Five ways AirSHIFT improved their React app's runtime performance

- Slow response, less productivity
- 1. Virtualize large tables
  - Results
- 2. Audit with User Timing API
  - Results
- 3. Lazy load components and move expensive logic to web workers
  - Results
- 4. Setting a performance budget
  - Results
- 5. Performance hackathons
  - Results
- Summary

Website performance is not just about load time. It is critical to provide a fast and responsive experience to users, especially for productivity desktop apps which people use everyday. The engineering team at Recruit Technologies went through a refactoring project to improve one of their web apps, AirSHIFT, for better user input performance. Here's how they did it.

Slow response, less productivity

AirSHIFT is a desktop web application that helps store owners, like restaurants and cafes, to manage the shift work of their staff members. Built with React, the single page application provides rich client features including various grid tables of shift schedules organized by day, week, month

and more.

A screenshot of the AirSHIFT web app.

As the Recruit Technologies engineering team added new features to the AirSHIFT app, they started seeing more feedback around slow performance. The engineering manager of AirSHIFT, Yosuke Furukawa, said:

> In a user research study, we were shocked when one of the store owners said she would leave her seat to brew coffee after clicking a button, just to kill time waiting for the shift table to load.

After going through the research, the engineering team realized that many of their users were trying to load massive shift tables on low spec computers, such as a 1 GHz Celeron M laptop from 10 years ago.

Endless spinner on low end devices.

The AirSHIFT app was blocking the main thread with expensive scripts, but the engineering team didn't realize how expensive the scripts were because they were developing and testing on rich spec computers with fast Wi-Fi connections.

A chart that shows the app's runtime activity.

When loading the shift table, around 80% of the load time was consumed by running scripts.

After profiling their performance in Chrome DevTools with CPU and network throttling enabled, it became clear that performance optimization was needed. AirSHIFT formed a task force to tackle this issue. Here are 5 things they focused on to make their app more responsive to user input.

1. Virtualize large tables

Displaying the shift table required multiple expensive steps: constructing the virtual DOM and rendering it on screen in proportion to the number of staff members and time slots. For example, if a restaurant had 50 working members and wanted to check their monthly shift schedule, it would be a table of 50 (members) multiplied by 30 (days) which would lead to 1,500 cell components to render. This is a very expensive operation, especially for low spec devices. In reality, things were worse. From the research they learned there were shops managing 200 staff members, requiring around 6,000 cell components in a single monthly table.

To reduce the cost of this operation, AirSHIFT virtualized the shift table. The app now only mounts the components within the viewport and unmounts the off-screen components.

An annotated screenshot that demonstrates that AirSHIFT used to render content outside of the viewport.

Before: Rendering all the shift table cells.

An annotated screenshot that demonstrates that AirSHIFT now only renders content that's visible in the viewport.

After: Only rendering the cells within the viewport.

In this case, AirSHIFT used react-virtualized as there were requirements around enabling complex two dimensional grid tables. They are also exploring ways to convert the implementation to use the lightweight react-window in the future.

Results

Virtualizing the table alone reduced scripting time by 6 seconds (on a 4x CPU slowdown + Fast 3G throttled Macbook Pro environment). This was the most impactful performance improvement in the refactoring project.

An annotated screenshot of a Chrome DevTools Performance panel recording.

Before: Around 10 seconds of scripting after user input.

Another annotated screenshot of a Chrome DevTools Performance panel recording.

After: 4 seconds of scripting after user input.

2. Audit with User Timing API

Next, the AirSHIFT team refactored the scripts that run on user input. The flame chart of Chrome DevTools makes it possible to analyze what's actually happening in the main thread. But the AirSHIFT team found it easier to analyze application activity based on React's lifecycle.

React 16 provides its performance trace via the User Timing API, which you can visualize from the Timings section of Chrome DevTools. AirSHIFT used the Timings section to find unnecessary logic running in React lifecycle events.

The Timings section of the Performance panel of Chrome DevTools.

React's User Timing events.

Related article: Profiling Components with the Chrome Performance Tab

Results

The AirSHIFT team discovered that an unnecessary React Tree Reconciliation was happening right before every route navigation. This meant that React was updating the shift table unnecessarily before navigations. An unnecessary Redux state update was causing this issue. Fixing it saved around 750 ms of scripting time. AirSHIFT made other micro optimizations as well which eventually led to a 1 second total reduction in scripting time.

3. Lazy load components and move expensive logic to web workers

AirSHIFT has a built-in chat application. Many store owners communicate with their staff members via the chat while looking at the shift table, which means that a user might be typing a message while the table is loading. If the main thread is occupied with scripts that are rendering the table, user input could be janky.

To improve this experience, AirSHIFT now uses React.lazy and Suspense to show placeholders for table contents while lazily loading the actual components.

The AirSHIFT team also migrated some of the expensive business logic within the lazily loaded components to web workers. This solved the user input jank problem by freeing up the main thread so that it could focus on responding to user input.

Typically developers face complexity in using workers but this time Comlink did the heavy lifting for them. Below is the pseudo code of how AirSHIFT workerized one of the most expensive operations they had: calculating total labor costs.

In App.js, use React.lazy and Suspense to show fallback content while loading

```
/** App.js */
import React, { lazy, Suspense } from 'react'

// Lazily loading the Cost component with React.lazy
```

```
const Hello = lazy(() => import('./Cost'))

const Loading = () => (
  <div>Some fallback content to show while loading</div>
)

// Showing the fallback content while loading the Cost component by Suspense
export default function App({ userInfo }) {
   return (
    <div>
      <Suspense fallback={<Loading />}>
        <Cost />
      </Suspense>
    </div>
  )
}
```

In the Cost component, use comlink to execute the calc logic

```
/** Cost.js */
import React from 'react';
import { proxy } from 'comlink';

// import the workerlized calc function with comlink
const WorkerlizedCostCalc = proxy(new Worker('./WorkerlizedCostCalc.js'));
export default async function Cost({ userInfo }) {
  // execute the calculation in the worker
  const instance = await new WorkerlizedCostCalc();
  const cost = await instance.calc(userInfo);
  return <p>{cost}</p>;
}
```

Implement the calculation logic that runs in the worker and expose it with comlink

```
// WorkerlizedCostCalc.js
import { expose } from 'comlink'
import { someExpensiveCalculation } from './CostCalc.js'

// Expose the new workerlized calc function with comlink
expose({
```

```
  calc(userInfo) {
    // run existing (expensive) function in the worker
    return someExpensiveCalculation(userInfo);
  }
}, self);
```

Related article: React + Redux + Comlink = Off-main-thread

Results

Despite the limited amount of logic they workerized as a trial, AirSHIFT shifted around 100 ms of their JavaScript from the main thread to the worker thread (simulated with 4x CPU throttling).

A screenshot of a Chrome DevTools Performance panel recording that shows that scripting is now occurring on a web worker rather than the main thread.

AirSHIFT is currently exploring whether they can lazy load other components and offload more logic to web workers to further reduce jank.

4. Setting a performance budget

Having implemented all of these optimizations, it was critical to make sure that the app remains performant over time. AirSHIFT now uses bundlesize to not exceed the current JavaScript and CSS file size. Aside from setting these basic budgets, they built a dashboard to show various percentiles of the shift table loading time to check whether the application is performant even in non-ideal conditions.

- The script completion time for every Redux event is now measured
- Performance data is collected in Elasticsearch
- 10th, 25th, 50th, and 75th percentile performance of each event is visualized with Kibana

AirSHIFT is now monitoring the shift table loading event to make sure it completes in 3 seconds for the 75th percentile users. This is an unenforced budget for now but they are considering auto-notifications via Elasticsearch when they exceed their budget.

> *A chart showing that the 75th percentile completes in around 2500 ms, the 50th percentile in around 1250 ms, the 25th percentile in around 750 ms, and the 10th percentile in around 500 ms.*
>
> The Kibana dashboard showing daily performance data by percentiles.

Related article: Performance budgets 101

Results

From the graph above, you can tell that AirSHIFT is now mostly hitting the 3 seconds budget for 75th percentile users and also loading the shift table within a second for 25th percentile users. By capturing RUM performance data from various conditions and devices, AirSHIFT can now check whether a new feature release is actually affecting the application's performance or not.

5. Performance hackathons

Even though all of these performance optimization efforts were important and impactful, it's not always easy to get engineering and business teams to prioritize non-functional development. Part of the challenge is that some of these performance optimizations can't be planned. They require experimentation and a trial-and-error mindset.

AirSHIFT is now conducting internal 1-day performance hackathons to let engineers focus only on performance related work. In these hackathons they remove all constraints and respect the engineers' creativity, meaning any implementation that contributes to speed is worth considering. To accelerate the hackathon, AirSHIFT splits the group into small teams and each team competes to see who can get the biggest Lighthouse performance score improvement. The teams get very competitive! □□

Photos of the hackathon.

Results

The hackathon approach is working well for them.

- Performance bottlenecks can be easily detected by actually trying out multiple approaches during the hackathon and measuring each with Lighthouse.
- After the hackathon, it's rather easy to convince the team which optimization they should be prioritizing for production release.
- It's also an effective way of advocating the importance of speed. Every participant can understand the correlation between how you code and how it results in performance.

A good side effect was that many other engineering teams within Recruit got interested in this hands-on approach and the AirSHIFT team is now facilitating multiple speed hackathons within the company.

Summary

It was definitely not the easiest journey for AirSHIFT to work on these optimizations but it certainly paid off. Now AirSHIFT is loading the shift table within 1.5 sec in median which is a 6x improvement from their performance before the project.

After the performance optimizations launched, one user said:

> “Thank you so much for making the shift table load fast. Arranging the shift work is so much more efficient now.

□□□ #3
□ □□□ □□□ 2026-10-31 21:52:49 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:09 UTC