Développement d'application Mobile avec React Native

Prérequis : JavaScript

Apprendre le JavaScript

Kouadio K. B. Clément G.
INGENIEUR DEVELOPPEUR

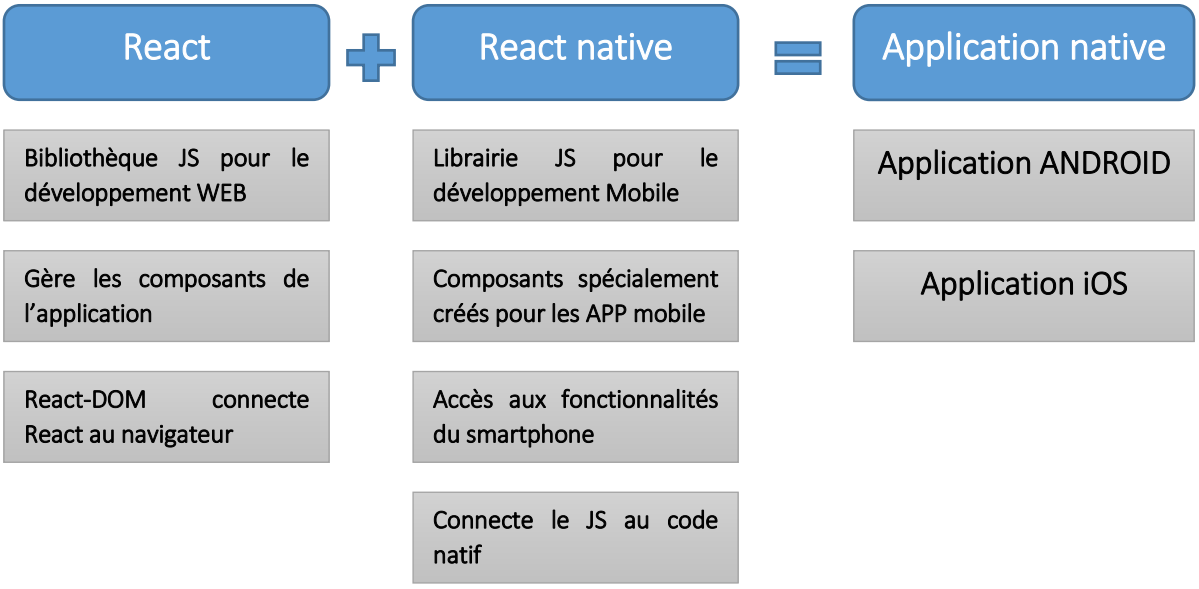
Sommaire

Chapitre 1 : Découverte du Framework React Native

1. Présentation de React Native

React Native est Framework d'application mobile multiplateforme créé par Facebook en 2015. Il est basé sur React, une bibliothèque JavaScript libre développée également par Facebook en 2011.

Avec React Native, l'on ne crée pas une « application Web mobile », une « application HTML5 » ou une « application hybride ». On construit une véritable application mobile « cross-platforms » qui ne se distingue pas d'une application créée avec Objective-C ou Java. React Native utilise les mêmes éléments fondamentaux que les applications iOS et Android classiques. Vous venez mettre ces blocs de construction ensemble en utilisant JavaScript et React.



2. Environnement de développement

Prérequis : Node 12 LTS ou version supérieure

2.1. Démarrage rapide : Expo

Sorti en 2015 sous le nom d'Exponent, Expo facilite grandement le développement d'applications avec React Native car il gère tout le processus de génération de l'application mobile (en développement comme pour la production), pour iOS et Android.

Pour initialiser une application, le seul élément à installer est expo-cli, paquet NPM à installer de manière globale et contenant les outils en ligne de commande d'Expo :

```
npm install -g expo-cli
```

Une fois le module installé, rendez-vous dans le répertoire où vous souhaitez créer votre application afin d'initialiser le projet :

```
expo init <nom de l'application>
```

NB : Il existe une version en ligne d'Expo pour développement d'applications. Voir https://snack.expo.dev/

2.2. Environnement pour la construction de projets avec du code natif : React Native Cli

Contrairement à l'environnement de démarrage rapide, les instructions pour créer du code natif dans votre projet sont un peu différentes selon votre système d'exploitation de développement et si vous souhaitez commencer à développer pour iOS ou Android. A titre d'exemple, nous ferons la configuration pour le développement Android sous Windows. Ci-dessous les étapes de la configuration :

Etape 1 : Installez Android Studio

Téléchargez et installez Android Studio. Dans l'assistant d'installation d'Android Studio, assurez-vous que les cases à côté de tous les éléments suivants sont cochées :

- Android SDK
- Android SDK Platform
- Android Virtual Device
- Si vous n'utilisez pas déjà Hyper-V : Performance (Intel ® HAXM)(Voir ici pour AMD ou Hyper-V)

Ensuite, cliquez sur "Suivant" pour installer tous ces composants.

Si les cases à cocher sont grisées, vous aurez la possibilité d'installer ces composants plus tard.

Une fois la configuration terminée et l'écran de bienvenue s'affiche, passez à l'étape suivante.

Etape 2 : Installez le SDK Android

Android Studio installe le dernier SDK Android par défaut. Cependant, la création d'une application React Native avec du code natif nécessite le SDK Android 10 (Q) en

particulier. Des SDK Android supplémentaires peuvent être installés via le gestionnaire de SDK dans Android Studio.

Pour ce faire, ouvrez Android Studio, cliquez sur le bouton "Configurer" et sélectionnez "SDK Manager".

Le gestionnaire de SDK se trouve également dans la boîte de dialogue "Préférences" d'Android Studio, sous Apparence et comportement → Paramètres système → SDK Android.

Sélectionnez l'onglet "Plateformes SDK" dans le gestionnaire de SDK, puis cochez la case "Afficher les détails du package" dans le coin inférieur droit. Recherchez et développez l'Android 10 (Q), puis assurez-vous que les éléments suivants sont cochés :

- Android SDK Platform 29
- Intel x86 Atom_64 System Image ou Google APIs Intel x86 Atom System Image

Ensuite, sélectionnez l'onglet "Outils SDK" et cochez la case "Afficher les détails du package" ici également. Recherchez et développez l'entrée "Android SDK Build-Tools", puis assurez-vous que le 29.0.2 est sélectionnée.

Enfin, cliquez sur « Appliquer » pour télécharger et installer le SDK Android et les outils de création associés.

Etape 3 : Configurez la variable d'environnement ANDROID_HOME

Les outils React Native nécessitent la configuration de certaines variables d'environnement afin de créer des applications avec du code natif.

1. Ouvrez le **Panneau de configuration de Windows.**
2. Cliquez sur **Comptes d'utilisateurs,** puis cliquez à nouveau sur **Comptes d'utilisateurs**
3. Cliquez sur **Modifier mes variables d'environnement**
4. Cliquez sur **Nouveau...** pour créer une nouvelle variable utilisateur **ANDROID_HOME** qui pointe vers le chemin d'accès à votre SDK Android :

Le SDK est installé, par défaut, à l'emplacement suivant :

```
%LOCALAPPDATA%\Android\Sdk
```

Vous pouvez trouver l'emplacement réel du SDK dans la boîte de dialogue "Paramètres" d'Android Studio, sous **Apparence et comportement** → **Paramètres système** → **SDK Android**.

Ouvrez une nouvelle fenêtre d'invite de commande pour vous assurer que la nouvelle variable d'environnement est chargée avant de passer à l'étape suivante.

1. Ouvrir powershell
2. Copiez et collez **Get-ChildItem -Path Env: ** dans powershell
3. Vérifier si **ANDROID_HOME** a été ajouté

Etape 4 : Ajouter des outils de plate-forme à Path

1. Ouvrez le **Panneau de configuration de Windows.**
2. Cliquez sur **Comptes d'utilisateurs,** puis cliquez à nouveau sur **Comptes d'utilisateurs**
3. Cliquez sur **Modifier mes variables d'environnement**
4. Sélectionnez la variable **Chemin**.
5. Cliquez sur **Modifier.**
6. Cliquez sur **Nouveau** et ajoutez le chemin d'accès à platform-tools à la liste.

L'emplacement par défaut de ce dossier est :

```
%LOCALAPPDATA%\Android\Sdk\platform-tools
```

Etape 5 : Création d'une nouvelle application

Pour initialiser une application, le paquet react-native-cli contenant les outils en ligne de commande de react-native doit être installé de manière globale. Afin d'accéder à la version stable actuelle du CLI au moment de l'exécution d'une commande, nous utiliserons npx qui est fourni avec Node.js.

Ainsi, la commande pour la création d'une nouvelle application est :

```
npx react-native init AwesomeProject
```

NB : Dans la suite du cours, nous utiliserons l'environnement de démarrage rapide avec Expo pour une exécution plus rapide et simple.

3. Structure d'une application React Native

- Création d'une application minimaliste :

```
expo init cours-react-native
```

La commande crée le répertoire cours-react-native et l'initialise avec les fichiers nécessaires pour lancer une application minimaliste avec Expo. Les fichiers principaux sont les suivants :

– package.json : il est initialisé avec les dépendances nécessaires et les commandes qui nous faciliteront le développement.

– app.json : contient des métadonnées de l'application pour Expo, notamment la version d'Expo à utiliser. À terme, il contiendra aussi des données nécessaires pour générer le binaire de l'application.

– App.js : le fichier principal de l'application.

Allons jeter un œil au fichier App.js afin de voir de quoi il retourne. Notez que depuis l'écriture de ce chapitre il se peut qu'Expo ait été mis à jour au point de changer légèrement le contenu des fichiers générés. Je n'ai cependant aucun mal à croire que les principes décrits ici resteront les mêmes.

Tout d'abord sont importées les bibliothèques React, et sans surprise quelques composants de React Native :

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
```

Rien de vraiment nouveau ici. Le composant principal de l'application, App se présente comme une classe héritant de Component comme nous en avons déjà vues.

Cette classe ne contient qu'une méthode render, renvoyant trois éléments (composant Text), intégrés dans une View :

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

```
      <Text>Open up App.js to start working on your app!</Text>
      <StatusBar style="auto" />
    </View>
  );
}
```

Cette View définit un attribut style, qui fait référence à la déclaration de feuille de style déclarée en dessous à l'aide de StyleSheet.create :

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Nous verrons un peu plus loin ce que représentent les composants View et Text ainsi que les feuilles de style.

4. Exécution d'une application React Native sur un smartphone/tablette

a. Application crée à partir du client Expo : expo-cli

Vous pouvez prévisualiser votre application sur un appareil en scannant le code QR avec l'application Expo disponible sur l'App Store et sur le Google Play.

Une fois l'application téléchargée, connectez votre ordinateur et le téléphone dans un même réseau local. Exécutez ensuite la commande pour afficher le métro dans le navigateur :

expo start ou npm start

Scannez enfin le code QR avec l'application Expo.

NB : Nous utiliserons Expo pour un développement plus facile et rapide.

b. Application crée à partir du client React Native : react-native-cli

Notons que l'exécution de l'application dans ce cas nécessite une configuration un peu plus complexe que dans le cas d'Expo. Tapez la commande ci-dessous pour installer et lancer votre application sur l'appareil :

npx react-native run-android

NB : Je vous invite à consulter Running On Device, la documentation complète et officielle de React Native.

Chapitre 2 : Les fondamentaux de la bibliothèque React

1. Introduction à JSX

a. Qu'est-ce que JSX :

JSX (JavaScript XML) est une extension syntaxique XML/HTML de JavaScript. Il nous permet de faciliter l'écriture et l'ajout de HTML dans React. JSX est basé sur ES6, et est traduit en JavaScript normal au moment de l'exécution.

b. Codage JSX :

JSX nous permet d'écrire des éléments HTML en JavaScript et de les placer dans le DOM sans aucune méthode `createElement()` et/ou `appendChild()`. JSX convertit les balises HTML en éléments React à l'aide de transcompilateur tel que Babel.

NB : Vous n'êtes pas obligé d'utiliser JSX, mais celui-ci facilite l'écriture d'applications React.

- Syntaxe avec JSX :

```
3
4   const firstElement = <h1>Hello world !</h1>
5
```

- Syntaxe avec JSX :

```
7
8   const firstElement = React.createElement('h1', {}, 'Hello world !');
9
```

c. Les Expressions dans JSX :

Avec JSX, vous pouvez écrire des expressions à l'intérieur d'accolades { }. L'expression peut être une variable React, une propriété ou toute autre expression JavaScript valide. JSX exécutera l'expression et renverra le résultat :

```
11
12   const myelement = <h1>React is {5 + 5} times better with JSX</h1>;
13
```

d. Insertion d'un grand bloc de code HTML :

Pour écrire du code HTML sur plusieurs lignes, placez le code HTML entre parenthèses :

```
const myelement = (
    <ul>
        <li>Apples</li>
        <li>Bananas</li>
        <li>Cherries</li>
    </ul>
);
```

e. Elément de niveau supérieur :

Le code HTML doit être encapsulé dans *UN* élément de niveau supérieur. Alors pour écrire *deux* paragraphes, il faudra mettre dans un élément parent, comme un élément div.

Exemple :

Encapsuler deux paragraphes dans un élément DIV :

```
const myelement = (
    <div>
        <p>I am a paragraph.</p>
        <p>I am a paragraph too.</p>
    </div>
);
```

NB : JSX génère une erreur si le code HTML n'est pas correct ou si le code HTML manque un élément parent.

f. Utilisation d'un Fragment comme élément de niveau supérieur :

Un Fragment permet d'encapsuler plusieurs lignes et cela évite d'ajouter des nœuds supplémentaires au DOM. Le Fragment se présente sous forme de balise HTML vide `<></>`.

Exemple :

Encapsuler deux paragraphes à l'intérieur d'un fragment :

```
const myelement = (
    <>
        <p>I am a paragraph.</p>
        <p>I am a paragraph too.</p>
    </>
);
```

g. Les éléments doivent être fermés :

JSX suit des règles XML et, par conséquent, les éléments HTML doivent être correctement fermés.

Exemples :

Fermez les éléments vides avec />

```
const myelement1 = <input type="text" />;
const myelement2 = <img src="" />;
```

NB : JSX génère une erreur si le code HTML n'est pas correctement fermé.

2. Composants et propriétés

Les composants sont des morceaux de code indépendants et réutilisables. Elles ont le même objectif que les fonctions JavaScript, mais fonctionnent de manière isolée et renvoient du HTML. Les composants sont de deux types, les composants de classe et les composants de fonction.

a. Composant de classe :

Un composant de classe est une classe héritant de la classe *Component* de *React.* Lors de la création d'un composant de classe, il est obligatoire d'implémenter la fonction *render()* retournant le code HTML à afficher.

Exemple :

Créer un composant Class appelé Etudiant

```
class Etudiant extends React.Component {
    render() {
        return <h2>Bonjour, je suis un(e) étudiant(e) !</h2>;
    }
}
```

b. Composant de fonction :

Il s'agit d'une fonction JavaScript qui renvoie du code HTML comme dans le cas d'un composant de classe. Cependant, le code d'un composant de fonction est plus simplifié en écriture que celui d'un composant de classe.

Exemple :

Créer le même composant Etudiant

```
function Etudiant() {
    return <h2>Bonjour, je suis un(e) étudiant(e) !</h2>;
}
```

c. Les propriétés d'un composant : props

Les accessoires ou propriétés sont des données transmises aux composants enfants en React. Ils permettent de transférer des données d'un composant parent à un composant enfant. Le composant enfant peut uniquement obtenir les propriétés transmises par le parent en utilisant **this.props.keyName** ou **props.keyName**. En utilisant des accessoires, on peut rendre son composant réutilisable.

NB : Les Props sont en lecture seule. Vous obtiendrez une erreur si vous essayez de modifier leur valeur.

Exemple :

Utilisez un attribut pour transmettre une couleur au composant ***Etudiant.***

- Cas d'un composant de fonction :

```
function Etudiant(props) {
    return <h2>Bonjour, je suis un(e) étudiant(e) de nationalité {props.nationalite} !</h2>;
}
ReactDOM.render(<Etudiant nationalite="Congolaise" />, document.getElementById('root'));
```

- Cas d'un composant de classe :

```
class Etudiant extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return <h2>Bonjour, je suis un(e) étudiant(e) de nationalité {this.props.nationalite} !</h2>;
    }
}
ReactDOM.render(<Etudiant nationalite="Congolaise" />, document.getElementById('root'));
```

Application 1 : Utilisation des props et affichage conditionnel

3. État d'un composant

Avant React 16.8, les composants de classe étaient le seul moyen de suivre l'état et le cycle de vie d'un composant React. Les composants fonctionnels ont été considérés comme "sans état". Avec l'ajout des Hooks, les composants de fonction sont désormais presque équivalents aux composants de classe.

a. État d'un composant de classe :

Les composants de classe ont un objet intégré nommé **state**. L'objet **state** est l'endroit où vous stockez les valeurs de propriété qui appartiennent au composant. Lorsque l'objet change, le composant effectue un nouveau rendu.

- Création et utilisation de l'objet state :

L'objet d'état est déclaré dans le constructeur.

```
4
5   class Etudiant extends React.Component {
6       constructor(props) {
7           super(props);
8           this.state = {
9               nom: "John",
10              moyenne: 13
11          };
12      }
13      render() {
14          return (
15              <div>
16                  <h1>{this.state.nom} a une moyenne de {this.state.moyenne}/20</h1>
17              </div>
18          );
19      }
20  }
```

- Changer l'objet state :

Pour modifier les valeurs dans l'objet d'état, vous pouvez utiliser **setState**, cela restituera le composant et tous ses composants enfants.

setState effectue une fusion superficielle entre l'état nouveau et précédent et déclenche un re-render du composant. La méthode prend soit un objet clé-valeur, soit une fonction qui retourne un objet clé-valeur.

Objet clé-valeur :

```
4   this.setState({ cle: valeur });
5
```

Fonction :

L'utilisation d'une fonction est utile pour mettre à jour une valeur basée sur l'état ou les props existants.

```
this.setState((previousState, currentProps) => {
    return { myInteger: previousState.myInteger + 1 }
});
```

NB : Vous pouvez également transmettre un rappel facultatif à setState qui sera déclenché lorsque le composant sera restitué avec le nouvel état.

```
this.setState({ myKey: 'myValue' }, () => {
    //Actions à effectuer après restitution du composant avec le nouvel état
});
```

Application 2 : Utilisation de l'état d'un composant de classe.

b. État d'un composant de fonction :

Avant React 16.8 les composants de fonction étaient considérés comme "sans état". Ceux-ci ont désormais accès à l'état et à d'autres fonctionnalités de React grâce aux hooks. Il existe 3 règles dans l'utilisation des hooks. En effet, ils ne peuvent :

- Être appelés qu'à l'intérieur des composants de fonction.

- Être appelés qu'au niveau supérieur d'un composant.

- Pas être conditionnel.

NB : Les hooks ne fonctionnent pas dans les composants de classe.

useState est le hook permettant de suivre l'état d'un composant de fonction. Pour l'utiliser, nous devons d'abord l'importer dans le composant comme suit :

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

- Initialisation du hook useState et lecture de l'état :

useState accepte un état initial et renvoie deux valeurs :

- L'état actuel.
- Une fonction qui met à jour l'état.

Exemple :

Initialiser l'état en haut du composant de fonction.

```
import { useState } from "react";
function Etudiant() {
    const [nom, setNom] = useState("John");

    return <h1>{nom} est un étudiant.</h1>
}
```

- Mise à jour de l'état :

Pour mettre à jour notre état, nous utilisons la fonction de mise à jour d'état.

NB : Ne jamais mettre à jour directement l'état. (Ex : `nom = "Kennedy"` n'est pas autorisé.)

Exemple :

Utilisons un bouton pour mettre à jour l'état :

```
import { useState } from "react";
function Etudiant() {
    const [nom, setNom] = useState("John");
    return (
        <>
            <h1>{nom} est un étudiant.</h1>
            <button
                type="button"
                onClick={() => setNom("Kennedy")}
            >Blue</button>
        </>
    )
}
```

NB :

- L'état peut contenir un objet :

```
const [etudiant, setEtudiant] = useState({
    nom: "John",
    moyenne: 13,
    age: 20
});
```

- Lorsque l'état est mis à jour, il est écrasé entièrement :

Si nous n'appelons que `setEtudiant({nom: "Kenedy"})`, cela supprimerait la moyenne et l'âge de notre état.

Nous pouvons utiliser l'opérateur de propagation JavaScript pour résoudre ce problème. L'opérateur nous permettra de mettre à jour uniquement le nom en conservant la moyenne et l'âge de notre état :

```
4
5   setEtudiant(previousState => {
6       return { ...previousState, nom: "Kennedy" }
7   });
8
```

Nous avons besoin de la valeur actuelle de l'état, nous passons donc une fonction dans notre fonction setEtudiant. Cette fonction reçoit la valeur précédente de l'état. Nous retournons ensuite un objet, en étalant le previousState et en écrasant uniquement le nom.

Application 3 : Suivre l'état d'un composant de fonction avec useState

4. Cycle de vie d'un composant

Chaque composant de React a un cycle de vie que vous pouvez surveiller et manipuler au cours de ses trois phases principales :

- Montage.
- Mise à jour.
- Démontage.

a. Composant de classe :

- Le montage :

Le montage signifie mettre des éléments dans le DOM. React dispose de quatre méthodes intégrées qui sont appelées, dans cet ordre, lors du montage d'un composant :

1. constructor() : Appelée avant toute autre chose lorsque le composant est initié, et c'est l'endroit naturel pour configurer les valeurs initiales et autres valeurs initiales.
2. getDerivedStateFromProps() : Appelée juste avant le rendu du ou des éléments dans le DOM. C'est l'endroit naturel pour définir l'objet en fonction de l'initiale.
3. render() : Méthode requise qui génère réellement le code HTML dans le DOM.

4. componentDidMount() : Appelée après le rendu du composant.

- Le mise à jour :

Un composant est mis à jour chaque fois qu'il y a une modification dans le composant.

React dispose de cinq méthodes intégrées qui sont appelées, dans cet ordre, lorsqu'un composant est mis à jour :

getDerivedStateFromProps() : Première méthode appelée lorsqu'un composant est mis à jour.

shouldComponentUpdate() : Méthode renvoyant une valeur booléenne qui spécifie si React doit poursuivre le rendu ou non.

render() : Appelée pour restituer le rendu.

getSnapshotBeforeUpdate() : Méthode dans accès aux valeurs du state et props avant la mise à jour. Si la méthode est présente, componentDidUpdate() doit etre appelée également, dans le cas contraire une erreur est générée.

componentDidUpdate() : Appelée après la mise à jour du composant dans le DOM.

- Le démontage :

La phase suivante du cycle de vie consiste à supprimer un composant du DOM. React n'a qu'une seule méthode intégrée qui est appelée lorsqu'un composant est démonté :

componentWillUnmount() : Appelée lorsque le composant est sur le point d'être supprimé du DOM.

Application 4 : Utilisation des méthodes de cycle de vie d'un composant de classe.

b. Composant de fonction :

Les composants de fonction utilisent le hook d'effet **useEffect** pour gérer le cycle de vie. Ce hook est une combinaison des méthodes **componentDidMount**, **componentDidUpdate** et **componentWillUnmount** utilisées dans les composants de classe.

useEffect permet d'effectuer des effets secondaires dans vos composants tels que la récupération de données, la mise à jour directe du DOM et minuterie…

useEffect accepte deux arguments, une fonction et un tableau facultatif de valeurs de dépendance : useEffect(<function>, <dependency>)

Exemple : Remplacer par une capture

Permet de compter 1 seconde après le rendu initial : setTimeout()

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom";

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

    useEffect(() => {
        setTimeout(() => {
            setCount((count) => count + 1);
        }, 1000);
    });

    return <h1>Je l'ai rendu {count} fois !</h1>;
}

ReactDOM.render(<Timer />, document.getElementById('root'));

```

NB :

- Si des valeurs de dépendance sont renseignées, l'effet s'exécute lorsque l'une des variables est mise à jour :

```
useEffect(() => {
    //Exécution au premier rendu
    //Et à chaque fois qu'une valeur de dépendance change
}, [val1, val2]);

```

- Nettoyage des effets :

Certains effets nécessitent un nettoyage pour réduire les fuites de mémoire. Les délais d'expiration, les abonnements, les écouteurs d'événements et les autres effets qui ne sont plus nécessaires doivent être éliminés (*Cela se fait dans la fonction* componentDidMount() pour les composants de classe).

Pour ce faire, nous incluons une fonction de retour à la fin du hook **useEffect** :

```
4   useEffect(() => {
5       let timer = setTimeout(() => {
6           setCount((count) => count + 1);
7       }, 1000);
8
9       return () => clearTimeout(timer)
10  }, [count]);
11
```

Dans cet exemple, la fonction de retour permet de nettoyer la minuterie nommée *timer*.

Application 5 : Utilisation du Hook d'effet (useEffect) dans un composant de fonction.

Chapitre 3 : L'essentiel de React Native

A la différence des Framework cross-platforms hybride, React Native ne produit pas d'application web mobile car il utilise les composants mobiles natifs. En effet, lorsque vous définissez par exemple une vue en React Native, sur iOS votre application affiche une **UIView** et sur Android une **android.view.View**, deux composants natifs. Le fonctionnement est le même pour tous les types d'éléments graphiques : boutons, textes, listes, chargement, etc. React Native convertit tous vos éléments en leur équivalent natif. C'est grâce à cette fonctionnalité que vos applications seront plus performantes, plus fluides et plus ressemblantes à une application mobile

1. Composants de base et APIs

React Native dispose de nombreux composants de base pour tout, des contrôles de formulaire aux indicateurs d'activité. Le tableau ci-dessous présente une liste non exhaustive des composants de base principalement utilisés :

COMPOSANT REACT NATIF	VUE ANDROID	VUE IOS	VUE WEB	DESCRIPTION
<View>	<ViewGroup>	<UIView>		Conteneur sans défilement
<Text>	<TextView>	<UITextView>	<p>	Afficheur de paragraphe de texte
<Image>	<ImageView>	<UIImageView>	<img>	Afficheur d'images
<ScrollView>	<ScrollView>	<UIScrollView>		Conteneur de défilement
<TextInput>	<EditText>	<UITextField>	<input type="text">	Zone de saisie de texte

En plus des composants de base, React Native propose quelques API pour les systèmes cibles. Par exemple l'API Android **BackHandler** permettant de détecter les pressions sur les buttons pour la navigation arrière. Pour les iOS, nous pouvons citer l'API **ActionSheetIOS** servant à afficher une feuille d'action iOS ou une feuille de partage.

NB :

- Vous trouvez tous les composants de base et API de React Native à cette adresse : documentés dans la section API.

- Pour vos recherches de bibliothèques faisant quelque chose de spécifique, veuillez-vous référer à la communauté des développeurs React Native.

2. Style des composants avec React Native

Les styles sont définis dans un objet JSON avec des noms d'attributs de style similaires, comme dans CSS. Un tel objet peut soit être mis en ligne dans la propriété **style** d'un composant, soit être transmis à la fonction **StyleSheet.create(StyleObject)** et être stocké dans une variable pour un accès en ligne plus court en utilisant un nom de sélecteur.

Syntaxe :

- <Component style={styleFromStyleSheet} />
- <Component style={{'cle' : 'valeur' }} />
- <Component style={[styleFromStyleSheet1, styleFromStyleSheet2]} />

Remarques :

La plupart des styles React Native ont leur équivalent CSS, mais dans un camelCase. Ainsi, la propriété **text-decoration** devient **textDecoration**. Contrairement aux CSS, les styles ne sont pas hérités. Si vous souhaitez que les composants enfants héritent d'un certain style, vous devez le fournir explicitement à l'enfant. Cela signifie que vous ne pouvez pas définir une famille de polices pour une View entière.

La seule exception à cette règle est le composant Text : les Text imbriqués héritent de leurs styles parents.

a. Style utilisant des styles en ligne

Chaque composant React Native peut prendre une propriété **style**. Vous pouvez lui transmettre un objet JavaScript avec des propriétés de style CSS :

```
4
5   <Text style={{ color: 'red' }}>Texte rouge</Text>
6
```

Cela peut être inefficace car il doit recréer l'objet chaque fois que le composant est rendu. Utiliser une feuille de style est préférable.

b. Style à l'aide d'une feuille de style

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


class Example extends Component {
    render() {
        return (
            <View>
                <Text style={styles.red}>Red</Text>
                <Text style={styles.big}>Big</Text>
            </View>
        );
    }
}

const styles = StyleSheet.create({
    red: {
        color: 'red'
    },
    big: {
        fontSize: 30
    }
});
```

StyleSheet.create() renvoie un objet dont les valeurs sont des nombres. React Native sait convertir ces identifiants numériques en objet de style correct.

c. Ajouter plusieurs styles à un composant

Vous pouvez passer un tableau à la propriété style pour appliquer plusieurs styles. En cas de conflit, le dernier de la liste est prioritaire.

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

class Example extends Component {
    render() {
        return (
            <View>
                <Text style={[styles.red, styles.big]}>Big red</Text>
                <Text style={[styles.red, styles.greenUnderline]}>Green underline</Text>
                <Text style={[styles.greenUnderline, styles.red]}>Red underline</Text>
                <Text style={[styles.greenUnderline, styles.red, styles.big]}>Big red
                    underline</Text>
                <Text style={[styles.big, { color: 'yellow' }]}>Big yellow</Text>
            </View>
        );
    }
}

const styles = StyleSheet.create({
    red: {
        color: 'red'
    },
    greenUnderline: {
        color: 'green',
        textDecoration: 'underline'
    },
    big: {
        fontSize: 30
    }
});

```

d. Style conditionnel

```

    <View style={[(this.props.isTrue) ? styles.bgcolorBlack : styles.bgColorWhite]} />

```

Si la valeur de **isTrue** est true, la couleur de fond est noire sinon sera blanche.

Application 6 : Composants de base et style

3. Mise en page avec Flexbox

Flexbox est un mode de mise en page permettant la disposition des éléments sur une page de manière à ce que les éléments se comportent de manière prévisible lorsque la mise en page doit prendre en charge différentes tailles d'écran et différents périphériques d'affichage. Par défaut, flexbox organise les enfants dans une colonne. Mais vous pouvez le changer en ligne en utilisant flexDirection : 'row'.

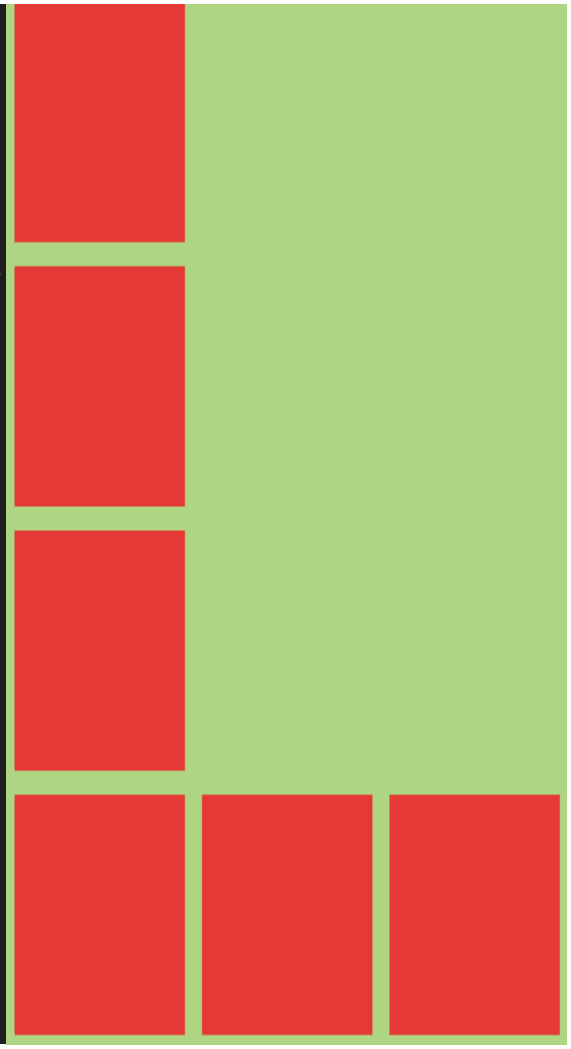
Pour obtenir une complète et bonne mise en page, nous utilisons la combinaison de **flex, flexDirection, alignItems et justifyContent**.

a. Style de direction : flex et flexDirection

flexDirection contrôle la direction dans laquelle les enfants d'un nœud (axe principal) sont disposés. Quant à **flex,** il définit comment les éléments vont « **remplir** » l'espace disponible le long de l'axe principal. L'espace sera divisé en fonction de la propriété **flex** de chaque élément.

```
const Direction = (props) => {
    return (
        <View style={styles.container}>
            <View style={styles.box} />
            <View style={styles.box} />
            <View style={styles.box} />
            <View style={{ flexDirection: 'row' }}>
                <View style={styles.box} />
                <View style={styles.box} />
                <View style={styles.box} />
            </View>
        </View>
    )
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: '#AED581',
    },
    box: {
        with: 50,
        height: 50,
        backgroundColor: 'red'
    }
});
```



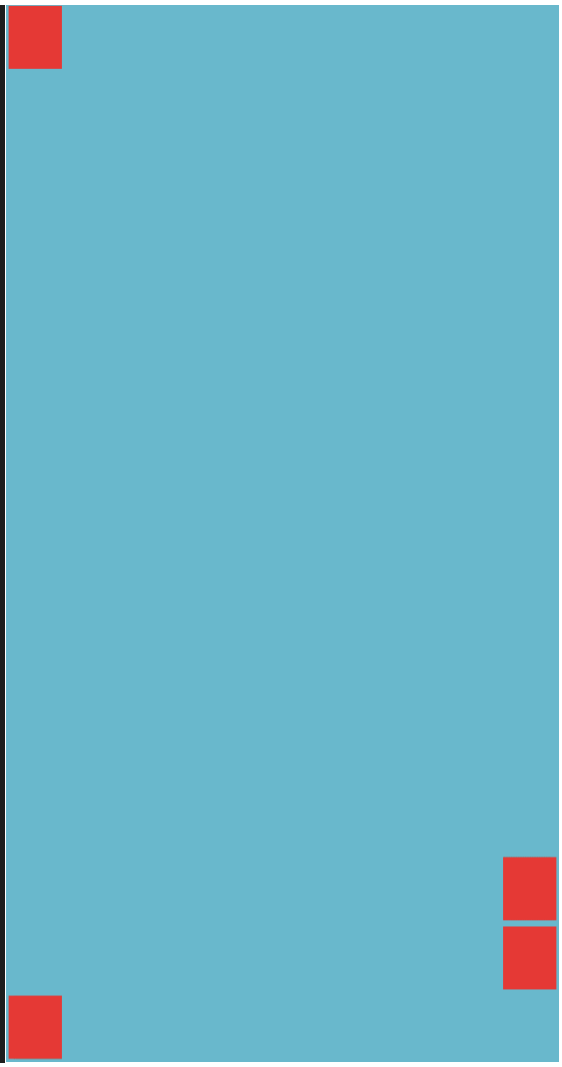
b. Style d'alignement : alignItems et justifyContent

justifyContent décrit comment aligner les enfants dans l'axe principal de leur conteneur. Quant à **alignItems**, il décrit comment aligner les enfants le long de l'axe transversal de

leur conteneur. Il est très similaire à **justifyContent** mais au lieu de s'appliquer à l'axe principal, **alignItems** s'applique à l'axe transversal.

```
const AlignmentAxis = (props) => {
    return (
        <View style={styles.container}>
            <View style={styles.box} />
            <View style={{
                flex: 1, alignItems: 'flex-end',
                justifyContent: 'flex-end'
            }}>
                <View style={styles.box} />
                <View style={styles.box} />
            </View>
            <View style={styles.box} />
        </View>
    )
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: `#69B8CC`,
    },
    box: {
        with: 50,
        height: 50,
        backgroundColor: 'red'
    }
});
```



NB : Consulter Layout with Flexbox pour une documentation complète sur la mise en page avec Flexbox.

Application 7 : Mise en page d'un composant

4. Navigation entre les vues

La navigation représente le passage d'une vue à une autre. C'est un point essentiel dans les applications mobiles. Une navigation bien gérée, cohérente, rend votre application intuitive et fluide. Nous utilisons la librairie React Navigation proposée par la communauté React Native pour gérer la navigation.

Cette librairie est devenue un incontournable du développement en React Native, si bien qu'aujourd'hui elle dispose de son propre site avec sa propre documentation. On y apprend comment utiliser les différents types de navigation :

- StackNavigator : c'est la navigation la plus basique où on pousse une vue sur iOS et présente une vue sur Android. Le *StackNavigator* gère une pile de vues qui augmente lorsque vous naviguez vers une nouvelle vue et diminue lorsque vous revenez en arrière. C'est cette navigation que l'on va utiliser dans ce chapitre.
- TabNavigator : permet de créer une barre d'onglets, en haut ou en bas, de votre application. On utilisera ce type de navigation un peu plus loin dans ce cours, lorsque l'on voudra couper notre application en plusieurs onglets.
- DrawerNavigator : permet de créer un menu dit "hamburger", à gauche de nos vues, avec une liste d'entrées pour chacune de nos vues.

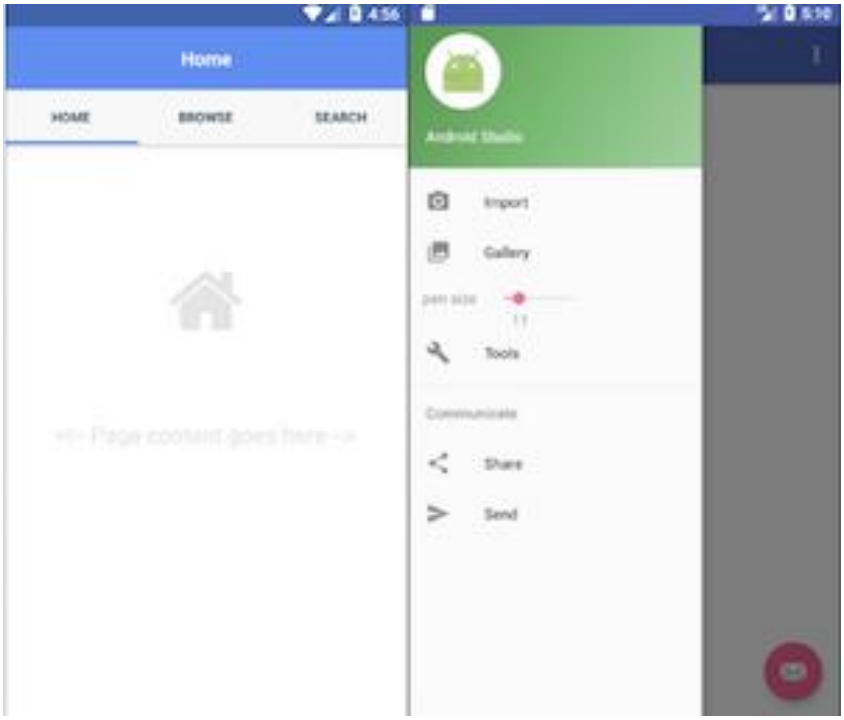


Figure 1 : À gauche, un TabNavigator / À droite, un DrawerNavigator

a. Installation de React Navigation et des dépendances

A présent, ajoutons la librairie React Navigation à notre projet :

```
npm install @react-navigation/native
```

Installation des dépendances dans un projet géré Expo :

```
expo install react-native-screens react-native-safe-area-context
```

Installation des dépendances dans un projet natif :

```
npm install react-native-screens react-native-safe-area-context
```

L'ajout de la navigation react dans un projet se fait principalement en 3 étapes :

- Définir les écrans dans notre application
- Créer un navigateur : createStackNavigator, createBottomTabNavigator, createDrawerNavigator…
- Configurer un <NavigationContainer> : est un composant qui gère notre arborescence de navigation et contient l' état de navigation .

b. Définir les écrans

Nous créons un composant pour chaque écran. Les écrans sont des composants React réguliers. Ils recevront des accessoires spécifiques à la navigation lorsqu'ils seront instanciés. Créons deux composants :

```jsx
import * as React from 'react';
import { View, Text } from 'react-native';

export const Home = ({ navigation, route }) => {
    return (
        <View>
            <Text>Home page</Text>
        </View>
    );
}
```

```jsx
import * as React from 'react';
import { View, Text } from 'react-native';

export const Screen1 = ({ navigation, route }) => {
    return (
        <View>
            <Text>Screen1</Text>
        </View>
    );
}
```

c. Créer un navigateur

Il s'agit de créer un type de navigation pour notre application. Nous choisissons d'abord l'un des navigateurs disponibles, par exemple Stack, qui servira de navigateur racine. Les navigateurs peuvent être imbriqués plus tard.

Chaque composant définit un itinéraire dans notre application. Si nous voulons des navigateurs imbriqués, par exemple un navigateur d'onglets dans un navigateur de pile, nous pouvons utiliser un autre navigateur comme un écran.

```
import { createStackNavigator } from '@react-navigation/stack'
import { Home } from '../components/home';
import { Screen1 } from '../components/screen1';
import { Screen2 } from '../components/screen2';

const Stack = createStackNavigator();

export const MyNavigation = () => {
    return (
        <Stack.Navigator initialRouteName="Home">
            <Stack.Screen name="Home" component={Home} />
            <Stack.Screen name="Screen1" component={Screen1} />
            <Stack.Screen name="Screen2" component={Screen2} />
        </Stack.Navigator>
    );
}
```

d. Configurer le NavigationContainer

Ce composant enveloppe toute la structure des navigateurs. Nous n'avons besoin que d'un seul NavigationContainer, même si nous avons des navigateurs imbriqués.

```
import { NavigationContainer } from '@react-navigation/native';
import React from 'react';
import { MyNavigation } from './navigation/navigation';

export default function App() {
  return (
    <NavigationContainer>
      <MyNavigation />
    </NavigationContainer>
  );
}
```

e. Navigation et itinéraires

Chaque navigateur prend en charge différentes manières de naviguer :

- Stack : push
- Tabs : navigate
- Drawer (Menu latéral) : openDrawer

NB : Lors de la navigation, nous spécifions généralement un nom d'écran et éventuellement des paramètres, par exemple :

```
navigator.push("Screen2", { paramA: "Hello!" })
```

f. Utilisation des Hooks de navigations

Pour les composants qui ne sont pas des écrans (descendants directs d'un navigateur), on peut accéder aux objets navigation et route à l'aide des Hooks.

```
import * as React from 'react';
import { View, Text } from 'react-native';

export const Screen = () => {
    const navigation = useNavigation();
    const route = useRoute();

    return (
        <View>
            <Text>Screen</Text>
        </View>
    );
}
```

Application 8 : La navigation en React Native

Table des matières