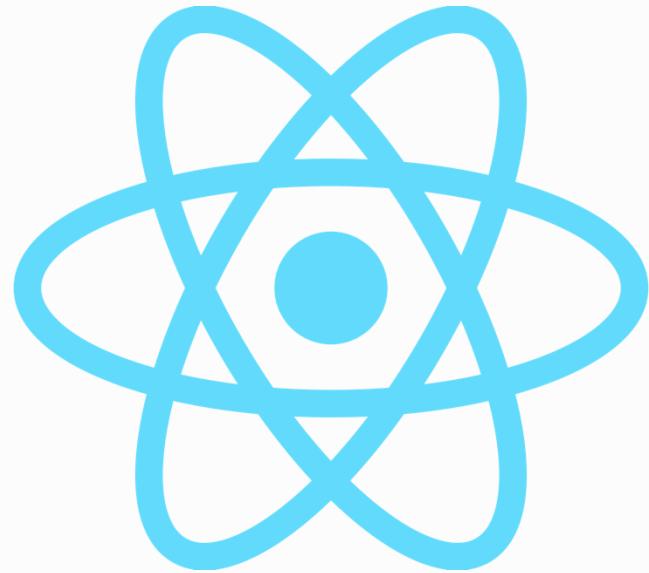


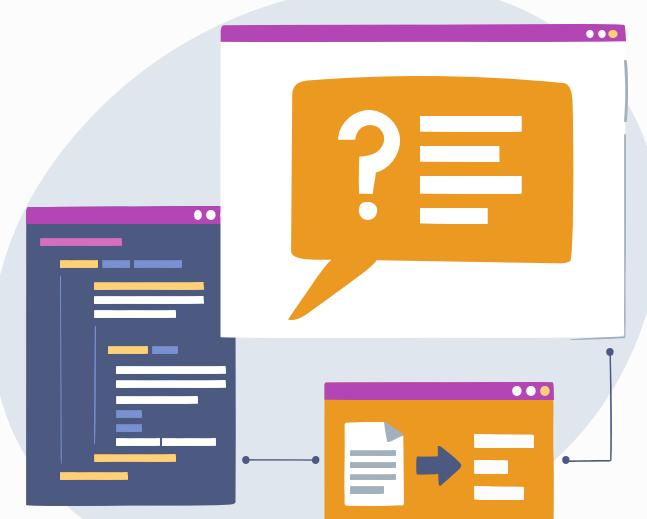
React Native



00

Rappels

ES6



Variables et Types de Données

Les variables en JavaScript peuvent être déclarées avec let, const, ou var (moins recommandé). let permet de déclarer des variables dont la valeur peut changer, tandis que const est pour des valeurs constantes.

Les types de données incluent les types primitifs (string, number, boolean, null, undefined, symbol) et les types non primitifs (objets, tableaux).

```
let age = 25; // Variable qui peut changer
const name = "John"; // Constante
let isStudent = false; // Type booléen
let score = null; // Type null
let x; // Undefined
const person = { firstName: "Alice", lastName: "Doe" }; // Objet
let numbers = [1, 2, 3]; // Tableau
```

Maîtriser les Opérateurs

```
let x = 10;
let y = 5;
let z = x + y; // 15
x += y; // x = x + y
let isEqual = x === y; // false
let isGreaterThan = x > y; // true
let andOperation = (x > 5) && (y < 10); // true
let orOperation = (x < 5) || (y > 2); // true
```

Les opérateurs en JavaScript incluent les opérateurs arithmétiques (+, -, *, /, %), d'assignation (=, +=, -=, etc.), de comparaison (==, ===, !=, !==, >, <, >=, <=) et logiques (&&, ||, !).

Structures de Contrôle

Les structures de contrôle dirigent le flux d'exécution du programme. Les conditions utilisent if, else if, else, et switch pour exécuter différents blocs de code basés sur des conditions. Les boucles for, while, et do...while permettent de répéter l'exécution d'un bloc de code.

```
if (x > 5) {
    console.log("x est supérieur à 5");
} else {
    console.log("x est inférieur ou égal à 5");
}

for (let i = 0; i < 5; i++) {
    console.log(i);
}

let i = 0;
while (i < 5) {
    console.log(i);
    i++;
}
```

Tout sur les Fonctions

Les fonctions peuvent être définies de plusieurs manières: déclarations de fonction, expressions de fonction, et fonctions fléchées.

```
function sayHello() {  
  console.log("Hello!");  
}  
  
const sayGoodbye = function() {  
  console.log("Goodbye!");  
};  
  
const add = (x, y) => x + y;  
  
sayHello(); // Affiche "Hello!"  
sayGoodbye(); // Affiche "Goodbye!"  
console.log(add(5, 3)); // 8
```

Manipulation d'Objets

```
const person = {  
    firstName: "John",  
    lastName: "Doe",  
    greet: function() {  
        console.log("Hello, " + this.firstName);  
    }  
};  
  
console.log(person.firstName); // John  
person.lastName = "Smith";  
person.greet(); // Hello, John
```

Les objets en JavaScript sont des collections de paires clé/valeur. La notation littérale permet de créer des objets, et on accède ou modifie leurs propriétés à l'aide de la notation pointée ou des crochets. Les méthodes d'objet sont des fonctions associées à des objets.

Exploiter les Tableaux

Les tableaux en JavaScript peuvent être manipulés et itérés à l'aide de méthodes telles que `.map()`, `.filter()`, `.reduce()`, et `.forEach()`. Ces méthodes permettent de traiter les éléments d'un tableau de manière efficace et concise.

```
const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(x => x * x);
console.log(squared); // [1, 4, 9, 16, 25]

const even = numbers.filter(x => x % 2 === 0);
console.log(even); // [2, 4]
```

```
numbers.forEach(x => console.log(x));
```

Fonctions Fléchées (=>)

Les fonctions fléchées offrent une syntaxe plus concise pour écrire des fonctions en JavaScript, permettant d'écrire des expressions de fonction plus courtes et directes. Elles sont particulièrement utiles pour les fonctions anonymes et les callbacks.

```
const numbers = [1, 2, 3, 4, 5];
// Exemple avec map utilisant une fonction fléchée
const squared = numbers.map(x => x * x);
console.log(squared); // Affiche [1, 4, 9, 16, 25]

// Fonction fléchée sans argument
const sayHello = () => console.log("Hello!");
sayHello();

// Fonction fléchée avec plusieurs arguments
const add = (a, b) => a + b;
console.log(add(5, 7)); // Affiche 12

// Fonction fléchée avec corps étendu
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
console.log(multiply(2, 3)); // Affiche 6
```

Modularité avec les modules ES6

```
// file: math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// file: app.js
import { add, subtract } from './math.js';
console.log(add(2, 3)); // 5
console.log(subtract(5, 2)); // 3
```

Les modules permettent une meilleure organisation du code en séparant les fonctionnalités en différents fichiers, favorisant la réutilisation et la maintenance.

Promesses et Async/Await

```
// Function to perform a GET request using Fetch and return a promise
function fetchExample(url) {
  return fetch(url)
    .then(response => {
      // Check if the response is OK (status 200)
      if (response.ok) {
        // Return the response in JSON format
        return response.json();
      } else {
        // Throw an error with an appropriate message
        throw new Error(`Error ${response.status}: ${response.statusText}`);
      }
    })
    .then(data => {
      // Resolve the promise with the retrieved data
      return data;
    })
    .catch(error => {
      // Reject the promise with the encountered error
      throw error;
    });
}

// Example of using the fetchExample function
const url = 'https://api.example.com/data';

fetchExample(url)
  .then(data => {
    console.log('Data retrieved successfully:', data);
    // Do something with the retrieved data
  })
  .catch(error => {
    console.error('An error occurred while retrieving data:', error);
    // Handle the error appropriately
  });
}
```

Les promesses et la syntaxe `async/await` simplifient l'écriture de code asynchrone, rendant le traitement des opérations asynchrones plus lisible et facile à gérer.

Déstructuration pour une Meilleure Lisibilité

La destructuration permet d'extraire facilement des données des objets et des tableaux, rendant le code plus propre et plus lisible.

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // John
console.log(age); // 30

const numbers = [1, 2, 3];
const [first, , third] = numbers;
console.log(first); // 1
console.log(third); // 3
```

Paramètres par Défaut, Rest et Spread

Les valeurs par défaut pour les fonctions, l'opérateur Rest pour regrouper les arguments, et l'opérateur Spread pour étaler/expander des éléments offrent plus de flexibilité dans la gestion des données.

```
function greet(name, greeting = "Hello") {
  console.log(`#${greeting}, ${name}!`);
}

greet('John'); // Hello, John!
greet('John', 'Good morning'); // Good morning, John!

const parts = ['shoulders', 'knees'];
const body = ['head', ...parts, 'toes'];
console.log(body); // ["head", "shoulders", "knees", "toes"]
```

Template Literals

Les template literals permettent de construire des chaînes de caractères interpolés avec des expressions, rendant le code plus expressif et facilitant la création de chaînes multilignes.

```
const name = "John";
const greeting = `Hello, ${name}!
How are you today?`;
console.log(greeting);
```

Nouvelles Méthodes pour Objets et Tableaux

```
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.find(x => x > 3)); // 4
console.log(numbers.includes(2)); // true

const person = { name: 'John', age: 30 };
console.log(Object.keys(person)); // ["name", "age"]
console.log(Object.values(person)); // ["John", 30]
```

ES6 et les versions ultérieures ont introduit de nouvelles méthodes pour travailler avec les objets et les tableaux, facilitant la recherche, la transformation, et la vérification des données.

L'Importance de l'Immutabilité

```
// Mauvaise pratique : modification directe d'un objet
let livre = { titre: "Programmation JavaScript" };
livre.titre = "Programmation Avancée"; // Modification directe

// Bonne pratique : immutabilité
const livreImmutable = Object.freeze({ titre: "Programmation JavaScript" });
const nouveauLivre = { ...livreImmutable, titre: "Programmation Avancée" };
```

L'immutabilité est un principe fondamental qui consiste à ne pas modifier directement les données. Au lieu de cela, toute modification produit une nouvelle instance des données. Cela aide à éviter les effets de bord et rend le code plus prévisible et facile à débugger.

Principes de la Programmation Fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui traite les calculs comme l'évaluation de fonctions mathématiques et évite les données changeantes ou mutables. Les principes clés incluent l'utilisation de fonctions pures et la composition de fonctions pour construire des logiciels.

```
// Fonction pure
const ajouter = (x, y) => x + y;

// Composition de fonctions
const multiplierParDeux = x => x * 2;
const ajouterEtMultiplier = (x, y) => multiplierParDeux(ajouter(x, y));

console.log(ajouterEtMultiplier(2, 3)); // 10
```

Stratégies de Gestion des Erreurs

La gestion des erreurs est cruciale pour développer des applications robustes.

L'utilisation de blocs try / catch permet de gérer les exceptions de manière élégante, tandis que les promesses offrent un mécanisme pour traiter les erreurs dans les opérations asynchrones.

```
// Utilisation de try / catch
try {
  const resultat = operationRisquee();
  console.log(resultat);
} catch (erreur) {
  console.error("Une erreur s'est produite:", erreur);
}

// Gestion des erreurs dans les promesses
fetch("https://api.exemple.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(erreur => console.error("Erreur lors de la récupération des données:"))
```

Maîtriser la Gestion des Paquets

```
// package.json avec un script personnalisé
{
  "name": "mon-projet",
  "version": "1.0.0",
  "scripts": {
    "start": "webpack --mode development && node server.js",
    "build": "webpack --mode production"
  },
  "dependencies": {
    "react": "^17.0.1"
  },
  "devDependencies": {
    "webpack": "^5.22.0",
    "babel-loader": "^8.2.2"
  }
}
```

NPM et Yarn sont des gestionnaires de paquets pour JavaScript qui simplifient l'installation, la mise à jour et la gestion des dépendances de projets. Ils permettent également de définir et d'exécuter des scripts personnalisés pour automatiser les tâches de développement courantes.

Linting et Formatage avec ESLint et Prettier

```
// .eslintrc.json
{
  "extends": "eslint:recommended",
  "rules": {
    "no-unused-vars": "warn",
    "eqeqeq": ["error", "always"]
  }
}

// .prettierrc
{
  "semi": false,
  "singleQuote": true
}
```

ESLint et Prettier sont des outils essentiels pour maintenir la qualité du code JavaScript. ESLint analyse le code pour détecter les erreurs et les problèmes de style, tandis que Prettier reformate automatiquement le code selon des règles définies, assurant une cohérence stylistique.

Introduction aux Proxys en JavaScript

Un Proxy est un objet permettant d'intercepter et de redéfinir des opérations fondamentales effectuées sur un objet cible. Ils peuvent être utilisés pour valider, formater ou gérer des accès aux propriétés de l'objet.

```
const cible = {};
const proxy = new Proxy(cible, {});

console.log(proxy); // Proxy {}
```

Handler get dans un Proxy

Le handler get permet d'intercepter les accès aux propriétés de l'objet cible. On peut personnaliser le comportement lors de la lecture des propriétés.

```
const cible = { message: "Salut" };
const handler = {
  get: (target, property) => {
    return property in target ? target[property] : "Propriété inexiste";
  }
};

const proxy = new Proxy(cible, handler);
console.log(proxy.message); // Salut
console.log(proxy.inexistant); // Propriété inexiste
```

Handler set dans un Proxy

Le handler set permet d'intercepter les opérations d'écriture sur les propriétés de l'objet cible. On peut ainsi valider ou modifier les valeurs avant qu'elles ne soient assignées.

```
const cible = {};
const handler = {
  set: (target, property, value) => {
    if (typeof value === "number") {
      target[property] = value;
      return true;
    } else {
      console.log("Seuls les nombres sont acceptés");
      return false;
    }
  }
};

const proxy = new Proxy(cible, handler);
proxy.age = 30; // OK
proxy.nom = "Jean"; // Seuls les nombres sont acceptés
console.log(proxy); // { age: 30 }
```

Handler has dans un Proxy

Le handler has permet d'intercepter les opérations de test de propriétés avec l'opérateur in.

```
const cible = { visible: true };
const handler = {
  has: (target, property) => {
    if (property === "secret") {
      return false;
    }
    return property in target;
  }
};

const proxy = new Proxy(cible, handler);
console.log("visible" in proxy); // true
console.log("secret" in proxy); // false
```

Nouveautés Javascript

ECMAScript 6 (ES6) - 2015

Apport majeur :

Classes : Introduction de la syntaxe de classe pour la programmation orientée objet, qui est plus propre et facile à utiliser.

Modules : Standardisation du support des modules ES6 pour une meilleure gestion du code en facilitant l'import et l'export de composants.

Promesses : Pour la gestion asynchrone, facilitant l'écriture de code asynchrone et la gestion des opérations asynchrones.

Nouveautés Javascript

Autres apports :

let et const : Nouveaux mots-clés pour les déclarations de variables, offrant des portées de bloc, réduisant ainsi les erreurs communes dues aux variables globales.

Fonctions fléchées : Syntaxe plus concise pour l'écriture de fonctions, et ne crée pas de nouveau contexte this.

Paramètres par défaut, Rest et Spread : Amélioration de la gestion des paramètres de fonctions.

Destructuration : Permet une affectation plus facile et plus claire des données à partir d'arrays ou d'objets.

Templates littéraux : Intégration facile des variables dans les chaînes de caractères avec les templates littéraux.

ECMAScript 7 (ES7) - 2016

Opérateur d'exponentiation (**): Pour calculer la puissance d'un nombre.

Méthode Array.prototype.includes : Vérifie si un tableau inclut un élément donné, renvoyant true ou false.

ECMAScript 8 (ES8) - 2017

async et await : Simplification de l'écriture de fonctions asynchrones pour rendre le code asynchrone aussi facile à lire et à écrire que le code synchrone.

Méthodes de l'objet : Object.values(), Object.entries(), et Object.getOwnPropertyDescriptors() pour une meilleure manipulation des objets.

ECMAScript 9 (ES9) - 2018

Opérateur de décomposition pour objets : Étend les fonctionnalités de décomposition aux objets.

Promesses finally() : Méthode ajoutée aux promesses pour exécuter du code une fois que la promesse est réglée, indépendamment du résultat.

ECMAScript 10 (ES10) - 2019

`Array.prototype.{flat, flatMap}` : Méthodes pour aplatiser des tableaux imbriqués et appliquer une fonction, puis aplatisir le résultat.

`Object.fromEntries()` : Transforme une liste de paires clé-valeur en un objet.

Chaînes de caractères et modifications de `Array.sort` : Améliorations mineures pour la manipulation de chaînes de caractères et la méthode `sort`.

ECMAScript 11 (ES11) - 2020

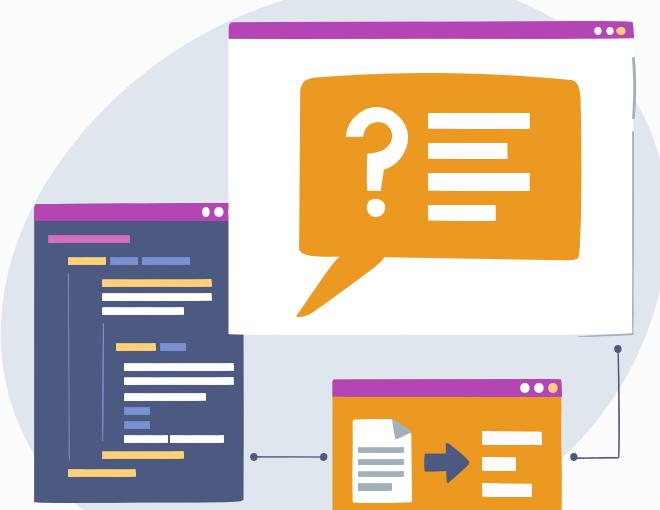
BigInt : Introduit un type pour représenter des entiers très grands.

Promise.allSettled : Une nouvelle méthode de promesse qui renvoie un tableau de résultats après que toutes les promesses données se soient résolues ou rejetées.

Dynamique import() : Importations de modules sur demande pour améliorer la performance du chargement de modules.

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 `npx create-expo-app --template` 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

```
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>
);
```

```
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" />
  </>
);

export default MyStatusBar;
```

- 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: Déetecter la plateforme et adapter le rendu

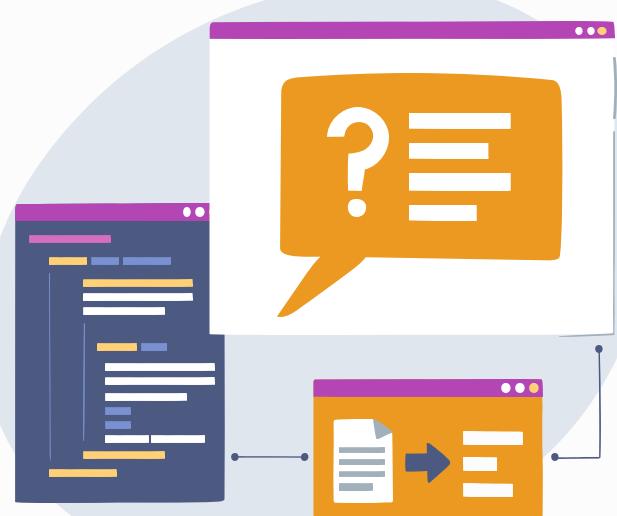
- Le module Platform permet de détecter la plateforme sur laquelle l'application est exécutée (iOS ou Android).
- Vous pouvez utiliser cette information pour adapter le rendu et le comportement de votre application en fonction de la plateforme.

```
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 windowHeight = Dimensions.get('window').width;

const App = () => (
  <View style={{ width: windowHeight * 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: Déetecte si l'appareil est en mode paysage.

isPortrait: Déetecte si l'appareil est en mode portrait.

Création d'une page d'accueil responsive

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

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>
  );
};
```

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>
  );
};
```

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>
  );
};
```

Navigation imbriquée

Créer des navigateurs imbriqués pour une navigation plus complexe.

```
function RootNavigator() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Main" component={MainTabNavigator} />
      <Stack.Screen name="Profile" component={ProfileScreen} />
    </Stack.Navigator>
  );
}
```

MainTabNavigator

Créer le composant MainTabNavigator pour la navigation imbriquée.

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

Navigation entre les navigateurs

Naviguer entre différents navigateurs imbriqués.

```
<Button  
    title="Go to Profile"  
    onPress={() => navigation.navigate('Profile')}  
/>
```

Header personnalisé

Créer un en-tête personnalisé pour les écrans.

```
function CustomHeader() {
  return (
    <View style={{ height: 60, backgroundColor: 'blue' }}>
      <Text style={{ color: 'white' }}>Custom Header</Text>
    </View>
  );
}

<Stack.Screen
  name="Home"
  component={HomeScreen}
  options={{ header: () => <CustomHeader /> }}
/>
```

Utilisation des icônes dans Tab Navigator

Ajouter des icônes aux onglets du Tab Navigator.

```
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import Ionicons from 'react-native-vector-icons/Ionicons';

const Tab = createBottomTabNavigator();

function MyTabs() {
  return (
    <Tab.Navigator
      screenOptions={({ route }) => ({  
        tabBarIcon: ({ color, size }) => {  
          let iconName;  
  
          if (route.name === 'Home') {  
            iconName = 'ios-home';  
          } else if (route.name === 'Settings') {  
            iconName = 'ios-settings';  
          }  
  
          return <Ionicons name={iconName} size={size} color={color} />;  
        },  
      })}
    >
      <Tab.Screen name="Home" component={HomeScreen} />
      <Tab.Screen name="Settings" component={SettingsScreen} />
    </Tab.Navigator>
```

Custom Drawer Content

Créer un contenu personnalisé pour le Drawer.

```
function CustomDrawerContent(props) {
  return (
    <DrawerContentScrollView {...props}>
      <DrawerItemList {...props} />
      <DrawerItem label="Help" onPress={() => alert('Link to help')} />
    </DrawerContentScrollView>
  );
}

<Drawer.Navigator drawerContent={props => <CustomDrawerContent {...props} />}>
  <Drawer.Screen name="Home" component={HomeScreen} />
</Drawer.Navigator>
```

Accessibilité et navigation

Assurer que la navigation est accessible, notamment pour les utilisateurs utilisant des technologies d'assistance.

```
import { AccessibilityInfo, findNodeHandle } from 'react-native';

function HomeScreen() {
  const ref = useRef(null);

  useEffect(() => {
    const handle = findNodeHandle(ref.current);
    AccessibilityInfo.setAccessibilityFocus(handle);
  }, []);

  return (
    <View>
      <Text ref={ref} accessibilityLabel="Home Screen">Home Screen</Text>
    </View>
  );
}
```

Gestion des erreurs et des exceptions de navigation

Stratégies pour gérer et récupérer des erreurs de navigation.

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

function App() {
  return (
    <NavigationContainer
      onStateChange={(state) => console.log('New state is', state)}
      onUnhandledAction={(action) => {
        console.warn('Unhandled action', action);
     }}
    >
      {/* Navigators */}
      </NavigationContainer>
    );
}

}
```

Navigation entre différentes piles de stack

Techniques pour passer d'une pile de stack à une autre sans perdre l'état de la première pile.

```
function navigateToDifferentStack(navigation, stackName, screenName) {  
  navigation.navigate(stackName, {  
    screen: screenName,  
    params: { someParam: 'someValue' },  
  });  
}  
  
// Utilisation  
navigateToDifferentStack(navigation, 'ProfileStack', 'ProfileScreen');
```

Persistance de l'état de navigation

Sauvegarder et restaurer l'état de navigation lors du redémarrage de l'application.

```
const PERSISTENCE_KEY = 'NAVIGATION_STATE';

function App() {
  const [initialState, setInitialState] = useState();

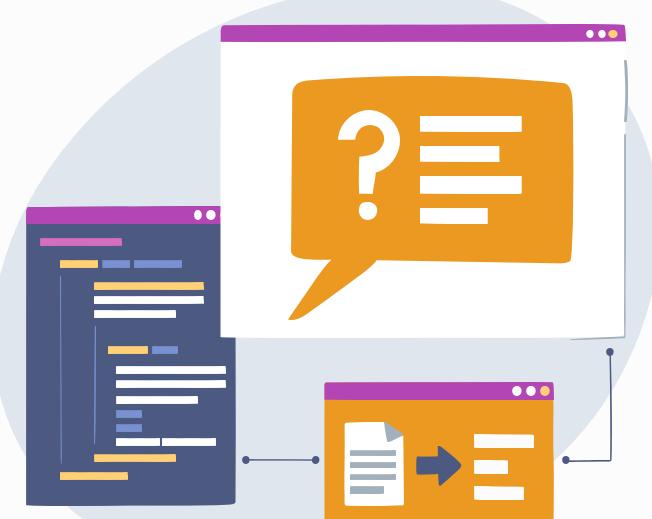
  useEffect(() => {
    const restoreState = async () => {
      const savedStateString = await AsyncStorage.getItem(PERSISTENCE_KEY);
      const state = savedStateString ? JSON.parse(savedStateString) : undefined;
      setInitialState(state);
    };

    if (!initialState) {
      restoreState();
    }
  }, [initialState]);

  return (
    <NavigationContainer
      initialState={initialState}
      onStateChange={(state) => AsyncStorage.setItem(PERSISTENCE_KEY, JSON.stringify(state))
    }
    /* Navigators */
  );
}
```

04.1

Expo Router



Introduction à Expo Router

Expo Router est une bibliothèque de routage open-source pour les applications React Native universelles, utilisant une approche basée sur les fichiers pour la navigation.

Créer des pages avec Expo Router

- Lorsqu'un fichier est créé dans le répertoire **de l'application**, il devient automatiquement un itinéraire dans l'application. Par exemple, les fichiers suivants créeront les routes suivantes :



Itinéraires dynamiques

- Les itinéraires dynamiques correspondent à n'importe quel chemin sans correspondance à un niveau de segment donné.

app/blog/[slug].js

/blog/123

app/blog/[...rest].js

/blog/123/settings

```
import { useLocalSearchParams } from 'expo-router';
import { Text } from 'react-native';

export default function Page() {
  const { slug } = useLocalSearchParams();

  return <Text>Blog post: {slug}</Text>;
}
```

Navigation

- Dans l'exemple suivant, deux <Link/> composants naviguent vers des itinéraires différents.

```
import { View } from 'react-native';
import { Link } from 'expo-router';

export default function Page() {
  return (
    <View>
      <Link href="/about">About</Link>
      {/* ...other links */}
      <Link href="/user/bacon">View user</Link>
    </View>
  );
}

import { router } from 'expo-router';

export function logout() {
  router.replace('/login');
}
```

Gestion des Layouts

```
import { Slot } from 'expo-router';

export default function HomeLayout() {
  return <Slot />;
}
```

app/home/_layout.js

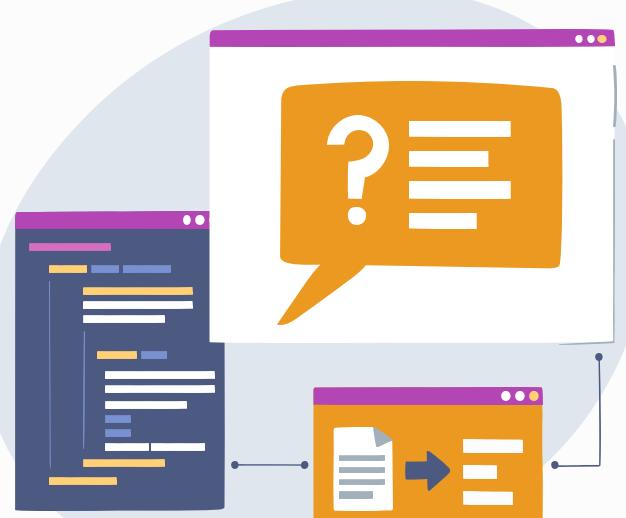
```
import { Slot } from 'expo-router';

export default function HomeLayout() {
  return (
    <>
      <Header />
      <Slot />
      <Footer />
    </>
  );
}
```

Expo prend en charge les Layouts.

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

```
// 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 imbriquations profondes et utiliser des noms clairs
```

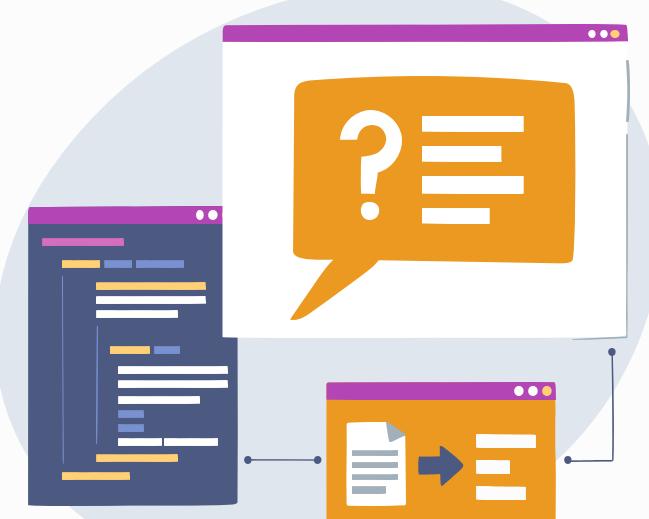
L'Importance du JSX et de la Composition

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>;
}

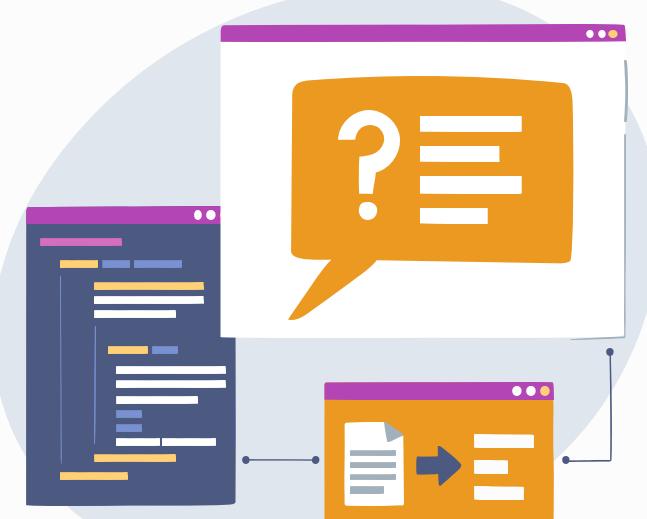
// Définition des propTypes et defaultProps
Welcome.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number
};

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);
```

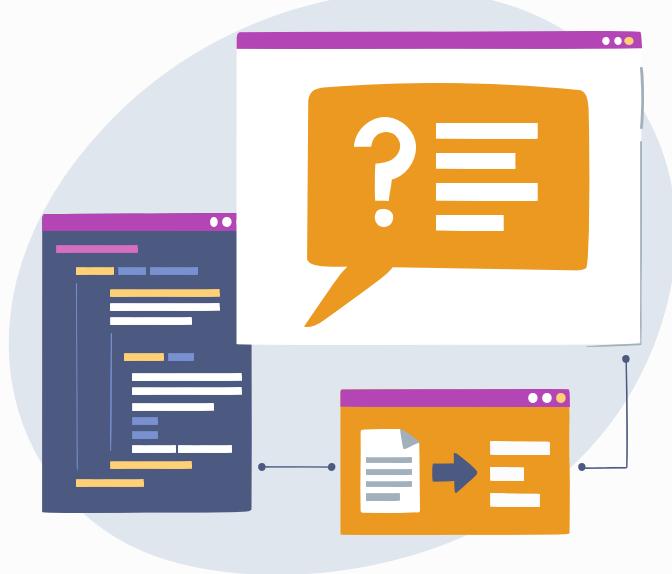
Accès aux Éléments DOM avec `useRef`

`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

```
// 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);
```

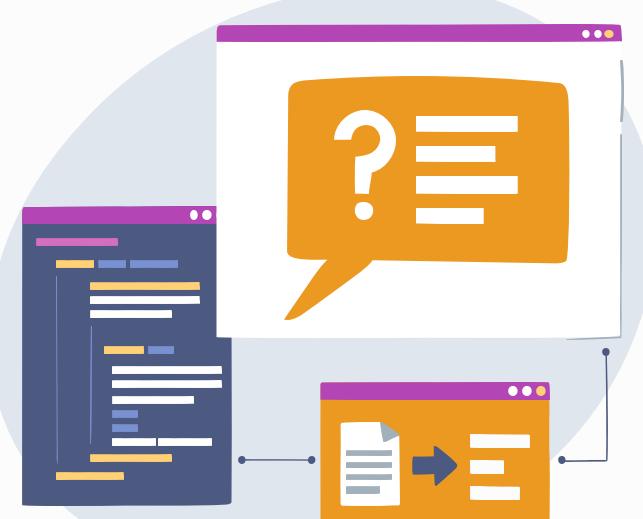
reduce accumule les valeurs d'un tableau en une seule valeur.

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 développabilité 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/${userId}`);
    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

```
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 useSelector et useDispatch 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';

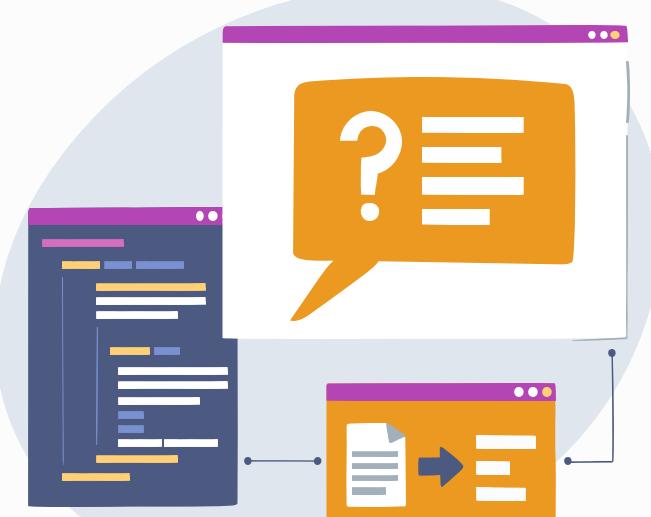
const useStore = create(persist(set => ({
  user: null,
  setUser: user => set({ user }),
}), {
  name: 'user-settings', // nom de la clé du local storage
}));
```

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.

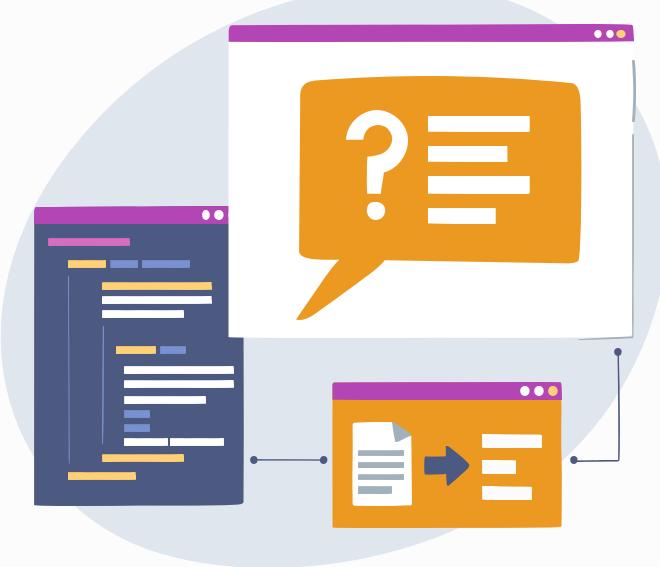
```
const value = useContext(MyContext);
```

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);
```

12.1

Intégrations de quelque fonctionnalités natives (expo)



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.

12.2

Intégrations de quelque fonctionnalités natives (RN)



Particularités d'iOS

Les spécificités du développement sur iOS, comme l'utilisation de Xcode, le simulateur iOS, et la gestion des permissions.

```
// Exemple de configuration spécifique à iOS dans App.js
import { Platform } from 'react-native';

if (Platform.OS === 'ios') {
  // Code spécifique à iOS
}
```

Particularités d'Android

Les spécificités du développement sur Android, comme l'utilisation d'Android Studio, l'émulateur Android, et la gestion des permissions.

```
// Exemple de configuration spécifique à Android dans App.js
import { Platform } from 'react-native';

if (Platform.OS === 'android') {
    // Code spécifique à Android
}
```

L'API Geolocation

Comment utiliser l'API Geolocation pour obtenir la localisation d'un utilisateur.

```
import React, { useState, useEffect } from 'react';
import { Text, View } from 'react-native';
import Geolocation from '@react-native-community/geolocation';

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

  useEffect(() => {
    Geolocation.getCurrentPosition(
      (position) => {
        const { latitude, longitude } = position.coords;
        setLocation({ latitude, longitude });
      },
      (error) => console.log(error),
      { enableHighAccuracy: true, timeout: 20000, maximumAge: 1000 }
    );
  }, []);

  return (
    <View>
      {location ? (
        <Text>Latitude: {location.latitude}, Longitude: {location.longitude}</Text>
      ) : (
        <Text>Obtention de la localisation...</Text>
      )}
    </View>
  );
};

export default App;
```

Gestion des Permissions

Comment gérer les permissions dans une application React Native pour accéder aux fonctionnalités natives comme la localisation et la caméra.

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

const requestCameraPermission = async () => {
  try {
    const granted = await PermissionsAndroid.request(
      PermissionsAndroid.PERMISSIONS.CAMERA,
      {
        title: "Permission d'utiliser la caméra",
        message: "Nous avons besoin de votre permission pour utiliser la caméra",
        buttonNeutral: "Plus tard",
        buttonNegative: "Annuler",
        buttonPositive: "OK",
      }
    );
    if (granted === PermissionsAndroid.RESULTS.GRANTED) {
      console.log("Vous pouvez utiliser la caméra");
    } else {
      console.log("Permission refusée");
    }
  } catch (err) {
    console.warn(err);
  }
};

requestCameraPermission();
```

Utiliser CameraRoll

Accéder aux photos et à la caméra du mobile en utilisant CameraRoll.

```
import React, { useEffect, useState } from 'react';
import { View, Button, Image } from 'react-native';
import { launchCamera, launchImageLibrary } from 'react-native-image-picker';

const App = () => {
  const [photo, setPhoto] = useState(null);

  const openCamera = () => {
    launchCamera({ mediaType: 'photo' }, (response) => {
      if (response.assets) {
        setPhoto(response.assets[0].uri);
      }
    });
  };

  return (
    <View>
      <Button title="Prendre une photo" onPress={openCamera} />
      {photo && <Image source={{ uri: photo }} style={{ width: 200, height: 200 }} />}
    </View>
  );
};

export default App;
```

Utilisation Avancée de CameraRoll

Techniques avancées pour utiliser CameraRoll, comme le choix de plusieurs images.

```
import React from 'react';
import { Button } from 'react-native';
import { launchImageLibrary } from 'react-native-image-picker';

const App = () => {
  const openImageLibrary = () => {
    launchImageLibrary({ selectionLimit: 0 }, (response) => {
      console.log(response.assets);
    });
  };

  return (
    <Button title="Choisir des photos" onPress={openImageLibrary} />
  );
};

export default App;
```

Affichage de la Galerie Photo

Comment afficher une galerie de photos en utilisant React Native.

```
import React, { useEffect, useState } from 'react';
import { View, Image, ScrollView } from 'react-native';
import CameraRoll from '@react-native-community/cameraroll';

const App = () => {
  const [photos, setPhotos] = useState([]);

  useEffect(() => {
    CameraRoll.getPhotos({
      first: 20,
      assetType: 'Photos',
    }).then(r => {
      setPhotos(r.edges);
    }).catch((err) => {
      console.log(err);
    });
  }, []);

  return (
    <ScrollView>
      {photos.map((p, i) => (
        <Image key={i} style={{ width: 100, height: 100 }} source={{ uri: p.node.image }}>
      ))}
    </ScrollView>
  );
};

export default App;
```

Introduction aux animations avec React Native

Découvrez comment animer vos applications React Native pour améliorer l'expérience utilisateur avec des transitions fluides et des effets visuels captivants.

Introduction aux animations avec React Native

Découvrez comment animer vos applications React Native pour améliorer l'expérience utilisateur avec des transitions fluides et des effets visuels captivants.

Introduction aux animations avec React Native

Découvrez comment animer vos applications React Native pour améliorer l'expérience utilisateur avec des transitions fluides et des effets visuels captivants.

Introduction aux animations avec React Native

Découvrez comment animer vos applications React Native pour améliorer l'expérience utilisateur avec des transitions fluides et des effets visuels captivants.

13

Animations avec React Native



Introduction aux animations avec React Native

Découvrez comment animer vos applications React Native pour améliorer l'expérience utilisateur avec des transitions fluides et des effets visuels captivants.

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

const FadeInView = (props) => {
  const fadeAnim = new Animated.Value(0); // Opacité initiale

  React.useEffect(() => {
    Animated.timing(fadeAnim, {
      toValue: 1,
      duration: 1000,
    }).start();
  }, [fadeAnim]);

  return (
    <Animated.View
      style={{
        ...props.style,
        opacity: fadeAnim,
      }}
    >
      {props.children}
    </Animated.View>
  );
};
```

Comprendre l'API Animated

- Explorez les fondamentaux de l'API Animated, essentielle pour créer des animations complexes et dynamiques dans React Native.

```
const FadeInView = () => {
  const fadeAnim = useRef(new Animated.Value(0)).current; // Opacité initiale à 0

  useEffect(() => {
    Animated.timing(fadeAnim, {
      toValue: 1,
      duration: 1000,
      useNativeDriver: true
    }).start();
  }, [fadeAnim]);

  return (
    <Animated.View
      style={[
        opacity: fadeAnim,
        width: '100%',
        height: 100,
        backgroundColor: 'blue'
      ]}
    />
  );
};
```

Principes de base des animations fluides

Apprenez à maintenir une haute performance et fluidité dans vos animations, en évitant les saccades et latences.

```
const SmoothScaleAnimation = () => {
  const scaleAnim = useRef(new Animated.Value(0)).current; // Échelle initiale à 0

  useEffect(() => {
    Animated.spring(scaleAnim, {
      toValue: 1,
      useNativeDriver: true // Utilisation du driver natif pour de meilleures performances
    }).start();
  }, [scaleAnim]);

  return (
    <Animated.View
      style={{
        transform: [{ scale: scaleAnim }],
        width: 100,
        height: 100,
        backgroundColor: 'red'
      }}
    />
  );
};
```

Techniques avancées avec l'API Animated

Approfondissez des techniques plus avancées telles que le chaînage, la parallélisation et la composition d'animations.

```
const CombinedAnimations = () => {
  const opacity = useRef(new Animated.Value(0)).current;
  const translateY = useRef(new Animated.Value(-100)).current;

  useEffect(() => {
    Animated.parallel([
      Animated.timing(opacity, {
        toValue: 1,
        duration: 1000,
        useNativeDriver: true
      }),
      Animated.timing(translateY, {
        toValue: 0,
        duration: 1000,
        useNativeDriver: true
      })
    ]).start();
  }, [opacity, translateY]);

  return (
    <Animated.View
      style={{
        opacity,
        transform: [{ translateY }],
        width: '100%',
        height: 100,
        backgroundColor: 'purple'
      }}
    />
  );
};
```

Introduction à Reanimated

Découvrez Reanimated, une librairie puissante pour des animations encore plus fluides et performantes.

```
import { useSharedValue, withTiming, Easing } from 'react-native-reanimated';

const opacity = useSharedValue(0);

useEffect(() => {
  opacity.value = withTiming(1, { duration: 1000, easing: Easing.out(Easing.exp) });
}, []);
```

Cas pratiques d'utilisation de Reanimated

Examinez des exemples concrets d'utilisation de Reanimated dans des applications React Native.

```
import React, { useEffect } from 'react';
import Animated, { useSharedValue, useAnimatedStyle, withTiming, Easing }

const FadeInExample = () => {
  const opacity = useSharedValue(0); // Opacité initiale à 0

  useEffect(() => {
    opacity.value = withTiming(1, {
      duration: 1000,
      easing: Easing.inOut(Easing.quad)
    });
  }, []);

  const animatedStyle = useAnimatedStyle(() => {
    return {
      opacity: opacity.value
    };
  });

  return (
    <Animated.View
      style={[
        animatedStyle,
        {
          width: 200,
          height: 200,
          backgroundColor: 'blue'
        }
      ]}
    />
  );
}
```

Utilisation des interpolations avec Reanimated

Apprenez à utiliser les interpolations pour transformer des valeurs d'entrée complexes en animations fluides.

```
rt { interpolate, Extrapolate } from 'react-native-reanimated';

t interpolatedValue = interpolate(opacity.value, [0, 1], [0, 100], Extrapolate.CLAMP);
```

Introduction à React Native Skia

Découvrez Skia pour React Native, une bibliothèque puissante pour dessiner des graphiques 2D.

```
// Exemple de code pour créer un cercle avec Skia
import { Skia, useDrawCallback } from '@shopify/react-native-skia';

const drawCircle = useDrawCallback((canvas, paint) => {
  paint.setColor(Skia.Color("blue"));
  canvas.drawCircle(50, 50, 40, paint);
}, []);
```

Fonctionnalités clés de React Native Skia

Explorez les principales fonctionnalités de Skia, y compris le rendu vectoriel, les filtres d'image, et la gestion des chemins et des formes complexes pour des graphiques 2D riches.

```
import { Skia, usePaint } from '@shopify/react-native-skia';

const paint = usePaint(paint => {
  paint.setAntiAlias(true);
  paint.setColor(Skia.Color('red'));
  paint.setStyle(Skia.PaintStyle.Fill);
});
```

Créer des animations 2D performantes avec Skia

Découvrez comment utiliser Skia pour développer des animations 2D qui combinent performance et qualité visuelle.

```
import { Skia, useState, useDrawCallback } from '@shopify/react-native-skia';

const rotation = useState(0);

const draw = useDrawCallback((canvas, paint) => {
  canvas.clear(Skia.Color('white'));
  canvas.save();
  canvas.rotate(rotation.current, { x: 100, y: 100 });
  canvas.drawRect(Skia.XYWHRect(50, 50, 100, 100), paint);
  canvas.restore();
}, []);
```

Enrichir les interfaces utilisateur avec Skia

Utilisez Skia pour ajouter des éléments graphiques sophistiqués et des animations à vos interfaces, rendant l'interaction plus intuitive et attrayante.

```
import { SkiaView, Skia, usePaint, useFont } from '@shopify/react-native-skia';

const paint = usePaint((p) => {
  p.setColor(Skia.Color('blue'));
  p.setAntiAlias(true);
});

const MySkiaComponent = () => (
  <SkiaView style={{ flex: 1 }}>
    <Rect x={10} y={10} width={100} height={50} paint={paint} />
    <Text x={120} y={35} text="Hello Skia" paint={paint} />
  </SkiaView>
);
```

Exemples d'animations 2D avec Skia

Voir des exemples pratiques d'animations 2D réalisées avec Skia, montrant la diversité et la flexibilité de cette technologie.

```
import { Skia, useTiming, Rect, usePaint } from '@shopify/react-native-skia';

const paint = usePaint(paint => {
  paint.setColor(Skia.Color('green'));
  paint.setStyle(Skia.PaintStyle.Fill);
});

const AnimatedRect = () => {
  const progress = useTiming({ duration: 1000, loop: true });
  const x = Skia.interpolate(progress, [0, 1], [0, 200]);

  return <Rect x={x} y={50} width={100} height={100} paint={paint} />;
};
```

Gestion des événements dans les animations

Skia

Apprenez à gérer les interactions utilisateur avec les éléments graphiques Skia pour des réponses dynamiques et interactives.

```
import { Skia, TouchEvent, useTouchHandler } from '@shopify/react-native-skia';

const touchHandler = useTouchHandler({
  onStart: (pt) => console.log("Touch started at:", pt),
  onActive: (pt) => console.log("Finger moved to:", pt),
  onEnd: () => console.log("Touch ended")
});

<SkiaView onTouch={touchHandler} />
```

InteractionManager

14



Introduction à InteractionManager

InteractionManager, un outil essentiel pour améliorer la réactivité des applications React Native lors de l'exécution de tâches lourdes.

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

const expensiveTask = () => {
  // Logique de calcul complexe
};

InteractionManager.runAfterInteractions(() => {
  expensiveTask();
});
```

Comprendre le rôle de InteractionManager

L'InteractionManager permet de retarder l'exécution de tâches longues jusqu'à ce que toutes les interactions en cours soient terminées, garantissant ainsi une interface utilisateur fluide.

```
InteractionManager.runAfterInteractions(() => {
    // Cette tâche s'exécutera seulement après que les animations en cours soient terminées
    performHeavyComputation();
});
```

Utilisation pratique de InteractionManager

Découvrez comment utiliser InteractionManager dans des scénarios réels pour améliorer la fluidité de l'interface utilisateur lors de chargements lourds ou calculs complexes.

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

const loadData = async () => {
  await fetchLargeDataset(); // Simulation de chargement de données volumineuses
};

const App = () => {
  useEffect(() => {
    const task = InteractionManager.runAfterInteractions(() => {
      loadData();
    });

    return () => task.cancel();
  }, []);
}

return <View><Text>Chargement des données...</Text></View>;
};
```

Techniques de programmation avec InteractionManager

Techniques avancées pour intégrer InteractionManager dans votre flux de développement pour optimiser les performances de tâches asynchrones.

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

const performTasks = async () => {
  const data = await fetchData();
  processHeavyData(data);
};

InteractionManager.runAfterInteractions(() => {
  performTasks();
});
```

Gérer les tâches longues sans bloquer l'interface utilisateur

Stratégies pour gérer efficacement les tâches de longue durée en utilisant InteractionManager, permettant une meilleure expérience utilisateur même pendant les opérations lourdes.

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

const longRunningTask = () => {
  // Processus complexe
};

// Utilisation de InteractionManager pour démarrer la tâche sans bloquer l'UI
InteractionManager.runAfterInteractions(() => {
  longRunningTask();
});
```

Exemples de code : InteractionManager en action

Illustration par des exemples concrets de l'utilisation de InteractionManager pour gérer des tâches en arrière-plan tout en maintenant l'interface réactive.

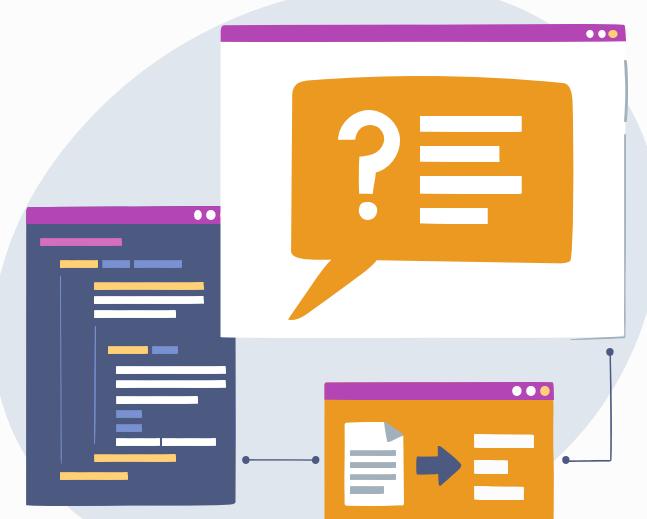
```
import { InteractionManager } from 'react-native';

const updateUIAfterHeavyTask = () => {
  // Mise à jour de l'UI après une tâche lourde
};

InteractionManager.runAfterInteractions(() => {
  heavyComputation();
  updateUIAfterHeavyTask();
});
```

15

Intégration de charts



Installation et configuration de React Native Chart Kit

Installer et configurer React Native Chart Kit dans votre projet React Native.

```
npm install react-native-chart-kit react-native-svg
```

```
import { LineChart } from 'react-native-chart-kit';
```

Créer un graphique à barres

Apprenez à créer un graphique à barres attrayant pour visualiser des données de manière efficace.

```
import { BarChart } from 'react-native-chart-kit';

const data = {
  labels: ['January', 'February', 'March', 'April'],
  datasets: [
    {
      data: [20, 45, 28, 80]
    }
  ]
};

const chartConfig = {
  backgroundColor: '#e26a00',
  backgroundGradientFrom: '#fb8c00',
  backgroundGradientTo: '#ffa726',
  color: (opacity = 1) => `rgba(255, 255, 255, ${opacity})`
};

<BarChart
  data={data}
  width={screenWidth}
  height={220}
  yAxialLabel="$"
  chartConfig={chartConfig}
  verticalLabelRotation={30}
/>
```

Implémenter un graphique linéaire

Création d'un graphique linéaire, idéal pour tracer des tendances et des changements au fil du temps.

```
import { LineChart } from 'react-native-chart-kit';

const data = {
    labels: ['January', 'February', 'March', 'April'],
    datasets: [
        {
            data: [30, 70, 100, 90]
        }
    ];
}

<LineChart
    data={data}
    width={screenWidth}
    height={256}
    chartConfig={chartConfig}
/>
```

16

Utilisation Tanstack Query



Démarrer avec React Query v5

```
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const App = () => {
  const queryClient = new QueryClient();

  return (
    <QueryClientProvider client={queryClient}>
      <MyComponent />
    </QueryClientProvider>
  );
};
```

Ce module vous guide à travers l'installation et la configuration de React Query v5 dans votre projet React. Vous apprendrez à installer les packages nécessaires, à configurer la bibliothèque et à comprendre les concepts fondamentaux.

Comprendre les concepts clés de React Query

Ce module explore les concepts fondamentaux de React Query, tels que les requêtes, les mutations, l'état des données, l'invalidation du cache et les hooks de base.

```
const { data, status, error } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
});

if (status === 'loading') {
  return <div>Loading...</div>;
}

if (error) {
  return <div>An error occurred.</div>;
}

return (
  <ul>
    {data.map(todo => (
      <li key={todo.id}>{todo.title}</li>
    )));
  </ul>
);
```

Gérer les erreurs et les chargements avec React Query

```
const { data, status, error, isFetching } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
});

if (status === 'loading') {
  return <div>Loading...</div>;
}

if (error) {
  return <div>An error has occurred: {error.message}</div>;
}

return (
  <ul>
    {data.map(todo => (
      <li key={todo.id}>{todo.title}</li>
    )));
  </ul>
);
```

Ce module vous montre comment gérer les erreurs et les chargements dans vos applications React Query. Vous apprendrez à afficher des messages d'erreur, à gérer les tentatives et à utiliser des statuts de chargement.

Optimiser les performances avec React Query

Ce module vous montre comment optimiser les performances de vos applications React Query. Vous apprendrez à utiliser le cache de données, à effectuer des requêtes sélectives et à mettre en place la pagination.

```
const { data, status, error } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
  options: {
    cacheTime: 10000, // Cache data for 10 seconds
    staleTime: 5000, // Allow data to be stale for 5 seconds
  },
});
```

Introduction aux Mutations avec Tanstack Query

Les mutations sont utilisées dans Tanstack Query pour effectuer des modifications de données comme les insertions, mises à jour, ou suppressions.

```
import { useMutation } from '@tanstack/react-query';

const createItem = async (item) => {
  const response = await fetch('/api/items', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  return response.json();
};

function AddItem() {
  const mutation = useMutation(createItem);
  return <button onClick={() => mutation.mutate({ name: 'New Item' })}>Add Item</button>;
}
```

Base des Mutations

Apprendre à utiliser useMutation pour créer, mettre à jour ou supprimer des données.

```
import { useMutation } from '@tanstack/react-query';

const updateItem = async (item) => {
  return fetch(`/api/items/${item.id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  }).then(res => res.json());
};

function EditItem({ item }) {
  const { mutate } = useMutation(updateItem, {
    onSuccess: () => alert('Item updated successfully!')
  });

  return <button onClick={() => mutate({ ...item, name: 'Updated Name' })}>Update Item</button>;
}
```

Gestion des États de Mutation

Gestion des états de mutation pour afficher les retours visuels appropriés tels que les chargements, les succès et les erreurs.

```
import { useMutation } from '@tanstack/react-query';

const deleteItem = id => fetch(`/api/items/${id}`, { method: 'DELETE' });

function DeleteItem({ itemId }) {
  const { mutate, isLoading, isError, error } = useMutation(() => deleteItem(itemId), {
    onSuccess: () => alert('Item deleted successfully!'),
    onError: () => alert('Error deleting item!')
  });

  if (isLoading) return <div>Deleting...</div>;
  if (isError) return <div>Error: {error.message}</div>;

  return <button onClick={() => mutate()}>Delete Item</button>;
}
```

Synchronisation avec le Cache après une Mutation

Comment utiliser les mutations pour non seulement modifier les données, mais aussi synchroniser ces modifications avec le cache local.

```
import { useMutation, useQueryClient } from '@tanstack/react-query';

const addItem = async (item) => {
  const response = await fetch('/api/items', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  return response.json();
};

function AddItemToList() {
  const queryClient = useQueryClient();
  const mutation = useMutation(addItem, {
    onSuccess: () => {
      queryClient.invalidateQueries(['items']);
    }
  });

  return <button onClick={() => mutation.mutate({ name: 'New Item' })}>Add to List</button>;
}
```

Utilisation des Rollbacks en cas d'Échec

Implémenter des rollbacks pour restaurer l'état précédent en cas d'échec de la mutation.

```
const updateItem = async (item) => {
  const response = await fetch(`api/items/${item.id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  if (!response.ok) {
    throw new Error('Failed to update item');
  }
  return response.json();
};

function UpdateItem({ item }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(updateItem, {
    onMutate: async (newItem) => {
      await queryClient.cancelQueries(['items']);
      const previousItems = queryClient.getQueryData(['items']);
      queryClient.setQueryData(['items'], old => old.map(d => d.id === newItem.id ? newItem : d));
      return { previousItems };
    },
    onError: (err, newItem, context) => {
      queryClient.setQueryData(['items'], context.previousItems);
    },
    onSettled: () => {
      queryClient.invalidateQueries(['items']);
    }
  });
  return <button onClick={() => mutate({ ...item, name: 'Updated Name' })}>Update Item</button>;
}
```

Requêtes paginées

Les requêtes paginées sont utilisées pour charger les données par page, ce qui réduit la charge sur le serveur et améliore l'expérience utilisateur en chargeant les données progressivement.

```
const fetchProjects = async (page = 0) => {
  const response = await fetch(`/api/projects?page=${page}`);
  return response.json();
};

function PaginatedProjects() {
  const [page, setPage] = useState(0);
  const { data: projects, isLoading, isError, error } = useQuery({
    queryKey: ['projects', page],
    queryFn: () => fetchProjects(page),
    keepPreviousData: true,
  });

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error: {error.message}</div>;

  return (
    <div>
      {projects.map(project => <div key={project.id}>{project.name}</div>)}
      <button onClick={() => setPage(old => old + 1)}>Next Page</button>
    </div>
  );
}
```



Introduction à useInfiniteQuery

Configurer useInfiniteQuery nécessite de spécifier comment récupérer les données et comment identifier la prochaine page à charger.

```
import { useInfiniteQuery } from '@tanstack/react-query';

const fetchPosts = async ({ pageParam = 1 }) => {
  const response = await fetch(`/api/posts?page=${pageParam}`);
  return response.json();
};

function InfinitePosts() {
  const { data, fetchNextPage, hasNextPage } = useInfiniteQuery(['posts'], fetchPosts, {
    getNextPageParam: lastPage => lastPage.nextPage ?? undefined
  });

  return (
    <div>
      {data.pages.map((page, i) => (
        <React.Fragment key={i}>
          {page.data.map(post => <p key={post.id}>{post.title}</p>)}
        </React.Fragment>
      ))}
      {hasNextPage && <button onClick={() => fetchNextPage()}>Load More</button>}
    </div>
  );
}
```



Configurer useInfiniteQuery

Configurer useInfiniteQuery nécessite de spécifier comment récupérer les données et comment identifier la prochaine page à charger.

```
function Products() {
  const {
    data,
    error,
    fetchNextPage,
    hasNextPage,
    isLoading,
    isFetchingNextPage
  } = useInfiniteQuery(['products'], fetchProducts, {
    getNextPageParam: lastPage => lastPage.nextPage ?? false
  });

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>An error occurred: {error.message}</div>;

  return (
    <div>
      {data.pages.map((page, index) => (
        <React.Fragment key={index}>
          {page.items.map(item => <div key={item.id}>{item.name}</div>)}
        </React.Fragment>
      ))}
      {hasNextPage && <button onClick={fetchNextPage}>Load More</button>}
      {isFetchingNextPage && <div>Loading more...</div>}
    </div>
  )
}
```

Requêtes simultanées

Utilisation du hook `useQueries` pour exécuter plusieurs requêtes simultanément. Cela est utile pour charger plusieurs données indépendantes en même temps.

```
import { useQueries } from '@tanstack/react-query';

function FetchMultipleData() {
  const results = useQueries({
    queries: [
      { queryKey: ['user', 1], queryFn: () => fetch('/api/user/1').then(res => res.json()) },
      { queryKey: ['post', 1], queryFn: () => fetch('/api/post/1').then(res => res.json()) }
    ]
  });

  return (
    <div>
      <h1>User: {results[0].data?.name}</h1>
      <h2>First Post: {results[1].data?.title}</h2>
    </div>
  );
}
```

Configurer useQueries pour des requêtes multiples

useQueries permet de lancer plusieurs requêtes simultanées. Chaque requête est indépendante mais configurée dans un seul appel de hook, ce qui simplifie la gestion de plusieurs sources de données.

```
const fetchResource = (resource) => fetch(`/api/${resource}`).then(res => res.json());

function Resources() {
  const resourceNames = ['users', 'posts', 'comments'];
  const queryResults = useQueries({
    queries: resourceNames.map(name => ({
      queryKey: [name],
      queryFn: () => fetchResource(name),
      staleTime: 5000
    }))
  });

  if (queryResults.some(query => query.isLoading)) return <div>Loading...</div>

  return (
    <div>
      {queryResults.map((result, idx) => (
        <div key={idx}>
          <h3>{resourceNames[idx]}</h3>
          <ul>
            {result.data.map(item => <li key={item.id}>{item.title || item.name}</li>)}
          </ul>
        </div>
      ))}
    </div>
  );
}
```

Exemple simple avec useQueries

Utilisation de useQueries pour charger des données de différents endpoints API simultanément.

```
const fetchById = (type, id) => fetch(`/api/${type}/${id}`).then(res => res.json());

function FetchMultipleIds() {
  const ids = [1, 2, 3];
  const result = useQueries({
    queries: ids.map(id => ({
      queryKey: ['data', id],
      queryFn: () => fetchById('data', id)
    }))
  });

  return (
    <div>
      {result.map((res, index) => (
        <div key={index}>
          <h3>Data {ids[index]}</h3>
          <p>{res.data?.detail}</p>
        </div>
      )));
    </div>
  );
}
```

Optimisation des requêtes simultanées

Stratégies pour optimiser les performances et l'efficacité des requêtes simultanées, comme le partage du cache et la réduction des appels réseau.

```
function OptimizedMultipleQueries() {
  const types = ['profile', 'settings', 'notifications'];
  const queries = useQueries({
    queries: types.map(type => ({
      queryKey: [type],
      queryFn: () => fetchData(type),
      staleTime: 10000, // Use a longer stale time to reduce refetching
      cacheTime: 300000 // Keep data in cache longer
    }))
  });

  return (
    <div>
      {queries.map((query, index) => (
        <div key={index}>
          <h3>{types[index]}</h3>
          {query.isLoading ? <p>Loading...</p> : <p>{query.data?.name}</p>}
        </div>
      )));
    </div>
  );
}
```

Gestion des dépendances entre requêtes

Gérer les dépendances entre les requêtes avec `useQuery` pour s'assurer que certaines données sont chargées avant d'autres.

```
import { useQuery } from '@tanstack/react-query';

const getUser = () => fetch('/api/user').then(res => res.json());
const getUserPosts = userId => fetch(`api/users/${userId}/posts`).then(res => res.json());

function DependentQueries() {
  const userQuery = useQuery(['user'], getUser);
  const postsQuery = useQuery(['user', 'posts'], () => getUserPosts(userQuery.data.id), {
    enabled: !!userQuery.data // Only run this query if the user data is available
  });

  return (
    <div>
      {userQuery.isLoading ? <p>Loading user...</p> : <h1>{userQuery.data.name}</h1>}
      {postsQuery.isLoading ? <p>Loading posts...</p> : postsQuery.data.map(post => <p key={post.id}>{post.title}</p>)}
    </div>
  );
}
```

Préchargement de données

Présentation de la fonctionnalité de préchargement dans Tanstack Query pour améliorer l'expérience utilisateur en chargeant les données avant qu'elles ne soient nécessaires.

```
import { useQuery, QueryClient } from '@tanstack/react-query';

const queryClient = new QueryClient();
const fetchProductDetails = id => fetch(`api/products/${id}`).then(res => res.json());

// Préchargement des données d'un produit
queryClient.prefetchQuery(['product', 1], () => fetchProductDetails(1));

function ProductDetails() {
  const { data: product } = useQuery(['product', 1], () => fetchProductDetails(1));

  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
    </div>
  );
}
```

Préchargement de données

Présentation de la fonctionnalité de préchargement dans Tanstack Query pour améliorer l'expérience utilisateur en chargeant les données avant qu'elles ne soient nécessaires.

Invalidation sélective du cache

L'invalidation sélective permet de rafraîchir certaines données dans le cache sans affecter les autres, ce qui est crucial pour maintenir les données synchronisées avec les serveurs.

```
import { useQueryClient, useQuery } from '@tanstack/react-query';

const fetchUser = id => fetch(`/api/users/${id}`).then(res => res.json());

function UserProfile({ userId }) {
  const queryClient = useQueryClient();
  useQuery(['user', userId], () => fetchUser(userId), {
    onSuccess: () => {
      // Invalider et rafraîchir les dépendances
      queryClient.invalidateQueries(['posts', userId]);
    }
  });

  return <div>User Profile</div>;
}
```

Invalidation sélective du cache

L'invalidation sélective permet de rafraîchir certaines données dans le cache sans affecter les autres, ce qui est crucial pour maintenir les données synchronisées avec les serveurs.

```
import { useMutation, useQueryClient } from '@tanstack/react-query';

const updateUser = userData => fetch('/api/user', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(userData)
});

function UpdateUser() {
  const queryClient = useQueryClient();
  const mutation = useMutation(updateUser, {
    onSuccess: () => {
      // Invalider uniquement les requêtes liées à l'utilisateur mis à jour
      queryClient.invalidateQueries(['userProfile']);
    }
  });

  return <button onClick={() => mutation.mutate({ id: 1, name: 'Jane Doe' })}>Update User</button>;
}
```

Introduction aux Optimistic Updates

Les Optimistic Updates sont une stratégie pour améliorer l'expérience utilisateur dans les applications web interactives, en appliquant immédiatement les modifications présumées sans attendre la confirmation du serveur.

```
function UpdateUserComponent({ user }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(updateUser, {
    onMutate: async newUser => {
      await queryClient.cancelQueries(['user', newUser.id]);
      const previousUser = queryClient.getQueryData(['user', newUser.id]);
      queryClient.setQueryData(['user', newUser.id], newUser);
      return { previousUser };
    },
    onError: (err, newUser, context) => {
      queryClient.setQueryData(['user', newUser.id], context.previousUser);
    },
    onSettled: () => {
      queryClient.invalidateQueries(['user', user.id]);
    }
  });

  return (
    <button onClick={() => mutate({ ...user, name: 'Updated Name' })}>
      Update Name
    </button>
  );
}
```

Exemple pratique d'Optimistic Update

Démonstration d'un Optimistic Update avec un exemple où un utilisateur peut activer ou désactiver une fonctionnalité de notification instantanément.

```
const toggleNotification = async (userId, enabled) => {
  return fetch('/api/users/${userId}/notification', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ enabled })
  }).then(res => res.json());
};

function NotificationToggle({ user }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(() => toggleNotification(user.id, !user.notificationEnabled), {
    onMutate: async newSetting => {
      await queryClient.cancelQueries(['user', user.id]);
      const previousUser = queryClient.getQueryData(['user', user.id]);
      queryClient.setQueryData(['user', user.id], {
        ...user,
        notificationEnabled: !user.notificationEnabled
      });
      return { previousUser };
    },
    onError: (err, newSetting, context) => {
      queryClient.setQueryData(['user', user.id], context.previousUser);
    }
  });

  return (
    <button onClick={() => mutate()}>
      {user.notificationEnabled ? 'Disable' : 'Enable'} Notifications
    </button>
  );
}
```

Principes de base du caching

Le caching avec Tanstack Query permet de stocker les résultats de requêtes et de les réutiliser pour améliorer la rapidité des chargements de données et réduire la charge sur les serveurs backend.

```
import { useQuery } from '@tanstack/react-query';

const fetchData = () => fetch('/api/data').then(res => res.json());

function DataComponent() {
  const { data } = useQuery(['data'], fetchData, {
    cacheTime: 5 * 60 * 1000, // Cache les données pendant 5 minutes
  });

  return <div>{data.title}</div>;
}
```

Configurer le caching pour des performances optimisées

Configuration du cache pour maximiser les performances, en définissant des politiques de durée de vie du cache et en personnalisant les stratégies de récupération des données.

```
import { useQuery, useQueryClient } from '@tanstack/react-query';

const fetchProduct = id => fetch(`/api/products/${id}`).then(res => res.json());

function Product({ productId }) {
  const queryClient = useQueryClient();
  const { data: product } = useQuery(['product', productId], () => fetchProduct(productId), {
    onSuccess: () => {
      // Précharger les données relatives dès que le produit est chargé
      queryClient.prefetchQuery(['reviews', productId], () => fetch(`/api/products/${productId}/reviews`));
    }
  });

  return <div>{product.name}</div>;
}
```

Utilisation de Suspense avec React Query

Explication de l'intégration de React Suspense avec React Query pour gérer les états de chargement de manière plus déclarative et homogène.

```
import { useQuery } from '@tanstack/react-query';
import { Suspense } from 'react';

const fetchData = () => fetch('/api/data').then(res => res.json());

function DataLoader() {
  const { data } = useQuery(['data'], fetchData, {
    suspense: true // Activer Suspense
  });

  return <div>{data.title}</div>;
}

function App() {
  return (
    <Suspense fallback=<div>Loading...</div>>
      <DataLoader />
    </Suspense>
  );
}
```

Intégration de Suspense avec les composants React

Techniques pour intégrer Suspense avec les composants React, facilitant la gestion des dépendances de données et la manipulation des états de chargement.

```
import { useQuery } from '@tanstack/react-query';
import { Suspense, useState } from 'react';

const fetchDetails = id => fetch(`/api/details/${id}`).then(res => res.json());

function DetailsComponent({ id }) {
  const [ data: details ] = useQuery(['details', id], () => fetchDetails(id), {
    suspense: true
  });

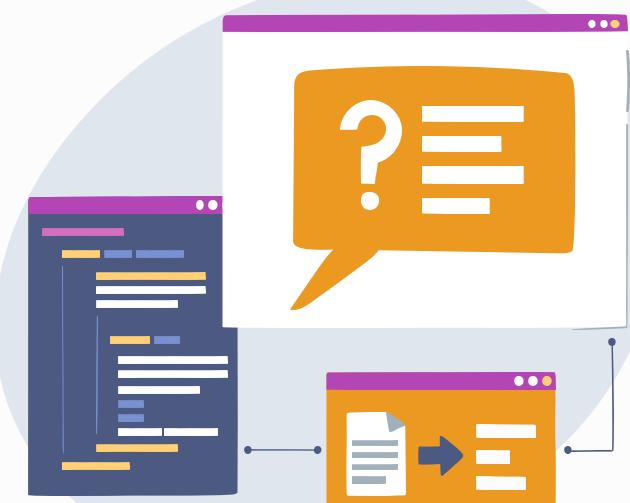
  return <div>{details.description}</div>;
}

function DetailsLoader() {
  const [id, setId] = useState(1);

  return (
    <div>
      <button onClick={() => setId(id + 1)}>Next</button>
      <Suspense fallback=<div>Loading details...</div>>
        <DetailsComponent id={id} />
      </Suspense>
    </div>
  );
}
```

17

Mise en place de tests dans React



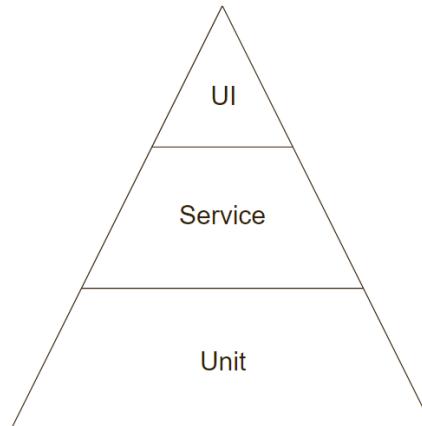
Introduction au test moderne de React

Les tests automatisés permettent de s'assurer que les fonctionnalités précédemment fonctionnelles continuent de l'être sans intervention manuelle, augmentant ainsi la confiance dans les modifications du code.

Les tests automatisés apportent confiance lors des refactorisations, servent de documentation toujours à jour et préviennent les bugs et régressions.

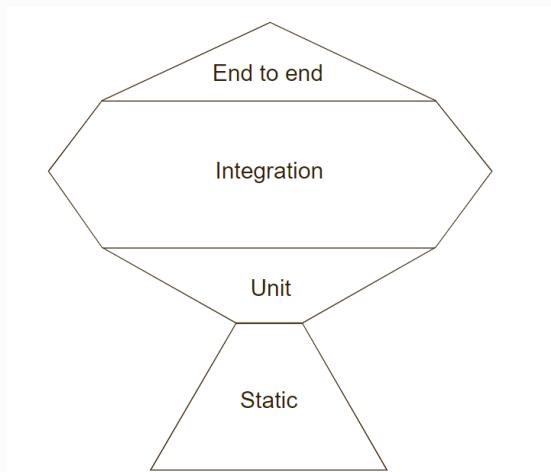
Présentation de la pyramide de tests

Explication de la pyramide des tests qui conseille une grande quantité de tests unitaires par rapport aux tests d'UI qui sont plus coûteux.



Approches alternatives de test : Le testing trophy de Kent C. Dodds

“It says that integration tests give you the biggest return on investment, so you should write more integration tests than any other kinds of tests.”



Quels outils ?

L'analyse statique permet de détecter les erreurs de syntaxe, les mauvaises pratiques et l'utilisation incorrecte des API : Formateurs de code, comme Prettier ; Les linters, comme ESLint ; Les vérificateurs de type, comme TypeScript et Flow.

Les tests unitaires vérifient que les algorithmes délicats fonctionnent correctement.
Outils : Jest.

Quels outils ?

Les tests d'intégration vous donnent l'assurance que toutes les fonctionnalités de votre application fonctionnent comme prévu. Outils : Jest et Enzyme : Jest et Enzyme ou react-testing-library.

Les tests de bout en bout permettent de s'assurer que votre application fonctionne comme un tout : le frontend, le backend, la base de données et tout le reste. Outils : Jest et Enzyme ou react-testing-librlibrary : Cypress.

Tests d'UI vs Tests unitaires : comparaison

Les tests d'UI sont coûteux et lents, tandis que les tests unitaires sont rapides et peu coûteux, idéaux pour tester des fonctions ou composants isolés.

```
// Exemple de test unitaire avec Jest
test('add function', () => {
  expect(add(1, 2)).toBe(3);
});
```

Introduction aux tests unitaires avec Jest

Jest est un outil permettant de réaliser des tests unitaires pour des algorithmes complexes ou des composants React.

```
// Test unitaire avec Jest
test('checks text content', () => {
  const { getByText } = render(<Button text="Click me!" />);
  expect(getByText(/click me/i)).toBeInTheDocument();
});
```

Tests d'intégration : maximiser le retour sur investissement

Les tests d'intégration couvrent des fonctionnalités entières ou des pages, offrant un meilleur retour sur investissement que les tests unitaires.

```
// Test avec React Testing Library
test('loads items eventually', async () => {
  const { getByText } = render(<ItemsList />);
  const item = await waitForElement(() => getByText('Item 1'));
  expect(item).toBeInTheDocument();
});
```

Comprendre les tests de bout en bout avec Cypress (E2E)

Cypress permet de réaliser des tests de bout en bout en simulant l'utilisation réelle de l'application dans un navigateur.

```
it('logs in successfully', () => {
  cy.visit('/login');
  cy.get('input[name=username]').type('user');
  cy.get('input[name=password]').type('password');
  cy.get('form').submit();
  cy.contains('Welcome, user!');
});
```

Écrire un test simple avec Jest

Structure d'un test simple avec Jest et comment exécuter un test.

```
function sum(a, b) {
  return a + b;
}

test('additionne 1 + 2 pour obtenir 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Matchers Jest

Introduction aux différents matchers disponibles dans Jest pour vérifier les résultats des tests.

```
test('deux plus deux fait quatre', () => {
  expect(2 + 2).toBe(4);
});
```

Tests d'assertion

Utilisation des assertions de base dans Jest comme toBe, toEqual, toBeNull, toBeUndefined, et toBeDefined.

```
test('null est nul', () => {
  expect(null).toBeNull();
  expect(undefined).toBeUndefined();
  expect(1).toBeDefined();
});
```

Tests de tableaux et objets

Vérification du contenu des tableaux et des objets avec les matchers `toContain` et `toHaveProperty`.

```
test('la liste contient le mot spécifique', () => {
  const shoppingList = ['couche', 'kleenex', 'savon'];
  expect(shoppingList).toContain('savon');
});
```

Tests d'exceptions

Tester les exceptions lancées par des fonctions avec le matcher `toThrow`.

```
function compilesAndroidCode() {
  throw new Error('Vous utilisez la mauvaise JDK');
}

test('compilation d\'Android génère une erreur', () => {
  expect(() => compilesAndroidCode()).toThrow('Vous utilisez la mauvaise JDK');
});
```

Tests de fonctions asynchrones

Tests de fonctions asynchrones avec des callbacks, des promesses et async/await.

```
test('les données de l\'API sont des utilisateurs', async () => {
  const data = await fetchData();
  expect(data).toEqual({ users: [] });
});
```

Jest Mock Functions

Création et utilisation de fonctions mock dans Jest avec jest.fn().

```
const myMock = jest.fn();
myMock();
expect(myMock).toHaveBeenCalled();

jest.mock('axios');
const axios = require('axios');

test('mocking axios', () => {
  axios.get.mockResolvedValue({ data: 'some data' });
  // test axios.get()
});
```

Exécuter des tests conditionnels

Exécution de tests conditionnels avec `test.only` et `test.skip`, et grouper les tests avec `describe`.

```
test.only('ce test est le seul à être exécuté', () => {
  expect(true).toBe(true);
});

test.skip('ce test est ignoré', () => {
  expect(true).toBe(false);
};

describe('groupe de tests', () => {
  test('test A', () => {
    expect(true).toBe(true);
  });

  test('test B', () => {
    expect(true).toBe(true);
  });
});
```

18

Ejecter d'expo



Introduction à Expo Prebuild

Expo prebuild est une commande CLI puissante permettant de générer les fichiers natifs pour iOS et Android à partir d'un projet Expo.

Expo prebuild automatise la création des fichiers nécessaires pour construire et exécuter des applications Expo sur des environnements natifs.

Pour utiliser Expo prebuild, il faut installer Expo CLI.

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

Commande Prebuild

La commande principale pour générer les fichiers natifs.

```
npx expo prebuild
```

Fichiers iOS Générés

Après la génération, plusieurs fichiers iOS sont créés, incluant le dossier ios.

- **Exemple de fichiers:**
- ios/Podfile
- ios/MyApp.xcodeproj

Podfile pour iOS

Le fichier Podfile gère les dépendances CocoaPods pour iOS.

```
platform :ios, '10.0'
target 'MyApp' do
  use_expo_modules!
  pod 'EXGL-CPP', :path => '../node_modules/expo-gl-cpp/cpp'
end
```

Fichiers Android Générés

De même, plusieurs fichiers Android sont créés, incluant le dossier android.

Exemple de fichiers:

android/build.gradle

android/app/src/main/AndroidManifest.xml

build.gradle pour Android

Le fichier build.gradle gère les configurations de construction pour Android.

```
buildscript {  
    ext {  
        buildToolsVersion = "30.0.2"  
        minSdkVersion = 21  
        compileSdkVersion = 30  
    }  
    repositories {  
        google()  
        mavenCentral()  
    }  
    dependencies {  
        classpath("com.android.tools.build:gradle:4.1.0")  
    }  
}
```

Configuration de AndroidManifest.xml

Le fichier AndroidManifest.xml configure les composants de l'application Android.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <application
        android:name=".MainApplication"
        android:label="@string/app_name"
        android:icon="@mipmap/ic_launcher">
        <activity
            android:name=".MainActivity"
            android:configChanges="keyboard|keyboardHidden|orientation|screenSize"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

19

Introduction à OneSignal



Introduction à OneSignal

Présentation de OneSignal, une plateforme de notifications push pour applications mobiles et web.

```
// Exemple d'importation de OneSignal dans une application React Native  
import OneSignal from 'react-native-onesignal';
```

Configuration Initiale

Étapes pour installer le SDK OneSignal dans une application React Native.

Installation **du** SDK OneSignal

```
npm install react-native-onesignal
```

Configuration Android

Configuration spécifique à Android pour utiliser OneSignal.

```
<!-- AndroidManifest.xml -->
<service android:name="com.onesignal.GcmIntentService" android:permission="com.google.android.c2dm.permission.SEND" />
<receiver android:name="com.onesignal.GcmBroadcastReceiver" android:permission="com.google.android.c2dm.permission.RECEIVE" >
    <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />
        <category android:name="YOUR_PACKAGE_NAME" />
    </intent-filter>
</receiver>
```

Configuration iOS

Configuration spécifique à iOS pour utiliser OneSignal.

```
// AppDelegate.m
#import <OneSignal/OneSignal.h>

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    [OneSignal initWithLaunchOptions:launchOptions appId:@"YOUR_ONESIGNAL_APP_ID"];
    return YES;
}
```

Gestion des Permissions

Demander les permissions nécessaires pour envoyer des notifications push.

```
// Demander la permission de l'utilisateur pour les notifications
OneSignal.promptForPushNotificationsWithUserResponse(response => {
    console.log("Permission accordée: ", response);
});
```

Envoi d'une Notification Push

Comment envoyer une notification push à partir du tableau de bord OneSignal.

```
// Exemple de payload JSON pour envoyer une notification via l'API OneSignal
{
    "app_id": "YOUR_ONESIGNAL_APP_ID",
    "include_player_ids": ["PLAYER_ID"],
    "headings": {"en": "Titre de la notification"},
    "contents": {"en": "Contenu de la notification"}
}
```

Réception de Notifications

Gérer la réception de notifications dans l'application React Native.

```
OneSignal.setNotificationOpenedHandler(notification => {
  console.log("Notification ouverte: ", notification);
});
```

Personnalisation des Notifications

Personnaliser les notifications avec des images, des sons, et des actions.

```
// Payload JSON pour une notification personnalisée
{
  "app_id": "YOUR_ONESIGNAL_APP_ID",
  "include_player_ids": ["PLAYER_ID"],
  "headings": {"en": "Titre de la notification"},
  "contents": {"en": "Contenu de la notification"},
  "big_picture": "URL_DE_L_IMAGE",
  "buttons": [{"id": "id1", "text": "Button 1"}, {"id": "id2", "text": "Button 2"}]
}
```

Suivi des Clics sur les Notifications

Comment suivre les clics sur les notifications push pour analyser l'engagement des utilisateurs.

```
OneSignal.setNotificationOpenedHandler(notification => {
    console.log("Notification ouverte: ", notification);
    // Logique supplémentaire pour suivre les clics
});
```

Segmentation des Utilisateurs

Segmenter les utilisateurs pour envoyer des notifications ciblées.

```
// Tagger un utilisateur pour la segmentation  
OneSignal.sendTag("user_type", "premium");
```

Gestion des Utilisateurs

Gérer les abonnements des utilisateurs aux notifications push.

```
// Désabonner un utilisateur des notifications  
OneSignal.disablePush(true);  
  
// Réabonner un utilisateur aux notifications  
OneSignal.disablePush(false);
```

BONUS

Type**Script**



TypeScript, c'est quoi ?

TypeScript est un sur-ensemble de JavaScript développé par Microsoft, ayant pour but d'ajouter un système de typage statique. Cela nous permettra d'écrire un code plus sécurisé, plus facilement maintenable, et de détecter des erreurs avant même l'exécution du code.

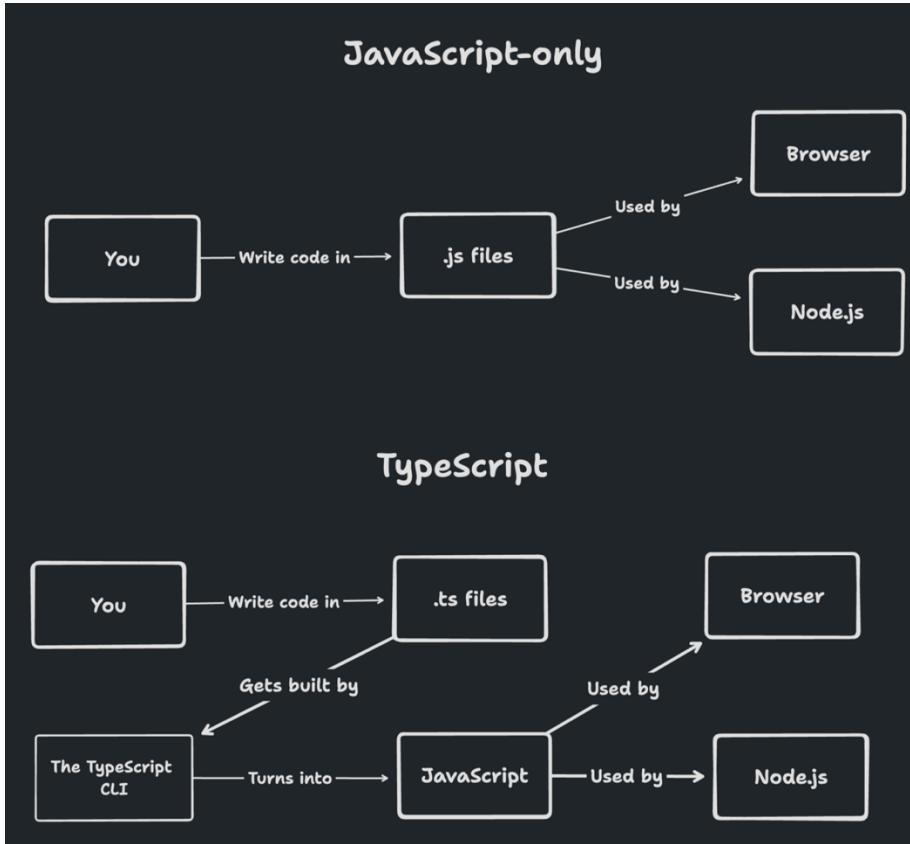
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

Les fichiers TypeScript auront une extension en .ts ou .tsx

Les avantages de **TypeScript**

1. Syntaxe proche du JavaScript
2. Permet de détecter des erreurs potentielles, même sans déclarer de type
3. Facilite la documentation du code
4. Permet à l'IDE de gérer les suggestion de code plus facilement
5. Meilleure maintenabilité
6. L'inférence de type

Le fonctionnement de **TypeScript**



La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.

Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;

person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string
```

Les types primitifs

TypeScript nous fournit 3 types primitifs, et 2 types utilitaires:

1. number
2. string
3. boolean
4. unknown (quand on ne sait pas quel type nous allons recevoir, on y reviendra après)
5. any (à éviter)

La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.

Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;
function sayHello(name : string) : string {
    return "Hello " + name;
}
person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string

sayHello(isMajor) // string attendu en paramètre, mais on passe un boolean
```

L'inférence de type

C'est sympa de pouvoir déclarer des types primitifs sur les variables lorsqu'on les déclare, mais ça va vite devenir très répétitif non ?

C'est là que l'inférence de type entre en scène: TS va être capable de « déduire » le type de certaines variables et signatures de fonction via leur valeurs d'origine. Pas besoin de tout typer explicitement !

```
let person = "Jean";
let age = 30;
let isMajor = true;
function sayHello(name : string)  {
    return "Hello " + name;
}
person = age; //Même erreur d'auparavant
isMajor = isMajor + person; //Idem

sayHello(isMajor) // Toujours pareil
```

L'inférence de type

Attention, TS n'est pas omniscient, et ne sera pas toujours capable d'inférer les types correctement.

Si il ne sait pas, il utilisera le type any, qui ne procède à aucune vérification !

Par exemple :

```
const myArray = []; //Type implicite: any[]
myArray.push(1) //Pas d'erreur car non strict
myArray.push('Hello') //Pas d'erreur car non strict, mais dangereux
```

Types non conformes à la réalité

TypeScript nous permet de détecter des erreurs dans le code, mais peut parfois autoriser des opérations qui généreraient des erreurs à l'exécution.

```
const names = ["Charles", "Pierrick"]
console.log(names[2].toLowerCase())
//TS ne signale pas d'erreur, mais la dernière ligne va générer une erreur à l'exécution
```

Type et Interface

TypeScript nous permet de créer nos propres types, grâce aux mots clés « type » et « interface ».

```
type Age = number;
interface Person {
    name: string;
    age: Age;
}
let driver: Person = {
    name: 'James May',
    age: 56
};
driver.age = 57; // OK
driver.age = '57'; // Error
```

Type ou Interface ?

Type et interface peuvent sembler interchangeables, car ils le sont majoritairement.

Normalement, type est plutôt utilisé pour définir les alias, et interface pour typer des objets complets. À notre niveau, nous pouvons nous contenter d'utiliser type au général, car ils nous permettent de faire des « union types » qui nous seront utiles pour la suite.

```
type item = {  
    name: string,  
    price: number,  
}
```

```
type weapon = item & {  
    damage: number,  
}
```

```
interface item {  
    name: string;  
    price: number;  
}
```

```
interface weapon extends item {  
    damage: number;  
}
```

Union types et Narrowing

Avec le mot clé `type`, nous pouvons créer des « union types », c'est-à-dire des variables qui peuvent accepter plusieurs types, par exemple:

```
type StringOrNumber = string | number;  
  
let myAge: StringOrNumber = 25;  
myAge = '25'; // OK
```

Pour savoir quel est le type réel d'une variable avec une union, on peut utiliser l'inférence de TS :

```
const doSomething = (value: StringOrNumber) => {  
    if (typeof value === 'string') {  
        //TypeScript infère que c'est une string  
        return value.toUpperCase();  
    }  
    //TypeScript infère que c'est un number  
    return value;  
}
```

Narrowing sur des objets ?

Pour faire du narrowing sur des types complexes, nous pouvons utiliser le mot clé « in » pour déterminer ou non l'existence d'une propriété propre à un des types.

```
type Fish = {
    swim : () => void;
}

type Bird = {
    fly : () => void;
}

function move(animal: Fish | Bird){
    if("swim" in animal){
        return animal.swim();
    }
    return animal.fly();
}
```

Les génériques

Les génériques sont une fonctionnalité de TS qui nous permettent de créer des types génériques réutilisables et de réduire le code à écrire. Prenons l'exemple ci-dessous:

```
type StringCollection = {
    name: string;
    items : string[];
}

type NumberCollection = {
    name: string;
    items : number[];
}

type BooleanCollection = {
    name: string;
    items : boolean[];
}
```

Les génériques

On peut créer un type générique Collection, qui prendra en « paramètre » un type T qui sera assigné à la propriété items. Nos types précédents seront donc déclarées par le générique.

```
type Collection<T> = {  
    name: string;  
    items : T[];  
}  
type StringCollection = Collection<string>;  
type NumberCollection = Collection<number>;  
type BooleanCollection = Collection<boolean>;
```

Les génériques

Il existe des génériques utilitaires, comme `<Partial>`, `<Omit>`, `<Record>`... que nous utiliserons peut être lors de la formation.

Vous pouvez également les retrouver sur la documentation TypeScript, juste ici:
<https://www.typescriptlang.org/docs/handbook/utility-types.html>

Pour aller plus loin

Compte Twitter et Youtube et Matt Pocock

<https://twitter.com/mattpcockuk>

<https://www.youtube.com/@mattpcockuk>