What’s in the box?

Boxing and unboxing objects into query params

Bartek Polańczyk | 27/11/2024

Agenda

- What? Where? Why?
- State management in 2024
- Why not Redux? Zustand? MobX?
- A bit on URLs and browsers
- Serialization - the basics
- Let’s BOX some data
- Let’s UNBOX something
- Conclusion

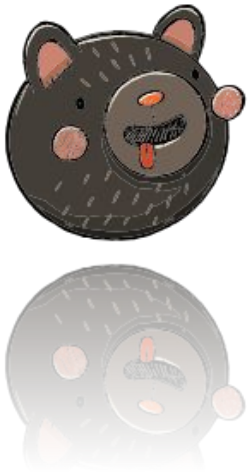
What? Where?

A complex page displaying analytics data

- “Analytics IA” project at ❄ - part of a new **Provider Experience** domain
- Asking questions how can we help Snowflake providers better understand their listings (and the revenue in the future)
- A lot of **read-only** visualizations
- Displayed data is modified by **interacting with filters**
- The state of the view needs to be persisted between re-renders and navigation events
- The filters **may or may not** be saved between different pages

Why?

Why did we need state management? Why did we decide to go with search params?



- Persisting filters -> needed a way to remember the state
- The ability to **pass the state down the component tree**
- The ability to **mutate the state**
- The ability to easily extend the solution with new filters/objects
  - How to serialize data?
- The ability to **save the data between re-renders**

- **Redux?** - opinionated, but hard to implement (and… Overkill?)
- **Zustand?** - simple, easy, but an added dependency
- **Jotai?** - primitive (words of authors)
- We just look for **something simple** - no data mutations, just fetching based on filters (but complex queries with a lot of filters)
- None of the above actually solved our problem

State management in 2024

State management libraries breakdown

Redux (+toolkit)

A solid framework, with much less boilerplate than earlier, good type system, relatively big and complex. Most UI devs know the basics

Zustand, Jotai, hooks

Simple to use and adapts, most are flux-compatible, small footprint. Easiest ones to add to an existing project

Mutable-based ones

MobX and similar. Intuitive, but hard to scale. The simplest API. A bit less mature than the others (e.g. devtools, common libraries)

High-level query libs

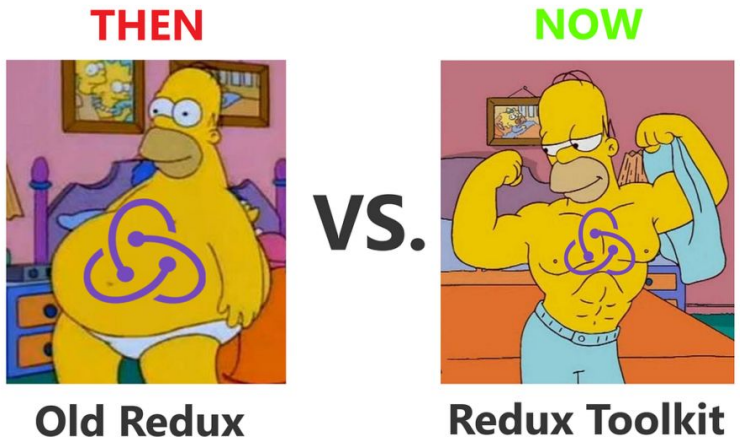
React Query, RTKQ. A lot of boilerplate, built on top of other opinionated solutions. Give a lot of flexibility and a lot of control over the data

Entities must not be multiplied beyond necessity

William of Ockham

Redux (+toolkit)

(almost) industry standard

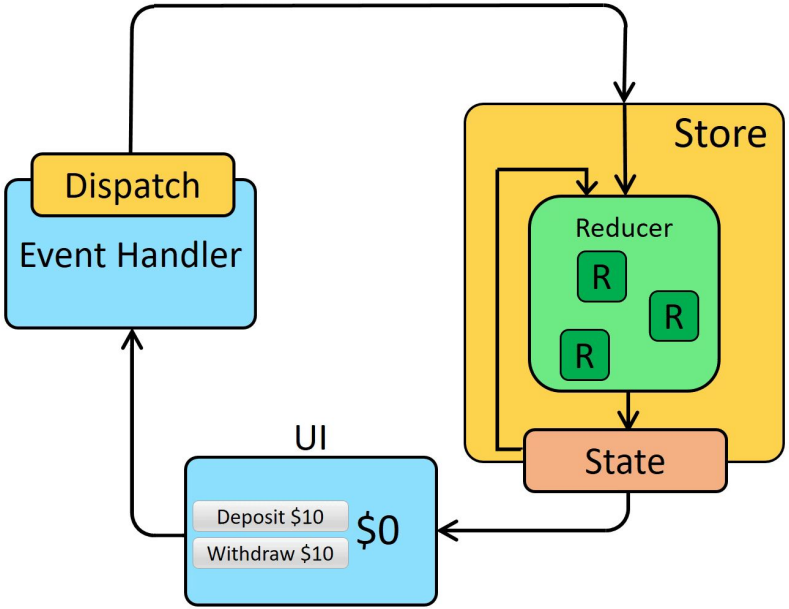


- A solid set of tools
- Ease of use (with Toolkit)
- Almost all standard use cases covered:
  - Actions + mutations
  - Data normalization
  - transformations
- Still - high entry threshold

- **NO** - because too complex for just synchronous data
- **NO** easy storing of the state (+plugins needed)

Flux/redux principle

Good idea/flow, but a bit too complex (for our use case)



Zustand, Jotai (+others)

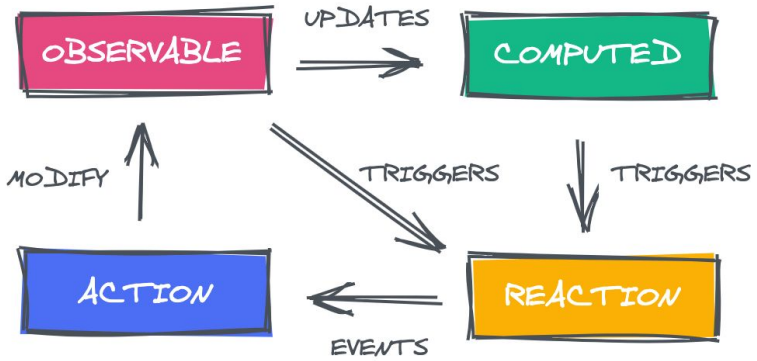
Simple, hook-based, declarative

- Two dominant approaches:
  - Flux principle simplified
  - Atom-based
- Easy to introduce (just add to the code!)
- Good docs, good alternatives to something bigger
- Small issues with complex schemas, but solvable
- (we actually used Zustand in our POC 😉)

- **NO** - because new dependency
- **NO -** because custom integration with search params needed

MobX

Mutable approach + observables - intuitive?



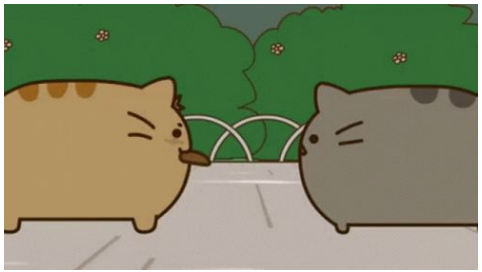
- A “honorable mention”
- Built on top of state management standards
- An additional level of abstraction
- Observable-based
  - Powerful
  - Flexible
- Devs need to learn it anew (a completely new concept)

- **NO** - too complex
- **NO** - too much to learn (but I personally like the idea!)

The power of simplicity

Leveraging the history API

- Modern History APIs allow for seamless URL updates
- Interacting it with React is almost given (React Router)
- Accessible via **first-class React citizens** (hooks)
- As long as you **encode the payload**, any data can be kept in the URL
- Best for small payloads (IDs, names, labels)
- The ability to **SHARE** (you can just copy the URL to share the state with someone else)
- But… There are **SOME** limitations
    - (remember about the server!)



URL limitations (spec vs browsers)

(based on StackOverflow)

Longer answer - first, the standards...

The original HTTP/1.1 specification, published in 1999, RFC 2616, said in section 3.2.1:

> The HTTP protocol does not place any a priori limit on the length of a URI. Servers MUST be able to handle the URI of any resource they serve, and SHOULD be able to handle URIs of unbounded length if they provide GET-based forms that could generate such URIs. A server SHOULD return 414 (Request-URI Too Long) status if a URI is longer than the server can handle (see section 10.4.15).

In 2014, this was obsoleted by an updated specification largely to match actual usage, RFC 7230; it suggests:

> Various ad hoc limitations on request-line length are found in practice. It is RECOMMENDED that all HTTP senders and recipients support, at a minimum, request-line lengths of 8000 octets.

Additional browser roundup

I tested the following against an Apache 2.4 server configured with a very large LimitRequestLine and LimitRequestFieldSize.

Browser	Address bar	document.location or anchor tag
Chrome	32779	>64k
Android	8192	>64k
Firefox	>300k	>300k
Safari	>64k	>64k
IE11	2047	5120
Edge 16	2047	10240

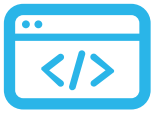
See also this answer from Matas Vaitkevicius below.

Is this information up to date?

This is a popular question, and as the original research is ~14 years old I'll try to keep it up to date: As of **Sep 2023**, the advice still stands. While modern browsers will support longer URLs, search engines do not so the headline figure remains "under 2000 chars"

What we need?

Building blocks



URL API

A way to interact with the URL. Given by React Router + hooks

UPDATE

A way to update the data and **serialize** it to the URL. All the values must become strings. **UPDATE** operation should only affect the updated data

DESERIALIZE

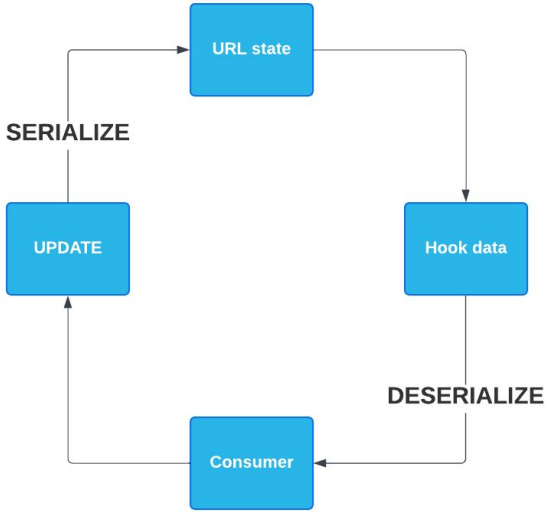
A way to **deserialize** the URL into a data type predictably and efficiently. The types must be preserved.

ISOLATE

Make sure that we are able to keep multiple data types in the URL without them conflicting with each other

The state lifecycle

original URL -> get -> update -> get -> update -> …



The state is derived from the URL **and only from it**

- Update **actually** updates the URL
- Each update triggers re-evaluation of the URL
- After that, we de-serialize the data and serve it to hook consumers
- We need to be able to serialize arbitrary data (arrays, objects, etc)

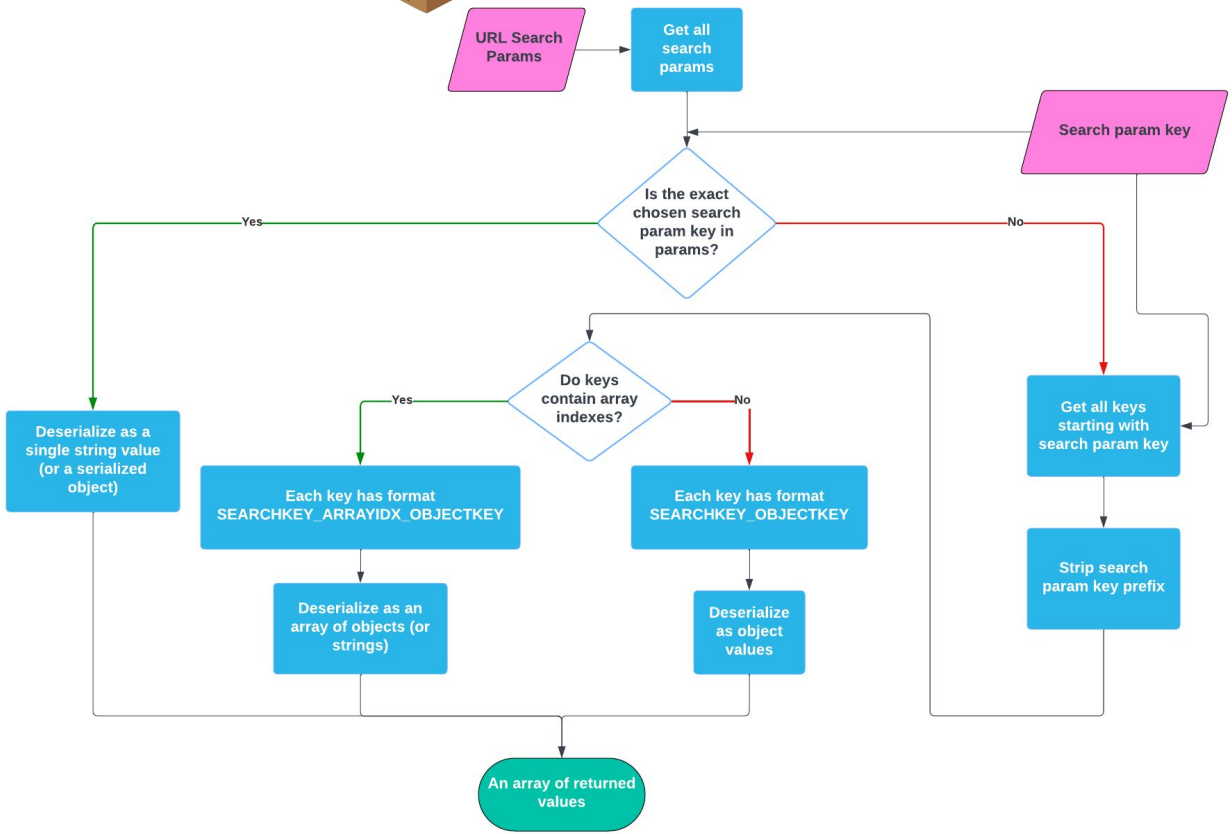
SERIALIZATION

“the process of converting the state of an object into a form that can be persisted or transported”

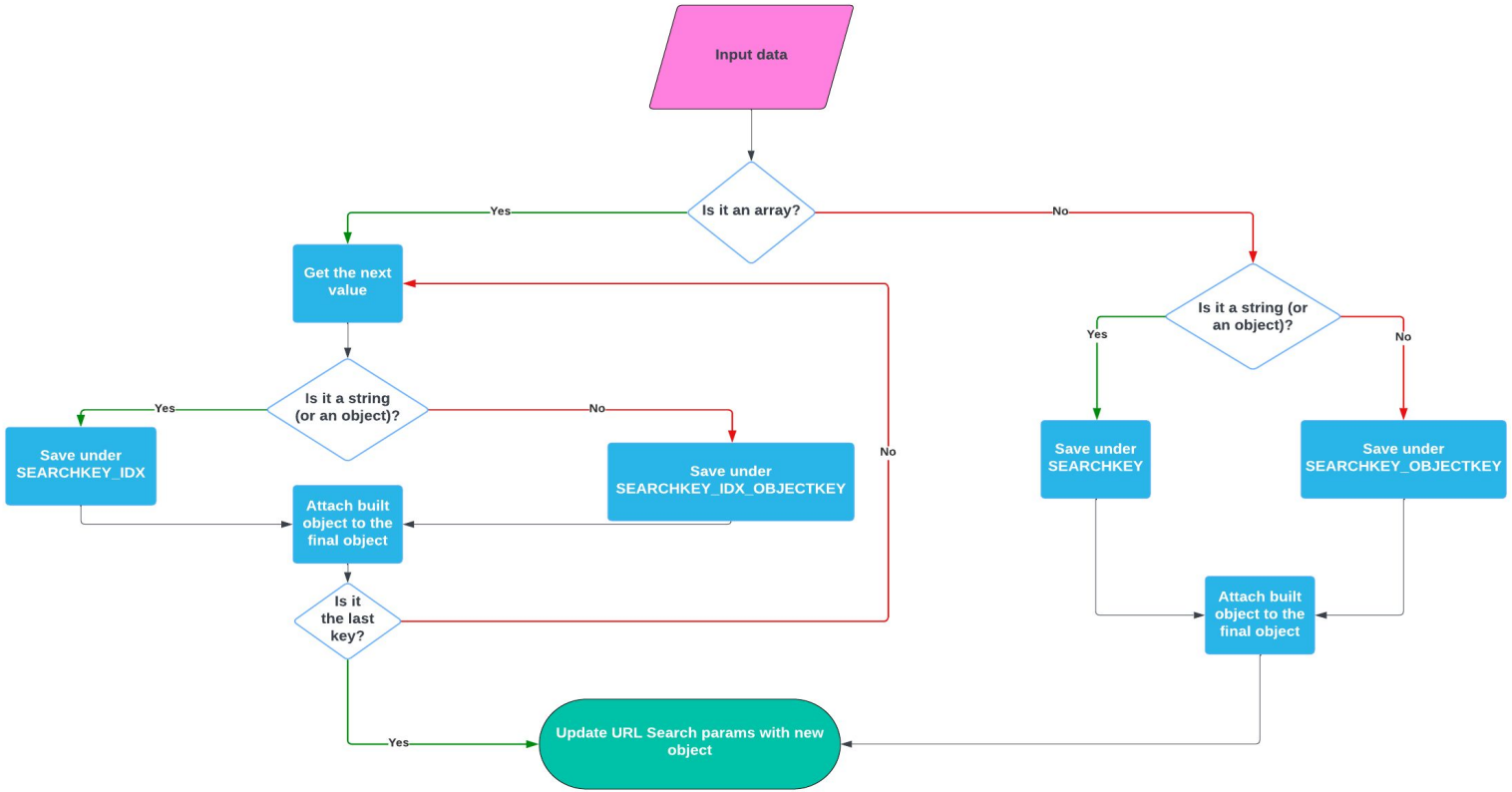
We need to be able to efficiently and predictably convert the object to URL-encoded string back and forth

Coding time!

Putting stuff in the 📦



Getting stuff out of the 📦



To sum up…

- If you just want to interact with the URL and manage the state, you may not need any additional library
- Current React Router APIs allow for robust interactions with search params
- You only need React Router DOM to 📦 stuff
- URL-based state is almost as good as one of other library-based ones (sync, persistence between reloads, bound to the view, can be persisted between the pages)

Useful links

More about our Marketplace (you can try out ❄️ for 30 days!)

- https://www.snowflake.com/en/data-cloud/marketplace

CodeSandbox:

- https://codesandbox.io/p/sandbox/typescript-react-router-forked-l7vk4g

Articles:

- https://www.reddit.com/r/reactjs/comments/1db5go3/learning_state_management_libraries_in_2024
- https://dev.to/nguyenhongphat0/react-state-management-in-2024-5e7l

Frameworks/libraries:

- https://jotai.org/
- https://github.com/pmndrs/zustand
- https://mobx.js.org/
- React Router v6: https://reactrouter.com/6.28.0

THANK YOU