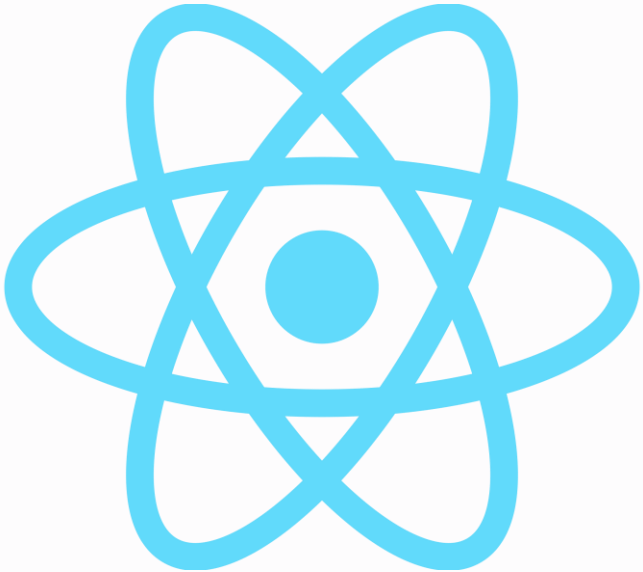
React Native



Support réalisé par Pierrick Hauguel

TABLE DES MATIÈRES

TABLE DES MATIÈRES

TABLE DES MATIÈRES

01
Introduction et Configuration

Qu'est-ce que React Native ?

React Native est un framework de développement mobile open-source créé par Facebook en 2015. Il permet aux développeurs de construire des applications mobiles performantes pour iOS et Android en utilisant le langage JavaScript et React.

Avec React Native, le code est partagé entre les plateformes, ce qui accélère le développement et réduit les coûts. Des entreprises comme Instagram, Airbnb, et Tesla utilisent React Native pour offrir une expérience utilisateur de haute qualité.

Découverte d'Expo

Expo est un ensemble d'outils et de services conçus pour améliorer le développement avec React Native. Il simplifie l'accès aux fonctionnalités natives des téléphones, comme la caméra et le système de notification, sans avoir à écrire de code natif. Expo permet également le développement rapide grâce à l'Expo Go app, où les développeurs peuvent tester leurs applications en temps réel. Contrairement à React Native CLI, Expo offre une expérience plus intégrée et accessible, idéale pour les nouveaux développeurs.

Configuration de l'environnement de développement

Pour démarrer avec React Native et Expo, vous devez configurer votre environnement de développement. Les étapes incluent :

Installer Node.js sur votre système.
Utiliser npm pour installer Expo CLI globalement avec npm install -g expo-cli.
Choisir un IDE, comme Visual Studio Code, pour écrire votre code. VSC offre des fonctionnalités comme l'achèvement du code, le débogage, et l'intégration Git.

Création du premier projet avec Expo

Pour commencer un nouveau projet React Native avec Expo, suivez ces étapes :

Ouvrez un terminal et exécutez expo init MonPremierProjet pour créer un nouveau projet.
Sélectionnez un template lorsque vous y êtes invité.
Une fois le projet initialisé, naviguez dans votre nouveau dossier de projet et lancez le serveur de développement avec expo start.
Ouvrez l'application Expo Go sur votre téléphone et scannez le QR code affiché dans le terminal pour voir votre application s'exécuter en temps réel.

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

export default function App() {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Bienvenue dans votre première application React Native!</Text>
    </View>
  );
}
```

02

Les composants de base

Le Composant View

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

const App = () => (
  <View>
    <Text>Hello, world!</Text>
  </View>
);
```

View est le conteneur de base dans React Native, équivalent à div dans le développement web. Il est utilisé pour envelopper et structurer les éléments d'interface utilisateur.

Composant Text

Text est utilisé pour afficher du texte. Il peut être stylisé à l'aide de StyleSheet et supporte la mise en forme à travers des composants Text imbriqués.

```
import { Text } from 'react-native';

const MyText = () => <Text>Hello, world!</Text>;
```

Le Composant Image

Image permet d'afficher des images. Il peut charger des images depuis un URL ou des ressources locales.

```
import { Image } from 'react-native';

const MyImage = () => (
  <Image
    source={{uri: 'https://example.com/image.png'}}
    style={{width: 200, height: 200}}
  />
);
```

ScrollView: Permettre le défilement de contenu long

```
import { ScrollView, Text } from 'react-native';

const MyScrollView = () => (
  <ScrollView>
    <Text>Item 1</Text>
    <Text>Item 2</Text>
    <Text>Item 3</Text>
    // Ajoutez plus d'éléments ici
  </ScrollView>
);
```

- Le composant ScrollView permet de créer des vues défilantes pour du contenu long.
- Il est optimisé pour les performances et prend en charge la virtualisation de la liste pour les grandes quantités de données.
- Vous pouvez personnaliser l'orientation du défilement, les indicateurs de défilement et le comportement de rebond.

TextInput: Permettre aux utilisateurs de saisir du texte

```
import { TextInput } from 'react-native';

const MyTextInput = () => (
  <TextInput
    style={{height: 40, borderColor: 'gray', borderWidth: 1}}
    defaultValue="You can type in me"
  />
);
```

- Le composant TextInput permet aux utilisateurs de saisir du texte dans votre application.
- Vous pouvez personnaliser le type de clavier, la validation du texte et le style de l'entrée.
- Il prend en charge les événements tels que la modification du texte et la soumission du formulaire.

FlatList: Affichage performant de listes de données

- Le composant FlatList est un composant de liste optimisé pour les performances.
- Il est idéal pour afficher de grandes listes de données de manière fluide et efficace.
- Vous pouvez personnaliser l'apparence des éléments de la liste et la logique de rendu.

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

const MyFlatList = () => (
  <FlatList
    data={[{key: 'a'}, {key: 'b'}]}
    renderItem={({item}) => <Text>{item.key}</Text>}
  />
);
```

SectionList: Affichage de listes avec des sections groupées

- Le composant SectionList est similaire à FlatList, mais permet de créer des listes avec des sections groupées.
- Il est idéal pour afficher des données organisées en catégories ou en sections.
- Vous pouvez personnaliser l'apparence des sections et des éléments de la liste.

```
import { SectionList, Text } from 'react-native';

const MySectionList = () => (
  <SectionList
    sections={[
      {title: 'D', data: ['Devin']},
      {title: 'J', data: ['Jackson']},
      // Ajoutez plus de sections ici
    ]}
    renderItem={({item}) => <Text>{item}</Text>}
    renderSectionHeader={({section}) => <Text style={{fontWeight: 'bold'}}>
  />
);
```

SafeAreaView: Ajuster le contenu pour éviter les interruptions de l'interface

- Le composant SafeAreaView ajuste automatiquement son enfant pour éviter le chevauchement avec les zones non sécurisées de l'écran, telles que les encoches ou les bords arrondis.
- C'est crucial pour garantir une expérience utilisateur cohérente sur différents appareils.

```
import { SafeAreaView, Text } from 'react-native';

const MySafeAreaView = () => (
  <SafeAreaView>
    <Text>This is safe!</Text>
  </SafeAreaView>
);
```

Button: Un bouton simple et personnalisable

```
import { Button, Alert } from 'react-native';

const MyButton = () => (
  <Button
    title="Press me"
    onPress={() => Alert.alert('Button Pressed!')}
  />
);
```

- Le composant Button permet d'implémenter des boutons d'action dans votre application.
- Il offre un titre personnalisable et déclenche une fonction lorsqu'il est pressé.
- Vous pouvez également styliser son apparence.

TouchableOpacity: Rendre n'importe quel composant interactif

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

const MyTouchableOpacity = () => (
  <TouchableOpacity onPress={() => console.log('Pressed!')}>
    <Text>Press Me</Text>
  </TouchableOpacity>
);
```

Le composant TouchableOpacity permet de transformer n'importe quel composant en un élément interactif qui réagit au toucher.
Il fournit un effet visuel subtil lors de la pression (diminution de l'opacité).
Vous pouvez l'utiliser pour créer des zones tactiles personnalisées.

TouchableHighlight: Feedback visuel sur le toucher

- Le composant TouchableHighlight est similaire à TouchableOpacity, mais il applique une légère surbrillance visuelle lors du toucher.
- Cela offre un feedback visuel plus important à l'utilisateur.

```
import { TouchableHighlight, Text } from 'react-native';

const MyTouchableHighlight = () => (
  <TouchableHighlight onPress={() => console.log('Pressed!')}>
    <Text>Press Me</Text>
  </TouchableHighlight>
);
```

TouchableWithoutFeedback: Capturer les interactions sans feedback visuel

- Le composant TouchableWithoutFeedback permet de capturer les interactions tactiles sans fournir de feedback visuel.
- Il est utile pour des situations où vous ne voulez pas d'effet visuel sur le toucher, mais que vous avez besoin de détecter l'interaction.

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

const MyTouchableWithoutFeedback = () => (
  <TouchableWithoutFeedback onPress={() => console.log('Pressed!')}>
    <View><Text>Press Me</Text></View>
  </TouchableWithoutFeedback>
);
```

Switch: Basculer entre deux états

```jsx
import { Switch, View } from 'react-native';
import React, { useState } from 'react';

const MySwitch = () => {
  const [isEnabled, setIsEnabled] = useState(false);
  return (
    <View>
      <Switch
        trackColor={{ false: "#767577", true: "#81b0ff" }}
        thumbColor={isEnabled ? "#f5dd4b" : "#f4f3f4"}
        ios_backgroundColor="#3e3e3e"
        onValueChange={() => setIsEnabled(previousState => !previousState)}
        value={isEnabled}
      />
    </View>
  );
};
```

Le composant Switch permet aux utilisateurs de basculer entre deux états (activé/désactivé).
Il est souvent utilisé pour des options de configuration simples.

ActivityIndicator: Indicateur de chargement

- Le composant ActivityIndicator permet d'afficher une roue de chargement pour indiquer une opération en cours à l'utilisateur.
- Il fournit un feedback visuel pendant les temps d'attente.

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

const MyActivityIndicator = () => (
  <View>
    <ActivityIndicator size="large" color="#0000ff" />
  </View>
);
```

```jsx
import { Modal, View, Text, Button } from 'react-native';
import React, { useState } from 'react';

const MyModal = () => {
  const [modalVisible, setModalVisible] = useState(false);
  return (
    <View>
      <Modal
        animationType="slide"
        transparent={false}
        visible={modalVisible}
        onRequestClose={() => {
          Alert.alert('Modal has been closed.');
        }}>
        <View>
          <Text>Hello World!</Text>
          <Button
            title="Hide Modal"
            onPress={() => setModalVisible(!modalVisible)}
          />
        </View>
      </Modal>
      <Button
        title="Show Modal"
        onPress={() => setModalVisible(true)}
      />
    </View>
  );
};
```

Le composant Modal

- Le composant Modal permet d'afficher du contenu au-dessus des autres vues de votre application.
- Il est souvent utilisé pour des fenêtres contextuelles, des boîtes de dialogue, des menus contextuels et des fenêtres de confirmation.
- Vous pouvez personnaliser son animation d'affichage et de fermeture.

Pressable

- Le composant Pressable est une abstraction unifiée permettant de gérer les interactions tactiles sur des éléments.
- Il prend en charge différents types de pression (appuyer une fois, appuyer et maintenir, etc.) et offre une API cohérente pour gérer les événements.
- C'est la solution recommandée pour la gestion des interactions tactiles dans les nouvelles applications.

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

const MyPressable = () => (
  <Pressable onPress={() => console.log('Pressed!')}>
    <Text>I'm pressable!</Text>
  </Pressable>
);
```

KeyboardAvoidingView: Ajuster le contenu pour éviter le clavier virtuel

```
import { KeyboardAvoidingView, TextInput } from 'react-native';

const MyKeyboardAvoidingView = () => (
  <KeyboardAvoidingView behavior="padding" style={{flex: 1}}>
    <TextInput
      style={{height: 40, borderColor: 'gray', borderWidth: 1}}
      defaultValue="Type here"
    />
  </KeyboardAvoidingView>
);
```

- Le composant KeyboardAvoidingView ajuste automatiquement le contenu de votre application pour éviter qu'il ne soit masqué par le clavier virtuel lorsqu'un champ de saisie de texte est activé.
- Cela garantit une bonne expérience utilisateur lors de la saisie de texte.

RefreshControl: Actualiser le contenu en tirant vers le bas

- Le composant RefreshControl permet d'ajouter une fonctionnalité de "pull to refresh" aux listes et aux vues défilantes.
- Lorsqu'un utilisateur tire la liste vers le bas, une icône de chargement s'affiche et vous pouvez déclencher une action pour actualiser le contenu.

```
import { ScrollView, RefreshControl, Text } from 'react-native';
import React, { useState } from 'react';

const wait = (timeout) => {
  return new Promise(resolve => setTimeout(resolve, timeout));
}

const MyRefreshControl = () => {
  const [refreshing, setRefreshing] = useState(false);

  const onRefresh = React.useCallback(() => {
    setRefreshing(true);
    wait(2000).then(() => setRefreshing(false));
  }, []);

  return (
    <ScrollView
      refreshControl={
        <RefreshControl refreshing={refreshing} onRefresh={onRefresh} />
      }
    >
      <Text>Pull down to see RefreshControl</Text>
    </ScrollView>
  );
};
```

StatusBar: Personnaliser la barre de statut

```
import { StatusBar } from 'react-native';

const MyStatusBar = () => (
  <>
    <StatusBar barStyle="dark-content" />
  </>
);
```

- Le composant StatusBar permet de contrôler l'apparence de la barre de statut du système d'exploitation (heure, batterie, etc.).
- Vous pouvez modifier sa couleur, son style et sa visibilité.

Platform: Détecter la plateforme et adapter le rendu

- Le module Platform permet de détecter la plateforme sur laquelle l'application est exécutée (iOS ou Android).
- Vous pouvez utiliser cette information pour adapter le rendu et le comportement de votre application en fonction de la plateforme.

```
import { Platform, Text } from 'react-native';

const MyPlatformSpecificCode = () => (
  <Text>
    {Platform.OS === 'ios' ? 'Welcome to iOS' : 'Welcome to Android'}
  </Text>
);
```

03

Le style

Définir et appliquer des styles avec StyleSheet

StyleSheet.create permet de créer des objets de style réutilisables.
La prop style des composants reçoit un objet de style.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  text: {
    fontSize: 20,
    color: 'blue',
  },
});

const App = () => (
  <View style={styles.container}>
    <Text style={styles.text}>Ceci est un texte stylisé</Text>
  </View>
);
```

Organiser et réutiliser les styles

- Stocker les styles dans des fichiers séparés par thème ou fonctionnalité.
- Importer et utiliser les styles dans différents composants.

```
// Fichier styles.js
export const styles = StyleSheet.create({
  ...
});

// Fichier App.js
import { styles } from './styles';

const App = () => (
  <View style={styles.container}>
    ...
  </View>
);
```

Avantages de StyleSheet pour la performance

```
// Style inline (à éviter)
const App = () => (
  <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <Text style={{ fontSize: 20, color: 'blue' }}>Ceci est un texte stylisé</Text>
  </View>
);
```

StyleSheet optimise les styles pour une meilleure mémoire et un rendu plus rapide.
Évitez les styles inline qui peuvent ralentir l'application.

Flexbox pour structurer les vues

```
<View style={{ flexDirection: 'row', justifyContent: 'space-around' }}>
  <View style={{ backgroundColor: 'red', width: 100, height: 100 }} />
  <View style={{ backgroundColor: 'blue', width: 100, height: 100 }} />
  <View style={{ backgroundColor: 'green', width: 100, height: 100 }} />
</View>
```

flexDirection: Axe principal du layout (row, column).
justifyContent: Alignement des éléments sur l'axe principal.
alignItems: Alignement des éléments sur l'axe secondaire.
flexWrap: Gestion des débordements sur l'axe principal.

Layouts verticaux, horizontaux et grille

flexDirection: 'column' pour un layout vertical.
flexDirection: 'row' pour un layout horizontal.
flexWrap: 'wrap' pour créer une grille.

```
<View style={{ flexDirection: 'column', flexWrap: 'wrap' }}>
  <View style={{ backgroundColor: 'red', width: 100, height: 100 }} />
  <View style={{ backgroundColor: 'blue', width: 100, height: 100 }} />
  <View style={{ backgroundColor: 'green', width: 100, height: 100 }} />
</View>
```

Récupérer les dimensions de l'écran

- Dimensions.get('window') renvoie les dimensions de l'écran.
- Adaptez le layout en fonction de la largeur et de la hauteur.

```
import { Dimensions } from 'react-native';

const windowWidth = Dimensions.get('window').width;

const App = () => (
  <View style={{ width: windowWidth * 0.8 }}>
    {/* ... */}
  </View>
);
```

Écoute des changements d'orientation

```
import { Dimensions, useEffect, useState } from 'react-native';

const App = () => {
  const [orientation, setOrientation] = useState(Dimensions.get('window').orientatio

  useEffect(() => {
    const handleChange = (event) => setOrientation(event.nativeEvent.orientation);
    Dimensions.addEventListener('change', handleChange);

    return () => Dimensions.removeEventListener('change', handleChange);
  }, []);

  return (
    <View style={{ ...styles.container, [orientation === 'LANDSCAPE' ? styles.landsc
      {/* ... */}
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  landscape: {
    backgroundColor: 'lightblue',
  },
  portrait: {
    backgroundColor: 'lightyellow',
  },
});
```

- Utilisez Dimensions.addEventListener('change', callback) pour écouter les changements d'orientation.
- Adaptez l'interface en fonction de la nouvelle orientation.

Approches de design responsive

```
<View style={{ flex: 1 }}>
  <Image
    source={{ uri: 'https://example.com/image.jpg' }}
    style={{ width: '100%', aspectRatio: 16 / 9 }}
  />
  <View style={{ padding: 10, backgroundColor: 'lightgray' }}>
    <Text style={{ fontSize: 16 }}>Description de l'image</Text>
  </View>
</View>
```

Utilisez des valeurs en pourcentage pour les largeurs/hauteurs.
Utilisez aspectRatio pour définir les proportions des composants.

React Native Responsive Screen

- Facilite la création de vues responsives en fonction de la taille de l'écran.
- Fournit des composants et des utilitaires pour simplifier le développement responsive.

Large choix de composants adaptables

- useResponsiveSize: Permet d'adapter la taille des polices et des images en fonction de la taille de l'écran.
- useResponsiveHeight: Permet d'adapter la hauteur des éléments en fonction de la taille de l'écran.
- useResponsiveWidth: Permet d'adapter la largeur des éléments en fonction de la taille de l'écran.
- ResponsiveText: Affiche du texte avec une taille de police adaptative.
- ResponsiveImage: Affiche une image avec une taille et un aspect ratio adaptables.
- ResponsiveView: Conteneur avec une taille et des marges adaptables.

Outils pour faciliter le développement responsive

```
import { scale, moderateScale } from 'react-native-size-matters';

const App = () => {
  const buttonWidth = scale(150);
  const fontSize = moderateScale(16);

  return (
    <View>
      <Button style={{ width: buttonWidth }} title="Bouton" />
      <Text style={{ fontSize }}>Texte adaptatif</Text>
    </View>
  );
};
```

getPercentageWidth: Convertit une valeur en pixels en pourcentage de la largeur de l'écran.
getPercentageHeight: Convertit une valeur en pixels en pourcentage de la hauteur de l'écran.
getResponsiveValue: Calcule une valeur adaptative en fonction de la taille de l'écran.
isLandscape: Détecte si l'appareil est en mode paysage.
isPortrait: Détecte si l'appareil est en mode portrait.

Création d'une page d'accueil responsive

```
import { useResponsiveHeight, ResponsiveText, ResponsiveImage } from 'react-native-

const App = () => {
  const height = useResponsiveHeight(80);

  return (
    <View style={{ flex: 1 }}>
      <ResponsiveImage
        source={{ uri: 'https://example.com/image.jpg' }}
        style={{ height }}
      />
      <ResponsiveText style={{ fontSize: 24 }}>Titre de la page</ResponsiveText>
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
        <ResponsiveText>Ceci est le contenu de la page d'accueil</ResponsiveText>
      </View>
    </View>
  );
};
```

Définir une mise en page flexible avec des composants responsives.
Adapter la taille des polices et des images en fonction de la taille de l'écran.
Afficher du contenu différent en fonction du mode portrait ou paysage.

04

La Navigation

Configuration de React Navigation

- Installez les packages React Navigation :

```
yarn add @react-navigation/native
```

- Ajoutez les dépendances spécifiques aux navigateurs que vous souhaitez utiliser :

```
yarn add @react-navigation/stack
yarn add @react-navigation/bottom-tabs
yarn add @react-navigation/drawer
```

Configuration du navigateur racine

Créez un fichier App.js et importez NavigationContainer depuis react-navigation.

Enveloppez votre application dans NavigationContainer:

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

const App = () => {
  return (
    <NavigationContainer>
      {/* ... Vos navigateurs ici ... */}
    </NavigationContainer>
  );
};

export default App;
```

Création de Stack Navigator

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

const Stack = createStackNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

Importez createStackNavigator depuis @react-navigation/stack.

Créez une fonction qui définit le Stack Navigator et les écrans qu'il contiendra:

Personnalisez les options de navigation par écran:

```
const Stack = createStackNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Home"
          component={HomeScreen}
          options={{ title: "Accueil" }}
        />
        <Stack.Screen
          name="Details"
          component={DetailsScreen}
          options={{ title: "Détails", headerStyle: { backgroundColor: 'red' } }}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

Gérez les transitions et animations entre les écrans:

```
const Stack = createStackNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Home"
          component={HomeScreen}
          options={{
            title: "Accueil",
            transitionSpec: {
              open: { animation: 'timing', duration: 1000 },
              close: { animation: 'timing', duration: 500 },
            },
          }}
        />
        <Stack.Screen
          name="Details"
          component={DetailsScreen}
          options={{ title: "Détails" }}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

Création de Tab Navigator

- Importez createBottomTabNavigator depuis @react-navigation/bottom-tabs.
- Créez un Tab Navigator et configurez les onglets:

```
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

const Tab = createBottomTabNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <Tab.Screen name="Home" component={HomeScreen} />
        <Tab.Screen name="Settings" component={SettingsScreen} />
      </Tab.Navigator>
    </NavigationContainer>
  );
};
```

Personnalisez les icônes, labels et animations des onglets:

```
const Tab = createBottomTabNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <Tab.Screen
          name="Home"
          component={HomeScreen}
          options={{
            tabBarLabel: "Accueil",
            tabBarIcon: ({ focused }) => (
              <Icon name="home" size={24} color={focused ? "blue" : "gray"} />
            ),
          }}
        />
        <Tab.Screen
          name="Settings"
          component={SettingsScreen}
          options={{
            tabBarLabel: "Paramètres",
            tabBarIcon: ({ focused }) => (
              <Icon name="settings" size={24} color={focused ? "blue" : "gray"} />
            ),
          }}
        />
      </Tab.Navigator>
    </NavigationContainer>
  );
};
```

Personnalisez le comportement des tabs:

```
const Tab = createBottomTabNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        {/* Vos onglets ici */}
      </Tab.Navigator>
    </NavigationContainer>
  );
};
```

Création de Drawer Navigator

- Importez createDrawerNavigator depuis @react-navigation/drawer.
- Créez un Drawer Navigator et configurez les écrans qu'il contiendra:

```
import { createDrawerNavigator } from '@react-navigation/drawer';

const Drawer = createDrawerNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Drawer.Navigator>
        <Drawer.Screen name="Home" component={HomeScreen} />
        <Drawer.Screen name="Settings" component={SettingsScreen} />
      </Drawer.Navigator>
    </NavigationContainer>
  );
};
```

Personnalisez le contenu du drawer:

```
import { createDrawerNavigator } from '@react-navigation/drawer';

const Drawer = createDrawerNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Drawer.Navigator>
        <Drawer.Screen
          name="Home"
          component={HomeScreen}
          options={{ drawerLabel: "Accueil" }}
        />
        <Drawer.Screen
          name="Settings"
          component={SettingsScreen}
          options={{ drawerLabel: "Paramètres" }}
        />
      </Drawer.Navigator>
    </NavigationContainer>
  );
};
```

Personnalisez les options du drawer:

```
import { createDrawerNavigator } from '@react-navigation/drawer';

const Drawer = createDrawerNavigator();

const App = () => {
  return (
    <NavigationContainer>
      <Drawer.Navigator
        drawerContent={(props) => <CustomDrawerContent {...props} />} //
        drawerWidth={250} // Largeur du drawer
        drawerStyle={{ backgroundColor: 'lightgray' }} // Style du drawer
      >
        <Drawer.Screen
          name="Home"
          component={HomeScreen}
          options={{ drawerLabel: "Accueil" }}
        />
        <Drawer.Screen
          name="Settings"
          component={SettingsScreen}
          options={{ drawerLabel: "Paramètres" }}
        />
      </Drawer.Navigator>
    </NavigationContainer>
  );
};
```

Passage de données entre les écrans

1. Utilisez les **paramètres** pour envoyer des données d'un écran à l'autre lors de la navigation
2. Récupérez les paramètres passés dans le composant de destination

```
const DetailsScreen = ({ route }) => {
  const { userId } = route.params; // Récupère les paramètres

  return (
    <Text>Utilisateur {userId}</Text>
  );
};
```

```
const HomeScreen = () => {
  const [userId, setUserId] = useState(1);

  const navigateToDetails = () => {
    navigation.navigate('Details', { userId });
  };

  return (
    <Button title="Go to Details" onPress={navigateToDetails} />
  );
};

const DetailsScreen = ({ route }) => {
  const { userId } = route.params;

  return (
    <Text>Utilisateur {userId}</Text>
  );
};
```

05

Introduction au JSX et aux Composants

Qu'est-ce que le JSX ?

JSX est une extension syntaxique pour JavaScript utilisée par React pour décrire l'apparence de l'interface utilisateur. Elle permet aux développeurs d'écrire du code qui ressemble à HTML dans leurs fichiers JavaScript, rendant la structure de l'interface utilisateur plus lisible et expressive. Grâce à la transpilation par Babel, le JSX est converti en appels de fonctions JavaScript efficaces.

```
// Exemple de JSX
const element = <h1>Bonjour, monde !</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Fondements des Composants React

```
// Exemple de composant fonctionnel
function Welcome(props) {
  return <h1>Bonjour, {props.name}</h1>;
}

// Utilisation du composant
const element = <Welcome name="Alice" />;
ReactDOM.render(element, document.getElementById('root'));
```

Les composants sont au cœur de React, servant de blocs de construction pour créer des interfaces utilisateur dynamiques. Ils peuvent être définis comme des fonctions ou des classes, mais avec l'introduction des hooks, les composants fonctionnels sont devenus la norme pour leur simplicité et leur efficacité.

Le Pattern de Composition

```jsx
// Exemple de composition pour séparer les préoccupations
function Container({ children }) {
  return <div className="container">{children}</div>;
}

function PresentationComponent() {
  return <div className="presentation">Contenu de présentation</div>;
}

// Utilisation de la composition
ReactDOM.render(
  <Container>
    <PresentationComponent />
  </Container>,
  document.getElementById('root')
);
```

React favorise la composition de composants, une méthode permettant de créer des interfaces utilisateur complexes et réutilisables en assemblant de petits composants. Cette approche offre des avantages significatifs en termes de réutilisabilité, d'abstraction et de maintenance du code.

Bonnes Pratiques avec JSX et la Composition

La composition peut être utilisée pour séparer les préoccupations dans une application, avec des composants de "Container" gérant la logique et des composants de "Présentation" gérant l'affichage. La spécialisation permet également de créer des composants personnalisés basés sur des composants génériques.

```
// Exemple de JSX propre
function App() {
  return (
    <div>
      <Header />
      <Content />
      <Footer />
    </div>
  );
}

// Éviter les imbrications profondes et utiliser des noms clairs
```

L'Importance du JSX et de la Composition

Le JSX et les composants sont fondamentaux pour le développement avec React, offrant une syntaxe expressive pour construire des interfaces et une stratégie solide pour organiser le code. La composition, en particulier, est centrale pour créer des applications évolutives et faciles à maintenir.

06

Les Props

Définition des Props

```
// Exemple de composant utilisant des props
function Welcome(props) {
  return <h1>Bonjour, {props.name}</h1>;
}

// Utilisation du composant avec des props
const element = <Welcome name="Alice" />;
ReactDOM.render(element, document.getElementById('root'));
```

Les props, ou propriétés, sont un concept clé dans React permettant de passer des données de composants parents à des composants enfants. En tant que valeurs immuables, elles favorisent un flux de données unidirectionnel, rendant les composants plus prévisibles et réutilisables.

Importance des Props

Les props rendent les composants dynamiques et réutilisables en permettant la personnalisation et la configuration. Elles jouent un rôle crucial dans la communication entre les composants et la gestion de l'état au sein de l'arbre des composants.

```
// Exemple de composant réutilisable avec des props
function Button(props) {
  return <button>{props.label}</button>;
}

// Utilisation du composant avec différentes props
ReactDOM.render(<Button label="Cliquez ici" />, document.getElementById('root'));
ReactDOM.render(<Button label="Soumettre" />, document.getElementById('root'));
```

Types de Props

```
function List(props) {
  return (
    <ul>
      {props.children}
    </ul>
  );
}

// Utilisation du composant avec la prop children
ReactDOM.render(
  <List>
    <li>Item 1</li>
    <li>Item 2</li>
  </List>,
  document.getElementById('root')
);
```

Props Standard : Chaînes de caractères, nombres, booléens, et autres types JavaScript.
Props Spéciales :
children : Pour passer des éléments enfants directement dans le rendu d'un composant.
key : Utilisée par React pour identifier de manière unique les éléments dans une liste.

Validation des Props avec PropTypes

PropTypes est un outil permettant de vérifier que les composants reçoivent des props du bon type. Cela aide à prévenir les bugs et facilite le développement en émettant des avertissements lorsqu'un type attendu n'est pas respecté.

```
import PropTypes from 'prop-types';

function MyComponent(props) {
  // Contenu du composant
}

MyComponent.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number
};
```

Bonnes Pratiques avec les Props

- **Prop Types et Defaults :** Définissez explicitement les types et valeurs par défaut des props pour améliorer la robustesse et la lisibilité.
- **Props Destructuring :** Améliore la lisibilité et simplifie l'accès aux props dans le corps du composant.

```
function Welcome(props) {
  const { name, age } = props;
  return <h1>Bonjour, {name} ({age} ans)</h1>;
}

// Définition des propTypes et defaultProps
Welcome.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number
};

Welcome.defaultProps = {
  age: 25
};
```

07

Les Hooks

Introduction aux Hooks

L'objectif principal des Hooks est de simplifier le code des composants fonctionnels en donnant accès à l'état et au cycle de vie, deux aspects auparavant réservés aux composants de classe. Cela rend le code plus court, plus lisible, et facilite la gestion de l'état et des effets secondaires.

Le Hook useState

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

useState est le Hook permettant d'ajouter un état local à un composant fonctionnel. Il retourne un tableau contenant la valeur actuelle de l'état et une fonction pour le modifier.

Effets Secondaires avec useEffect

```
useEffect(() => {
  document.title = `Vous avez cliqué ${count} fois`;
});
```

useEffect permet d'exécuter du code pour des effets secondaires dans les composants fonctionnels, remplaçant ainsi les méthodes de cycle de vie des composants de classe.

Le Hook useReducer

useReducer est une alternative à useState, idéale pour gérer des états plus complexes avec une logique d'état local basée sur des actions.

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Accès aux Éléments DOM avec useRef

useRef permet de conserver une référence mutable à travers les re-renders du composant, souvent utilisé pour accéder à un élément DOM directement.

```
const myRef = useRef(initialValue);
```

08

Manipulation de Listes

Fondamentaux des Listes en JavaScript

```javascript
// Exemple de manipulation de liste avec map
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6, 8, 10]
```

Les listes (ou tableaux) en JavaScript sont manipulées à l'aide de méthodes telles que map, filter, et reduce. Ces outils puissants permettent de traiter et de transformer des tableaux de manière efficace, en construisant de nouveaux tableaux sans modifier les originaux.

La Méthode map pour le Rendu des Listes

Dans React, map est essentiel pour convertir des données en éléments visuels. Elle permet de transformer chaque élément d'un tableau en un composant React, facilitant ainsi le rendu dynamique des listes.

```
// Exemple d'utilisation de map pour le rendu de listes en React
const items = data.map(item => <ListItem key={item.id} {...item} />);
```

Rôle des Key dans les Listes React

```
// Exemple d'utilisation de key dans une liste React
const items = data.map(item => <ListItem key={item.id} {...item} />);
```

L'attribut key est crucial dans les listes React pour optimiser les performances du rendu. Il aide React à identifier les éléments modifiés, ajoutés ou supprimés, et doit être un identifiant unique pour chaque élément de la liste.

Filtrage de Listes avec filter

filter crée un nouveau tableau contenant uniquement les éléments qui répondent à une condition spécifiée, permettant ainsi de manipuler les données affichées sans altérer le tableau original.

```
// Exemple de filtrage de liste avec filter
const users = [
  { id: 1, name: 'John', active: true },
  { id: 2, name: 'Jane', active: false },
  { id: 3, name: 'Doe', active: true }
];

const activeUsers = users.filter(user => user.active);
console.log(activeUsers);
```

Autres Méthodes de Raccourcis

```
// Exemple d'utilisation de reduce, forEach, find et findIndex
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue
console.log(sum);

numbers.forEach(num => console.log(num));

const foundNumber = numbers.find(num => num === 3);
console.log(foundNumber);

const foundIndex = numbers.findIndex(num => num === 3);
console.log(foundIndex);
```

reduce accumule les valeurs d'un tableau en une seule valeur.
forEach exécute une action pour chaque élément du tableau.
find et findIndex permettent de localiser des éléments spécifiques dans un tableau.

09

Redux et Zustand

Introduction à Redux

Redux est une bibliothèque de gestion d'état prévisible pour applications JavaScript. Elle aide à écrire des applications qui se comportent de manière consistante, tournent dans différents environnements (client, serveur et natif), et sont faciles à tester.

Principes fondamentaux de Redux

Redux repose sur quelques principes clés : l'état global de l'application est stocké dans un objet arbre unique au sein d'un seul store. L'état est en lecture seule. Les changements d'état sont effectués en envoyant des actions. Les réducteurs sont des fonctions pures qui prennent l'état précédent et une action, et retournent le nouvel état.

Configuration initiale de Redux

```
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);
```

Pour commencer avec Redux, installez redux et react-redux. Ensuite, créez un store Redux et fournissez-le à votre application via le Provider de React-Redux.

Actions

Les actions sont des objets JavaScript qui envoient des données de votre application vers votre store. Elles sont la seule source d'informations pour le store.

```
const addAction = { type: 'ADD', payload: 1 };
```

Reducers

```
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'ADD':
      return state + action.payload;
    default:
      return state;
  }
}
```

Les reducers spécifient comment l'état de l'application change en réponse aux actions envoyées au store. Rappelez-vous que les actions décrivent le fait que quelque chose s'est passé, mais ne spécifient pas comment l'état de l'application change.

useSelector et useDispatch

useSelector permet d'accéder à l'état du store, tandis que useDispatch vous donne accès à la fonction dispatch pour envoyer des actions.

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

function Counter() {
  const count = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch({ type: 'ADD', payload: 1 })}>
        Increment
      </button>
      <span>{count}</span>
    </div>
  );
}
```

Store et gestion de l'état

```
import { combineReducers, createStore } from 'redux';
import counterReducer from './reducers/counterReducer';

const rootReducer = combineReducers({
  counter: counterReducer
});

const store = createStore(rootReducer);
```

- Le store de Redux sert de conteneur pour l'état global de votre application. Utilisez combineReducers pour diviser l'état et la logique de réduction en plusieurs fonctions gérant des parties indépendantes de l'état.

Middleware Redux

Les middlewares offrent un point d'extension entre l'envoi d'une action et le moment où elle atteint le réducteur. Utilisez redux-thunk pour gérer la logique asynchrone.

```
import { applyMiddleware, createStore } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));
```

Introduction à Redux Toolkit

Le Redux Toolkit (RTK) est un ensemble d'outils visant à simplifier le code Redux, encourager les bonnes pratiques et améliorer la dévelopabilité avec Redux. Il offre des utilitaires pour simplifier la configuration du store, la définition des reducers, la gestion de la logique asynchrone, et plus encore.

Avantages par rapport à Redux standard

RTK réduit la quantité de code boilerplate nécessaire pour configurer un store Redux, simplifie la gestion des actions et des reducers avec createSlice, automatise la création d'actions, et facilite la gestion de la logique asynchrone avec createAsyncThunk.

Installation et configuration

```
import { configureStore } from '@reduxjs/toolkit';

export const store = configureStore({
  reducer: {
    // Reducers vont ici
  },
});
```

Pour démarrer avec RTK, installez le paquet @reduxjs/toolkit ainsi que react-redux si vous travaillez avec React.

Configurer le Store avec configureStore

configureStore simplifie la configuration du store en incluant automatiquement des middlewares comme Redux Thunk et en activant les outils de développement Redux.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {
    increment: state => state + 1,
    decrement: state => state - 1,
  },
});

export const { increment, decrement } = counterSlice.actions;
```

Création de Slice avec createSlice

```
import { createAsyncThunk } from '@reduxjs/toolkit';

export const fetchUserData = createAsyncThunk(
  'users/fetchById',
  async (userId, thunkAPI) => {
    const response = await fetch(`https://api.example.com/users/
    return await response.json();
  }
);
```

createSlice permet de regrouper les reducers et les actions correspondantes dans un seul objet, simplifiant ainsi la gestion de l'état.

Gestion des effets secondaires avec createAsyncThunk

createAsyncThunk simplifie la gestion des opérations asynchrones en encapsulant la logique asynchrone et le traitement des états de la requête (loading, success, error) dans une seule fonction.

```
import { createSelector } from '@reduxjs/toolkit';

const selectCounterValue = state => state.counter.value;
const selectIsCounterEven = createSelector(
  [selectCounterValue],
  value => value % 2 === 0
);
```

Utilisation de useDispatch et useSelector avec Redux Toolkit et React

```jsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment } from './slices/counterSlice';

function CounterComponent() {
  const count = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <span>{count}</span>
      <button onClick={() => dispatch(increment())}>Increment</button>
    </div>
  );
}
```

- RTK fonctionne de manière transparente avec les hooks useDispatch et useSelector de react-redux, facilitant l'accès à l'état et la dispatch d'actions dans les composants React.

Introduction à Zustand

Zustand est une petite bibliothèque de gestion d'état pour React qui offre une approche plus simple et plus directe que Redux. Avec Zustand, la création de stores globaux pour gérer l'état est simplifiée et ne nécessite pas de boilerplate ou de middleware supplémentaire.

Philosophie et avantages de Zustand

- Zustand se concentre sur une API minimaliste et un hook personnalisé pour accéder au store. Les avantages incluent une configuration facile, pas de dépendance à Redux ou Context API, et une intégration naturelle avec les hooks de React.

Installation de Zustand

L'installation de Zustand est simple et directe, en utilisant npm ou yarn. Ceci installe la dernière version de Zustand dans votre projet.

```
npm install zustand
// ou
yarn add zustand
```

Création d'un store avec Zustand

```
import create from 'zustand';

const useStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 })),
  decrement: () => set(state => ({ count: state.count - 1 })),
}));
```

La création d'un store dans Zustand se fait en utilisant la fonction create. Vous pouvez y définir l'état initial du store et les actions qui manipuleront cet état. Zustand permet de créer des stores sans l'overhead traditionnellement associé à Redux.

Gestion de l'état avec des hooks

Zustand utilise des hooks pour accéder et manipuler l'état du store. Cela rend l'intégration avec les composants fonctionnels React naturelle et efficace.

```
function Counter() {
  const { count, increment, decrement } = useStore();
  return (
    <div>
      <button onClick={decrement}>-</button>
      <span>{count}</span>
      <button onClick={increment}>+</button>
    </div>
  );
}
```

Gestion des effets secondaires

```
import create from 'zustand';
import { useEffect } from 'react';

const useStore = create(set => ({
  data: null,
  fetchData: async () => {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    set({ data });
  },
}));

// Dans un composant
const Component = () => {
  const { data, fetchData } = useStore();
  useEffect(() => {
    fetchData();
  }, [fetchData]);

  return <div>{data && <p>{data.someField}</p>}</div>;
};
```

Zustand permet de gérer des effets secondaires en utilisant des actions ou en réagissant à des changements d'état spécifiques à l'aide de middlewares ou de l'API native React.useEffect.

Sélecteurs et abonnements

Zustand permet de sélectionner une partie de l'état lors de l'utilisation du hook du store, réduisant ainsi les re-renders inutiles. Les abonnements aux changements d'état sont également possibles pour une gestion fine des mises à jour.

```
const count = useStore(state => state.count);
```

Exemples d'utilisation avancée de Zustand

```
import create from 'zustand';
import { persist } from 'zustand/middleware';

const useStore = create(persist(set => ({
  user: null,
  setUser: user => set({ user }),
}), {
  name: 'user-settings', // nom de la clé du local storage
}));
```

Zustand supporte des cas d'utilisation avancés comme le partage de l'état entre différents composants, la persistance de l'état dans le local storage, et l'intégration avec des outils de débogage.

10

Les Contexts

Introduction au Contexte dans React

Le Contexte permet de partager des valeurs facilement entre plusieurs composants, sans nécessiter de passer explicitement une prop à chaque niveau de l'arbre des composants. Il est idéal pour des données dites "globales" telles que le thème, les préférences utilisateur, etc.

Création et Utilisation du Contexte

La mise en place d'un contexte commence par React.createContext(), qui retourne un objet contenant un composant Provider et Consumer. Le Provider permet de fournir une valeur de contexte à tous les composants enfants qui l'encapsulent.

```
const MyContext = React.createContext(defaultValue);
```

Fournir des Valeurs avec le Provider

```
<MyContext.Provider value={/* some value */}>
  {/* child components */}
</MyContext.Provider>
```

Le composant Provider sert à fournir une valeur de contexte à l'arbre des composants, permettant ainsi aux composants enfants d'accéder à ces données sans passer par une prop.

Accéder aux Valeurs de Contexte

Les valeurs de contexte sont accessibles aux composants enfants via le composant Consumer, le hook useContext dans les composants fonctionnels, ou MyContext.Consumer et static contextType dans les composants de classe.

```
const value = useContext(MyContext);
```

Meilleures Pratiques avec le Contexte

- **Restriction d'Usage :** Utilisez le contexte pour des données globales qui changent rarement.
- **Mise à Jour du Contexte :** Optimisez les mises à jour pour éviter les rendus inutiles.
- **Composition de Contextes :** Utilisez plusieurs contextes pour séparer les préoccupations sans recourir excessivement au "prop drilling".

11

Manipulation de données

Fetch API et Axios pour les appels réseau

```
const fetchUsers = () => {
  fetch('https://api.example.com/users')
    .then((response) => response.json())
    .then((data) => {
      // Traitement des données récupérées
      console.log(data);
    })
    .catch((error) => {
      // Gestion des erreurs
      console.error(error);
    });
};
```

- **Fetch API**
- **Description:**
- API native de JavaScript pour effectuer des requêtes HTTP.
- Permet de réaliser des requêtes GET, POST, PUT, DELETE.
- Fonctionne avec des promesses pour gérer les réponses asynchrones.

Axios

```
import axios from 'axios';

const axiosInstance = axios.create({
  baseURL: 'https://api.example.com/',
});

const fetchUsers = () => {
  axiosInstance.get('/users')
    .then((response) => {
      // Traitement des données récupérées
      console.log(response.data);
    })
    .catch((error) => {
      // Gestion des erreurs
      console.error(error.response.data);
    });
};
```

- Bibliothèque HTTP populaire pour React Native.
- Offre une interface plus simple et plus flexible que Fetch API.
- Permet de configurer des intercepteurs pour gérer les tokens d'authentification globalement.

AsyncStorage pour le stockage local des données

```
const saveAuthToken = (token) => {
  AsyncStorage.setItem('authToken', token);
};

const getAuthToken = () => {
  return AsyncStorage.getItem('authToken');
};
```

- Solution (anciennement) native de React Native pour stocker des données simples (tokens, préférences utilisateur).
- Les données sont stockées de manière asynchrone.

Installation de SQLite

Expo SQLite:
expo install expo-sqlite

- Créer une instance de base de données:

```
const db = SQLite.openDatabase({
  name: 'MyDatabase',
  location: 'default',
});
```

- Exécuter des requêtes SQL:

```
db.executeSql(
  `CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    email TEXT
  )`,
);

db.executeSql(
  `INSERT INTO users (name, email) VALUES (?, ?)`,
  [user.name, user.email],
);
```

- Récupérer des données:

```
db.executeSql(
  `SELECT * FROM users WHERE id = ?`,
  [id],
).then((results) => results.rows.item(0));
```

Installation de Realm `yarn add realm`

```
import Realm from 'realm';

const realm = new Realm({
  schema: [
    {
      name: 'User',
      properties: {
        id: 'int',
        name: 'string',
        email: 'string',
      },
    },
  ],
});
```

- Initialiser Realm:

Utilisation de Realm

- Lire des données:

```
const users = realm.objects('User');

const user = users.filtered('id = $0', id)[0];
```

- Enregistrer des données:

```
const user = {
  name: 'John Doe',
  email: 'johndoe@example.com',
};

realm.write(() => {
  realm.create('User', user);
});
```

Utilisation de Realm

- Mettre à jour des données:

```
realm.write(() => {
  user.name = 'Jane Doe';
});
```

- Supprimer des données:

```
realm.write(() => {
  realm.delete(user);
});
```

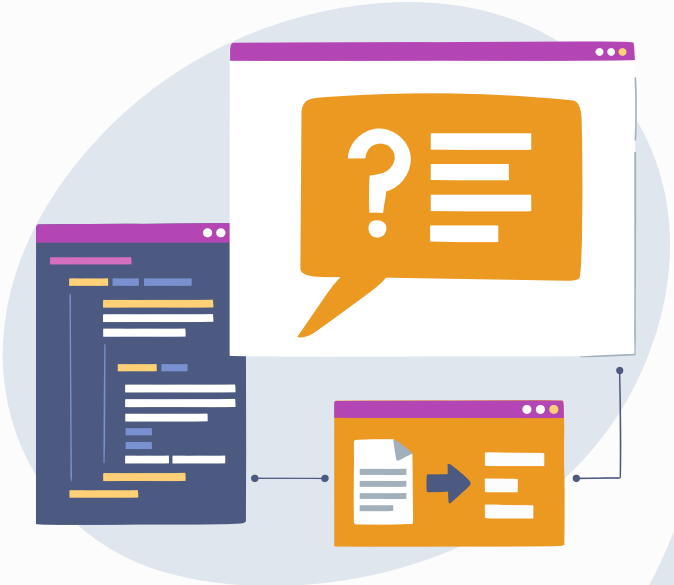
Accès aux Éléments DOM avec useRef

useRef permet de conserver une référence mutable à travers les re-renders du composant, souvent utilisé pour accéder à un élément DOM directement.

```
const myRef = useRef(initialValue);
```

12

Intégrations de quelques fonctionnalités natives



Caméra

```
import { Camera } from 'expo-camera';

const App = () => {
  const [cameraType, setCameraType] = useState(Camera.Constants.Type.back);
  const [hasPermission, setHasPermission] = useState(null);
  const [photo, setPhoto] = useState(null);

  useEffect(() => {
    (async () => {
      const { status } = await Camera.requestPermissionsAsync();
      setHasPermission(status === 'granted');
    })();
  }, []);

  const takePicture = async () => {
    if (!camera) return;
    const photo = await camera.takePictureAsync();
    setPhoto(photo.uri);
  };

  return (
    <View style={{ flex: 1 }}>
      <Camera style={{ flex: 1 }} type={cameraType} ref={ref => {
        camera = ref;
      }}>
        <View style={{ flex: 1, justifyContent: 'flex-end' }}>
          <Button title="Take Picture" onPress={takePicture} />
        </View>
      </Camera>
      {photo && <Image source={{ uri: photo }} style={{ flex: 1 }} />}
    </View>
  );
};

export default App;
```

- Accédez à la caméra du téléphone pour prendre des photos ou enregistrer des vidéos.
- Configurez les options de la caméra comme la résolution, le flash et le type de caméra.
- Stockez et affichez les médias capturés dans l'application.

Géolocalisation

```
import { Location } from 'expo-location';

const App = () => {
  const [location, setLocation] = useState(null);
  const [errorMessage, setErrorMessage] = useState(null);

  useEffect(() => {
    (async () => {
      let { status } = await Location.requestPermissionsAsync();
      if (status !== 'granted') {
        setErrorMessage('Permission de localisation refusée');
        return;
      }

      const location = await Location.getCurrentPositionAsync();
      setLocation(location);

      const watchId = Location.watchPositionAsync({}, (location) => {
        setLocation(location);
      });

      return () => Location.removeWatchAsync(watchId);
    })();
  }, []);

  return (
    <View style={{ flex: 1 }}>
      {location && <Text>Votre position: {JSON.stringify(location)}</Text>}
      {errorMessage && <Text>Erreur: {errorMessage}</Text>}
    </View>
  );
};

export default App;
```

- Obtenez la position géographique actuelle de l'utilisateur en temps réel.
- Suivez les changements de position avec des écouteurs d'événements.
- Utilisez les données de localisation pour des fonctionnalités comme la cartographie ou la géolocalisation inversée.

Gestion du FileSystem : Lecture

```
import { FileSystem } from 'expo-file-system';

const App = () => {
  const [fileContent, setFileContent] = useState(null);

  useEffect(() => {
    (async () => {
      const fileUri = await FileSystem.documentDirectory + 'myfile.txt';
      const content = await FileSystem.readAsStringAsync(fileUri);
      setFileContent(content);
    })();
  }, []);

  return (
    <View style={{ flex: 1 }}>
      {fileContent && <Text>Contenu du fichier: {fileContent}</Text>}
    </View>
  );
};

export default App;
```

Accédez aux fichiers stockés localement ou dans le stockage externe du téléphone.
Lisez le contenu de fichiers, tels que des documents texte, images ou données JSON.

Gestion du FileSystem : Ecriture

```
import { FileSystem } from 'expo-file-system';

const App = () => {
  const [fileName, setFileName] = useState('');
  const [fileContent, setFileContent] = useState('');

  const saveFile = async () => {
    if (!fileName.trim() || !fileContent.trim()) {
      alert('Le nom du fichier et le contenu ne peuvent pas être vides!');
      return;
    }
    const fileUri = FileSystem.documentDirectory + fileName;
    await FileSystem.writeAsStringAsync(fileUri, fileContent);
    alert('Fichier sauvegardé avec succès!');
  };

  return (
    <View style={{ flex: 1 }}>
      <TextInput
        value={fileName}
        onChangeText={setFileName}
        placeholder="Nom du fichier"
      />
      <TextInput
        value={fileContent}
        onChangeText={setFileContent}
        placeholder="Contenu du fichier"
        multiline={true}
      />
      <Button title="Sauvegarder le fichier" onPress={saveFile} />
    </View>
  );
};

export default App;
```

- Créez et écrivez des fichiers dans le stockage local de l'appareil.
- Gérez les permissions d'accès au stockage pour sauvegarder des fichiers.

13

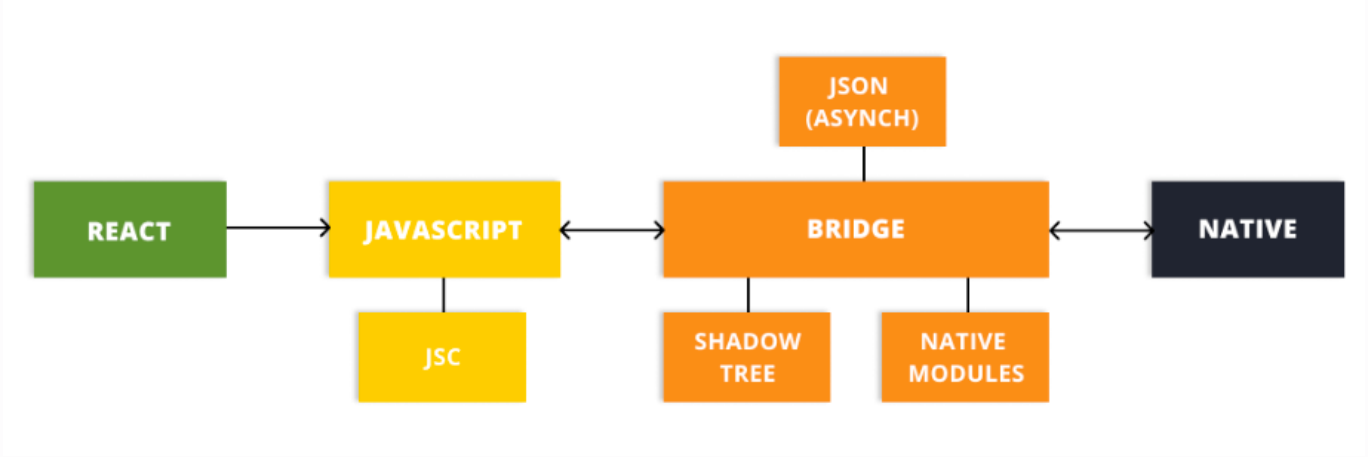
Fonctionnement de l'architecture de React Native

Fonctionnement

- Le pont JavaScript est un élément crucial de l'architecture React Native. Il permet la communication entre le code JavaScript et le code natif (iOS et Android). Le pont fonctionne de la manière suivante :

1. **Sérialisation des données JavaScript:** Le pont JavaScript sérialise les données JavaScript en JSON. Cela signifie que les objets JavaScript sont convertis en une chaîne de caractères au format JSON.
2. **Envoi des données au thread natif:** Le pont JavaScript envoie ensuite les données JSON au thread natif via un canal de communication. Ce canal peut être une file d'attente de messages ou un socket réseau.
3. **Déserialisation des données natives:** Le thread natif déserialise ensuite les données JSON et les utilise pour interagir avec les APIs natives.
4. **Renvoi des résultats:** Les résultats des appels aux APIs natives sont renvoyés au thread JavaScript. Le pont JavaScript les déserialise et les transmet au code JavaScript.

Fonctionnement



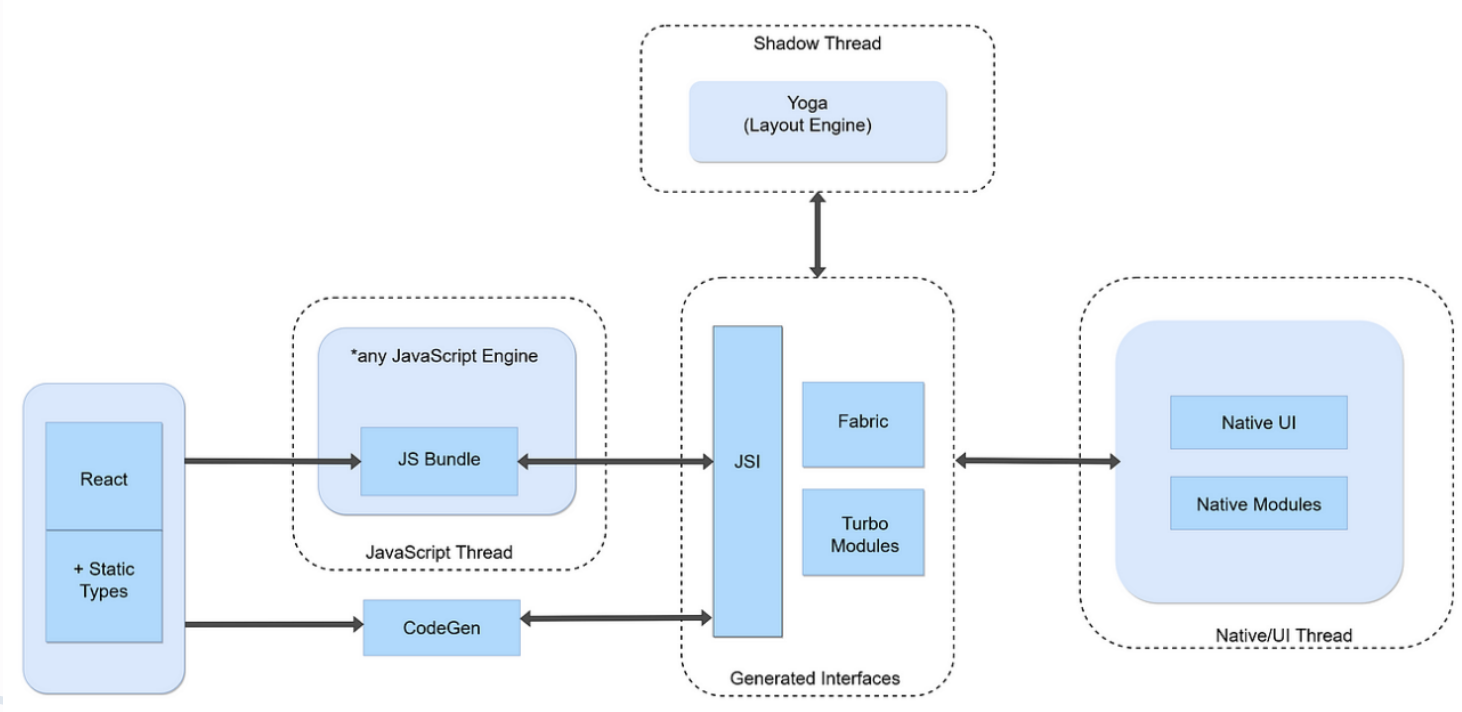
Hermes

- Hermes est un moteur JavaScript optimisé pour React Native.
- **Performances accrues:** Hermes offre des performances accrues par rapport au moteur JavaScript standard de JavaScriptCore (iOS) et V8 (Android). Cela se traduit par un rendu plus fluide et une expérience utilisateur plus réactive.
- **Compilation à l'avance (JIT):** Hermes utilise la technique de compilation à l'avance (Just-In-Time) pour optimiser le code JavaScript au moment de l'exécution. Cela permet d'exécuter le code plus rapidement et d'améliorer la réactivité de l'application.

JSI

- React Native JSI (Javascript Interface) est la nouvelle couche qui facilite et accélère la communication entre Javascript et les plateformes natives. C'est l'élément central de la réarchitecture de React Native avec Fabric UI Layer et Turbo Modules.

Nouvelle architecture



En quoi JSI est-il différent ?

- JSI supprime le besoin d'un pont entre le code natif (Java/ObjC) et le code Javascript. Il supprime également l'obligation de sérialiser/désérialiser toutes les informations en JSON pour la communication entre les deux mondes. JSI ouvre des portes à de nouvelles possibilités en rapprochant les mondes Javascript et natif.

Nouvelle architecture:

- La nouvelle architecture de React Native a été conçue pour résoudre les limitations des modules natifs classiques. Elle introduit les technologies suivantes :
- **Turbo Modules:** Les Turbo Modules sont une nouvelle génération de modules natifs qui utilisent JSI pour une communication plus performante entre le code JavaScript et le code natif.
- **Fabric:** Fabric est un nouveau système de rendu qui s'appuie sur les Turbo Modules et utilise Yoga pour un layout performant.
- **Codegen:** Codegen est un outil qui génère automatiquement le code natif et JavaScript nécessaire pour les Turbo Modules.

Yoga

- Yoga est une bibliothèque open source écrite en C++ utilisée par React Native pour gérer le layout des vues. Elle permet de calculer la position et la taille des éléments d'interface utilisateur de manière performante et flexible.
- **I. Fonctionnalités:**
- Yoga offre plusieurs fonctionnalités clés pour le layout des vues :
- **Flexbox:** Yoga implémente le layout Flexbox, qui permet de positionner les éléments d'interface utilisateur de manière flexible et réactive.
- **Shadow DOM:** Yoga utilise un Shadow DOM pour représenter la structure hiérarchique des vues, ce qui permet de calculer le layout de manière efficace.
- **Performances:** Yoga est optimisé pour les performances et permet de calculer le layout des vues rapidement et de manière fluide.

Thread Natif Android

- Le thread natif Android dans React Native est un thread d'exécution distinct du thread JavaScript. Il est responsable de l'exécution du code natif écrit en Java et offre plusieurs fonctionnalités importantes :
- **Affichage des éléments natifs:** Le thread natif est responsable de l'affichage des éléments natifs tels que les boutons, les images et les vues de texte.
- **Gestion des interactions utilisateur:** Le thread natif gère les interactions utilisateur telles que les touches, les clics et les gestes.
- **Accès aux APIs natives:** Le thread natif permet d'accéder aux APIs natives d'Android telles que l'appareil photo, le GPS et les capteurs.
- **Exécution de tâches en arrière-plan:** Le thread natif peut exécuter des tâches en arrière-plan sans bloquer le thread JavaScript.

Architecture Android

- Le thread natif Android est composé de plusieurs éléments clés :
- **Looper:** Le looper est une boucle d'événements qui gère les messages et les événements du système d'exploitation.
- **Handler:** Le handler est un objet qui reçoit et traite les messages du looper.
- **Activity:** L'activité est une classe qui représente une seule écran de l'application.
- **View:** La vue est une classe qui représente un élément d'interface utilisateur.

Thread Natif iOS

- Le thread natif iOS dans React Native fonctionne de manière similaire au thread natif Android, mais il s'appuie sur les concepts et les classes propres à iOS. Voici un aperçu de son fonctionnement :
- **Affichage des éléments natifs:** Le thread natif est responsable de l'affichage des éléments natifs tels que les boutons, les images et les vues de texte.
- **Gestion des interactions utilisateur:** Le thread natif gère les interactions utilisateur telles que les touches, les clics et les gestes.
- **Accès aux APIs natives:** Le thread natif permet d'accéder aux APIs natives d'iOS telles que la caméra, le GPS et les capteurs.
- **Exécution de tâches en arrière-plan:** Le thread natif peut exécuter des tâches en arrière-plan sans bloquer le thread JavaScript.

Architecture iOS

- **Run loop:** Le run loop est une boucle d'événements qui gère les messages et les événements du système d'exploitation.
- **NSOperationQueues:** Les NSOperationQueues sont des files d'attente qui permettent d'exécuter des tâches en arrière-plan.
- **UIViewController:** Le UIViewController est une classe qui représente une seule écran de l'application.
- **UIView:** La UIView est une classe qui représente un élément d'interface utilisateur.