



パブリックから半年
Remix使い倒してみた

@aiji42_dev



Who am I ?

Uejima Aiji | @aiji42_dev


- 🏢 株式会社エイチームライフデザイン
- 🧘 リードエンジニア
- 🥑 最近の活動
 - CloudflareでISRを実現したり
 - CSRなサイトをPrerenderでSSRぽくしたり
 - PrismaからSupabase APIを叩くミドルウェア作ったり
 - 社内でフロントエンド版ISSUCON開催したり

今日はRemixを紹介したい

The word "Remix" is displayed in a bold, sans-serif font. Each letter is a different color, creating a rainbow gradient: 'R' is blue, 'e' is green, 'm' is yellow, 'i' is purple, and 'x' is red. The letters have a soft, glowing effect around them, making them stand out against the dark background.

Remix

What is Remix ?

- **React SSRフレームワーク**
- **React Routerの開発チームが開発を主導**
- **昨年11月末にv1がリリースされたタイミングでパブリックに**
-  **のアイコンがよく使われる**

Remixの特徴 ①

loader と action

```
// app/routes/posts/$slug.tsx
export const loader = async ({ params }) => {
  const post = await db.post.findUnique({
    where: params.slug
  })

  return { post }
}

export const action = async ({ request, params }) => {
  if (request.method === 'POST') {
    await db.post.create({ ... })
  }
  if (request.method === 'DELETE') {
    await db.post.delete({ ... })
  }
  if (request.method === 'PATCH') {
    await db.post.update({ ... })
  }
}

const Page: FC = () => {
  const { post } = useLoaderData()
  return ...
}

export default Page
```

loaderとaction

Next.js の `getServerSideProps` や `API Routes` のようなもの

ページコンポーネントと同一ファイルに定義可能

loader

GETアクセス時のデータフェッチを定義

ページ(コンポーネント)からは`useLoaderData`で取得し、
`useFetcher`で再フェッチ可能

action

POSTやDELETEなどのミューテーションを定義する

`useSubmit`や`useFormAction`、form要素からリクエストする

Remixの特徴 ②

File system routing と Nested Routing (Layout)

レイアウトルート / 共通処理

```
app/
├── routes/
│   ├── blog/
│   │   ├── $postId.tsx
│   │   ├── categories.tsx
│   │   └── index.tsx
│   ├── about.tsx
│   ├── blog.tsx
│   └── index.tsx
└── root.tsx
```

ディレクトリ構成やファイル名がそのままURLになるという点は、Next.jsのpagesとよく似ている


```
app/  
├── routes/  
│   ├── blog/  
│   │   ├── $postId.tsx  
│   │   ├── categories.tsx  
│   │   └── index.tsx  
│   ├── about.tsx  
│   ├── blog.tsx  
│   └── index.tsx  
└── root.tsx
```

URL

Matched Route

/

app/routes/index.tsx

/about

app/routes/about.tsx

```
app/
├── routes/
│   ├── blog/
│   │   ├── $postId.tsx
│   │   ├── categories.tsx
│   │   └── index.tsx
│   ├── about.tsx
│   ├── blog.tsx
│   └── index.tsx
└── root.tsx
```

ディレクトリの入れ子そのままURLに変換される

\$(ドルマーク)をつけると、
パラメータとしてloader/actionで扱える

URL	Matched Route
/blog	app/routes/blog/index.tsx
/blog/categories	app/routes/blog/categories.tsx
/blog/my-post	app/routes/blog/\$postId.tsx

```
app/  
├── routes/  
│   ├── blog/  
│   │   ├── $postId.tsx  
│   │   ├── categories.tsx  
│   │   └── index.tsx  
│   ├── about.tsx  
│   ├── blog.tsx  
│   └── index.tsx  
└── root.tsx
```

root.tsxがトップレイヤレイアウト
ディレクトリと同一名ファイルが子レイアウト

URL	Matched Route	Layout
/	app/routes/index.tsx	app/root.tsx
/about	app/routes/about.tsx	app/root.tsx
/blog	app/routes/blog/index.tsx	app/routes/blog.tsx
/blog/categories	app/routes/blog/categories.tsx	app/routes/blog.tsx
/blog/my-post	app/routes/blog/\$postId.tsx	app/routes/blog.tsx

```
app/
├── routes/
│   ├── __authenticated/
│   │   ├── dashboard.tsx
│   │   └── $userId/
│   │       └── profile.tsx
│   └── __authenticated.tsx
└── root.tsx
```

ダブルアンダースコアで始めると
URL化されないレイアウトルートになる
(pathless layout routes)

```
app/
├── routes/
│   ├── blog.tsx
│   └── blog.$slug.tsx
└── root.tsx
```

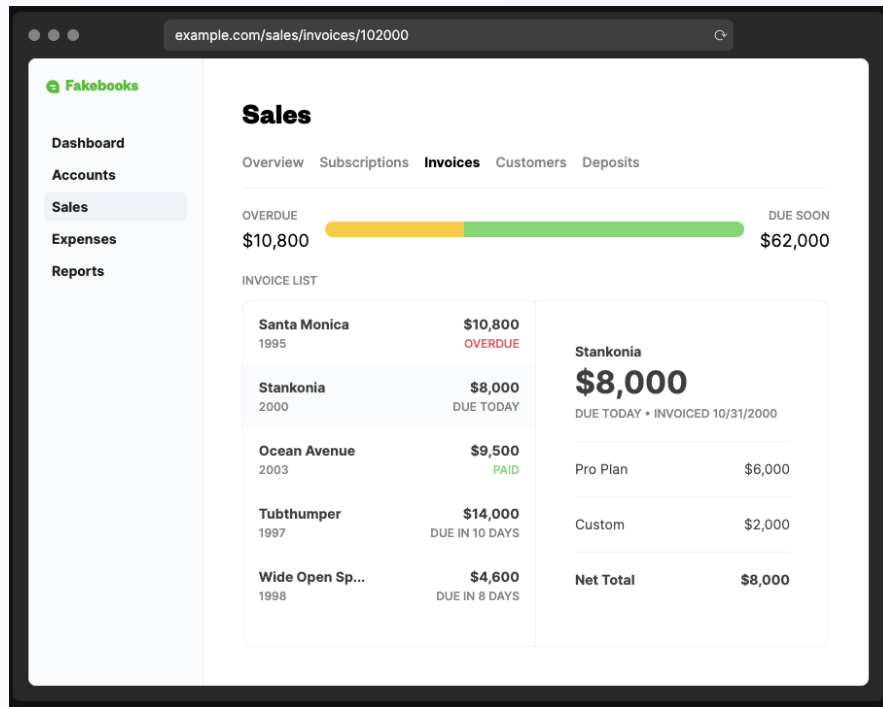
ディレクトリ構造の代わりにドットでも表現可能

```
app/
├── routes/
│   ├── blog/
│   │   ├── $postId.tsx
│   │   ├── categories.tsx
│   │   └── index.tsx
│   └── $.tsx
└── root.tsx
```

Catch all route (*)

ドキュメントで紹介されている例

```
app/  
├── routes/  
│   ├── sales/  
│   │   ├── invoices/  
│   │   │   └── $id.tsx  
│   │   ├── invoices.tsx  
│   └── sales.tsx  
└── root.tsx
```



<https://example.com/sales/invoices/102000>

こんな感じのダッシュボード

```

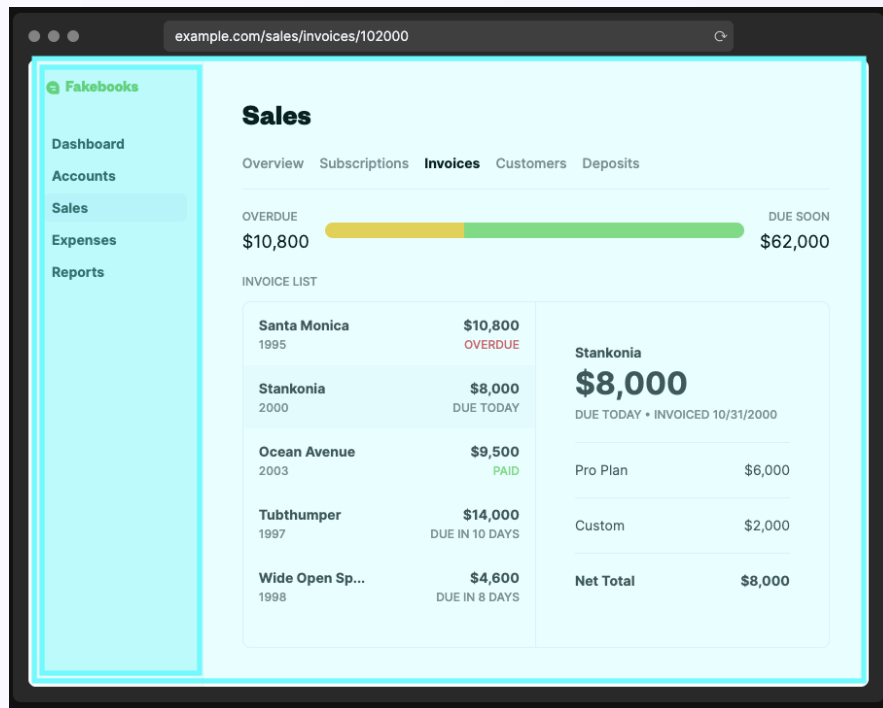
app/
├── routes/
│   ├── sales/
│   │   ├── invoices/
│   │   │   └── $id.tsx
│   │   ├── invoices.tsx
│   └── sales.tsx
└── root.tsx

```

```

export default function Root() {
  return (
    <Sidebar>
      <Outlet />
    </Sidebar>
  )
}

```



root.tsxがトップレイアウト

Outletコンポーネント部分がレンダリング時に
子レイアウト・子ページになる

```

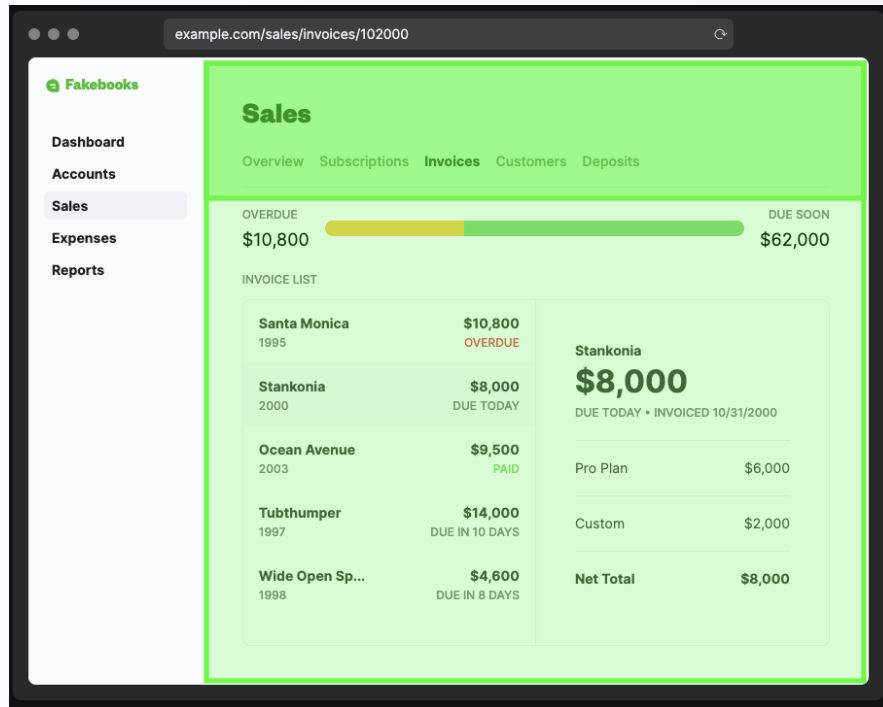
app/
├── routes/
│   ├── sales/
│   │   ├── invoices/
│   │   │   └── $id.tsx
│   │   └── invoices.tsx
│   └── sales.tsx
└── root.tsx

```

```

export default function Sales() {
  return (
    <>
      <h1>Sales</h1>
      <Tabs />
      <Outlet />
    </>
  )
}

```



ディレクトリと同一名ファイルで子レイアウトを定義

```

app/
├── routes/
│   ├── sales/
│   │   ├── invoices/
│   │   │   ├── $id.tsx
│   │   │   └── invoices.tsx
│   │   └── sales.tsx
│   └── root.tsx

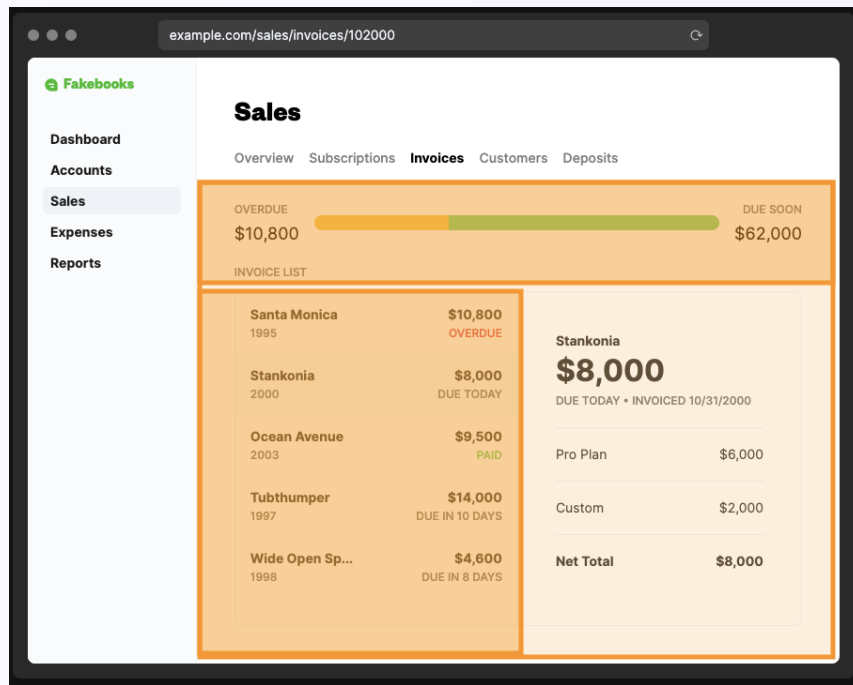
```

```

export const loader = async () => {
  const overview = await fetch(...).then((res) => res.json())
  // ...
  return { overviewData, invoiceListData }
}

export default function InvoiceList() {
  const { overviewData, invoiceListData } = useLoaderData()
  return (
    <>
      <Overview data={overviewData}>
      <InvoiceList items={invoiceListData} >
        <Outlet />
      </InvoiceList>
    </>
  )
}

```



layoutにもloaderを設置可能


```

app/
├── routes/
│   ├── sales/
│   │   ├── invoices/
│   │   │   ├── $id.tsx
│   │   │   ├── invoices.tsx
│   │   └── sales.tsx
└── root.tsx

```

```

export const loader = async ({ params }) => {
  const data = await db.invoice.findOne({ where: { id: params.id } })
  return { data }
}

```

```

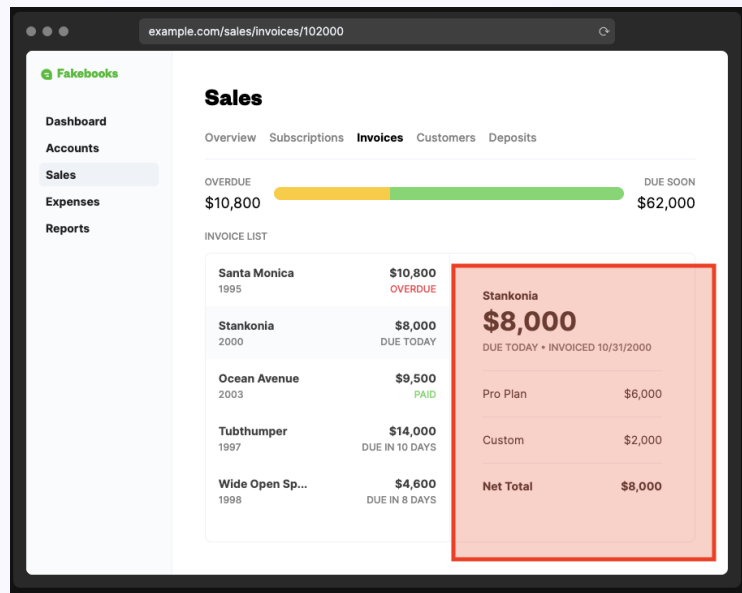
export default function Invoice() {
  const { data } = useLoaderData()
  return (
    <InvoiceItem data={data} />
  )
}

```

```

export function ErrorBoundary({ error }) {
  return (
    <ErrorMessage>{error.message}</ErrorMessage>
  )
}

```



各ページ・レイアウトごとに
ErrorBoundaryを定義可能

```

app/
├── routes/
│   ├── sales/
│   │   ├── invoices/
│   │   │   ├── $id.tsx
│   │   │   ├── invoices.tsx
│   │   └── sales.tsx
└── root.tsx

```

```

export const loader = async ({ params }) => {
  const data = await db.invoice.findOne({ where: { id: params.id } })
  return { data }
}

```

```

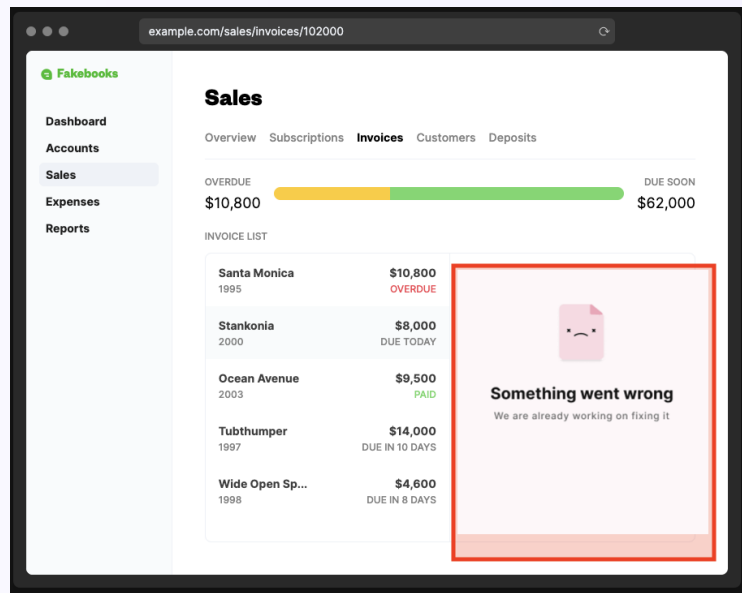
export default function Invoice() {
  const { data } = useLoaderData()
  return (
    <InvoiceItem data={data} />
  )
}

```

```

export function ErrorBoundary({ error }) {
  return (
    <ErrorMessage>{error.message}</ErrorMessage>
  )
}

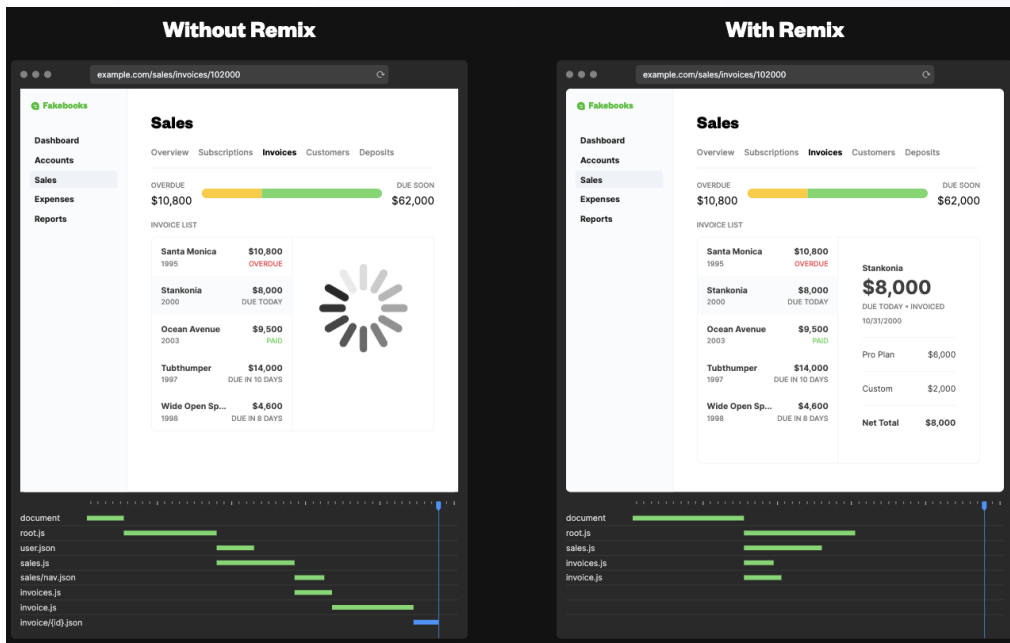
```



エラーの伝搬を留めることができる
フォールバックが最小限になる

Nested Routes があると何が嬉しいか

- レイアウトのグルーピングと階層適応
- 並列データフェッチ
- 各loaderは並列に処理されるため、高速化につながる



- **ロジックの共通化**

- 特定ルート配下はログイン必須にするなどの、共通処理を階層的にもたせられる
- pathless routesと組み合わせて特定のディレクトリ下は暗黙的に認証必須にするなど

- **差分ロード・差分再フェッチ**

- ナビゲーション時にフルページロードではなく、必要なレイアウトルート分のロードが行われる
- 任意にページ内を更新する再フェッチ処理も実装しやすい

ちょうど先日**Next.js**にも同等な機能の**RFC**が公開された

Layouts RFC

Tuesday, May 24th 2022 (about 15 hours ago)



Tim Neutkens
@timneutkens



Sebastian Markbåge
@sebjmarkbage



Delba de Oliveira
@delba_oliveira



Lee Robinson
@leerob

<https://nextjs.org/blog/layouts-rfc>

- **現プロポーザルではおおよそRemixと同等の機能をカバーする予定**
 - pathlessやErrorBoundaryに関しても、ドキュメントにはないが「パート2で言及する」とのこと
 - かなりRemixのノウハウが意思決定に影響を及ぼしている印象
 - デフォルトでServerComponentになる (Remixでも同様に議論は起きている)

Remixの特徴 ③

マルチランタイム

- Node / web worker / Deno
- Vercel / Netlify / Cloudflare Workers • Pages, etc
- もちろんセルフホスト可能



プラットフォームに依存しないという点で、Next.jsとVercelの関係と比較される事が多い

ただし、RemixはSSRだけしかサポートしていないので単純比較は正直フェアではない
(Next.jsもSSRだけならプラットフォームには依存しないので)

Remixの一番の強みは Cloudflare Workers 上で動くという点

だと個人的には思う。

低コスト・例レイテンシでリアルタイムにコンテンツをデリバリできるのが強み

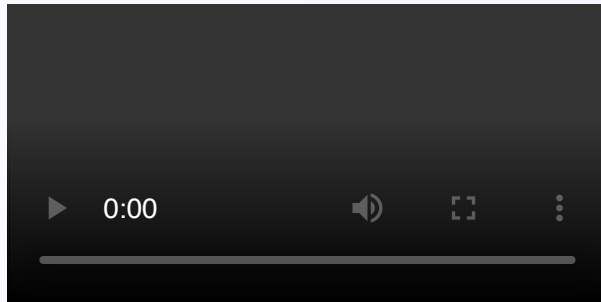
裏を返せば静的なページをデリバリするのであれば Remix は向いていない

Remixの特徴 ④

Formとヘルパー

前述のactionと通信を行うための、Formコンポーネントや多数のヘルパーを備えている

特に **useTransition** はフォームのsubmitの状態 (idle / submitting / loading)を管理したり submit中のデータを取り扱うことが可能なので、楽観的UIの実装も容易




```
import { ValidatedForm } from "remix-validated-form";
import { withZod } from "@remix-validated-form/with-zod";

export const validator = withZod(
  // your zod role
);

export const action = async ({ request }) => {
  const result = await validator.validate(await request.formData());
  if (result.error) return validationError(result.error);

  const { firstName, lastName, email } = result.data;
  // Do something with the data
};

export default function MyPage() {
  return (
    <ValidatedForm validator={validator} method="post">
      <FormInput name="firstName" label="First Name" />
      <FormInput name="lastName" label="Last Name" />
      <FormInput name="email" label="Email" />
      <SubmitButton />
    </ValidatedForm>
  );
}
```

remix-validated-form

<https://www.remix-validated-form.io/>

remix-validated-formとzodを使用するとactionとコンポーネントを一体化できる

クライアントとサーバとでバリデーションを共通化できるだけでなく

エラーメッセージの返却・レンダリング、例外処理の実装から開放される

別のデータソースと通信して別途バリデーションするなど、サーバサイドオンリーなバリデーションもかんたんに追加可能

Remixの特徴 ⑤

Cookieヘルパー

CookieやSessionを取り扱うためのヘルパーが標準装備

シリアライズ&検証の機能もデフォルトで実装されている

loader/actionと組み合わせることでこれまでフロントに実装していた、
状態管理や認証などをサーバサイドへ移譲できる

ロジックがフロントに露出しないため、
秘匿性の高い情報の漏洩防止や、バンドルサイズの軽減につながる

```
export const loader = async ({ request }) =>
  supabaseStrategy.checkSession(request, {
    successRedirect: '/private'
  });

export const action = async ({ request }) =>
  authenticator.authenticate('sb', request, {
    successRedirect: '/private',
    failureRedirect: '/login'
  });

export default function LoginPage() {
  return (
    <Form method="post">
      <input type="email" name="email" />
      <input type="password" name="password" />
      <button>Sign In</button>
    </Form>
  );
}
```

remix-auth

<https://github.com/sergiodxa/remix-auth>
(サンプルコードはremix-auth-supabase)

認証方法・認可ルール・フォールバックルールなどを簡単に設定・制御できる

クライアント側には一切ロジックが露出しない

実際実用に耐えられるの？ 🤔



Aiji Uejima
@aiji42_dev



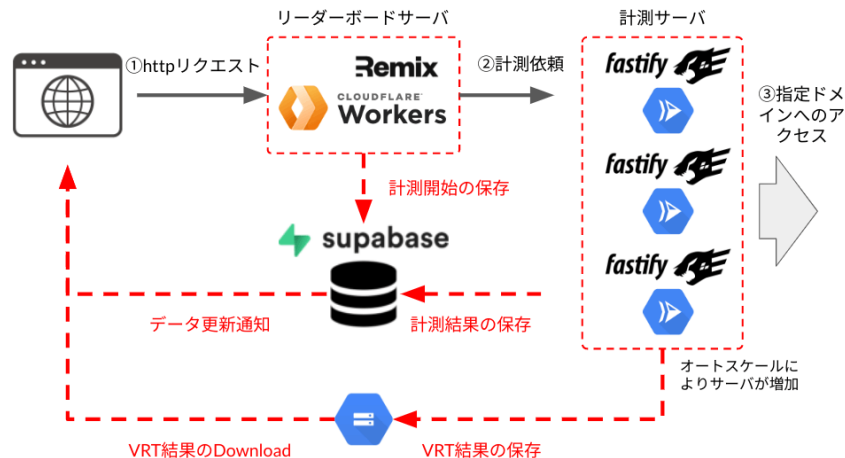
先日社内のエンジニア・デザイナー(総勢80人強)でフロントエンド版Isuconを開催しました。

運営である自分はRemix、Supabase、CloudflareWorkersでリーダーボードを作成したのですが、トラブルなくすんなり同時接続を捌いたのを見て、上記構成のポテンシャルを肌で感じました。

Web Speed Hackathon 2021 をISUCONぽく開催する技術

@chimame

Qiita



良かったところ

- **ステート管理ライブラリが不要**
 - 前述の通り
- **認証ロジック・データフェッチロジックすべてがサーバサイド簡潔**
 - クライアントの実装はデータの描画のみに集中できる
- **情報更新をきめ細かくリアルタイムに、かつ高速に**
 - 前述の例ではSupabaseのsubscribeと組み合わせて、DBに変更が加わったらスタッツを再フェッチする
 - ネストされた部分的なレイアウトのみ再フェッチ(フルページロードしない)
 - 実際のデータフェッチはloader側で行うので、ロジックは一切露出しない

- **エッジレンダリング(Cloudflare Workers)による恩恵**

- ゼロコールドスタート
 - KV使わなくても、200-300msで応答できる(TTFB)
 - KV使えばSSRで100ms切る
 - 同一構成の Next.js on Vercel のSSRで300-500ms
 - もちろんデータソースに引っ張られるが
- スケールを気にしなくて良い

苦しみ

- **Nested Routeは想像以上に難易度が高い**
 - URL設計とディレクトリ設計をセットで行わなければならない
 - 良くも悪くもロジックが分散し、脳内メモリを圧迫する
 - 実際ネストは2階層くらいにとどめておくのがよい
 - 最初のサンプルに上げたような構成は正直無理

Workersに限った話になるが。。。

- **UIライブラリ入ると1MBにバンドルサイズを抑えるのは結構きつい**
 - SSRなので全てバンドルしないといけない(チャンクもできない)
 - Service Bindingsを駆使して回避した
- **まだまだWorker非対応なライブラリが多い**
 - esbuildでpolyfillしたり、injectしたりなど職人芸が求められる
 - しかし、そもそもRemixのコンパイラ(esbuild)の設定の拡張が不可能
 - next.config.jsからwebpackの設定をいじるみたいな感じのことはできない
 - 拡張自体がコミュニティのポリシー的にNG (マネジメントコストとリスクの問題)
 - 何度もDiscussionやPRは起票されているがことごとくリジェクト
 - 最終的に自分でRemixのesbuild を拡張可能にするプラグインを書いた
 - <https://github.com/aiji42/remix-esbuild-override>

初心者の方はランタイムNodeからスタートすることをオススメします

相性の良いサービスやサイト

下記ケースならNext.jsで実装するよりもRemixで実装したほうが開発コストは小さくなる(と個人的には思う)

- **React Routeで構築されたSPAサイトをSSRに移行したい、SEO対策したい**
 - React Routeの思想をベースに構築されているため、大きく構造を変えずに移行できる
- **複数ページに渡るエントリーフォームを簡単に実装したい**
 - 前述の通り、remix-validated-formとzodで簡単に作れる
 - ステートライブラリも不要
 - MVCなWebフレームワーク(Rails)使ってた人は比較的とっつきやすい(と個人的には思う)
- **管理画面やダッシュボードをフルスクラッチしたい**
 - React Adminの導入を試みたが、適したアダプターがなかったなど
 - Nested Routes と remix-validated-form, remix-auth を駆使する
 - loader/actionを各フォームやデータフィールドと1対1で配置し、UIとデータロジックをそれぞれで凝縮

まとめ

なんと言っても

- **Cloudflare Workerで動く**
- **Nested routesの先駆け**
 - レイアウトとデータフェッチロジックの分散管理
 - Next.jsに影響を及ぼす程の先見性

この2つを備えていて、ここまで完成度の高いフレームワークは他にはありません。
なので、十分に試す価値はあると思います。

また、Next.jsの新しいLayout戦略がGAされるまでの素振りとしても良いと思います。

One more thing

Next.jsにRemixのエッセンスを取り入れる

```
import { handle, json, Form, useFormSubmit } from 'next-runtime';

export const getServerSideProps = handle({
  async get() {
    return json({ name: 'smeijer' });
  },
  async post({ req: { body } }) {
    await db.comments.insert(body);

    return json({ message: 'thanks for your comment!' });
  },
});

export default function MyPage({ name, message }) {
  const { isSubmitting } = useFormSubmit();

  if (message) return <p>{message}</p>;
  return (
    <Form method="post">
      <input name="name" defaultValue={name} />
      <input name="message" />
      <button type="submit" disabled={pending}>
        {isSubmitting ? 'submitting' : 'submit'}
      </button>
    </Form>
  );
}
```

next-runtime

<https://next-runtime.meijer.ws/getting-started/1-introduction>



getServerSidePropsを拡張し、リクエストメソッドごとに実装を書き分けることができる
フォームのアクションをapi routesにせず、自パスに向ければ、擬似的なloader/actionになる

楽観的UIのためのヘルパーやCookieを取り扱うヘルパーなどが用意されており、かなりRemixに似ている

というか、Remixをインスパイアされて実装したと作者がドキュメントで明言している

こちらの記事でも紹介しています

Remixのコンセプトの一部を Next.jsに導入する(next-runtime)



aji42



<https://zenn.dev/aji42/articles/23a88a7b111694>