Engineer

サーバーサイドレンダリング

リクエストごとにサーバーでページのHTMLを生成すること

どのようなものか

サーバーサイドレンダリングは、誰かがページを要求した時点で、そのHTMLをサーバー上で生成することです。通常はSSRと略されます。ブラウザーにはコンテンツがすでに入った完全な文書が届き、スクリプトが動く前から表示できます。JavaScriptフレームワークでは、その後、同じコンポーネントがブラウザーでももう一度動き、受け取ったHTMLを引き継いで操作できるようにします。

この用語が答えるのは、HTMLをいつ、どこで作るかという問いです。よくある答えは3つあります。静的サイトジェネレーターはビルド時に1回作り、全員に同じファイルを配信します。サーバーサイドレンダリングはリクエストごとにサーバーで作るため、要求した人や、その時点のデータに応じたページにできます。CSRとも呼ばれるクライアントサイドレンダリングは、ほぼ空の文書とJavaScriptのバンドルを送り、ブラウザーでページを組み立てます。実際のアプリケーションは、ルートごとにこの3つを使い分けることがよくあります。

Next.jsも同じように説明しています。ドキュメントでは、ダイナミックレンダリングをSSRの別名として挙げ、ページのHTMLをリクエストごとに生成するとしています。Pages Routerでは、ページが `getServerSideProps` という非同期関数をエクスポートするとSSRが有効になり、サーバーはリクエストのたびにこの関数を呼びます。対になる `getStaticProps` は、同じ役割をビルド時に1回だけ果たします。その土台となる仕組みを提供するのがReactです。 `react-dom/server` のAPIは、コンポーネントのツリーをサーバー上でHTMLに変換します。Node.js向けの `renderToPipeableStream` やWeb Streams向けの `renderToReadableStream` ならストリームとして、従来の `renderToString` なら一括で生成します。Reactは最後のAPIを、機能に制限のあるレガシーAPIとして位置付けています。

後半の処理はブラウザーで行い、ハイドレーションと呼びます。Next.jsは最初の読み込みについて、まずHTMLで操作のできないプレビューを素早く表示し、その後JavaScriptでコンポーネントをハイドレーションすると説明しています。ハ

イドレーションとは、Reactが既存のDOMにイベントハンドラーを結び付け、ページが入力に反応するようにする処理です。Reactの `hydrateRoot` リファレンスは、その利点と代償を率直に述べています。HTMLのスナップショットによりアプリケーションの読み込みが早く見えますが、ページは本当に操作できるようになる少し前から、準備ができたように見えます。

概念を判断に生かす

「サーバーサイドレンダリングとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 読み込みが速く、スマートフォンでも使え、お客様がどのブラウザーを使っていても同じように動く画面を提供します。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 フルスタックのGIS地図開発を指揮 — DBからブラウザーまでのフルスタック地図開発

フルスタックのGIS地図開発を指揮 お問い合わせ →

なぜ重要なのか

最初の応答にコンテンツが入っています。低速な回線の利用者にも、JavaScriptがまだ届いていない間から文字や画像を見せられます。JavaScriptを実行しないクローラーも、空の外枠ではなく、人が見るのと同じページを読み取れます。ブラウザーは、スクリプトを実行してからデータを取得するまで待たずに、実際のコンテンツを描画できます。

リクエストに応じてページを変えられる点も、静的ビルドとの違いです。Next.jsは、認可ヘッダーや訪問者の所在地など、利用者固有のデータやリクエスト時にしか分からないデータには `getServerSideProps` を勧め、それ以外にはビルド時の生成を勧めています。データの取得は、そのデータがある場所の近くで行います。関数内のコードはサーバーでしか動かず、そのインポートはクライアント向けバンドルから除かれ、データベースやコンテンツ管理システムへ直接問い合わせられます。ブラウザーならスクリプトを読み込んだ後に順番に送る複数のリクエストを、サーバーとの1往復で置き換えられます。

静的サイトで省ける分が、そのまま負担になります。動作中のサーバーが必要で、リクエストのたびにレンダリングを繰り返します。データがそろうまで応答を開始できないため、ストリーミングを使わなければ、最も遅いクエリがページ全体の速度を決めます。また、サーバーで描画してもJavaScriptが不要になるわけではありません。ブラウザーはハイドレーションに必要なすべてのコンポーネントを引き続きダウンロードし、処理が完了するまでは、画面に見えるボタンが押しても動かないことがあります。

そのため、プロジェクト全体ではなくページごとに選ぶべき仕組みです。誰が読んでも同じページをリクエストのたびに描画し直す利点はわずかです。利用者によって変わるページや、分刻みで動くデータを使うページでこそ、SSRのためにサーバーを用意する意味があります。当サイトは静的生成を採用しています。技術構成では、適した箇所では静的生成を優先しており、これらのページはリクエスト時にアプリケーションを実行する必要がありません。サイトのビルドにはHugoを使っています。

実際の活用例

次のページはこの記事のために書いたもので、Pages Routerを使っています。ログイン済みの顧客の注文を一覧表示し、セッションのない訪問者はログインページへ送り、どのキャッシュにも結果を保存しないよう指示します。`readSession` と `db` は、アプリケーションが実際に使うセッションとデータの仕組みを表す仮の名前です。

```
// pages/orders.js
export async function getServerSideProps({ req, res }) {
  const session = await readSession(req.cookies.session);
  if (!session) {
    return { redirect: { destination: "/sign-in", permanent: false } };
  }

  const rows = await db.orders.listFor(session.userId);
  res.setHeader("Cache-Control", "no-store");

  return {
    props: {
      orders: rows.map((o) => ({
        id: o.id,
        number: o.number,
        status: o.status,
      })),
    },
  };
}

export default function Orders({ orders }) {
  return (
    <ul>
      {orders.map((o) => (
        <li key={o.id}>
          {o.number}: {o.status}
        </li>
      ))}
```

```
    </ul>
  );
}
```

Next.jsは `/orders` へのリクエストごとにこの関数を実行します。 `req.cookies` は、すでに解析されたリクエストのCookieヘッダーです。 `redirect` で `permanent: false` を指定すると、恒久的ではなく一時的なリダイレクトになります。 `props` として返す値は `JSON.stringify` でシリアライズできなければなりません。また、Reactがサーバーと同じデータで一覧をハイドレーションできるよう、最初のHTMLの一部としてブラウザーにも届きます。このため、この関数はデータベースの行をそのまま渡さず、ページに表示する3つのフィールドだけに絞っています。利用者が後から `next/link` のリンクを通ってページへ移る場合、Next.jsはサーバーに関数の再実行を求め、ページ全体を読み直さずにpropsをJSONで受け取ります。

創業者が現在NextMarinerで取り組んでいるNext.js 16のフロントエンドは、この仕組みが解決する問題を示しています。その実績記事の出発点は、素早く表示され、操作への反応も保つ必要があった認証付きダッシュボードです。すべてをブラウザーで取得すると初回読み込みが遅くなり、N+1パターンも生じます。各コンポーネントが動き始めて個別にリクエストを送るため、1つのページで通信の往復が連鎖するからです。このダッシュボードと大規模なディレクトリー一覧では、現在は `createPrefetchedServerPage` というファクトリーがサーバーでページのデータを取得し、データをそろえた状態でクライアントへ渡します。コンポーネントは個別に問い合わせに出ず、最初からデータを持って動き始めます。このファクトリーへ移行したページは、多数の往復ではなく1往復で表示されます。同じ記事では、他の2方式も適した場所に残しています。公開ページは利用者ごとに変わらないため静的に生成し、本当に対話的な部分はクライアントで描画します。また、データを新鮮に保ちつつ、アプリケーションがすでに持つデータを再取得しないよう、データリクエストごとのキャッシュ無効化戦略も記録しています。

問題が起きるところ

サーバーとブラウザーで出力が一致しません。 ハイドレーションは、ブラウザーの最初のレンダリングでサーバーと同じ出力が得られることを前提にしています。 `typeof window` を調べるコンポーネント、描画中に `window.matchMedia` を呼ぶコンポーネント、現在時刻を表示するコンポーネントは、サーバーとブラウザーで異なるものを描画します。Reactが挙げる原因には、両者のデータの相違や、ルート要素内の余分な空白もあります。不一致は、軽くてもハイドレーションを遅らせ、最悪の場合はイベントハンドラーを別の要素に結び付けてしまいます。そのため、 `hydrateRoot` リファレンスは「不一致はバグとして扱い、修正すべき

です」と明言しています。両側で同じものを描画してください。ブラウザーで本当に異なる表示が必要なら、まずサーバーと同じものを描画し、ハイドレーション後にエフェクトで切り替えます。ただし、これには2回目のレンダリングが必要です。`suppressHydrationWarning` はタイムスタンプなど避けられない差に限って使い、1階層分にしか効かないことにも注意します。

秘密情報がpropsと一緒に届きます。 propsとして返すものはすべて、ブラウザーでハイドレーションできるよう最初のHTMLに書き込まれます。Next.jsのドキュメントは、機密情報をこの方法で渡さないよう警告しています。利用者のレコード全体を返す関数は、ページのソースを開いた誰にでもパスワードハッシュや内部フラグを渡してしまいます。上の例のように、ページに表示するフィールドだけを返します。

共有キャッシュが、ある利用者のページを次の利用者へ配ります。 Next.jsのドキュメントは、サーバーで描画したページのキャッシュ例に `Cache-Control: public, s-maxage=10, stale-while-revalidate=59` を使っています。これを個人向けのページにコピーすると、サーバーの前にあるコンテンツ配信ネットワークやプロキシに、応答を保存して次に要求した人へ渡すよう許可してしまいます。RFC 9111では、`public` は通常なら拒む応答の保存をキャッシュへ許可するもの、`s-maxage` は共有キャッシュ向けの有効期間と定義されています。どちらの指定も、`Authorization` ヘッダーを持つリクエストへの応答を共有キャッシュが再利用できないという規則を解除します。Cookieや認証情報に基づくページには、共有キャッシュでの保存を禁止する `private` か、すべてのキャッシュでの保存を禁止する `no-store` が必要です。

最も遅いクエリがページ全体を待たせます。 ページを一括で描画すると、その裏で動くすべてのクエリが返るまで何も送られません。1つの遅いクエリが、ヘッダーもナビゲーションも他のすべても遅らせます。Reactのストリーミングレンダラーは、`<Suspense>` の境界でページを分けます。どの境界の内側にもない部分であるシェルは、準備でき次第、未完了部分の代替表示を入れて送ります。その後、残りの部分を順に流し、小さなインラインスクリプトで差し替えます。早く送ることには注意点があります。ストリーミングの開始後はステータスコードを変更できないため、レコードの欠落や検査の失敗はシェルを送る前に確定させる必要があり、エラーステータスを設定する場所は `onShellError` です。

参考資料

- サーバーサイドレンダリング（SSR） — リクエストごとのHTML生成と、`getServerSideProps` と `getStaticProps` の違いについてのNext.jsの説明です。

- getServerSideProps — 使う場面、最初のHTMLでクライアントに届くもの、`Cache-Control` によるキャッシュを説明しています。
- サーバー向けReact DOM API — コンポーネントをサーバー上でHTMLに変換するための、ストリーミングAPIと従来のAPIです。
- hydrateRoot — Reactによるサーバー生成HTMLの引き継ぎと、ハイドレーションの不一致のよくある原因を説明しています。
- renderToPipeableStream — シェル、`<Suspense>` の境界、ストリーミング時のステータスコードの注意点を説明しています。
- RFC 9111：HTTPキャッシュ — `public`、`private`、`s-maxage`、`no-store` の各ディレクティブと共有キャッシュの規則を説明しています。

用語集

実際の制約に合わせて選ぶ

業務の流れ、既存のインターフェース、変更を必要とする利用者をお知らせください。検証可能な納品内容と、お客様のチームが保守を続けられる引き継ぎ内容を定められます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「サーバーサイドレンダリングとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く 連絡先カード

心を込めて丁寧に制作： D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/server-side-rendering/