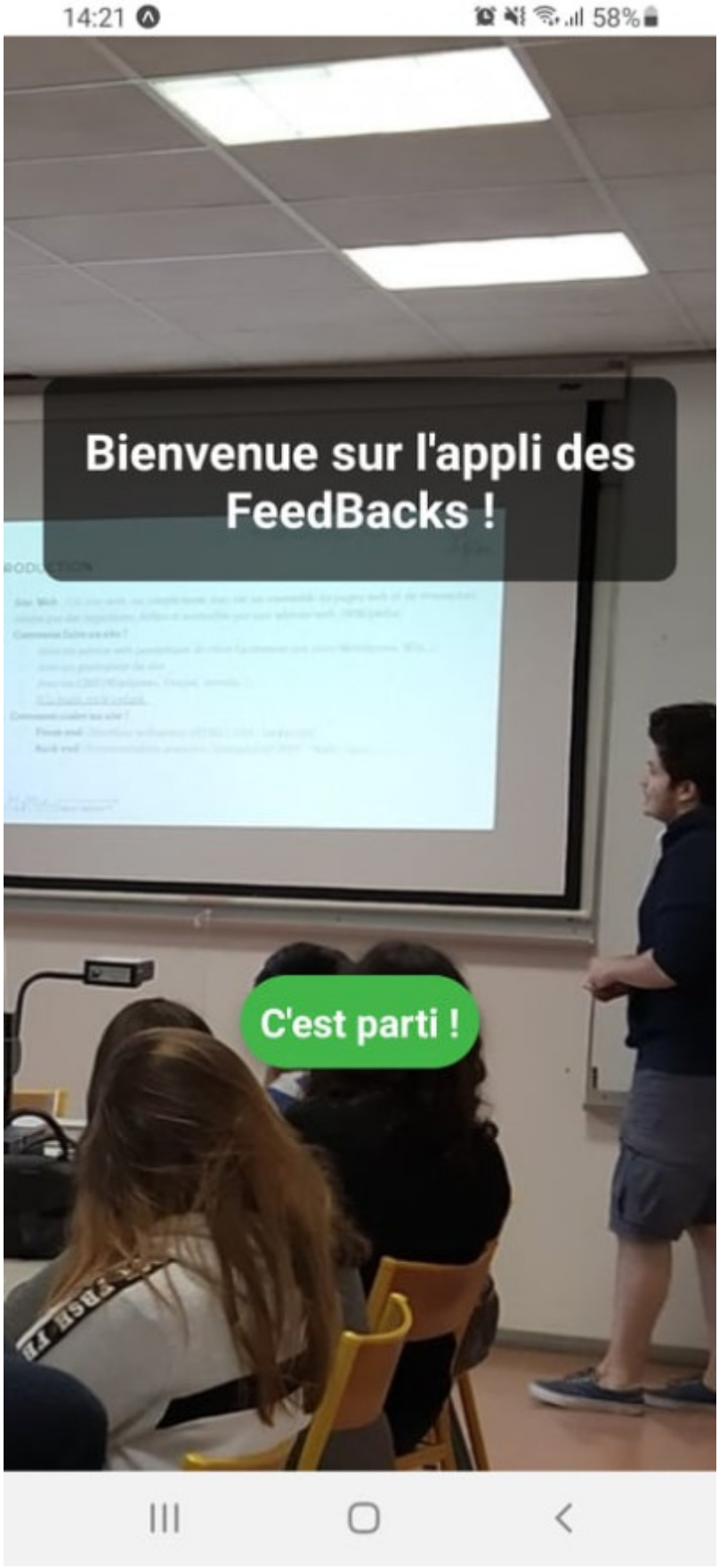
Le component en entier si vous avez pas le temps, pour voir ce que ça peut donner :

```
class FormaItem extends React.Component {

    render() {
        return (
            <View style={styles.redView}>
                <Image
source={{uri:'https://feedback-forma.ginfo.centrale-marseille.fr/img/'+this
.props.forma.image}}
                    style={styles.imgForma}
                />
                <View style={styles.titleView}>
                    <Text
style={styles.titleText}>{this.props.forma.nom}</Text>
                </View>
            </View>
        )
    }
}

const styles = StyleSheet.create({
    redView: {
        backgroundColor: 'grey',
        alignItems: 'center',
        flexDirection: 'row',
        padding: 10,
        margin: 10
    },
    imgForma: {
        height: 100,
        width: 100,
        backgroundColor: 'white'
    },
    titleView: {
        marginLeft: 50
    },
    titleText: {
        color: '#fff',
        fontSize: 25,
        fontWeight: 'bold',
        textAlign: 'center',
    },
});
```

On obtiens quelque chose comme ça :

Pratiquez !

L'objectif est d'avoir toutes les pages prêtes :


14:21
58%
Bienvenue sur l'appli des FeedBacks !
C'est parti !


14:22
58%
Les formations du GInfo
Sélectionne une formation pour voir les détails et laisser des commentaires !
Dev Web 5
Dev Mobile 1
Dev Mobile 2
Dev Mobile 3


14:22
57%
Dev Mobile 2
Dans cette formation, tu apprendra à connecter ton application à une API, pour afficher cette formation par exemple !
Pilouf
2021-11-18 Par Pilouf et foder
2021-11-19 Par Pilouf et foder
2021-11-20 Par Glenn et Louradou
2021-11-20 Par Marie et foder
2021-11-23 Par Marie, foder et Pilouf
2021-11-23 Par vory et foder

La correction sur le gitlab

From:
https://wiki.centrale-med.fr/ginfo/ - **Wiki GInfo**

Permanent link:
https://wiki.centrale-med.fr/ginfo/formations:devmobile_2

Last update: **18/11/2021 18:36**