全4回で学ぶNext.js入門講座
3回目

2025/11/18

Kazuma SEKIGUCHI

前回のアジェンダ

- CSS Modules または Tailwind でのスタイル
- レイアウト（layout.js）の紹介と共通デザイン
- ルーティングの拡張
- 動的ルーティング
- use client の利用

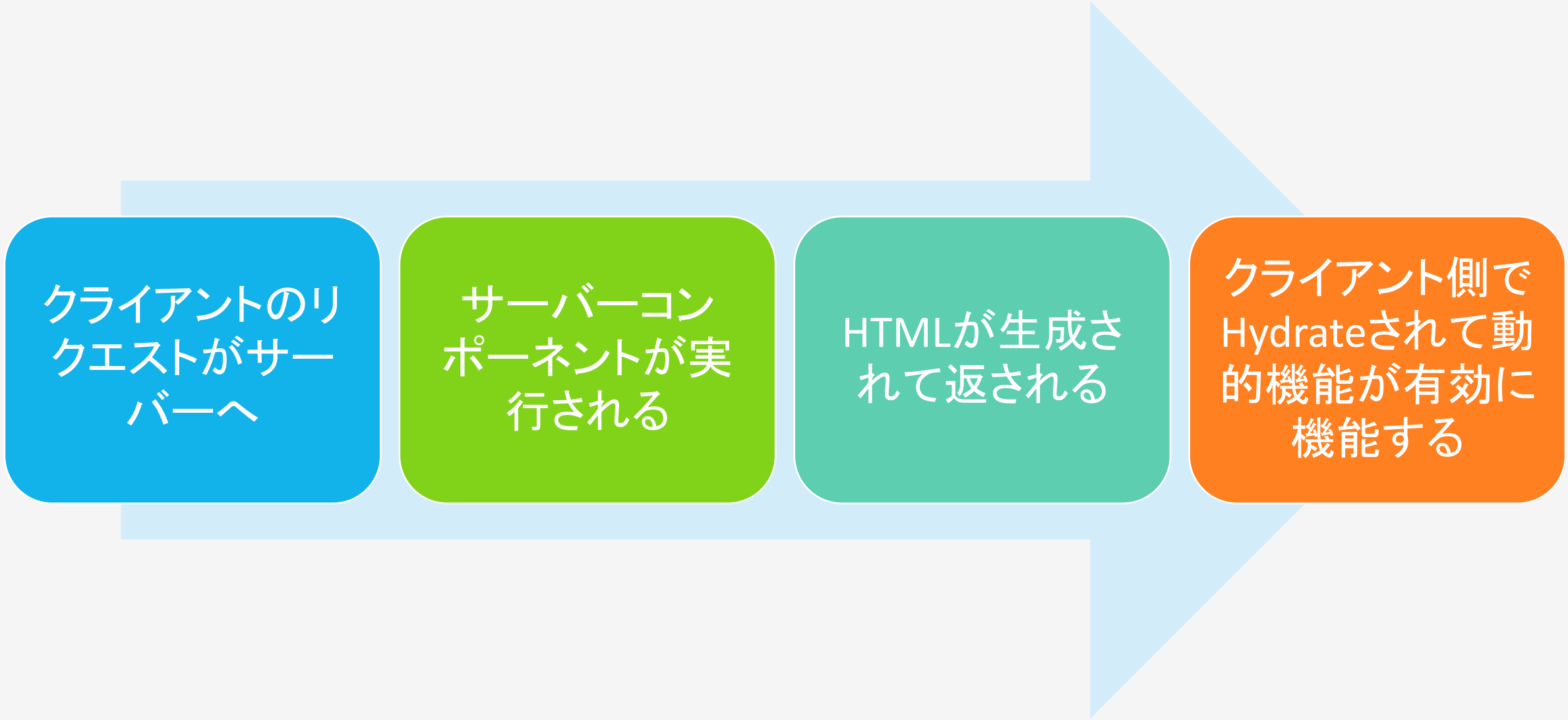
今回のアジェンダ

- SSRとは？
- サーバーによるレンダリングの仕組み
- サーバーコンポーネントでの fetch
- 外部APIからのデータ取得
- revalidate（キャッシュと再検証）の体験

SSRとは

- SSR（Server-Side Rendering）は、HTMLをサーバー側で生成して返す仕組み
- リクエスト時にサーバーでページをレンダリング
  - 完成したHTMLがクライアントへ送られる
  - ブラウザ側での初期表示が早くなるため、SEO評価を得やすい
    - 検索エンジンがレンダリング済みHTMLを取得できるため
- App Routeを利用した場合デフォルトでSSRが有効になる
  - つまるところ、Next.jsを使うとSSRになる

サーバーにおけるレンダリング



RSC（React Server Components）

- Reactのコンポーネントをサーバーで実行して描画できるという、React側の仕組み
  - Next.js13以降で使われている
  - どのコンポーネントをサーバー側で実行するか、どれをクライアントで実行するかを自由に選択できる
- Next.jsでは大部分がServerComponents
  - SSRのようにHTMLがサーバーで生成される
  - つまり、RSCがSSRのためのメイン手段となった
    - 実際SSRは過去から存在している
- 分かりづらいがRSCはSSRとは別概念
  - SSRは方式の名称、RSCはReactの機能

Hydrate？

- SSRで作成されるのは単なるHTML
  - つまり静的なページであるため、JSを使った動的な動きなどを実現することができない
  - ブラウザー側でReactが起動して、イベントや状態管理などの動きを復元するのがHydrate
- ブラウザーでReactが起動してHTMLに機能を結びつけるようになる
  - イベント登録など
  - ページが動的に動くようになる

Hydrate？

- Next.jsの場合、Hydrateが必要なのは、
  - クライアントコンポーネントを使う（use clientを使う）
  - インタラクティブなUIを使う

  ときだけ
- use clientを記述すれば自動的にHydrateが有効になる
  - use clientを記述するとクライアントコンポーネントになるため、どうしてもブラウザー側でのReact実行が必要になる

fetchによるデータアクセス

- JSを利用してデータを取得する場合
  - APIなどにリクエストを出して、JSONを取得するのが一般的
  - JSのFetchメソッドを利用して、クライアント側のJSからサーバー側のAPIにアクセスを行なう
- クライアント側からアクセスするのは分かりやすい
  - Next.jsはサーバー側でもクライアント側でも動作する
    - 基本的にサーバー側で実行する方がメリットが多い
  - サーバー側からFetchを使ってデータ取得が可能

サーバーコンポーネントでのFetch

- fetchは外部からデータを取得するときに利用される
  - APIへアクセスをおこない必要なデータを取得してくる
- Next.jsの場合、Fetchもサーバー側で実行できる
  - 結果だけHTMLとしてユーザーに届けることが可能
  - APIにアクセスするためのAPIキーやシークレットキーが漏れない
    - JSだとどうしても漏れてしまう
- APIのリクエストがクライアントから飛ばないため、CORSエラーが発生しづらい
- use clientを付けると、クライアント側で実行することになるので注意

サーバーコンポーネントでのFetch

- fetchは自動的に結果がキャッシュされる

```
// app/page.js
export default async function Page() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return <pre>{JSON.stringify(data, null, 2)}</pre>;
}
```

- 都度リクエストすることも可能

```
fetch(url, { cache: 'no-store' })
```

サーバーコンポーネントでのFetch

- revalidate
  - 指定秒数後にキャッシュを自動的に再生成する仕組み
  - Next.jsではISR（Incremental Static Regeneration）と同義
  - 外部APIやDBの内容などがあまり更新されない場合、極めて有効

```
const res = await fetch('https://api.example.com/posts', { next: { revalidate: 60 } });
```

  - 初回アクセスでキャッシュを生成し、以後60秒経過したらアクセスがあったときにはキャッシュしている内容を返しつつ、バックグランドでキャッシュを再生成する
    - 次回アクセスから新データに切り替わる

サーバーコンポーネントでFetchする

- サーバーからもFetchメソッドを使ってデータ取得が可能
  - サーバーコンポーネントでURLを指定して、アクセスしデータを取得する

```
async function getSampleData(){
const response = await fetch(url,{cache:'no-store'})
const result = response.json()
return result;
```

キャッシュを確保せずに
常に最新を取得する

fetchは非同期関数なので
結果取得まで待つ

- publicフォルダーをappフォルダー内に作成して、sample.jsonファイルを格納しておく
  - このJSONファイルに対してアクセスして取得してみる

非同期扱い

- fetchは非同期で動作するため、async,awaitを組み合わせて利用するのが通常

```
async function getSampleData(){
  const response = await fetch(
    url,
    {cache:'no-store'}
  )
  return await response.json()
}

export default async function Home() {
  const data = await getSampleData()
```

awaitを使うため、functionにasyncを付ける

awaitを使うため、functionにasyncを付ける

- exportの方のfunctionに付けるのを忘れやすいので注意

ちなみにNext.jsでAPIを作る

- APIを作成する場合、返すのはJSONデータ
  - 常にサーバーで実行する必要がある
  - ファイル名をpage.jsではなく、route.jsとすることで、APIとしてアクセスされたときに結果を返すことが可能
  - HTTPのメソッド名で関数を作成しておく
    - GETでアクセスするAPIの場合

```
export async function GET() {
  const nowJST = new Date().toLocaleString("ja-JP", {
    timeZone: "Asia/Tokyo"
  });
  return Response.json({
    now: nowJST
  });
}
```

関数名をGET()にしておく
POST()なども利用可能

ありがとうございました。

また次回。