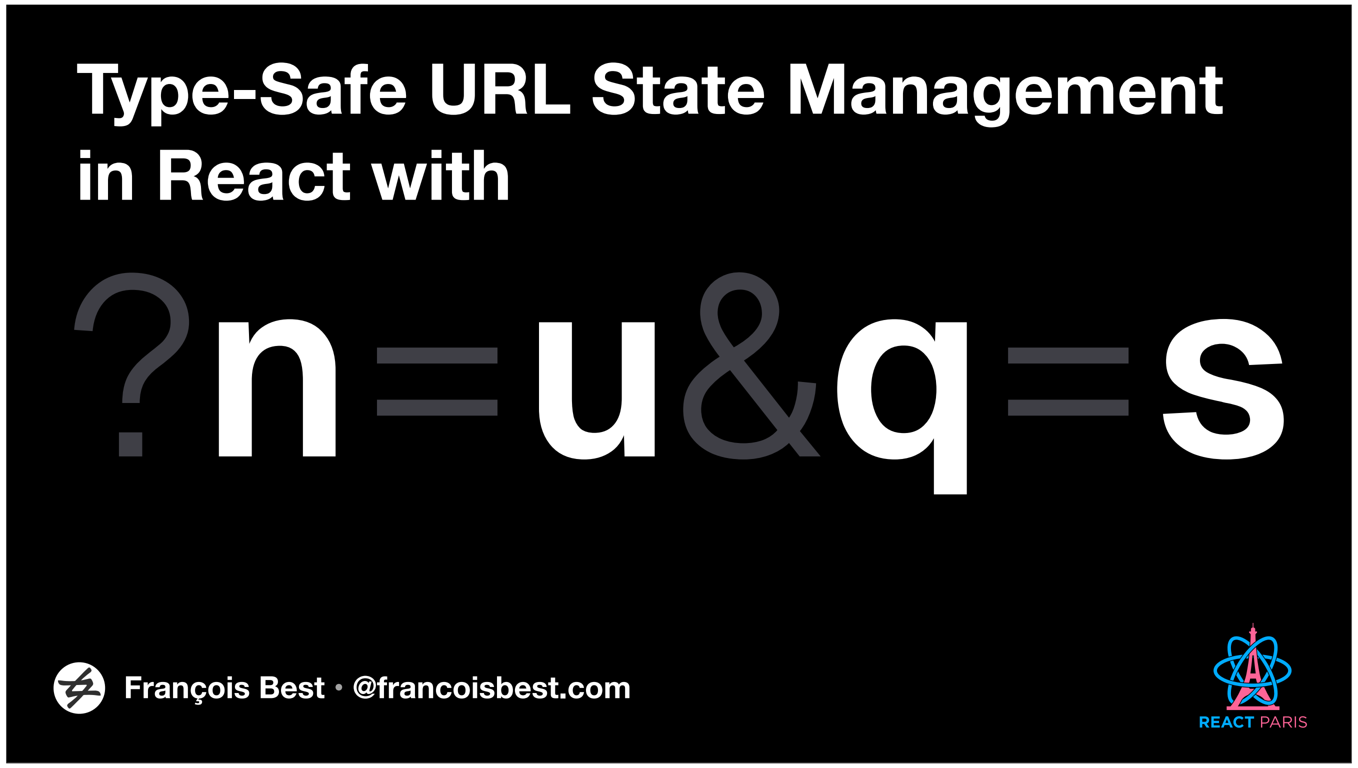


Type-Safe URL State Management in React with

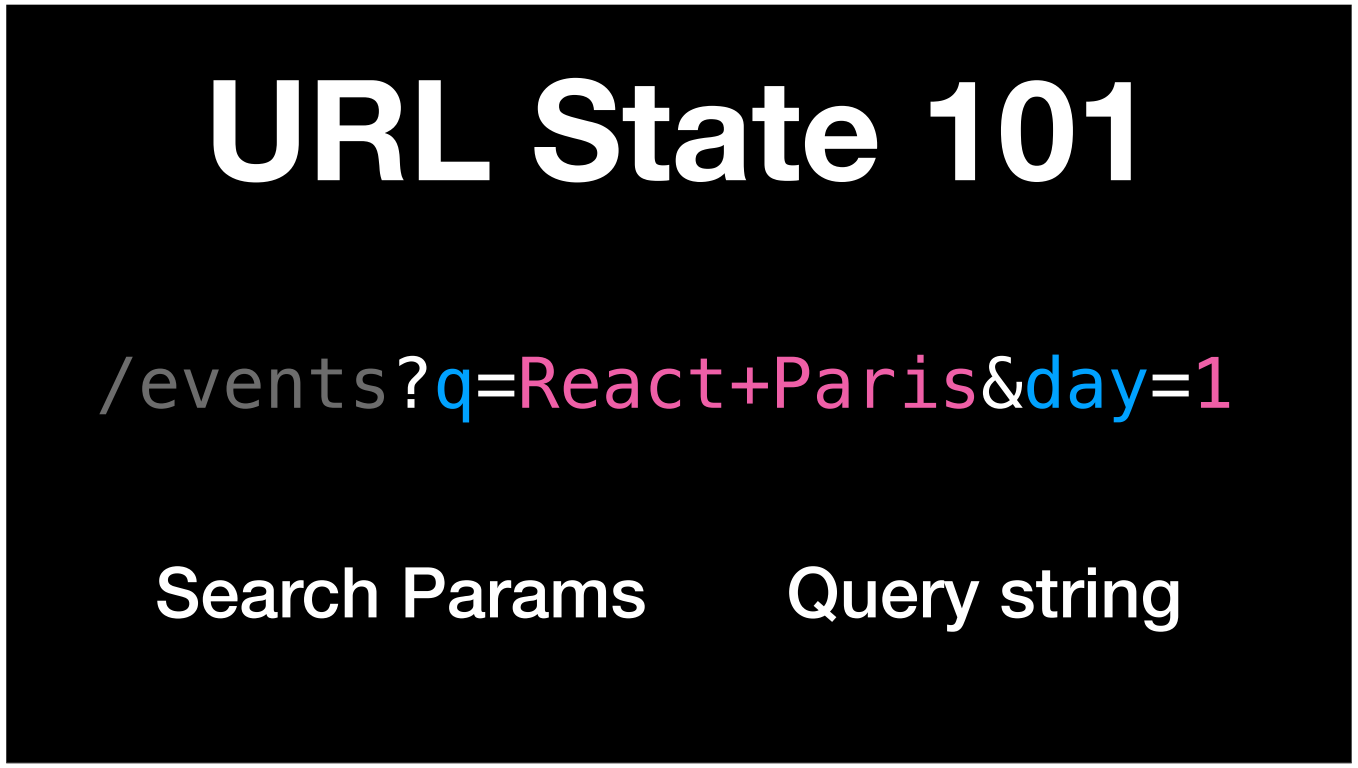
?n=u&q=s

François Best • @francoisbest.com

REACT PARIS

Alright let's talk about URL state in React, with a little library that I made, called nuqs.

So hi everyone, my name is François Best, I'm a freelance software engineer from Grenoble, in the French Alps. I've been working with React since 2017, and I publish open-source packages on NPM to solve problems for my own apps and that of my clients.
You can find me on Bluesky at francoisbest.com



URL State 101

```
/events?q=React+Paris&day=1
```

Search Params Query string

So this talk is about URL state, but what are we talking about exactly? Let's do a quick recap:

URL state is the idea of storing structured data in the query string or search params part of the URL. Everything after the question mark is a key/value store where we can persist the state of our applications.

Why URL State?

But why would we want to do that?

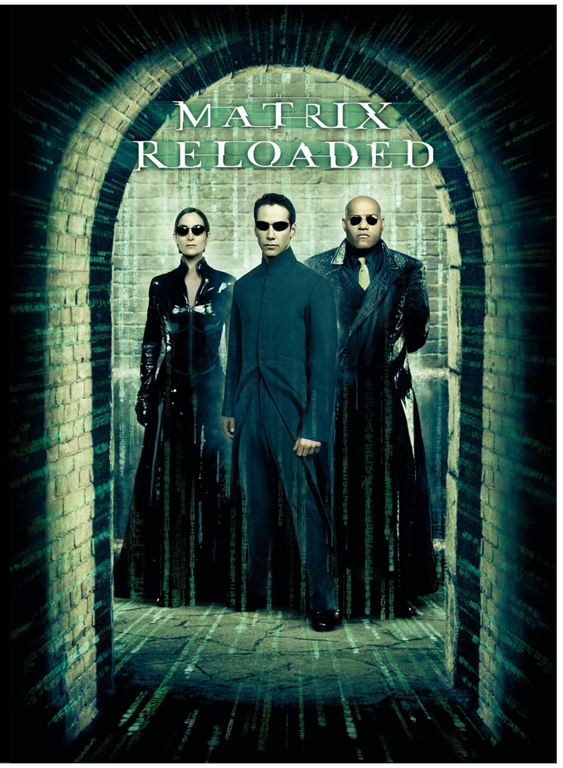
One of the wrong reasons (at least on its own), is...

Reload resilience

Side-effect

localStorage

The Matrix Reloaded, 2003
The Wachowskis



Reload resilience.

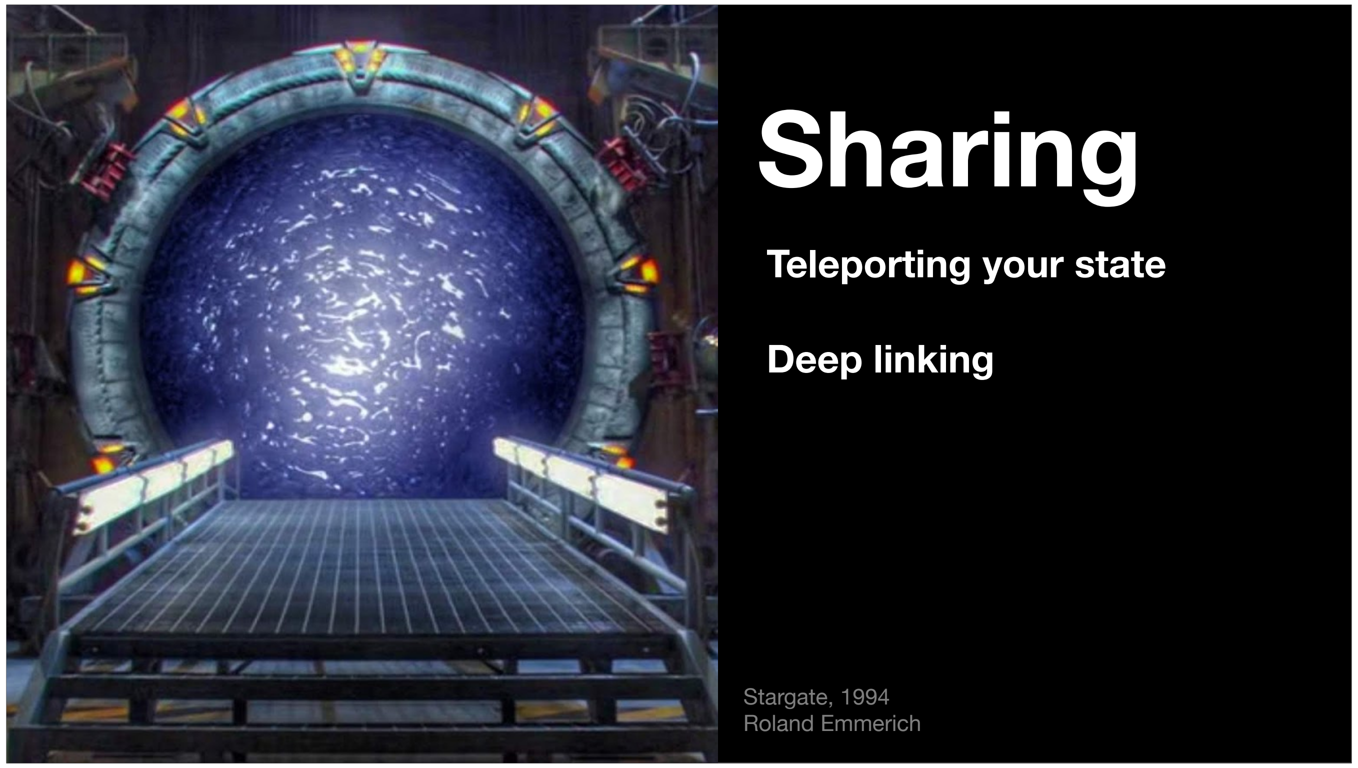
Because traditional state management happens in-memory, any changes you do gets lost if you reload the page.

Putting them in the URL allows you to persist those changes across reloads, but it's more of a side-effect, and there are other solutions to achieve this, like localStorage.

No, the real reason why we want to put state in the URL is because it gives our applications...

🦸 Superpowers

Superpowers. Namely, two superpowers.

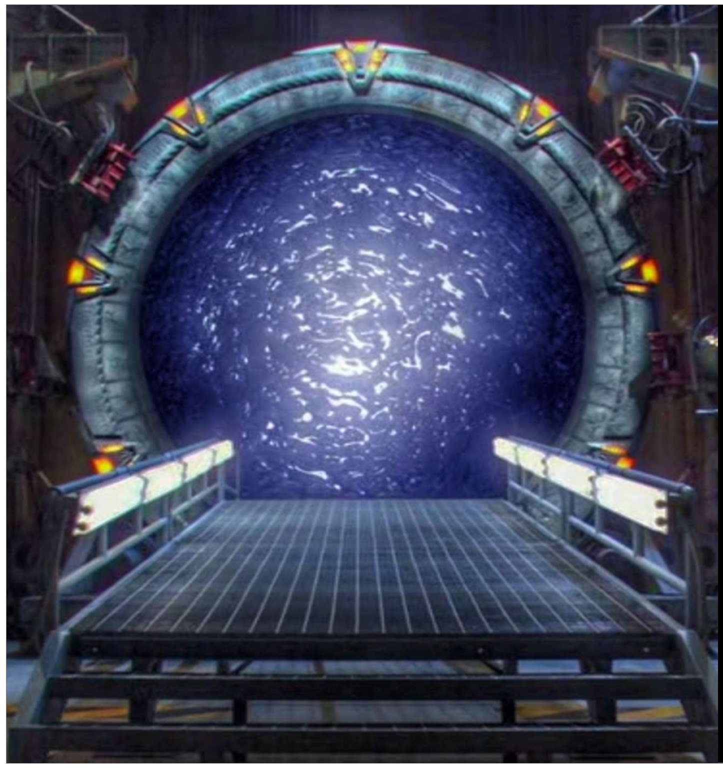


Sharing

Teleporting your state

Deep linking

Stargate, 1994
Roland Emmerich



The first one is teleportation.

By sharing a link with someone, it teleports the current state of the application from your browser to theirs.

This allows them to pick up where you left off, and to see the same thing as you do, and it's a very powerful pattern, also called deep linking

Bookmarking

History navigation

Back / Forward
Undo / Redo

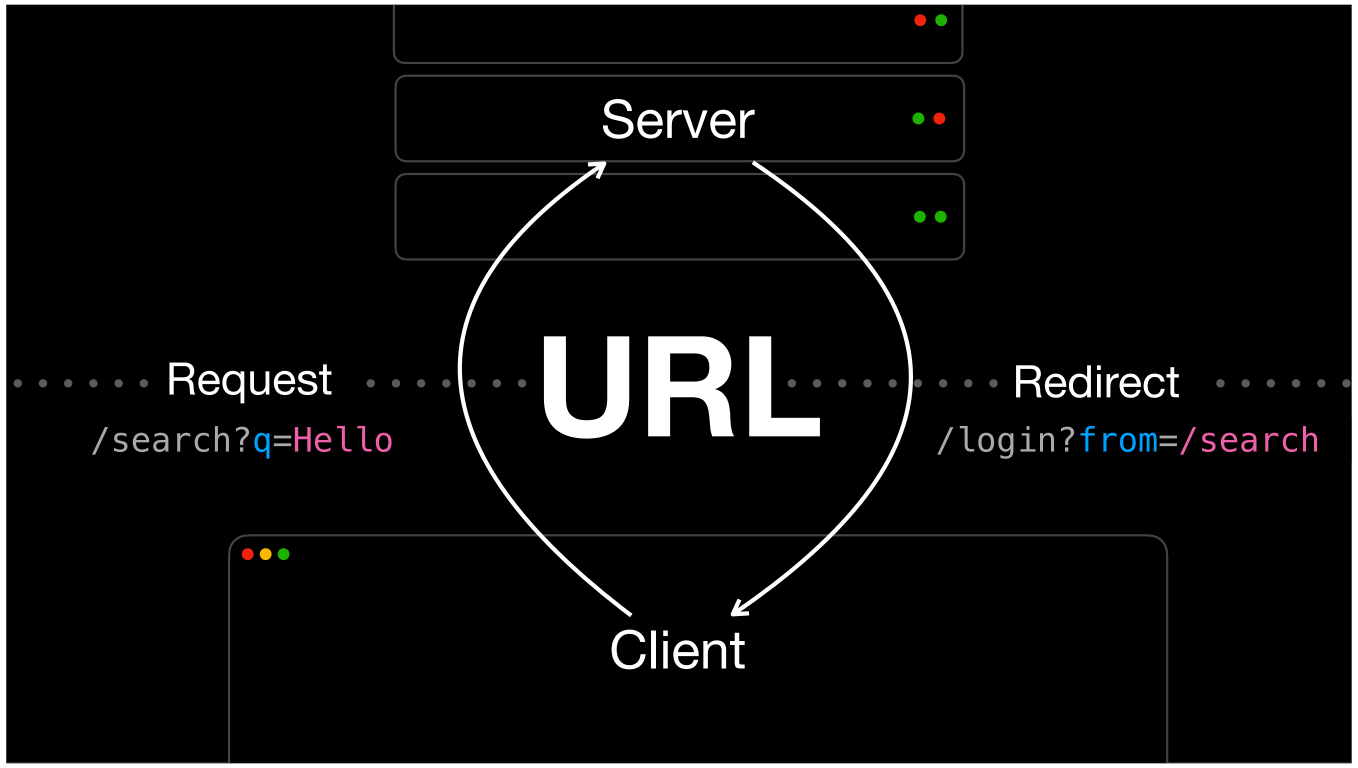
Back To The Future, 1985
Robert Zemeckis

The second superpower is time travel.

Bookmarking is a way to share a link with yourself in the future, and history navigation allows you to recall state from the past.

Also, if you were to push your state updates onto the browser history stack, you can now use the Back & Forward buttons of the browser as an Undo/Redo feature; I won't say "for free", as there are some caveats that we'll talk about in moment.

But essentially it allows you to time-travel debug your state updates, which is pretty cool.



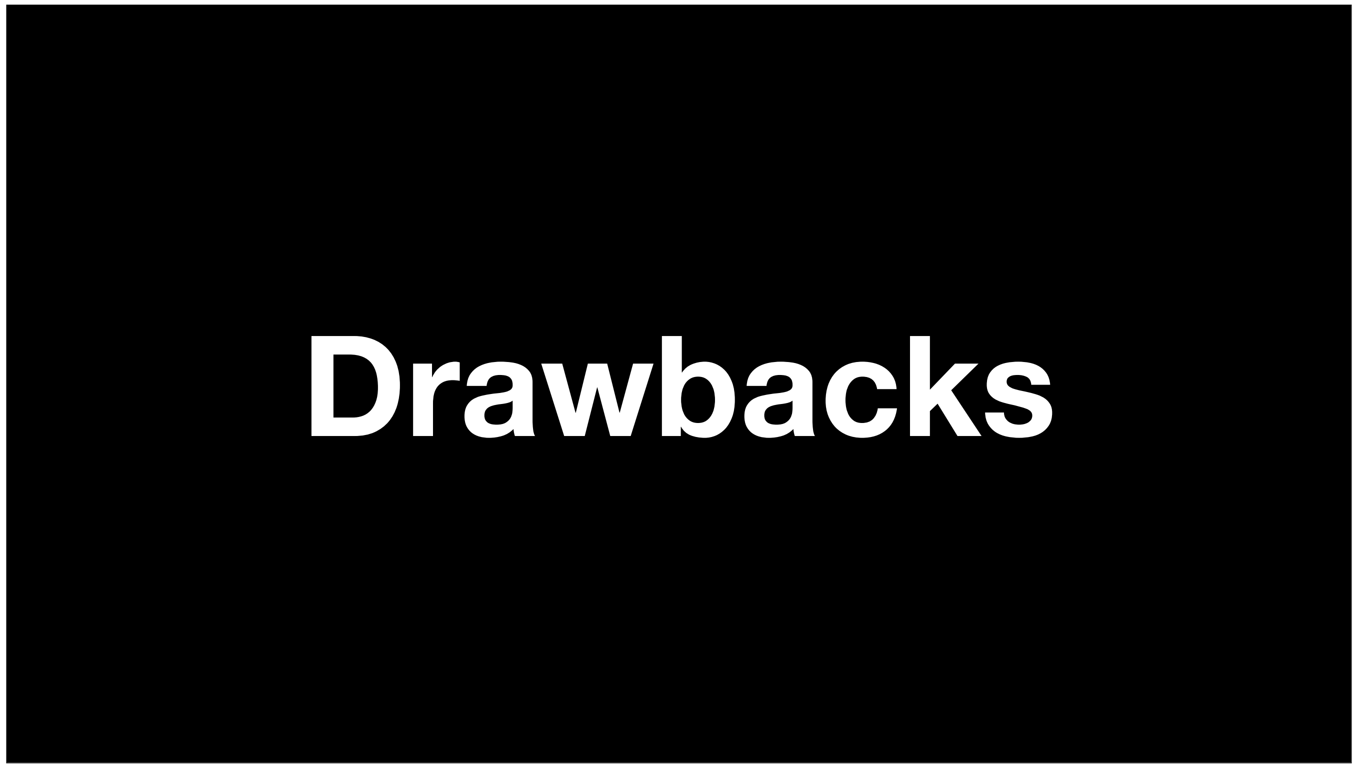
Now, there is another interesting property of the URL itself.

Since our frameworks are orienting more and more towards server-side rendering (with React Server components), we can picture the URL as the boundary between the client and the server.

And we can use this to send some data to the server, by adding state to the request for the page we are on, rather than going through a separate API. The server can then do its data fetching by using this data and return customised content.

But the server can also use this to send us back some state. Not via the response body, but via the URL itself. With redirects, the server can attach relevant pieces of state and tell the client: "here's what you'll need when rendering".

So we have this bidirectional data link between client and server using only the URL.



Drawbacks

So this is cool, but there are some drawbacks to this approach.

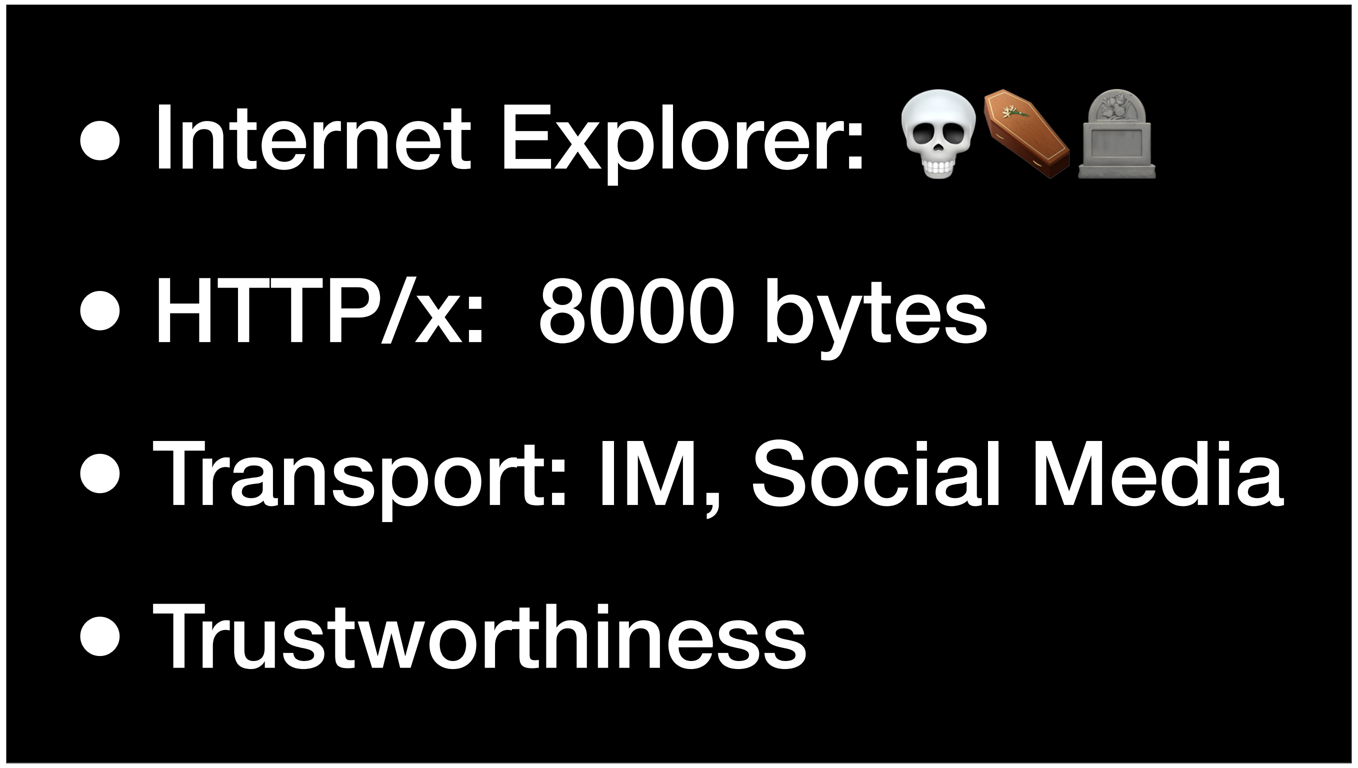
I don't want this talk to sound too much like a sales pitch for a particular solution, instead I'd like to focus on 5 different things that you can go home with, and think about when implementing URL state in your applications.

2000

characters: safe URL size

The first one is that the URL has a size limit.

About 2000 characters is generally considered safe,



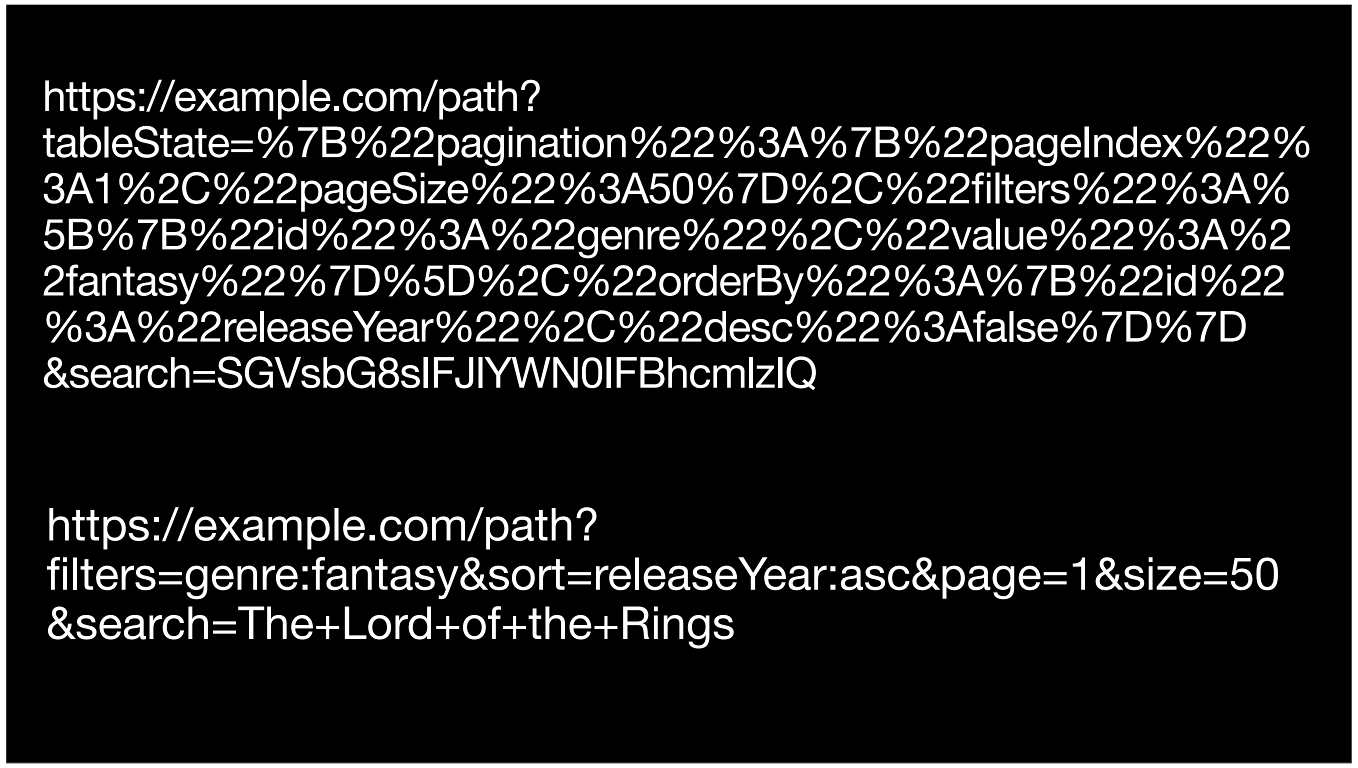
- Internet Explorer: 💀⚰️🪦
- HTTP/x: 8000 bytes
- Transport: IM, Social Media
- Trustworthiness

Though that number mostly comes from the days of Internet Explorer (RIP)

The actual HTTP spec is more allowing, at around 8kB

But the practical, technical limitations that you'll find will be the way you share links with someone else. If you were to send a link via a messaging app like Signal or Whatsapp, those have limited message sizes, and so does social media. Even with a URL shortener, the platform might not accept a link that is too large, or render it as text rather than a clickable link.

There is another limitation which is more of a social one, and that's the trustworthiness of your links.



```
https://example.com/path?
tableState=%7B%22pagination%22%3A%7B%22pageIndex%22%
3A1%2C%22pageSize%22%3A50%7D%2C%22filters%22%3A%
5B%7B%22id%22%3A%22genre%22%2C%22value%22%3A%2
2fantasy%22%7D%5D%2C%22orderBy%22%3A%7B%22id%22
%3A%22releaseYear%22%2C%22desc%22%3Afalse%7D%7D
&search=SGVsbG8sIFJlYWN0IFBhcmlzIQ
```

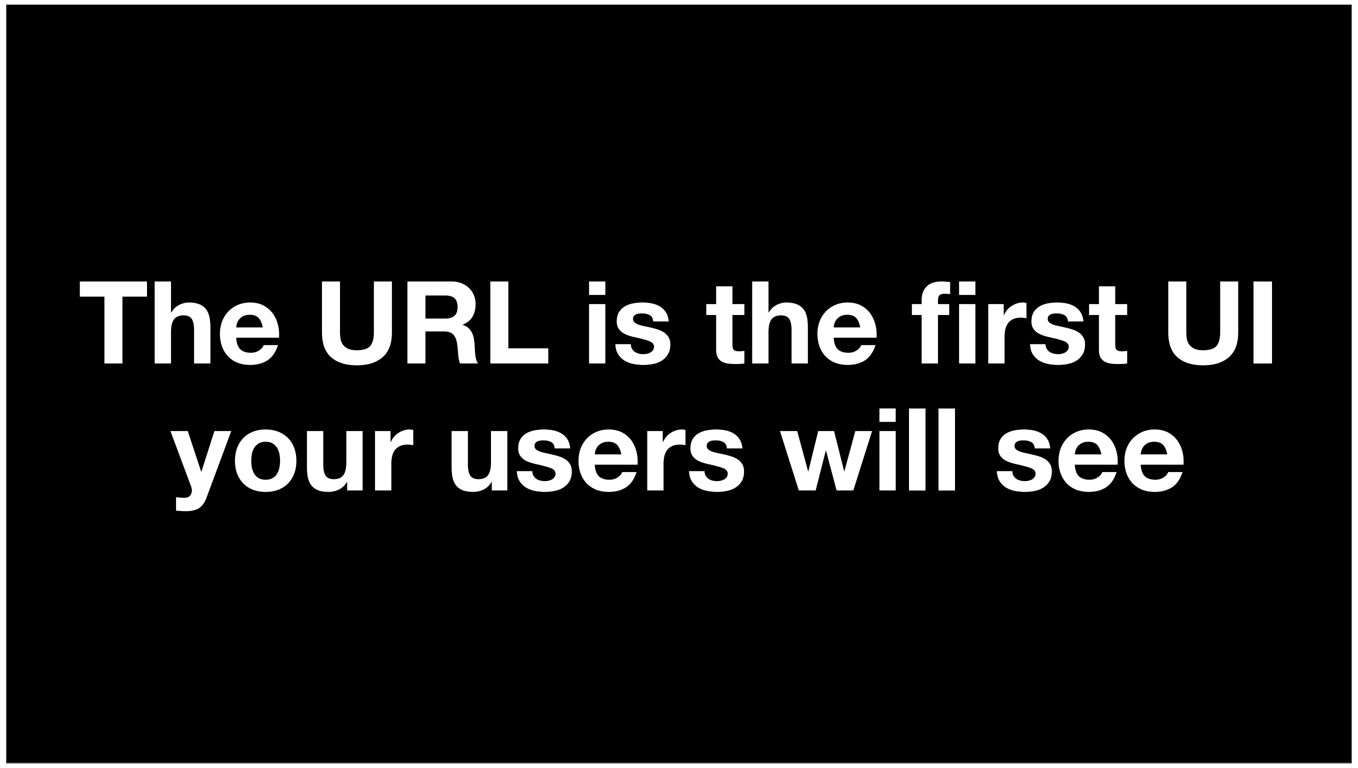
```
https://example.com/path?
filters=genre:fantasy&sort=releaseYear:asc&page=1&size=50
&search=The+Lord+of+the+Rings
```

For example, between these two links (which are semantically equivalent), which one would you rather click on?

The second one, right?

The first one contains a lot of URL-encoded JSON, and some base64, while the second one is not only shorter, it's readable: you can read the link and understand what it means.

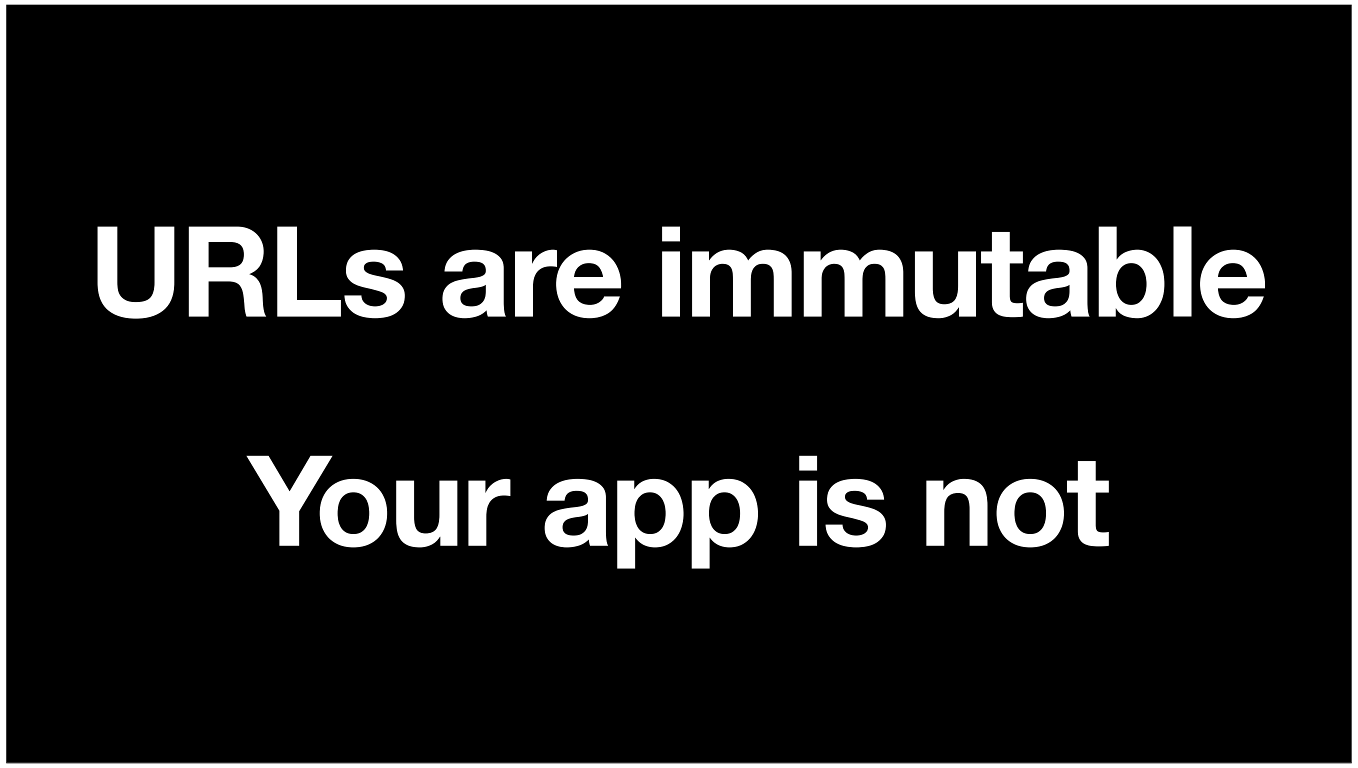
And that's something important to remember:



The URL is the first UI your users will see

The URL is the first piece of UI your users will see, even before they land on your application. So you'll want to make them feel welcoming. Better URLs, better conversion, more sales, right?

Now once you've shared that link, this is where our second drawback kicks in.

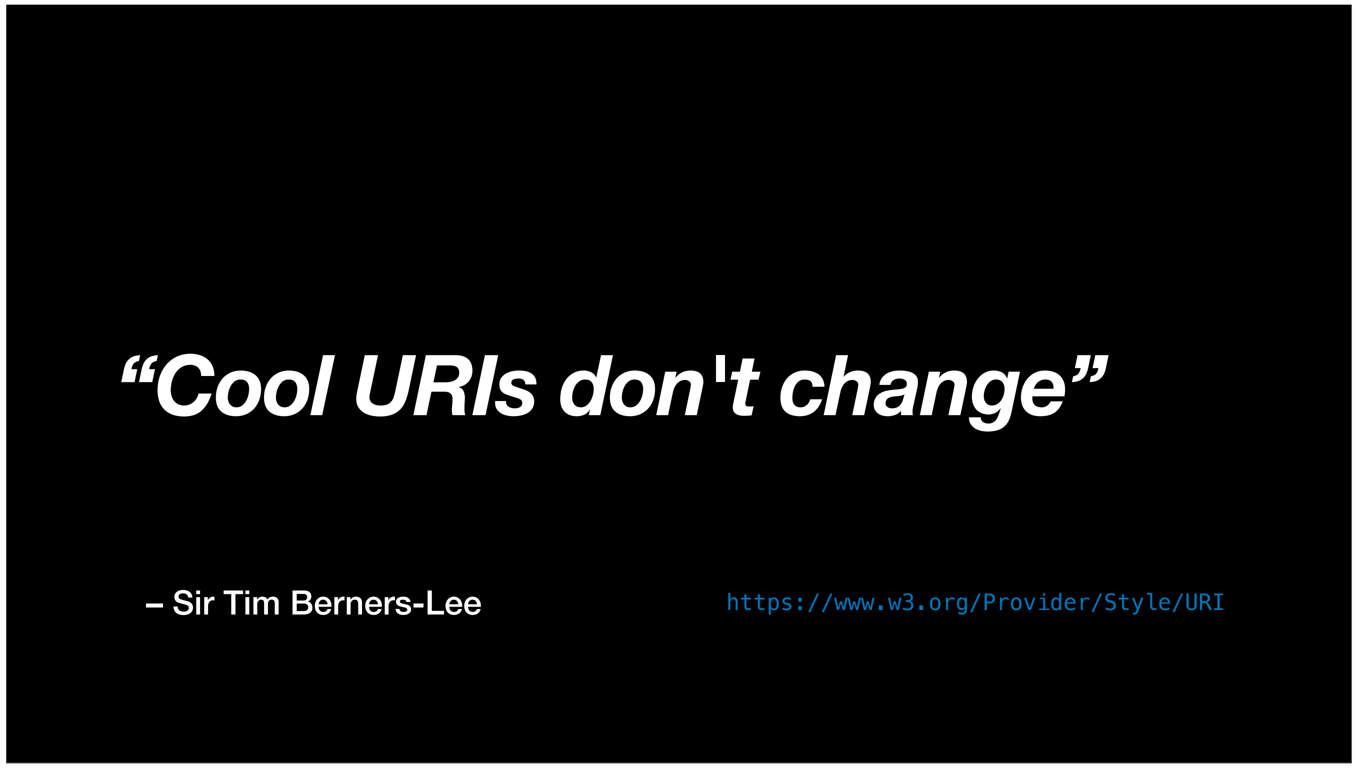


URLs are immutable

Your app is not

The links you've sent are **immutable** once in the wild. But your application is not, it will evolve through time.

And so by introducing statefulness in the URL, it's like if you'd shared little immutable databases that your app needs to support.



“Cool URIs don't change”

– Sir Tim Berners-Lee

`https://www.w3.org/Provider/Style/URI`

Because cool URIs don't change, you want your app to be able to understand those old links.

So now you're left to do schema migrations.

Middleware Redirects

You can do so with middleware & redirects, to massage old URLs into the current schema that your application expects.

And this is something I'd like to work on in nuqs now that React Router has initial support for middlewares.

Now, we need to use our time-travel superpower, and go back to the Undo/redo case.

Undo / Redo

I said it didn't come for free, and the cost is that by pushing state updates onto the history stack, you'll risk

Hijacking the Back button

Spider-Man, 2002
Sam Raimi

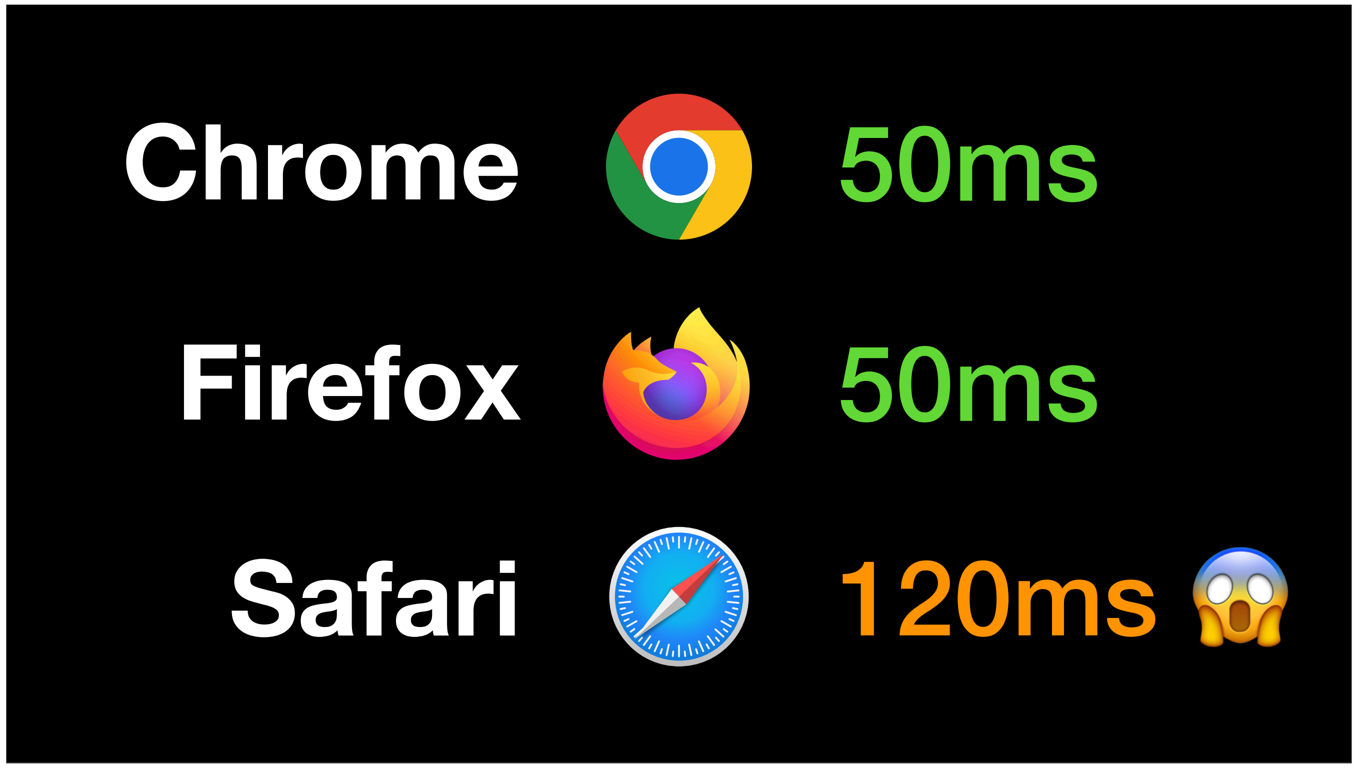
breaking the back button, which is something users hate.

You'll definitely want to reserve that behaviour for state updates that **feel** like a page navigation.

Now when you're updating the URL, there's something that is not often known:

Rate Limits

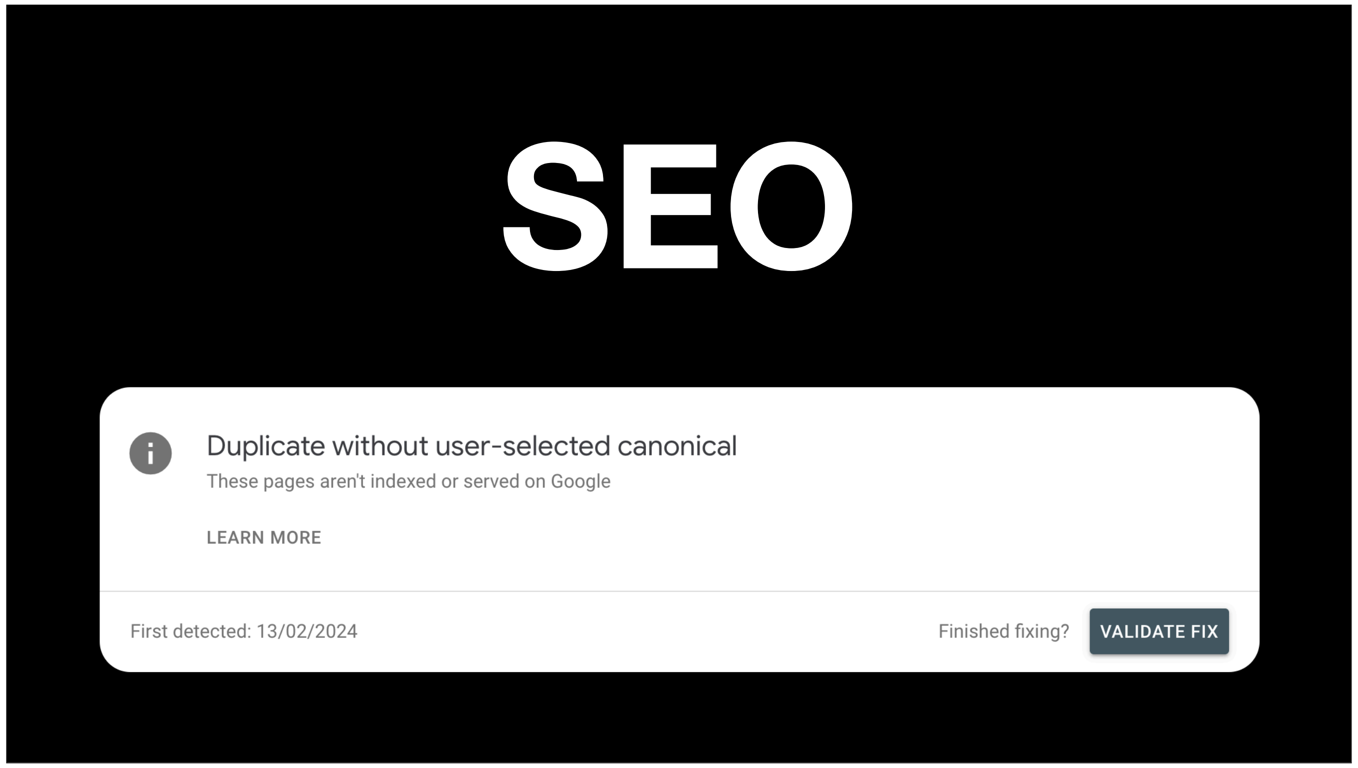
Browsers will rate-limit those URL update calls, and throw errors if you call them too fast, potentially crashing your app in the process.



Chrome	50ms
Firefox	50ms
Safari	120ms 😱

Chrome & Firefox allow about 50ms between calls to be safe, but Safari has much higher limits, which you can hit if you were to directly connect a search input to the URL, by updating it on every keystroke. If you type at a reasonable pace, you can hit those limits.

nuqs solves this with a throttled queue, which you can configure, but it's something to keep in mind.



SEO

Duplicate without user-selected canonical

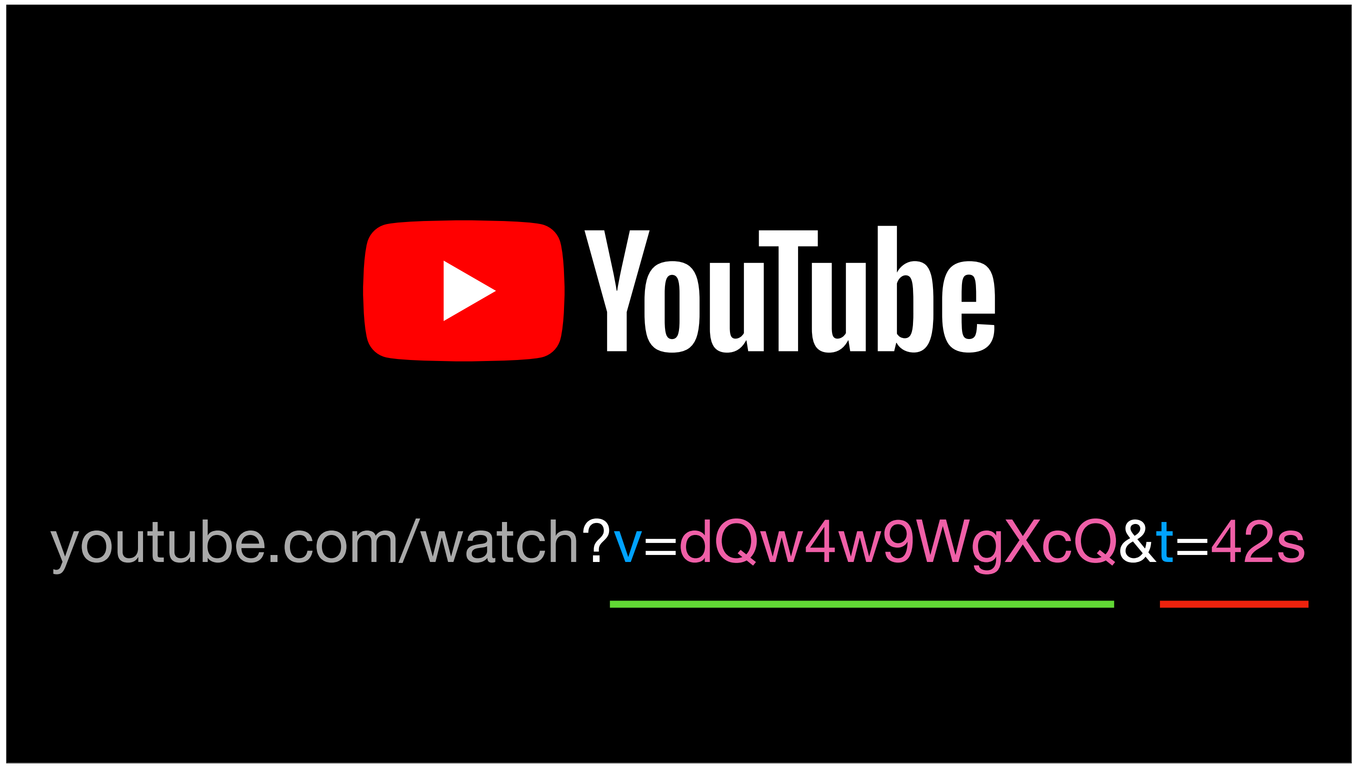
These pages aren't indexed or served on Google

LEARN MORE

First detected: 13/02/2024

Finished fixing?

The last point I want to cover is SEO. It's very easy to shoot yourself in the foot by adding too much state in the URL, because search engine crawlers, like Googlebot, will happily find lots of variants of your URL, and report a lot of duplicates, because it doesn't know which one is the correct one to index.



youtube.com/watch?v=dQw4w9WgXcQ&t=42s

An example for this is the YouTube watch page. There is only **one** page for every single video on YouTube: /watch.

The video ID is encoded in the v search param, but it also accepts other search params, like t, to start the video at a particular point in time, and those two search params, t & v, don't carry the same weight.

We want to crawler to index v, but ignore the rest.

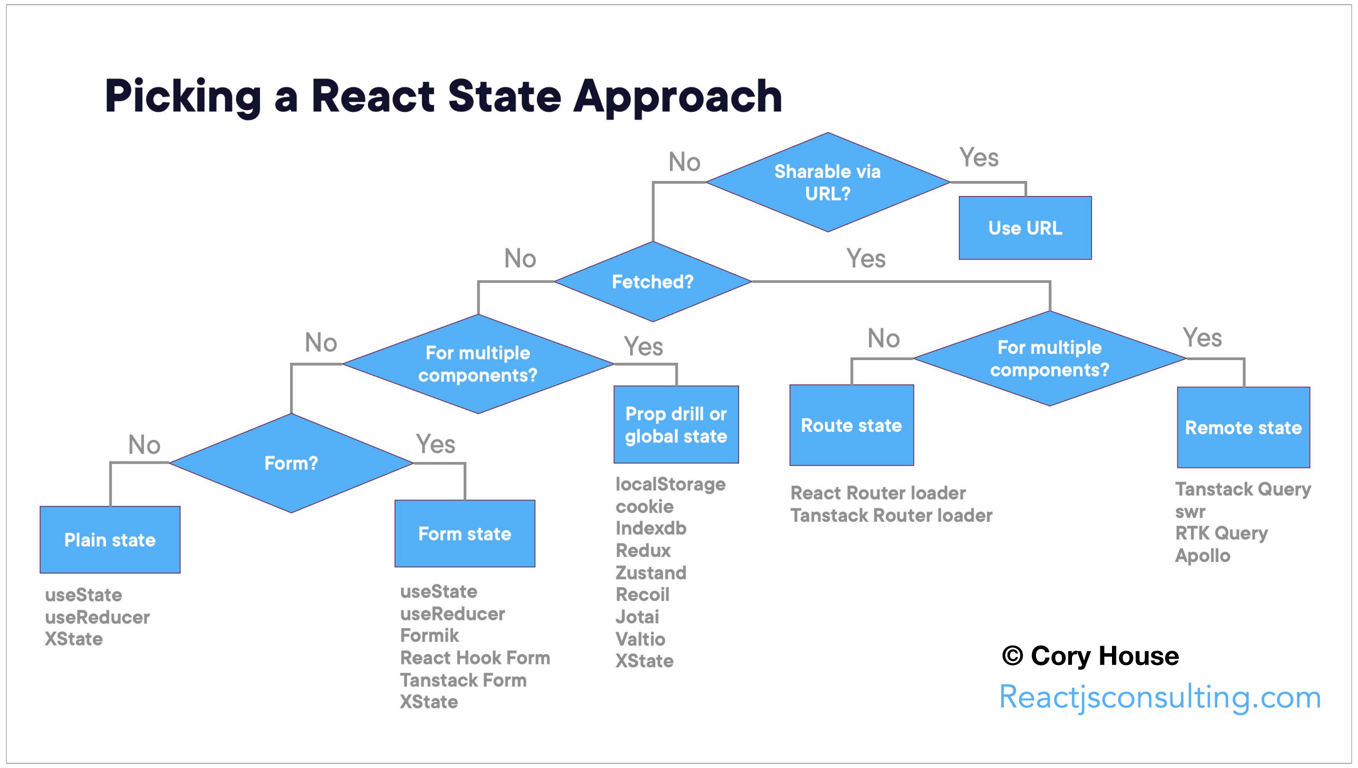
```
<link rel="canonical" href="...">


HTTP/1.1 200 OK
Content-Length: 19

Link: <https://...>; rel="canonical"


sitemap.xml
```

That is done through the canonical representation of your URL, which you can specify either in a link tag in the head of your HTML, or as a response header from your server, or via a sitemap.

Now, hopefully I haven't scared you off too much of the concept of URL state, and so

What state goes in the URL?

Let's talk about what actually should go in there.



There is this diagram by Cory House, there is a link to the slides at the end if you want to see a bigger version, but it lists a lot of the state management libraries that we know and love, things like TanStack Query for server state, or Zustand, Jotai for global state; XState is mentioned a few times.

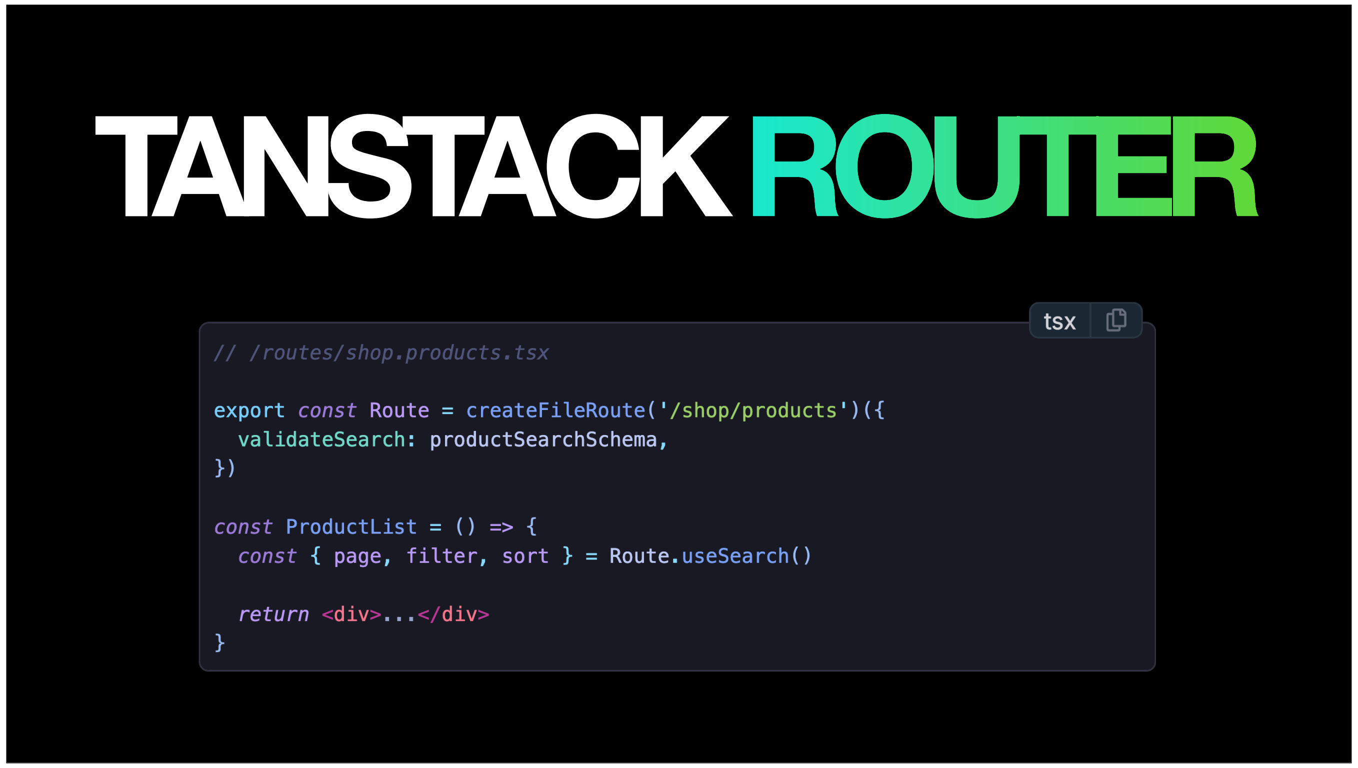
The diagram starts at the very top with "Is this shareable via the URL?" if so, "use the URL". And this is the prime factor: does it make sense for that state to be publicly shared? That implies a few other properties:
- Don't put private data in there
- And it works better for flat-ish state, with deeply nested objects you'll end up with a huge URL that will be hard to migrate over time when your state schema changes.

So for everything else (that lives in-memory), we have a lot of options, but...

How?

how do we "use the URL"?

If you're lucky enough to be using one of the latest frameworks:

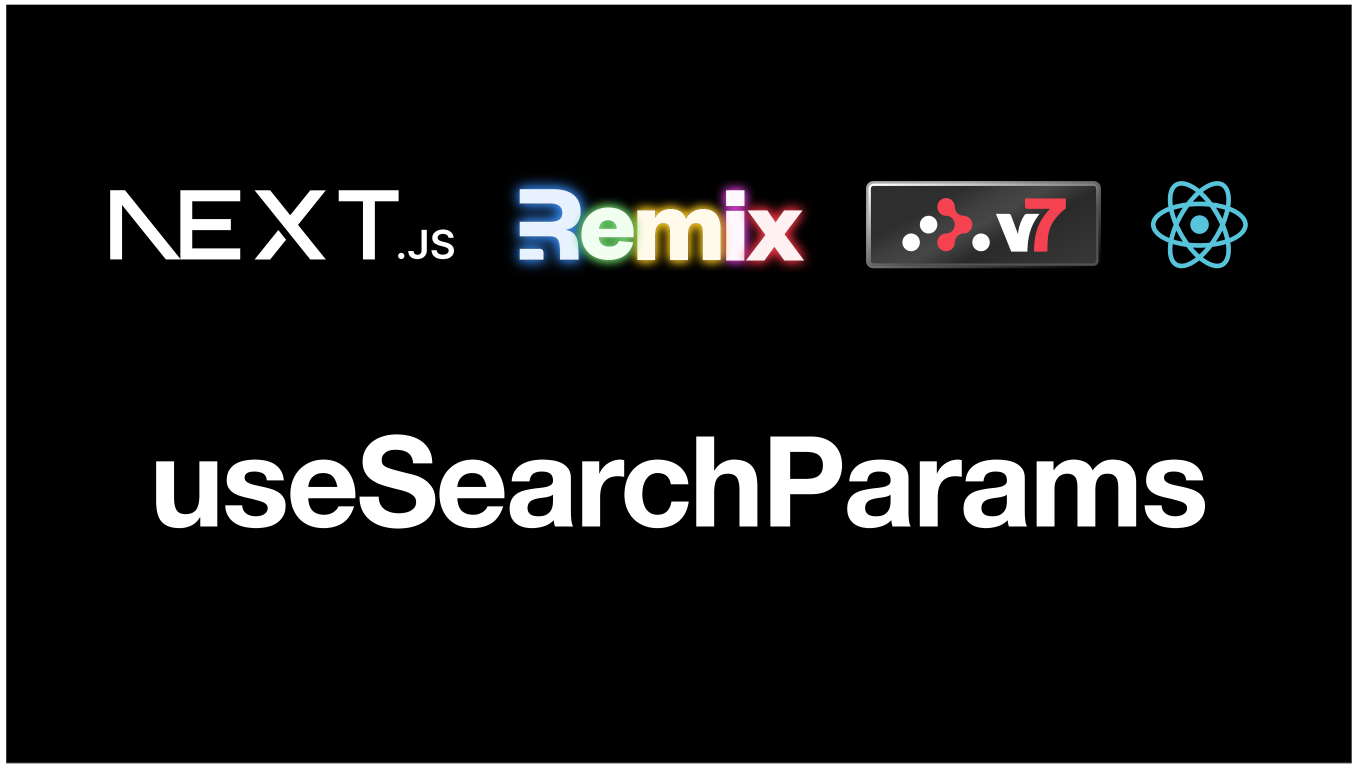


TANSTACK ROUTER

tsx

```tsx
// /routes/shop.products.tsx

export const Route = createFileRoute('/shop/products')({
  validateSearch: productSearchSchema,
})

const ProductList = () => {
  const { page, filter, sort } = Route.useSearch()

  return <div>...</div>
}
```

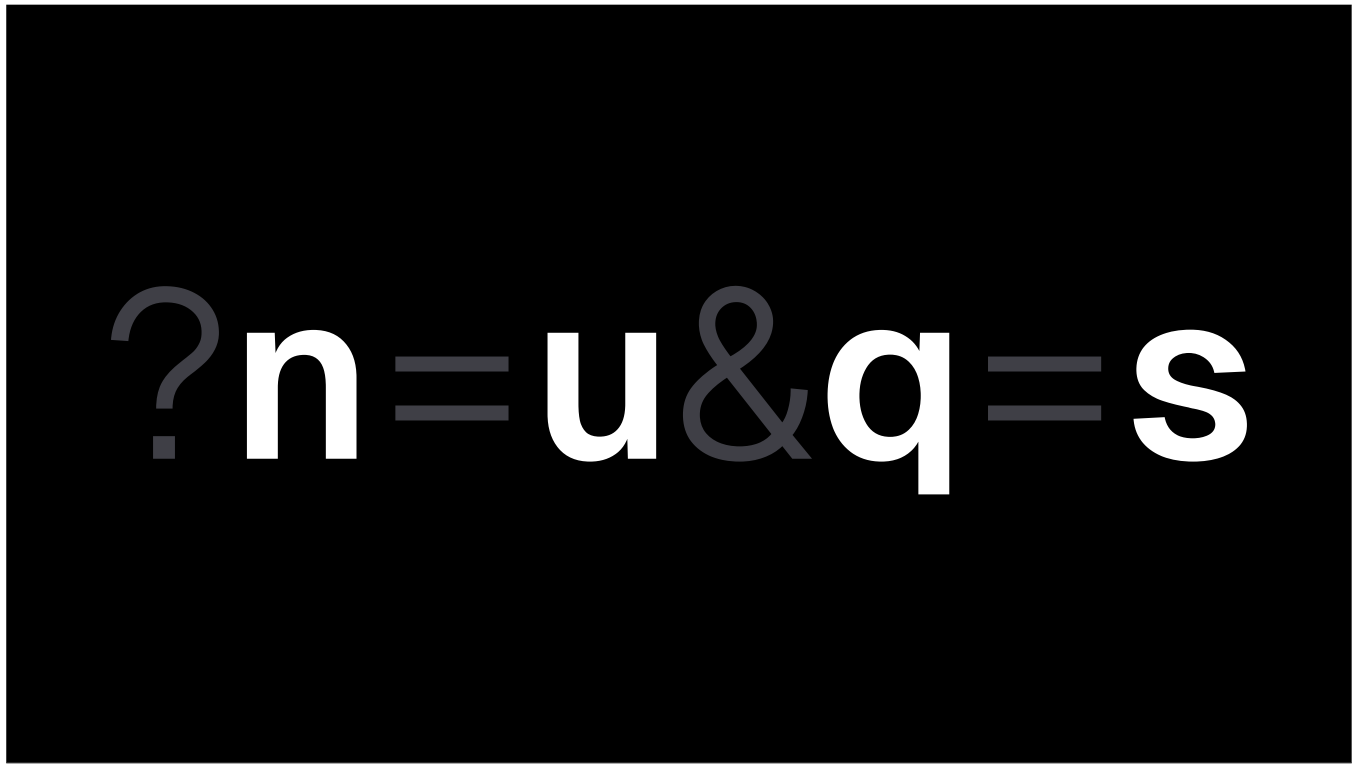
TanStack Router/Start, everything is baked-in for you. Tanner and his team have done an incredible job here, setting the DX bar very high for type-safe search params, so check it out.



useSearchParams

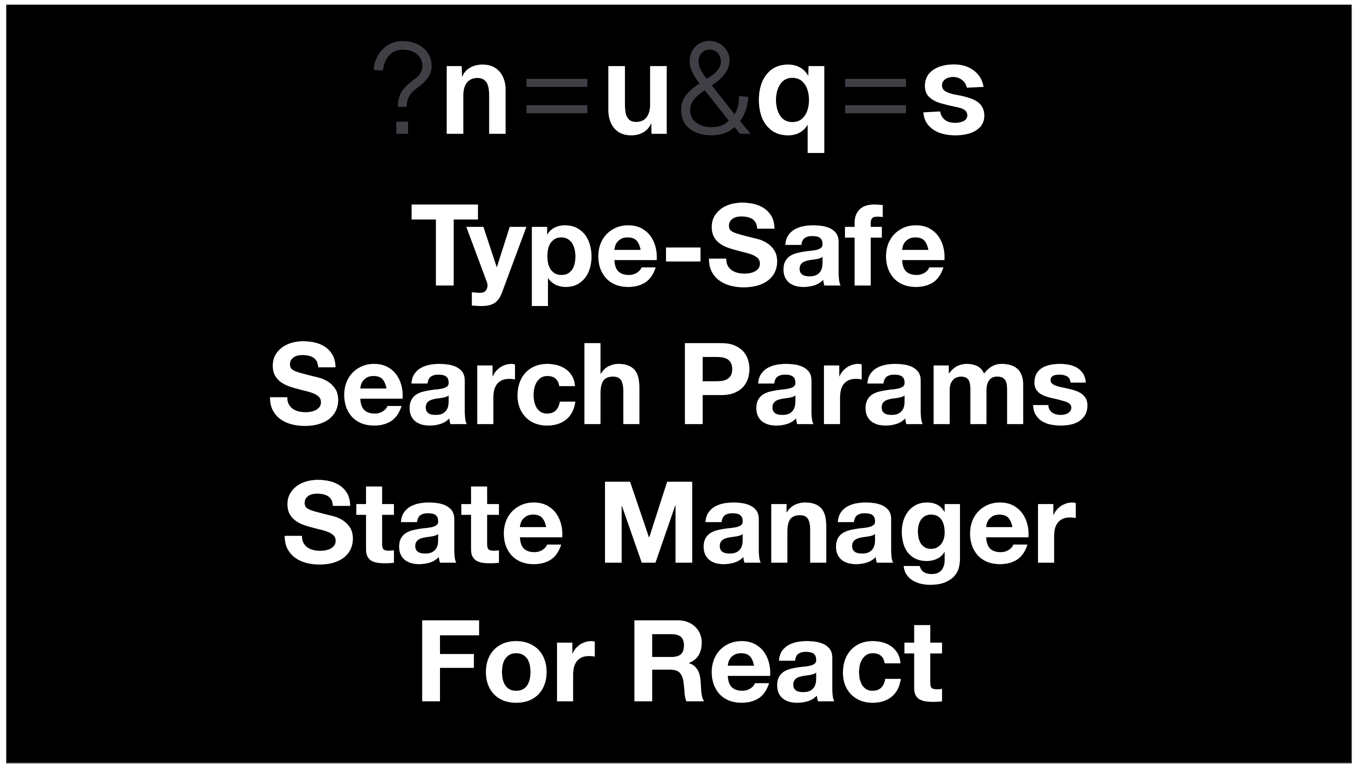
But what about the other 99% of applications out there, that are based on Next.js, on Remix, on React Router, or all the Single Page Apps out there?

In some cases, you might get a useSearchParams hook, but it gives you all the search params, and all as strings, which is not super helpful, because the state of our applications might be something else, like booleans, numbers, dates, objects, and we want to strongly bind them to known keys: we want type-safety.



?n=u&q=s

So over time, I kept reusing the same hook to do that in my own apps, I eventually put it on NPM and used it to help my clients, and it became that project called nuqs



?n=u&q=s

Type-Safe Search Params State Manager For React

nuqs is a type-safe search params state manager for React.

Type-safe because it does the translation back & forth between the query string and meaningful data types for your application

Search params because it stores its state in the URL

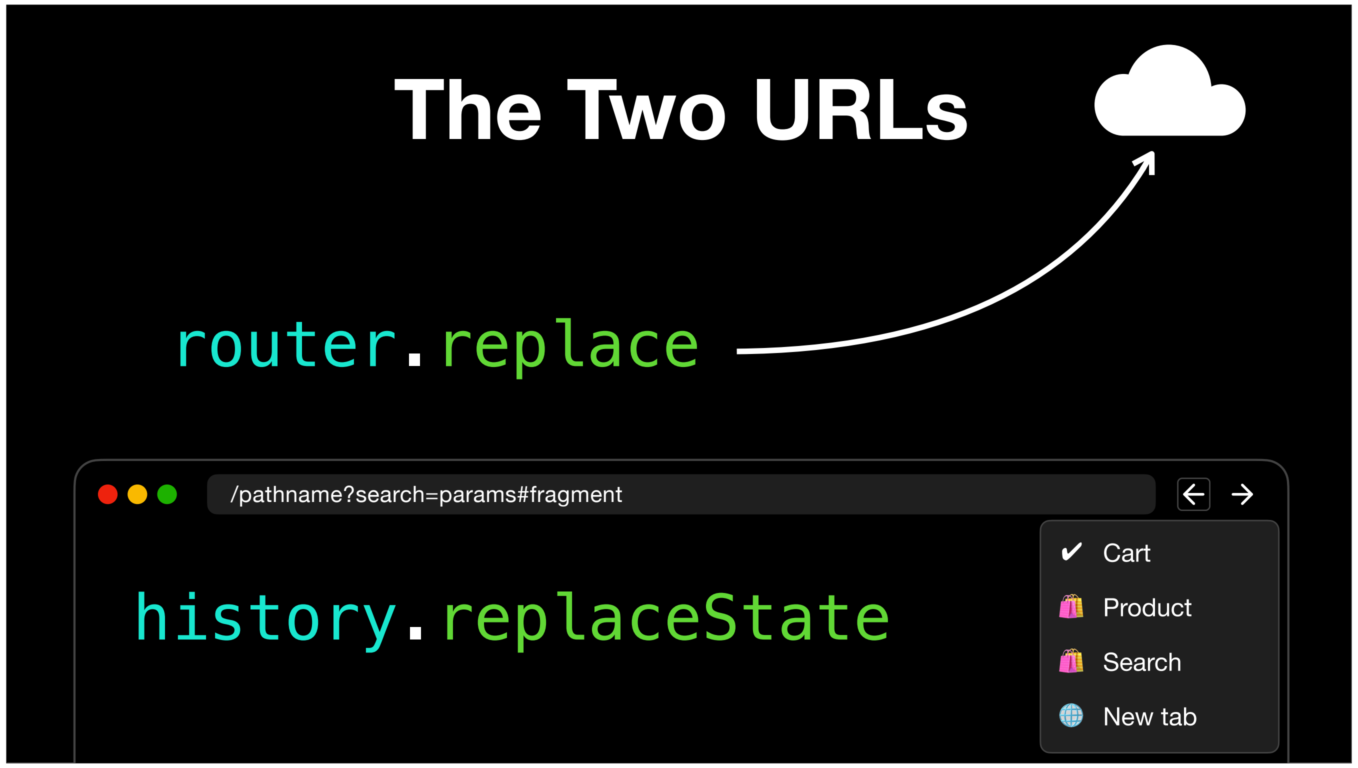
State manager because you'd use it the same way you would Zustand or a global state manager: you consume its hooks as far down the component tree as needed, and it lifts the state out into the URL for you.

And _for React_ because it works with most React frameworks

Demo

I could tell you more about it, but I can actually show you how it works.

We're going to look at a demo using both Next.js (app router) and React Router v7 (if you know Remix, this is essentially the same thing).



The Two URLs

```
router.replace
```

```
history.replaceState
```

(Back from the demo, explaining why clicking the button doesn't trigger network requests)

There are two ways of updating the URL: one is provided by our frameworks' router, and usually triggers network requests to the server to do server-side rendering.

But there is another, built in the browser: the History API, which only updates the URL client-side. This is what we call shallow routing, and nuqs uses this by default.

To opt-into server-side rendering, which is a side effect, you can opt-out of this shallow routing, and we're going to look at that with another demo.

Declarative
Un-opinionated
Composition

So to recap, we saw that nuqs uses a declarative approach to defining search params behaviours.

It is not particularly opinionated about where you declare these search params, you could do it near the components that use them, in shared files, or near the page that accepts them, and

Compose them by spreading out these definitions into larger objects.
Since the hooks are scoped to the search params they manage, you can isolate those declarations and nuqs will do the stitching for you.

Custom parsers & adapters
useTransition
RSC cache
Validation
Testing

Now there are a lot of other things we haven't had time to cover, like custom parsers to translate custom data types into beautiful representations in the URL, custom adapters for other frameworks (we have community adapters for Waku, Inertia, and some are in the works for Expo Router, for deep linking in React Native).

useTransition to get loading states on RSC updates,

the RSC cache to avoid prop-drilling search params in nested React Server Components (like a server-side Context replacement)

Validation to add runtime constraints to your data types, with validators like Zod, Valibot or Arktype

And testing for unit-testing your components and hooks without needing to mock your framework.

Questions, feedback, slides, docs, repository, and more:

go.47ng.com/react-paris

Thank you !

If you want to learn more, I invite you to scan that QR code or go to https://go.47ng.com/react-paris, you'll find the questions and feedback channel on Discord, and links to the docs, the repository, the slides, and how to find me on social media.

Thank you!