HF ドップラー観測データの利活用を目的とする Web アプリケーションのレンダリング設計

5421 中嶋 柊（指導教員：才田 聡子）

Rendering Design of a Web Application for HF-Doppler Visualization

概要： 電離圏研究を目的とする短波ドップラー (HFD) の観測データを保管しているユーザー体験の向上を目的として,Web アプリケーションの新規実装を行うものである.
本研究ではアプリケーション実装におけるデータの取得と画面描画の高速化に視点を置き, Next.js を用いたページレンダリングの最適化を目指し実装・検証する.

キーワード Web アプリケーション フロントエンド ページレンダリング TypeScript Next.js

1. はじめに

1.1 背景

電離圏研究のための短波ドップラー (HFD) 観測データは, オープンデータとして公開され, 自然科学分野で自由に使用できる.
現在, データの効率的な再利用を目指し, いくつかの web アプリケーションが開発されているが, 開発運用やユーザー体験に問題がある.
これらのアプリケーションは MySQL, PHP, HTML/CSS, JavaScript を使用し, 技術の幅広さが開発と保守のコストを増加させている. JavaScript（Node.js, React）を用いたアプリケーションも存在するが, データ呼び出しの遅さやインタラクティブ性の不足が課題となっている. [1)]

1.2 目標

本研究では既存アプリケーションの問題点から派生した要件について新規開発を行い, パフォーマンスと UX の向上を実現することを目指す.

1.3 全体像

開発の全体像を記す.
大きく分けて次の 3 つのタスクに担当者を分けて開発を行うこととした.

1. デザイン作成
2. バックエンド実装
3. フロントエンド実装

1.3.1 デザイン作成

デザイン作成ではクラウドベースのデザインツールである Figma を使用して, ワイヤフレームの作成を行いその後に詳細なデザインを構築する.
本開発では, ユーザーインターフェース（UI）, ユーザーエクスペリエンス（UX）, そしてアクセシビリティ（a11y）に配慮した設計の重要性が認識されている. [2)]
またコンポーネント単位で統一感のあるデザインが求められるため, デジタル庁が公開しているデザインシステムを参照し極力それに準拠することとした.[3)4)]
具体的実装は中村が担当する.

1.3.2 バックエンド実装

バックエンド実装では TypeScript を使用し Next.js 上で関数実装を行う.
実装内容としては既存サイトからのデータのスクレイピングと整形である.
スクレイピングを行う上で, 規約の確認を行った上で実行速度や適切なインターフェースの定義を意識した上で実装を行う必要がある.
具体的実装は藤本と中嶋が担当する.

1.3.3 フロントエンド実装

フロントエンド実装は大きく UI 実装とパフォーマンス・チューニングの 2 つに分けられる.
Next.js 上で TailwindCSS を活用したスタイリング実装を行い a11y を意識して UI を実装していく担当と, データ取得部との結合やレンダリングの工夫によりパフォーマンス向上を目指す担当とに分けて実装をおこなう.
具体的実装としては UI 実装を中村, 木村, 高橋, データ結合部との結合及びレンダリング実装を中嶋が担当する.

1.4 本研究の目標

本研究では, フロントエンド実装のうちデータ取得とページのルーティング・レンダリングに視点を置き, 実行速度や体感的待機時間を意識しての実装と検証について述べるものとする.

2. 使用技術

2.1 Next.js について

Next.js は, Vercel 社が提供する Web フレームワークで, React.js をベースにしている.
その最大の特徴は, 多様なページレンダリング手法のサポートで, 開発者はアプリケーションの特定のニーズに応じて最適なレンダリング戦略を選択できることである.[5)]
Next.js のレンダリングオプションには, クライアントサイドレンダリング (CSR), サーバサイドレンダリング (SSR), スタティックジェネレーション (SG), インクリメンタル静的再生成 (ISR) が含まれる.
2024 年現在, クライアントサイドでのレンダリングをデフォルトとする Page Router と, サーバーサイドでのレンダリングをデフォルトとする App Router の 2 種

類のルーティング手法がある．本研究では Page Router を使用する．

2.2 ページレンダリング手法

2.2.1 ページレンダリングとは

Web アプリケーションでのページレンダリングは, ウェブサーバーまたはブラウザが HTML/CSS/JavaScript などのコードを解釈し, ユーザーのデバイス上で視覚的なページを表示するプロセスである．このプロセスの各ステップの実行タイミングにより分類され, 適切な手法の選定によりサイトの高速化, ユーザー体験の向上, SEO の改善が期待できる．

多くのアプリケーションでは, サーバーに送られたリクエストに基づき HTML を生成・返す ServerSideApplication や, 非同期リクエストを送る Ajax などが用いられている．

近年は, 仮想 DOM を用いることでパフォーマンス向上と効率的な UI 更新を実現する手法が普及しており Next.js でも仮想 DOM が使用されている．Vercel 社の CEO, Guillermo Rauch 氏は”7 Principles of Rich Web Applications”の中で, 'Pre rendered pages are not optional'（サーバーによるページの事前レンダリングは必須）と述べている．これは, サーバーサイドでのレンダリングがユーザー体験にとって重要であることを示唆している．[6)]

本文では, Next.js で用いられる主要なレンダリング手法とそのメリット・デメリットについて述べる．[7)]

2.2.2 CSR

CSR（クライアントサイドレンダリング）は, ウェブページのレンダリングをブラウザで行う手法である．サーバーからは HTML の基本骨格と JavaScript を送信し, ブラウザが HTML を動的に生成する．この手法は, ウェブページの内容を動的に更新し, ユーザーのインタラクションに応じて即時にコンテンツを変更できるため, SPA（シングルページアプリケーション）の構築に適している．また, サーバーへのリクエスト回数が減るため, サーバーの負荷を軽減できるという利点も存在する．

しかし, 初回ロード時に JavaScript のダウンロードと実行が必要で, 表示までの時間が長くなり SEO に不利な点がデメリットである．

また, JavaScript が無効の環境ではコンテンツが正しく表示されないリスクがある．

CSR のアーキテクチャ図を図 1 に示す．

2.2.3 SSR

SSR（サーバーサイドレンダリング）は, ウェブページのレンダリングをサーバー側で行う手法である．SSR ではユーザーのリクエストに応じてサーバーが HTML を生成し, 完成したページがクライアントに送信される．

SSR はクライアントサイドよりも高速なサーバーサイドで HTML が生成されるため, 初回ページロードの速度が向上し SEO に有効である．また, 完成した HTML が送信されるため SEO に強く, クライアントサイドの

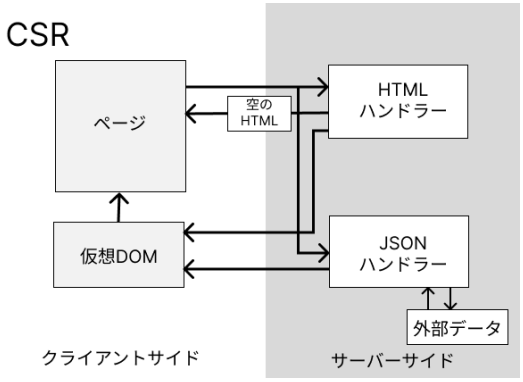

図 1　CSR のアーキテクチャ

JavaScript 実行環境に依存しにくい点もメリットである．しかし, サーバー負荷が大きい, ページ間移動にリロードが伴う, SG や ISR と比べコンテンツの表示に時間がかかるなどのデメリットがある．

SSR のアーキテクチャ図を図 2 に示す．

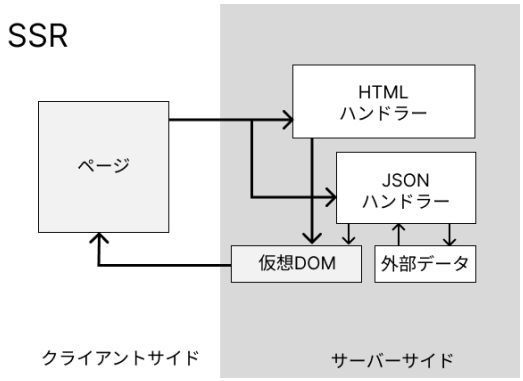

図 2　SSR のアーキテクチャ

2.2.4 SG

SG（スタティックジェネレーション）は, ビルド時に静的な HTML ファイルを生成する手法である．SG ではビルド時にすべてのページが静的 HTML として生成され, リクエスト時にはその HTML が提供される．

このアプローチの大きなメリットは, ビルド時に HTML が生成されるため, レスポンスが非常に高速であることである．しかし, ビルド後の変更が困難で, 動的なページ生成には向かないというデメリットがある．

SG のアーキテクチャ図を図 3 に示す．

2.2.5 ISR

ISR（インクリメンタルスタティックリジェネレーション）は, 静的サイト生成とサーバーサイドレンダリングの組み合わせ手法である．ISR ではビルド時に一部のページを静的に生成し, 残りのページはユーザーのリクエストに応じて生成・キャッシュされる．この方法のメリットは, サイトのビルド時間を短縮しつつ, 動的なコンテンツの提供が可能になる点にありパフォーマンスとコンテンツの更新の柔軟性が両立されている．しかし, デメリットとして再生成後の初回リクエスト時にオーバーヘッドが大きく, 読み込み時間が長くなる可能性がある．

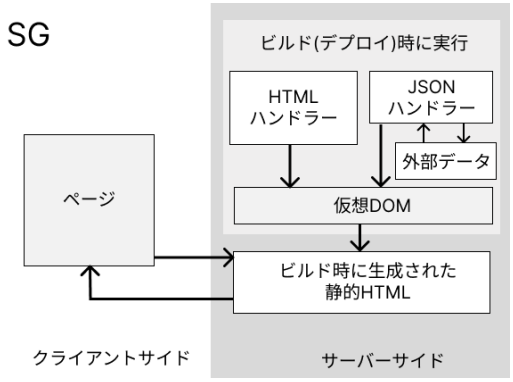

図 3　SG のアーキテクチャ

ISR のアーキテクチャ図を図 4 に示す.

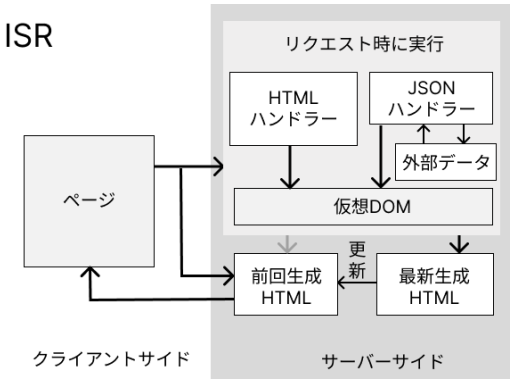

図 4　ISR のアーキテクチャ

2.2.6　Next.js におけるページレンダリング

Next.js におけるページレンダリングは, ページコンポーネントのライフサイクルとデータ取得方法に依存する. このフレームワークは, 異なるレンダリング手法をページごとに適用する柔軟性を持っている. 例えば, 単一のアプリケーション内で, 一部のページでは SSR を使用して動的にデータを取得し, 他のページでは SG を使用して静的に生成することが可能である.
この柔軟性により, 開発者はパフォーマンス, SEO, ユーザーエクスペリエンスのバランスを考慮しながら, アプリケーションの特定の要件に合わせて最適なレンダリング戦略を選択できる. これにより, 高速で対話的なウェブアプリケーションの構築が可能となり, 結果としてユーザーに優れた体験を提供することが可能となる.

2.3　スクレイピングについて

このアプリケーションでは, 各観測所で観測されたデータを公開している既存アプリケーションからデータをスクレイピングし, 再描画を行う. スクレイピングは, web アプリケーションから HTML データをクローラで取得し, 必要な情報を抽出する技術のことである.
スクレイピング時には, 取得元サイトの規約を遵守する必要がある. 今回の場合, 取得先のデータが OSS 化されており, 学術目的での使用に関して団体から許可を得ているため, 過剰な負荷をかけない範囲での利用が可能と判断した.
実装には Superagent と Cheerio ライブラリを用い, Next.js 上で TypeScript 関数として実装した.
一般に, 外部サイトにアクセスする際は CORS（Cross-Origin Resource Sharing）エラーに注意が必要である. これは, 異なるオリジン（ドメイン, プロトコル, ポート）からのリソースにクライアントサイドでアクセスしようとした際に生じるセキュリティ問題である.[8)]
今回のように異なるドメインの API からデータを取得しようとする場合, ブラウザはこれを同一オリジンポリシーによりブロックする可能性がある. しかし, 今回はサーバーサイドでのレンダリングを行い, クライアント側で直接アクセスされないため, Next.js 上でのスクレイピングにおいて CORS エラーは発生しない.

3.　アプリケーション設計と実装方法

3.1　実装の要件

本研究では, アプリケーション実装の主要部分として外部データの取得とページの描画に焦点を当てている. 特に, データフェッチからの画面描画速度の高速化を UX 向上のための主要な目標と設定した.
今回作成するアプリケーションに含まれる, web スクレイピングを伴うページは以下の 2 つである.

- 波形データの検索ページ
  (index.tsx)
- 波形データの描画ページ
  (/data/[type]/[loc]/[year]/[date].tsx)

これらのページの実装において, ビルド時間とスクレイピングのリクエスト数の 2 つの軸で最適化を行うこととする.

3.2　スクレイピングに用いた関数

今回実装したスクレイピング関数は二つあり, 一つは指定された日時と観測所の周波数データを取得して CSV から JSON に変換するもの, もう一つは存在するデータの一覧を取得して整形するものである.
スクレイピング対象のアプリケーションでは, 観測所, 年度, 日時に従ってネストされたパスから周波数データを取得できる. 第一の関数では, URL を使って具体的なデータの所在ページを指定し, スクレイピングを行う. 第二の関数では, 観測所の一つ上の階層から始まり, 子要素のページを再帰的に全探索してスクレイピングし, 整形する.

3.3　レンダリング設計

レンダリング設計に関しては, 以下の二つのパターンで実装し, ビルドにかかる時間を比較して考察する.

1. ISR を用いる方法
2. SG と SSR を用いる方法

これにより, スクレイピングによるデータ取得とページレンダリングの最適な組み合わせを評価する.

3.3.1　ISR を用いる方法

最初の手法は, ISR を用いた実装である. この手法では, ビルド時に一括でスクレイピング処理を行い, 静的なサイトを生成する.
実装方法としては, まず getStaticPaths 関数でディレ

クトリ一覧をスクレイピングし, 静的にビルドされるページを特定する. 次に, getStaticProps 関数を用いて, それぞれのページの波形データを取得しビルドを行う. さらに, ISR 技術を活用することで, 再ビルドのタイミングを制御し, 定期的に新しく追加されたデータを取得できるように設定する.
図 5 で, ISR を用いた場合のアプリケーション設計を示す.

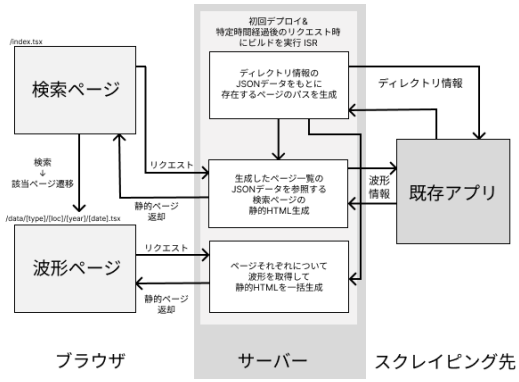

図 5　ISR を用いた場合の設計

3.3.2　SG と SSR を用いる方法

二つ目の手法は, SG と SSR を組み合わせた方法である.
この手法では, ディレクトリの取得と波形データの取得をそれぞれ SG と SSR を用いて分離させる. 具体的な実装方法としては, getStaticProps を使用して検索ページに存在するパスを静的に生成し, getServerSideProps を利用して各ページがリクエストされた際に動的に波形データを返すという流れになっている.
図 6 で, SG と SSR を併用した場合のアプリケーション設計を示す.

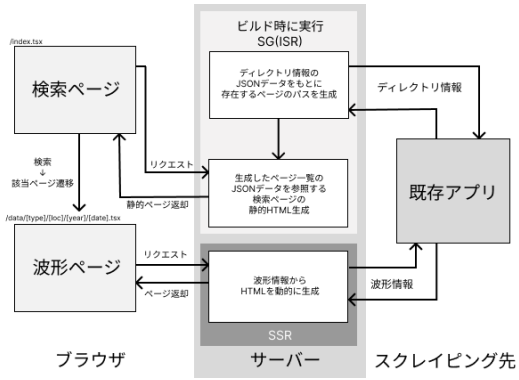

図 6　SG と SSR を併用した場合の設計

3.4　環境

本研究での実行環境に関して, 使用した PC の仕様を表 1 に, 使用したパッケージのバージョンを表 2 に示す.

4.　実装結果

4.1　データ取得

本研究での実装において, 2 種類の方法いずれでもデータを取得することができた.

表 1　実行環境

名称	MacBookPro
CPU	Apple M1 Max
メモリ	32GB
コマンドシェル	zsh (v1.85.1)

表 2　パッケージのバージョン

名称	バージョン
Node.js	18.12.0
Next.js	13.0.6
TypeScript	4.9.4

データ取得の成功は, スクレイピングとレンダリングの設計が適切に機能していることを示している.
図 7 と図 8 は, 取得したデータを JSON 形式でブラウザ上で表示した様子を示している. これにより, データ取得の結果が視覚的に確認できる.

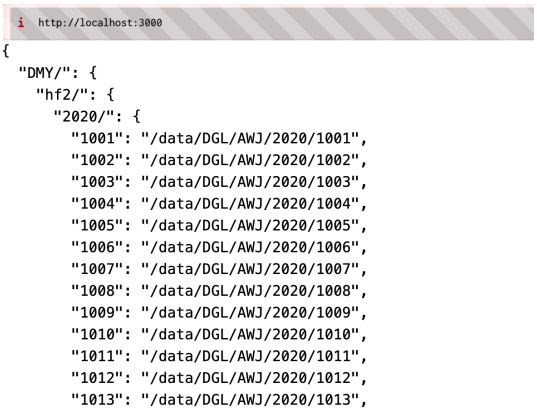

図 7　検索ページでデータ取得している様子

4.2　スクレイピング関数の実行時間

実装した 2 つのスクレイピング関数の実行時間について詳述する.
実行時間の計測には performance.now() メソッドを用い, 関数の実行前後の時間差で測定した.

4.2.1　scrapeWaveData

この関数は, 指定されたデータ形式, 観測所名, 年, 日時を引数として受け取り, 観測データの CSV を取得し JSON 形式に変換する.
この関数の実行時間を表 3 に示す.
表**??**から, この関数の実行には平均して約 1.2 秒かかることがわかる.

4.2.2　scrapeDirectory

この関数は, 指定された URL 配下に存在するページを再帰的に全探索し, ディレクトリ名をキーとするネストされたオブジェクトとして返すものである.
この関数の実行時間を表 4 に示す.
表 4 から, この関数の実行には平均して約 292 秒かかることがわかる.

4.3　静的ビルドの実行時間の比較

ISR を用いる手法と SG と SSR を併用する方法との 2 パターンでビルドを実行し, ビルドログに出た実行時間

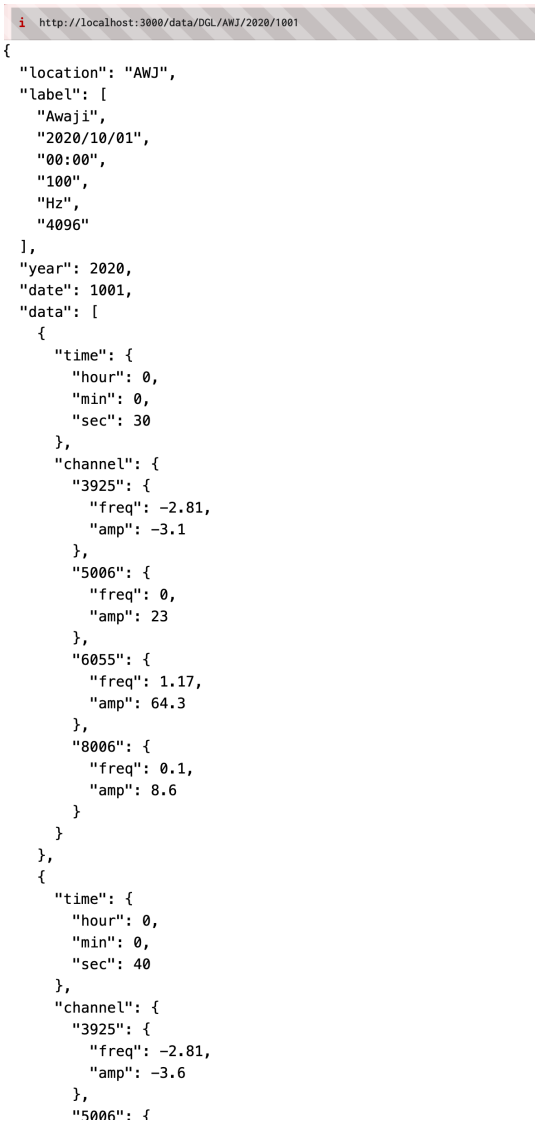

図 8　波形ページでデータ取得している様子

表 3　scrapeWaveData の実行時間

回数	実行時間 (ms)
1 回目	1435
2 回目	1166
3 回目	1054
4 回目	997
平均	1163

を示す.

4.3.1　ISR を用いる方法

ISR(SG) を用いて, ビルド時に存在するパスを生成した後各ページまでを静的に生成する方法を用いた場合

表 4　scrapeDirectory の実行時間

回数	実行時間 (ms)
1 回目	319411
2 回目	325276
3 回目	263359
4 回目	261885
平均	292482.75

のビルドログを図 9 に示す.

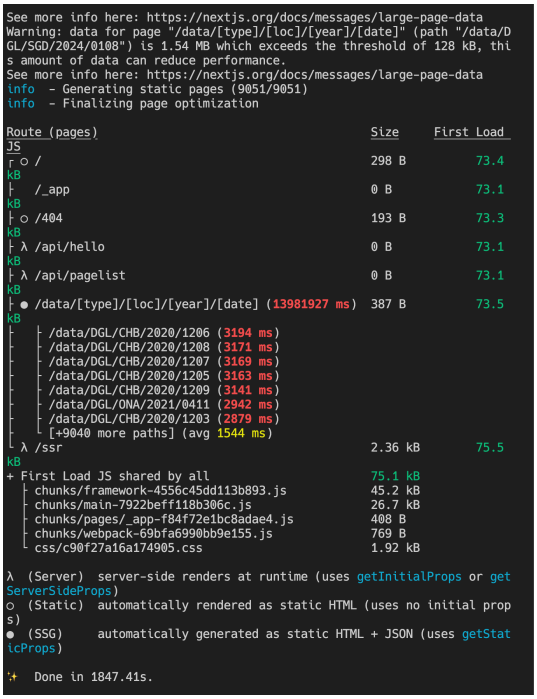

図 9　ISR(SG) を用いた際のビルドログ

実行結果からビルド時間は約 188 秒であることが確認できる.

4.3.2　SG と SSR を用いる方法

SG と SSR を併用して, ビルド時には存在するパスのみを静的に生成しておき各ページはアクセスされた際に動的に生成した場合のビルドログを図 10 に示す.

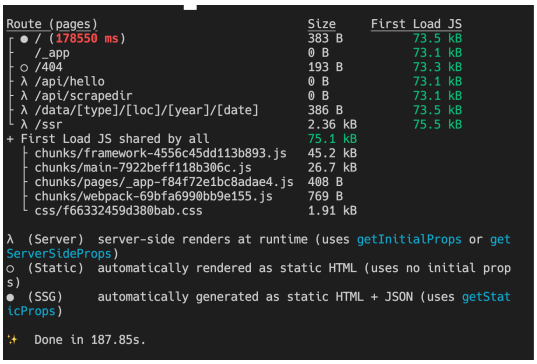

図 10　SG と SSR を併用した場合のビルドログ

実行結果からビルド時間は約 1847 秒であることが確認できる.

5.　考察

5.1　スクレイピングの実行結果について

スクレイピングの実行結果から, ディレクトリ情報の取得には波形データの取得よりも多くの時間が必要であることが明らかになった. これは, 波形データの取得が一度のページアクセスで済むのに対し, ディレクトリ情報の取得にはディレクトリ全体を探索する必要があり, その結果, 複数回のページアクセスが必要になるた

めである.
この結果は, スクレイピングの効率化に向けた改善の必要性を示唆している.

5.2 ビルドの実行結果について

ビルド時間の比較結果によれば, ISR を用いた場合のビルド時間は, SG と SSR を併用した場合の約 10 倍であることが確認できる. この結果は, ISR を用いる場合には, スクレイピング処理と静的ファイル生成のための時間が著しく増加することを示している. この差は, 特に大規模なサイトや頻繁に更新されるコンテンツを扱う際に, ビルドの効率性を考慮する上で重要な考慮点となる.

5.3 ビルド方法の選定

本研究での実装結果を踏まえ, アプリケーションのビルド方法の選定について考察する.
ISR の使用ではビルド時間に約 30 分かかることが確認された. ISR の定期再ビルド機能を考慮すると, 初回アクセス時のビルド時間が実質的なオーバーヘッドとなり, ユーザーが長時間待たされる可能性がある. これはユーザーエクスペリエンスの観点から見て適切ではない.
一方, SG と SSR を併用した場合, ページへの移行時に約 1 秒のスクレイピング時間が発生する. ISR と比較して多少の時間はかかるが, オーバーヘッドの問題を考慮すると, これは許容範囲内であると考えられる. SG の欠点として, 動的なページ生成が難しい点があるが, 本アプリケーションではスクレイピング元のサイトでデータの追加が 1 日に数回であるため, GitHub Actions などの CI/CD ツールを用いて定期的な再ビルドを行うことで対応が可能である.
ISR 単体または SG 単体の使用と比較して, CI/CD ツールを用いた再ビルドの方が, GitHub Actions の料金体系を考慮しても, 長いビルド時間によるコスト増加を抑えることができる.
したがって, これらの理由から本研究では SG と SSR を併用することが現状において最適な手法と結論づけた. この方法では, ユーザー体験, ビルドの効率性, コストのバランスを適切に取ることができる.

5.4 結果を踏まえた追加仮説

本実験の結果から, SG と SSR を併用する方法が適切であるとの結論に至ったが, この方法ではデータページへのアクセス時に SSR で行われるデータフェッチに約 1 秒かかることが UX 上の課題となる.
この課題に対し, プリフェッチ技術の活用によりさらなる高速化が可能であるという新たな仮説を立てた. プリフェッチとは, 次にアクセスする可能性が高いページのデータを事前に取得しておく技術である.
Next.js には NextLink 機能があり, 現在表示されているページ上の遷移リンクに関する情報をプリフェッチすることができる.
主要なユースケースとして, 以下のフローが考えられる.

1. データの検索ページにアクセスする.
2. データを検索し, リンクから任意のデータページにアクセスする.

この仮説では, データ検索ページで任意のデータページへの遷移リンクを表示した際にプリフェッチを行うことで, 実際にアクセスされた際のデータ取得よりも高速化が期待できる.
しかし, Next.js のドキュメントと有志の技術ブログによると, SSR 時に使用される getServerSideProps においては NextLink によるプリフェッチが実行されない. この場合, クライアントサイドでのデータ取得を行い, 次ページのプリフェッチを行う方がユーザー体験の速度向上に寄与する可能性がある.[9)]
さらに, useSWR という React Hooks ライブラリを使用することで, キャッシュ戦略とプリフェッチを最適化することも考えられる.[10)]
これらの仮説に対しては, 実際に実装して検証する必要があり, その価値は十分にあると考えられる. このような最適化は, ユーザー体験の向上に大きく寄与し, アプリケーションの全体的なパフォーマンスを高める可能性を秘めている.

6. おわりに

本研究では, 電離圏研究のための短波ドップラー観測データをスクレイピングし, 再描画するアプリケーションにおけるレンダリング戦略について調査と検証を行った.
研究結果を踏まえ, 適切な UI 実装を施すことにより, 高速でユーザーエクスペリエンス（UX）的に優れたアプリケーションの作成が可能であると確信している. さらに, 考察時に提出した追加仮説に関しても, 調査を行い, それらを導入することで, UX の向上という観点からさらに有益な実装が可能になると考えられる.
この研究は, データを効率的にスクレイピングし, ユーザーフレンドリーなインターフェースで提示するアプローチを示しており, 今後の電離圏研究をサポートするアプリケーション開発において重要な指針となるだろう.
現時点での実装成果としてアプリケーションの完成までには至らなかったが, 本研究で得られた知見・実装は目的の一つであった UX 向上に対しアプリケーションの高速化という点で一定の成果となったのではないかと考える.
今後の実装としては分担して行った実装を統合する必要がある. 実装完了後にユーザーからのフィードバックを得ることによって, より実際のユースケースに即した UX 向上につなげることができるだろう.

参考文献

1) 仲純平, 短波ドップラーデータ公開用 Web アプリケーションの開発, 特別研究公開発表会, 北九州工業高等専門学校, 2022
2) ソシオメディア株式会社 上野学 藤井幸多, オブジェクト指向 UI デザイン 使いやすいソフトウェアの原理, 技術評論社, 東京, 2020

3) デジタル庁, デザインシステム — デジタル庁, https://www.digital.go.jp/policies/servicedesign/designsystem/
4) 伊原力也 小林大輔 桝田草一 山本怜, Web アプリケーションアクセシビリティ 今日から始める現場からの改善, 技術評論社, 東京, 2023
5) Vercel, Next.js by Vercel - The React Framework, https://nextjs.org, (参照 2024-1-20)
6) Guillermo Rauch, 7 Principles of Rich Web Applications, https://rauchg.com/2014/7-principles-of-rich-web-applications, (参照 2024-1-20)
7) 手島拓也 吉田健人 高林佳稀, TypeScript と React/Next.js でつくる実践 Web アプリケーション開発, 技術評論社, 東京, 2022
8) オリジン間リソース共有 (CORS), https://developer.mozilla.org/ja/docs/Web/HTTP/CORS (参照 2024-1-20)
9) Next.js の prefetch に関して, https://taroodr.com/posts/nextjs-prefetch (参照 2024-1-20)
10) Vercel, SWR, https://swr.vercel.app/ja, (参照 2024-1-20)