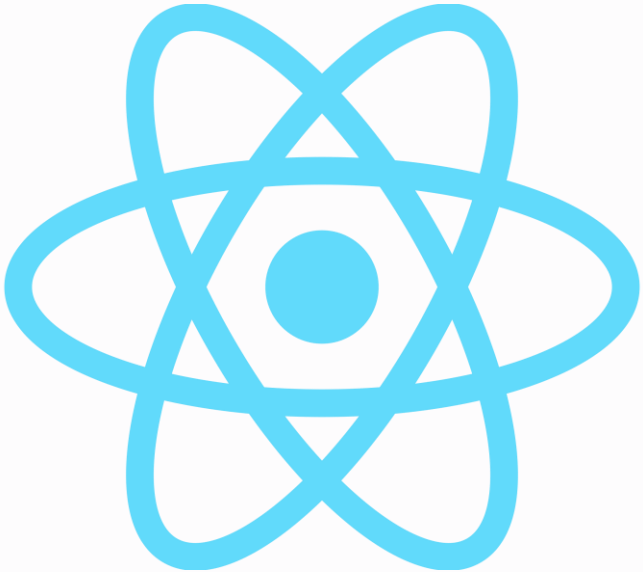

OXiane
INSTITUT
make it clever

React JS



Support réalisé par Pierrick Hauguel

TABLE DES MATIÈRES

TABLE DES MATIÈRES

TABLE DES MATIÈRES

TABLE DES MATIÈRES

01

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/expanser 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 à Vite

Vite est un outil de build moderne conçu pour les projets JavaScript, TypeScript, et les frameworks tels que Vue, React, et Svelte. Il offre un démarrage rapide de serveur de développement et des rechargements à chaud ultra-rapides en tirant parti de l'importation de modules ES natifs dans les navigateurs et en pré-bundling des dépendances avec esbuild.

```
# Installation de Vite pour un nouveau projet React
npm create vite@latest mon-projet-react -- --template react

# Démarrage du projet
cd mon-projet-react
npm install
npm run dev
```

Configuration de Vite

```
// vite.config.js
import reactRefresh from '@vitejs/plugin-react-refresh';

export default {
  plugins: [reactRefresh()],
  resolve: {
    alias: {
      '@': '/src',
    },
  },
  build: {
    outDir: 'build',
  },
};
```

Vite est hautement configurable via son fichier vite.config.js. Vous pouvez ajuster la résolution des modules, les plugins, les options CSS, et bien plus, permettant une personnalisation profonde pour répondre aux besoins spécifiques de votre projet.

Optimisations et Déploiement avec Vite

Vite offre des fonctionnalités d'optimisation prêtes à l'emploi pour le déploiement de production, telles que la minification, le découpage de code (code splitting), et le préchargement des modules. Ces optimisations améliorent la performance de chargement de l'application en production.

```
# Commande pour construire un projet pour la production
npm run build

# Analyse du bundle pour optimisation
npm run preview
```

02

Introduction à ReactJS

Découvrir ReactJS

React est une bibliothèque JavaScript puissante et flexible conçue pour la construction d'interfaces utilisateur. Développée par Facebook et soutenue par une large communauté, elle favorise l'approche orientée composant, permettant ainsi de construire des UI complexes et dynamiques à partir de petits, isolés et réutilisables morceaux de code.

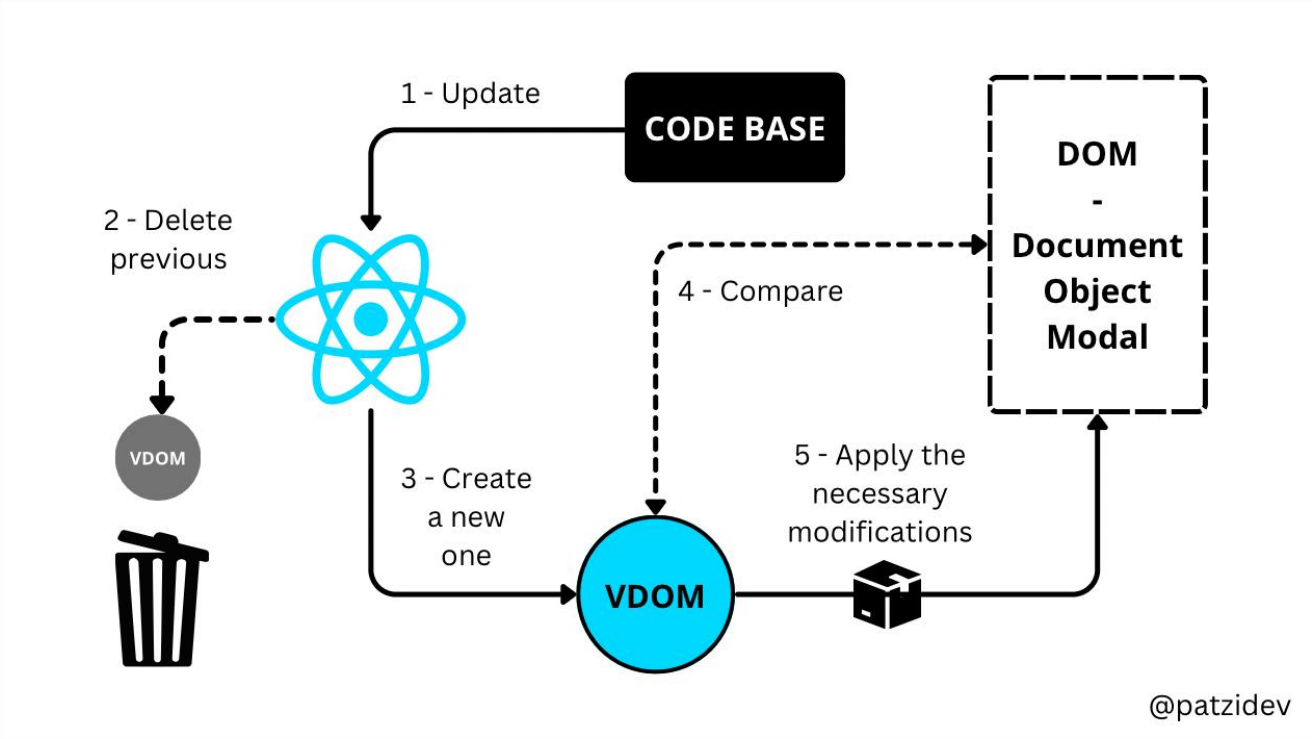
Comparaison entre React, Angular, et Vue

- **Angular** est un framework tout-en-un offrant une boîte à outils complète pour le développement front-end, incluant des solutions pour le routing, la gestion d'état, et plus encore.
- **React** se concentre sur la couche de vue, offrant une grande flexibilité et une intégration facile avec d'autres bibliothèques pour le routing et la gestion d'état.
- **Vue** combine la simplicité de React concernant les composants avec des fonctionnalités out-of-the-box pour une expérience de développement intégrée.
- **Points forts de React:**
- Utilisation efficace du DOM virtuel pour optimiser les performances.
- Une communauté vaste et active, avec un écosystème riche en outils et bibliothèques.
- Flexibilité et modularité exceptionnelles.

Concepts Clés de ReactJS

- **Composants:** Blocs de construction de toute application React, pouvant être fonctionnels ou de classe.
- **Props et State:** Mécanismes pour gérer les données et l'état interne des composants.
- **DOM Virtuel:** Une abstraction légère du DOM permettant des mises à jour efficaces et performantes.
- **Les Hooks:** Introduction récente permettant l'utilisation de l'état et d'autres fonctionnalités React dans les composants fonctionnels.

Dom Virtuel


1 - Update
CODE BASE
DOM
-
Document
Object
Modal
2 - Delete
previous
4 - Compare
VDOM
3 - Create
a new
one
VDOM
5 - Apply the
necessary
modifications
@patzidev

Écosystème et Outils de React

React est accompagné d'un riche écosystème comprenant des solutions de gestion d'état comme Redux et Context API, des outils de routing comme React Router, et des bibliothèques pour les appels API et les tests unitaires, offrant aux développeurs une boîte à outils complète pour créer des applications robustes et maintenables.

03

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

```jsx
// 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.

Bonnes Pratiques avec JSX et la Composition

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

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

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

L'Importance du JSX et de la Composition

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

Fragments dans les composants fonctionnels React

Un fragment est un type spécial de composant React qui permet de grouper plusieurs éléments sans ajouter de nœud supplémentaire au DOM.

```
function MyComponent() {
  return (
    <>
      <h1>Title</h1>
      <p>This is a paragraph.</p>
      <button>Button</button>
    </>
  );
}
```

04

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
};
```

Exercice !

Grâce aux notions que l'on vient de voir, créez un composant Card qui est composée de 3 éléments :

- 1 Header, avec un titre, modifiable grâce à une prop
- 1 Body, dans lequel on peut passer n'importe quel contenu
- 1 Footer, qui est un texte simple, modifiable grâce à une prop.

05

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);
```

Exercice !

-> Créez un composant fonctionnel nommé MouseTracker.
-> Utilisez useState pour stocker les coordonnées X et Y de la souris.
-> Utilisez useEffect pour écouter les événements de déplacement de la souris (mousemove). À chaque événement, mettez à jour les coordonnées de la souris.
-> Affichez les coordonnées X et Y de la souris dans votre composant.
➔ (listener : mousemove)

06
Manipulation de Listes

Fondamentaux des Listes en JavaScript

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

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

La Méthode map pour le Rendu des Listes

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

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

Rôle des Key dans les Listes React

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

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

Filtrage de Listes avec filter

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

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

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

Autres Méthodes de Raccourcis

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

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

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

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

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

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

Exercice !

Avec les listes, useState, fetch et useEffect, récupérez et affichez la liste des planètes de Star Wars en utilisant cette API :

https://swapi.dev/

07
Les Événements
en JSX

Comprendre les Événements en JSX

React simplifie la gestion des événements en encapsulant les événements natifs du DOM dans des événements synthétiques, assurant ainsi une cohérence entre les navigateurs. Les noms des événements en JSX utilisent la notation camelCase, par opposition à la notation minuscule en HTML traditionnel.

```
// Exemple d'utilisation d'un événement onClick en JSX
function handleClick() {
  console.log('Le bouton a été cliqué');
}

const buttonElement = <button onClick={handleClick}>Cliquez-moi</button>;
```

Syntaxe Standard pour les Événements

Les événements dans JSX sont assignés à l'aide d'attributs spécifiques, similaires à ceux du HTML, mais avec une syntaxe camelCase. Les gestionnaires d'événements sont des fonctions JavaScript passées comme attributs.

```jsx
// Exemple de syntaxe standard pour les événements en JSX
function handleSubmit(event) {
  event.preventDefault();
  console.log('Formulaire soumis');
}

const formElement = <form onSubmit={handleSubmit}>...</form>;
```

Création de Gestionnaires d'Événements

```
// Exemple de déclaration d'un gestionnaire d'événement comme fonction fléchée
const handleClick = (e) => console.log(e);

const buttonElement = <button onClick={handleClick}>Cliquez-moi</button>;
```

Les gestionnaires d'événements peuvent être définis comme des fonctions fléchées directement dans JSX pour de petites actions ou des démonstrations

Types d'Événements en React

React gère une variété d'événements, permettant aux développeurs de réagir à presque toutes les interactions des utilisateurs, incluant les événements de souris, de clavier, de formulaire et de contrôle.

Exemples:

Événements de Souris: onClick, onMouseOver
Événements de Clavier: onKeyDown
Événements de Formulaire: onChange

```
// Exemple d'événements communs en React
<input type="text" onChange={handleChange} />
<button onClick={handleClick}>Cliquez-moi</button>
```

Passer des Arguments dans les Gestionnaires

Il est possible de transmettre des arguments supplémentaires aux gestionnaires d'événements, en utilisant soit des fonctions fléchées directement dans JSX

```
// Exemple de passage d'arguments aux gestionnaires d'événements avec une fonction f
const handleClick = (id) => console.log('ID:', id);

const buttonElement = <button onClick={() => handleClick(id)}>Cliquez-moi</button>;
```

Exercice !

En reprenant l’exercice précédent, ajoutez la gestion de la pagination sur la page.

08

React Router

Introduction au Routing Web

Le routing permet de naviguer entre différents composants dans une application, transformant les Single Page Applications (SPA) en expériences riches et dynamiques, où chaque composant représente une vue ou une "page" différente sans nécessiter de rechargement complet.

Pourquoi React Router ?

React Router est une bibliothèque de routage déclarative conçue spécifiquement pour React. Elle facilite la gestion de la navigation et le partage d'URL dans les applications React, rendant le routing côté client simple et efficace.

Configurer les Routes avec React Router

Avec React Router, les routes sont déclarées en utilisant createBrowserRouter, où chaque route est un objet dans un tableau, spécifiant le chemin, le composant à rendre, et éventuellement des routes enfants

```
const router = createBrowserRouter([
  { path: "/", element: <Root />, children: [
      { path: "team", element: <Team /> },
    ]
  },
]);
```

Concepts Clés dans React Router

```
const router = createBrowserRouter([
  { path: "/", element: <Root />, children: [
      { path: "team", element: <Team /> },
    ]
  },
]);
```

```
function Dashboard() {
  return (
    <div>
      <h2>Tableau de Bord</h2>
      <Outlet />
    </div>
  );
}
```

- **element :** Spécifie le composant rendu pour la route.
- **loader :** Charge des données avant le rendu de la route, idéal pour le pré-chargement d'état.
- **children :** Routes imbriquées offrant une structure hiérarchique pour la navigation complexe.

Amélioration de la Gestion des Données

Les data APIs de React Router centralisent la logique de chargement des données, permettant une gestion d'état plus claire et une expérience utilisateur améliorée grâce au préchargement des données nécessaires.

```
const router = createBrowserRouter([
  {
    element: <Teams />,
    path: "teams",
    loader: async () => {
      return fakeDb.from("teams").select("*");
    },
    children: [
      {
        element: <Team />,
        path: ":teamId",
        loader: async ({ params }) => {
          return fetch(`/api/teams/${params.teamId}.json`);
        },
      },
    ],
  },
]);
```

Naviguer avec React Router

```
// Exemple d'utilisation des data APIs de React Router
import { useRoutes, useNavigate } from 'react-router-dom';

function App() {
  const navigate = useNavigate();

  const handleClick = () => {
    // Navigation programmée après la soumission d'un formulaire
    navigate('/about');
  };

  return (
    <div>
      <button onClick={handleClick}>Go to About</button>
    </div>
  );
}
```

- **<Link> :** Permet de créer des liens navigables sans recharger la page.
- **useNavigate :** Pour la navigation programmée, par exemple après la soumission d'un formulaire.

Routes Dynamiques avec React Router

React Router permet de définir des routes dynamiques, s'adaptant à des chemins variables pour une flexibilité maximale dans la gestion des paramètres d'URL.

```
// Exemple de définition d'une route dynamique avec des paramètres d'URL
{ path: "users/:userId" }
```

```
// Exemple de création de liens navigables avec <Link>
import { Link } from 'react-router-dom';

<Link to="/about">About</Link>
```

Exercice !

En utilisant le router et react-hooks-form, créer un CRUD relié à la plateforme crudcrud.com.

Attention ! Vous avez seulement 100 tokens. Si vous avez dépassé ce montant, vous pouvez avec une nouvelle API Key en vous mettant en navigation privée.

Pour aller plus loin

Compréhension du router SSR (server side rendering) :
https://github.com/remix-run/react-router/tree/main/examples/ssr-data-router

Remix :
https://remix.run/

08.1

Tanstack Router

Configuration du projet

- Créez un nouveau projet React avec Vite et le modèle TypeScript :

```
npm create vite@latest tanstack-router-demo -- --template react-ts
```

- Ensuite, installez TanStack Router :

```
npm install @tanstack/router
npm install --save-dev @tanstack/router-vite-plugin
```

Routes

Définissez les routes de votre application en créant des fichiers dans le dossier src/routes. Les routes doivent être spécifiées dans ces fichiers, et la structure des fichiers sera automatiquement générée par le plugin.

```
import { createFileRoute } from '@tanstack/react-router';

export const Route = createFileRoute('/profile')({
  component: () => <div>Hello /profile!</div>,
});
```

Fournisseur de routage

```
import './App.css';
import { RouterProvider, createRouter } from '@tanstack/react-router';
import { routeTree } from './routeTree.gen';

const router = createRouter({ routeTree });

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

Configurez le fournisseur de routage dans votre application en remplaçant le fichier App.tsx :

Navigation

```
import { Link, Outlet, createRootRoute } from '@tanstack/react-router';

export const Route = createRootRoute({
  component: () => (
    <>
      <h1>My App</h1>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/profile">Profile</Link>
        </li>
      </ul>
      <Outlet />
    </>
  ),
});
```

- Permettez la navigation entre les pages en ajoutant des liens dans le composant racine de vos routes :

Pour aller plus loin...

Vidéo de Jack Herrington :
https://www.youtube.com/watch?v=qOwnQJOClrw
Compte de Tanner Linsley :
https://twitter.com/tannerlinsley

09

Redux et Zustand

Introduction à Redux

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

Principes fondamentaux de Redux

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

Configuration initiale de Redux

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

const store = createStore(rootReducer);
```

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

Actions

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

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

Reducers

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

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

useSelector et useDispatch

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

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

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

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

Store et gestion de l'état

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

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

const store = createStore(rootReducer);
```

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

Middleware Redux

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

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

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

Introduction à Redux Toolkit

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

Avantages par rapport à Redux standard

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

Installation et configuration

- 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 { configureStore } from '@reduxjs/toolkit';

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

Création de Slice avec createSlice

```
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;
```

- 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 { createAsyncThunk } from '@reduxjs/toolkit';

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

Utilisation de useDispatch et useSelector avec Redux Toolkit et React

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

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

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

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

Introduction à Zustand

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

Philosophie et avantages de Zustand

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

Installation de Zustand

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

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

Création d'un store avec Zustand

```
import create from 'zustand';

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

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

Gestion de l'état avec des hooks

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

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

Gestion des effets secondaires

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

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

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

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

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

Sélecteurs et abonnements

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

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

Exemples d'utilisation avancée de Zustand

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

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

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

Exercice !

Remplacez la centralisation d’état par zustand !

Pour aller plus loin…

Daishi Kato : https://twitter.com/dai_shi

Vidéo de Jack Herrington : https://www.youtube.com/watch?v=sqTPGMipjHk

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".

Exemple

```
const ThemeContext = createContext({
  theme: "light",
  toggleTheme: () => {},
});

const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

const MyComponent = () => {
  const { theme } = useContext(ThemeContext);

  return (
    <div>
      The current theme is: {theme}
    </div>
  );
};
```

Exercice !

Centralisez vos appels grâce à un context.

11

Stylisation et CSS en React

Approches de la Stylisation dans React

React offre plusieurs méthodes pour styliser les composants, du CSS traditionnel à des approches plus modernes et dynamiques, jouant un rôle crucial dans la création d'interfaces utilisateur attrayantes et cohérentes.

Utiliser le CSS Traditionnel dans React

L'importation de feuilles de style CSS est la méthode la plus directe et familière pour appliquer des styles, permettant une intégration facile des styles globaux ou spécifiques à un composant.

```
import './App.css';
```

Styled-components

```
const Button = styled.button`
  background-color: blue;
  color: white;
`;
```

Styled-components permet de créer des composants stylisés en utilisant des littéraux de gabarit tagués, intégrant étroitement styles et logique de composants pour un theming dynamique et une réutilisation facilitée.

Appliquer des Styles Inline

```jsx
// Exemple d'application de styles inline dans un composant React
const dynamicStyles = {
  backgroundColor: 'green',
  color: 'white',
};

function MyComponent() {
  return <div style={dynamicStyles}>Hello World</div>;
}
```

L'utilisation de styles inline via l'attribut style permet d'appliquer des styles dynamiques directement sur des éléments, utile pour des ajustements rapides ou des styles conditionnels.

Explorer Tailwind CSS et Emotion

Des bibliothèques comme Tailwind CSS et Emotion offrent des approches alternatives pour la stylisation, combinant les avantages de la modularité, de la réutilisabilité et de l'encapsulation des styles.

11.1

Les Styled-components

Introduction aux composants stylisés

- Les composants stylisés permettent de créer des composants réutilisables avec des styles encapsulés.
- Ils offrent une alternative aux approches CSS traditionnelles, avec une syntaxe plus concise et une meilleure séparation des préoccupations

```
import styled from 'styled-components';

const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Cliquez-moi</Button>
    </div>
  );
};
```

Définition et avantages des composants stylisés

```
const Button = styled.button`
  background-color: ${props => props.color};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button color="red">Bouton rouge</Button>
      <Button color="blue">Bouton bleu</Button>
    </div>
  );
};
```

- **Avantages:**
  - Amélioration de la lisbilité du code.
  - Réutilisation des styles.
  - Meilleure séparation des préoccupations.
  - Facilité de maintenance.

Créer et utiliser des composants stylisés

Créer un composant stylisé avec styled. Utiliser le composant comme n'importe quel autre composant React.

```
const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Cliquez-moi</Button>
    </div>
  );
};
```

Utilisation de props et de variantes

```
const Button = styled.button`
  background-color: ${props => props.color || 'blue'};
  color: white;
  font-size: 16px;
  padding: 10px;

  ${props => props.primary && `
    background-color: green;
  `}
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Bouton bleu</Button>
      <Button color="red">Bouton rouge</Button>
      <Button primary>Bouton principal</Button>
    </div>
  );
};
```

- Les props permettent de personnaliser les styles des composants stylisés.
- Les variantes facilitent la création de variations de style pour un composant.

Héritage de styles et composition de composants

- L'héritage de styles permet de réutiliser les styles d'un composant parent dans un composant enfant.
- La composition de composants permet de créer des composants plus complexes en combinant des composants plus simples.

```
const Container = styled.div`
  padding: 20px;
  border: 1px solid #ddd;
`;

const Title = styled.h2`
  margin-bottom: 10px;
`;

const Content = styled.p`
`;

const MyComponent = () => {
  return (
    <Container>
      <Title>Titre</Title>
      <Content>Contenu du composant</Content>
    </Container>
  );
};
```

Gestion des styles dynamiques

```
const Button = styled.button`
  background-color: ${props => props.isHovered ? 'red' : 'blue'};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  const [isHovered, setIsHovered] = useState(false);

  return (
    <div>
      <Button isHovered={isHovered} onMouseEnter={() => setIsHovered(true)}>
        Bouton avec style dynamique
      </Button>
    </div>
  );
};
```

- Les styles dynamiques permettent de générer des styles en fonction de l'état du composant ou des props.
- Ils peuvent être utilisés pour créer des interfaces utilisateur plus réactives et interactives.

Animations et transitions

- Les animations et les transitions permettent de rendre les interfaces utilisateur plus fluides et interactives.
- Elles peuvent être utilisées pour attirer l'attention de l'utilisateur ou pour améliorer l'expérience utilisateur.

```
const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;

  transition: all 0.2s ease-in-out;

  &:hover {
    background-color: red;
  }
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Hover Me</Button>
    </div>
  );
};
```

11.2

Le Package Emotion

Introduction à Emotion

```
import styled from '@emotion/styled';

const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Click me</Button>
    </div>
  );
};
```

- Emotion est une bibliothèque JavaScript pour créer des styles CSS dynamiques et réutilisables.
- Elle offre une alternative aux solutions CSS traditionnelles, avec une syntaxe plus concise et une meilleure séparation des préoccupations.

Définition et avantages d'Emotion

```
const Button = styled.button`
  background-color: ${props => props.color};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button color="red">Red Button</Button>
      <Button color="blue">Blue Button</Button>
    </div>
  );
};
```

- Amélioration de la lisbilité du code.
- Réutilisation des styles.
- Meilleure séparation des préoccupations.
- Facilité de maintenance.

Fonctionnement et concepts clés d'Emotion

```
const Button = styled.button`
  background-color: ${props => props.theme.colors.primary};
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Button with theme</Button>
    </div>
  );
};
```

Emotion injecte des styles CSS dans le DOM via des attributs data-emotion.
Les styles sont générés de manière unique pour chaque composant.

Créer et utiliser des styles avec Emotion

Création de composants stylisés avec styled d'Emotion.
Utilisation des composants stylisés comme des composants React classiques.

```
import styled from '@emotion/styled';

const Button = styled.button`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Click me</Button>
    </div>
  );
};
```

Syntaxe de base avec @emotion/core

```
import { css } from '@emotion/core';

const buttonStyles = css`
  background-color: blue;
  color: white;
  font-size: 16px;
  padding: 10px;
`;

<button className={buttonStyles}>Click me</button>
```

@emotion/core est le package principal d'Emotion.
Il fournit les fonctionnalités de base pour la création de styles.
Syntaxe de base pour définir des styles avec css:

Utilisation de props et de variantes

```
const Button = styled.button`
  background-color: ${props => props.color || 'blue'};
  color: white;
  font-size: 16px;
  padding: 10px;

  ${props => props.primary && `
    background-color: green;
  `}
`;

const MyComponent = () => {
  return (
    <div>
      <Button>Blue Button</Button>
      <Button color="red">Red Button</Button>
      <Button primary>Main Button</Button>
    </div>
  );
};
```

- **Props:**
  - Permettent de personnaliser les styles des composants stylisés.
- **Variantes:**
  - Facilité la création de variations de style pour un composant.

12

Utilisation de React Hook Form

Introduction à React Hook Form

React Hook Form est une bibliothèque légère pour gérer les formulaires dans React, en se basant sur les hooks pour simplifier le processus. Elle offre des performances optimisées et une syntaxe simple pour une meilleure expérience de développement.

Utilisation des Hooks dans React Hook Form

React Hook Form utilise des hooks comme useForm, useFieldArray, et useWatch pour gérer l'état et le comportement des formulaires, offrant ainsi une approche minimaliste et performante.

```
import { useForm } from 'react-hook-form';

const { register, handleSubmit } = useForm();
```

Configuration de Base

```
npm install react-hook-form
```

```
import { useForm } from 'react-hook-form';

const { register, handleSubmit } = useForm();
```

Pour commencer à utiliser React Hook Form, installez-le dans votre projet React et initialisez l'état du formulaire à l'aide du hook useForm.

Enregistrement des Champs et Gestion des Soumissions

Utilisez register pour enregistrer les champs du formulaire et handleSubmit pour gérer la soumission du formulaire et exécuter une fonction de rappel.

```
<input {...register("email")} />

const onSubmit = (data) => console.log(data);

<form onSubmit={handleSubmit(onSubmit)}>
```

Performance Optimisée et Syntaxe Légère

React Hook Form offre une performance optimisée en réduisant les rerenders inutiles et une syntaxe légère qui simplifie la gestion des formulaires dans les applications React.

13

TypeScript

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.

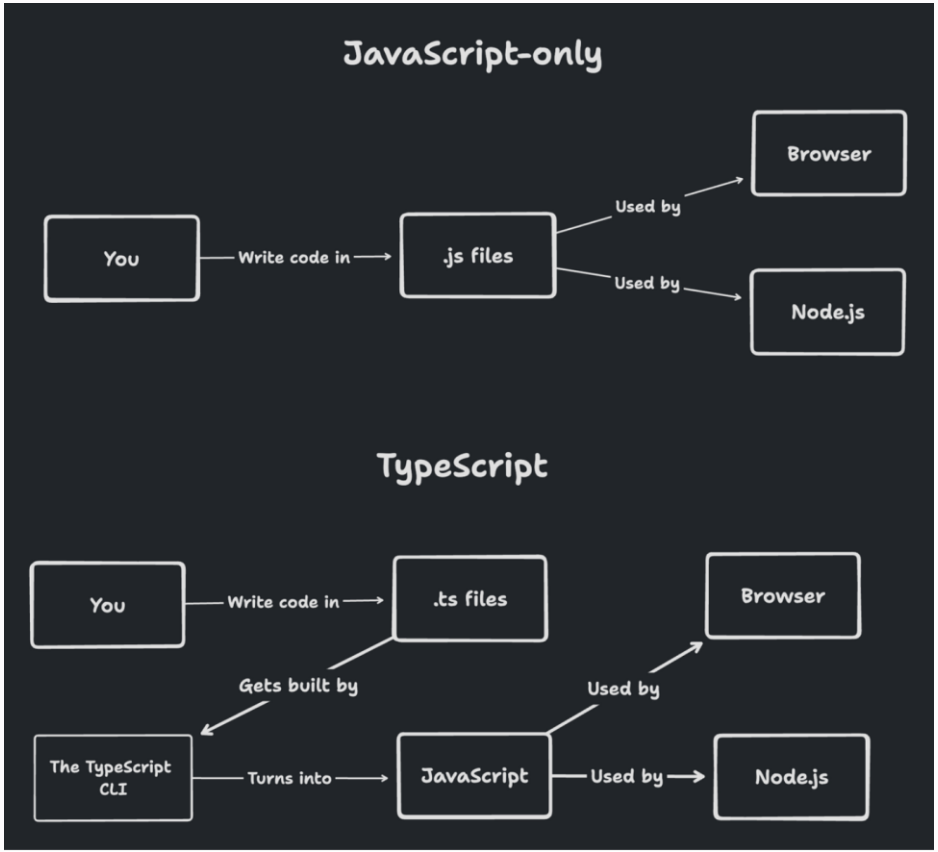
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

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.

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

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

```ts
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 doSometing = (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/mattpocockuk
https://www.youtube.com/@mattpocockuk

14

Utilisation de React 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 re-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
  },
});
```

Pour aller plus loin

TkDodo: https://twitter.com/TkDodo
Le compte du concurrent : (SWR)
https://twitter.com/huozhi

15

Quelques hooks avancés

useMemo

useMemo est un hook qui permet de mémoriser une valeur calculée et de l'empêcher d'être recalculée à chaque rendu du composant. Cela peut être utile pour les valeurs qui sont coûteuses à calculer, comme les fonctions complexes ou les appels à des API.

```
const ProductList = () => {
  const products = [
    { name: "Produit 1", price: 10 },
    { name: "Produit 2", price: 20 },
    { name: "Produit 3", price: 30 },
  ];

  const memoizedTotalPrices = useMemo(() => {
    const totalPrices = products.map(product => product.price);
    return totalPrices;
  }, [products]);

  return (
    <ul>
      {products.map((product, index) => (
        <li key={product.name}>
          {product.name} - {memoizedTotalPrices[index]}
        </li>
      ))}
    </ul>
  );
};
```

memo

```
const Button = memo(({ onClick }) => {
  const [isHovered, setIsHovered] = useState(false);

  const handleMouseEnter = () => {
    setIsHovered(true);
  };

  const handleMouseLeave = () => {
    setIsHovered(false);
  };

  return (
    <button
      onMouseEnter={handleMouseEnter}
      onMouseLeave={handleMouseLeave}
      style={{
        backgroundColor: isHovered ? "red" : "blue",
      }}
      onClick={onClick}
    >
      Bouton
    </button>
  );
});
```

memo est une fonction qui permet d'encapsuler un composant fonctionnel et de l'empêcher de se rendre inutilement. Cela est utile pour les composants purs, qui ne devraient se rendre que si leurs props ou leur état changent.

useCallback

useCallback est un hook qui permet de mémoriser une fonction et de l'empêcher d'être recréée à chaque rendu du composant. Cela peut être utile pour les callbacks qui sont transmis aux composants enfants, car cela évite de les recréer à chaque fois que le composant parent se rend.

```
const CommentList = () => {
  const [sortBy, setSortBy] = useState("date");

  const memoizedSortFn = useCallback((comments, sortBy) => {
    // Fonction de tri
    return comments.sort((a, b) => {
      if (sortBy === "date") {
        return a.date - b.date;
      } else {
        return a.author.localeCompare(b.author);
      }
    });
  }, [sortBy]);

  const sortedComments = memoizedSortFn(comments, sortBy);

  return (
    <ul>
      {sortedComments.map(comment => (
        <li key={comment.id}>
          {comment.author} - {comment.date}
        </li>
      ))}
    </ul>
  );
};
```

NextJS

Support réalisé par Pierrick Hauguel

TABLE DES MATIÈRES

TABLE DES MATIÈRES

TABLE DES MATIÈRES

01
Introduction et Configuration

Qu'est-ce que NextJS ?

Next.js est un framework JavaScript open source basé sur React qui permet de créer des applications web universelles performantes et optimisées pour le référencement naturel (SEO), développé par Vercel.

Next est aujourd'hui le principal framework JS pour créer des applications universelles (côté serveur et client).
Des alternatives sont également disponibles, comme Remix ou Astro.

Les avantages de NextJS

NextJS propose de nombreux avantages, notamment concernant :

- Le SEO, grâce à ses modules de génération statique (SSG) et de rendu côté serveur (SSR)
- Un routage optimisé basé sur l'architecture des fichiers (file-system routing)
- Une communauté active et nombreuse
- Une bonne intégration dans l'écosystème React
- La possibilité d'exécuter du code côté serveur avec les Server Actions et les Route Handlers

Les inconvénients de NextJS

Tout n'est pas parfait, et l'utilisation de Next amène quelques inconvénients :

- Dépendance à Vercel pour les mises à jour
- Un projet de base plus lourd qu'une SPA classique (avec Vite par exemple)
- Un hébergement plus puissant requis par rapport à un projet React classique
- Une gestion du cache parfois énervante

Installation et configuration

Pour créer un projet NextJS, il faut lancer la commande suivante :

```
npx create-next-app@latest
```

On choisira ensuite :

1. Le nom du projet
2. Si on utiliste TypeScript (ici oui)
3. Si on utilise ESLint (oui recommandé)
4. Si on veut utiliser Tailwind (au choix)
5. Si on veut utiliser un répertoire /src (non)
6. Si on veut utiliser le App Router (oui, sinon le projet sera créé avec le Page Router)
7. La configuration de l'alias pour les imports (au choix, ici oui)

Architecture NextJS

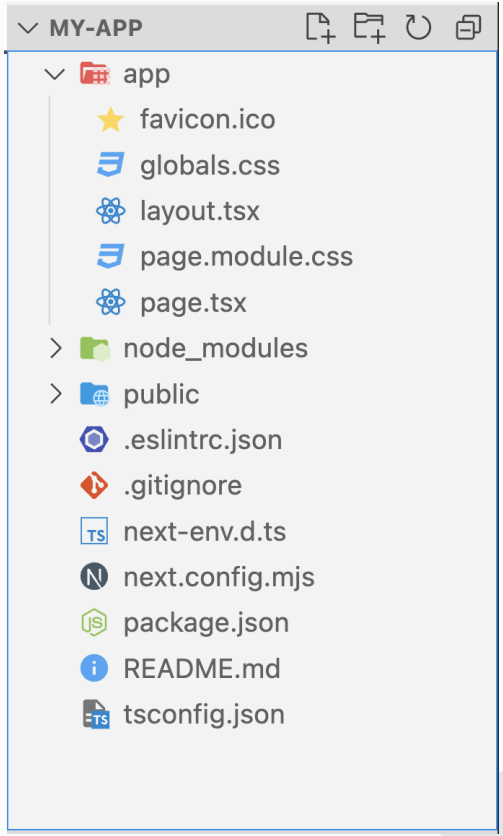
Avant de s'intéresser à la structure du projet, il faut bien faire la distinction entre les deux environnements sur lequel notre code peut tourner.

1. L'environnement client, dans le navigateur, où nous auront accès aux APIs web
2. L'environnement serveur, utilisé pour le SSR et SSG, ainsi que les Server Actions.
    1. Nous n'avons pas accès aux APIs du navigateur ici, donc pas de window.location ou similaire
    2. On a quand même accès au fetch

Structure d'un projet NextJS

Notre projet de base est composé comme tel :

1. /app, le dossier de l'App Router, où nous allons écrire notre code. On rajoutera souvent ici un dossier /ui pour nos composant, et /lib pour la logique métier.
2. Le dossier /public pour les assets
3. Nos fichiers de configuration ESLint, Next, le package.json et TypeScript
4. Les node_modules



02

TypeScript

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.

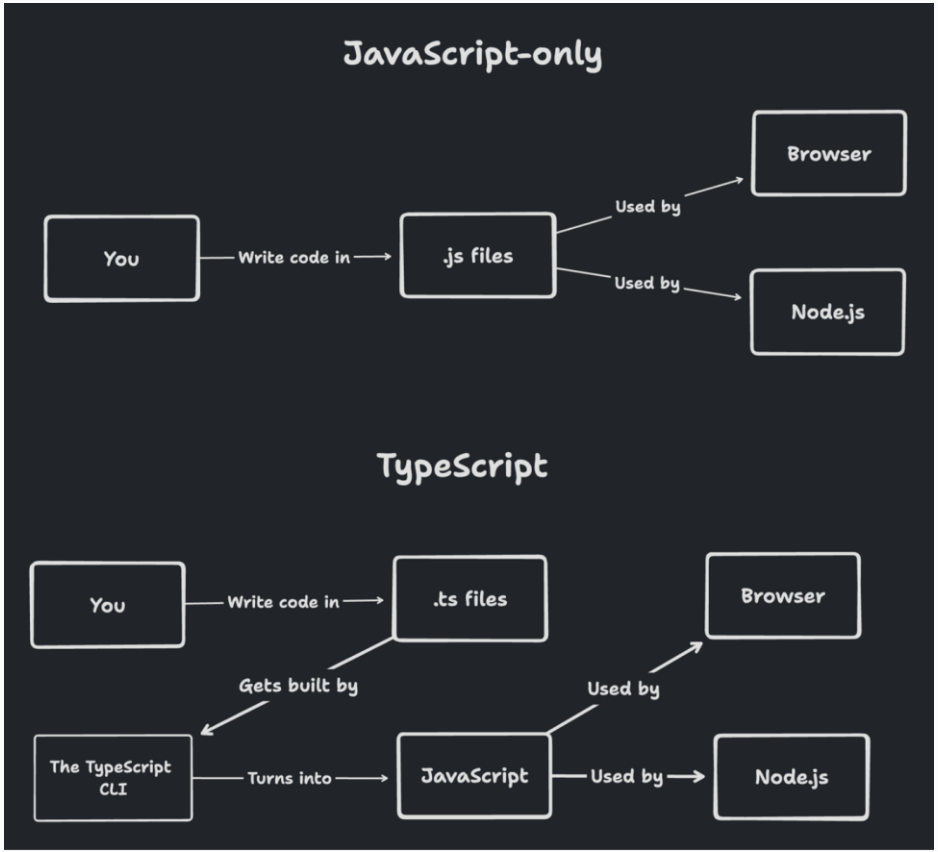
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

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.

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

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

```ts
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 doSometing = (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

03

Pages et App Router

Pages et Router

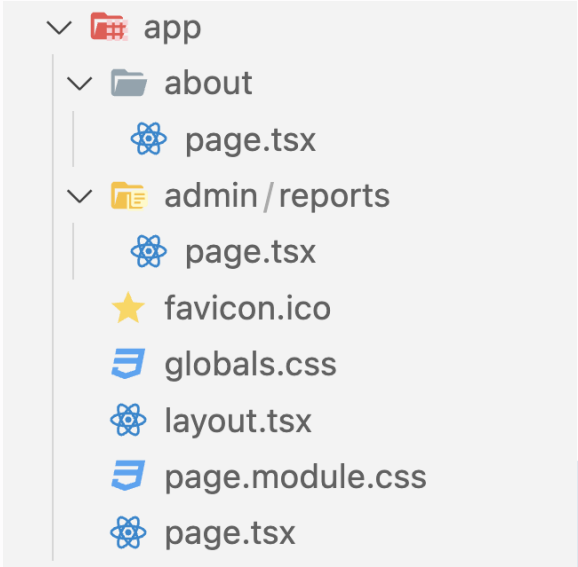
Next utilise son propre système de navigation, nommé App Router, basé sur les fichiers. On parle de file system routing.
Chaque dossier contenu dans le dossier app générera une route pour notre application. La page sera contenue dans un fichier page.js.

Par exemple, un fichier situé à /app/about/page.js générera une route sur /about. Un fichier localisé dans /app/admin/reports/page.js générera une route /admin/reports.

Pages et Router

Next utilise son propre système de navigation, nommé App Router, basé sur les fichiers. On parle de file system routing.
Chaque dossier contenu dans le dossier app générera une route pour notre application. La page sera contenue dans un fichier page.js.

Par exemple, un fichier situé à /app/about/page.js générera une route sur /about. Un fichier localisé dans /app/admin/reports/page.js générera une route /admin/reports.



Pages et Router

Chaque fichier de page doit exporter par défaut un composant React valide pour pouvoir être affiché.

```
const About = ()  => {
    return ( <h1>About</h1> );
}

export default About;
```

Les Layouts

Les layouts sont des composants qui englobent d'autres composants pour fournir une structure cohérente à l'ensemble de l'application.
Ils sont souvent utilisés pour définir la structure de base de la page, telle que la barre de navigation, le pied de page ou d'autres éléments communs à toutes les pages de votre application.
Pour créer un layout, il suffit de créer un fichier layout.js dans le dossier correspondant à la route souhaitée. Toutes les routes enfants utiliseront ce layout.

Les Layouts

/app/dashboard/layout.js

```
export default function DashboardLayout({
  children, // will be a page or nested layout
}) {
  return (
    <section>
      {/* Include shared UI here e.g. a header or sidebar */}
      <nav></nav>

      {children}
    </section>
  )
}
```

Le RootLayout

Le Root Layout est le layout principal de l'application, et sa présence est obligatoire. Il est défini au premier niveau de notre dossier app, et c'est le seul à pouvoir contenir les tags <html> et <body>.

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

Les routes dynamiques

Quand on ignore le nom exact d'un segment en avance (par exemple un id), nous pouvons utiliser les segments dynamiques pour générer des routes dynamiques.

On crée des pages dynamiques en nommant le dossier entre [], avec un nom générique Par exemple, app/posts/[id]/page.js captera les routes, /posts/1, /posts/toto...

Les pages issues de routes dynamiques reçoivent un props "params", qui sera un objet ayant en propriété les segments dynamiques de l'url.

```
export default function Page({ params }) {
  return <div>My Post: {params.id}</div>
}
```

Gérer les erreurs

Pour gérer les erreurs, nous pouvons créer une interface d'erreur dans un fichier error.js.
Cela créera automatiquement une ErrorBoundary. error.js doit nécéssairement s'executer côté client, pensez à utiliser la directive "use client".
Les erreurs seront captées par l'ErrorBoundary la plus proche dans notre hiérarchie.
Les erreurs dans les layouts sont captées par les ErrorBoundaries supérieures.
Le composant d'erreur reçoit en props une fonction reset qui permet de retenter l'action qui a échoué, et les informations sur l'erreur dans un props error.

Gérer les erreurs

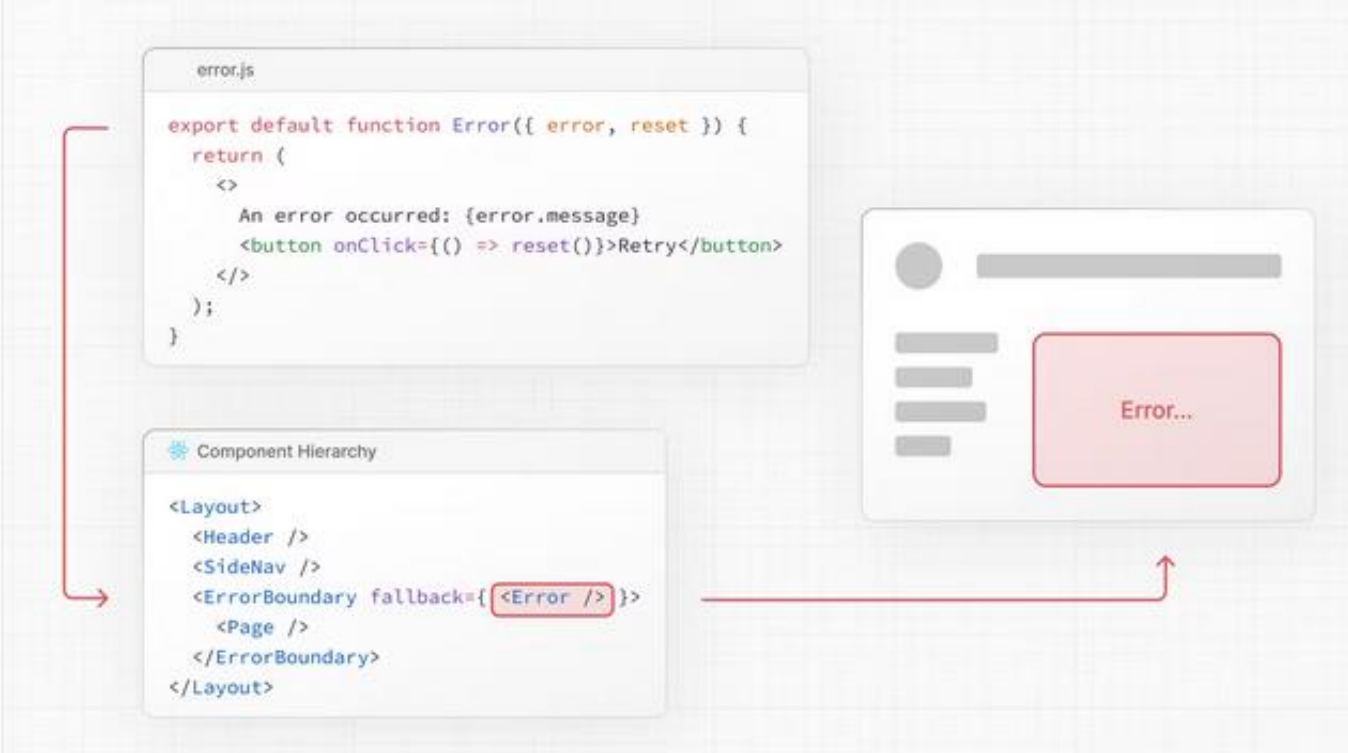
```tsx
'use client' // Error components must be Client Components

import { useEffect } from 'react'

export default function Error({ error, reset }) {
  useEffect(() => {
    // Log the error to an error reporting service
    console.error(error)
  }, [error])

  return (
    <div>
      <h2>Something went wrong!</h2>
      <button
        onClick={
          // Attempt to recover by trying to re-render the segment
          () => reset()
        }
      >
        Try again
      </button>
    </div>
  )
}
```

Gérer les erreurs

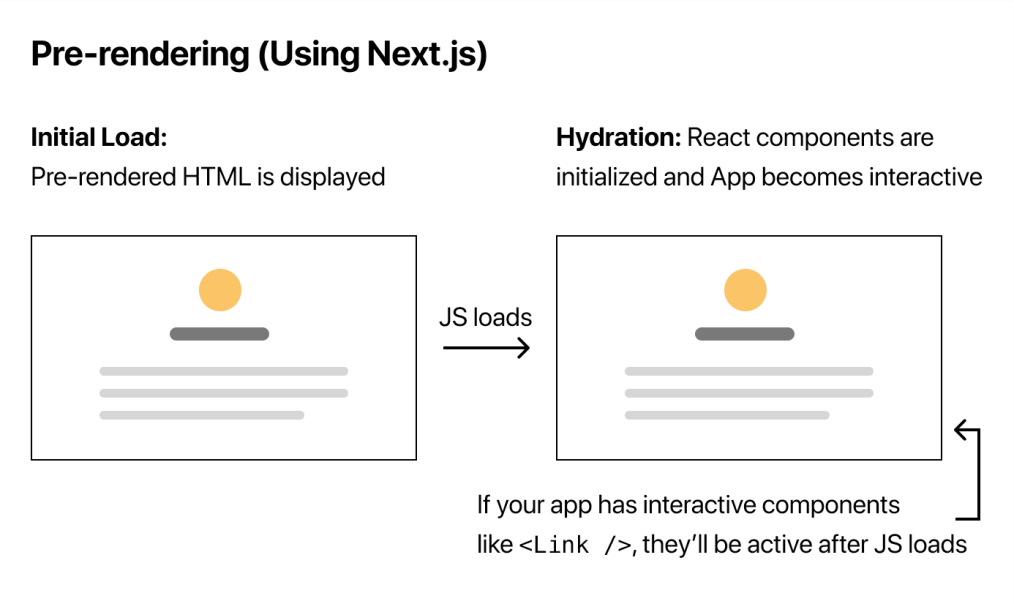


04

Server Side Rendering

Principe du prerendering

Par défaut, Next fait un “pré-rendu” de chaque page de l’application. Cela signifie que Next génère du HTML pour chaque page, en avance, pour limiter la charge du javascript sur le navigateur.



Static Generation et Server Side Rendering

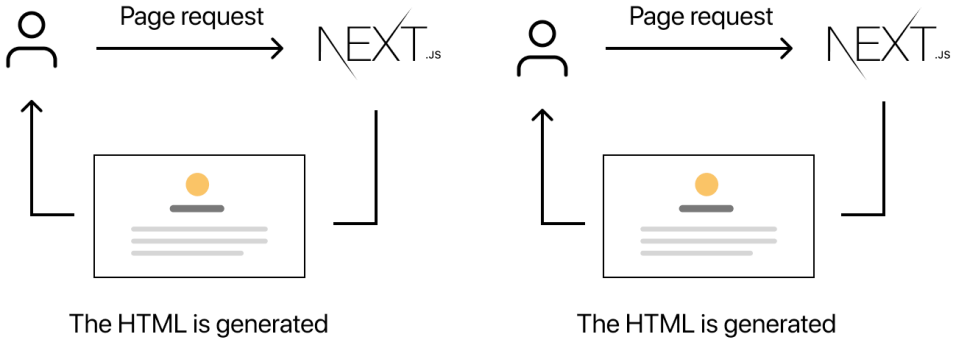
Next propose deux systèmes pour le pre-rendering :

- **La génération statique**, où le HTML est généré lors du build et est réutilisé à chaque requête
- Le **rendu côté serveur**, où le HTML est généré à chaque requête

Server Side Rendering

Server-side Rendering

The HTML is generated on **each request**.



Server Side Rendering

Avec App Router, le choix du SSR est implicite.
Si votre page se situe à une URL comportant un segment dynamique, ou qu'on utilise le props searchParams, la page sera soumise au SSR, sauf si on utilise la directive "use client" en première ligne du fichier.
Dans la slide suivante, nous verrons comment créer un composant serveur (async), qui chargera la donnée côté serveur, pour afficher une page pré-rendue comportant déjà la donnée dont nous avons besoin.
Attention: il n'est pas possible d'utiliser de hooks dans un composant serveur !

Server Side Rendering

```
async function getData() {
  const res = await fetch('https://api.example.com/...')
  return await res.json()
}

export default async function Page() {
  const data = await getData()

  return <main></main>
}
```

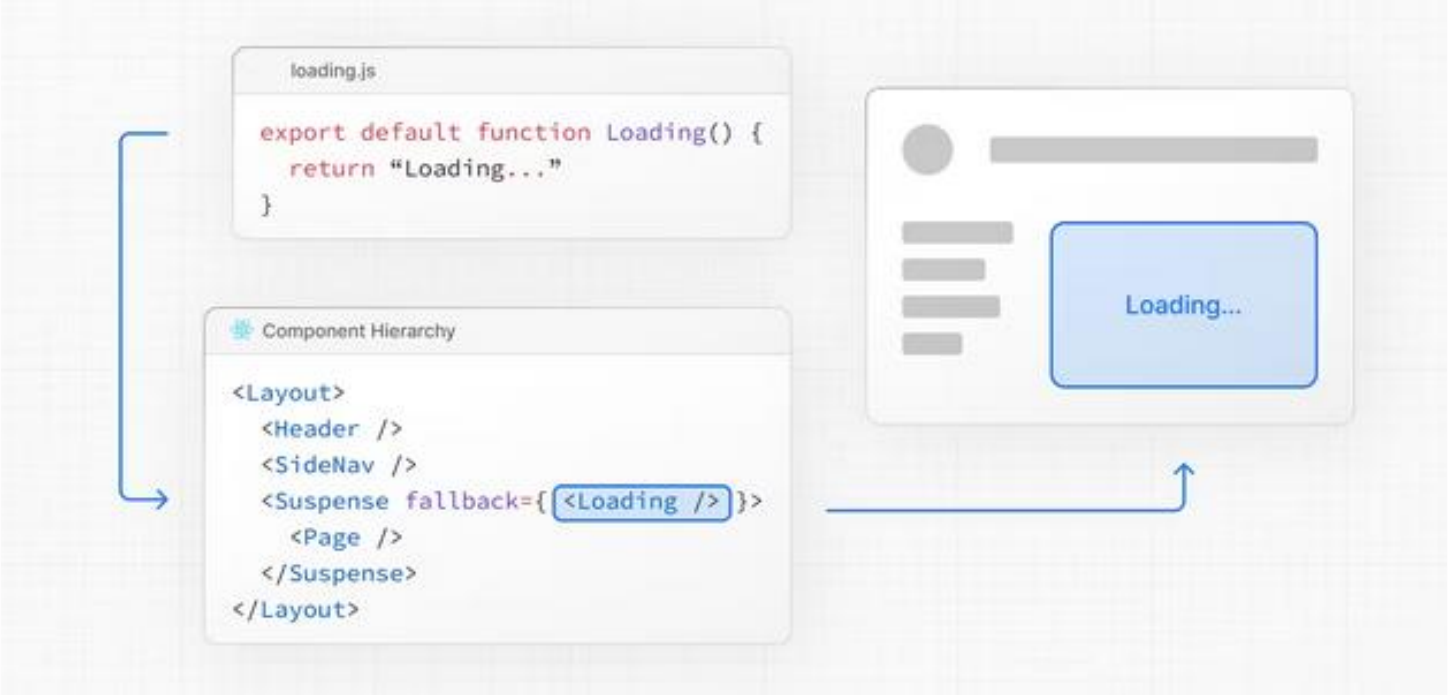
Afficher des loaders

Lorsque nos pages demandent un temps de chargement, nous pouvons afficher une interface d'attente.

Next utilise React Suspense pour nous faciliter la tâche.
Il nous suffit de créer un fichier loading.js dans au même endroit que la page concernée.

Le composant contenu dans le fichier sera affiché à la place de notre page le temps que celle-ci charge

Afficher des loaders



05

Static Site Generation

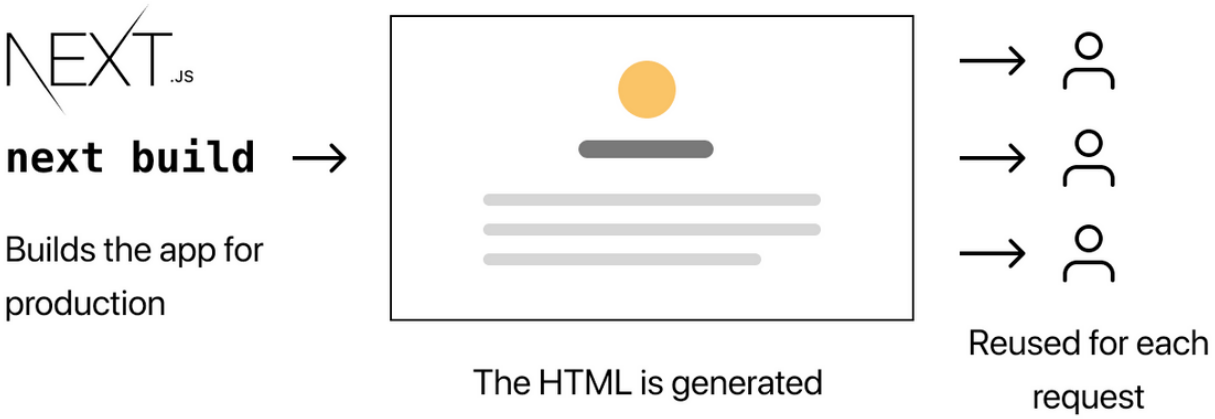
Static Site Generation

La génération statique (SSG), est utilisée pour les pages dont le contenu peut-être connu lors du build de l'application.
Pas de donnée à charger avant l'affichage, pas de routes dynamiques, pas de hooks.
Tout ce qui n'est pas du rendu côté client, ni du SSR sera donc généré statiquement sur le serveur !

Static Site Generation

Static Generation

The HTML is generated at **build-time** and is reused for each request.



06

Optimisation

La navigation

Comme vu précédemment, Next utilise un système de cache pour réduire les temps de chargement sur le site. Il existe aussi une autre façon d'accélérer l'expérience utilisateur, grâce à la balise Link. Next pratique par défaut le « code splitting », c'est-à-dire que chaque page est servie au client sous la forme d'un bundle js dédié (sans code splitting, tout le js du site est servi en une seule fois). La balise Link permet de faire du « prefetching » (comprendre pré-chargement) sur les routes liées à la page actuelle en arrière-plan.

```
import Link from "next/link";

<Link href="/product/1/page">Product 1</Link>;
```

next / image

Image est un composant puissant qui vous aide à optimiser les images pour le Web. Il gère automatiquement le redimensionnement des images, le chargement paresseux et la conversion de format, ce qui se traduit par des temps de chargement plus rapides et une expérience utilisateur améliorée.

```
import Image from "next/image";

<Image src="/me.jpg" alt="me" width={500} height={500} />
```

next / image

Pour utiliser des images « distantes » (non contenues dans le dossier /public), il est nécessaire de définir une liste de patterns d'url spécifique dans notre fichier de configuration Next, next.config.js :

```
const nextConfig = {
  images: {
    remotePatterns: [
      {
        protocol: "https",
        hostname: "s3.amazonaws.com",
        port: "",
        pathname: "/my-bucket/**",
      },
    ],
  },
}
```

next / font

Next font est un système nous permettant de charger et d'optimiser des polices de caractères, qui seront ensuite chargées localement lors du build, pour être servies statiquement.

```
import { Inter } from "next/font/google";
const inter = Inter({ subsets: ["latin"], variable: "--font-inter" });
<body className={inter.className}></body>;
```

07
Route Handlers et Server Actions

Route Handlers

Les Route Handlers de Next.js vous permettent de créer des gestionnaires de requêtes personnalisés pour une route donnée en utilisant les API Web Request et Response. Ils offrent un moyen plus flexible et puissant de gérer les requêtes et les réponses pour des routes spécifiques dans votre application Web.
On les retrouve généralement dans le dossier /app/api/*/route.ts

Elles sont particulièrement pratiques pour récupérer de la donnée côté serveur dans des composants clients.

Attention, pas de route.ts au même niveau qu'une page.ts, sinon il y a un conflit pour la méthode GET !

Route Handlers

```
export async function GET(request: Request) {
  const { searchParams } = new URL(request.url);
  const id = searchParams.get("id");
  const res = await fetch(`https://data.mongodb-api.com/product/${id}`, {
    headers: {
      "Content-Type": "application/json",
      "API-Key": process.env.DATA_API_KEY!,
    },
  });
  const product = await res.json();

  return Response.json({ product });
}

export async function POST(request: Request) {}

export async function PUT(request: Request) {}

export async function DELETE(request: Request) {}

export async function PATCH(request: Request) {}
```

Server Actions

Les Server Actions de Next.js sont des fonctions asynchrones qui s'exécutent sur le serveur. Elles vous permettent d'interagir avec la base de données, les API externes et de modifier l'état de l'application, le tout côté serveur. Vous pouvez utiliser les Server Actions dans les composants côté serveur et client, ce qui vous offre une grande flexibilité dans la gestion de votre logique métier.

```
export default function Page() {
    // Action
    async function create(formData: FormData) {
      'use server';
      // Logic to mutate data...
    }
    // Invoke the action using the "action" attribute
    return <form action={create}>...</form>;
}
```

Server Actions

Si vous souhaitez rediriger votre utilisateur après une action, vous pouvez utiliser la fonction redirect() de next/navigation.
Attention, si vous utilisez un block try/catch, redirect doit être utilisé après celui-ci !
Si vous avez modifié de la donnée chargée sur la page vers laquelle vous redirigez, utilisez également la fonction revalidatePath pour éviter de rencontrer des problèmes de cache !

Server Actions

```
"use server";
import { sql } from "@vercel/postgres";
import { revalidatePath } from "next/cache";
import { redirect } from "next/navigation";

export async function createPost() {
  try {
    //Logique métier
    const result = await sql`INSERT INTO posts...`;
  } catch (error) {
    // gestion de l'erreur
  }
  revalidatePath("/posts");
  redirect("/posts");
}
```