# 作って学ぶ Next.js/React Webサイト構築

エビスコム 著

副読本

Next.js 13 対応ガイド

マイナビ

この PDF は、「作って学ぶ Next.js React Web サイト構築」の Next.js 13 対応ガイドです。必要に応じて、下記を参照してください。


**Next.js 13で新規にプロジェクトを作成して本書を進める場合** ⟶ **P.5**

**本書で作成したプロジェクトをApp Routerへ移行する場合** ⟶ **P.11**



作って学ぶ

## Next.js/React Webサイト構築

https://book.mynavi.jp/ec/products/detail/id=130848

https://ebisu.com/next-react-website/

https://amzn.to/3RvfR8D

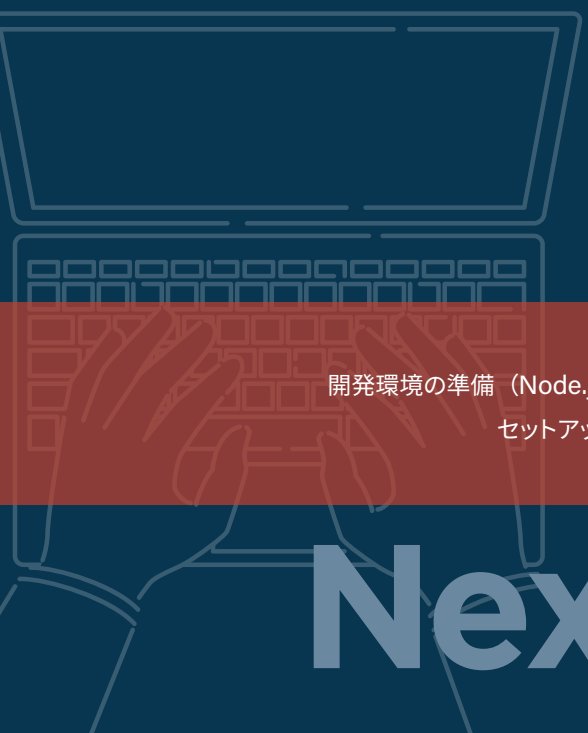
# もくじ

Chapter

# 1

# Next.js 13 で本書を進める方法

ここでは、Next.js 13 で新規にプロジェクトを作成し、本書を進める際に必要になる設定についてまとめています。



開発環境の準備（Node.js のインストール＆セットアップ）については
セットアップ PDF を参照してください

# 1.0 create-next-app

現在の create-next-app は Next.js 13 用のものになっています。そのため、本書が解説に使っている create-next-app とは挙動が異なります。

本書で解説しているプロジェクトを進めるうえで、選択肢は次の 2 つです。

**1** **Next.js 12** を使ってプロジェクトを作成して進める ⟶ P.7

本書の解説のままプロジェクトを進めることができます。

**2** **Next.js 13** を使ってプロジェクトを作成して進める ⟶ P.8

本書で解説している `next/image` と `next/link` に関しては修正が必要です（修正方法はこの PDF で解説しています）。

**Next.js 13** でも Pages Router（本書で解説している、`pages` ディレクトリを使ってサイトを作成していく方法）は問題なく使えます。そのため、**2** をオススメします。

Next.js 12、Next.js 13 のそれぞれでプロジェクトを作成する方法は、P.7 および P.8 のとおりです。

# 1.1 create-next-app
# Next.js 12 で プロジェクトを作成する場合

Next.js 12 を使ってプロジェクトを作成する方法は以下のとおりです。

```
$ npx create-next-app@12 blog

Need to install the following packages:
  create-next-app@12.3.4
Ok to proceed? (y) y
Creating a new Next.js app in /home/xxxx/blog.

Using yarn.

Installing dependencies:
- react
- react-dom
- next

yarn add v1.22.19
info No lockfile found.
…略…
success Saved lockfile.
success Saved 20 new dependencies.
info Direct dependencies
├─ next@13.4.4
├─ react-dom@18.2.0
└─ react@18.2.0
…略…

Initialized a git repository.

Success! Created blog at /home/xxxx/blog


$ cd blog
$ rm yarn.lock

$ npm install next@12 eslint-config-next@12
```

Next.js 12の世代のcreate-next-appを使って、プロジェクトを作成します。
ただし、create-next-appのバージョンとNext.jsのバージョンの関連性はありません。

プロジェクトの構成はnext@12になりますが、next@13がインストールされてしまいます。

プロジェクトの作成が完了。

yarnを使うのであれば、残しても問題ありません。

最後に、next@12に入れ替えます。

ここからは、書籍の制作ステップに進んでください。

# 1.2 create-next-app
# Next.js 13 で プロジェクトを作成する場合

Next.js 13 を使ってプロジェクトを作成する方法は以下のとおりです。

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

Need to install the following packages:
  create-next-app@13.4.4
Ok to proceed? (y) y

? What is your project named? › blog
```

「プロジェクト名は何ですか」… プロジェクト名を指定。

```
? Would you like to use TypeScript with this project? › No / Yes
```

「このプロジェクトでTypeScriptを使いたいですか?」… Noを選択。

```
? Would you like to use ESLint with this project? › No / Yes
```

「このプロジェクトでESLintを使いたいですか?」… Yesを選択。

```
? Would you like to use Tailwind CSS with this project? › No / Yes
```

「このプロジェクトでTailwind CSSを使用したいですか?」… Noを選択。

```
? Would you like to use `src/` directory with this project? › No / Yes
```

「このプロジェクトでsrc/ディレクトリを使用したいですか?」… Noを選択。

```
? Use App Router (recommended)? › No / Yes
```

「App Routerを使いますか?」… 作っていても問題ないのでYesを選択。

```
? Would you like to customize the default import alias? › No / Yes
```

「デフォルトのインポートエイリアスをカスタマイズしたいですか?」… Noを選択。

```
Creating a new Next.js app in /home/xxxx/blog.

Using npm.

Initializing project with template: app


Installing dependencies:
- react
- react-dom
- next
- eslint
- eslint-config-next
…略…

Success! Created blog at /home/xxxx/blog
```

Next.js 13 で作成したプロジェクトでは、Next.js を起動した際のページの表示が異なります。初期のページのデザインが変更になっただけですので、気にせずに進めてください。

ただし、「作って学ぶ Next.js React Web サイト構築」は Next.js 12 を前提として解説しているため、Next.js 13 では以下の 2 つの点で注意する必要があります。

- next/image
- next/link

## ❖ next/image

`next/image` は、次のように変更されました。

- v12 の `next/image` → v13 の `next/legacy/image`
- v12 の `next/future/image` → v13 の `next/image`

本書では従来の `next/image`、つまり、`next/legacy/image` を前提としたスタイリングをしていますので、`next/legacy/image` を import して利用する形にします。

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

> `next/image`（従来の `next/future/image`）を使う場合には、スタイリングの変更が必要です。詳しくは下記を参照してください。
>
> Next.js 13のnext/image（next/future/image）へ移行する
> https://ebisu.com/note/next-image-migration/

## ❖ next/link

`next/link` も変更されました。これまで、以下のように <a> を独立させて書かなければならなかった <Link> コンポーネントですが、

```
<Link href="/">
  <a className={boxOn ? styles.box : styles.basic}>CUBE</a>
</Link>
```

Next.js 13 では、以下のように書く形になりました。

```
<Link href="/" className={boxOn ? styles.box : styles.basic}>
  CUBE
</Link>
```

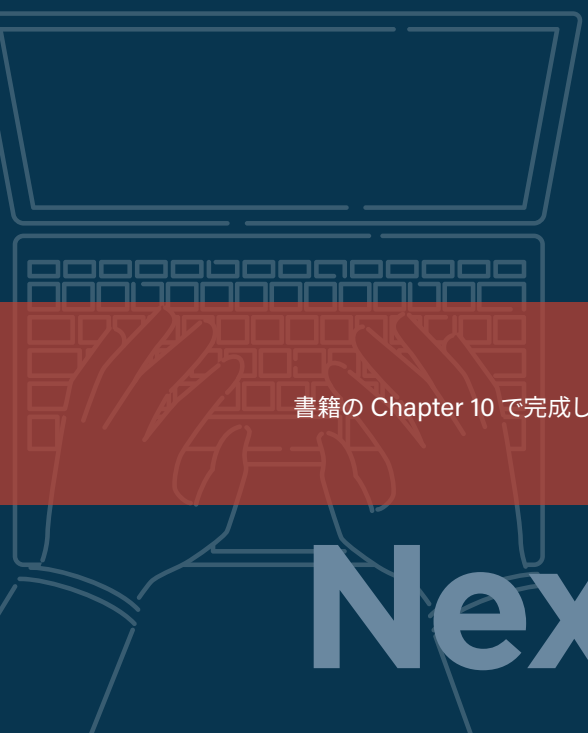
以上を踏まえて、書籍の制作ステップに進んでください。

Chapter

# 2

# 本書のプロジェクトを App Router へ移行する方法

本書で作成したプロジェクトを App Router へ移行する方法を解説します。



書籍の Chapter 10 で完成したプロジェクトの移行を想定して解説しています

Next.js/React

# 2.1 Next 13 プロジェクトの Next.js を最新バージョンへ

Next.js 13.4 で App Router が Stable となりました。

ただし、React Server Components を中心とした React を取り巻く環境の整備はこれからです。また、Next.js で進められている新機能の追加による影響も少なくなく、App Router の真価が発揮できるようになるまでには今しばらく時間がかかりそうです。

そこで、本書で解説した Pages Router ベースのサイトを App Router へ移行する形で、App Router に関して簡単に解説していきます。移行方法に関しては、以下のドキュメントを参考にしています。

App Router Incremental Adoption Guide
https://nextjs.org/docs/app/building-your-application/upgrading/app-router-migration

v13.4.3 で確認しています。

## プロジェクトの Next.js を最新バージョンへ

※すでに完了している方は P.17 へ進んでください。
まず、プロジェクトで扱っている Next.js を最新バージョンへアップデートします。

```
# next.js と React を最新バージョンへ
$ npm install next@latest react@latest react-dom@latest

# 合わせて、eslint も最新バージョンへ
$ npm install -D eslint-config-next@latest
```

## ❖ next/image と next/link の修正

Next.js 13 では、`next/image` と `next/link` が大きく変わりました。`next/image` は、

- v12 の `next/image` → v13 の `next/legacy/image`
- v12 の `next/future/image` → v13 の `next/image`

へと変更されました。そのため、少なくとも `next/legacy/image` を import するように修正する必要があります。

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

`next/link` は、これまで以下のように <a> を独立させて書かなければならなかった <Link> コンポーネントですが、

```
<Link href="/">
  <a className={boxOn ? styles.box : styles.basic}>CUBE</a>
</Link>
```

Next.js 13 では、以下のように書く形になりました。

```
<Link href="/" className={boxOn ? styles.box : styles.basic}>
  CUBE
</Link>
```

そこで、プロジェクトの中で、`next/image` と `next/link` を使っている部分を修正します。修正が必要なファイルは以下のとおりです。

### next/imageの修正が必要なファイル

- components/convert-body.js
- components/hero.js
- components/posts.js
- pages/about.js
- pages/blog/[slug].js

### next/linkの修正が必要なファイル

- components/logo.js
- components/nav.js
- components/pagination.js
- components/post-categories.js
- components/posts.js

```
# プロジェクトのディレクトリに移動した状態で、以下のコマンドを実行します

# git で commit or stash する
$ npx @next/codemod new-link ./components/

# git で commit or stash する
$ npx @next/codemod next-image-to-legacy-image ./components/

# git で commit or stash する
$ npx @next/codemod next-image-to-legacy-image ./pages/
```

これで、Next.js 13 でも問題なくビルドできるようになります。

ただし、components/nav.js を確認してみると、`next/link` が `legacyBehavior` 属性を使う形で、従来のままになっているのが確認できます。`onClick` 属性による処理があるためのようです。

```
      <li>
        <Link href="/" legacyBehavior>
          <a onClick={closeNav}>Home</a>
        </Link>
      </li>
      <li>
        <Link href="/about" legacyBehavior>
          <a onClick={closeNav}>About</a>
        </Link>
      </li>
      <li>
        <Link href="/blog" legacyBehavior>
          <a onClick={closeNav}>Blog</a>
        </Link>
      </li>
    </ul>
  </nav>
```

components/nav.js

このままでも問題はありませんし、以下のように修正することもできます。

```
      <li>
        <Link href="/" onClick={closeNav}>
          Home
        </Link>
      </li>
      <li>
        <Link href="/about" onClick={closeNav}>
          About
        </Link>
      </li>
      <li>
        <Link href="/blog" onClick={closeNav}>
          Blog
        </Link>
      </li>
    </ul>
  </nav>
```

components/nav.js

`next/image` は、`next/legacy/image` を使っている状態です。さらに、Next.js 13 の `next/image`（next/future/image）に置き換えるための codemod も用意されています。
ただし、CSS の検討もしなければならないため、`next/legacy/image` を使った状態で進めます。
`next/image`（next/future/image）への変更に関しては、下記を参照してください。

> Next.js 13のnext/image（next/future/image）へ移行する
> https://ebisu.com/note/next-image-migration/

## ❖ domains から remotePatterns へ

外部サイトの画像を使う場合、`next.config.js` の `domains` でドメインを設定していましたが、より詳細に設定するため、`remotePatterns` の使用が推奨されています。

> remotePatterns
> https://nextjs.org/docs/app/api-reference/components/image#remotepatterns

たとえば、microCMS の画像は URL が

```
https://images.microcms-assets.io/assets/xxxxxxxxxx/…/～.jpg
```

という形式になっています。「https://images.microcms-assets.io/assets/xxxxxxxxxx/」から始まる画像の処理を許可する場合、次のように指定します。

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  reactStrictMode: true,
  images: {
    // domains: ['images.microcms-assets.io'],
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'images.microcms-assets.io',
        port: '',
        pathname: '/assets/xxxxxxxxxx/**',
      },
    ],
  },
}

module.exports = nextConfig
```

next.config.js

「xxxxxxxxxx」の部分は microCMS のサービスごとに異なります。

# 2.2 Next 13 App Router の基本と React Server Components

App Router への移行を始める前に、おさえておきたいポイントがいくつかあります。そこで、そうしたポイントの確認から始めましょう。

## React Server Components

これまでの React では、コンポーネントはクライアントでレンダリングされます。これは、Next.js で SG や SSR を使っていても変わりません。

たとえば、デプロイしたサイトをブラウザーのデベロッパーツールで確認しながらページを移動してみましょう。JavaScript が有効なブラウザーであれば、最初のわずかな HTML 以外は JSON がダウンロードされ、レンダリングの結果としてページが表示されていることが確認できます。


ログをクリアしてページを移動

JSON

この中身は
pagePropsです。

また、本書の中で作成した `ConvertDate` コンポーネントに date-fns がバンドルされるのも、`ConverBody` コンポーネントに html-react-parser がバンドルされるのも、クライアントでレンダリングするためです。
しかし、すべてのコンポーネントがクライアント側で変化するわけではないため、クライアントでレンダリングする必要はありません。`ConvertDate` や `ConverBody` も同様です。

そこで、React18 で導入されたのが **React Server Components** です。

Server Components はこれまでのコンポーネント（Client Components）とは異なり、サーバーでレンダリングされます。そして、そのレンダリング結果をクライアントへ送ります。クライアントでの処理は軽くなり、バンドルされる JavaScript が削減されることになります。
Server Components は、Client Components と比べて以下のような違いがあります。

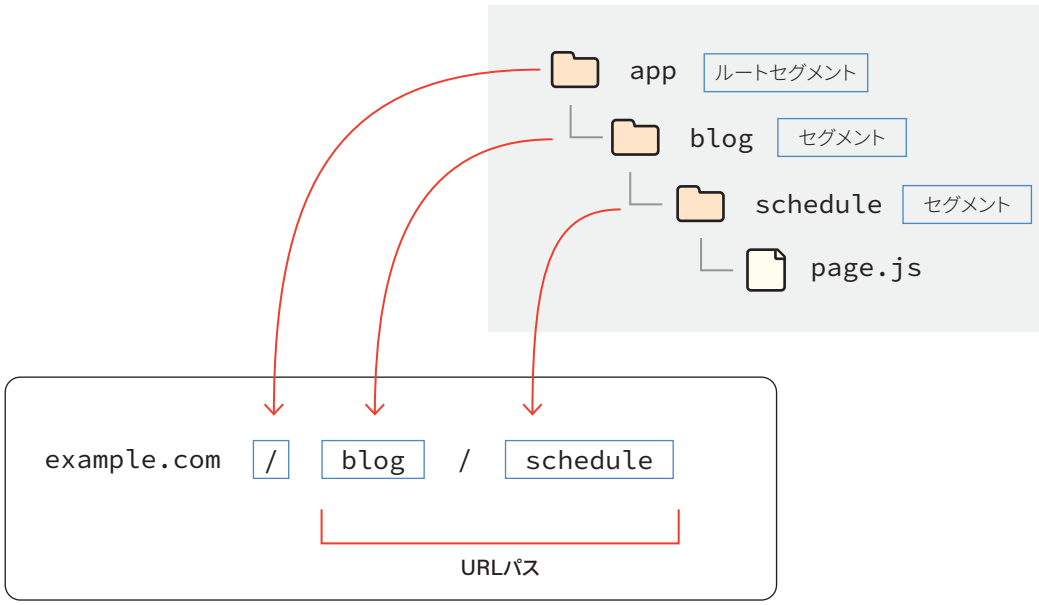
- サーバーでレンダリングされるため、データに直接アクセスできます
- レンダリング結果を扱うため、JavaScript はバンドルされません
- onClick() や onChange() などのイベント処理は扱えません
- React の State（useState）を使えません
- クライアントでのレンダリングがないため、ライフサイクルがなく、useEffects などは使えません
- ブラウザーでのみ使える API は使えません

そのため、コンポーネントの機能に応じて Server Components と Client Components を使い分けることになります。

このような Server Components ですが、Next.js 13 で導入された App Router では、コンポーネントを標準で Server Components として扱います。逆に、これまで通り Client Components としてレンダリングしたいコンポーネントは明示しなければなりませんので、注意が必要です。

## ❖ App Router のルーティング

App Router では、ページコンポーネントを page.js（jsx|ts|tsx）で、セグメントをディレクトリで構成します。




そのため、本書のサンプルで作成したブログの

- URL パスが `/blog/schedule` のページは、`app/blog/schedule/page.js`
- URL パスが `/blog/category/fun` のページは、`app/category/fun/page.js`

といった構成になります。

また、各セグメント（ディレクトリ）には以下のコンポーネントを配置できます。これらのファイルは各セグメントに配置して、ネストすることもできます。

| | |
|---|---|
| page.js | ページコンポーネント |
| layout.js | 子階層のセグメントやページコンポーネントをラップするコンポーネント。共通するレイアウトや UI などのためのもの。`app/layout.js` は Root Layout と呼ばれ、pages ディレクトリの `_document.js` と `_app.js` の代わりとして使うことができる。 |
| loading.js | 子階層のセグメントやページコンポーネントをラップし、それらがロードされる間に表示するコンポーネント（今回は扱っていません）。 |
| error.js | エラーが発生した際に表示するコンポーネント（今回は扱っていません）。 |
| not-found.js | ページが見つからなかった場合に表示されるコンポーネント。 |
| route.js | API のエンドポイントの作成。 |

ここでは標準的なコンポーネントを紹介しています。すべてのコンポーネントに関しては、公式のドキュメントを参照してください。

File Conventions
https://nextjs.org/docs/app/building-your-application/routing#file-conventions

## ❖ SG や SSR、ISR はそのまま

React Server Components を使うことになりましたが、SG や SSR、ISR といった機能がなくなったわけではありません。設定方法は変わりましたが、これまで通り使うことができます。

## ❖ グローバルスタイルはどのコンポーネントでも使えます

グローバルスタイルは、Pages Router では `app.js` でしか使えないという制限がありましたが、App Router ではそういった制限はなくなりました。

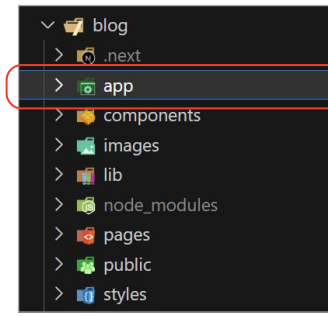
• • •

それでは、app ディレクトリへの移行を進めます。必要な設定は、移行作業を進めながら解説していきます。

# 2.3 Next 13 app ディレクトリの準備と Root Layout コンポーネントの作成

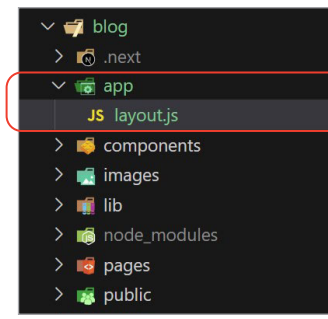
## App Router の準備

まず、プロジェクトのディレクトリに `app` というディレクトリを作成します。`pages` との共存も可能ですので、URL のコンフリクトに注意しながら移行を進めていきます。




## Root Layout コンポーネントの作成

続いて、Root Layout コンポーネントを作成します。以下の内容で `app/layout.js` を作成します。




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

app/layout.js

`app` ディレクトリでは、`layout.js` というファイルで Layout コンポーネントを作成することで、共通のレイアウトや UI を扱います。そのため、ページコンポーネントなどを children props として受け取れる必要があります。

また、`layout.js` は各セグメントに自由に用意できますが、`app` ディレクトリの一番上の階層の Layout コンポーネントは **Root Layout** と呼ばれ、必ず用意する必要があります。すべてのページに共有されるコンポーネントであり、<html> と <body> を定義するのもこのコンポーネントです。

さらに、`pages` ディレクトリの `_document.js` と `_app.js` の代わりとして使うことができます。そこで、`_document.js` で設定していた <html> タグの lang 属性、`_app.js` で設定していたグローバルスタイルと Layout コンポーネント、さらに Font Awesome の設定を Root Layout に持ってきます。

```
import 'styles/globals.css';
import Layout from 'components/layout';

import { siteMeta } from 'lib/constants';
const { siteLang } = siteMeta;

// Font Awesome の設定
import '@fortawesome/fontawesome-svg-core/styles.css';
import { config } from '@fortawesome/fontawesome-svg-core';
config.autoAddCss = false;

export default function RootLayout({ children }) {
 return (
  <html lang={siteLang}>
   <body>
    <Layout>{children}</Layout>
   </body>
  </html>
 );
}
```

app/layout.js

pages を機能させておくため、`_document.js` と `_app.js` はそのままにしておきます。

## ❖ メタデータ

App Router でメタデータを扱うには、次のような機能を利用します。

| | |
|---|---|
| Config-based Metadata | layout.js と page.js から metadata オブジェクトを export することで、メタデータを設定します。 |
| File-based Metadata | 特定のファイルを配置することで、メタデータが自動的に出力されます。 |

これらの機能を組み合わせて、metadata を構成していくことになります。そのため、各ページの移行を済ませたあとで、じっくりと検討することにします。

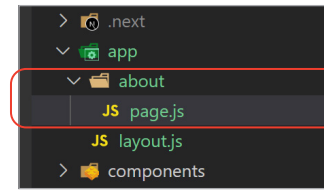
# 2.4 アバウトページの移行

Next 13

外部からのデータを必要としないアバウトページの移行から始めます。

## アバウトページの移動

アバウトページを移行するために、`pages/about.js` を `app/about/page.js` へ移動します。




そして、`npm run dev` を実行し、アバウトページ `/about` を表示すると、次のようなエラーメッセージが表示されます。

```
error - ./components/accordion.js

You're importing a component that needs useRef. It only works in a Client
Component but none of its parents are marked with "use client", so they're
Server Components by default.

   ,----
 1 | import { useState, useRef } from 'react'
   :                    ^^^^^^
   `----

Maybe one of these should be marked as a client entry with "use client":
  components/accordion.js
  app/about/page.js
```

メッセージにある通り、エラーの原因は `components/accordion.js`（アコーディオン）です。その理由は「useRef を使ったコンポーネントをインポートしていますが、useRef は Server Component では機能しません。Client Component として処理するため、"use client" とマークしてください」といった内容です（useState も使っています）。

そのため、`components/accordion.js` のコードの先頭に `'use client'` を追加します。

```
'use client'

import { useState, useRef } from 'react'
import styles from 'styles/accordion.module.css'
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'
import { faCircleChevronDown } from '@fortawesome/free-solid-svg-icons'
…略…
```

components/accordion.js

再びアバウトページ `/about` を表示すると、エラーメッセージが変わり、次のように表示されます。

```
error - ./components/meta.js

You have a Server Component that imports next/router. Use next/navigation
instead.

Maybe one of these should be marked as a client entry "use client":
  components/meta.js
  app/about/page.js
```

今度は、`components/meta.js`（メタデータ）が原因のエラーです。「Server Component では next/router ではなく、next/navigation から import してください」といった内容です。

`components/meta.js` を修正しても良いのですが、App Router でのメタデータは metadata オブジェクトを使って指定する必要があります。そのため、`app/about/page.js` の `<Meta />` に関連する部分をコメントアウトしておきます。

```
// import Meta from 'components/meta'
import Container from 'components/container'
import Hero from 'components/hero'
import PostBody from 'components/post-body'
import Contact from 'components/contact'
…略…

export default function About() {
  return (
    <Container>
      {/* <Meta
        pageTitle="アバウト"
        pageDesc="About development activities"
        pageImg={eyecatch.src}
        pageImgW={eyecatch.width}
        pageImgH={eyecatch.height}
      /> */}

      <Hero title="About" subtitle="About development activities" />
      …略…
```

app/about/page.js

エラーが `components/nav.js`（ナビゲーションメニュー）が原因のものに変わります。

```
error - ./components/nav.js

You're importing a component that needs useState. It only works in a Client
Component but none of its parents are marked with "use client", so they're
Server Components by default.

   ,----
 1 | import { useState } from  'react'
   :          ^^^^^^^^
   `----

Maybe one of these should be marked as a client entry with "use client":
  components/nav.js
  components/header.js
  components/layout.js
  app/layout.js
```

最初のエラーと同様に useState を使っていることが原因です。`components/nav.js` の先頭に `'use client'` を追加します。

```
'use client'

import { useState } from 'react'
import Link from 'next/link'
import styles from 'styles/nav.module.css'

export default function Nav() {
  const [navIsOpen, setNavIsOpen] = useState(false)
…略…
```

components/nav.js

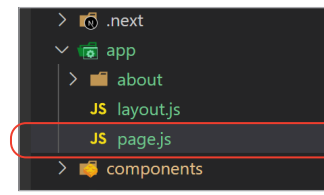
これでエラーが消え、アバウトページが無事に表示されます。Root Layout も設定が済んでいますので、ヘッダーとフッターも表示されます。

# 2.5 Next 13 トップページと記事一覧ページの移行

データの取得が必要なトップページと記事一覧ページの App Router への移行を進めます。

## ❖ トップページの移動

トップページを App Router に移行するため、`pages/index.js` を、`app/pages.js` へ移動します。

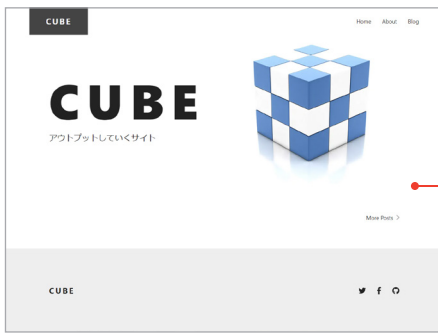



しかし、トップページではインデックスを表示するために getStaticProps でデータを取得しているため、そのままでは機能しません。
データを取得して表示している部分をコメントアウトし、ページが表示できることを確認します。また、Meta コンポーネントも metadata オブジェクトへ移行しますので、コメントアウトしておきます。

```
import { getAllPosts } from 'lib/api'
// import Meta from 'components/meta'
import Container from 'components/container'
import Hero from 'components/hero'
import Posts from 'components/posts'
import Pagination from 'components/pagination'
import { getPlaiceholder } from 'plaiceholder'

// ローカルの代替アイキャッチ画像
import { eyecatchLocal } from 'lib/constants'

export default function Home({ posts }) {
  return (
    <Container>
      {/* <Meta /> */}

      <Hero title="CUBE" subtitle="アウトプットしていくサイト" imageOn />
```

```
      {/* <Posts posts={posts} /> */}
      <Pagination nextUrl="/blog" nextText="More Posts" />
    </Container>
  )
}

// export async function getStaticProps() {
//   const posts = await getAllPosts(4)

//   for (const post of posts) {
//     if (!post.hasOwnProperty('eyecatch')) {
//       post.eyecatch = eyecatchLocal
//     }
//     const { base64 } = await getPlaiceholder(post.eyecatch.url)
//     post.eyecatch.blurDataURL = base64
//   }

//   return {
//     props: {
//       posts: posts,
//     },
//   }
// }
```

app/page.js




トップページ
/

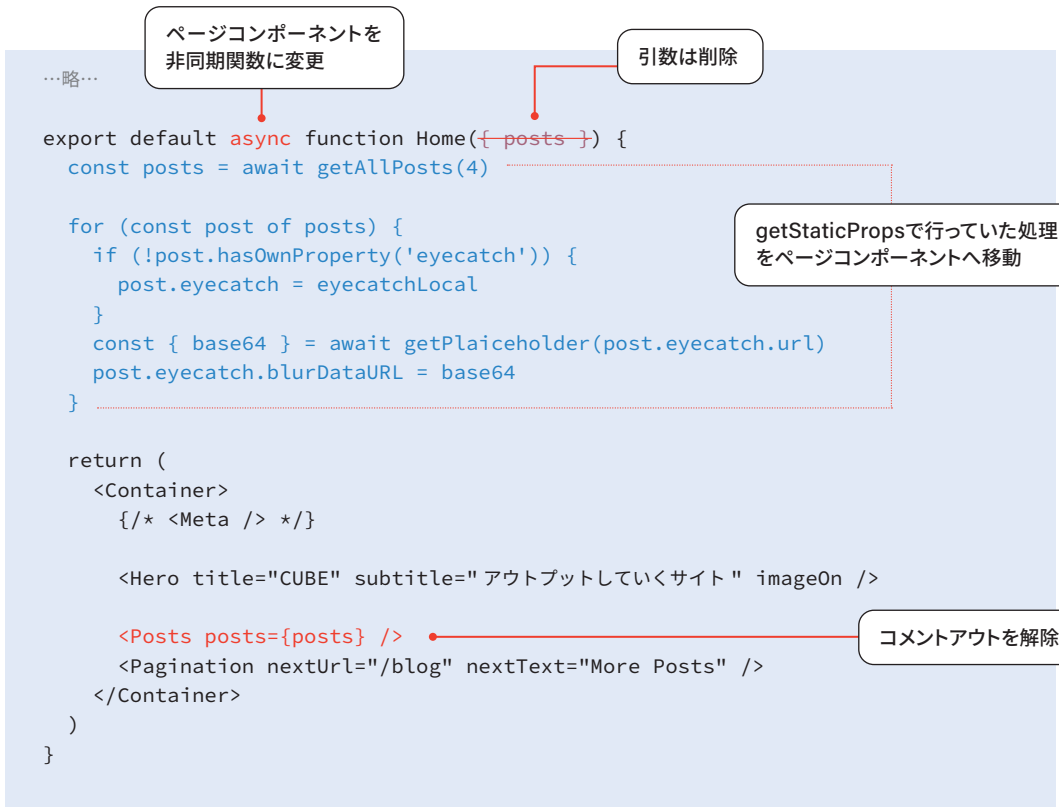
データを取得している部分（最新記事一覧）以外が表示されます

## ❖ getStaticProps からの移行

App Router では `getStaticProps`（`getServerSideProps`）は使えません。
Server Component からデータへ直接アクセスできるようになったため、ページコンポーネントを非同期関数として定義して直接データを取得します。

そこで、`Promise` を扱うためにページコンポーネントを非同期関数に変更し、`getStaticProps` で行っていた処理をページコンポーネントへ移します。`getStaticProps`（`getServerSideProps`）がなくなり、ページコンポーネントへ渡される引数 `posts` も必要ありませんので削除します。

以上の修正が完了したら、コメントアウトしていた <Posts /> コンポーネントをもとに戻します。これで元通りにページを表示できます。

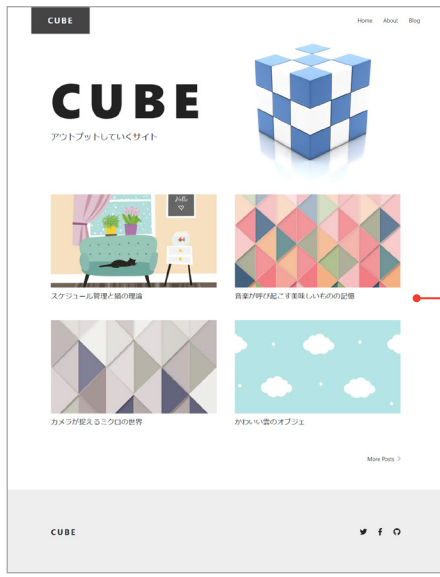



```
…略…

export default async function Home({ posts }) {
  const posts = await getAllPosts(4)

  for (const post of posts) {
    if (!post.hasOwnProperty('eyecatch')) {
      post.eyecatch = eyecatchLocal
    }
    const { base64 } = await getPlaiceholder(post.eyecatch.url)
    post.eyecatch.blurDataURL = base64
  }

  return (
    <Container>
      {/* <Meta /> */}

      <Hero title="CUBE" subtitle="アウトプットしていくサイト" imageOn />

      <Posts posts={posts} />
      <Pagination nextUrl="/blog" nextText="More Posts" />
    </Container>
  )
}
```

```
// export async function getStaticProps() {
//   const posts = await getAllPosts(4)

//   for (const post of posts) {
//     if (!post.hasOwnProperty('eyecatch')) {
//       post.eyecatch = eyecatchLocal
//     }
//     const { base64 } = await getPlaiceholder(post.eyecatch.url)
//     post.eyecatch.blurDataURL = base64
//   }

//   return {
//     props: {
//       posts: posts,
//     },
//   }
// }
```

getStaticPropsは削除

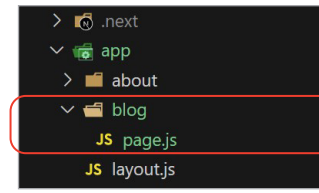
app/page.js




トップページ
/

データを取得している部分（最新記事一覧）が表示されます

## ❖ 記事一覧ページの移行




記事一覧ページも、トップページと同様の流れで移行できます。`pages/blog/index.js` を `app/blog/page.js` へ移動し、以下のように修正します。

`<Meta>` に関する部分はコメントアウトしておきます。

```
import { getAllPosts } from 'lib/api'
// import Meta from 'components/meta'
import Container from 'components/container'
import Hero from 'components/hero'
import Posts from 'components/posts'
import { getPlaiceholder } from 'plaiceholder'

// ローカルの代替アイキャッチ画像
import { eyecatchLocal } from 'lib/constants'

export default async function Blog({ posts }) {
  const posts = await getAllPosts()

  for (const post of posts) {
    if (!post.hasOwnProperty('eyecatch')) {
      post.eyecatch = eyecatchLocal
    }
    const { base64 } = await getPlaiceholder(post.eyecatch.url)
    post.eyecatch.blurDataURL = base64
  }

  return (
    <Container>
      {/* <Meta pageTitle="ブログ" pageDesc="ブログの記事一覧" /> */}

      <Hero title="Blog" subtitle="Recent Posts" />

      <Posts posts={posts} />
    </Container>
  )
}
```

ページコンポーネントを非同期関数に変更

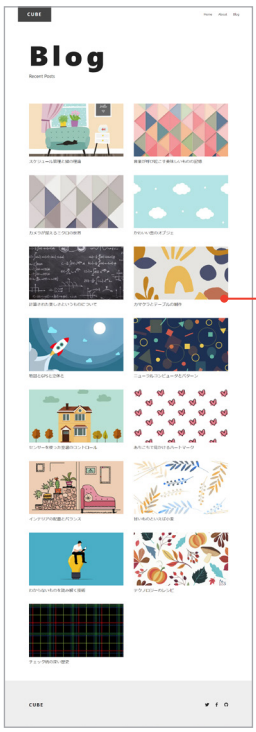
引数は削除

getStaticPropsで行っていた処理をページコンポーネントへ移動

```
export async function getStaticProps() {
  const posts = await getAllPosts()

  for (const post of posts) {
    if (!post.hasOwnProperty('eyecatch')) {
      post.eyecatch = eyecatchLocal
    }
    const { base64 } = await getPlaiceholder(post.eyecatch.url)
    post.eyecatch.blurDataURL = base64
  }

  return {
    props: {
      posts: posts,
    },
  }
}
```

getStaticPropsは削除

app/blog/page.js




記事一覧ページ
/blog

データを取得している部分（記事一覧）が表示されます

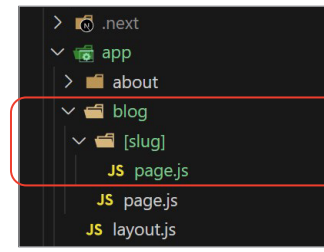
# 2.6 記事ページの移行

Next 13

Pages Router でダイナミックルート（Dynamic Routes）を使っていた記事ページを、App Router のダイナミックセグメント（Dynamic Segments）を使って移行します

## 記事ページの移動

記事ページを App Router へ移行するため、`pages/blog/[slug].js` を `app/blog/[slug]/page.js` へ移動します。




## getStaticPaths から generateStaticParams へ

App Router のダイナミックセグメントの環境では、`getStaticPaths` の代わりに `generateStaticParams` を使う必要があります。基本的な処理は変わりませんが、返り値としてセグメントパラメータの配列を返す必要があり、`getStaticPaths` で使えていた URL の形は使えません。

```
…略…
export async function getStaticPaths() {
  const allSlugs = await getAllSlugs()

  return {
    paths: allSlugs.map(({ slug }) => `/blog/${slug}`),
    fallback: false,
  }
}

export async function generateStaticParams() {
 const allSlugs = await getAllSlugs();

 return allSlugs.map(({ slug }) => {
  return { slug: slug };
 });
}
```

getStaticPathsを削除

generateStaticParams に置き換え

app/blog/[slug]/page.js

## getStaticProps からページコンポーネントへ

続いて、`getStaticProps` の処理をページコンポーネントへ移します。ページコンポーネントはそのまま使いたいので、`getStaticProps` から返していた右の構成に合わせます。
また、`generateStaticParams` からページコンポーネントに渡される値は、`props.params` となっています。このあたりを踏まえて以下のように修正します。もちろん、ページコンポーネントを非同期関数に変更するのも忘れないでください。

```
props: {
  title: post.title,
  publish: post.publishDate,
  content: post.content,
  eyecatch: eyecatch,
  categories: post.categories,
  description: description,
  prevPost: prevPost,
  nextPost: nextPost,
},
```

`<Meta>` に関する部分はコメントアウトしておきます。

```
import { getPostBySlug, getAllSlugs } from 'lib/api'
import { extractText } from 'lib/extract-text'
import { prevNextPost } from 'lib/prev-next-post'
// import Meta from 'components/meta'
import Container from 'components/container'
…略…

export default async function Post({ param }) {
  const slug = params.slug

  const post = await getPostBySlug(slug)

  const { title, publishDate: publish, content, categories } = post

  const description = extractText(content)

  const eyecatch = post.eyecatch ?? eyecatchLocal

  const { base64 } = await getPlaiceholder(eyecatch.url)
  eyecatch.blurDataURL = base64

  const allSlugs = await getAllSlugs()
  const [prevPost, nextPost] = prevNextPost(allSlugs, slug)
```

ページコンポーネントを非同期関数に変更

generateStaticParamsから渡された値を受け取り

getStaticPropsで行っていた処理をページコンポーネントへ移動（緑色の部分は修正箇所）

```
  return (
    <Container>
      {/* <Meta
        pageTitle={title}
        pageDesc={description}
        pageImg={eyecatch.url}
        pageImgW={eyecatch.width}
        pageImgH={eyecatch.height}
      /> */}

      <article>
        …略…
      </article>
    </Container>
  )
}

export async function generateStaticParams() {
  const allSlugs = await getAllSlugs()

  return allSlugs.map(({ slug }) => {
    return { slug: slug }
  })
}

export async function getStaticProps(context) {
  const slug = context.params.slug

  const post = await getPostBySlug(slug)

  const description = extractText(post.content)

  const eyecatch = post.eyecatch ?? eyecatchLocal

  const { base64 } = await getPlaiceholder(eyecatch.url)
  eyecatch.blurDataURL = base64

  const allSlugs = await getAllSlugs()
  const [prevPost, nextPost] = prevNextPost(allSlugs, slug)

  return {
    props: {
      title: post.title,
      publish: post.publishDate,
      content: post.content,
      eyecatch: eyecatch,
      categories: post.categories,
      description: description,
      prevPost: prevPost,
      nextPost: nextPost,
    },
  }
}
```

generateStaticParams

getStaticPropsは削除

app/blog/[slug]/page.js

これで、blog の記事ページが表示されます。


記事ページ
/blog/schedule


記事ページ
/blog/music


記事ページ
/blog/micro

ところが、`/blog/schedule2` といった存在しない記事ページにアクセスすると、データの存在しない slug をもとにデータを取得しようとするため、「Unhandled Runtime Error - Error: Cannot destructure property 'title' of 'post' as it is undefined.」と表示されます（開発サーバー）。これは、ページが存在しない場合に 404 ページを返す「`getStaticPaths` の `fallback: false`」に相当する設定をしていないためです。


存在しない記事ページ
/blog/schedule2

App Router でその設定に相当するのは、

```
export const dynamicParams = false
```

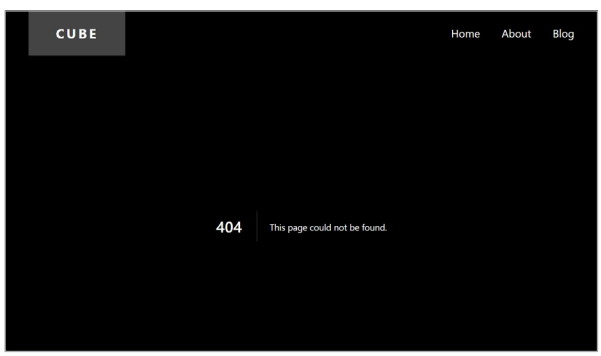
です。これを次のように追加します。

```
…略…
export default async function Post({ params }) {
  …略…
}

export const dynamicParams = false
export async function generateStaticParams() {
  const allSlugs = await getAllSlugs()

  return allSlugs.map(({ slug }) => {
    return { slug: slug }
  })
}
```

dynamicParamsの指定を追加

app/blog/[slug]/page.js

`/blog/schedule2` にアクセスすると、404 ページが表示されるようになります。




存在しない記事ページ
/blog/schedule2

## ❖ Route Segment Config

ここで追加した `export const dynamicParams = false` は、Route Segment Config の1つです。Route Segment Config を使うことで SSR でページを生成するようにしたり、ISR の設定などを行えます。Route Segment Config は layout.js や page.js、route.js で使えます。

### SSR（Server-side Rendering）

Pages Router の getServerSideProps を使った SSR に相当する処理へと切り替える場合には、次の設定を追加します。

```
export const dynamic = "force-dynamic"
```

### ISR（Incremental Static Regeneration）

ISR に相当する設定を行う場合には、次の設定を追加します。ここでは 60 秒に指定しています。

```
export const revalidate = 60
```

Route Segment Config では、Pages Router に比べてより細かな設定ができます。詳細に関しては、こちらのページを参照してください。

File Conventions: Route Segment Config
https://nextjs.org/docs/app/api-reference/file-conventions/route-segment-config

また、Route Segment Config がその名前の通りセグメントレベルでのコントロールなのに対して、コンポーネントレベルでコントロールできる fetch() API も用意されています。詳細は、こちらのページを参照してください。

Data Fetching
https://nextjs.org/docs/app/building-your-application/data-fetching

v2.5.0 以降の microcms-js-sdk では、fetch() API のオプションを利用できます。

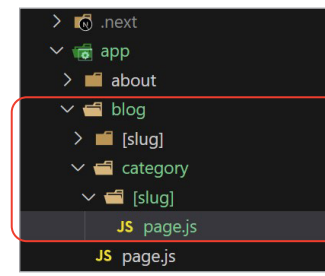
microcms-js-sdkでfetchリクエストオプションが追加できるようになりました
https://blog.microcms.io/microcms-js-sdk-2_5_0/

# 2.7 Next 13 カテゴリーインデックスページの移行

作業の流れは記事ページと同様です。

## カテゴリーインデックスページの移動

カテゴリーインデックスページを App Router へ移行するため、`pages/blog/category/[slug].js` を `app/blog/category/[slug]/page.js` へ移動します。




## getStaticPaths から generateStaticParams へ

`getStaticPaths` を元に、`generateStaticParams` を追加します。

```
…略…
export async function getStaticPaths() {
  const allCats = await getAllCategories()
  return {
    paths: allCats.map(({ slug }) => `/blog/category/${slug}`),
    fallback: false,
  }
}

export async function generateStaticParams() {
  const allCats = await getAllCategories()

  return allCats.map(({ slug }) => {
    return { slug: slug }
  })
}
```

getStaticPathsを削除

generateStaticParams に置き換え

app/blog/category/[slug]/page.js

## getStaticProps からページコンポーネントへ

`getStaticProps` の処理を、非同期関数にしたページコンポーネントへ移します。その際、ページコンポーネントをそのまま流用するため、`getStaticProps` から返していた右の構成に合わせます。

```
props: {
  name: cat.name,
  posts: posts,
},
```

また、記事ページと同様に存在しないページには 404 ページを返すように、右の指定を追加しておきます。

```
export const dynamicParams =
false
```

`<Meta>` に関する部分はコメントアウトしておきます。

```
import { getAllCategories, getAllPostsByCategory } from 'lib/api'
// import Meta from 'components/meta'
import Container from 'components/container'
…略…

export default async function Category({ params }) {
  const catSlug = params.slug

  const allCats = await getAllCategories()
  const cat = allCats.find(({ slug }) => slug === catSlug)
  const name = cat.name

  const posts = await getAllPostsByCategory(cat.id)

  for (const post of posts) {
    if (!post.hasOwnProperty('eyecatch')) {
      post.eyecatch = eyecatchLocal
    }
    const { base64 } = await getPlaiceholder(post.eyecatch.url)
    post.eyecatch.blurDataURL = base64
  }

  return (
    <Container>
      {/* <Meta pageTitle={name} pageDesc={`${name}に関する記事`} /> */}
      <PostHeader title={name} subtitle="Blog Category" />
```

ページコンポーネントを非同期関数に変更

generateStaticParamsから渡された値を受け取り

getStaticPropsで行っていた処理をページコンポーネントへ移動（緑色の部分は修正箇所）

```
      <Posts posts={posts} />
    </Container>
  )
}

export const dynamicParams = false
export async function generateStaticParams() {
  const allCats = await getAllCategories()

  return allCats.map(({ slug }) => {
    return { slug: slug }
  })
}

~~export async function getStaticProps(context) {~~
~~  const catSlug = context.params.slug~~

~~  const allCats = await getAllCategories()~~
~~  const cat = allCats.find(({ slug }) => slug === catSlug)~~

~~  const posts = await getAllPostsByCategory(cat.id)~~

~~  for (const post of posts) {~~
~~    if (!post.hasOwnProperty('eyecatch')) {~~
~~      post.eyecatch = eyecatchLocal~~
~~    }~~
~~    const { base64 } = await getPlaiceholder(post.eyecatch.url)~~
~~    post.eyecatch.blurDataURL = base64~~
~~  }~~

~~  return {~~
~~    props: {~~
~~      name: cat.name,~~
~~      posts: posts,~~
~~    },~~
~~  }~~
~~}~~
```

dynamicParamsの指定を追加

generateStaticParams

getStaticPropsは削除

app/blog/category/[slug]/page.js


カテゴリーページ
/blog/category/fun

カテゴリーページが表示されることを確認します。以上で、各ページのApp Router への移行は完了です。

# 2.8 Next 13 Config-based Metadata によるメタデータの構成

各ページが移行できたところで、メタデータを考えます。P.24 のように Config-based Metadata と File-based Metadata がありますが、Config-based Metadata を基本にして進めます。そこで、metadata オブジェクトの構成を検討します。

metadata オブジェクトで扱えるパラメータと `components/meta.js` をもとに、metadata オブジェクトの構成を考えると、ベースは以下のような構成になります。

```
export const metadata = {
  metadataBase: new URL(siteUrl),
  alternates: {
    canonical: './',
  },
  viewport: {
    width: 'device-width',
    initialScale: 1,
    maximumScale: 1,
  },
  title: {
    template: `%s | ${siteTitle}`,
    default: siteTitle,
  },
  description: siteDesc,
  icons: {
    icon: siteIcon,
    apple: siteIcon,
  },

  openGraph: {
    title: siteTitle,
    description: siteDesc,
    url: siteUrl,
    siteName: siteTitle,
    images: [
      {
        url: siteImg.src,
        width: siteImg.width,
        height: siteImg.height,
```

ページのURL

ビューポートの設定（省略可）

ページのタイトル

ページの説明

サイトのアイコン

OGPの設定

```
      },
    ],
    locale: siteLocale,
    type: siteType,
  },
  twitter: {
    card: 'summary_large_image',
    title: siteTitle,
    description: siteDesc,
    images: [siteImg.src],
  },
}
```

Twitterの設定

そして、各ページの `<Meta />` を確認します。

### app/pages.js（トップページ）

```
<Meta />
```

### app/about/page.js（アバウトページ）

```
<Meta
 pageTitle="アバウト"
 pageDesc="About development activities"
 pageImg={eyecatch.src}
 pageImgW={eyecatch.width}
 pageImgH={eyecatch.height}
 />
```

### app/blog/page.js（記事一覧ページ）

```
<Meta pageTitle="ブログ" pageDesc="ブログの記事一覧" />
```

### app/blog/[slug]/page.js（記事ページ）

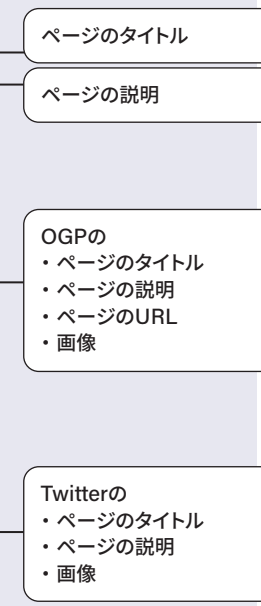
```
<Meta
 pageTitle={title}
 pageDesc={description}
 pageImg={eyecatch.url}
 pageImgW={eyecatch.width}
 pageImgH={eyecatch.height}
 />
```

### app/blog/category/[slug]/page.js（カテゴリーページ）

```
<Meta pageTitle={name} pageDesc={`${name}に関する記事`} />
```

これをもとに上書きしたいプロパティをまとめると、次のような感じになります。

```
export const metadata = {
  title: pageTitle,
  description: pageDesc,
  openGraph: {
    title: pageTitle,
    description: pageDesc,
    url: pageUrl,
    images: [
      {
        url: pageImg,
        width: pageImgW,
        height: pageImgH,
      },
    ],
  },
  twitter: {
    title: pageTitle,
    description: pageDesc,
    images: [pageImg],
  },
}
```




つまり、ベースをこのオブジェクトで上書きすればよいわけです。ただし、metadata オブジェクトの処理には次のような注意点があります。

## ❖ 注意点

### 浅いマージ

次のページのサンプルにある通り、metadata オブジェクトのマージは浅いマージになります。そのため、必要なプロパティだけを上書きするためには、オブジェクトを事前に展開しておく必要があります。

Metadata - Merging
https://nextjs.org/docs/app/building-your-application/optimizing/metadata#merging

### ページのURL

次の設定で、現在のページの URL を <link rel="canonical" href=" 〜 " /> の形で出力してくれます。

```
export const metadata = {
  metadataBase: new URL(siteUrl),
  alternates: {
    canonical: './',
  },
}
```

```
<link rel="canonical" href="https://example.com/blog/music" />
```

ただし、この URL は流用できません。openGraph や twitter で必要なページの URL は別途用意する必要があります。

### title のテンプレート

title のテンプレートを `layout.js` に指定することで、`page.js` で指定した title と組み合わせることができます。たとえば、以下のようにテンプレートとタイトルを指定すれば、各ページのタイトルは `pageTitle | siteTitle`（ページタイトル | サイト名）という形で出力できます。

#### layout.js

```
export const metadata = {
  title: {
    template: `%s | ${siteTitle}`,
    default: siteTitle,
  },
}
```

#### page.js

```
export const metadata = {
  title: pageTitle,
}
```

```
<title>pageTitle | siteTitle</title>
```

ただし、`page.js` で指定したタイトルと組み合わせますので、テンプレートは `layout.js` でしか指定できません。また、openGraph や twitter に流用はできません。openGraph や twitter でも同じような構成にする場合は、あらかじめ用意する必要があります。

このあたりを踏まえて、次のステップで metadata オブジェクトを設定していきます。

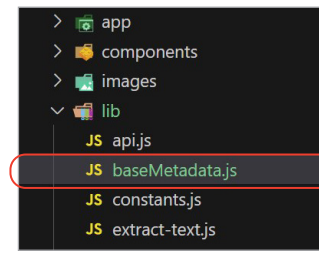
# 2.9 metadata オブジェクトの設定

Next 13

metadata オブジェクトを設定していきます。

## ❖ lib/baseMetadata.js の作成

まず、`lib/baseMetadata.js` を作成し、ベースとなる metadata オブジェクトを用意します。プロパティをピンポイントで上書きしやすいように、openGraph と twitter に関する設定は別のオブジェクトに分けて以下のような構成にしています。




```
// サイトに関する情報
import { siteMeta } from 'lib/constants'
const {
  siteTitle,
  siteDesc,
  siteLang,
  siteUrl,
  siteLocale,
  siteType,
  siteIcon,
} = siteMeta

// 汎用 OGP 画像
import siteImg from 'images/ogp.jpg'

// ベースとなる設定
export const baseMetadata = {
  metadataBase: new URL(siteUrl),
  alternates: {
    canonical: './',
  },
```

サイトに関する情報をlib/constantsから取得

- siteTitle …… サイト名
- siteDesc…… サイトの説明
- siteUrl…… サイトのURL
- siteLang…… 言語
- siteLocale … ロケール
- siteType …… コンテンツの種類
- siteIcon …… アイコン画像

汎用的に使用するOGP画像（meta.jsで指定したもの）を指定

- siteImg …… 汎用OGP画像

ページのURL：
サイトのURL（siteUrl）を指定

```
  viewport: {
    width: 'device-width',
    initialScale: 1,
    maximumScale: 1,
  },
  title: {
    template: `%s | ${siteTitle}`,
    default: siteTitle,
  },
  description: siteDesc,
  icons: {
    icon: siteIcon,
    apple: siteIcon,
  },
}

// openGraph に関する設定
export const openGraphMetadata = {
  title: siteTitle,
  description: siteDesc,
  url: siteUrl,
  siteName: siteTitle,
  images: [
    {
      url: siteImg.src,
      width: siteImg.width,
      height: siteImg.height,
    },
  ],
  locale: siteLocale,
  type: siteType,
}

// twitter に関する設定
export const twitterMetadata = {
  card: 'summary_large_image',
  title: siteTitle,
  description: siteDesc,
  images: [siteImg.src],
}
```

ビューポートの設定

ページのタイトル：
ページタイトルの指定がある場合はテンプレート（P.48）、ない場合はサイト名を使用するように指定

ページの説明：
サイトの説明（siteDesc）を指定

サイトのアイコン：
サイトのアイコン（siteIcon）を指定

OGPの設定

- ページのタイトル：
  サイト名（siteTitle）を指定
- ページの説明：
  サイトの説明（siteDesc）を指定
- ページのURL：
  サイトのURL（siteUrl）を指定
- 画像：
  汎用OGP画像（siteImg）のURL、横幅、高さを指定
- ロケール：
  Localeを指定
- コンテンツの種類：
  siteTypeを指定

Twitterの設定

- Twitterカードの種類：
  summary_large_imageに指定
- ページのタイトル：
  サイト名（siteTitle）を指定
- ページの説明：
  サイトの説明（siteDesc）を指定
- 画像：
  汎用OGP画像（siteImg）を指定

lib/baseMetadata.js

## ❖ Root Layout（app/layout.js）の設定

`lib/baseMetadata.js` を使って、ベースとなるメタデータの設定を Root Layout コンポーネント `app/layout.js` に追加します。これで、各ページにベースとなるメタデータが出力されるようになります。

```
import {
  baseMetadata,
  openGraphMetadata,
  twitterMetadata,
} from 'lib/baseMetadata'

import 'styles/globals.css'
…略…
export default function RootLayout({ children }) {
  return (
    …略…
  )
}

export const metadata = {
  ...baseMetadata,
  openGraph: {
    ...openGraphMetadata,
  },
  twitter: {
    ...twitterMetadata,
  },
}
```

app/layout.js

これで、各ページにベースとなるメタデータが出力されるようになりますので、必要に応じて上書きする設定を追加していきます。

## トップページ（app/page.js）のメタデータ


Root Layout で設定したベースのメタデータで問題ないため、トップページへの設定は必要ありません。メタデータの出力は次のようになっています。

コメントアウトした `<Meta>` に関する部分は削除しておきます。

ベースとなる設定（baseMetadata）の出力

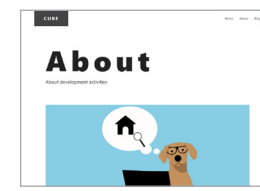
OGPの設定（openGraphMetadata）の出力

```
<title>CUBE</title>
<meta name="description" content="アウトプットしていくサイト" />
<meta name="viewport"
 content="width=device-width, initial-scale=1, maximum-scale=1" />
<link rel="canonical" href="https://*********/" />
<link rel="icon" href="/favicon.png" />
<link rel="apple-touch-icon" href="/favicon.png" />
<meta property="og:title" content="CUBE" />
<meta property="og:description" content="アウトプットしていくサイト" />
<meta property="og:url" content="https://*********/" />
<meta property="og:site_name" content="CUBE" />
<meta property="og:locale" content="ja_JP" />
<meta property="og:image"
 content="https://*********/_next/static/media/ogp.a13e6712.jpg" />
<meta property="og:image:width" content="1200" />
<meta property="og:image:height" content="630" />
<meta property="og:type" content="website" />
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:title" content="CUBE" />
<meta name="twitter:description" content="アウトプットしていくサイト" />
<meta name="twitter:image"
 content="https://*********/_next/static/media/ogp.a13e6712.jpg" />
```

Twitterの設定（twitterMetadata）の出力

## アバウトページ（app/about/page.js）のメタデータ

アバウトページのメタデータを設定します。コメントアウトした `<Meta />` に関連する部分を参考に設定すると、次のような構成になります。




設定ができたら `<Meta>` に関する部分は削除しておきます。

```
// import Meta from 'components/meta'
import Container from 'components/container'
…略…
import eyecatch from 'images/about.jpg'

// サイトに関する情報
import { siteMeta } from 'lib/constants'
const { siteTitle, siteUrl } = siteMeta

// ベースのメタデータ
import { openGraphMetadata, twitterMetadata } from 'lib/baseMetadata'

export default function About() {
  return (
    <Container>
      {/* <Meta
        pageTitle="アバウト"
        pageDesc="About development activities"
        pageImg={eyecatch.src}
        pageImgW={eyecatch.width}
        pageImgH={eyecatch.height}
      /> */}

      …略…
    </Container>
  )
}

// メタデータ
const pageTitle = 'アバウト'
const pageDesc = 'About development activities'
const ogpTitle = `${pageTitle} | ${siteTitle}`
const ogpUrl = new URL('/about', siteUrl).toString()

export const metadata = {
  title: pageTitle,
  description: pageDesc,

  openGraph: {
    ...openGraphMetadata,
    title: ogpTitle,
    description: pageDesc,
    url: ogpUrl,
    images: [
      {
        url: eyecatch.src,
        width: eyecatch.width,
        height: eyecatch.height,
```

サイトに関する情報とベースのメタデータを取得

アバウトページの情報を指定

上書き用のmetadataオブジェクトを用意

ページのタイトルと説明を指定

OGPの設定を指定

```
      },
    ],
  },
  twitter: {
    ...twitterMetadata,
    title: ogpTitle,
    description: pageDesc,
    images: [eyecatch.src],
  },
}
```

Twitterの設定を指定

app/about/page.js

出力を確認すると、ベースのメタデータが次のように上書きされています。以上で、アバウトページの設定は完了です。

```
<title>アバウト | CUBE</title>
<meta name="description" content="About development activities" />
<meta name="viewport"
 content="width=device-width, initial-scale=1, maximum-scale=1" />
<link rel="canonical" href="https://*********/about" />
<link rel="icon" href="/favicon.png" />
<link rel="apple-touch-icon" href="/favicon.png" />
<meta property="og:title" content="アバウト | CUBE" />
<meta property="og:description" content="About development activities" />
<meta property="og:url" content="https://*********/about" />
<meta property="og:site_name" content="CUBE" />
<meta property="og:locale" content="ja_JP" />
<meta property="og:image"
 content="https://*********/_next/static/media/about.d19731d6.jpg" />
<meta property="og:image:width" content="1920" />
<meta property="og:image:height" content="960" />
<meta property="og:type" content="website" />
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:title" content="アバウト | CUBE" />
<meta name="twitter:description" content="About development activities" />
<meta name="twitter:image"
 content="https://*********/_next/static/media/about.d19731d6.jpg" />
```

## ❖ 記事一覧ページ（app/blog/page.js）のメタデータ

記事一覧ページのメタデータを設定します。構成としては、アバウトページのメタデータと大きな違いはありません。


設定ができたら `<Meta>` に関する部分は削除しておきます。

```
import { getAllPosts } from 'lib/api'
// import Meta from 'components/meta'
import Container from 'components/container'
…略…

// サイトに関する情報
import { siteMeta } from 'lib/constants'
const { siteTitle, siteUrl } = siteMeta

// ベースのメタデータ
import { openGraphMetadata, twitterMetadata } from 'lib/baseMetadata'

export default async function Blog() {
  …略…
  return (
    <Container>
      {/* <Meta pageTitle="ブログ" pageDesc="ブログの記事一覧" /> */}
      …略…
    </Container>
  )
}

// メタデータ
const pageTitle = 'ブログ'
const pageDesc = 'ブログの記事一覧'
const ogpTitle = `${pageTitle} | ${siteTitle}`
const ogpUrl = new URL('/blog', siteUrl).toString()

export const metadata = {
  title: pageTitle,
  description: pageDesc,

  openGraph: {
    ...openGraphMetadata,
    title: ogpTitle,
    description: pageDesc,
    url: ogpUrl,
  },
```

サイトに関する情報とベースのメタデータを取得

記事一覧ページの情報を指定

上書き用のmetadataオブジェクトを用意

ページのタイトルと説明を指定

OGPの設定を指定

```
  twitter: {
    ...twitterMetadata,
    title: ogpTitle,
    description: pageDesc,
  },
}
```

Twitterの設定を指定

app/blog/page.js

出力を確認すると、ベースのメタデータが次のように上書きされています。以上で、記事一覧ページの設定は完了です。

```
<title>ブログ | CUBE</title>
<meta name="description" content="ブログの記事一覧" />
<meta name="viewport"
 content="width=device-width, initial-scale=1, maximum-scale=1" />
<link rel="canonical" href="https://*********/blog" />
<link rel="icon" href="/favicon.png" />
<link rel="apple-touch-icon" href="/favicon.png" />
<meta property="og:title" content="ブログ | CUBE" />
<meta property="og:description" content="ブログの記事一覧" />
<meta property="og:url" content="https://*********/blog" />
<meta property="og:site_name" content="CUBE" />
<meta property="og:locale" content="ja_JP" />
<meta property="og:image"
 content="https://*********/_next/static/media/ogp.a13e6712.jpg" />
<meta property="og:image:width" content="1200" />
<meta property="og:image:height" content="630" />
<meta property="og:type" content="website" />
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:title" content="ブログ | CUBE" />
<meta name="twitter:description" content="ブログの記事一覧" />
<meta name="twitter:image"
 content="https://*********/_next/static/media/ogp.a13e6712.jpg" />
```

## ❖ 記事ページ（app/blog/[slug]/page.js）のメタデータ：Dynamic Metadata

記事ページはダイナミックセグメントを使っているため、メタデータの作成にも外部のデータを必要とします。そのため、ページコンポーネントと同じように `generateStaticParams` からの `props.params` を受け取ることができる `generateMetadata` を使って、メタデータを export します。


設定ができたら `<Meta>` に関する部分は削除しておきます。

```
import { getPostBySlug, getAllSlugs } from 'lib/api'
import { extractText } from 'lib/extract-text'
import { prevNextPost } from 'lib/prev-next-post'
// import Meta from 'components/meta'
…略…

// サイトに関する情報
import { siteMeta } from 'lib/constants'
const { siteTitle, siteUrl } = siteMeta

// ベースのメタデータ
import { openGraphMetadata, twitterMetadata } from 'lib/baseMetadata'

export default async function Post({ params }) {
  …略…
  return (
    <Container>
      {/* <Meta
        pageTitle={title}
        pageDesc={description}
        pageImg={eyecatch.url}
        pageImgW={eyecatch.width}
        pageImgH={eyecatch.height}
      /> */}
      …略…
    </Container>
  )
}

export const dynamicParams = false
```

サイトに関する情報とベースのメタデータを取得

```
export async function generateStaticParams() {
  …略…
}

// メタデータ
export async function generateMetadata({ params }) {
  const slug = params.slug
  const post = await getPostBySlug(slug)
  const { title: pageTitle, publishDate: publish, content, categories } = post

  const pageDesc = extractText(content)
  const eyecatch = post.eyecatch ?? eyecatchLocal

  const ogpTitle = `${pageTitle} | ${siteTitle}`
  const ogpUrl = new URL(`/blog/${slug}`, siteUrl).toString()

  const metadata = {
    title: pageTitle,
    description: pageDesc,

    openGraph: {
      ...openGraphMetadata,
      title: ogpTitle,
      description: pageDesc,
      url: ogpUrl,
      images: [
        {
          url: eyecatch.url,
          width: eyecatch.width,
          height: eyecatch.height,
        },
      ],
    },
    twitter: {
      ...twitterMetadata,
      title: ogpTitle,
      description: pageDesc,
      images: [eyecatch.url],
    },
  }

  return metadata
}
```

generateStaticParamsから渡された値を受け取り

記事ページの情報を指定

上書き用のmetadataオブジェクトを用意

ページのタイトルと説明を指定

OGPの設定を指定

Twitterの設定を指定

外部データ(microCMS)に合わせているため、.urlになっていることに注意してください

app/blog/[slug]/page.js

出力を確認すると、ベースのメタデータが次のように上書きされています。以上で、記事ページの設定は完了です。

```
<title>スケジュール管理と猫の理論 | CUBE</title>
<meta name="description" content="何でもすぐに忘れてしまうので、予定を忘れないようにスケジュール管理手帳で予定を管理しています。でも、本当はスケジュールをスケジュールとして正しく認識できていない…" />
<meta name="viewport"
 content="width=device-width, initial-scale=1, maximum-scale=1" />
<link rel="canonical" href="https://*********/blog/schedule" />
<link rel="icon" href="/favicon.png" />
<link rel="apple-touch-icon" href="/favicon.png" />
<meta property="og:title" content="スケジュール管理と猫の理論 | CUBE" />
<meta property="og:description" content="何でもすぐに忘れてしまうので、予定を忘れないようにスケジュール管理手帳で予定を管理しています。でも、本当はスケジュールをスケジュールとして正しく認識できていない…" />
<meta property="og:url" content="https://*********/blog/schedule" />
<meta property="og:site_name" content="CUBE" />
<meta property="og:locale" content="ja_JP" />
<meta property="og:image"
 content="https://images.microcms-assets.io/assets/…/schedule.jpg" />
<meta property="og:image:width" content="1920" />
<meta property="og:image:height" content="1280" />
<meta property="og:type" content="website" />
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:title" content="スケジュール管理と猫の理論 | CUBE" />
<meta name="twitter:description" content="何でもすぐに忘れてしまうので、予定を忘れないようにスケジュール管理手帳で予定を管理しています。でも、本当はスケジュールをスケジュールとして正しく認識できていない…" />
<meta name="twitter:image"
 content="https://images.microcms-assets.io/assets/…/schedule.jpg" />
```

## ❖ カテゴリーページ（app/blog/category/[slug]/page.js）のメタデータ

カテゴリーページのメタデータもダイナミックセグメントを使っていますが、記事ページのメタデータの設定と大きな違いはありません。


設定ができたら `<Meta>` に関する部分は削除しておきます。

```
import { getAllCategories, getAllPostsByCategory } from 'lib/api'
// import Meta from 'components/meta'
…略…

// サイトに関する情報
import { siteMeta } from 'lib/constants'
const { siteTitle, siteUrl } = siteMeta

// ベースのメタデータ
import { openGraphMetadata, twitterMetadata } from 'lib/baseMetadata'

export default async function Category({ params }) {
  …略…
  return (
    <Container>
      {/* <Meta pageTitle={name} pageDesc={`${name}に関する記事`} /> */}
      …略…
    </Container>
  )
}

export const dynamicParams = false
export async function generateStaticParams() {
  …略…
}

// メタデータ
export async function generateMetadata({ params }) {
  const catSlug = params.slug

  const allCats = await getAllCategories()
  const cat = allCats.find(({ slug }) => slug === catSlug)
```

サイトに関する情報とベースのメタデータを取得

generateStaticParamsから渡された値を受け取り

記事ページの情報を指定

```
  const pageTitle = cat.name
  const pageDesc = '${pageTitle}に関する記事'
  const ogpTitle = `${pageTitle} | ${siteTitle}`
  const ogpUrl = new URL('/blog/category/${catSlug}', siteUrl).toString()


  const metadata = {
    title: pageTitle,
    description: pageDesc,

    openGraph: {
      ...openGraphMetadata,
      title: ogpTitle,
      description: pageDesc,
      url: ogpUrl,
    },
    twitter: {
      ...twitterMetadata,
      title: ogpTitle,
      description: pageDesc,
    },
  }

  return metadata
}
```



app/blog/category/[slug]/page.js

これで、カテゴリーページには次のようにメタデータが出力されます。

```
<title>楽しいものいろいろ | CUBE</title>
<meta name="description" content="楽しいものいろいろに関する記事" />
<meta name="viewport"
 content="width=device-width, initial-scale=1, maximum-scale=1" />
<link rel="canonical" href="https://*********/blog/category/fun" />
<link rel="icon" href="/favicon.png" />
<link rel="apple-touch-icon" href="/favicon.png" />
<meta property="og:title" content="楽しいものいろいろ | CUBE" />
<meta property="og:description" content="楽しいものいろいろに関する記事" />
<meta property="og:url" content="https://*********/blog/category/fun" />
<meta property="og:site_name" content="CUBE" />
<meta property="og:locale" content="ja_JP" />
<meta property="og:image"
 content="https://*********/_next/static/media/ogp.a13e6712.jpg" />
<meta property="og:image:width" content="1200" />
<meta property="og:image:height" content="630" />
<meta property="og:type" content="website" />
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:title" content="楽しいものいろいろ | CUBE" />
<meta name="twitter:description" content="ブ楽しいものいろいろに関する記事" />
<meta name="twitter:image"
 content="https://*********/_next/static/media/ogp.a13e6712.jpg" />
```

• • •

以上で、メタデータの設定は完了です。

今回は metadata オブジェクトのマージを意識した構成にしていますが、`conponents/meta.js` をもとに、metadata オブジェクトを生成する関数を用意しても問題はありません。

# 2.10 File-based Metadata によるメタデータ

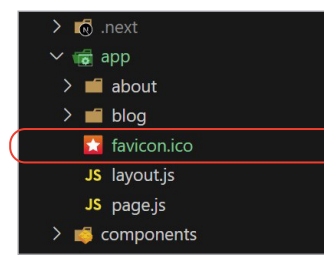
Next 13

ここまでの設定でも問題はありませんが、サイトのアイコン（ファビコン）の設定を File-based Metadata に切り替えてみます。そのため、`lib/baseMetadata.js` から、アイコンに関する設定をコメントアウトします。

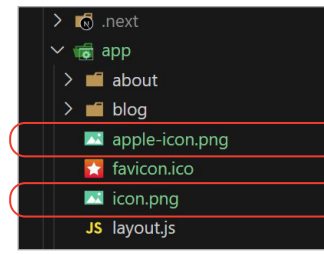
```
// ベースとなる設定
export const baseMetadata = {
  metadataBase: new URL(siteUrl),
  alternates: {
    canonical: './',
  },
  viewport: {
    width: 'device-width',
    initialScale: 1,
    maximumScale: 1,
  },
  title: {
    template: `%s | ${siteTitle}`,
    default: siteTitle,
  },
  description: siteDesc,
  // icons: {
  //   icon: siteIcon,
  //   apple: siteIcon,
  // },
}
```

lib/baseMetadata.js

続いて、File-based Metadata を設定していきます。まず、`public/favicon.ico` を `app/favicon.ico` へコピーします。



さらに、`public/favicon.png` を `app/icon.png` と `app/apple-icon.png` にコピーします。
これでそれぞれのファイルが認識され、次のようにメタデータが出力されます。




```
<link rel="icon" href="/favicon.ico" type="image/x-icon" sizes="any">
<link rel="icon" href="/icon.png?19d0d2e0ed959fbc" type="image/png"
sizes="192x192">
<link rel="apple-touch-icon" href="/apple-icon.png?19d0d2e0ed959fbc" type="image/
png" sizes="192x192">
```

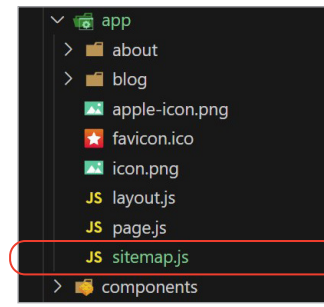
以上で、メタデータの設定は完了です。

ここでは `public/favicon.png` を流用していますが、対応したフォーマットでそれぞれ用意することもできます。

# 2.11 サイトマップ

Next 13

サイトマップは sitemap.xml を `app/sitemap.xml` として用意することもできますし、`app/sitemap.js` を用意して sitemap.xml を生成することもできます。




ここでは、`app/sitemap.js` を用意してみます。各ページの URL を集めて以下の構成のオブジェクトにし、配列としてまとめて返すだけです。

```
{
    url: "https://xxx.xxx/xxxxxxx",
    lastModified: new Date(),
}
```

コードは次のような感じになります。

```
// サイトに関する情報
import { siteMeta } from 'lib/constants'
const { siteUrl } = siteMeta

import { getAllSlugs, getAllCategories } from 'lib/api'


export default async function sitemap() {
  // 各記事の URL
  const posts = await getAllSlugs()
  const postFields = posts.map((post) => {
    return {
      url: new URL(`/blog/${post.slug}`, siteUrl).toString(),
      lastModified: new Date(),
    }
  })
```

```
  // 各カテゴリーインデックスの URL
  const cats = await getAllCategories()
  const catFields = cats.map((cat) => {
    return {
      url: new URL(`/blog/category/${cat.slug}`, siteUrl).toString(),
      lastModified: new Date(),
    }
  })

  return [                                                    まとめて返す配列
    {
      url: new URL(siteUrl).toString(),
      lastModified: new Date(),
    },
    {
      url: new URL('/about', siteUrl).toString(),
      lastModified: new Date(),
    },
    ...postFields,
    ...catFields,
  ]
}
```

app/sitemap.js

これで、次のようにサイトマップが生成されます。

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://*********/</loc>
    <lastmod>2023-06-27T03:28:13.199Z</lastmod>
  </url>
  <url>
    <loc>https://*********/about</loc>
    <lastmod>2023-06-27T03:28:13.199Z</lastmod>
  </url>
  <url>
    <loc>https://*********/blog/schedule</loc>
    <lastmod>2023-06-27T03:28:13.197Z</lastmod>
  </url>
  …略…
</urlset>
```

/sitemap.xml

# 2.12 Google アナリティクスの移行

Next 13

Google アナリティクスの設定を App Router へ移行していきます。

まず、Google アナリティクスの既存の設定を確認します。Pages Router では `pages/_app.js` で設定しています。

```
import { useEffect } from 'react'
import { useRouter } from 'next/router'
import 'styles/globals.css'
import Layout from 'components/layout'
import Script from 'next/script'
import * as gtag from 'lib/gtag'

// Font Awesome の設定
import '@fortawesome/fontawesome-svg-core/styles.css'
import { config } from '@fortawesome/fontawesome-svg-core'
config.autoAddCss = false

function MyApp({ Component, pageProps }) {
  const router = useRouter()
  useEffect(() => {
    const handleRouteChange = (url) => {
      gtag.pageview(url)
    }
    router.events.on('routeChangeComplete', handleRouteChange)
    return () => {
      router.events.off('routeChangeComplete', handleRouteChange)
    }
  }, [router.events])

  return (
    <>
      <Script
        strategy="afterInteractive"
        src={`https://www.googletagmanager.com/gtag/js?id=${gtag.GA_MEASUREMENT_ID}`}
      />
      <Script
        id="gtag-init"
        strategy="afterInteractive"
```

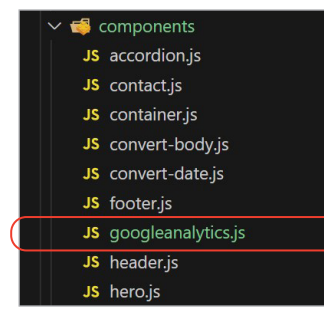
```
          dangerouslySetInnerHTML={{
            __html: `
              window.dataLayer = window.dataLayer || [];
              function gtag(){dataLayer.push(arguments);}
              gtag('js', new Date());

              gtag('config', '${gtag.GA_MEASUREMENT_ID}');
            `,
          }}
        />

        <Layout>
          <Component {...pageProps} />
        </Layout>
      </>
    )
}

export default MyApp
```

pages/_app.js

この設定をそのまま `app/layout.js` へ持っていきたいところですが、useEffect を使ったクライアントサイドの処理なため、そのままでは持っていけません。そのため、Google アナリティクスの設定をクライアントコンポーネントとして抜き出し、`app/layout.js` へ追加します。

ここでは `components/googleanalytics.js` を作成し、`pages/_app.js` から設定をコピーしてきます。

`'use client'` を追加して、クライアントコンポーネントにすることを忘れないように注意してください。



```
'use client'

import { useEffect } from 'react'
import { useRouter } from 'next/router'
import Script from 'next/script'
import * as gtag from 'lib/gtag'

function GoogleAnalytics() {
  const router = useRouter()
  useEffect(() => {
    const handleRouteChange = (url) => {
      gtag.pageview(url)
    }
    router.events.on('routeChangeComplete', handleRouteChange)
    return () => {
      router.events.off('routeChangeComplete', handleRouteChange)
    }
  }, [router.events])

  return (
    <>
      <Script
        strategy="afterInteractive"
        src={`https://www.googletagmanager.com/gtag/js?id=${gtag.GA_MEASUREMENT_ID}`}
      />
      <Script
        id="gtag-init"
        strategy="afterInteractive"
        dangerouslySetInnerHTML={{
          __html: `
            window.dataLayer = window.dataLayer || [];
            function gtag(){dataLayer.push(arguments);}
            gtag('js', new Date());

            gtag('config', '${gtag.GA_MEASUREMENT_ID}');
          `,
        }}
      />
    </>
  )
}

export default GoogleAnalytics
```

components/googleanalytics.js

そして、`app/layout.js` にこのコンポーネントを追加します。

```
…略…
import GoogleAnalytics from 'components/googleanalytics'

// Font Awesome の設定
import '@fortawesome/fontawesome-svg-core/styles.css'
import { config } from '@fortawesome/fontawesome-svg-core'
config.autoAddCss = false

export default function RootLayout({ children }) {
  return (
    <html lang={siteLang}>
      <body>
        <GoogleAnalytics />
        <Layout>{children}</Layout>
      </body>
    </html>
  )
}
```

app/layout.js

ところが、エラーが表示されます。

```
- error node_modules/next/dist/client/router.js (146:14) @ useRouter
- error Error: NextRouter was not mounted. https://nextjs.org/docs/messages/
next-router-not-mounted
    at GoogleAnalytics (./components/googleanalytics.js:20:74)
null
```

リンク先を確認すると、次のような内容のエラーであることがわかります。

> コンポーネントが Next.js アプリケーションの外で useRouter を使用したり、Next.js アプリケーションの外でレンダリングされたりしました。useRouter フックを使用するコンポーネントのユニットテストを行う際に、Next.js のコンテキストが設定されていないため、このような現象が発生することがあります。

Next.js の中でこのエラーということで、ちょっと混乱します。しかし、原因はシンプルなもので、App Router では `next/router` からインポートした useRouter は使えず、`next/navigation` からインポートした useRouter を使う必要があるためです。

さらに、`next/navigation` の useRouter では、router.events に関する機能はバッサリとなくなっており、その代替手段が紹介されています。

useRouter - Router Events
https://nextjs.org/docs/app/api-reference/functions/use-router#router-events

そのため、`next/navigation` の usePathname と useSearchParams を使って、`components/googleanalytics.js` を以下のように書き換えます。

```
'use client'

import { useEffect } from 'react'
import { usePathname, useSearchParams } from 'next/navigation'
import Script from 'next/script'
import * as gtag from 'lib/gtag'

function GoogleAnalytics() {
  const pathname = usePathname()
  const searchParams = useSearchParams()

  useEffect(() => {
    const url = pathname + searchParams.toString()
    gtag.pageview(url)
  }, [pathname, searchParams])

  return (
    …略…
  )
}

export default GoogleAnalytics
```

components/googleanalytics.js

そして、`app/layout.js` には `<Suspense>` を追加します。

```
import { Suspense } from 'react'
…略…

export default function RootLayout({ children }) {
  return (
    <html lang={siteLang}>
      <body>
        <Suspense>
          <GoogleAnalytics />
        </Suspense>
        <Layout>{children}</Layout>
      </body>
    </html>
  )
}
```

app/layout.js

この `<Suspense>` は、useSearchParams に対するものです。`<Suspense>` を追加しない場合、生成するすべてのページに対して、次のようなワーニングが出ます。

```
- warn Entire page /blog/[slug] deopted into client-side rendering. https://
nextjs.org/docs/messages/deopted-into-client-rendering /blog/[slug]
```

ワーニングで表示されるリンクの先を確認すると、

> 静的レンダリング中にそれを捕らえるサスペンス境界がなかったため、ページ全体が useSearchParams によってクライアント側レンダリングに切り替わりました。

ということで、useSearchParams を原因として、ページコンポーネントがクライアントコンポーネントとして扱われていることがわかります。その対策として、useSearchParams を含むコンポーネントを `<Suspense>` で囲む必要があるということです。

useSearchParams - Static Rendering
https://nextjs.org/docs/app/api-reference/functions/use-search-params#static-rendering

• • •

ここでは一般的な設定方法を選択しましたが、本書のサンプルのようなブログの場合、useSearchParams を使う URL クエリパラメータ（クエリ文字列）は存在しません。
そのため、useSearchParams に関連する部分を削除してしまい、`<Suspense>` も使わないという選択もできます。

A
Appendix

# 静的サイトジェネレーターと next/image のローダー

Next.js を静的サイトジェネレーター（SSG：Static Site Generator）として利用する場合、これまでは `next export` を使っていましたが、Next.js 13 では `next.config.js` に以下のような設定を追加します。

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export',
}

module.exports = nextConfig
```

next.config.js

そして、`next build` を実行することで、静的サイトジェネレーターとして出力します。静的サイトジェネレーターとして利用する際にサポートされる機能は、基本的には Pages Router の場合と大きな違いはありません。詳細は、以下のページで確認してください。

Static Exports - Supported Features
https://nextjs.org/docs/app/building-your-application/deploying/static-exports#supported-features

## ❖ next/image のローダー

静的サイトジェネレーターとして利用する際には、`next/image` の扱いも変わりなく、外部の画像処理 API を利用することになります。ただし、内蔵のローダーを使うことは非推奨となり、個別にローダーを指定する形へと変更されました。

ローダーを用意したうえで、以下のように設定します。

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export',
  images: {
    loader: 'custom',
    loaderFile: './lib/image.js',
  },
}

module.exports = nextConfig
```

next.config.js

代表的な画像処理サービス用のローダーのひな形は、以下のページに用意されています。

images
https://nextjs.org/docs/app/api-reference/next-config-js/images

`<Image>` コンポーネントの loader 属性でも設定できることになっていますが、Next.js 13.4 ではうまく動きません。

B
Appendix

# Route Handlers と On-demand ISR

App Router でも、API を利用した Ondemand-ISR を設定できます。

## Route Handlers

Pages Router では `/pages/api` の中に作成していた API Route ですが、App Router では Route Handlers となりました。

> Route Handlers
> https://nextjs.org/docs/app/building-your-application/routing/router-handlers

Route Handlers を利用するには、App Router の中に `route.js` を作成します。`page.js` と同様に、ディレクトリを使ってセグメントを作成することもできます。ただし、`page.js` と `route.js` を同じディレクトリの中に置くことはできません。

Pages Router でプロエジェクトを作成した際にサンプルとして用意される `/pages/api/hello` と同じように機能する API を作ると、次のような感じになります。

```
import { NextResponse } from 'next/server'

export async function GET() {
  return NextResponse.json({ name: 'John Doe' })
}
```

app/api/hello/route.js

HTTP メソッドを関数名にして、export します。対応している HTTP メソッドは、GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS です。

また、NextResponse という関数も用意されており、これを使うことでレスポンスをシンプルに構成できます。

NextResponse
https://nextjs.org/docs/app/api-reference/functions/next-response

## ❖ On-demand ISR

Route Handlers で API を用意して、On-demand ISR を設定していきます。App Router で On-demand ISR を実現するには、revalidatePath を使います。

revalidatePath
https://nextjs.org/docs/app/api-reference/functions/revalidatePath

書籍サンプルと同様な機能をもたせると、以下のような構成になります。

```
import { NextResponse } from 'next/server'
import { revalidatePath } from 'next/cache'

export async function GET(request) {
  const { searchParams } = new URL(request.url)

  const secret = searchParams.get('secret')

  if (secret !== process.env.SECRET_TOKEN) {
    return NextResponse.json({ message: 'Invalid token' }, { status: 401 })
  }

  try {
    revalidatePath('/blog/[slug]')
    return NextResponse.json({ revalidated: true, now: Date.now() })
  } catch (err) {
    return NextResponse.json({ message: 'Error revalidating' }, { status: 500 })
  }
}
```

app/api/revalidate/route.js

書籍のサンプルと同様に、トークンを使って

```
/api/revalidate?secret=1234567890
```

へアクセスすると、On-demand ISR が実行されます。

ただし、ここで注意しなければならないのは、revalidatePath に渡すパスです。
res.unstable_revalidate ではページの URL を指定して、ページ単位でリビルドすることができました。しかし、revalidatePath ではダイナミックセグメントに対してセグメント単位でのリビルドとなります。そのため、`/blog/[slug]` といった形で指定します。

C
Appendix

# 404 ページのカスタマイズ

App Router で 404 ページをカスタマイズするには、`app/not-found.js` で notFound 関数を使い、以下のような構成で作成します。

not-found.js
https://nextjs.org/docs/app/api-reference/file-conventions/not-found

```
import Container from 'components/container'
import Hero from 'components/hero'

export default function NotFound() {
  return (
    <>
      <title>404: ページが見つかりません </title>
      <Container>
        <Hero title="404" subtitle=" ページが見つかりません " />
      </Container>
    </>
  )
}
```

app/not-found.js

`not-found.js` では metadata オブジェクトが使えないため、`<title>` を直接指定しています。


カスタムで用意した404ページ

# エビスコムの書籍


## 作って学ぶ WordPressブロックテーマ

サイトエディターや theme.json など、WordPress のブロックテーマを理解・制作するのに必要なことまとめた 1 冊。

https://amzn.to/3WGSfj7

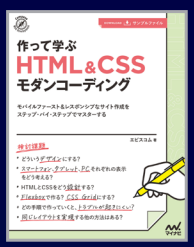
https://ebisu.com/wp-blocktheme/


## 作って学ぶ　Next.js/React Webサイト構築

ステップバイステップでマスターする、Next.js による Web 制作入門実践書。

https://amzn.to/3RvfR8D

https://ebisu.com/next-react-website/




## 作って学ぶ　HTML＆CSSモダンコーディング

モバイルファースト＆レスポンシブなサイト作成を、ステップバイステップでマスターする。デザインを実現するＣＳＳのバリエーションも解説。

https://amzn.to/2XsZHoU

https://ebisu.com/html-css-modern-coding/


## HTML5＆CSS3デザイン 現場の新標準ガイド

HTML と CSS の最新仕様を整理し、制作の現場で必要不可欠な情報をまとめたガイドブック。

https://amzn.to/378x17B

https://ebisu.com/html5-css3-practical-design-guide-2/

# EBISUCOM

エビスコムの書籍

https://amzn.to/3BMBZTf

印刷書籍

https://ebisu.com/book/

オリジナル電子書籍

https://ep.ebisu.com/

最新の情報は下記でも適宜お伝えしていますので、参考にしてください。

- 著者 NOTE
  https://ebisu.com/note/
- エビスコム Twitter
  https://twitter.com/ebisucom
- 電子書籍出版部 Twitter
  https://twitter.com/ep_ebisucom

■著者紹介

エビスコム
https://ebisu.com/

Web と出版を中心にフロントエンド開発・制作・デザインを行っています。
HTML/CSS、WordPress、GatsbyJS、Next.js、Astro、Docusaurus、Figma、etc.

主な編著書：『作って学ぶ WordPress ブロックテーマ』マイナビ出版刊
『作って学ぶ Next.js/React Web サイト構築』同上
『作って学ぶ HTML & CSS モダンコーディング』同上
『HTML5 & CSS3 デザイン　現場の新標準ガイド【第 2 版】』同上
『Web サイト高速化のための 静的サイトジェネレーター活用入門』同上
『CSS グリッドレイアウト デザインブック』同上
『フレキシブルボックスで作る HTML5&CSS3 レッスンブック』ソシム刊
『CSS グリッドで作る HTML5&CSS3 レッスンブック』同上
『WordPress ノート クラシックテーマにおける theme.json の影響と対策 2023』エビスコム電子書籍出版部刊
『Astro v2 と TinaCMS でシンプルに作るブログサイト』同上
『HTML&CSS コーディング・プラクティスブック 1〜8』同上
ほか多数

作って学ぶ Next.js/React Web サイト構築

# Next.js 13 対応ガイド

2023 年 7 月 5 日　　ver.1.0 発行