

Accelerating React applications with Server-Side React Frameworks



With a look at Next.js, Remix.run and React 18

Ken Rimple
Philly Emerging Technologies for the Enterprise
April 20, 2022

Who am I?

Ken Rimple

Director of Training/Mentoring, Chariot Solutions,
Consultant for 30+ years
Have seen a lot of tools come and go...

Musician and Father of 4 (sorry about the
interstitial two-chord thing, that was my fault)

I wrestle JS to the ground, then it hits me with a
lawn chair when I get up and walk away...



Another framework? That's it, I'm gonna skydive!

Attendee assumptions...

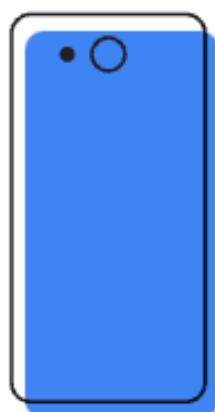
Maximum enjoyment potential if you have:

- Understanding of functional React and hooks
- Understanding of Promises with `then()` and `async/await`
- Experience using React and client-side APIs to fetch data

When is vanilla React not enough?

Do any of these scenarios fit?

- Static or close-to-static content but need React activity
- Existing app sluggish - data loading issues / initialization complexity, monolith
- Clients with slow networks, high latency



The probability of bounce **increases 32%** as page load time goes **from 1 second to 3 seconds.**

source: <https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/page-load-time-statistics/>

The platforms we'll discuss today

Next.js and Remix.run...



&



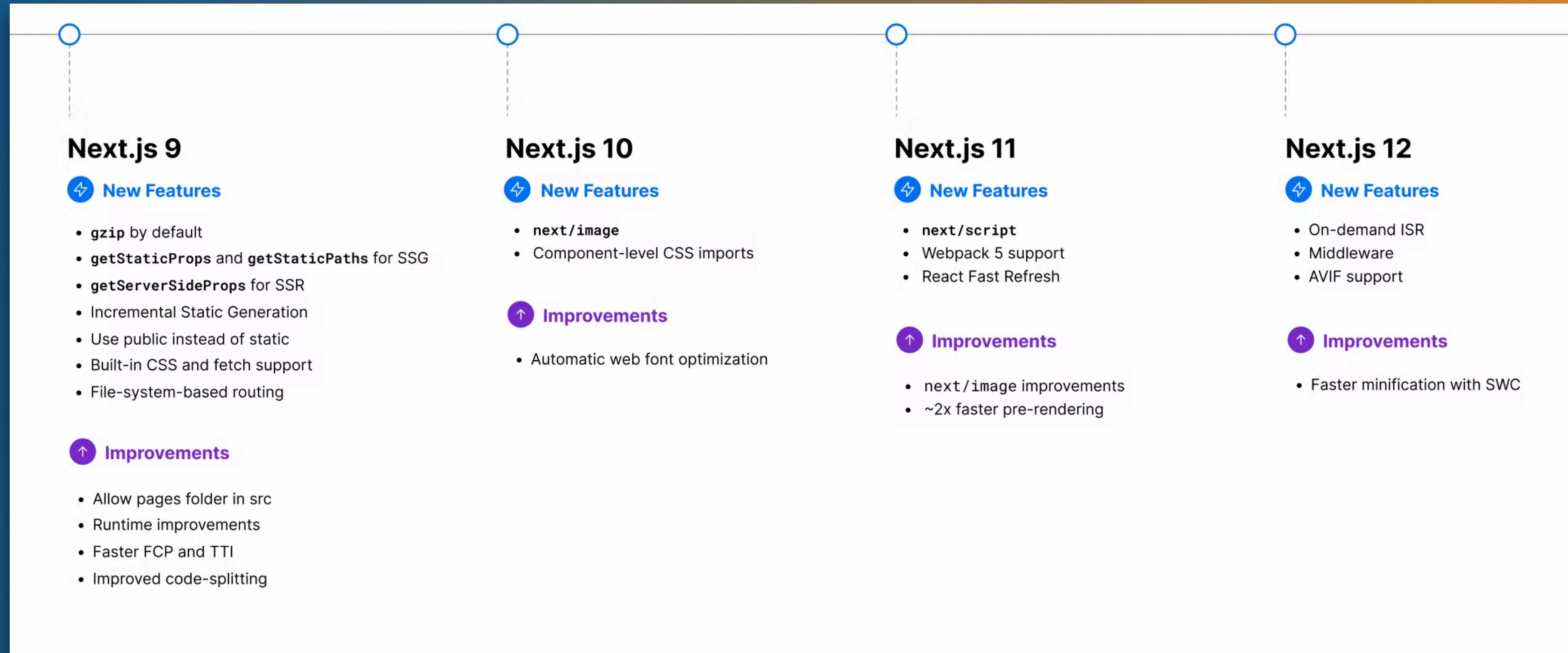
What is Next.js?

Next.js is a React-driven Framework

...with a focus on performance. Features:

- Server Side Rendering (SSR) - executing data fetches on the server before rendering the component
- Static Site Generation (SSG) - pre-renders static content to serve based on data APIs
- Incremental Static Regeneration (ISR) - update view of generated data once it becomes "stale"
- SWR - a query library for client caching, reloading of data after the page loads
- next/router - a file-based router supports both server and client-side routing

Next.js Version History



Source: <https://vercel.com/blog/upgrading-nextjs-for-instant-performance-improvements>

Remix.run, the newcomer



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE

Remix Philosophy - use the web!

Focusing on HTTP-based standards:

- Server-Side Rendering - standard browser Web Fetch APIs!
- A file-based router - file-based routes similar to Next.js
- Remix supports progressive enhancement
 - This means JS can be disabled and forms can be completely server-driven



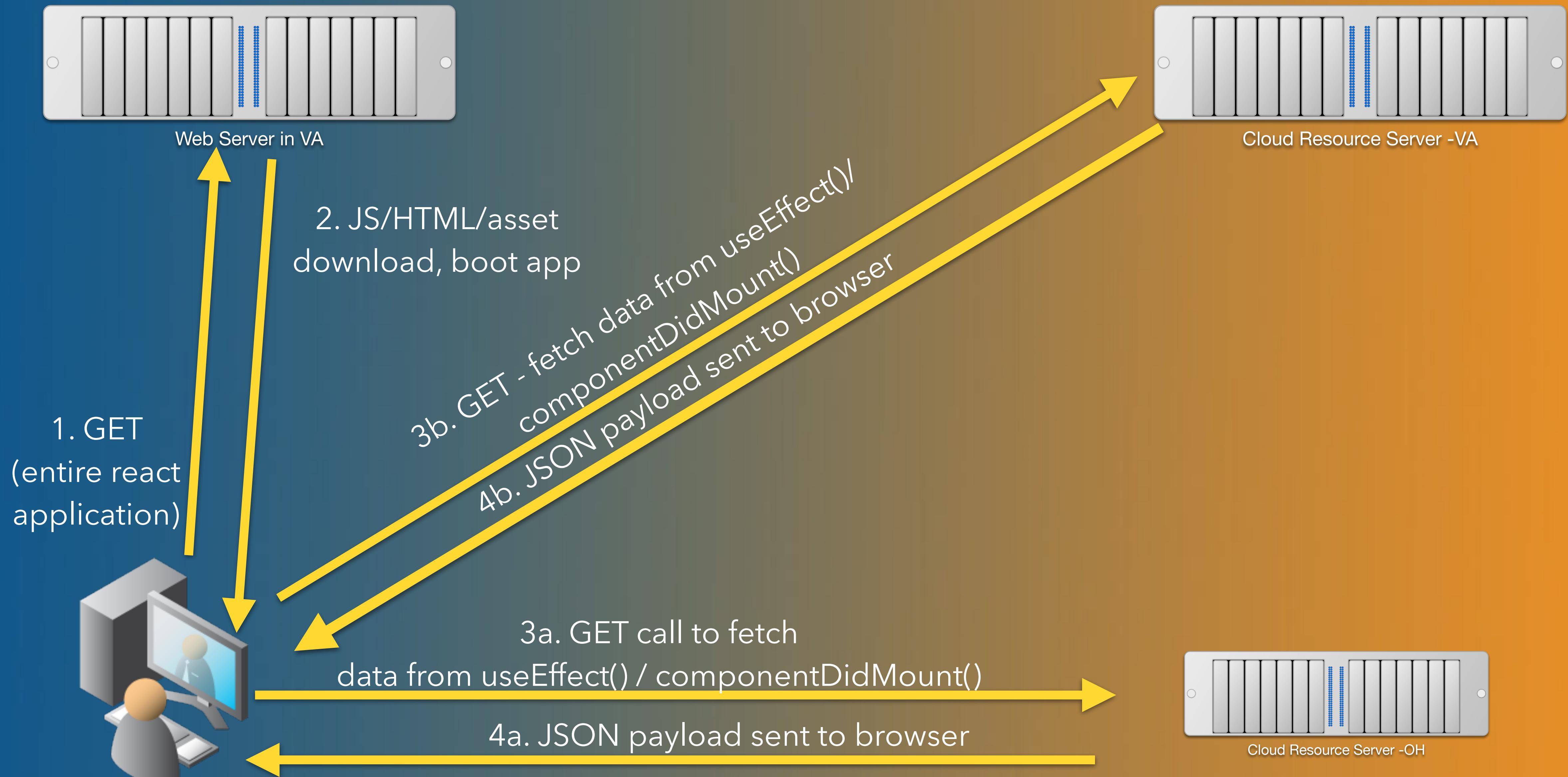
Remix.run Philosophy



The Remix philosophy can be summed up in four points:

1. Embrace the server/client model, including separation of source code from content/data.
2. Work with, not against, the foundations of the web: Browsers, HTTP, and HTML. It's always been good and it's gotten *really good* in the last few years.
3. Use JavaScript to augment the user experience by emulating browser behavior.
4. Don't over-abstract the underlying technologies

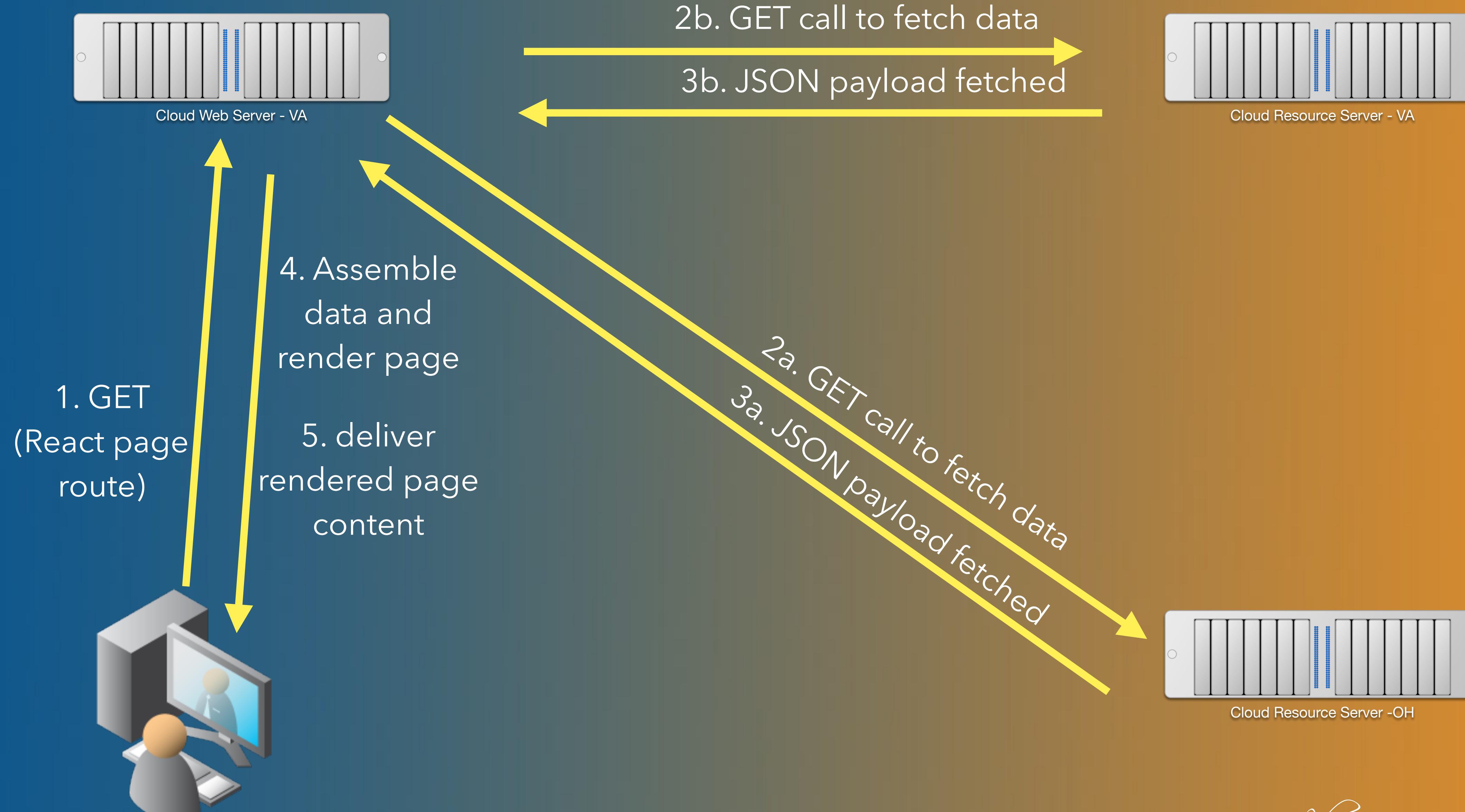
Client Fetching with React - onus on client



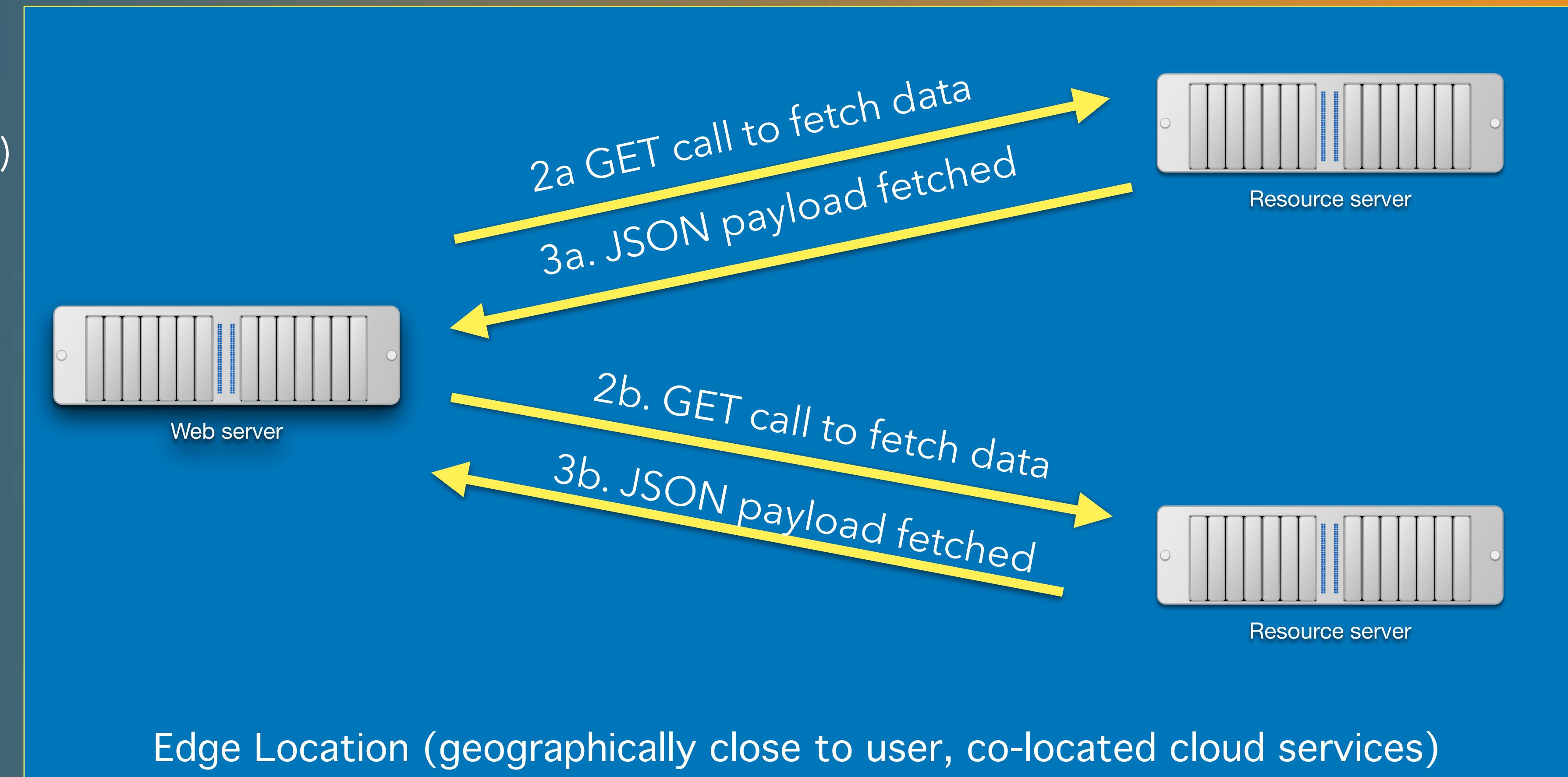
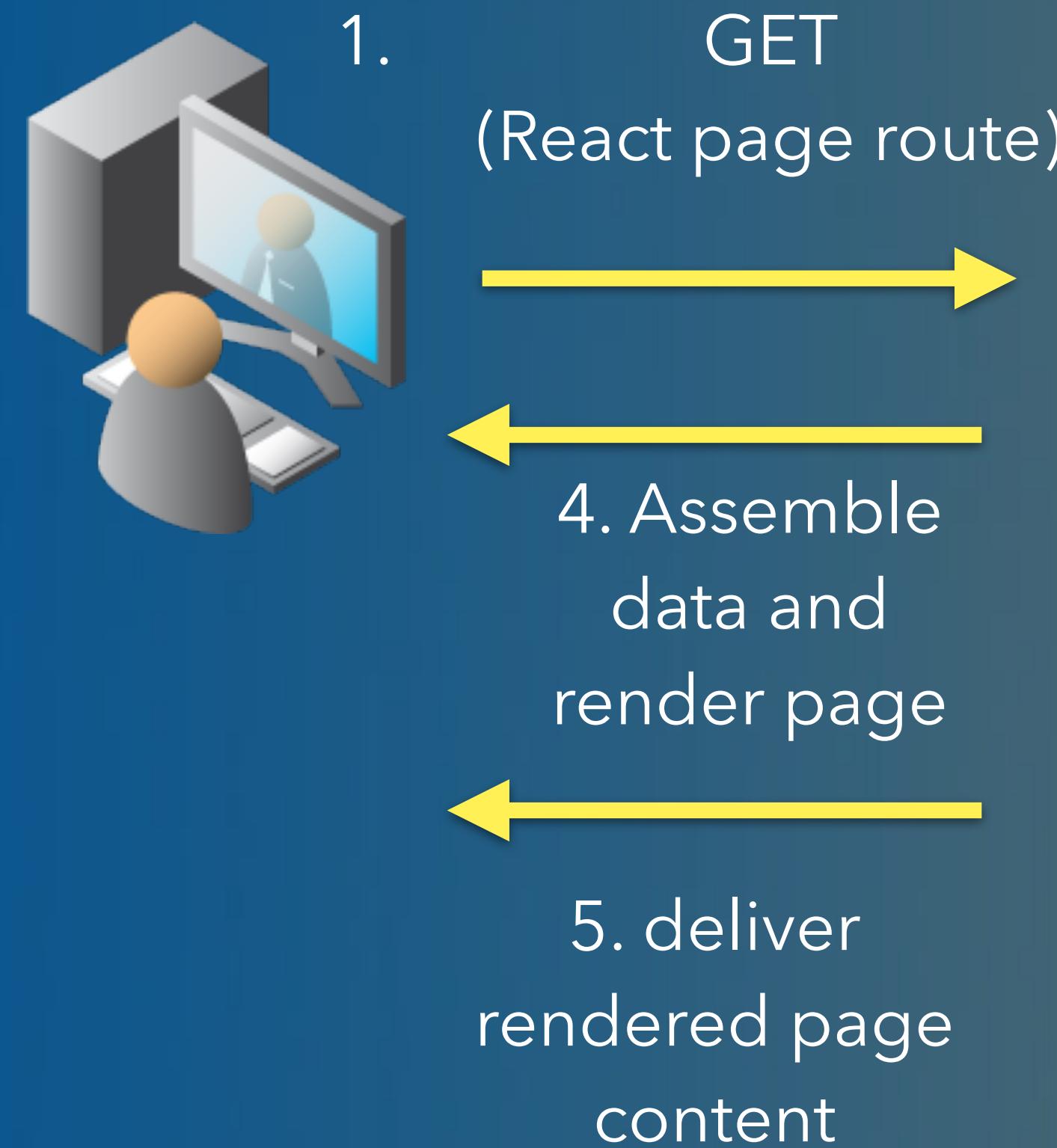
Server-side Rendering



Server-side Rendering - same topology...



Server-side Rendering - using edge computing



Next.js - Support for SSR

```
export async function getServerSideProps() {
  const data = await getFeeds();
  return { props: { feeds: [...data] } };
}

export default function Podcasts({feeds}) {
  const cards = feeds.map((p:FeedDetails) =>
    <ImageCard
      key={p.slug}
      image={p.imageUrl}>
      <div
        className="
          p-6 py-4 h-auto
          overflow-clip"
        dangerouslySetInnerHTML={{__html: p.description}} />
    </ImageCard>
  );
  return (
    <div>
      <h2>Podcasts</h2>
      <div>
        {cards}
      </div>
    </div>
  );
}
```



Next.js - SSR w/path (/pages/podcasts/[slug].tsx)

```
export async function getServerSideProps(context) {
  // get route param for slug
  const slug = context.params.slug;
  const data = await getFeedAndEpisodesBySlug(slug);
  return {
    props: data
  };
}

export default function PodcastEpisodes({ feedDetails, episodes }) {
  let episodeCards = episodes.map((e:FeedItem) => (
    <Card
      key={e.guid}
      title={e.title}>
      <div className="grid grid-cols-2 gap-4">
```



Remix - Support for SSR

```
export const loader = async () => {
  return await getFeeds();
};
```

```
export default function Podcasts() {
  const feedData = useLoaderData();

  const cards = feedData.map((p: FeedDetails) => (
    <ImageCard
      key={p.slug}
      image={p.imageUrl}>
      <div className="...">
        <button className="...">
          <Link to={`/podcast/${p.slug}`}>Show Details</Link></button>
      </div>
    </ImageCard>
  ));
  return (
    <CardContainer title="Your podcasts..." cards={ cards }/>
  )
};
```



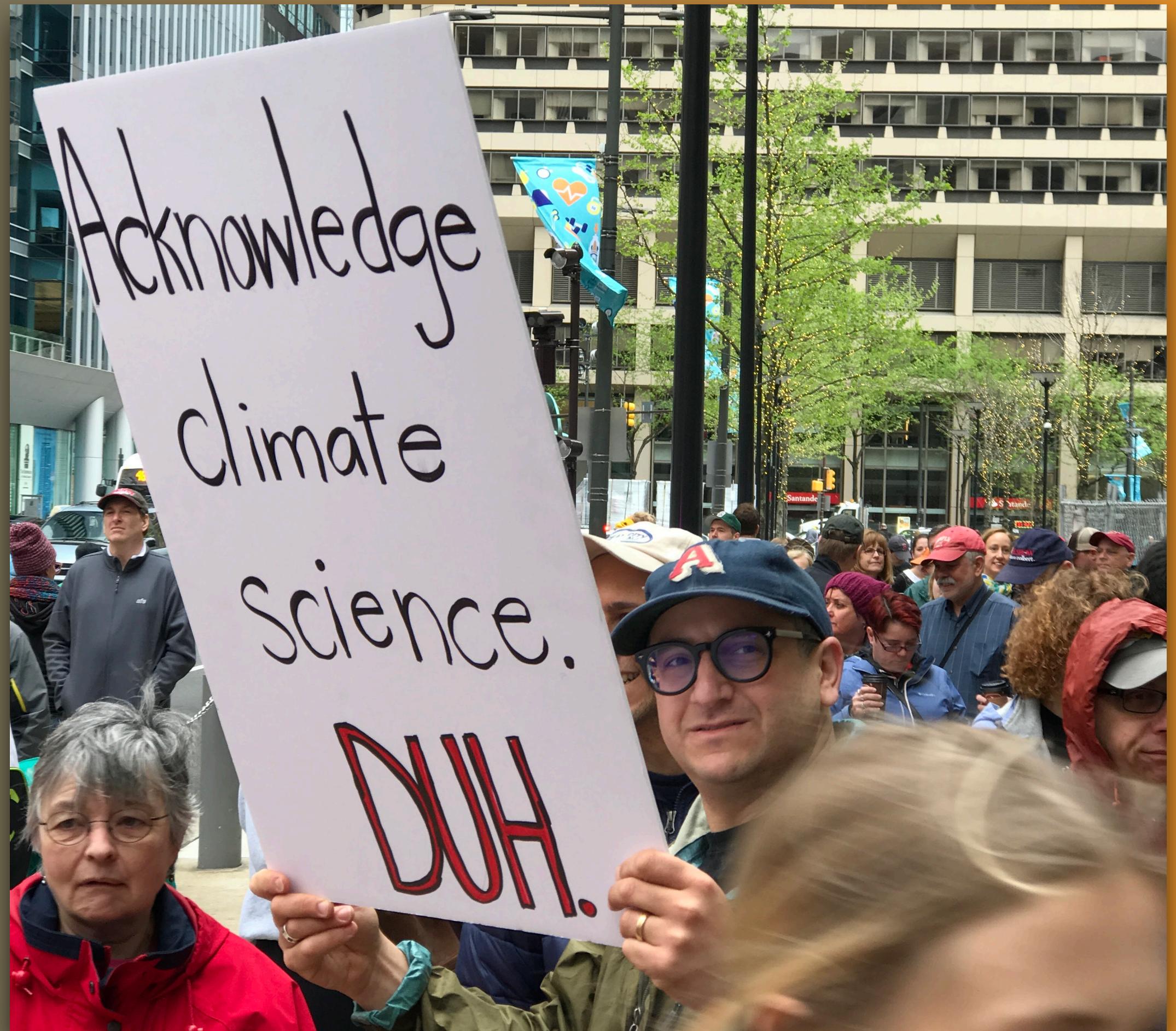
Remix - SSR w/path (/routes/podcasts/\$slug.tsx)

```
export const loader = async ({params}) => {
  // get route param for slug
  return await getFeedAndEpisodesBySlug(params.slug);
}

export default function PodcastEpisodes() {
  const loader = useLoaderData();
  const { feedDetails, episodes } = loader;
  const episodeCards = episodes.map((e:any) => (
    <Card
      key={e.guid}
      title={e.title}>
      <div className="grid grid-cols-2 gap-4">...
    </div>
  )
}
```



SSR Demo



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE

Server-side Rendering Tips

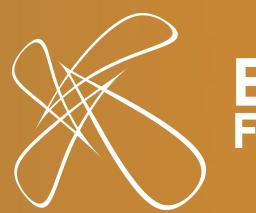
SSR is not a magic bullet, so...



- You could be shifting a distributed performance problem into a shared services bottleneck!
- You can use Node.js Database APIs to directly access database resources
- Monitor server CPU metrics and scale appropriately for load

"Did I leave the iron on?"

Static Site Generation



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE

Static-Site Generated Pages

Pre-render content at build time

- Generates statically rendered content during application build
- Rendered page downloads quickly (not accounting for location)
- React then bootstraps asynchronously



SSG in Next.js - single render for URL

```
export async function getStaticProps() {
  const feeds = await getFeeds();
  return { props: { feeds } };
}

export default function Podcasts({feeds}) {
  const cards = feeds.map((p:FeedDetails) =>
    (
      <ImageCard
        key={p.slug}
        image={p.imageUrl}>
        <div
          className="
            p-6 py-4 h-auto
            overflow-clip"
          dangerouslySetInnerHTML={{__html: p.description}} />
    )
  )
}
```



SSG in Next.js - dynamic rendering by path

how do we deal with pages/podcasts/[slug].tsx in SSG?

```
export async function getStaticPaths(context) {
  const slugs = await getPodcastSlugs();
  const slugPaths = slugs.map(s => ({ params: {slug: s.slug}}));

  return {
    paths: slugPaths,
    fallback: false
  }
}

export async function getStaticProps(context) {
  const slug = context.params.slug;
  const response = await getFeedAndEpisodesBySlug(slug);
  return {
    props: {feedDetails: response.feedDetails, episodes: response.episodes}
  }
}
```



SSG with path - the component

The component is the same as before, receives the path props

```
export async function getStaticProps(context) {  
  const slug = context.params.slug;  
  const response = await getFeedAndEpisodesBySlug(slug);  
  return {  
    props: {feedDetails: response.feedDetails, episodes: response.episodes}  
  }  
}
```

```
export default function PodcastEpisodes({episodes, feedDetails}) {  
  let episodeCards = episodes.map((e:FeedItem) => (  
    <Card  
      key = {e.guid}  
      title={e.title}  
    >  
      <div className="grid grid-cols-2 gap-4">
```



SSG Demos

Using Next.js



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE

Next.js Server-Side Generation (SSG) Tips



- With dynamic routes: don't build millions of SSG pages, limit them with `getStaticPaths`
- Use Next's Incremental Static Regeneration feature to allow for runtime caching once fetched for other pages
- Use SSG to provide SEO-crawlable content, then replace with fresh data after loading (see SWR coming up)
- Don't use SSG on time-sensitive data unless truly shared without user details



Remix team postulates SSG isn't always faster

Remix @remix_run · Jan 18

One common misconception among web devs today is that you need static site generation (SSG) in order to be fast.

We ran benchmarks using [@RealWebPageTest](#) that demonstrate this isn't true, and SSG may actually be *worse* for performance on dynamic pages, like this search page.

Framework	Load Time (seconds)
Remix Rewrite	0.8
Remix Port	1.3
Next.js	1.9

GIF

2 11 92

- Why?

- Too much content to render in time
- For dynamic pages, still must boot React and do async work anyway
- Speed depends on how close you are to the resource as well, so edge functions can be faster

see thread at https://twitter.com/remix_run/status/1483512990741659656

Client-Side Rendering (SWR)

Next.js's special client-side data management hook and HOC

- Will automatically fetch and refresh in a React client based on:
 - Visibility of web page in Chrome tabs
 - Invalidation after a period of time
 - Network connectivity loss
- This can be configured via a HOC - <SWRConfig />

From RFC-5861

The **stale-while-revalidate** HTTP Cache-Control extension allows a cache to immediately return a stale response while it revalidates it in the background, thereby hiding latency (both in the network and on the server) from clients.



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE

SWR Example

```
export default function Podcasts() {
  const [numRows, setNumRows] = useState();
  const {data, error, isValidating, mutate} = useSWR(
    `http://localhost:3010/podcasts/feeds?numRows=${numRows}`, fetcher);

  if (error) { return <pre>{ JSON.stringify(error) }</pre>; }

  if (isValidating) { return <Spinner/>; }

  const cards = data?.map((p:any) =>
    <ImageCard
      key={p.slug}
      image={p.image_url}>
      <div>
        <button className="...">
          <Link href={`/standard/podcast/${p.slug}`}>Show Details</Link>
        </button>
      </ImageCard>);
  return (
    <>
      <select>
```



SWR Demo

Using Next.js



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE

Comparing the two frameworks

Feature	Next.js	Remix.run
Server-side Rendering	Yes	Yes
Static Site Generation/Regeneration	Yes	No (use infra cache?)
Page-based Routing	Yes	Yes
Client-side Smart Fetch Library	Yes, SWR	Use 3rd party (SWR?)
Streaming Results	Yes	Yes
Image optimization	Via <Image> component	No
Script loading optimization	Via <Script> component	No

React Server Components and Suspense

In preview for Next.js, see <https://nextjs.org/docs/advanced-features/react-18/streaming>

```
import { Suspense } from 'react'  
import Note from '../components>Note.server'  
import NoteSkeleton from '../components>NoteSkeleton'  
import Page from '../components/Page.server'
```

```
return (  
  <Page login={login} searchText={searchText}>  
    <Suspense fallback={<NoteSkeleton isEditing={false} />}>  
      <Note login={login} selectedId={id} isEditing={false} />  
    </Suspense>  
  </Page>  
)
```

Wrap-up

Next.js and Remix.run are extending React to the server

- To decrease latency
- To provide a great Backend-for-Frontend server solution for SPAs
- To break up monolithic React applications into page-based routes
- To support deployment to edge servers, static generation of content (Next.js) and server-side rendering (both)
- Beware of vendor lock-in by embracing these frameworks without considering all of the angles (hosting fees, target platforms and topologies, data usage patterns, etc.)

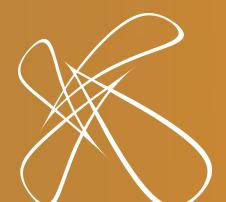


Questions? and Thank You!

My sample code and slides:

<https://github.com/chariotsolutions/ken-rimple-ete-2022-next-and-remix>

Ask me questions on the ETE Slack!



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE