Handling big data in React

**Data Structures**

The data that we render in our frontends originates from different sources: databases, files, APIs, and more.

Note that keep data structures flat when they reach frontend.

JSON data object with a nested structure. A big no-no!

* Keep your data structures flat.

**Parsing and Preparing Data**

When we receive — preferably flat-structured — data in our React application and store it in our state, we have to send it further down the line to our components in one way or other.

* Prepare the data outside of components.

Don’t:

* Send the original data to component
* Manipulate the data inside the component, and then Render component

Because:

What if another component — now or in the future — needs the manipulated data as well? What if the manipulation process is heavy? Or complex and extensive?

Make **dedicated selector-functions** that manipulate the data isolated from components and rendering. And make sure these functions are small and testable, and do not rely upon — or have —a state whatsoever. They should not read our state; they should receive data from our state as parameters.

A selector is a function that receives a subset of the data in state, optionally manipulates this data, and then returns it.

**Memoizing selector functions**

memoize the results from these function calls, especially if the manipulation process is heavy.

React application can achieve unbelievable performance when doing so, even with large amounts of data with useMemo and useCallback.

Take this form of caching outside of components, so it can be re-used, tested in isolation, and optimized in general.

[Reselect](https://www.npmjs.com/package/reselect) is a great tool for this, and there are many others.

**Rendering Data**

And then we reach a point where we have to render the data. A large amount of data. If it’s tabular and/or chronological data we could distinguish between:

* **Infinite scrolling** — load more data in the same view
* **Paginated data** — separate views for each page

For many corporations and business users — paginated data is recognizable and preferred.

If we are creating a casual or social application, or presenting a continuous flow of data in any other form, infinite rendering can be useful and user-friendly.

One more comment about tabular data:

*Show the full amount of data for the specific record in a separate view pane on the side of the screen, and in a dedicated area at a fixed position, when the user hovered the row with their mouse cursor. It was a bit of a gamble, but it worked. And the users, despite not being used to it in the first place, loved it.*

***Rendering many rows***

Don’t just simply and blindly throw them in DOM. Not every user of application has a top-notch machine as many developers do.

Consider lazy loading the rows that are currently not visible. There are nice tools for this, such as [react-window](https://www.npmjs.com/package/react-window) (~500k downloads per week) or its predecessor [react-virtualized](https://www.npmjs.com/package/react-virtualized) (~700k downloads per week).

**Dynamic Data**

Sometimes our data is highly dynamic — as in, it changes often. This brings a whole other set of challenges with it.

## What is virtualization?

**Virtualization** as a scrolling technique is the rendering process that keeps track of the scrolling position of a user, and visually displays only the DOM contents within that current viewport’s position. This technique simply involves handling large dataset in an infinite scroll pattern but incrementally load and render the long lists of data just as the data enters the viewport. This concept presents developers with all the performance advantages of **pagination** while at the same time providing the UX benefits gained from **infinite scrolling**.

Using virtualization is a common solution used in rendering large amount of posts in social media applications, delivering large news data in news media applications, and also in bookkeeping (Accounting) software. The endless two dimensional tile scrolling in Map applications also involves virtual scrolling which aids easy display of the constant changing locations of the users when using it. And virtualization supports displaying these large datasets in grid format or list (spreadsheets-like) format.

[React-window](https://react-window.now.sh/#/examples/list/fixed-size) is a simplified version of react-virtualized package for easily implementing virtualization in React applications through the APIs (components) provided by the library.

4 react-window components:

* **FixedSizeList** — This component helps render elements of fixed size defined through the index prop. Common available props that can be passed into this component : useIsScrolling, itemCount , itemSize, itemData, innerElementType, etc
* **VariableSizeList** — This component accepts the same props as the FixedSizeList component, but with additional props like an estimatedItemSize, and itemSize which takes in a function in this case and returns the size of a item in the direction being windowed
* **FixedSizeGrid** — With this component, we’re rendering using dimensions: vertical (columns) and horizontal (rows) directions. Another difference is that we’ll have to add a data’s count.

Common props are rowCount, rowWidth, columnCount, columnWidth, height, width e.t.c

* **VariableSizeGrid** — This component is as FixedSizeGrid, with additional props: columnWidth & rowHeight which accepts functions as their prop values.

## Open Source Session Replay

Debugging a web application in production may be challenging and time-consuming. [OpenReplay](https://github.com/openreplay/openreplay" \t "_blank) is an Open-source alternative to FullStory, LogRocket and Hotjar. It allows you to monitor and replay everything your users do and shows how your app behaves for every issue. It’s like having your browser’s inspector open while looking over your user’s shoulder. OpenReplay is the only open-source alternative currently available.