Chapitre 3
Routage et navigation avancée

1. Comprendre l'évolution du routage dans Next.js

1.1 Pourquoi deux systèmes de routage ?

Next.js, depuis ses débuts, s'est distingué par son système de routage basé sur la structure des fichiers. Jusqu'à la version 12, ce mécanisme reposait exclusivement sur le *Pages Router*, une solution simple et efficace : chaque fichier dans le dossier `pages/` correspondait directement à une route de l'application. Cette convention a grandement contribué à la popularité de Next.js en réduisant la configuration nécessaire pour créer une SPA (*Single Page Application*) optimisée et performante.

Cependant, avec l'évolution de React – notamment l'arrivée des server components, du streaming via `Suspense`, et des layouts imbriqués – les limites du Pages Router sont devenues plus apparentes. Next.js 13 a donc introduit un nouveau paradigme : l'App Router. Plutôt que de venir remplacer brutalement l'ancien système, ce nouveau routeur a d'abord été proposé en parallèle, laissant aux développeurs la possibilité d'adopter progressivement cette nouvelle architecture.

Remarque

Le Pages Router n'est pas supprimé : il continue de fonctionner pour des raisons de rétrocompatibilité. Mais il n'évoluera plus et ne prend pas en charge les fonctionnalités modernes de React.

Ce changement structurel n'est pas anodin : il implique une réorganisation du code, une compréhension plus poussée des comportements côté serveur et client, et l'adoption de nouvelles conventions. Mais il reflète aussi une ambition claire : permettre à Next.js de devenir la plateforme de référence pour les applications React modernes, capable de tirer pleinement parti des innovations introduites dans le cœur même de React.

En résumé, l'App Router est une réponse directe aux évolutions du Web moderne. Il ne s'agit pas simplement d'un changement d'API, mais d'un changement de paradigme, qui rapproche encore davantage Next.js d'un framework full-stack, performant et évolutif. Le Pages Router, quant à lui, reste utilisable, mais s'adresse surtout aux projets hérités ou à ceux qui ne nécessitent pas les fonctionnalités les plus avancées.

1.2 Le Pages Router : héritage et limites

Le Pages Router constitue l'architecture historique du routage dans Next.js. Jusqu'à la version 12 incluse, il représentait le système par défaut pour structurer une application. Son fonctionnement repose sur un principe simple : chaque fichier `.tsx` présent dans le dossier `pages/` correspond à une route.

Par exemple, `pages/index.tsx` affiche la page d'accueil, `pages/about.tsx` génère la route `/about`, et `pages/blog/[slug].tsx` permet de gérer les routes dynamiques via des paramètres. Cette convention de routage par fichier a longtemps facilité la prise en main du framework.

En ce qui concerne la récupération de données, le Pages Router repose sur un trio de fonctions spécifiques : `getStaticProps`, `getServerSideProps` et `getStaticPaths`. Ces fonctions sont exécutées à la compilation (*static generation*) ou à chaque requête (*server-side rendering*), selon les besoins du développeur.

Cependant, malgré sa simplicité, le Pages Router présente aujourd'hui plusieurs limitations majeures :

– Il ne permet pas la persistance des layouts entre les pages. Chaque navigation entraîne un rechargement complet du layout.
– Il ne prend pas en charge les server components, introduits avec React 18.
– Il ne bénéficie pas des optimisations modernes liées au streaming ou à la gestion avancée de l'interface.

Ces contraintes ont conduit l'équipe de Next.js à concevoir une nouvelle approche, plus flexible et tournée vers l'avenir : l'App Router.

Le Pages Router reste maintenu pour assurer la compatibilité avec les projets existants, mais il n'est plus recommandé pour les nouveaux développements. Il est prévu que son rôle se limite à des cas de migration ou à des applications simples ne nécessitant pas les fonctionnalités avancées introduites à partir de Next.js 13.

Remarque

Si vous débutez un nouveau projet, utilisez toujours le dossier app/ et l'App Router. Le Pages Router reste supporté mais n'évoluera plus.

1.3 L'App Router : l'architecture moderne

L'App Router représente la nouvelle approche du routage introduite dans Next.js à partir de la version 13. Conçu pour tirer pleinement parti des avancées récentes de React, ce système repose sur une structure plus modulaire, orientée composant et pensée pour les applications complexes, évolutives et performantes.

Le principe fondamental de l'App Router est simple : chaque segment d'URL correspond à un sous-dossier dans le répertoire `app/`. À l'intérieur de chaque dossier, des fichiers spécifiques définissent le comportement de ce segment.

Les fichiers les plus courants sont :

– `page.tsx` : composant principal affiché pour une route donnée ;
– `layout.tsx` : layout persistant entre les pages d'un même segment ou de segments enfants ;
– `loading.tsx` : composant affiché pendant le chargement asynchrone d'une page ou de ses données ;
– `error.tsx` : gestion des erreurs locales à un segment.

Cette organisation permet d'imbriquer des layouts, de scinder les responsabilités et d'améliorer la maintenabilité de l'application.

Autre changement majeur : l'App Router prend en charge nativement les React server components. Ces composants s'exécutent côté serveur, ne sont pas envoyés au navigateur et permettent de réduire la charge côté client. Ils sont utilisés par défaut dans le dossier `app/`, ce qui améliore les performances sans effort supplémentaire.

La navigation côté client évolue également. De nouveaux hooks comme `useRouter`, `usePathname` ou `useSearchParams` ont été introduits pour remplacer ceux du Pages Router et s'adapter à cette nouvelle architecture. Ces outils facilitent l'accès aux informations de l'URL, la gestion des paramètres et les changements d'état sans rechargement.

Enfin, le système repose sur la capacité de React à effectuer du streaming via `Suspense`. Cela signifie que l'affichage de la page peut commencer dès que certaines parties sont prêtes, tandis que les autres chargent en arrière-plan. Le résultat : une interface plus rapide, plus fluide et plus résiliente face aux lenteurs réseau ou aux appels asynchrones.

L'App Router redéfinit ainsi les bonnes pratiques de développement avec Next.js, en intégrant par défaut les concepts modernes de React et en simplifiant la création d'expériences complexes, tout en restant accessible aux développeurs habitués à la convention par dossier.

1.4 Tableau comparatif synthétique

Ce tableau récapitule les principales différences entre le Pages Router et l'App Router. Il permet de visualiser en un coup d'œil les évolutions majeures du système de routage dans Next.js, ainsi que les raisons pour lesquelles l'App Router s'impose comme la nouvelle norme.

Caractéristiques	Pages Router	App Router
Dossier racine	`pages/`	`app/`
Layouts imbriqués	Non	Oui
Server components	Non	Oui (par défaut)
Data fetching	`getStaticProps`, etc.	`fetch` + `async/await`
Routage dynamique	`[slug].tsx`	`[slug]/page.tsx`
Streaming/Suspense	Non	Oui
Support long terme	Maintenance uniquement	Standard actuel

1.5 En résumé : la transition vers le modèle App Router

Le Pages Router constitue l'ancienne approche du routage dans Next.js. Bien qu'il reste fonctionnel et maintenu pour assurer la compatibilité des projets existants, il n'intègre pas les avancées majeures du framework ni celles de React.

À l'inverse, l'App Router introduit une architecture moderne, pensée pour répondre aux besoins des applications actuelles : modularité, performances accrues, meilleure expérience utilisateur et intégration native des React server components.

Adopter l'App Router, c'est s'aligner sur l'évolution naturelle de l'écosystème React. Il devient donc indispensable de comprendre sa structure, ses principes de fonctionnement et les opportunités qu'il offre en matière de développement web moderne. Cette transition marque une étape importante dans la manière de concevoir des applications Next.js robustes, évolutives et performantes.

2. Routage dynamique avec l'App Router

2.1 Introduction : routes dynamiques et segments

Dans une application web, il est courant d'avoir besoin d'afficher des pages différentes selon un paramètre présent dans l'URL. Par exemple, pour un blog, chaque article doit être accessible via une URL unique comme `/blog/mon-article`. Pour une plateforme utilisateur, chaque profil peut correspondre à une route de type `/users/john`.

C'est à cela que servent les routes dynamiques : elles permettent de capturer une partie variable de l'URL et d'en faire un paramètre accessible dans le code. Le composant affiché peut alors s'adapter dynamiquement à cette valeur.

L'App Router conserve ce principe, hérité du Pages Router, tout en l'intégrant dans une structure modulaire basée sur les dossiers. Chaque segment dynamique est représenté par un dossier nommé entre crochets, comme `[slug]` ou `[id]`, à l'intérieur du répertoire `app/`.

Cette nouvelle organisation s'inscrit dans la logique globale de l'App Router : découper l'interface en segments d'URL explicites, chacun possédant sa propre page, son propre layout et ses éventuels états de chargement ou de gestion d'erreur. Le routage dynamique reste donc simple à mettre en œuvre, tout en s'adaptant à une architecture plus flexible et évolutive.

2.2 Créer une route dynamique : [param]

Pour créer une route dynamique avec l'App Router, il suffit de créer un dossier nommé entre crochets à l'intérieur du dossier `app/`. Ce nom représente le paramètre dynamique que l'on souhaite capturer dans l'URL.

```
app/
├─ blog/
│  ├─ [slug]/
│  │  └─ page.tsx
```

Dans cet exemple :

– Le dossier `[slug]` correspond à une route dynamique.

– `slug` est le nom du paramètre que l'on récupérera dans le code.

– Le fichier `page.tsx` définit la page affichée pour chaque valeur de `slug`.

Ainsi, les URL comme `/blog/article-1` ou `/blog/hello-world` pointeront toutes vers `app/blog/[slug]/page.tsx`, avec une valeur différente pour le paramètre `slug`.

Comme pour toute route dans l'App Router, il est également possible d'ajouter d'autres fichiers dans ce dossier dynamique :

– `layout.tsx` pour définir un layout propre à chaque page dynamique ;

– `loading.tsx` pour gérer un état de chargement personnalisé ;

– `error.tsx` pour intercepter les erreurs propres à ce segment.

Cette organisation permet de gérer les routes dynamiques de manière claire, modulaire et entièrement alignée avec la philosophie du App Router.

2.3 Accéder aux paramètres dynamiques

Dans l'App Router, lorsqu'une route contient un segment dynamique (par exemple `[slug]`), il est possible de récupérer la valeur transmise dans l'URL directement à l'intérieur d'un composant client grâce au hook `useParams`, fourni par le module `next/navigation`.

Ce hook permet d'accéder aux paramètres définis dans la structure du dossier `app/`, et d'en tirer les valeurs dynamiques à utiliser dans le rendu ou la logique métier du composant. Il est important de noter que `useParams` ne peut être utilisé que dans un composant client, c'est-à-dire un composant commençant par la directive `'use client'`.

Voici un exemple concret dans le cas d'un article de blog, où chaque article est identifié par un `slug` dynamique :

```
'use client';

import { useParams } from 'next/navigation';
```

```
export default function BlogPost() {
  const { slug } = useParams();

  return <h1>Article : {slug}</h1>;
}
```

Dans ce composant :

– `'use client'` indique à Next.js que le composant doit être rendu côté client, condition nécessaire pour utiliser les hooks React.
– Le hook `useParams()` retourne un objet contenant tous les paramètres dynamiques présents dans la route active.
– La clé `slug` correspond ici au nom du dossier `[slug]` utilisé dans l'arborescence du répertoire `app/blog/`.

Cette approche permet de rendre la page de manière entièrement dynamique, en fonction de l'URL consultée, sans rechargement complet de l'application. Elle est particulièrement utile pour des cas d'usage comme l'affichage d'un article de blog, le détail d'un produit, ou encore le profil d'un utilisateur. Grâce à `useParams`, la navigation devient réactive et fluide, en s'appuyant directement sur la structure modulaire du routeur.

2.4 Cas d'usage typiques

Les routes dynamiques trouvent leur utilité dès qu'une partie de l'URL dépend d'une information variable propre à chaque page. Ce mécanisme est particulièrement courant dans les applications web modernes, où les contenus sont souvent générés à partir d'identifiants uniques.

Un premier exemple typique concerne l'affichage d'un article de blog. Le site peut définir une route du type `/blog/[slug]`, où le `slug` représente un identifiant textuel unique. En visitant une URL comme `/blog/decouverte-nextjs`, l'application affiche dynamiquement le contenu de l'article correspondant.

Autre cas courant : les profils utilisateurs. Une plateforme sociale ou un espace membre peut permettre d'accéder à chaque profil via une URL structurée comme `/users/[username]`. Ainsi, en consultant `/users/jane-doe`, l'utilisateur visualise les informations personnelles liées à ce compte.

Dans un contexte e-commerce, les fiches produits utilisent également des routes dynamiques. Une URL du type `/products/[productId]` permet d'accéder à chaque produit en fonction de son identifiant unique. Une page telle que `/products/42` affichera alors les détails du produit portant l'identifiant 42.

Dans chacun de ces cas, la page dynamique extrait la variable contenue dans l'URL, interroge une API ou une base de données pour récupérer les données pertinentes, puis rend le contenu adapté. Ce fonctionnement est naturellement intégré dans l'App Router de Next.js et constitue une base solide pour construire des interfaces complexes à partir de modèles simples.

2.5 Routes catch-all et optionnelles : [...param] et [[...param]]

Une route dynamique de type `[...param]` permet de capturer plusieurs segments consécutifs de l'URL sous forme de tableau. On parle alors de route *catch-all*, car elle intercepte tous les segments situés après un chemin donné, sans limite de profondeur. Ce mécanisme est utile lorsque la structure exacte de l'URL n'est pas connue à l'avance, ou lorsqu'elle est volontairement flexible.

Pour définir une telle route dans l'App Router, il suffit de créer un dossier nommé `[...param]`, avec un fichier `page.tsx` à l'intérieur. Par convention, le nom `param` peut être remplacé par n'importe quel identifiant.

Exemple de structure :

```
app/
├─ docs/
│  ├─ [...slug]/
│  │  └─ page.tsx
```