Hes·so VALAIS WALLIS
Haute Ecole de Gestion
Hochschule für Wirtschaft

Filière Informatique de gestion
Studiengang Wirtschaftsinformatik

NEXT.JS

LI : 1ère

Étudiant : Zotrim Uka

Compétence : B,M,

HES-SO Valais-Wallis • rte de la Plaine 2 • CP 80 • 3960 Sierre
+41 58 606 93 11 • info.informatique@hevs.ch • www.hevs.ch/ig

Table des matières

1. Introduction

Next.js est un framework basé sur React, conçu pour simplifier et améliorer le développement web. En utilisant les fonctionnalités bien connues de React, Next.js étend les capacités de création d'applications web, offrant une gamme de fonctionnalités avancées.

L'une des caractéristiques les plus remarquables de Next.js est son système de routage, qui facilite la navigation à travers les différentes pages de l'application. Il permet de créer une application web complète de A à Z en rationalisant la gestion des routes, offrant ainsi une expérience utilisateur fluide.

L'un des avantages les plus significatifs de Next.js est son rendu côté serveur, également appelé "server-side rendering" (SSR). Cela signifie que les pages web générées avec Next.js sont pré-rendues sur le serveur, ce qui améliore considérablement les performances et permet aux utilisateurs de voir le contenu plus rapidement, ce qui est essentiel pour une expérience utilisateur de qualité.

Outre le rendu côté serveur, Next.js prend également en charge le "pré-rendu" statique. Cette fonctionnalité permet de générer des pages HTML statiques pour des parties spécifiques de notre application, ce qui contribue encore davantage à l'optimisation des performances.

Next.js excelle également dans l'optimisation des images, permettant aux développeurs de gérer efficacement les images et de les afficher de manière optimale, améliorant ainsi la vitesse de chargement des pages.

En ce qui concerne le référencement (SEO), Next.js offre des avantages considérables par rapport à React. Il permet aux moteurs de recherche d'indexer facilement le contenu, contribuant ainsi à améliorer la visibilité de notre site web dans les résultats de recherche.

Next.js facilite également la création d'API directement intégrées à notre application, ce qui simplifie la gestion des données et des requêtes.

Enfin, le framework prend en charge l'utilisation des modules CSS, facilitant la gestion et la stylisation des composants de notre application.

02. Rappel React & javascript

JavaScript 2023.pdf

2.1 Javascript (cours openclassrooms – Zotrim):

2.1.1 Fonction fléchées VS Fonction classique

Les fonctions en JavaScript présentent deux principales variations dans leur syntaxe et leur utilité : les fonctions classiques et les fonctions fléchées. Cette distinction repose sur la forme et la manière dont ces fonctions sont définies.

fonction classique : ne fonction classique est déclarée à l'aide du mot-clé function. Par exemple :

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

Fonction fléchée : En contraste, les fonctions fléchées sont définies de manière plus concise en utilisant la syntaxe () =>. Reprenons l'exemple précédent :

```
const additionner = (a, b) => a + b;
```

2.1.2 HOF

Les fonctions de haut niveau, souvent abrégées sous le terme "HOF" (pour "Higher-Order Functions"), jouent un rôle essentiel dans la programmation fonctionnelle en JavaScript. Ce concept se réfère à des fonctions qui prennent d'autres fonctions en tant qu'arguments ou qui renvoient des fonctions en résultat. Les HOF sont particulièrement puissantes et polyvalentes pour la manipulation des données et la modularité du code.

Exemple au point suivant : 2.1.3

2.1.3 Les méthodes des tableaux

Lors de la manipulation de tableaux en JavaScript, deux méthodes fondamentales à connaître sont .map et .filter. Ces méthodes nous permettent d'effectuer des opérations complexes sur les données contenues dans un tableau de manière élégante et efficace.

1. .map - Transformer les Données

La méthode .map est utilisée pour créer un nouveau tableau en appliquant une fonction à chaque élément du tableau d'origine. Cette fonction prend en général un élément du tableau en entrée, effectue une opération sur cet élément, puis renvoie le résultat. Le nouveau tableau résultant contient ces résultats.

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

const nouveauTableau = tableauOriginal.map(element => element * 2);

// nouveauTableau contiendra [2, 4, 6, 8, 10]
```

Dans cet exemple, la méthode .map a multiplié chaque élément du tableau d'origine par 2 pour créer un nouveau tableau.

2. .filter - Filtrer les Données

La méthode .filter est utilisée pour créer un nouveau tableau en filtrant les éléments du tableau d'origine en fonction d'une condition définie dans une fonction. Seuls les éléments pour lesquels la condition est vraie sont inclus dans le nouveau tableau.

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

const nouveauTableau = tableauOriginal.filter(element => element > 2);

// nouveauTableau contiendra [3, 4, 5]
```

Dans cet exemple, la méthode .filter a créé un nouveau tableau contenant uniquement les éléments supérieurs à 2 du tableau d'origine.

Ces deux méthodes, .map et .filter, sont des outils puissants pour manipuler et transformer des tableaux de données en JavaScript. Elles sont largement utilisées dans le développement web pour effectuer des opérations sur des listes d'éléments, ce qui facilite la gestion des données et améliore la lisibilité du code.

2.1.4 Destructuring

La déstructuration (ou destructuring) en JavaScript permet d'extraire des valeurs d'objets {} et de tableaux [] en utilisant des syntaxes spécifiques : {} pour les objets et [] pour les tableaux. Cette fonctionnalité facilite la manipulation des données en associant des noms de propriétés ou des indices à des variables pour extraire des valeurs. Nous pouvons également définir des valeurs par défaut pour gérer les cas où les valeurs extraites sont indéfinies. La déstructuration est un outil précieux pour simplifier le code JavaScript et améliorer la gestion des données.

2.2 React : Pour plus de détail se référer à la LI de Rafael

2.2.1 Création d'un app react

Pour créer un projet React en utilisant la ligne de commande, nous utilisons la commande suivante : npx create-react-app nomDuProjet

Une fois le projet ouvert dans le langage de programmation, il faut lancer le projet dans le terminal en utilisant la commande : npm run start

2.2.2 le JSX

JSX, qui signifie "JavaScript XML," est une extension de syntaxe couramment utilisée dans le développement d'applications React pour créer des interfaces utilisateur dynamiques et réactives. Elle permet d'incorporer des éléments HTML dans du code JavaScript, offrant ainsi une manière intuitive de décrire la structure de l'interface utilisateur.

Dans JSX, on utilise des balises HTML pour créer des éléments, et voici quelques exemples de balises couramment utilisées :

- ** ** : Utilisée pour créer des conteneurs ou des sections dans un composant React. **<h1>**, **<p>**, **<a>**, **<img>** : Balises pour les titres, les paragraphes, les liens et les images, respectivement, etc.

Dans JSX, pour ajouter des classes CSS à des éléments HTML, on utilise l'attribut **className** au lieu de **class**.

Toutes les balises auto-fermantes, comme
, doivent être fermées de la même manière, bien que l'on puisse également les fermer sans le slash dans JSX, par exemple
.

Un élément JSX doit toujours être encapsulé dans une seule balise parente. Cela signifie qu'on doit avoir un seul conteneur englobant tous les autres éléments JSX, généralement une

On peut mélanger JavaScript et HTML en utilisant des accolades {} pour incorporer des expressions JavaScript dans le JSX. Par exemple, si on a une variable data contenant la valeur 15 : <h1>{data}</h1>

Pour le contenu conditionnel dans JSX, on peut utiliser des opérateurs ternaires pour afficher différentes parties de l'interface utilisateur en fonction de conditions. Par exemple :

```
const loggedIn = true;
return (
  <div>
    {loggedIn ? <p>Bienvenue, utilisateur connecté !</p> : <p>Connectez-vous pour accéder au contenu.</p>}
  </div>
);
```

L'opérateur "short-circuit" (&&) est également couramment utilisé pour le contenu conditionnel. Il permet d'afficher un élément JSX uniquement si une condition est vraie. Par exemple :

```
const afficherElement = true;
return (
  <div>
    {afficherElement && <p>Cet élément s'affichera si afficherElement est vrai.</p>}
  </div>
);
```

2.2.3 les composants

Les composants en React sont définis par des fonctions ou des classes, et il est essentiel de les exporter pour les rendre accessibles à d'autres parties de l'application. Ces composants ont pour objectif de retourner du contenu, qu'il s'agisse de données sans état (stateless) ou de données avec état (stateful).

Un composant simple peut être défini comme suit :

function App() { return (// Contenu du composant); } export default App;

Dans cet exemple, la fonction **App** est un composant React. Elle retourne le contenu que nous souhaitons afficher à l'écran.

Un concept clé en React est la relation entre les composants parents et enfants. Dans une application React, les composants sont souvent imbriqués les uns dans les autres, créant ainsi une hiérarchie de composants. Cette hiérarchie peut être vue comme une structure d'arborescence, où un composant parent englobe un ou plusieurs composants enfants.

Les composants parents sont responsables de la création et de la gestion des composants enfants. Ils peuvent transmettre des données ou des propriétés aux composants enfants sous forme de "props" (abréviation de "properties"). Les composants enfants, à leur tour, reçoivent ces "props" et peuvent les utiliser pour personnaliser leur comportement ou leur affichage.

Voici un exemple simple de composant parent et enfant :

```
function ParentComponent() { const data = "Données du parent"; return ( <div> <ChildComponent data={data} /> </div> ); } function ChildComponent(props) { return ( <p>{props.data}</p> ); } export default ParentComponent;
```

Dans cet exemple, **ParentComponent** est le composant parent et **ChildComponent** est le composant enfant. **ParentComponent** transmet des données à **ChildComponent** via la "prop" **data**, et ce dernier l'affiche à l'écran.

Cette relation parent-enfant est fondamentale pour la construction d'applications React modulaires et maintenables, où chaque composant a une responsabilité spécifique et peut être réutilisé à plusieurs endroits de l'application.

2.2.4 le fonctionnement de l'app

Dans le cadre de notre projet, plusieurs dossiers jouent un rôle essentiel. Tout d'abord, il y a le dossier "public" où se trouve le fichier "index.html". Ce fichier est la page HTML principale de notre application React. Il sert de point d'entrée pour le chargement de l'application dans le navigateur. À l'intérieur du dossier "public", se trouvent également des images, y compris le favicon, qui est l'icône affichée dans l'onglet du navigateur. Ces ressources statiques sont accessibles directement depuis le navigateur sans nécessiter de traitement spécial.

Ensuite, il y a le dossier "node_modules". Ce dossier est essentiel car il contient toutes les dépendances et les modules nécessaires au bon fonctionnement de notre application. Lorsque nous installons des bibliothèques ou des packages via npm (Node Package Manager), ils sont téléchargés et stockés dans ce dossier. Cela garantit que notre application a accès à toutes les fonctionnalités et bibliothèques dont elle a besoin.

Le dossier le plus important est "src" (abréviation de "source"). Il s'agit du cœur de notre application React. À l'intérieur du dossier "src", nous organisons le code source JavaScript, y compris les fichiers de composants, les styles, et la logique métier. Les fichiers JavaScript écrits, tels que les composants React, sont généralement stockés dans ce dossier. C'est ici que la magie opère, où nous définissons la structure de notre application, créons des composants réutilisables, et mettons en œuvre la logique qui donne vie à notre interface utilisateur.

2.2.5 les props

Les "props," abréviation de "propriétés," sont en effet des données que nous pouvons transmettre à des composants dans le but de personnaliser leur comportement ou leur affichage. Elles permettent de passer des informations d'un composant parent à un composant enfant au sein de l'architecture React.

En React, les "props" sont utilisées pour rendre les composants réutilisables et configurables. Un composant parent peut définir des "props" et les transmettre à ses composants enfants. Les composants enfants peuvent ensuite accéder à ces "props" et les utiliser pour afficher des données dynamiques ou adapter leur comportement en fonction des valeurs des "props" reçues.

Les "props" sont un mécanisme puissant pour la communication entre les composants et sont largement utilisées dans le développement d'applications React pour personnaliser et rendre les interfaces utilisateur dynamiques en fonction des données et des paramètres passés aux composants.

2.2.6 le State avec useState

Les hooks sont des méthodes qui vont nous procurer certaines fonctionnalités essentielles pour la gestion de l'état et le cycle de vie des composants.

Un hook couramment utilisé est **useState**. Il nous permet de déclarer un état local à un composant en utilisant la syntaxe suivante :

```
const [state, setState] = useState(initialValue);
```

Lorsque nous utilisons **useState**, nous pouvons initialiser l'état avec n'importe quel type de données, que ce soit une chaîne de caractères, un nombre, un tableau, etc.

Lorsque l'état change grâce à **setState**, le composant se met automatiquement à jour et se réaffiche pour refléter ce nouvel état. Pour déclencher un changement d'état, il est généralement nécessaire d'avoir un événement, par exemple, en associant une fonction à un événement tel que **onClick** sur un bouton :

```
<button onClick={() => setState(nouvelleValeur)}>Changer l'état</button>
```

Dans cet exemple, lorsque le bouton est cliqué, la fonction **setState** est appelée avec une nouvelle valeur, ce qui entraîne la mise à jour de l'état et le rafraîchissement du composant pour refléter cette modification.

Les hooks, tels que **useState**, sont une partie fondamentale du développement d'applications React modernes. Ils nous permettent de gérer efficacement l'état des composants et de rendre les interfaces utilisateur réactives aux interactions de l'utilisateur.

2.2.7 Retourner une liste

Avec **useState**, nous pouvons également initialiser l'état avec un tableau d'objets, comme ceci :

```
const [state, setState] = useState([ { id: 1, nom: "Enzo" }, { id: 2, nom: "Julie" }, { id: 3, nom: "Bastien" }, ]);
```

Lorsque nous souhaitons afficher la liste de ces objets dans notre interface utilisateur, nous utilisons la méthode **.map**, qui itère à travers chaque élément du tableau pour le transformer en un élément HTML. Par exemple, pour afficher cette liste dans une liste à puces (<**ul**>) en utilisant des éléments <**li**>, nous pouvons faire ce qui suit :

```
<ul> {state.map(item => ( <li key={item.id}>{item.nom}</li> ))} </ul>
```

Il est important de noter que nous ajoutons un attribut **key** à chaque élément <**li**> avec la valeur de **item.id**. Cette clé est cruciale pour React, car elle permet de suivre chaque élément de manière unique. Elle aide React à savoir quel élément a été modifié, supprimé ou ajouté lors des mises à jour de l'état. Les clés doivent être uniques au sein de la liste et stables dans le temps, généralement associées à une valeur unique telle qu'un identifiant (**id** dans notre cas).

L'utilisation appropriée de la clé est une bonne pratique en React pour garantir que les mises à jour de l'interface utilisateur sont efficaces et sans erreurs lors de l'affichage de listes dynamiques.

2.2.8 Utiliser du CSS

Nous avons la possibilité de créer un dossier nommé "components" dans lequel nous organisons nos composants réutilisables. À l'intérieur de ce dossier, nous pouvons créer des sous-dossiers pour regrouper des composants similaires. Par exemple, nous pouvons avoir un dossier "Card" contenant les fichiers "Card.js" pour le composant et "Card.css" pour les styles associés.

Pour créer un mécanisme de basculement (toggle), nous pouvons utiliser l'état (**useState**) en l'initialisant à **false** comme ceci :

```
const [state, setState] = useState(false);
```

Ensuite, pour effectuer le basculement, nous créons une fonction qui inverse l'état actuel. Voici comment cela pourrait être fait :

```
const toggleState = () => { setState(!state); };
```

Lorsque nous appelons **toggleState**, cela inversera la valeur actuelle de **state**, c'est-à-dire de **true** à **false** ou de **false** à **true**. Cela permet de basculer entre deux états, ce qui peut être utile pour afficher ou masquer des éléments dans l'interface utilisateur en réponse à des événements tels que les clics de boutons.

Cette technique est couramment utilisée pour créer des fonctionnalités interactives dans les applications React, telles que les boutons de basculement, les panneaux expansibles, et bien d'autres. Elle repose sur le concept fondamental de gestion de l'état dans React pour rendre les composants dynamiques et réactifs aux actions de l'utilisateur.

2.2.9 Les « controlled » et les « uncontrolled » inputs

Nous continuons à explorer les concepts liés à la gestion des formulaires en React. Tout d'abord, nous avons la déclaration de l'état du formulaire avec **useState**, ainsi que la fonction de gestion **handleInput** :

```
const [inpData, setInpData] = useState(""); // Initialisation de l'état du formulaire const handleInput = (e) => setInpData(e.target.value); // Fonction de gestion pour mettre à jour l'état
```

Dans cet exemple, **inpData** représente la valeur du champ de saisie, et **setInpData** est utilisé pour mettre à jour cette valeur en fonction de l'entrée de l'utilisateur.

Nous créons ensuite un formulaire avec un champ de saisie contrôlé (controlled input) :

```
<form> <label htmlFor="nom">Notre prénom</label> <input id="nom" type="text" value={inpData} onChange={handleInput} /> <button>Validez</button> </form>
```

Ce champ de saisie est contrôlé par React grâce à l'utilisation de l'attribut **value** lié à **inpData** et de l'événement **onChange** qui déclenche la fonction **handleInput** à chaque modification du champ.

Il est important de noter qu'il existe deux méthodes courantes pour gérer les formulaires en React : les "controlled inputs" et les "uncontrolled inputs". Les "controlled inputs" sont liés par l'attribut **value**, et à chaque modification, l'événement **onChange** est déclenché pour mettre à jour l'état. C'est généralement utilisé lorsque nous avons besoin d'une prévisualisation des données.

En revanche, les "uncontrolled inputs" utilisent le hook **useRef**. Dans ce cas, nous n'utilisons pas **value** ni **onChange**, ce qui peut être utile lorsque nous n'avons pas besoin d'une visualisation immédiate des données. Voici comment cela peut être implémenté :

```
const inpRef = useRef(); const handleForm = (e) => { e.preventDefault(); console.log("Données du champ non contrôlé : ", inpRef.current.value); }; return ( <form onSubmit={handleForm}> <label htmlFor="nom">Notre prénom</label> <input ref={inpRef} id="nom" /> <button>Validez</button> </form> );
```

Dans cet exemple, nous utilisons **useRef** pour accéder aux données du champ de saisie non contrôlé lorsque le formulaire est soumis. Cela peut être utile dans des situations telles que la saisie d'un e-mail ou d'un mot de passe, où la visualisation en temps réel n'est pas nécessaire.

2.2.10 Les « React fragments »

Les "React fragments" sont une fonctionnalité clé en React qui permet de simplifier la structure du code en évitant d'ajouter une balise parente inutile autour de plusieurs éléments adjacents. Auparavant, il était courant d'utiliser une balise <**div**> pour englober ces éléments, mais cela pouvait entraîner une structure HTML peu souhaitable. Les "React fragments" résolvent ce problème en permettant de retourner plusieurs éléments sans nécessiter de balise parente.

Pour utiliser des fragments, on peut simplement entourer les éléments avec <> et </>. Par exemple :

```
return ( <> <h1>Titre</h1> <p>Paragraphe 1</p> <p>Paragraphe 2</p> </> );
```

Cela permet de maintenir une structure HTML propre et d'éviter d'ajouter des éléments inutiles à la hiérarchie du DOM. Les "React fragments" sont particulièrement utiles lorsque l'on travaille avec des composants de plus haut niveau qui attendent un seul élément enfant, car ils permettent de regrouper plusieurs éléments sans introduire de conteneur inutile.

2.2.11 le hook « useEffect »

En rapport avec le cycle de vie des composants, les "React hooks" incluent un hook essentiel appelé **useEffect**. Ce hook est utilisé pour effectuer des actions après que le composant soit monté, et il est essentiel pour la gestion des effets secondaires dans vos composants React.

Voici comment déclarer et utiliser le hook **useEffect** :

```
import { useEffect } from 'react'; function MonComposant() { // ... useEffect(() => { // Code à exécuter après que le composant soit monté // Exemple : Mettre en place un abonnement à un événement window.addEventListener('scroll', handleScroll); // Exemple : Nettoyer l'effet lorsque le composant est démonté return () => { window.removeEventListener('scroll', handleScroll); }; }, []); // Le tableau vide signifie que cet effet s'exécutera uniquement après le montage initial // ... return ( // ... ); }
```

Dans cet exemple, nous importons tout d'abord le hook **useEffect** depuis React. Ensuite, à l'intérieur de la fonction du composant, nous déclarons **useEffect**. À l'intérieur de la fonction de **useEffect**, nous pouvons placer le code que nous voulons exécuter après que le composant soit monté.

Il est important de noter que nous passons un tableau vide **[]** en tant que deuxième argument à **useEffect**. Cela signifie que cet effet ne s'exécutera qu'après le montage initial du composant. Si nous avions une dépendance dans ce tableau (par exemple, une variable d'état), l'effet s'exécuterait chaque fois que cette dépendance changerait.

Le hook **useEffect** est particulièrement utile pour effectuer des opérations telles que la gestion d'abonnements à des événements, des appels à des API, ou tout autre effet secondaire nécessaire dans vos composants React. Il permet de synchroniser ces opérations avec le cycle de vie du composant de manière propre et efficace.

2.2.12 Créer un router

Dans le contexte de la création d'un routeur (router) en React, nous avons un exemple concret de structure de fichiers. Imaginons que nous souhaitons créer une barre de navigation

(Navbar) pour notre application et utiliser un routeur pour gérer la navigation entre différentes pages. Voici comment organiser les fichiers :

Tout d'abord, nous créons un dossier "Navbar" dans notre projet. À l'intérieur de ce dossier, nous avons deux fichiers :

1. **Navbar.js** : C'est le fichier qui contiendra notre composant de barre de navigation (Navbar). Ce composant définira les liens ou boutons permettant à l'utilisateur de naviguer entre les différentes pages de l'application.

2. **Navbar.css** : Ce fichier contient les styles CSS spécifiques à notre barre de navigation. Nous pouvons personnaliser l'apparence de la barre de navigation en utilisant des classes CSS définies ici.

Ensuite, une fois que nous avons créé le composant de la barre de navigation dans **Navbar.js**, nous pouvons l'intégrer dans notre application React globale. C'est à ce moment-là que nous pouvons également configurer un routeur (par exemple, React Router) pour gérer la navigation entre les pages de l'application.

La mise en place d'un routeur permettra de définir des itinéraires pour chaque page de l'application, de manière que lorsque l'utilisateur clique sur un lien dans la barre de navigation, le routeur affiche la page correspondante sans recharger complètement la page.

La création d'un routeur et la gestion de la navigation entre les pages peuvent être plus détaillées, en fonction des besoins spécifiques de notre application. Cependant, la première étape consiste à organiser proprement les fichiers, comme décrit ci-dessus, en créant un dossier distinct pour chaque composant (dans ce cas, la barre de navigation) et en définissant les styles associés dans un fichier CSS séparé.

2.2.13 Utiliser l'API de contexte

L'utilisation de l'API de contexte est une fonctionnalité puissante de React qui permet de partager des données entre différents composants de notre application, sans avoir à passer ces données via des "props" à travers plusieurs niveaux de composants. Cela simplifie la gestion de l'état global de notre application.

Pour commencer, nous devons créer un contexte en utilisant **createContext()** :

```
export const DataContext = createContext();
```

Ce contexte agira comme un "conteneur" pour les données que nous souhaitons partager.

Ensuite, nous créons un composant qui servira de fournisseur de contexte. Ce composant sera responsable de la définition des données que nous souhaitons partager et de les mettre à disposition pour d'autres composants :

```
const ContextProvider = (props) => { const [data, setData] = useState(1234567); // Exemple de donnée à partager return ( <DataContext.Provider value={{ data }}> {props.children} </DataContext.Provider> ); }; export default ContextProvider;
```

Dans cet exemple, **ContextProvider** est un composant qui utilise le hook **useState** pour définir une donnée (**data**) que nous souhaitons partager. Ensuite, il enveloppe ses enfants dans le **DataContext.Provider**, en fournissant **data** comme valeur pour le contexte.

Enfin, nous enveloppons notre application avec ce fournisseur de contexte au niveau le plus élevé de notre hiérarchie de composants. Par exemple, dans notre composant principal ou dans l'index de notre application :

```
import React from 'react'; import ReactDOM from 'react-dom'; import App from './App'; import ContextProvider from './ContextProvider'; // Importons le fournisseur de contexte ReactDOM.render( <React.StrictMode> <ContextProvider> <App /> </ContextProvider> </React.StrictMode>, document.getElementById('root') );
```

Maintenant, toutes les composantes de notre application peuvent accéder aux données partagées à l'aide du contexte, sans avoir besoin de les transmettre via des "props". Pour ce faire, elles doivent simplement importer le contexte (**DataContext** dans cet exemple) et utiliser le hook **useContext** pour accéder aux données.

Cela simplifie grandement la gestion de l'état global dans notre application React, en évitant le "forage des props" à travers plusieurs niveaux de composants.

3. Le Routing avec Next

3.1 Mise en place

Dans le contexte de la mise en place du routage avec Next.js, il est important de noter que Next.js est un framework basé sur React. Contrairement à React, qui est une bibliothèque JavaScript pour la création d'interfaces utilisateur, Next.js fournit un cadre de travail plus complet pour le développement d'applications web, y compris des fonctionnalités avancées telles que le routage.

Pour commencer un projet Next.js en ligne de commande, on peut utiliser la commande suivante :

```
npx create-next-app nomProjet
```

Cela générera la structure de base du projet Next.js, comprenant plusieurs dossiers importants :

1. **node_modules** : Ce dossier contient toutes les dépendances nécessaires au bon fonctionnement du projet. Il est généré automatiquement lors de l'installation des packages nécessaires via **npm** ou **yarn**.

2. **pages** : Dans ce dossier, se trouvent toutes les pages de l'application Next.js. Chaque fichier JavaScript ou TypeScript dans ce dossier représente une page de l'application. Par exemple, **index.js** représente la page d'accueil, **about.js** représente la page "À propos", etc.

3. **public** : Le dossier "public" est destiné aux ressources statiques telles que les images, les fichiers CSS, les fichiers JavaScript, etc. Ces fichiers sont accessibles directement depuis l'URL du site.

4. **styles** : Ce dossier contient les fichiers CSS pour la stylisation de l'application. On peut utiliser ces fichiers pour définir des styles globaux ou spécifiques à des composants.

Une fois que le projet est configuré, on peut lancer le serveur de développement en utilisant la commande suivante : npm run dev

Cela démarre l'application Next.js localement, et on peut la visualiser dans le navigateur en accédant à l'adresse **http://localhost:3000**.

3.2 Créer des routes

Pour créer des routes dans Next.js, on peut utiliser le dossier "pages". Par exemple, dans notre projet Next.js, le fichier **index.js** est considéré comme la page d'accueil (home page) par défaut, de la même manière que dans React, on déclare un composant comme page principale.

Voici un exemple simple pour créer des routes dans Next.js :

Figure 1: Ressemblance à React

Pour créer des routes dans Next.js, nous utilisons le dossier "pages". Chaque fichier JavaScript que nous créons dans ce dossier devient une page de notre application. Par exemple, si nous avons un fichier nommé contact.js dans le dossier "pages", nous pouvons accéder à la page de contact en utilisant l'URL correspondante, comme http://localhost:3000/contact. Voici comment cela fonctionne :

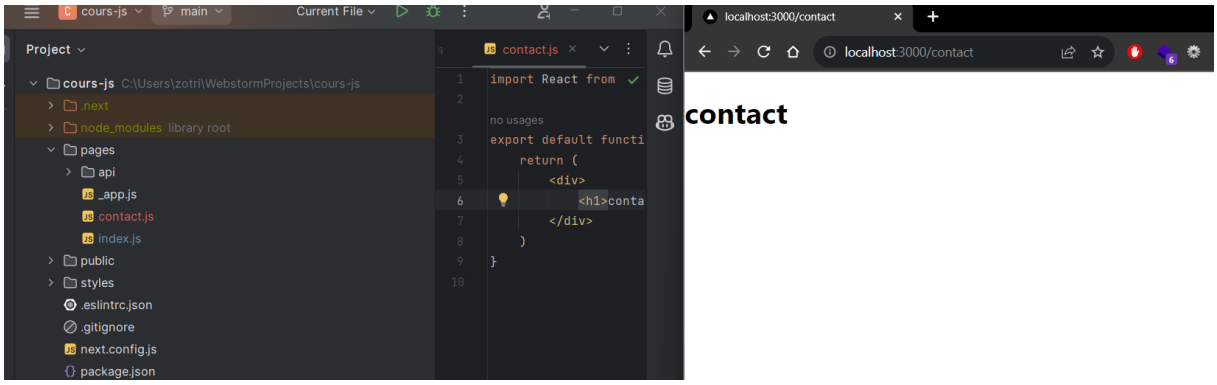

Figure 2 : Créer une nouvelle page

Si nous souhaitons créer des sous-dossiers pour organiser nos pages, nous devons nous assurer d'inclure un fichier index.js à l'intérieur de chaque sous-dossier. Ce fichier index.js représente la page principale du sous-dossier et est accessible via l'URL portant le nom du

sous-dossier. Par exemple, un sous-dossier "blog" avec un fichier index.js sera accessible via l'URL /blog.

3.3 Créer des routes dynamiques

Pour créer des routes dynamiques dans Next.js, nous utilisons généralement des fichiers avec un nom entre crochets, par exemple [slug].js. Cette approche permet de gérer des routes qui acceptent des paramètres variables dans l'URL. Voici comment cela fonctionne :

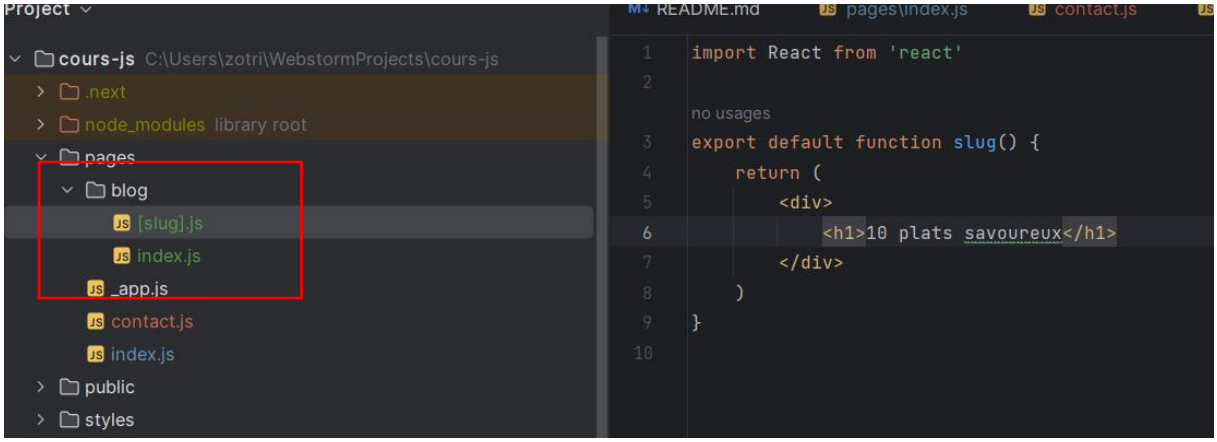

Figure 3 : Route dynamique

De plus, si nous avons besoin de gérer des chemins d'URL avec plusieurs segments variables, nous pouvons utiliser un fichier **[...slug].js**. Ce type de fichier nous permet de traiter les paramètres de l'URL comme un tableau, ce qui nous donne la flexibilité de gérer des chemins d'URL contenant plusieurs segments variables.

En utilisant ces fichiers de routage dynamique, nous pouvons créer des pages web qui s'adaptent facilement aux besoins de notre application, en rendant la gestion des paramètres d'URL et des chemins flexibles et puissante.

Figure 4 : Extension

3.4 Les liens

Dans Next.js, pour créer des liens vers différentes pages, nous utilisons la dépendance Link. En important Link depuis "next/link", nous pouvons utiliser les balises Link pour définir des liens vers nos différentes pages. Le lien est spécifié en utilisant l'attribut href à l'intérieur de la balise Link. Cela nous permet de naviguer facilement entre les pages de notre application sans avoir à recharger la page complète.

```
export default function Home() {
    return (
        <div className={styles.container}>
            <h1>Home</h1>
            <a href="https://www.google.com">Google</a>
                <Link href="/blog">
                <a> href=Blog</a>
                </Link>
                </div>
                )
            }
```

Figure 5 : Balise Link

En Next.js, pour créer des liens dynamiques, nous déclarons une variable et utilisons des backticks (``) pour définir le lien de manière dynamique. Cette approche nous permet de générer des liens en fonction de données variables, ce qui peut être particulièrement utile pour créer des liens vers des pages avec des paramètres dynamiques dans l'URL. Il est essentiel de mentionner que pour incorporer des variables dans le lien, nous utilisons également le signe "$". Ainsi, nous pouvons facilement naviguer entre les pages de notre application sans avoir à recharger la page complète.

```
export default function Home() {
   const id : string  = "article";
    return (
        <div className={styles.container}>
            <h1>Home</h1>
            <a href="https://www.google.com">Google</a>

                <Link href={`/blog/${id}`}>
                 Blog
                </Link>
                </div>
                )
            }
```

Figure 6 : Lien dynamique

3.5 Créer des composants classiques

Il est crucial de maintenir une distinction claire entre les composants et les pages dans Next.js. Pour ce faire, il est recommandé de créer un dossier spécifique nommé "components". Dans cet exemple, un dossier "Navbar" a été créé à l'intérieur du dossier "components", et à l'intérieur du dossier "Navbar", un fichier "Navbar.js" a été ajouté. Cette organisation permet de séparer clairement les composants réutilisables de la logique des pages, ce qui simplifie la gestion et la structure de l'application.

```
import React from 'react'
import Link from "next/link";

no usages  new *
export default function Navbar() {
    return (
        <nav>
            <Link href="/">
                Accueil
            </Link>
            <Link href="/blog">
                Blog
            </Link>
            <Link href="/contact">
                Contact
            </Link>
        </nav>
)
}
```

Figure 7 : Navbar

Après avoir créé le composant Navbar dans le fichier "Navbar.js" à l'intérieur du dossier "components", nous pouvons l'importer dans notre page principale, généralement dans le fichier "index.js". Pour ce faire, nous utilisons la balise <Navbar/> dans le fichier "index.js". Cette étape permet d'inclure le composant de la barre de navigation dans la page principale de notre application et de l'afficher à l'endroit souhaité.

3.6 Utiliser useRouter

le hook **useRouter** est une fonctionnalité essentielle de Next.js qui permet de gérer la navigation au sein de notre application. Il est utilisé pour accéder aux informations de l'URL et pour déclencher des changements de page programmablement.

Voici comment l'utiliser généralement :

Tout d'abord, nous importons le hook **useRouter** depuis "next/router".

```
import { useRouter } from 'next/router';
```

Ensuite, nous l'utilisons dans notre composant en le déclarant comme suit :

```
const router = useRouter();
```

Une fois que nous avez accès à **router**, nous pouvons utiliser différentes propriétés et méthodes pour gérer la navigation. Par exemple :

router.pathname : Nous donne le chemin d'URL actuel.

router.query : Nous permet d'accéder aux paramètres de l'URL.

router.push() : Nous permet de naviguer vers une autre page de manière programmatique.

router.back() : Nous permet de revenir en arrière dans l'historique de navigation.

Voici un exemple d'utilisation courante pour déclencher une navigation vers une autre page :

```
import {useRouter} from 'next/router';

function MonComposant() {

const router = useRouter(); const handleNavigation = () => {
// Naviguer vers une autre page router.push('/autre-page');
}; return ( <div> <button onClick={handleNavigation}>Aller
vers une autre page</button> </div> ); } export default
MonComposant;
```

Ainsi, **useRouter** est un outil puissant pour gérer la navigation entre les pages de notre application Next.js, que ce soit en réagissant aux interactions de l'utilisateur ou en déclenchant des changements de page depuis notre code.

3.6 Créer un container global avec _app

Dans Next.js, pour éviter d'ajouter manuellement la barre de navigation (**Navbar**) à chaque page, nous pouvons utiliser le fichier **_app.js**. Ce composant est généré automatiquement par Next.js et sert de point d'entrée principal pour la création de pages de notre site.

Pour inclure la barre de navigation sur toutes les pages, nous pouvons utiliser un composant d'enveloppe, généralement appelé **Container**. Pour ce faire :

Nous créons un dossier et un fichier nommés "Container.js" dans le dossier de notre projet.

Dans ce fichier "Container.js", nous pouvons définir la structure du composant **Container**. Il peut s'agir d'un conteneur qui englobe le contenu principal de nos pages.

Ensuite, nous importons ce composant **Container** dans le fichier **_app.js**.

Pour appliquer le composant **Container** à toutes les pages, nous entourons la balise **<Component {...pageProps} />** avec les balises **Container** dans le fichier **_app.js**.

Cette approche nous permet de créer une mise en page cohérente pour toutes les pages de notre site en utilisant le composant **Container**. Ainsi, la barre de navigation ou d'autres éléments que nous souhaitons afficher sur toutes les pages peuvent être inclus de manière centralisée dans **_app.js**, simplifiant ainsi la gestion de la mise en page globale de notre application Next.js.

```
import '../styles/globals.css'
import Container from "../components/Container/Container";

1 usage  new *
function MyApp({Component, pageProps}) {
    return (
        <Container>
            <Component {...pageProps} />
        </Container>
    )

}

no usages  zotrim
export default MyApp
```

Figure 8 : Container

Pour inclure le contenu des pages dans notre mise en page globale à l'aide du composant **Container**, nous devons faire quelques ajustements dans notre fichier **Container.js**. Voici comment cela fonctionne :

1. Dans le fichier **Container.js**, nous devons ajouter une prop nommée **children**. Cette prop permettra d'inclure le contenu des pages à l'intérieur du composant **Container**.

2. Dans le fichier **Container.js**, nous utilisons **{props.children}** (ou simplement **children**) à l'endroit où nous souhaitons afficher le contenu spécifique de chaque page. Cela permet d'injecter le contenu des pages à l'intérieur du composant **Container**.

Ainsi, en utilisant la prop **children** dans notre composant **Container**, nous pouvons associer la mise en page globale, y compris la barre de navigation, avec le contenu unique de chaque page, créant ainsi une structure cohérente pour toutes les pages de notre application Next.js. Cela facilite également la gestion de l'apparence et de la structure globale de notre site.

Figure 9 : Props children

3.7 Gérer l'erreur 404

Pour créer une page d'erreur 404 dans Next.js, il suffit de créer un fichier nommé "404.js" dans le dossier "pages". Ce fichier peut être personnalisé à volonté pour afficher le contenu et le style appropriés, indiquant aux utilisateurs qu'ils ont atteint une page introuvable (erreur 404). Next.js gère automatiquement le routage vers cette page chaque fois qu'une URL non existante est accédée, simplifiant ainsi la gestion des erreurs de page non trouvée dans l'application.

3.8 Résumé du Routing avec Next

Le dossier "pages" est essentiel dans Next.js, car il nous permet de créer toutes les pages de notre application. Il suit des règles précises, notamment que "index.js" est la racine du projet et qu'une page d'erreur 404 peut être créée simplement en ajoutant un fichier "404.js".

Le fichier "_app.js" joue un rôle central en permettant la création de toutes les pages de l'application. Il est important de ne pas modifier les composants de base tels que "Component" et "pageProps", mais nous pouvons les entourer avec des éléments de mise en page, comme un conteneur.

Next.js prend en charge la création de pages avec un rendu dynamique, où le nom de la page n'est pas connu à l'avance. Pour cela, on utilise des fichiers "[slug].js" ou "[...slug].js", ce qui fonctionne également pour les dossiers.

Enfin, nous avons la possibilité de créer des composants React dans un dossier "components" situé au même niveau que le dossier "pages". Cela permet d'organiser de manière efficace nos composants réutilisables pour les intégrer dans différentes parties de notre application.

4. Optimiser avec Next

4.1 Utiliser du CSS

Dans le répertoire "Styles", nous rencontrons deux fichiers importants, à savoir "globals.css" et "Home.module.css". Le fichier "globals.css" est intégré dans le fichier "_app.js" afin que son contenu soit appliqué à l'ensemble des pages de l'application. Il s'agit donc du fichier de feuilles de style global qui influence l'apparence globale de l'application.

D'autre part, le fichier "Home.module.css" est importé spécifiquement au sein des pages individuelles. Pour l'utiliser, nous employons une syntaxe spécifique, à savoir "styles.container". Cette méthode permet d'appliquer des styles CSS de manière ciblée à des éléments spécifiques de la page "Home".

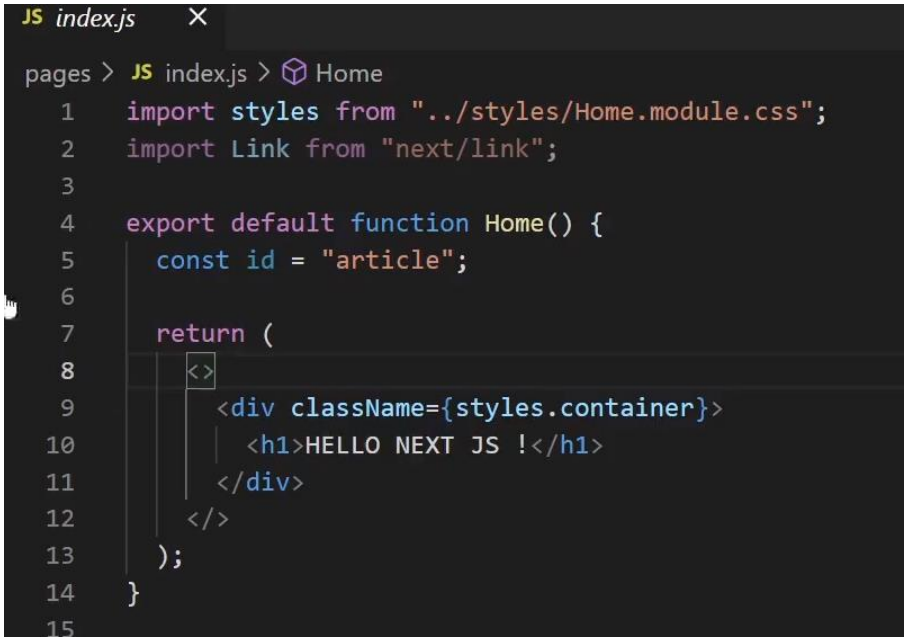

Figure 10 : styles.contaier

Pour créer un fichier CSS dans Next.js, nous le plaçons dans le dossier de la page correspondante en le nommant selon le format suivant : "nomFichier.module.css". Dans ce fichier CSS, la définition des styles commence généralement par la création d'une classe.

Pour appliquer ces styles CSS à un élément HTML dans le fichier JavaScript de la page, nous commençons par définir une classe dans le fichier JavaScript en utilisant la syntaxe : **className="nomClasse"**. Ensuite, pour utiliser la classe définie dans le fichier CSS, nous

l'importons dans le fichier JavaScript de la page et remplaçons **className="nomClasse"** par **className={styles.nomFichierCss}**.

Cette approche permet de lier de manière modulaire les styles CSS aux éléments spécifiques de chaque page, améliorant ainsi la gestion de la mise en page et de la présentation dans Next.js.

Figure 11 : importer fichier CSS

4.2 Utiliser un framework CSS

Dans Next.js, il est possible d'intégrer des frameworks CSS populaires tels que Bootstrap, Tailwind CSS, et bien d'autres. Pour illustrer ce processus, prenons l'exemple de l'installation de Bootstrap.

1. Pour intégrer Bootstrap au projet, ouvrons le terminal dans le répertoire du projet et exécutons la commande suivante : "npm install bootstrap".

2. Une fois l'installation terminée, la dépendance Bootstrap apparaît dans le fichier "package.json" du projet, indiquant que Bootstrap est désormais inclus.

3. Ensuite, pour utiliser Bootstrap dans l'ensemble de l'application Next.js, nous nous rendons dans le fichier "_app.js" où nous pouvons importer Bootstrap, le rendant ainsi disponible sur toutes les pages de l'application.

Il est à noter que d'autres frameworks tels que Tailwind CSS peuvent également être installés en suivant un processus similaire. L'intégration de tels frameworks CSS simplifie considérablement la création d'une interface utilisateur attrayante et réactive dans une application Next.js.

4.3 Optimiser le Head

Dans Next.js, nous avons la possibilité de personnaliser les balises **<head>** pour chaque page individuellement en fonction de nos besoins. Pour ce faire, nous importons le composant **Head** depuis "next/head".

À l'intérieur de la balise **<head>**, nous pouvons inclure des métadonnées (metadata) qui sont utiles pour divers aspects tels que l'encodage, le SEO (référencement), et la prise en charge du responsive design.

Voici quelques métadonnées couramment utilisées :

1. **<meta charset="UTF-8" />** : Cette balise indique l'encodage des caractères utilisé dans la page. Dans Next.js, il n'est généralement pas nécessaire de l'ajouter manuellement, car Next.js gère automatiquement l'encodage pour toutes les pages.

2. **<meta http-equiv="X-UA-Compatible" content="IE=edge" />** : Cette balise est utilisée pour spécifier le mode de compatibilité d'Internet Explorer. Elle est généralement utilisée pour assurer la compatibilité avec les anciennes versions d'IE, bien que son utilité puisse être limitée dans les projets modernes.

3. **<meta name="viewport" content="width=device-width, initial-scale=1.0" />** : Cette balise est cruciale pour le responsive design. Elle indique au navigateur de s'adapter à la largeur de l'appareil (comme les téléphones mobiles) et de définir l'échelle initiale à 1.0.

4. **<title>Document</title>** : Le titre de la page est essentiel pour le SEO, car il représente la description de la page dans les résultats de recherche.

Il est important de noter que ces balises **<head>** sont spécifiques à chaque page, ce qui signifie que nous devrons les ajouter pour chaque page individuellement. Si nous avez besoin d'un titre dynamique, nous pouvons utiliser des variables pour le personnaliser en fonction du contenu de la page, par exemple : **<title>{router.query.slug}</title>**.

En somme, la personnalisation des balises **<head>** permet de contrôler les métadonnées de chaque page, ce qui est essentiel pour l'optimisation du référencement et la gestion du responsive design.

4.4 Optimiser les images

Les images jouent un rôle essentiel en matière de référencement (SEO). Il est crucial de les gérer de manière efficace en utilisant des techniques telles que le "lazy loading" (chargement différé) et l'optimisation de leur taille pour garantir une génération rapide.

Pour intégrer des images dans un projet Next.js, nous pouvons utiliser des ressources telles qu'Unsplash pour obtenir des images libres de droits. Une fois les images téléchargées, elles doivent être organisées dans un dossier nommé "assets" dans le répertoire "public".

Lorsque nous inspectons la page en utilisant l'outil de développement (F12), nous pouvons accéder à l'onglet "Réseau" pour voir la liste des images. Il est important de noter les dimensions de chaque image, car cela permettra à Next.js de mieux les optimiser.

Pour optimiser les images, Next.js propose un composant spécifique appelé "Image". En utilisant ce composant, nous pouvons spécifier les dimensions de chaque image à l'aide des attributs "width" et "height". Cette information est cruciale pour permettre à Next.js d'optimiser automatiquement la taille et le chargement des images.

```
<Image src={img1}
       width="5064"
       height="3376"
       alt=""/>

    <Image src={img2}
           width="6000"
           height="4000"
           alt=""/>
        <Image src={img3}
               width="4725"
               height="3150"
               alt=""/>
```

Figure 12 : Le composant Image

L'un des avantages de l'utilisation du composant "Image" est la mise en œuvre automatique du "lazy loading". Cela signifie que les images ne sont chargées que lorsque l'utilisateur les regarde réellement, ce qui réduit le temps de chargement initial de la page et améliore les performances globales de l'application.

État	Type	Initiateur	Taille
200	jpeg	galery:0	1.1 MB
200	jpeg	galery:0	2.3 MB
200	jpeg	galery:0	1.1 MB
200	webp	galery:0	187 kB
200	webp	Autre	405 kB
200	webp	Autre	159 kB

Figure 13 : la taille des images

Il est possible d'utiliser d'autres attributs pour personnaliser davantage le comportement des images dans Next.js. Voici quelques-uns de ces attributs :

1. **layout="responsive"** : Cet attribut permet d'adapter les images à différents écrans, du mobile à l'ordinateur, en maintenant leur aspect ratio. Cela garantit une expérience utilisateur optimale sur divers appareils.

2. **layout="fill" objectFit="cover"** : Bien que disponible, cet attribut n'est pas fortement recommandé, car il peut déformer les images pour les faire tenir dans leur conteneur. Il est préférable de privilégier d'autres options pour éviter la distorsion des images.

3. **layout="fixed"** : Lorsque nous définissons cet attribut, l'image conserve une taille fixe et ne s'adapte pas dynamiquement à la taille du conteneur. Cela peut être utile lorsque nous souhaitons une image de taille constante.

4. **placeholder="blur"** : Cet attribut ajoute un effet de flou à l'image pendant son chargement. Cela améliore l'expérience utilisateur en fournissant un aperçu visuel de l'image en cours de chargement, plutôt que de laisser un espace vide.

Ces attributs offrent une flexibilité accrue pour gérer l'affichage des images dans notre application Next.js, en fonction de vos besoins spécifiques et de l'expérience utilisateur que nous souhaitons offrir.

```
<Image layout="responsive" src={img1}
       width="5064"
       height="3376"
       alt=""/>

    <Image layout="responsive" src={img2}
           width="6000"
           height="4000"
           alt=""/>
        <Image layout="responsive" src={img3}
               width="4725"
               height="3150"
               alt=""/>
```

Figure 14 : Layout responsive

4.5 Utiliser le _document.js

Dans le répertoire "pages", nous créons un fichier spécial nommé "_document.js". Cette page joue un rôle particulier en permettant des personnalisations au niveau du rendu HTML global de l'application Next.js.

Lors de la création de cette page, nous devons importer les éléments nécessaires, tels que "Html", "Head", "Main", et "NextScript", afin de personnaliser le rendu HTML global de notre application.

Ce fichier "_document.js" nous offre la possibilité d'apporter des ajustements au rendu global de notre application Next.js, comme la modification de la langue du HTML ou l'ajout de contenu en dehors de l'application web. Cela permet d'étendre les fonctionnalités de notre

application au-delà des composants individuels et de gérer des aspects spécifiques au niveau du document HTML global.

```
import Document, {Html, Head, Main, NextScript} from "next/document";

no usages  new *
class MyDocument extends Document {
    no usages  new *
    render() {
        return (
            <Html lang="de">
                <Head/>
                <body class="loader-on">
                    <Main/>
                    <NextScript/>
                <div className="modal"> Hello World</div>
                </body>
            </Html>
        )
    }
}

no usages  new *
export default myDocument;
```

Figure 15 : Exemple de la page _document.js

5. Gérer le rendu des pages et des données

5.1 Explication du rendu des pages

Lorsque nous examinons les pages en React, il peut arriver que nous ne puissions pas voir leur contenu dans l'inspecteur du navigateur. En revanche, avec Next.js, lorsque nous inspectons une page, nous pouvons visualiser l'intégralité du contenu, y compris le titre, les boutons, et autres éléments.

Cela est dû au fonctionnement interne de React, qui utilise un rendu côté client (client-side rendering) pour générer la page. Les moteurs de recherche, tels que Google, peuvent avoir des difficultés à indexer correctement les pages en React en raison de cette approche de rendu côté client.

En revanche, Next.js offre un rendu côté serveur (server-side rendering) et du "pré-rendu" statique, ce qui signifie que les pages sont générées côté serveur avant d'être envoyées au navigateur. Cette approche permet aux moteurs de recherche de mieux indexer le contenu des pages Next.js, améliorant ainsi leur visibilité dans les résultats de recherche.

En résumé, Next.js propose un meilleur support pour l'indexation par les moteurs de recherche grâce à son rendu côté serveur, ce qui peut être un avantage significatif en matière de référencement par rapport à React, qui utilise principalement un rendu côté client.

5.2 Lancer un “npm run build”

Une étape essentielle du processus de développement de l'application Next.js consiste à exécuter la commande "npm run build" une fois que le codage de l'application est terminé. Cette commande déclenche un processus de construction qui génère des pages statiques à partir de l'application.

Le résultat de cette opération est un ensemble de fichiers optimisés prêts à être déployés sur un serveur ou une plateforme d'hébergement. Ces pages statiques peuvent être servies rapidement aux utilisateurs, ce qui améliore considérablement les performances de l'application.

5.3 Utiliser la méthode « GetStaticProps »

Pour importer des données depuis une API ou une base de données et les utiliser dans les pages statiques de notre application Next.js, nous pouvons utiliser la fonction getStaticProps". Voici un exemple de son utilisation :

Figure 16 : getStaticProps

Il est essentiel de noter que nous devons toujours retourner un objet avec la clé "props" depuis "getStaticProps". Tout ce qui se trouve dans "getStaticProps" ne sera pas renvoyé au client, donc nous pouvons y inclure des clés API ou d'autres données sensibles en toute sécurité.

Pour utiliser les données récupérées via "props" dans nos pages, nous devons nous assurer d'ajouter "props" entre parenthèses lors de la définition de notre composant, par exemple :

```
<table className={styles.tableau}>
    <tbody>
    {props.array.map(el => (
        <tr>
            <td>{el.en}</td>
            <td>{el.fr}</td>
        </tr>
    ))
    }
    </tbody>
</table>
```

Figure 17 : Utiliser les données

Il est important de noter que lorsque nous utilisons des parenthèses pour entourer le contenu JSX, nous n'avons pas besoin du mot-clé "return". Cela nous permet d'inclure directement le contenu JSX dans la fonction, ce qui est une convention courante en React.

getStaticProps est une méthode puissante de Next.js qui nous permet de générer des pages statiques avec des données dynamiques, améliorant ainsi les performances de notre application et garantissant une expérience utilisateur optimale.

5.4 Faire de l'ISR (Incremental static generation)

Pour mettre en place l'ISR (Incremental Static Generation) dans Next.js, nous pouvons prendre l'exemple où les données peuvent changer, comme les prix sur un site e-commerce. Voici comment nous pouvons utiliser l'ISR :

Tout d'abord, nous créons une page statique normale. Ensuite, lors de la construction (build) de notre application, nous plaçons cette page statique sur un serveur. En utilisant la propriété "revalidate", nous pouvons définir un délai. Une fois que ce délai est écoulé, le code sera réexécuté côté serveur pour générer une nouvelle version de la page statique avec les données mises à jour.

Voici un exemple de comment nous pourrions utiliser "getStaticProps" avec ISR :

```
export default function contact(props) {
    console.log(props);
    return (
        <div>
            <h1>{props.data.quotes[0].text}</h1>
        </div>
    )
}
no usages
export async function getStaticProps() : Promise<{...}> {
    const quote : Response  = await fetch( input: "https://goquotes-api.herokuapp.com/api/v1/random?count=1")
    const data = await quote.json()
    return {
        props: {
            data
        },
        revalidate: 20
    }
}
```

Figure 18 : ISR

Dans cet exemple, nous récupérons des données depuis une API externe pour générer notre page statique. Nous utilisons la propriété "revalidate" pour spécifier le délai en secondes. Après 20 secondes, le code sera réexécuté côté serveur pour mettre à jour la page statique avec de nouvelles données.

L'ISR est une technique puissante qui nous permet de combiner les avantages des pages statiques avec la capacité de mettre à jour ces pages de manière dynamique lorsque cela est nécessaire, tout en maintenant des performances élevées pour nos utilisateurs.

5.5 Les autres propriétés utiles

Les propriétés liées à "getStaticProps" sont essentielles pour gérer les cas où une page statique ne peut pas être générée correctement. Parmi ces propriétés, nous avons "notFound" qui nous permet d'afficher une page 404 lorsque des erreurs se produisent. Voici un exemple d'utilisation :

```
no usages  new *
export async function getStaticProps() : Promise<...>  {
    const data = await import(`../data/vocabulary.json`);
    const array = data.vocabulary;

    if (array.length === 0) {
        return {
            notFound: true,
        }
    }
    return {
        props: {
            array
        }
    }
```

Figure 19 : Propriété notFound

Nous avons également la possibilité de rediriger une page en utilisant la propriété "redirect"

Figure 20 : propriété Redirect

Dans cet exemple, nous vérifions une certaine condition. Si cette condition est satisfaite, nous retournons un objet avec la propriété "redirect", où nous spécifions la destination de la redirection avec "destination". Nous pouvons également indiquer si la redirection est permanente en utilisant "permanent" (false par défaut).

Cela nous permet de rediriger les utilisateurs vers une autre page en fonction de certaines conditions, offrant ainsi une flexibilité pour gérer la navigation de l'application.

5.6 Utiliser la méthode « getStaticsPaths »

Lorsque nous travaillons avec des chemins dynamiques dans Next.js, nous utilisons généralement la méthode "getStaticPaths" pour informer Next.js du nombre de chemins dynamiques que nous souhaitons transformer en pages statiques. Nous récupérons les données nécessaires et nous retournons tous les chemins que nous souhaitons créer sous forme d'un objet. Cet objet a une propriété qui contient un tableau d'objets, chaque objet spécifiant à la fois le chemin dynamique et le nom de la page que nous voulons générer.

Une fois que nous avons configuré "getStaticPaths", nous devons également créer une méthode "getStaticProps" pour fournir les données correctes à notre page dynamique. Dans ce contexte, nous pouvons utiliser la propriété "context" pour accéder aux informations spécifiques à la page actuelle, ce qui nous permet de personnaliser le contenu en fonction du chemin dynamique ou de tout autre paramètre nécessaire.

```
no usages  new *
export async function getStaticProps(context) : Promise<{...}>  {
    const slug : string  = context.params.liste;
    const data = await import("/data/listes.json");

    const listeEnCours = data.default.englishList.find(list => list.name === slug);
    return {
        props: {
            listeEnCours: listeEnCours.data

        }
    }

}

no usages  new *
export async function getStaticPaths() : Promise<{...}>  {
    const data = await import("/data/listes.json");

    return {
        paths: [
            {params: {liste: "words"}},

        ],
        fallback: false,
    }
}
```

Figure 21 : getStaticProps et getStaticPaths

5.7 Finir UI

Dans le code (ci-dessous), nous avons une page appelée "liste" qui est une page dynamique. Elle affiche des listes de mots dans différentes langues en fonction du chemin dynamique. Par exemple, si le chemin est "/liste/words", elle affichera une liste de mots en anglais.

Pour rendre cette page dynamique, nous utilisons deux fonctions spéciales de Next.js : "getStaticProps" et "getStaticPaths".

getStaticProps: Cette fonction est utilisée pour obtenir les données nécessaires pour la page. Dans ce cas, elle prend le paramètre "context" qui contient les informations sur le chemin dynamique. Nous extrayons le nom de la liste à partir de ce chemin (par exemple, "words" à partir de "/liste/words"). Ensuite, nous importons les données depuis un fichier JSON qui contient des listes de mots. Nous cherchons la liste correspondante dans ces données en fonction du nom extrait du chemin, puis nous renvoyons cette liste sous la forme de "listeEnCours" dans les "props".

getStaticPaths: Cette fonction est utilisée pour spécifier quels chemins dynamiques doivent être rendus en tant que pages statiques. Dans ce cas, nous importons également les données

de notre fichier JSON, puis nous créons un tableau de paramètres pour chaque liste disponible. Chaque paramètre contient un objet avec une propriété "params" qui a elle-même une propriété "liste" qui correspond au nom de la liste. Par exemple, pour une liste nommée "words", nous aurons {params: {liste: "words"}}. Ce tableau de paramètres est renvoyé à Next.js.

Lorsque Next.js génère le site, il sait qu'il doit créer des pages statiques pour chaque chemin dynamique spécifié dans le tableau de paramètres. Ainsi, si vous avez plusieurs listes de mots, chaque liste aura sa propre page statique. Lorsqu'un utilisateur visite l'un de ces chemins dynamiques, Next.js peut alors récupérer les données appropriées grâce à "getStaticProps" et les afficher sur la page.

```
export async function getStaticProps(context) : Promise<{…}> {
    const slug = context.params.liste;
    const data = await import("/data/listes.json");

    const listeEnCours = data.default.englishList.find(list => list.name === slug);
    return {
        props: {
            listeEnCours: listeEnCours.data

        }
    }

}
no usages  new *
export async function getStaticPaths() : Promise<{…}> {
    const data = await import("/data/listes.json");

    const paths = data.englishList.map(item => ({
        params: {liste: item.name}
    }));

    return {
     //  paths: [
     //      {params: {liste: "words"}},

     //  ],
        paths,
        fallback: false,
    }
}
```

Figure 22 : getSaticsPaths (plusieurs chemins)

5.8 la propriété « fallback »

Dans Next.js, la propriété "fallback" est utilisée dans la fonction **getStaticPaths** pour déterminer comment nous gérons les pages qui correspondent à des chemins dynamiques qui ne sont pas générés lors de la construction initiale du site.

Voici comment cela fonctionne :

fallback: false : Si nous définissons "fallback" sur **false**, cela signifie que seuls les chemins dynamiques spécifiés dans le tableau de paramètres de **getStaticPaths** seront générés en tant que pages statiques lors de la construction initiale du site. Si un utilisateur visite un chemin dynamique qui n'est pas dans ce tableau, il obtiendra une page 404 car la page n'existe pas.

fallback: true : Si nous définissons "fallback" sur **true**, cela signifie que Next.js générera automatiquement des pages statiques pour les chemins dynamiques qui ne sont pas spécifiés dans le tableau de paramètres de **getStaticPaths**. Ces pages seront générées à la volée lorsqu'un utilisateur les visite pour la première fois. Cela signifie que la construction initiale du site génère uniquement les pages spécifiées dans le tableau de paramètres, tandis que les autres sont générées dynamiquement au besoin. Cela peut être utile si nous avons un grand nombre de chemins dynamiques et que nous ne voulons pas les générer tous à l'avance.

fallback: "blocking" : Cette option est similaire à **fallback: true**, mais elle fonctionne de manière synchrone. Cela signifie que lorsque l'utilisateur visite un chemin dynamique qui n'est pas encore généré, il devra attendre que la page soit générée avant de la voir. Cette approche est souvent utilisée lorsque la génération des pages est rapide et que nous souhaitons garantir que chaque page est générée avant d'être servie.

5.9 Le rendu côté serveur

Lorsque nous parlons du rendu côté serveur avec Next.js, nous faisons référence à la manière dont Next.js génère le contenu d'une page sur le serveur au lieu de le générer côté client (comme dans une application React traditionnelle). Voici comment cela fonctionne :

1. **Chargement initial côté serveur** : Lorsqu'un utilisateur visite une page de notre site Next.js, la première demande est traitée par le serveur. Le serveur exécute le code de la page et génère le HTML correspondant à cette page. Cela signifie que le contenu de la page est créé sur le serveur lui-même.

2. **Envoi de la page au client** : Une fois que le serveur a généré la page, il l'envoie au client sous forme de HTML prêt à être affiché. Le client reçoit ensuite cette page HTML et peut l'afficher immédiatement, ce qui améliore considérablement le temps de chargement initial de la page.

3. **Réactivité côté client** : Une fois que la page est affichée côté client, elle se comporte comme une application React classique. Cela signifie que les interactions et les mises à jour d'état se produisent côté client sans avoir besoin de recharger la page entière. Cela améliore l'expérience utilisateur en rendant l'application réactive.

L'utilisation du rendu côté serveur avec Next.js présente plusieurs avantages :

- **Amélioration des performances** : Le chargement initial côté serveur réduit considérablement le temps nécessaire pour afficher le contenu initial de la page, ce qui est essentiel pour l'expérience utilisateur.
- **SEO amélioré** : Les moteurs de recherche, tels que Google, apprécient le contenu généré côté serveur, car il est directement visible dans le HTML de la page. Cela peut améliorer le classement de votre site dans les résultats de recherche.
- **Réactivité** : Une fois que la page est chargée, elle se comporte comme une application React, offrant une expérience utilisateur fluide et interactive.

Cependant, le rendu côté serveur peut être plus intensif en termes de ressources serveur, car il nécessite une exécution du code sur le serveur pour chaque demande de page. Par conséquent, il est essentiel de surveiller les performances du serveur, en particulier pour les sites à fort trafic.

```jsx
import React from "react";

1 usage  new *
export default function cours({ results }) {
    return (
        <div>
            <h1 className="text-center">Le cours du Bitcoin est à {results.bpi.EUR.rate} EUR</h1>
        </div>
    );
}

no usages  new *
export async function getServerSideProps(context) : Promise<{…}>  {
    console.log("context", context)

    const response : Response  = await fetch( input: "https://api.coindesk.com/v1/bpi/currentprice.json");
    const data = await response.json();

    return {
        props: {
            results: data
        }
    }
}
```

Figure 23 : getServerSideProps

6. Créer une API avec Next

6.1 Créer une API de base

Pour créer une API de base dans Next.js, nous suivons ces étapes simples :

1. **Création de l'API** : Tout d'abord, nous créons un dossier nommé "api" dans le répertoire "pages". C'est là que nous stockerons nos fichiers d'API.

2. **Fichier API** : À l'intérieur du dossier "api", nous créons un fichier JavaScript pour notre API. Par exemple, nous pouvons nommer ce fichier "exemple.js".

3. **Définition de l'API** : Dans le fichier "exemple.js" (ou tout autre nom que vous avez choisi), nous définissons notre API. Cela peut inclure la gestion de différentes requêtes HTTP, telles que GET, POST, PUT ou DELETE, en fonction de nos besoins.

4. **Utilisation de l'API** : Nous utilisons ensuite cette API dans notre application front-end. En faisant des requêtes vers l'URL de l'API depuis notre application, nous pouvons obtenir des données en réponse.

```
import fs from 'fs';
import path from 'path';

no usages  new *
export default function handler(req, res) : void {
    if (req.method === 'GET') {

        res.status(200).json({data:5});
    }
}
```

Figure 24 : Création API

6.2 Afficher un mot au hasard

Ce chapitre a été consacré à la création d'une fonctionnalité permettant d'afficher aléatoirement un mot à partir d'une liste de mots stockée dans notre base de données. Cette fonctionnalité permet à l'utilisateur de générer un mot aléatoire en appuyant sur un bouton dédié. Le mot est sélectionné de manière aléatoire à partir de la liste de mots préalablement chargée depuis notre base de données. Nous avons également pris en compte des scénarios où la liste de mots est vide, en redirigeant l'utilisateur ou en affichant un message approprié.

Cette fonctionnalité peut être utilisée dans divers contextes éducatifs ou de divertissement, notamment pour des applications d'apprentissage de langues, de jeux de quiz, ou simplement pour pimenter l'expérience utilisateur sur un site web.

```
const newWord = () : void  => {
    fetch( input: '/api/vocapi') Promise<Response>
        .then(response : Response  => response.json()) Promise<any>
        .then(data => setState(data))
}
console.log(state)

let randomWord;
if (state) {
    const array = state.englishList[0].data;
    randomWord = array[Math.floor( x: Math.random() * array.length)].en;
}
```

Figure 25 : Fonctionnalité aléatoire

6.3 Créer un formulaire

Nous souhaitons ajouter des mots à notre base de données. Pour ce faire, nous utilisons les "uncontrolled inputs" pour contrôler notre formulaire. Lorsqu'un utilisateur saisit un mot en anglais et en français, nous les récupérons et les stockons dans un objet. Cette fonctionnalité nous permet d'ajouter des mots à notre base de données ou de réaliser d'autres opérations nécessaires. Nous pouvons personnaliser cette fonction en fonction de nos besoins spécifiques.

```
const add = () => {
    const enWord : MutableRefObject<undefined>  = useRef();
    const frWord : MutableRefObject<undefined>  = useRef();

    1 usage  new *
    const handleSubmit = (e) : void  => {
        e.preventDefault()

        const newWord : {en: any, fr: any}  = {
            en: enWord.current.value,
            fr: frWord.current.value
        }
    }
    return (
        <div className="container p-4">
            <form onSubmit={handleSubmit}>
                <label htmlFor="addEn" className="form-label">
                    Ajouter un mot en anglais
                    <input ref={enWord} type="text" className="form-control" id="addEn"/>
                </label>

                <label htmlFor="addFr" className="form-label">
                    Ajouter un mot en français
                    <input ref={frWord} type="text" className="form-control" id="addFr"/>
                </label>
                <button className="btn btn-primary mt-4">Ajouter</button>
            </form>
        </div>
    );
```

Figure 26 : Créer un formulaire

6.4 Utiliser une requête « POST » pour changer les données

Dans cette section, nous avons mis en place la gestion de l'ajout de nouveaux mots à notre base de données à l'aide d'une requête API HTTP POST. Pour ce faire, nous utilisons la méthode **fetch** côté client pour envoyer les données du nouveau mot au serveur.

Lorsque le serveur reçoit une requête POST, il extrait les données du mot en anglais (enWord) et du mot en français (frWord) provenant du corps de la requête. Ensuite, un nouvel objet "newWord" est créé avec ces données.

Le serveur accède ensuite au fichier JSON de notre base de données, ajoute le nouveau mot à la liste existante, puis réécrit le fichier avec les données mises à jour. Une réponse JSON est renvoyée au client pour indiquer que l'opération a réussi.

Du côté client, nous avons inclus cette logique dans un gestionnaire d'événements qui s'exécute lorsque l'utilisateur soumet le formulaire d'ajout. Nous utilisons la méthode **fetch** pour envoyer les données au serveur, en veillant à spécifier le type de contenu en tant qu'application JSON.

Une fois que le serveur a confirmé l'ajout avec succès, nous réinitialisons les champs du formulaire pour permettre à l'utilisateur d'ajouter plus de mots si nécessaire.

Cette fonctionnalité d'ajout de mots à notre base de données est essentielle pour permettre aux utilisateurs de contribuer et de développer notre collection de mots de manière collaborative.

7. Déployer son app sur Vercel

Dans ce dernier chapitre, nous avons exploré le processus de déploiement d'une application Next.js sur la plateforme Vercel. Voici un résumé des étapes clés :

1. Création d'un dépôt GitHub : Tout d'abord, nous avons créé un dépôt GitHub pour notre projet. C'est l'endroit où nous stockons notre code source et le partageons avec d'autres collaborateurs.
2. Préparation de l'application : Avant de déployer, nous avons exécuté la commande "npm run build" pour construire notre application. Cette étape a généré des pages statiques prêtes à être déployées.
3. Initialisation de Git : Nous avons utilisé Git pour suivre les modifications de notre code. Pour ce faire, nous avons exécuté les commandes "git init", "git add .", et "git commit -m 'first commit'" pour créer notre dépôt local.
4. Renommage de la branche principale : Par souci d'inclusivité, nous avons renommé la branche principale de notre dépôt de "master" à "main" en utilisant la commande "git branch -m main".
5. Liaison du dépôt GitHub : Nous avons lié notre dépôt local à notre dépôt GitHub en utilisant la commande "git remote add origin [URL du dépôt GitHub]" pour établir une connexion.
6. Poussée du code vers GitHub : Nous avons ensuite poussé notre code vers le dépôt GitHub en utilisant la commande "git push -u origin main". Cela a transféré notre code sur GitHub pour le stockage et la gestion.
7. Connexion à Vercel : Nous nous sommes connectés à Vercel, une plateforme de déploiement, en créant un nouveau projet. Nous avons associé notre dépôt GitHub à notre projet Vercel.
8. Déploiement : Une fois notre projet lié, nous avons initié le déploiement en appuyant sur le bouton "Deploy" sur Vercel. La plateforme a automatiquement détecté notre configuration Next.js et déployé notre application.

En fin de compte, cela nous a permis d'obtenir un lien public vers notre application déployée sur Vercel, que nous pouvons partager avec d'autres utilisateurs. Ce processus de déploiement facilite la mise en ligne de nos projets Next.js de manière transparente et efficace.

Lien du projet : https://test-deploy-next-fawn.vercel.app/

8. Références

1. TUTO Next.js : Cours Complet sur Tuto.com. Aperçu sur Tuto.com. https://fr.tuto.com/javascript/next-js-cours-complet,167441.html le 26 septembre 2023