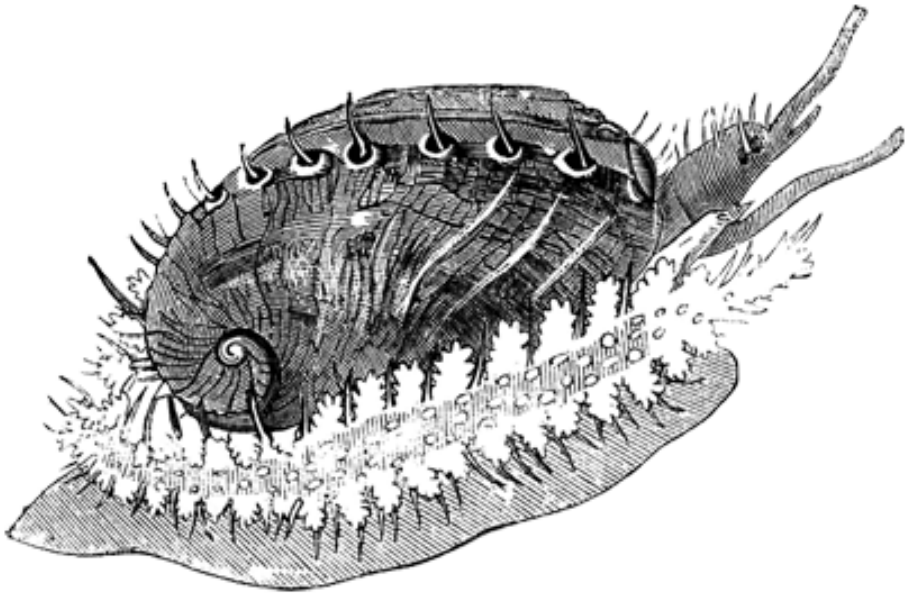

Mastering Next.js

The Definitive Guide

O RLY?

ANTHROPIC:claude-3-5-sonnet-20241022

Next.js 入門: 実践的な Web アプリケーション開発ガイド

2024 年 10 月 23 日

目次

第 1 章

Next.js の基礎と環境構築

Next.js の概要と特徴について説明し、開発環境の構築方法を解説します。Next.js プロジェクトの作成から、基本的なファイル構造、開発サーバーの起動方法までを扱います。

1.1 Next.js とは

Next.js の概要、特徴、React との関係性について説明します。フレームワークとしての利点や、主要な機能についても触れます。

Next.js は、React ベースのフルスタックフレームワークです。Vercel 社によって開発され、React の機能を拡張して、より効率的な Web アプリケーション開発を可能にします。

Next.js の主な特徴として、ファイルベースのルーティング、サーバーサイドレンダリング（SSR）、静的サイト生成（SSG）などが挙げられます。ファイルベースのルーティングでは、`pages` ディレクトリ内のファイル構造がそのまま URL パスとして反映されるため、直感的なルーティング設定が可能です。

Next.js は React の上に構築されているため、React の基本的な機能やコンポーネントベースの開発手法をそのまま活用できます。さらに、開発者体験を向上させる機能として、ホットリロードや TypeScript のサポート、最適化された画像コンポーネントなども提供しています。以下は、Next.js の基本的なページコンポーネントの例です：

```
// pages/index.js
export default function Home() {
  return (
    <div>
      <h1>Welcome to Next.js!</h1>
      <p>This is a simple Next.js page</p>
    </div>
  )
}
```

1.2 開発環境の準備

Node.js のインストール方法から、必要な開発ツールのセットアップまでを解説します。効率的な開発のための VSCode の設定についても説明します。

Next.js の開発を始めるにあたって、まずは開発環境を整えていきましょう。最初に必要なのは、Node.js のインストールです。Node.js の公式サイト（https://nodejs.org/）から、LTS（Long Term Support）版をダウンロードしてインストールしてください。インストールが完了したら、ターミナルで以下のコマンドを実行して、Node.js と npm が正しくインストールされていることを確認します。

```
node -v
npm -v
```

次に、開発効率を高めるためのエディタとして、Visual Studio Code（VSCode）のインストールをお勧めします。VSCode は無料で利用できる高機能なコードエディタです。VSCode をインストールしたら、Next.js 開発に便利な以下の拡張機能をインストールしましょう：

- ES7+ React/Redux/React-Native snippets
- ESLint
- Prettier - Code formatter

これらの拡張機能により、コードの補完や自動フォーマット、構文チェックなどの機能が使えるようになります。また、VSCode の設定ファイル `settings.json` に以下の設定を追加することで、ファイル保存時に自動的にコードがフォーマットされるようになります：

```
{
    "editor.formatOnSave": true,
    "editor.defaultFormatter": "esbenp.prettier-vscode"
}
```

1.3 プロジェクトの作成と構造

create-next-app を使用したプロジェクトの作成方法、生成されるファイル構造の説明、開発サーバーの起動と基本的な操作方法について解説します。

Next.js プロジェクトを作成する最も簡単な方法は、`create-next-app` コマンドを使用することです。このツールを使用することで、必要な設定やファイル構造が自動的に生成されます。以下のコマンドでプロジェクトを作成できます。

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

コマンドを実行すると、いくつかの設定オプションが表示されます。TypeScript の使用や Tailwind CSS の導入などを選択できます。プロジェクトが作成されると、以下のような基本的なディレクトリ構造が生成されます。

```
my-next-app/
```

```
├── app/
│   ├── page.js
│   └── layout.js
├── public/
├── node_modules/
├── package.json
└── next.config.js
```

app ディレクトリには、アプリケーションのメインコードが配置されます。page.js はホームページのコンポーネント、layout.js は共通レイアウトを定義します。public ディレクトリには画像などの静的ファイルを配置します。

開発サーバーを起動するには、プロジェクトディレクトリで以下のコマンドを実行します。

```
npm run dev
```

これにより、デフォルトで http://localhost:3000 でアプリケーションにアクセスできます。開発サーバーはホットリロードに対応しており、ファイルを保存すると自動的に変更が反映されます。

第 2 章

ルーティングとページ管理

Next.js のファイルシステムベースのルーティングについて解説します。動的ルーティング、ネストされたルート、カスタム 404 ページの作成方法を学びます。

2.1 ファイルシステムベースのルーティング入門

Next.js のファイルシステムベースのルーティングの基本概念と利点について説明します。pages ディレクトリの構造と URL パスの対応関係、基本的なページ作成方法を解説します。

Next.js のファイルシステムベースのルーティングは、直感的でシンプルな URL 構造を実現する重要な機能です。このルーティングシステムでは、`pages` ディレクトリ内のファイル構造がそのまま Web サイトの URL 構造となります。

たとえば、`pages/about.js` というファイルを作成すると、自動的に `/about` という URL パスでアクセス可能になります。以下は基本的なページコンポーネントの例です：

```
// pages/about.js
export default function About() {
  return (
    <div>
      <h1>About Page</h1>
      <p>This is the about page content.</p>
    </div>
  )
}
```

また、ネストされたルートを作成する場合は、ディレクトリを作成することで実現できます。例えば、`pages/blog/first-post.js` というファイルは、`/blog/first-post` という URL でアクセスできるようになります。このようなファイルシステムベースのルーティングにより、複雑な設定ファイルを記述することなく、直感的なルーティング構造を実現できます。

2.2 動的ルーティングの実装

動的な URL パラメータを使用したルーティングの実装方法を解説します。[id].js や [[...slug]].js などの動的ルートファイルの作成方法と、パラメータの取得方法について説明します。

Next.js の動的ルーティングは、URL パラメータを使用して柔軟なページ生成を可能にする機能です。動的ルーティングを実装するには、ファイル名に特殊な命名規則を使用します。

最も基本的な動的ルーティングは、`[id].js` のような形式で実装します。例えば、`pages/posts/[id].js` というファイルを作成すると、`/posts/1` や `/posts/2` などの URL にマッチするページを生成できます。以下は実装例です：

```
// pages/posts/[id].js
import { useRouter } from 'next/router'

export default function Post() {
  const router = useRouter()
  const { id } = router.query

  return <h1>Post: {id}</h1>
}
```

より複雑なルーティングには、`[[...slug]].js` 形式を使用します。これは任意の数のパラメータをキャッチできるキャッチオールルートです。例えば、`/posts/a/b/c` のような多段階の URL に対応できます：

```
// pages/posts/[[...slug]].js
import { useRouter } from 'next/router'

export default function Post() {
  const router = useRouter()
  const { slug } = router.query

  return <div>Post: {slug ? slug.join('/') : 'Home'}</div>
}
```

パラメータの値は `useRouter` フックを使用して取得でき、`router.query` オブジェクトからアクセスできます。

2.3 高度なルーティング設定

ネストされたルートの作成方法とカスタム 404 ページの実装方法について説明します。また、ミドルウェアを使用したルーティングの制御方法についても触れます。

Next.js では、より複雑なアプリケーション構造を実現するために、高度なルーティング設定が可能

です。ここでは、ネストされたルートの作成方法とカスタム 404 ページの実装、そしてミドルウェアを使用したルーティング制御について説明します。

ネストされたルートを作成するには、pages ディレクトリ内にサブディレクトリを作成します。例えば、以下のような構造を考えてみましょう：

```
pages/
  blog/
    [id].js
    index.js
  index.js
```

この構造により、/blog や/blog/1 のような URL パスが自動的に生成されます。

カスタム 404 ページを実装するには、pages ディレクトリに 404.js ファイルを作成します：

```
// pages/404.js
export default function Custom404() {
  return <h1>404 - ページが見つかりません</h1>
}
```

ミドルウェアを使用したルーティング制御は、middleware.js ファイルをプロジェクトのルートに作成することで実現できます：

```
// middleware.js
export function middleware(request) {
  if (request.nextUrl.pathname.startsWith('/protected')) {
    return NextResponse.redirect(new URL('/login', request.url))
  }
}
```

このミドルウェアは、保護されたルートへのアクセスを制御し、必要に応じてリダイレクトを行います。

第 3 章

データフェッチングと SSR

Next.js におけるデータフェッチングの方法と SSR（サーバーサイドレンダリング）の実装について説明します。getStaticProps と getServerSideProps の使い分けも解説します。

3.1 データフェッチングの基礎

Next.js におけるデータフェッチングの基本概念と主要な方法について説明します。クライアントサイドとサーバーサイドでのデータ取得の違いや、それぞれのユースケースについて解説します。

Next.js におけるデータフェッチングには、主にクライアントサイドとサーバーサイドの 2 つのアプローチがあります。クライアントサイドでのデータフェッチングは、ブラウザ上で JavaScript を使用してデータを取得する方法です。以下に基本的な例を示します。

```
import { useEffect, useState } from 'react'

function UserProfile() {
  const [data, setData] = useState(null)

  useEffect(() => {
    fetch('/api/user')
      .then((res) => res.json())
      .then((data) => setData(data))
  }, [])

  if (!data) return <div>Loading...</div>
  return <div>{data.name}</div>
}
```

一方、サーバーサイドでのデータフェッチングは、getServerSideProps や getStaticProps を使用してページの初期レンダリング時にデータを取得します。これにより、SEO の向上やパフォーマンスの最適化が可能になります。

3.2 SSR の仕組みと実装方法

サーバーサイドレンダリング（SSR）の基本的な仕組みと、Next.js での実装方法について説明します。SSR がパフォーマンスと検索エンジン最適化にもたらす利点についても触れます。

サーバーサイドレンダリング（SSR）は、Web ページの初期表示時にサーバー側で HTML を生成する仕組みです。Next.js では、`getServerSideProps` 関数を使用することで、ページごとに SSR を実装することができます。

SSR の主な利点として、初期表示の高速化と SEO 対策が挙げられます。クライアント側で JavaScript を実行して DOM を構築する必要がないため、ユーザーは素早くコンテンツを閲覧することができます。また、検索エンジンのクローラーに対して完全な HTML を提供できるため、SEO 面でも有利です。

以下は、SSR を実装した簡単な例です：

```
// pages/posts/[id].js
export async function getServerSideProps(context) {
  const { id } = context.params;
  const res = await fetch(‘https://api.example.com/posts/${id}‘);
  const post = await res.json();

  return {
    props: { post },
  };
}

export default function Post({ post }) {
  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </div>
  );
}
```

この例では、ページアクセス時に `getServerSideProps` 関数が実行され、API からデータを取得して HTML を生成します。これにより、常に最新のデータを含んだページをユーザーに提供することができます。

3.3 getStaticProps と getServerSideProps の活用

静的生成のための getStaticProps と、サーバーサイドレンダリングのための getServerSideProps の違いと使い分けについて、具体的な実装例を交えて解説します。それぞれのメリット・デメリットや

適切な使用シーンについても説明します。

3.3.1 データフェッチング手法の概要

Next.js におけるデータフェッチングの主要な 2 つの方法である getStaticProps と getServerSideProps の基本的な概念と役割について説明します。それぞれの特徴と使用目的を概観し、以降の詳細な解説の導入とします。

Next.js におけるデータフェッチングには、主に 2 つの重要な方法があります。1 つ目は `getStaticProps` で、ビルド時にデータを取得する手法です。このメソッドは、ブログ記事やドキュメントなど、頻繁に更新されないコンテンツに適しています。

2 つ目は `getServerSideProps` で、リクエストごとにサーバーサイドでデータを取得する方法です。ユーザー固有のデータや、リアルタイムで更新される情報を扱う場合に使用します。以下に基本的な使用例を示します：

```
// getStaticProps の例
export async function getStaticProps() {
  const data = await fetchBlogPosts()
  return {
    props: { posts: data }
  }
}

// getServerSideProps の例
export async function getServerSideProps() {
  const userData = await fetchUserData()
  return {
    props: { user: userData }
  }
}
```

3.3.2 getStaticProps による静的生成

ビルド時にデータを取得してページを生成する getStaticProps の実装方法を解説します。具体的なコード例を用いて、静的生成のメリットである高速なページ表示とキャッシュの活用方法について説明します。また、revalidate オプションを使用した増分静的再生成（ISR）についても触れます。

`getStaticProps` は、Next.js の静的生成機能を実現する重要なメソッドです。このメソッドを使用することで、ビルド時にページのデータを事前に取得し、静的な HTML を生成することができます。

以下は `getStaticProps` の基本的な使用例です：

```
export async function getStaticProps() {
  const data = await fetchData()
  return {
```

```
    props: {
      data
    }
  }
}
```

静的生成の最大のメリットは、ページの読み込み速度が高速になることです。事前にデータを取得して HTML を生成するため、ユーザーがページにアクセスした際に追加のデータ取得が不要となります。このアプローチは、ブログ記事やドキュメントページなど、頻繁に更新されないコンテンツに特に適しています。getStaticProps は、Next.js の静的生成機能を実装するための重要な関数です。この関数を使用することで、ビルド時にデータを取得し、静的な HTML を生成することができます。以下に基本的な実装例を示します。

```
// pages/posts.js
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
  }
}

export default function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

上記の例では、外部 API からブログ記事のデータを取得し、それをページコンポーネントに渡しています。getStaticProps は必ず props オブジェクトを返す必要があります。Next.js の増分静的再生成（ISR: Incremental Static Regeneration）は、静的生成されたページを定期的に再生成する機能です。getStaticProps 関数内で revalidate オプションを設定することで実装できます。以下に具体例を示します。

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data')
```

```
  const data = await res.json()

  return {
    props: {
      data,
    },
    revalidate: 60 // 60 秒ごとに再生成
  }
}
```

上記の例では、ページは 60 秒ごとにバックグラウンドで再生成されます。revalidate の値は秒単位で指定し、その間はキャッシュされたコンテンツが提供されます。これにより、常に最新のデータを提供しながら、パフォーマンスを維持することができます。

3.3.3 getServerSideProps を用いたサーバーサイドレンダリング

リクエスト時にデータを取得する getServerSideProps の実装方法について、実践的な例を用いて解説します。動的なデータ表示が必要なケースでの活用方法や、認証情報の取り扱いなど、サーバーサイドレンダリングならではの特徴を説明します。

getServerSideProps は、Next.js のページコンポーネントでサーバーサイドレンダリング時にデータを取得するための関数です。この関数はページがリクエストされるたびにサーバー側で実行され、取得したデータをページコンポーネントに props として渡すことができます。

基本的な実装例を見てみましょう。以下は外部 API からデータを取得する例です：

```
export async function getServerSideProps(context) {
  const res = await fetch('https://api.example.com/data')
  const data = await res.json()

  return {
    props: {
      data,
    },
  }
}
```

この関数は context パラメータを受け取り、その中にはリクエストやレスポンスに関する情報が含まれています。戻り値として props オブジェクトを含むオブジェクトを返す必要があります。動的なデータを表示する際に、Next.js の getServerSideProps 関数は非常に強力なツールとなります。この関数を使用することで、リクエストごとにサーバーサイドでデータを取得し、ページをレンダリングすることができます。以下に、外部 API からデータを取得する具体的な例を示します。

```
export async function getServerSideProps() {
  const res = await fetch('https://api.example.com/data')
```

```
  const data = await res.json()

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

このコードでは、getServerSideProps 関数内で外部 API からデータを取得し、そのデータをページコンポーネントに渡しています。この方法により、SEO 対策やパフォーマンスの最適化が可能となります。Next.js での認証情報の取り扱いについて、特に getServerSideProps での実装方法を説明します。認証情報を安全に管理するためには、サーバーサイドでの適切な処理が重要です。以下に基本的な実装例を示します。

```
export async function getServerSideProps(context) {
  const session = await getSession(context)

  if (!session) {
    return {
      redirect: {
        destination: '/login',
        permanent: false,
      },
    }
  }

  return {
    props: { session }
  }
}
```

このコードでは、getSession 関数を使用してセッション情報を取得し、認証されていないユーザーを自動的にログインページにリダイレクトしています。これにより、保護されたページへの不正アクセスを防ぐことができます。セッション情報はサーバーサイドで安全に管理され、クライアントには必要最小限の情報のみが渡されます。

第 4 章

最適化とデプロイメント

画像最適化、メタデータの設定、パフォーマンスチューニングについて解説します。また、Vercel へのデプロイ方法も説明します。

4.1 パフォーマンス最適化の基礎

Next.js における基本的なパフォーマンス最適化手法について解説します。画像の最適化機能（next/image）の使用方法や、メタデータの適切な設定方法を具体的な例を用いて説明します。

Next.js では、パフォーマンスの最適化のために様々な機能が提供されています。その中でも特に重要な機能の一つが、`next/image` コンポーネントを使用した画像の最適化です。

`next/image` コンポーネントを使用することで、画像の自動最適化が可能になります。以下のように実装することで、画像の読み込み時に最適なサイズとフォーマットに自動変換されます：

```
import Image from 'next/image'

function MyComponent() {
  return (
    <Image
      src="/example.jpg"
      alt="Example Image"
      width={500}
      height={300}
      priority={true}
    />
  )
}
```

また、メタデータの適切な設定もパフォーマンスに影響を与える重要な要素です。`next/head` コンポーネントを使用することで、SEO に最適化されたメタデータを設定できます：

```
import Head from 'next/head'

function MyPage() {
```

```
  return (
    <>
      <Head>
        <title>ページタイトル</title>
        <meta name="description" content="ページの説明" />
      </Head>
      {/* ページコンテンツ */}
    </>
  )
}
```

これらの最適化機能を適切に使用することで、Web アプリケーションの読み込み速度と表示パフォーマンスを大幅に改善することができます。

4.2 高度なパフォーマンスチューニング

Lighthouse score の改善方法、コード分割、動的インポートなど、より進んだパフォーマンス最適化テクニックについて解説します。実際のパフォーマンス計測方法とその改善手順を示します。

Next.js アプリケーションのパフォーマンスを最適化するために、まず Lighthouse score の改善から始めましょう。Lighthouse は、Web ページの品質を測定する Google のオープンソースツールです。Performance、Accessibility、Best Practices、SEO の 4 つの観点からスコアを算出します。

特に Performance スコアを向上させるための重要な施策として、コード分割（Code Splitting）があります。Next.js では、`dynamic import` を使用することで、必要なコンポーネントを必要なタイミングで読み込むことができます。以下に具体例を示します：

```
import dynamic from 'next/dynamic'

const DynamicComponent = dynamic(() => import('../components/hello'))

function HomePage() {
  return (
    <div>
      <DynamicComponent />
    </div>
  )
}
```

また、画像の最適化も重要です。Next.js の `Image` コンポーネントを使用することで、自動的に画像の最適化が行われます。`next.config.js` でキャッシュの挙動をカスタマイズすることも可能です。

パフォーマンスの計測には、Chrome DevTools の Performance タブや Next.js の組み込みアナリティクス機能を活用します。特に初期ロード時間（First Contentful Paint）とインタラクティブになるまでの時間（Time to Interactive）に注目して改善を進めることをお勧めします。

4.3 Vercel へのデプロイメント

Next.js アプリケーションを Vercel にデプロイする手順を詳しく説明します。GitHub との連携方法、環境変数の設定、デプロイ後の動作確認まで、実践的なデプロイフローを解説します。

Next.js アプリケーションを本番環境にデプロイする方法として、Vercel を利用する方法を説明します。Vercel は、Next.js の開発元が提供している高性能なホスティングプラットフォームです。

まず、GitHub のリポジトリにプロジェクトをプッシュします。その後、Vercel のウェブサイト（`vercel.com`）にアクセスし、アカウントを作成します。「New Project」ボタンをクリックし、GitHub リポジトリとの連携を行います。

```
# GitHub へのプッシュ
git add .
git commit -m "Initial commit"
git push origin main
```

Vercel でプロジェクトを作成する際、環境変数の設定が必要な場合は、「Environment Variables」セクションで設定できます。例えば、API キーなどの機密情報を設定する場合は、以下のように名前と値のペアで登録します：

```
DATABASE_URL=your_database_url
API_KEY=your_api_key
```

設定が完了したら、「Deploy」ボタンをクリックしてデプロイを開始します。デプロイが完了すると、自動的にプロジェクトの URL が生成されます。この URL にアクセスして、アプリケーションが正常に動作していることを確認します。

その後のコード変更は、GitHub リポジトリにプッシュするだけで自動的に Vercel に反映されます。これにより、継続的なデプロイメント（CD）が実現できます。