

NextJS



TABLE DES MATIÈRES

01

**Introduction et
Configuration**

02

TypeScript

03

Pages et Router

04

Server Side Rendering (SSR)

TABLE DES MATIÈRES

05

**Static Site Generation
(SSG)**

07

**Server Actions et Route
Handlers**

06

Optimisation et SEO

08

**Intégration d'un ORM
(Prisma)**

TABLE DES MATIÈRES

09

Sécurité (Next-Auth)

10

Sécurité (Middlewares)

11

**Environnement Vercel
(Déploiement)**

12

**Environnement Vercel
(Vercel Functions)**

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 utilise 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

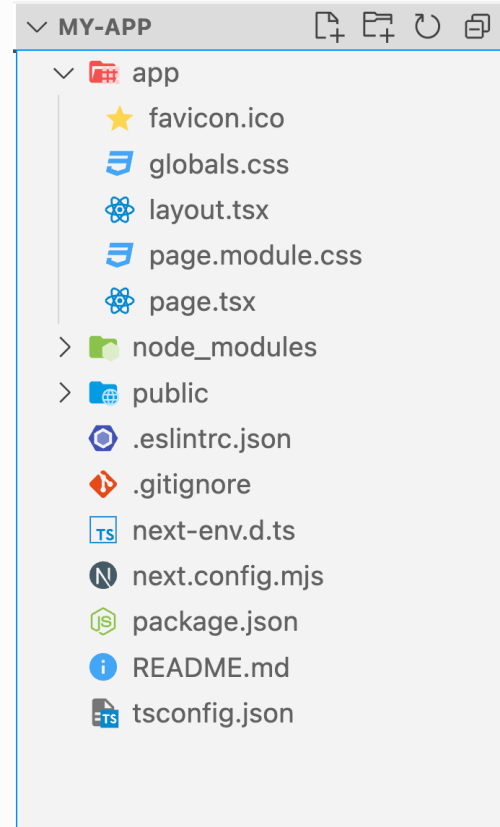
Avant de s'intéresser à la structure du projet, il faut bien faire la distinction entre les deux environnements sur lequel notre code peut tourner.

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 composants, 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.

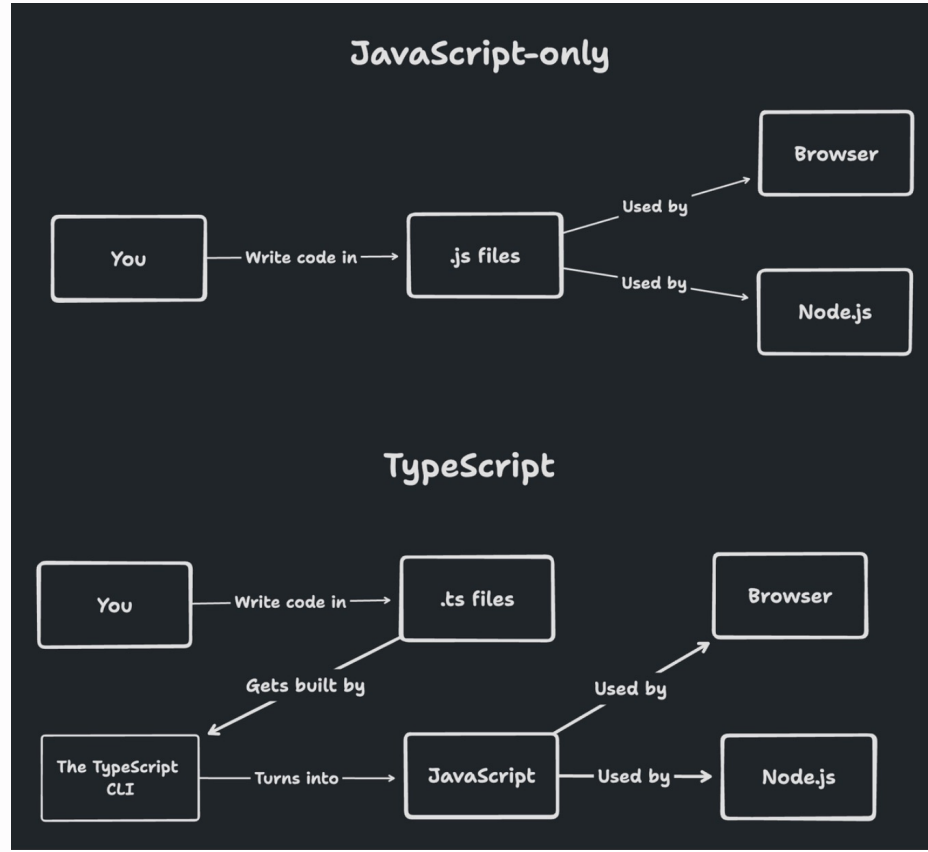
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

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 génèreraient des erreurs à l'exécution.

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

Type et Interface

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

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

Type ou Interface ?

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

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

```
interface item {  
  name: string;  
  price: number;  
}  
  
interface weapon extends item {  
  damage: number;  
}
```

Union types et Narrowing

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

```
type StringOrNumber = string | number;
```

```
let myAge: StringOrNumber = 25;  
myAge = '25'; // OK
```

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

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


Narrowing sur des objets ?

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

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

Les génériques

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

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

Les génériques

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

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

Les génériques

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

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

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.

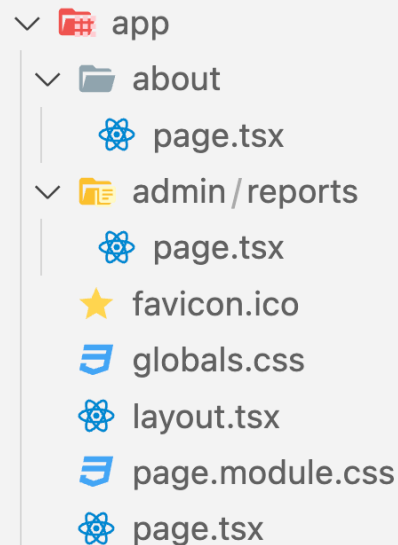
Par exemple, un fichier situé à `/app/about/page.js` générera une route sur `/about`. Un fichier localisé dans `/app/admin/reports/page.js` générera une route `/admin/reports`.

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 nécessairement s'exécuter 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

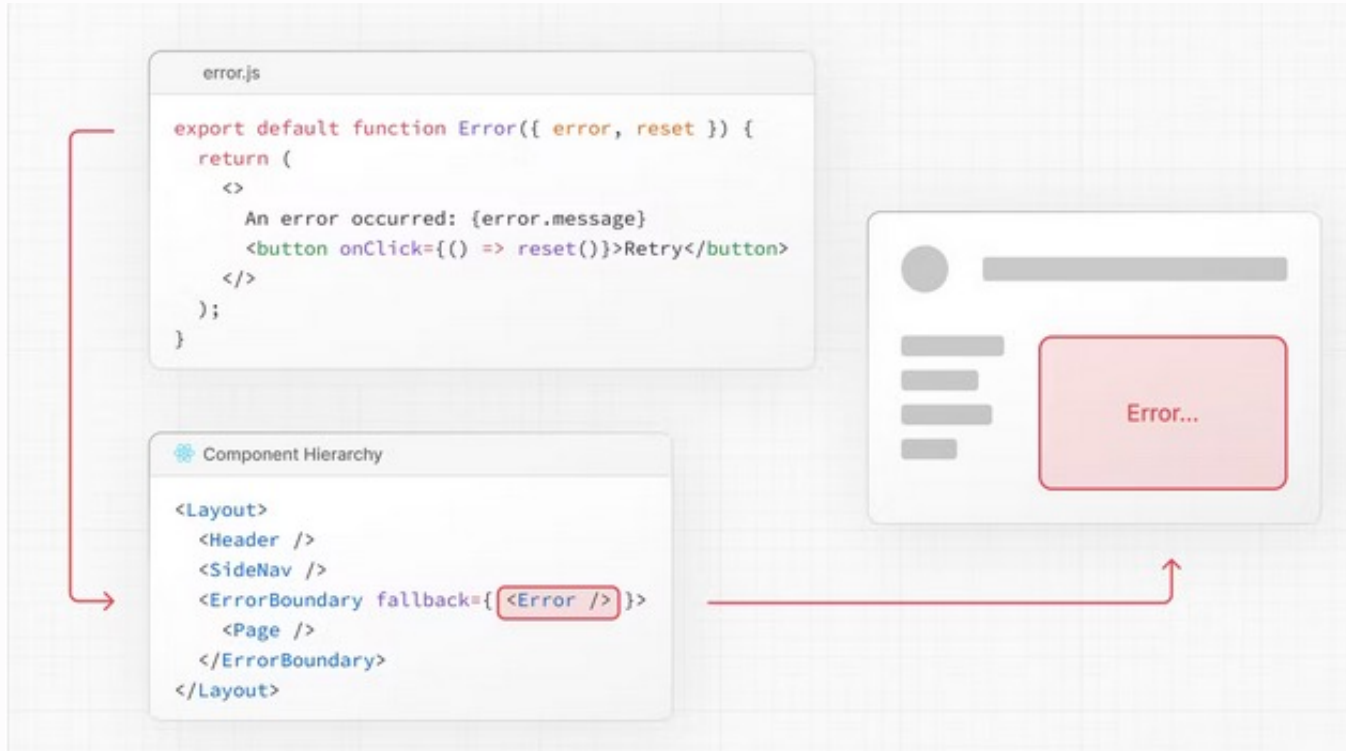
```
'use client' // Error components must be Client Components

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



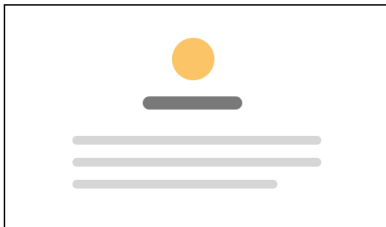
Principe du prerendering

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.

Pre-rendering (Using Next.js)

Initial Load:

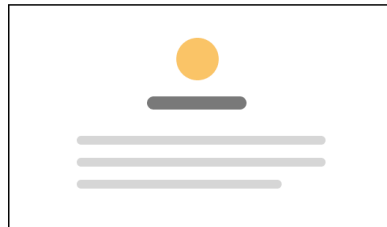
Pre-rendered HTML is displayed



JS loads
→

Hydration:

React components are initialized and App becomes interactive



If your app has interactive components like `<Link />`, they'll be active after JS loads

Static Generation et Server Side Rendering

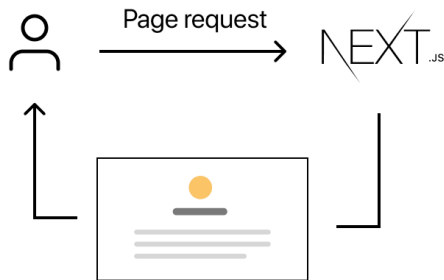
Next propose deux systèmes pour le pre-rendering :

- **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

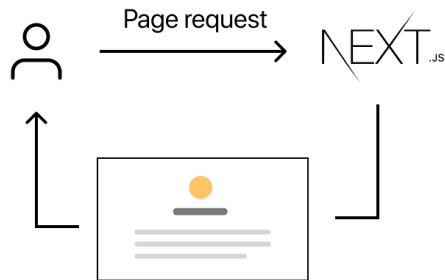
Server Side Rendering

Server-side Rendering

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



The HTML is generated



The HTML is generated

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

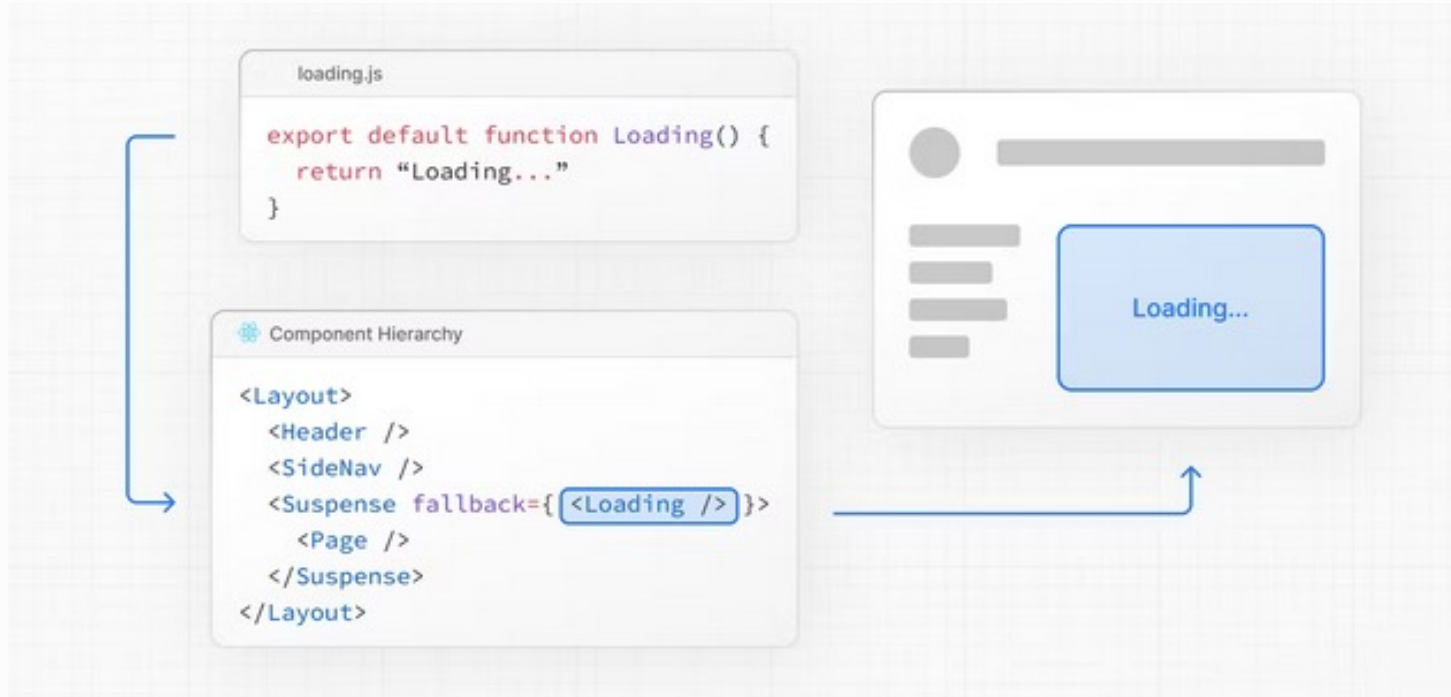
Lorsque nos pages demandent un temps de chargement, nous pouvons afficher une interface d'attente.

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

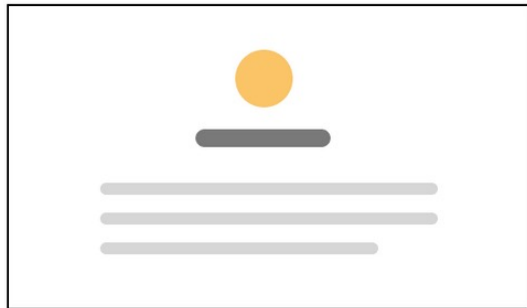
Static Generation

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

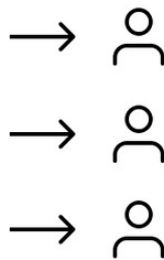
NEXT
_JS

next build →

Builds the app for
production



The HTML is generated



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");
}
```

Mutation de la donnée avec Zod

Lorsque l'on utilise les Server Actions avec les formulaires, l'action reçoit en paramètre un objet de type FormData.

Il nous faut vérifier la donnée pour des raisons de sécurité (le code client a pu être modifié par l'utilisateur), et pour éviter des erreurs de types.

Par exemple, pour un input « amount » de type number, la valeur inscrite dans notre objet FormData sera... une string !

Il nous faut donc vérifier et muter la donnée pour que la donnée récupérée par le formulaire corresponde bien à nos types et schéma de base de données.

On peut utiliser la librairie Zod pour nous faciliter la tâche :

npm install zod

Mutation de la donnée avec Zod

Voici un type Invoice, qui correspond également à une table dans notre base de données:

```
export type Invoice = {  
  id: string; // Will be created on the database  
  customer_id: string;  
  amount: number; // Stored in cents  
  status: "pending" | "paid";  
  date: string;  
};
```

Mutation de la donnée avec Zod

Lors de l'envoi de notre formulaire, amount sera reçu sous forme de string, et nous n'aurons pas d'id, puisqu'il sera créé par la base de données. La date de création de la facture sera également générée côté serveur.

Nous allons pouvoir créer un objet via Zod qui nous permettra de typer correctement les données depuis notre type Invoice, et qui nous permettra de nous assurer un bon format de données pour l'insertion dans notre base de donnée.

Mutation de la donnée avec Zod

```
import { z } from 'zod';

const FormSchema = z.object({
  id: z.string(),
  customerId: z.string(),
  amount: z.coerce.number(),
  status: z.enum(['pending', 'paid']),
  date: z.string(),
});

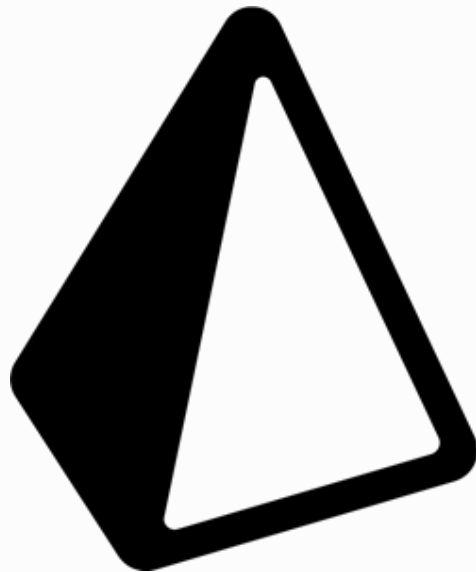
const CreateInvoice = FormSchema.omit({ id: true, date: true });
```

Mutation de la donnée avec Zod

```
export async function createInvoice(formData: FormData) {  
  const { customerId, amount, status } = CreateInvoice.parse(Object.fromEntries(formData));  
  const amountInCents = amount * 100;  
  const date = new Date().toISOString().split('T')[0];  
  
  await sql`  
    INSERT INTO invoices (customer_id, amount, status, date)  
    VALUES (${customerId}, ${amountInCents}, ${status}, ${date})  
  `;  
  revalidatePath('/invoices');  
  redirect('/invoices');  
}
```

08

Intégration d'un **ORM**



ORM ?

Un ORM, ou **Object Relational Mapper** (mappeur objet-relationnel en français), est un logiciel qui sert d'intermédiaire entre votre programme orienté objet et une base de données relationnelle.

Il se charge de traduire vos instructions en requêtes SQL compréhensibles par la base de données et de convertir les résultats de ces requêtes en objets utilisables par votre programme.

Présentation de Prisma

Prisma est un ORM JavaScript parfaitement adapté pour une utilisation avec NextJS. Il est supporté par défaut avec l'hosting Vercel d'une base de données (les variables d'environnement sont fournies directement par Vercel), nous permettra de définir le schéma de la base de données directement dans notre code, et s'assurera que les types soient bien respectés.

```
npm install prisma --save-dev  
npm install @prisma/client
```

Définition du schéma

Pour définir notre schéma de base de données, nous allons créer un dossier prisma, dans lequel nous créerons un fichier prisma.schema (schéma sur slide suivante).

Une fois le schéma créé, nous pouvons lancer la migration de notre schéma avec la commande : ***npx prisma db push***

Prisma propose aussi l'utilisation d'un GUI pour interagir avec la base de données : ***npx prisma studio***

Définition du schéma

Documentation des schema: <https://www.prisma.io/docs/orm/prisma-schema>

```
generator client {  
  provider = "prisma-client-js"  
}  
  
datasource db {  
  provider = "postgresql"  
  url = env("POSTGRES_PRISMA_URL") // uses connection pooling  
  directUrl = env("POSTGRES_URL_NON_POOLING") // uses a direct connection  
}  
  
model Post {  
  id      String      @default(cuid()) @id  
  title   String  
  content String?  
  published Boolean @default(false)  
  author  User?       @relation(fields: [authorId], references: [id])  
  authorId String?  
}  
  
model User {  
  id      String      @default(cuid()) @id  
  name    String?  
  email   String?     @unique  
  createdAt DateTime @default(now()) @map(name: "created_at")  
  updatedAt DateTime @updatedAt @map(name: "updated_at")  
  posts   Post[]  
  @@map(name: "users")  
}
```

Mise en place du client

Notre schéma étant intimement lié à notre base de données, il faut les resynchroniser à chaque modification du schéma avec la commande: ***npx prisma generate***

Maintenant que notre installation est faite, nous allons pouvoir créer une instance de client prisma, que nous pourrons utiliser dans toute notre application pour interagir avec la base de données.

Mise en place du client

```
import { PrismaClient } from '@prisma/client';

let prisma: PrismaClient;

if (process.env.NODE_ENV === 'production') {
  prisma = new PrismaClient();
} else {
  if (!global.prisma) {
    global.prisma = new PrismaClient();
  }
  prisma = global.prisma;
}

export default prisma;
```

Récupération de la donnée

```
'use server';
import prisma from '../prisma'
export async function getPosts() {
  const feed = await prisma.post.findMany({
    where: { published: true },
    include: {
      author: {
        select: { name: true },
      },
    },
  });
  return feed;
}
```

Insertion de donnée

```
export async function savePost(formData: FormData) {
  try {
    //Vérification des données avec Zod
    const result = await prisma.post.create({
      data: {
        title: title,
        content: content,
        author: { connect: { email: email } },
      },
    });
  } catch (error) {
    // gestion de l'erreur
  }
  revalidatePath("/posts");
  redirect("/posts");
}
```

Documentation de prisma client

Vous retrouverez toutes les méthodes disponibles pour les interactions avec la base de données via le client prisma (update, connexion des relations, suppression, pagination...) dans la documentation:

<https://www.prisma.io/docs/orm/prisma-client/queries>

09

Next-Auth



Next Auth

NextAuth.js est une solution d'authentification open-source conçue spécifiquement pour les applications Next.js.

Elle nous permet d'ajouter facilement et de manière sécurisée de nombreux providers d'authentification.

Dans cette formation, nous allons ajouter le provider GitHub.

```
npm install next-auth
```

Next Auth

Tout d'abord, nous allons créer une application github:

<https://docs.github.com/en/apps/creating-github-apps/registering-a-github-app/registering-a-github-app>, afin de recevoir un clientId et un secret.

Nous avons également besoin d'un secret pour next-auth afin de vérifier nos JWT grâce à la commande suivante:

```
openssl rand -base64 32
```

Next Auth

Ensuite, créez un fichier auth.ts dans le dossier lib. C'est ici que nous gérerons nos options d'authentification.

```
import { NextAuthOptions } from 'next-auth';
import GithubProvider from 'next-auth/providers/github';
export const authOptions: NextAuthOptions = {
  // Secret for Next-auth, without this JWT encryption/decryption won't work
  secret: process.env.NEXTAUTH_SECRET,

  // Configure one or more authentication providers
  providers: [
    GithubProvider({
      clientId: process.env.GITHUB_APP_CLIENT_ID as string,
      clientSecret: process.env.GITHUB_APP_CLIENT_SECRET as string,
    }),
  ],
};
```


Next Auth

Nous allons ensuite créer un Route Handler pour gérer nos routes d'authentification dans `/app/api/auth/[...nextauth]`.

```
import NextAuth from 'next-auth';  
import { authOptions } from '@app/lib/auth';  
  
const handler = NextAuth(authOptions);  
export { handler as GET, handler as POST };
```

Next Auth

Pour nous connecter, il nous suffit désormais d'aller sur `/api/auth/signIn` pour accéder à la connexion avec GitHub.

Une fois la connexion faite, nous recevrons des cookies utilisés pour l'auth.

La route de déconnexion est `/api/auth/signOut`.

Récupérer la session dans les composants serveurs

Pour récupérer les informations dans les composants serveurs, on peut utiliser la méthode `getSession()`, qui prend en paramètre notre configuration Next Auth. Cette méthode est aussi disponible pour les `RouteHandler` et les `Server Actions`.

```
const Page = () => {  
  const session = await getSession(authOptions);  
  if (!session) {  
    redirect("/login");  
  }  
  return <div>...</div>;  
};  
  
export default Page;
```

Récupérer la session dans les composants client

Pour récupérer les informations de la session, nous avons besoin d'ajouter un provider à notre RootLayout.

```
'use client';
import { SessionProvider } from 'next-auth/react';
import { ReactNode } from 'react';

export default function NextAuthProvider({
  children,
}: {
  children: ReactNode;
}) {
  return <SessionProvider>{children}</SessionProvider>;
}
```

Récupérer la session dans les composants client

Ensuite, nous pouvons utiliser le hook `useSession()` pour récupérer les informations.

```
'use client';
import { useSession } from 'next-auth/react';

export default function ClientComponent() {
  const { data: session, status } = useSession();
  return (
    <div>
      ClientComponent {status}{' '}
      {status === 'authenticated' && session.user?.name}
    </div>
  );
}
```