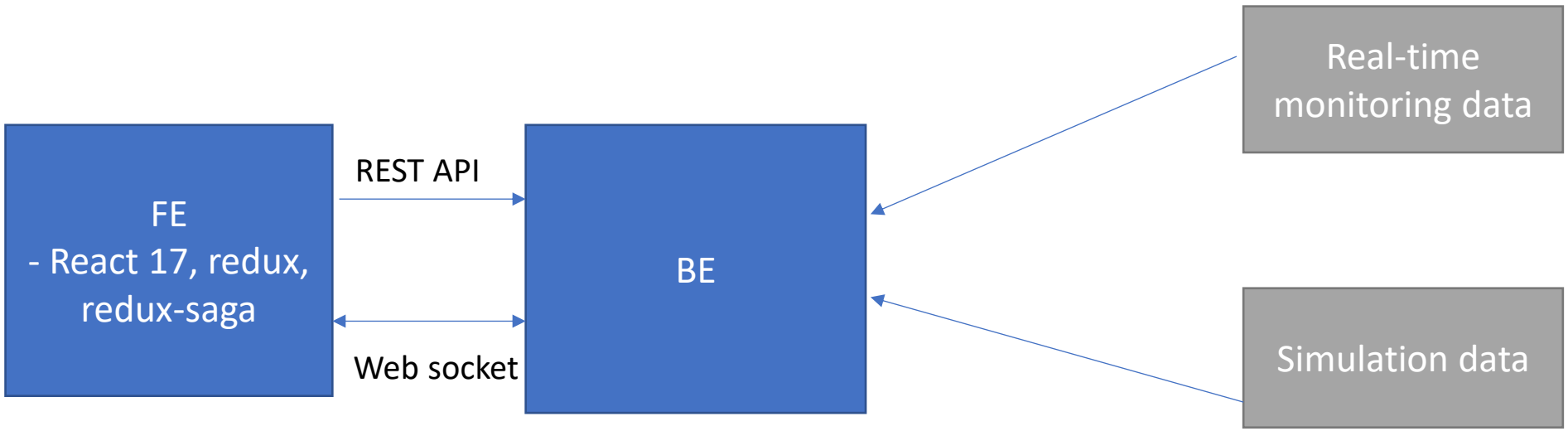
8 ways to fix the performance of react app

in monitoring applications

Aino Haikala, 2022



- Client could be used from a remote island with **bad network connections** for **occasional checks on mobile or continuous monitoring on screen**.

Problem 1: Problem was not seen in time!

- The requirements management was lacking
- Team's understanding of the conditions in which the product would be used was lacking.
- Team didn't have the domain knowledge to create realistic projects.

=> Developers used their own test projects, which were too small for the problems to show. Lots of more data and datapoints were needed to get these problems!

FIX:

- "Real" projects were added to Confluence and EVERYONE was expected to use them.
- Team collaborated with the engineering team so that we could get mock test projects to emulate the future needs.
- Knowledge sharing in the team how to use network throttling and other performance tools.

Problem 2: It took several seconds for user to see anything on the screen

- We were loading more stuff than was needed on one screen.
- No attention hadn’t been paid to the loading order.

FIX:

- Adding an initializerSaga, which handled loading the data on all screens.
- Loading only information needed to display the current view.

Problem 3: It was slow to navigate from view to another

- FE and BE used slightly different data and data conversions were made in FE.
- BE responses had unneeded duplicate data.

FIX:

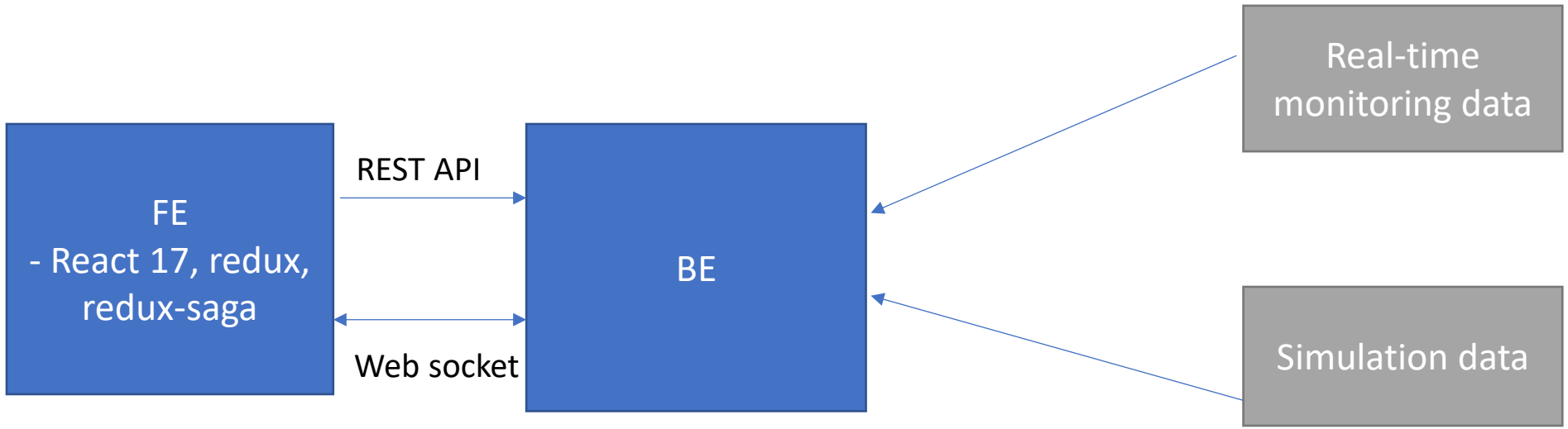
- Unifying the some of the data in BE and FE. Data conversions were moved in BE.
- Pruning unneeded things out from BE responses and using the normalized data in FE.
  - https://redux.js.org/tutorials/essentials/part-6-performance-normalization

Problem 4: On long term runs application started to slow down

- Monitoring app can be running on some screen for days!
- Incoming data is probably the most interesting. Yet, user may want to view long term data.
- More data was just coming in => on long term runs memory started to bloat.

FIX:

- Flushing the old unneeded data on regular intervals.
- Data decimation algorithm was used to reduce the number of data points, still retaining the information as much as possible.



- Websocket was used to get datapoints.
  - Real-time monitoring data: maybe once / second.
  - Simulation data: as fast as simulations could run.

Problem 5: We were blocking the UI with long synchronous tasks

- The handling of incoming datapoints had an inefficient synchronous loop structure.
- What the heck is event loop anyway: https://www.youtube.com/watch?v=8aGhZQkoFbQ
- https://javascript.info/event-loop

FIX:

- Rethinking how to do the storing without looping everything unnecessarily => this reduced handling of one data point batch to 1/10 of time.
- This was one of my biggest surprises. We got a significant change to user experience by rewriting 10 lines of code.

Problem 6: We were blocking the UI with crazy amount of re-rendering

- In React, rule of thumb is: (Mostly) Everything re-renders.
  - https://blog.isquaredsoftware.com/2020/05/blogged-answers-a-mostly-complete-guide-to-react-rendering-behavior/
  - https://alexsidorenko.com/blog/react-render-always-rerenders/
  - React lifecycle
- We had lots of components doing all sorts of rendering calculations on every store change.
  - One of our most time-taking component was a svg link icons which never changed.
- When running simulations store changed multiple times / second.
- **We are talking about pre-react 18 world here!** (Automatic batching in 18 may change things)

FIX:

- Rethinking when we want to fetch the data from store to components.
- Components get only the props they need and those props stay the same if they don't really change (Note this when using objects or functions as props)
- Memoizing components.
- Adding reselect on the mix to memorize selectors. Recoil POC, but didn't have time.

Problem 6 continues…

FIX:

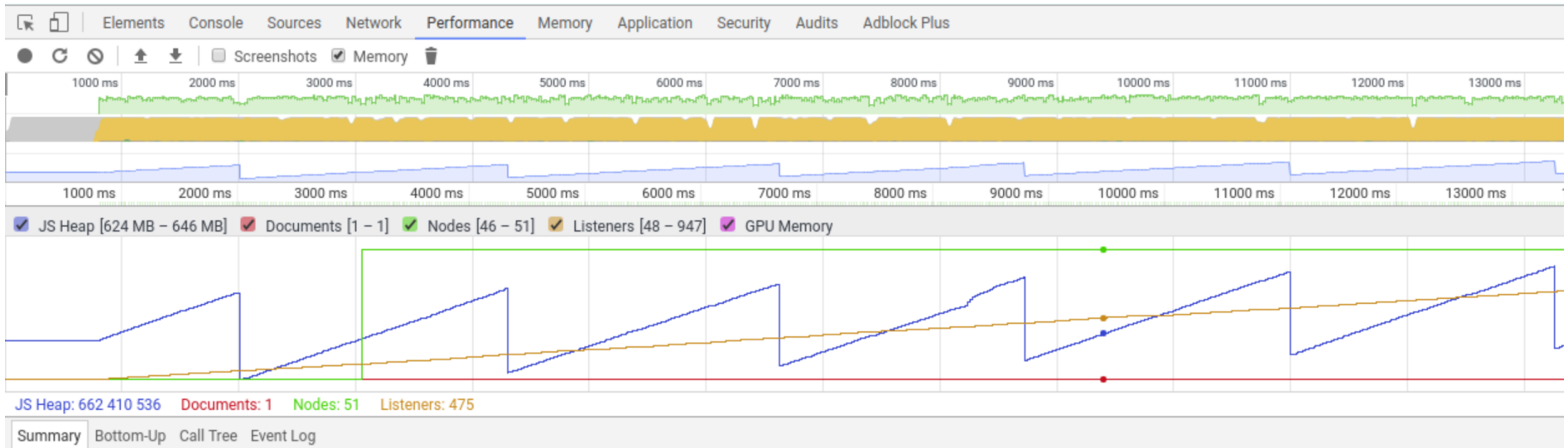
- BE takes into account what kind of data it sends to the FE and when. Not just pushing data without batching!
- FE doesn’t make extra store calls.

Problem 7: Hovering timeline was slowing down the whole UI

- When user hovered timeline, we didn’t have any delay for firing the next event.
- Event => store change => rendering whole UI again.

FIX:

- Adding timeouts before firing event again.
  - Keywords: debouncing and throttling
- And by the way… we were also not cleaning up event listeners from useEffect hooks, which caused a leak.


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits
Adblock Plus
Screenshots
Memory
1000 ms
2000 ms
3000 ms
4000 ms
5000 ms
6000 ms
7000 ms
8000 ms
9000 ms
10000 ms
11000 ms
12000 ms
13000 ms
JS Heap [624 MB – 646 MB]
Documents [1 – 1]
Nodes [46 – 51]
Listeners [48 – 947]
GPU Memory
JS Heap: 662 410 536
Documents: 1
Nodes: 51
Listeners: 475
Summary
Bottom-Up
Call Tree
Event Log

Problem 8: When browser tab had been inactive, the app became unresponsive for a while

- How tabs behave when they are hidden depends on browser a lot.
- This was a really difficult problem to investigate.
- The theory: handling data slows down so much when tab is inactive / frozen that FE gets behind. When user comes back, a huge amount of data needs handling.

FIX:

- We didn’t have time to fix this at that point.
- I believe the way to fix this would be closing the websocket when tab has been inactive for a way and fetch a decimated data batch when user comes back.
  - https://developer.chrome.com/blog/page-lifecycle-api/#developer-recommendations-for-each-state

Don’t panic, keep on coding

- We had problems which may not be a problem at all in your application.
- Making performance improvements later was totally doable. But… keep your architecture and data model nice and clean.
- How to check:
  - React browser plugin has a performance tool.
  - Browsers have performance tools (love you chrome <3)
  - Follow how memory consumption behaves.
  - Add network / CPU throttling in the mix.
  - In our case, redux browser plugin affected negatively in performance, even if it was closed at the moment.
- When making performance tests use a special performance build!
- “It’s working on my monster machine!” …but is it enough?