

Next.js App Router 採用の振り返り

@Quramy 2024.04.30 #tng42

About me

- id: @Quramy (GitHub, X)
- 所属: 株式会社リクルート
(アプリケーションソリューション G)
- 職種: フロントエンドエンジニア



Agenda

- はじめに
- App Router でどのようにアプリケーションを作ったのか
- App Router に取り組む上で重要と思うこと

Agenda

- はじめに
- App Router でどのようにアプリケーションを作ったのか
- App Router に取り組む上で重要と思うこと

Next.js App Router is 何

- Next.js における新しいアプリケーション開発スタイルのこと
- 標準で React Server Components (以下 RSC) 及び Server Actions (以下 SA) が利用可能

App Router 採用に至った背景

- 対象案件: 結婚領域(ゼクシィをやっているところ)における新規事業
 - 同領域の他プロダクトでは Next.js Pages Router を使った開発を行っているため、App Router の使い心地を味見してみたい
→ Quramy も一枚噛ませてもらうことに
- 開発期間: 2023.6月 - 2023.12月
- フロントエンド規模: 約40ページ, 180 Component

アプリケーションのイメージ

諸般の事情によりキャプチャなどは非公開ですが、ある種の性格診断テストのようなプロダクトです

1. ユーザーは定められたいくつかの質問に対して回答を行う
2. ユーザーの回答に沿った性格診断結果が表示される

App Router に対する所感

2023.6 ～ 2023.12 までの約 6 ヶ月間の開発期間を通じた App Router に対する所感

- ポジティブ:
 - Data Fetching 関連について、従来よりも直感的な実装が可能である。
「BFF のための API エンドポイント」などの中間成果物にかかる設計・実装コストが削減される
 - チューニングを行っていない状態でも、十分に高速なアプリケーションを構築できた

App Router に対する所感

2023.6 ~ 2023.12 までの約 6 ヶ月間の開発期間を通じた App Router に対する所感

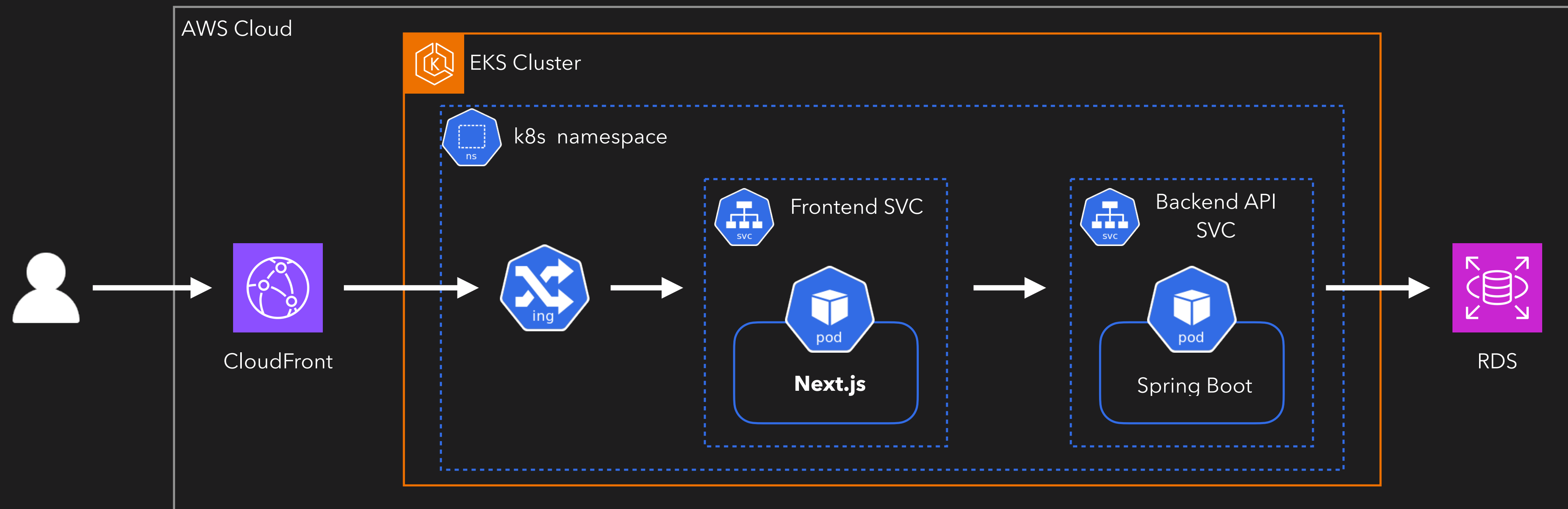
- ネガティブ:
 - 周辺エコシステム(e.g. Styling, UI Lib, Form lib, テスト)が RSC に追いついていない
- とはいえニーズはあるので、時間が解決していく類の話と思う

Agenda

- はじめに
- **App Router** でどのようにアプリケーションを作ったのか
- App Router に取り組む上で重要と思うこと

全体的なアーキテクチャ

- インフラ: AWS (EKS)
- Backend API: Spring Boot
- Frontend: Next.js (メインユースケースは SSR)



フロントエンドスタック

主な Dependencies

Data Fetching	RSC / Server Actions
HTTP Client	WHATWG Fetch (undici)
API Client	Open API Generator
スタイリング (CSS)	CSS Modules
UI ライブラリ	-
UI State Management	jotai
Form ライブラリ	-
Scaffolding	hygen
Component カタログ	@storybook/react
Unit Testing	Jest
Component Testing	Jest + RTL
E2E Testing	Playwright

フロントエンドスタック

Data Fetching

App Router の採用は Data Fetching に RSC / SA を用いることを意味している(逆にこれらを使わないのであれば、技術検証の意味が薄れてしまう)

- 「データを表示する」「データを更新する」のどちらであっても、SC / SA で完結する
- SWR や tRPC、Apollo Client のような Data Fetch ライブラリを不要とできるため、標準化が楽
- SC / SA とともに「React の使い方」の話でしかない。仮に Next.js でなかったとしても応用の効く知識となるはず

フロントエンドスタック

スタイリング(CSS)

Runtime CSS in JS 系が軒並み RSC で動作しないため、検討時点(2023.6月)で最も枯れていた CSS Modules を採用

- もしいま選択するのであれば Zero Runtime CSS in JS をもう少し検討していたかも
- ビルドツールチェーンが過渡期であることを考えると PandaCSS のような、トランスパイルに侵襲しないアプローチのものを選びたい

フロントエンドスタック

UI State Management

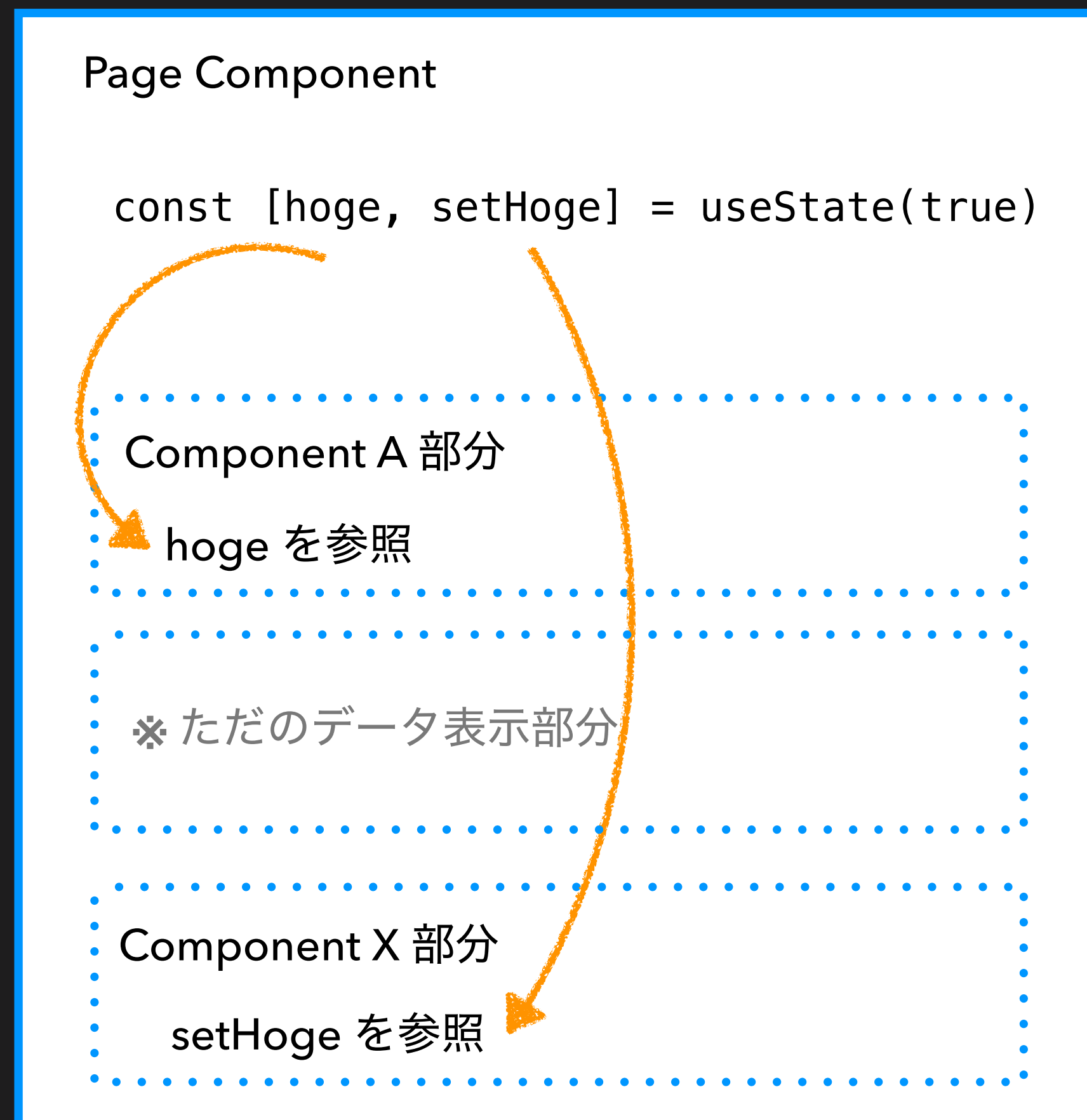
State management に jotai を採用

- RSC では小さく自立分散な CC 同士が相互に連携する世界になるのでは？
という予感があった
- 例: Component X が Viewport に交差時、Component A の表示状態が変化
- Pages Router の場合、X と A 両方を含んだ一個のComponent として記述していたりする => useState だけで何とかしてしまうこともできた
- App Router で X / A が CC として分散配置されると、そうはいかなくなるのでは？

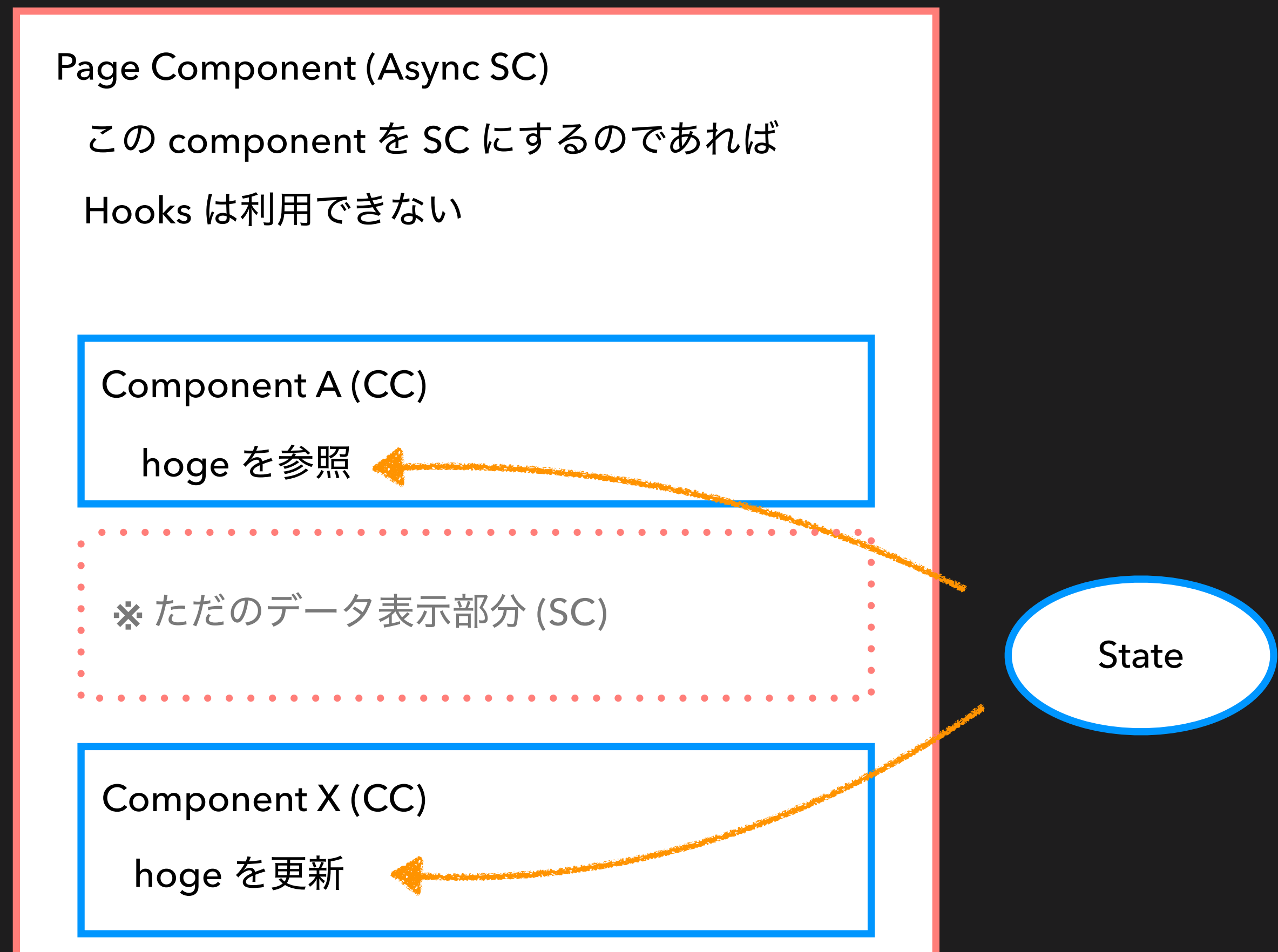
フロントエンドスタック

UI State Management

すべてが CC の世界



SC / CC を気にする世界



フロントエンドスタック

UI State Management

- jotai + Next.js を使う上での諸注意:
 - SSR 時に atomFamily がメモリリークするため、お掃除が必要

<https://jotai.org/docs/utilities/family#caveat-memory-leaks>

```
import { type Atom } from 'jotai'
import { atomFamily as atomFamilyiDelegate } from 'jotai/utils'

export function atomFamily<Param, AtomType extends Atom<unknown>>(<
  initializeAtom: (param: Param) => AtomType,
  areEqual?: (a: Param, b: Param) => boolean,
) {
  const delegate = atomFamilyiDelegate<Param, AtomType>(initializeAtom, areEqual)
  return new Proxy(delegate, {
    apply: (createAtom, _, [param]: [Param]) => {
      if (typeof document !== 'undefined') {
        return createAtom(param)
      }
      const atomItem = createAtom(param)
      queueMicrotask(() => createAtom.remove(param))
      return atomItem
    },
  })
}
```

フロントエンドスタック

その他

何も採用しなかった系統

- Form ライブラリ

- いわゆる react-hook-form などの系統

画面入力要件がラジオボタン/チェックボックスのみ。そもそもライブラリが不要なレベルだったため

- UI Component ライブラリ

- 選定している余裕がなかったという消極的な理由

RSC と相性の良い UI Component ライブラリがあれば知りたいです！

パフォーマンス/キャッシュ戦略

Next.js で避けて通れないのがキャッシュの話題

Mechanism	What	Where	Purpose	Duration
Request Memoization	Return values of functions	Server	Re-use data in a React Component tree	Per-request lifecycle
Data Cache	Data	Server	Store data across user requests and deployments	Persistent (can be revalidated)
Full Route Cache	HTML and RSC payload	Server	Reduce rendering cost and improve performance	Persistent (can be revalidated)
Router Cache	RSC Payload	Client	Reduce server requests on navigation	User session or time-based

<https://nextjs.org/docs/app/building-your-application/caching#overview>

パフォーマンス/キャッシュ戦略

Next.js で避けて通れないのがキャッシュの話題

Mechanism	What	Where	Purpose	Duration
Request Memoization	Return values of functions	Server	Re-use data in a React Component tree	Per-request lifecycle
Data Cache	Data	Server	Store data across user requests and deployments	Persistent (can be revalidated)
Full Route Cache	HTML and RSC payload	Server	Reduce rendering cost and improve performance	Persistent (can be revalidated)
Router Cache	RSC Payload	Client	Reduce server requests on navigation	User session or time-based

オプトアウト

<https://nextjs.org/docs/app/building-your-application/caching#overview>

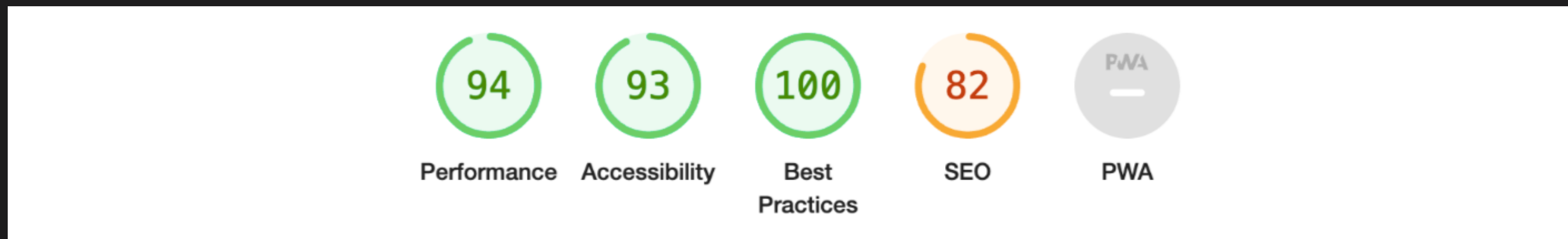
パフォーマンス/キャッシュ戦略

なぜ Next.js の Cache を利用していないか:

- アプリケーション特性上、Data / Full Route Cache に対する hit rate が高くないことがわかっており、キャッシュを頑張る旨味が少なかった
- EKS Pod 間で Cache を共有する仕組みについて未検討
- デフォルトでは Data Cache, Full Route Cache はファイルシステム上に永続化される。すなわち Container Local な データとなる
このため、 Custom Cache Handler + ElastiCache などを使って、永続化方式を AWS フレンドリなものに変更する必要がある
- <https://nextjs.org/docs/app/api-reference/next-config-js/incrementalCacheHandlerPath>

パフォーマンス/測定

Next.js Cache は消極的であっても、そこそこ実用的なパフォーマンス



同領域での Pages Router なプロダクトと比較すると、やはり First Load
JavaScript のボリュームに圧倒的な差が出る

(App Router 案件は依存ライブラリが少ないため、「それはそう」とも言えるけど)

- App Router 案件: 約 200 kB
Pages Router 案件: 約1,720 kB

テスト

小規模かつ新規開発案件であったため、テストはそこそこの規模に留めた

構築した自動テストは以下の 2 種類

- 粒度の細かいテスト: jest + RTL
- 粒度の粗いテスト: Playwright

細粒度なテスト

- 全ての Component で Storybook の ファイル (.stories) 作成を強制しておく
- RSC のテスト方法について:
 - 非同期な SC については、Container / Presentation で分割しておき、Storybook や jest で実行されるのは Presentational な部分のみにしておく
 - ※ 詳細: [React Server Component のテストと Container / Presentation Separation](#)
 - すべての Story に対して、jest + RTL で composeStories & render の動作保証をしておく
- Storybook 8 の experimentalRSCは使っていない
 - ※ 開発当初は Storybook 7.x だったというのもあるが、SC を 無理やり気味に CC として扱っていること自体がイケてないと思っているため

粗粒度なテスト

- Playwright による E2E を実施
 - DB の Seed + Spring Boot の起動 + next start をすべて CI で Service 実行するようにしている
 - Pros: E2E としては正しい姿
 - Cons: CI が重たい. SA を一度実行してしまうと DB の状態が変更されるため、Flaky を踏んでも Playwright で同一 URL にリトライしても意味が無い
- Next.js の Experimental Test Proxy は利用していない

粗粒度なテスト

- Next Server の挙動と密に連動する箇所は E2E 側で動作を担保するようにしている
- テストシナリオの例:
 - 前提: SA で revalidatePaths を用いて Router Cache をパージしている
 - シナリオ: ページ表示 -> SA 実行して画面遷移 -> ブラウザバック
 - 確認観点: ブラウザバック後の表示では Router Cache が最新化されているか？

Agenda

- はじめに
- App Router でどのようにアプリケーションを作ったのか
- **App Router** に取り組む上で重要と思うこと

App Router に取り組む上で重要と 思うこと

- App Router 案件を通して少しずつ理解してきたこと達:
 - RSC のメンタルモデル
 - データフェッチの粒度
 - HTML / CSS 知識の重要性

RSC のメンタルモデル

まずはフロントエンドの開発者全員が RSC に脳をアップデートすべき

- RSC (App Router) は Pages Router での SSR と本質的に意味が異なる
- 「SC は事前計算である」を叩き込んでおく
- 参考になるもの
 - uhyo さんの記事: <https://zenn.dev/uhyo/articles/react-server-components-multi-stage>

データフェッチの粒度

- Pages Router 時代に「事前処理」に相当していたもの:
getServerSideProps or getStaticProps
- これらは、Page 単位の Component にしか作用できなかった
- → gSSP で利用する API の設計がユースケースに引きずられる

データフェッチの粒度

- App Router では SC がデータの取得という事前処理を担っている
 - データ取得が page 単位である必要が無い. 末端の Component で、その Component が必要なデータを直接フェッチできる
- リソース志向に基づいた API 設計と親和する
- 細粒度 API としておいた方が、Next.js の Data Cache にも載せやすいはず

HTML / CSS の知識

- HTML + CSS の知識は unnecessary CC を削減する上でも重要
- 例: カルーセルっぽい見た目のコンポーネント
 - CC の JavaScript 書かなくても、CSS の scroll-snap で実現できますよね？
- <https://web.dev/blog/interop-2022>

Thank you !