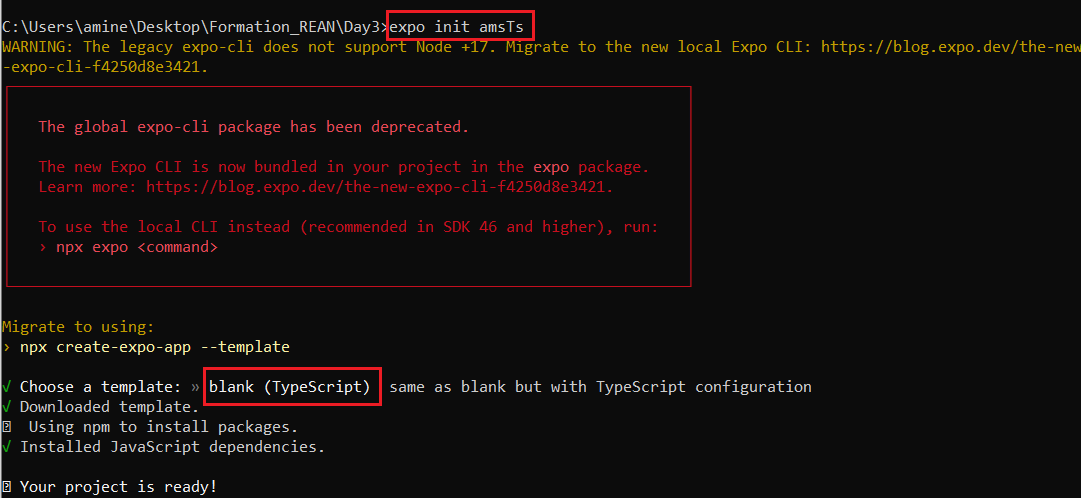
1. **Créer un projet React Native avec template TypeScript**

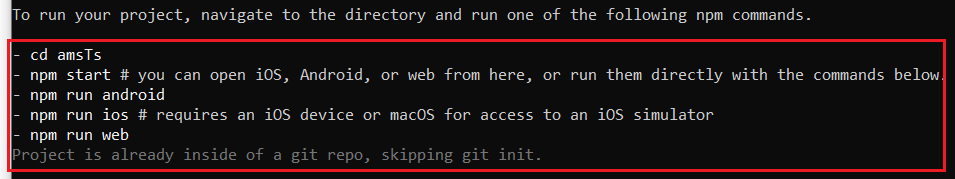
Pour créer un projet React Native avec TypeScript en utilisant Expo, vous pouvez utiliser la commande suivante :

Lorsque vous exécutez cette commande, Expo vous demandera de choisir un template. Pour un projet TypeScript, sélectionnez l'option **tabs (TypeScript)** ou **blank (TypeScript)** selon vos préférences. Le template **tabs (TypeScript)** fournit quelques onglets de navigation comme point de départ, tandis que **blank (TypeScript)** est plus minimaliste.

Voici les étapes en détail :

1. Ouvrez un terminal.
2. Exécutez la commande **expo init NomDuProjet** pour démarrer le processus de création du projet.
3. Sélectionnez un template TypeScript (par exemple, **tabs (TypeScript)** ou **blank (TypeScript)**).
4. Une fois la commande terminée, naviguez dans votre nouveau projet en exécutant **cd NomDuProjet**.
5. Démarrez le serveur de développement en exécutant **expo start**

****

****

**Voici le code du fichier App.tsx**

import { StatusBar } from 'expo-status-bar';

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

export default function App() {

  return (

    <View style={styles.container}>

      <Text>Open up App.tsx to start working on your app!</Text>

      <StatusBar style="auto" />

    </View>

  );

}

const styles = StyleSheet.create({

  container: {

    flex: 1,

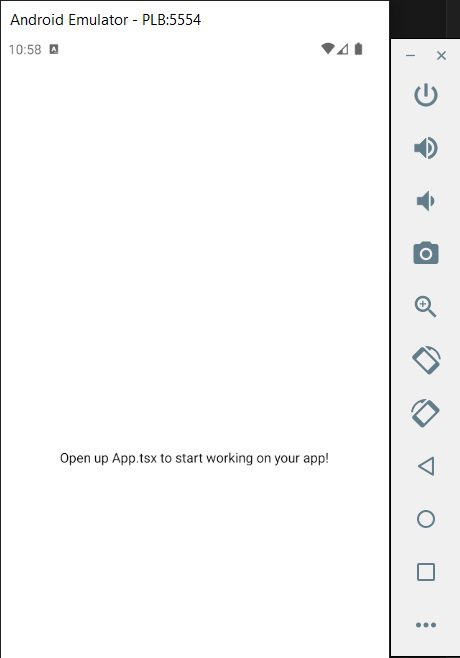
    backgroundColor: '#fff',

    alignItems: 'center',

    justifyContent: 'center',

  },

});

****

1. **Les tests unitaires**

Pour écrire un test unitaire en TypeScript pour un composant React Native, vous utiliserez généralement des outils comme Jest et React Native Testing Library. Voici un exemple de test unitaire pour un composant simple en React Native utilisant TypeScript.

### Étape 1 : Installation des dépendances

Si elles ne sont pas déjà installées, vous aurez besoin de Jest et de React Native Testing Library. Installez-les en exécutant :

**npm install --save-dev jest @testing-library/react-native**

Assurez-vous également que vous avez les types TypeScript pour Jest :

**npm install --save-dev @types/jest**

### Étape 2 : Configurer Jest pour TypeScript

Si ce n'est pas déjà fait, configurez Jest pour qu'il comprenne TypeScript. Vous pourriez avoir besoin d'un fichier **jest.config.js** avec un contenu comme :

**module.exports = {**

**preset: 'react-native',**

**moduleFileExtensions: ['ts', 'tsx', 'js', 'jsx', 'json', 'node'],**

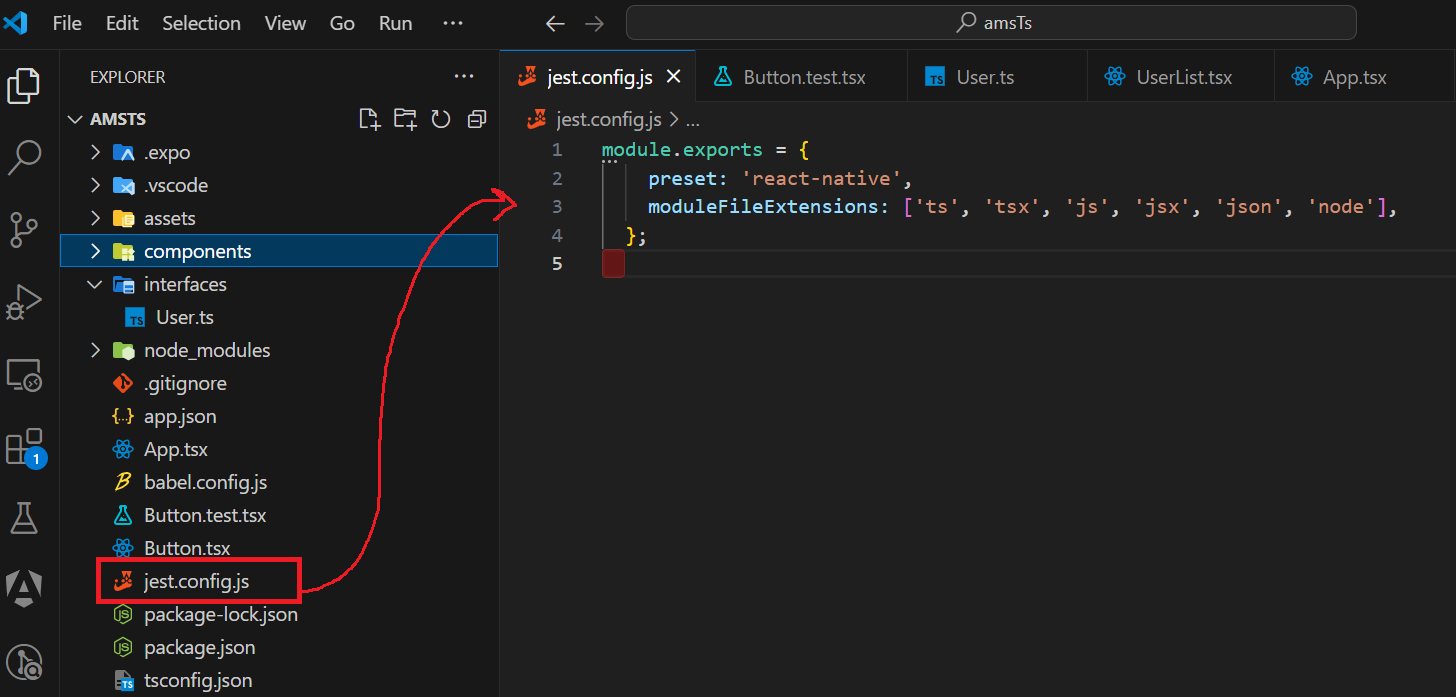
**};**

module.exports = {

    preset: 'react-native',

    moduleFileExtensions: ['ts', 'tsx', 'js', 'jsx', 'json', 'node'],

  };

****

Vous devez également définir un script **test** dans la section **scripts** de votre **package.json**.

Voici comment vous pouvez ajouter un script de test pour un projet typique React Native :

1. Ouvrez le fichier **package.json** à la racine de votre projet.
2. Localisez la section **scripts**. Elle devrait ressembler à quelque chose comme ceci
3. Ajoutez une ligne pour le script **test**. Si vous utilisez Jest pour les tests (ce qui est le standard dans les projets React Native), votre script pourrait ressembler à ceci :

  "scripts": {

    "start": "expo start",

    "android": "expo start --android",

    "ios": "expo start --ios",

    "web": "expo start --web",

    "test": "jest"

  },

### Étape 3 : Écrire un test unitaire

Imaginons que vous avez un composant **Button.tsx** dans votre application React Native :

// Button.tsx

import React from 'react';

import { TouchableOpacity, Text } from 'react-native';

interface ButtonProps {

  title: string;

  onPress: () => void;

}

const Button: React.FC<ButtonProps> = ({ title, onPress }) => (

  <TouchableOpacity onPress={onPress}>

    <Text>{title}</Text>

  </TouchableOpacity>

);

export default Button;

Vous pouvez créer un fichier de test, par exemple **Button.test.tsx**, pour écrire un test pour ce composant :

**// Button.test.tsx**

// Button.test.tsx

import React from 'react';

import { render, fireEvent } from '@testing-library/react-native';

import Button from './Button';

describe('Button', () => {

  it('should call onPress when pressed', () => {

    const onPressMock = jest.fn();

    const { getByText } = render(<Button title="Click me" onPress={onPressMock} />);

    fireEvent.press(getByText('Click me'));

    expect(onPressMock).toHaveBeenCalled();

  });

});

Dans cet exemple de test :

* Nous importons le composant **Button** et les fonctions nécessaires de **@testing-library/react-native**.
* Nous utilisons **describe** pour grouper les tests liés au composant **Button** et **it** pour définir un test individuel.
* **onPressMock** est une fonction jest mock pour simuler la fonction **onPress**.
* Nous utilisons **render** pour rendre le composant avec un titre et la fonction mock **onPress**.
* **fireEvent.press** simule un utilisateur appuyant sur le bouton.
* **expect(onPressMock).toHaveBeenCalled()** vérifie que la fonction **onPress** a été appelée lorsque le bouton est pressé.

**getByText** est une fonction de recherche fournie par la bibliothèque React Testing Library (et par extension, par React Native Testing Library). Elle est utilisée pour trouver des éléments dans le DOM (ou dans l'arbre de composants pour React Native) qui contiennent un texte spécifique. Cette fonction est très utile pour écrire des tests qui interagissent avec des composants en se basant sur le contenu textuel visible à l'utilisateur, ce qui aide à garantir que vos tests se concentrent sur le comportement de l'interface utilisateur.

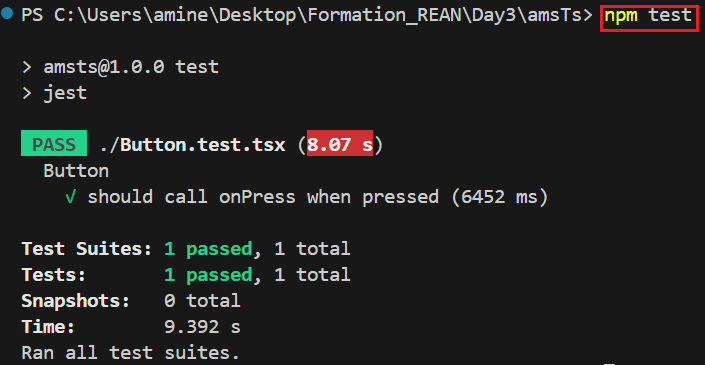
Voici comment **getByText** fonctionne :

* **Recherche par texte**: **getByText** recherche un nœud dans l'arbre de rendu qui contient le texte donné. Le texte doit correspondre exactement, mais la recherche est insensible à la casse et aux espaces blancs.
* **Utilisation dans les tests**: Elle est souvent utilisée dans les tests pour sélectionner des éléments spécifiques avec lesquels un utilisateur pourrait interagir, comme des boutons, des liens, ou des éléments d'affichage de texte. Si le texte est unique dans le rendu actuel, **getByText** fournira une référence directe à cet élément.

### Étape 4 : Exécuter le test

Exécutez vos tests en utilisant la commande :

npm test



Cette commande lancera Jest, qui exécutera tous les fichiers de test dans votre projet.

**Exercice (15 min)**

Créer un composant permettant d’afficher un compteur et deux boutons pour incrémenter et décrémenter sa valeur de +-1.

Créer un fichier de tests unitaires permettant de tester le bon fonctionnement de votre composant.

Correction :

**Composant (Counter.tsx):** C'est un composant simple de compteur.

import React, { useState } from 'react';

import { View, Text, Button } from 'react-native';

const Counter = () => {

  const [count, setCount] = useState(0);

  return (

    <View>

      <Text testID="counterValue">{count}</Text>

      <Button title="Incrémenter" onPress={() => setCount(count + 1)} />

      <Button title="Décrémenter" onPress={() => setCount(count - 1)} />

    </View>

  );

};

export default Counter;

**Test Unitaire (Counter.test.tsx):** Voici le test unitaire pour le composant **Counter** en utilisant TypeScript.

import React from 'react';

import { render, fireEvent } from '@testing-library/react-native';

import Counter from './Counter';

describe('Counter', () => {

  it('devrait afficher le compteur initial', () => {

    const { getByTestId } = render(<Counter />);

    const counterText = getByTestId('counterValue');

    expect(counterText.props.children).toBe(0);

  });

  it('devrait incrémenter le compteur', () => {

    const { getByTestId, getByText } = render(<Counter />);

    const incrementButton = getByText('Incrémenter');

    fireEvent.press(incrementButton);

    const counterText = getByTestId('counterValue');

    expect(counterText.props.children).toBe(1);

  });

  it('devrait décrémenter le compteur', () => {

    const { getByTestId, getByText } = render(<Counter />);

    const decrementButton = getByText('Décrémenter');

    fireEvent.press(decrementButton);

    const counterText = getByTestId('counterValue');

    expect(counterText.props.children).toBe(-1);

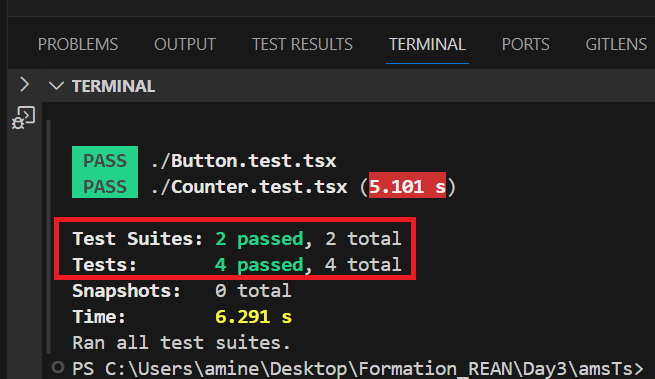
  });

});

Dans ce fichier de test, nous avons trois tests :

* Le premier test vérifie si le compteur initial est affiché correctement.
* Le deuxième test vérifie si le compteur est incrémenté lorsque le bouton d'incrémentation est pressé.
* Le troisième test vérifie si le compteur est décrémenté lorsque le bouton de décrémentation est pressé.

Pour exécuter ces tests, assurez-vous que votre projet React Native est configuré avec TypeScript, Jest et React Testing Library. Exécutez les tests en utilisant une commande comme **npm test** ou **yarn test**.



**3-Consommayion des API avec fetch**

Voici un exemple de comment utiliser **fetch** dans React Native pour récupérer des données depuis l'API JSONPlaceholder, un service en ligne gratuit qui fournit des données factices pour les tests et les prototypes.

1. **Composant FetchExample (FetchExample.js):** Ce composant effectue une requête **fetch** pour récupérer des publications (posts) depuis JSONPlaceholder et les affiche.

<https://jsonplaceholder.typicode.com/posts>



import React, { useState, useEffect } from 'react';

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

interface Post {

  userId: number;

  id: number;

  title: string;

  body: string;

}

const FetchExample = () => {

  const [posts, setPosts] = useState<Post[]>([]);

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

  useEffect(() => {

    fetch('https://jsonplaceholder.typicode.com/posts')

      .then(response => response.json())

      .then((data: Post[]) => {

        setPosts(data);

        setLoading(false);

      })

      .catch(error => {

        console.error('There was an error!', error);

        setLoading(false);

      });

  }, []);

  if (loading) {

    return <Text>Loading...</Text>;

  }

  return (

    <View style={styles.container}>

      <FlatList

        data={posts}

        keyExtractor={item => item.id.toString()}

        renderItem={({ item }) => (

          <View style={styles.post}>

            <Text style={styles.title}>{item.title}</Text>

            <Text>{item.body}</Text>

          </View>

        )}

      />

    </View>

  );

};

const styles = StyleSheet.create({

  container: {

    flex: 1,

    paddingTop: 20,

  },

  post: {

    backgroundColor: '#f9f9f9',

    padding: 15,

    borderBottomWidth: 1,

    borderBottomColor: '#eee',

  },

  title: {

    fontSize: 16,

    fontWeight: 'bold',

    marginBottom: 5,

  },

});

export default FetchExample;

**Composant App.tsx**

import { StatusBar } from 'expo-status-bar';

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

import FetchExample from './FetchExample';

export default function App() {

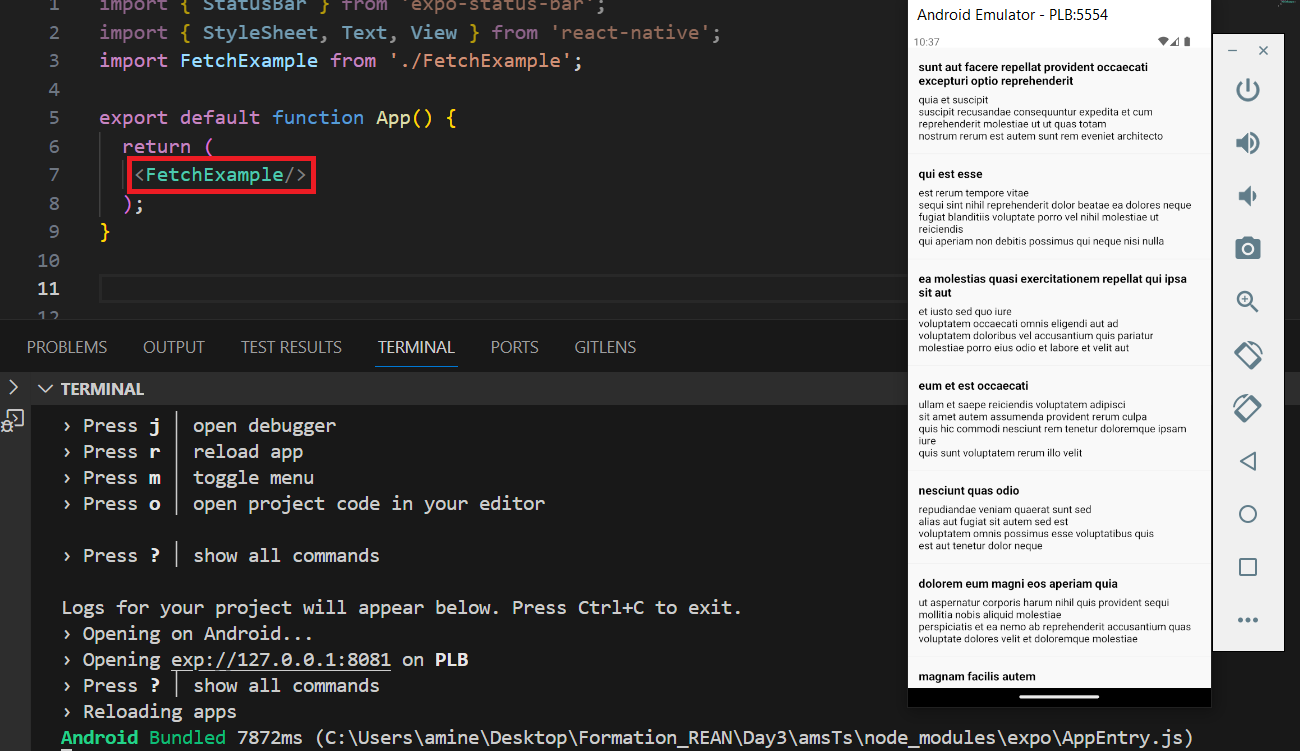
  return (

   <FetchExample/>

  );

}

**Voici le résultat**

****

Dans cet exemple TypeScript, nous avons défini une interface **Post** pour typifier les données récupérées, ce qui aide à assurer la sécurité des types dans l'application. Les autres parties du code restent assez similaires à l'exemple en JavaScript, avec l'ajout des annotations de type pour les états (**useState<Post[]>([])** et **useState<boolean>(true)**).

Ce composant peut être utilisé dans votre application React Native pour afficher une liste de posts récupérés depuis JSONPlaceholder.

**Exercice 2 (10 min) : Test unitaire pour le composant FetchExample**

Tester le composant **FetchExample** avec Jest, et React Testing Library, vous pouvez créer un fichier de test qui simule les appels réseau et vérifie que les données sont affichées correctement. Voici comment vous pourriez procéder :

Réponse : T**est Unitaire pour FetchExample (FetchExample.test.tsx):**

import React from 'react';

import { render, waitFor } from '@testing-library/react-native';

import FetchExample from './FetchExample';

// Définir le type pour les posts

interface Post {

  userId: number;

  id: number;

  title: string;

  body: string;

}

// Simuler la réponse de l'API JSONPlaceholder

const mockPosts: Post[] = [

  { userId: 1, id: 1, title: 'Post 1', body: 'Body 1' },

  { userId: 2, id: 2, title: 'Post 2', body: 'Body 2' }

];

// Typage de global.fetch

(global.fetch as jest.Mock) = jest.fn(() =>

  Promise.resolve({

    json: () => Promise.resolve(mockPosts)

  })

);

describe('FetchExample', () => {

  it('fetches and displays data', async () => {

    const { getByText } = render(<FetchExample />);

    await waitFor(() => {

      expect(getByText('Post 1')).toBeTruthy();

      expect(getByText('Body 1')).toBeTruthy();

      expect(getByText('Post 2')).toBeTruthy();

      expect(getByText('Body 2')).toBeTruthy();

    });

    // Vérifier que fetch a été appelé avec la bonne URL

    expect(fetch).toHaveBeenCalledWith('https://jsonplaceholder.typicode.com/posts');

  });

});

Dans ce code TypeScript :

* Nous définissons une interface **Post** pour typer les objets dans le tableau **mockPosts**.
* Nous utilisons **jest.Mock** pour typer le mock de **fetch**.
* L'annotation de type **(global.fetch as jest.Mock)** est utilisée pour indiquer que **fetch** sur l'objet global doit être considéré comme un mock de Jest.

Ces modifications assurent que le code est conforme aux exigences de typage de TypeScript, ce qui aide à détecter les erreurs à la compilation et améliore l'autocomplétion et la vérification du code dans les éditeurs de texte.

**Exercice 3 : (20 min) Consommation d’une API Users avec model**

[**https://jsonplaceholder.typicode.com/users/**](https://jsonplaceholder.typicode.com/users/)

****

**Correction**

### Étape 1: Définir l'interface pour les données

Tout d'abord, définissez le type de données que vous attendez de l'API. Imaginons que vous récupériez des utilisateurs d'une API, vous pourriez avoir une interface comme celle-ci :

// interfaces/User.ts

export interface User {

    id: number;

    name: string;

    email: string;

  }

### Étape 2: Créer le composant

Ensuite, créez un composant qui utilise **fetch** pour récupérer les données de l'API. Nous utiliserons les hooks **useState** et **useEffect** pour gérer l'état et le cycle de vie du composant.

// components/UserList.tsx

import React, { useState, useEffect } from 'react';

import { View, Text, ActivityIndicator } from 'react-native';

import { User } from '../interfaces/User';

const UserList: React.FC = () => {

  const [users, setUsers] = useState<User[]>([]);

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

  useEffect(() => {

    const fetchUsers = async () => {

      try {

        const response = await fetch('https://jsonplaceholder.typicode.com/users');

        const data = await response.json() as User[];

        setUsers(data);

        setLoading(false);

      } catch (error) {

        console.error(error);

        setLoading(false);

      }

    };

    fetchUsers();

  }, []);

  if (loading) {

    return <ActivityIndicator />;

  }

  return (

    <View>

      {users.map(user => (

        <Text key={user.id}>{user.name}</Text>

      ))}

    </View>

  );

};

export default UserList;

### Étape 3: Utiliser le composant dans votre application

Enfin, vous pouvez utiliser le composant **UserList** dans votre application :

import React from 'react';

import { SafeAreaView, StyleSheet } from 'react-native';

import UserList from './components/UserList';

const App: React.FC = () => {

  return (

    <SafeAreaView style={styles.container}>

      <UserList />

    </SafeAreaView>

  );

};

const styles = StyleSheet.create({

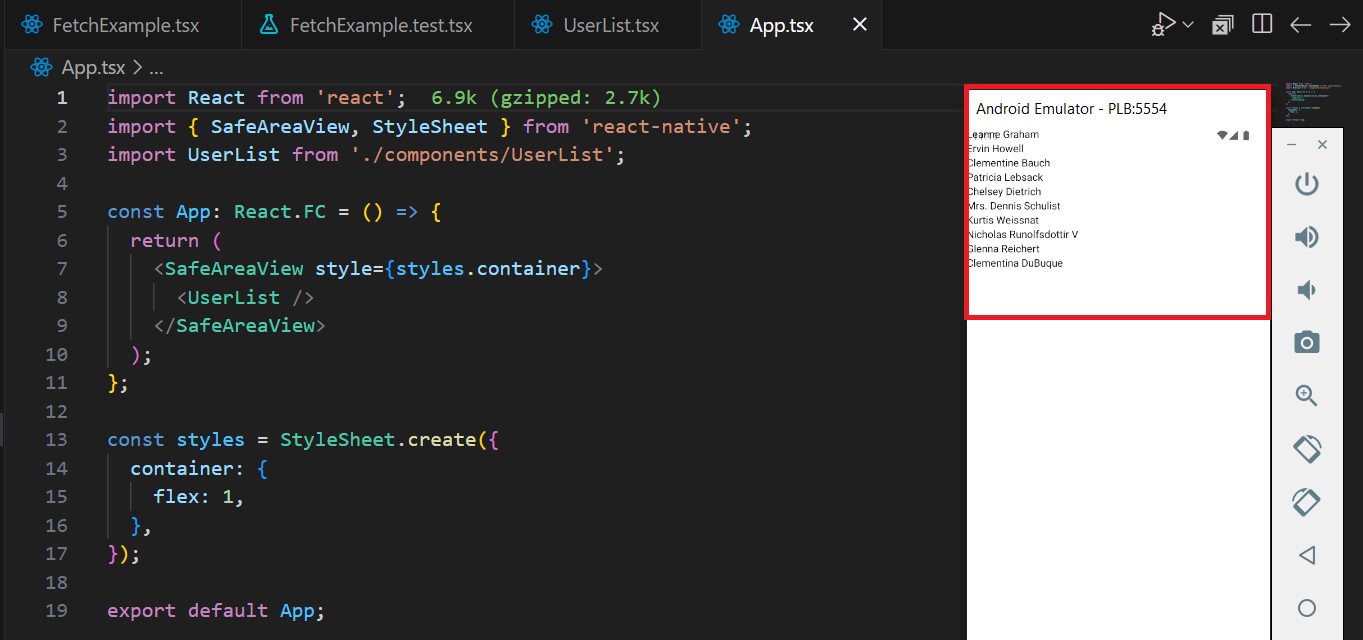
  container: {

    flex: 1,

  },

});

export default App;

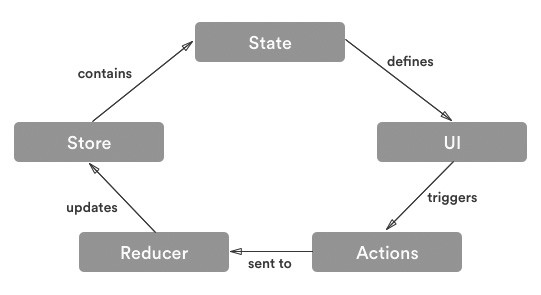
****

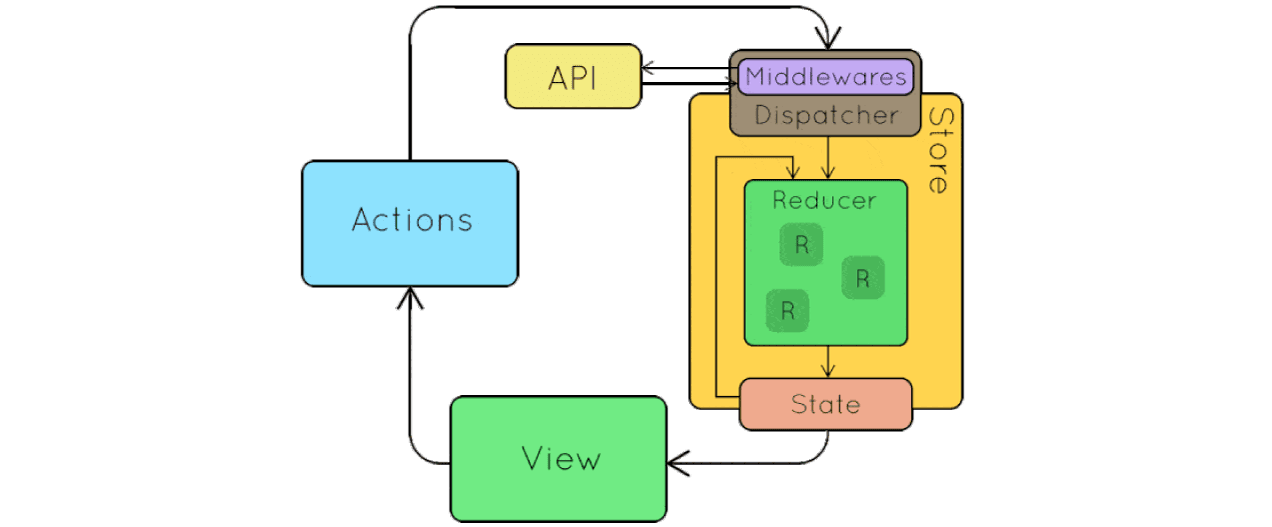
1. **Les States managers : Redux/Rematch**

Redux et Rematch sont des bibliothèques de gestion d'état pour applications JavaScript, telles que celles utilisant React ou React Native. Bien que Rematch soit construit sur Redux, il vise à simplifier et à améliorer l'expérience de développement.

1. **Redux**

Redux est une libraire javascript permettant de gérer un état global de notre application dans un store(cas d’un compteur global par exemple, plusieurs utilisateurs qui se connectent). Il se base principalement sur 3 Concepts : Actions, Reducers et le Store.





Redux doit être installé via npm.

Nous allons créer une simple application qui fait une mise à jour d’un compteur.

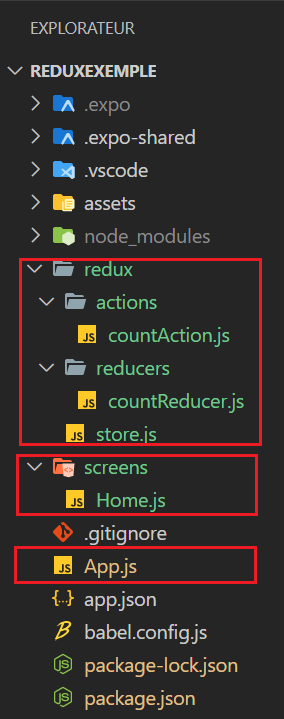
La première version sans Redux basé sur le Hook useState, la deuxième version fera appel à Redux et le store.

expo init compteurRedux

cd compteurRedux

npm start

La structure du projet sera comme suit :



App.js

import Home from './screens/Home';

export default function App() {

    return <Home />;

}

Home.js

// screens/Home.js

import React, { useState } from 'react';

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

export default function Home() {

  const [counter, setCounter] = useState(0);

  const handleIncreament = () => {

    setCounter(counter + 1);

  };

  const handleDecreament = () => {

    setCounter(counter - 1);

  };

  return (

    <View style={styles.container}>

      <Text style={styles.title\_text}>Counter App</Text>

      <Text style={styles.counter\_text}>{counter}</Text>

      <TouchableOpacity onPress={handleIncreament} style={styles.btn}>

        <Text style={styles.btn\_text}> Increment </Text>

      </TouchableOpacity>

      <TouchableOpacity

        onPress={handleDecreament}

        style={{ ...styles.btn, backgroundColor: '#6e3b3b' }}>

        <Text style={styles.btn\_text}> Decrement </Text>

      </TouchableOpacity>

   </View>

  );

}

const styles = StyleSheet.create({

  container: {

    flex: 1,

    backgroundColor: '#fff',

    alignItems: 'center',

    flexDirection: 'column',

    padding: 50,

  },

  title\_text: {

    fontSize: 40,

    fontWeight: '900',

    marginBottom: 55,

  },

  counter\_text: {

    fontSize: 35,

    fontWeight: '900',

    margin: 15,

  },

  btn: {

    backgroundColor: '#086972',

    padding: 10,

    margin: 10,

    borderRadius: 10,

  },

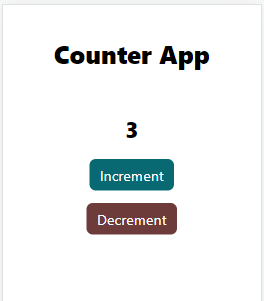
  btn\_text: {

    fontSize: 23,

    color: '#fff',

  },

});



Nous allons maintenant installer les dépendances Redux et créer les objets nécessaires.

npm install redux

npm install react-redux

countAction.js

// redux/actions/countAction.js

export const increment = () => {

    return {

      type: 'COUNT\_INCRESE',

    };

  };

  export const decrement = () => {

    return {

      type: 'COUNT\_DECRESE',

    };

  };

countReducer.js

// redux/reducers/countReducer.js

const initialState = {

    count: 0,

  };

  export default (state = initialState, action) => {

    switch (action.type) {

      case 'COUNT\_INCRESE':

        return {

          state,

          count: state.count + 1,

        };

      case 'COUNT\_DECRESE':

        return {

          state,

          count: state.count - 1,

        };

      default:

        return state;

    }

  };

Store.js

import { createStore, combineReducers} from 'redux';

import CountReducer from './reducers/countReducer';

const rootReducer = combineReducers({

  monCount: CountReducer,

});

export const store = createStore(rootReducer);

Maintenant nous devons connecter notre application React Native à Redux et le store crée.

Au niveau App.js on demande le chargement du store comme objet global.

import { Provider } from 'react-redux';

import Home from './screens/Home';

import { store } from './redux/store';

export default function App() {

  return (

    <Provider store={store}>

      <Home />

    </Provider>

  );

}

Ensuite au niveau Home.js, on fera appel au dispatcher d’actions

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

import { useSelector, useDispatch } from 'react-redux';

import { increment, decrement } from '../redux/actions/countAction';

export default function Home() {

  const dispatch = useDispatch();

  const count = useSelector((store) => store.monCount.count);

  const handleIncrement = () => {

    dispatch(increment());

  };

  const handleDecrement = () => {

    dispatch(decrement());

  };

  return (

    <View style={styles.container}>

      <Text style={styles.title\_text}>Counter App</Text>

      <Text style={styles.counter\_text}>{count}</Text>

      <TouchableOpacity onPress={handleIncrement} style={styles.btn}>

        <Text style={styles.btn\_text}> Increment </Text>

      </TouchableOpacity>

      <TouchableOpacity

        onPress={handleDecrement}

        style={{ ...styles.btn, backgroundColor: '#6e3b3b' }}>

        <Text style={styles.btn\_text}> Decrement </Text>

      </TouchableOpacity>

    </View>

  );

}

const styles = StyleSheet.create({

  container: {

    flex: 1,

    backgroundColor: '#fff',

    alignItems: 'center',

    flexDirection: 'column',

    padding: 50,

  },

  title\_text: {

    fontSize: 40,

    fontWeight: '900',

    marginBottom: 55,

  },

  counter\_text: {

    fontSize: 35,

    fontWeight: '900',

    margin: 15,

  },

  btn: {

    backgroundColor: '#086972',

    padding: 10,

    margin: 10,

    borderRadius: 10,

  },

  btn\_text: {

    fontSize: 23,

    color: '#fff',

  },

});

**B – Rematch**

Rematch vise à réduire redux en offrant une API simplifiée. Il encapsule les concepts de Redux en utilisant des modèles et des plugins, réduisant la nécessité de séparer les actions et les reducers. **Configuration minimale**: Il nécessite moins de configuration initiale par rapport à Redux. Par exemple, les actions et les reducers sont regroupés dans des "modèles" qui simplifient la gestion de l'état.

Voici un exemple plus simple d'utilisation de Rematch dans une application React Native avec TypeScript pour gérer un compteur. Ce sera une application minimaliste qui illustre uniquement la logique d'incrément et de décrément.

**1-Installer les dépendances:**

Vous aurez besoin de **@rematch/core** et **react-redux** pour utiliser Rematch.

npm install @rematch/core react-redux

**2- Configurer le store Rematch:**

Créez un fichier **store.ts** pour configurer le store de Rematch. Nous allons définir un modèle **counter** simple.

// store.ts

import { init } from '@rematch/core';

interface CounterState {

  value: number;

}

const counter = {

  state: { value: 0 }, // ensure the state has the correct shape

  reducers: {

    increment(state: CounterState) {

      return { value: state.value + 1 };

    },

    decrement(state: CounterState) {

      return { value: state.value - 1 };

    },

  },

};

const store = init({

  models: { counter },

});

export default store;

**3-Créer un composant d'affichage:**

Dans **App.tsx**, utilisez le **Provider** de **react-redux** pour englober l'application avec le store de Rematch. Ensuite, créez un composant simple pour montrer le compteur et les boutons pour incrémenter et décrémenter la valeur.

// App.tsx

import React from 'react';

import { View, Text, Button } from 'react-native';

import { Provider, useDispatch, useSelector } from 'react-redux';

import store from './store';

const CounterApp = () => {

  const count = useSelector((state: { counter: { value: number } }) => state.counter.value);

  const dispatch = useDispatch();

  return (

    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>

      <Text>Count: {count}</Text>

      <Button title="Increment" onPress={() => dispatch({ type: 'counter/increment' })} />

      <Button title="Decrement" onPress={() => dispatch({ type: 'counter/decrement' })} />

    </View>

  );

};

export default function App() {

  return (

    <Provider store={store}>

      <CounterApp />

    </Provider>

  );

}