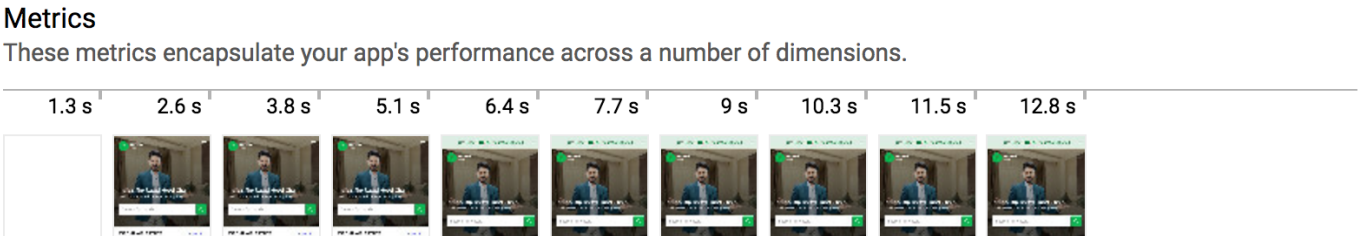
A React And Preact Progressive Web App Performance Case Study: Treebo

Addy Osmani
Sep 13, 2017 · 14 min read

Authors: *Treebo: Lakshya Ranganath, Chrome: Addy Osmani*

Treebo is India's top rated budget hotel chain, operating in a segment of the travel industry worth $20 billion. They recently shipped a new Progressive Web App as their default mobile experience, initially using React and eventually switching to Preact in production.

What they saw compared to their old mobile site was **a 70%+ improvement in time to first paint , 31% improvement in time-to-interactive. and loaded in under 4 seconds over 3G** for many typical visitors and on their target hardware. It was interactive in under 5s using WebPageTest's slower 3G emulation in India.



Switching from React to Preact was responsible for a 15% improvement in time-to-interactive alone. You can check out Treebo.com for their full experience but today we would like to dive into some of the technical journey that made shipping this PWA possible.

Treebo's Progressive Web App

A Performance Journey

The old mobile site

Treebo's old mobile site was powered by a monolithic Django setup. Users had to wait for a server side request for every page transition on the website. This original setup had a first paint time of 1.5s, a first meaningful paint time of 5.9s and was first interactive in 6.5s.

A basic single-page React app

For their first iteration of the rewrite Treebo started off with a **Single Page Application** built using React and a simple webpack setup.

You can take a look at the actual code used below. This generates some simple (monolithic) JavaScript and CSS bundles.

```
entry: {
  main: './client/index.js',
},
output: {
  path: path.resolve('./build/client'),
  filename: 'js/[name].[chunkhash:8].js',
},
module: {
  rules: [
    { test: /\.js$/, exclude: /node_modules/, use: ['babel-loader'] },
    { test: /\.css$/, loader: ExtractTextPlugin.extract({ fallback: ['style-loader'], use: ['cs
  ],
}
new ExtractTextPlugin('css/[name].[contenthash:8].css'),
```

webpack.js hosted with ♡ by GitHub view raw

This experience had a first paint of 4.8s, was first interactive in about 5.6s and their meaningful header images painted in about 7.2s.

Server-side Rendering

Next, they went about optimizing their first paint a little so they tried out **Server-side Rendering. It's important to note, server side rendering is not free. It optimizes one thing at the cost of another.**

> *With server-side rendering, your server's response to the browser is the HTML of your page that is ready to be rendered so the browser can start rendering without having to wait for all the JavaScript to be downloaded and executed.*

Treebo used React's renderToString() to render components to an HTML string and injecting state for the application on initial boot up.

```
const serverRenderedHtml = async (req, res, renderProps) => {
  const store = configureStore();
  //call, wait, and set api responses into redux store's state (ghub.io/redux-connect)
  await loadOnServer({ ...renderProps, store });
  //render the html template
  const template = html(
    renderToString(
      <Provider store={store} key="provider">
        <ReduxAsyncConnect {...renderProps} />
      </Provider>,
    ),
    store.getState(),
  );
  res.send(template);
};

const html = (app, initialState) => `
  <!doctype html>
  <html lang="en">
    <head>
      <link rel="stylesheet" href="${assets.main.css}">
    </head>
    <body>
      <div id="root">${app}</div>
      <script>window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}</script>
      <script src="${assets.main.js}"></script>
    </body>
  </html>
```

```
29    `;
```

reactMiddleware.js hosted with ♡ by GitHub view raw

In Treebos' case, using server side rendering dropped their first paint time to 1.1s and first meaningful paint time down to 2.4s — this improved how quickly users *perceived* the page to be ready, they could read content earlier on and it performed slightly better at SEO in tests. But the downside was that it had a pretty negative impact on time to interactive.

Although users could *view* content, the main thread got pegged while booting up their JavaScript and just hung there.

With SSR, the browser had to fetch and process a much larger HTML payload than before and then still fetch, parse/compile and execute the JavaScript. It was effectively doing more work.

This meant that first interactive happened about 6.6s, regressing.

SSR can also push TTI back by locking up the main thread on lower-end devices.

Code-splitting & route-based chunking

The next thing Treebo looked at was **route-based chunking** to help bring down their time-to-interactive numbers.

> *Route-based chunking aims to serve the minimal code needed to make a route interactive, by code-splitting the routes into "chunks" that can be loaded on demand. This encourages delivering resources closer to the granularity they were authored in.*

What they did here was they split out their vendor dependencies, their Webpack runtime manifests and their routes — into separate chunks.

```
//add the webpackManifest and vendor script files to your html
<body>
  <div id="root">${app}</div>
  <script>window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}</script>
  <script src="${assets.webpackManifest.js}"></script>
  <script src="${assets.vendor.js}"></script>
  <script src="${assets.main.js}"></script>
</body>
```

reactMiddleware.js hosted with ♡ by GitHub view raw

```
import 'redux-pack';
import 'redux-segment';
import 'redux-thunk';
import 'redux';
// import other external dependencies
```

vendor.js hosted with ♡ by GitHub view raw

```
entry: {
  main: './client/index.js',
  vendor: './client/vendor.js',
},
new webpack.optimize.CommonsChunkPlugin({
  names: ['vendor', 'webpackManifest'],
  minChunks: Infinity,
}),
```

webpack.js hosted with ♡ by GitHub view raw

```
<Route
  name="landing"
  path="/"
  getComponent={
    (_, cb) => import('./views/LandingPage/LandingPage' /* webpackChunkName: 'landing' */)
      .then((module) => cb(null, module.default))
      .catch((error) => cb(error, null))
  }
</Route>
```

routes.js hosted with ♡ by GitHub view raw

```
//extract css from all the split chunks into main.hash.css
new ExtractTextPlugin({
  filename: 'css/[name].[contenthash:8].css',
  allChunks: true,
}),
```

webpack.js hosted with ♡ by GitHub view raw

This reduced the time to first interactive down to 4.8s. Awesome!

The only downside was that it started the current route's JavaScript download only after their initial bundles were done executing, which was also not ideal.

But it did at least have some positive impact on the experience. For route-based code-splitting and this experience, they're doing something a little bit more implicit. They're using React Router's declarative support for getComponent with a webpack import() call to asynchronously load in chunks.

The PRPL Performance Pattern

Route-based chunking is a great first step in intelligently bundling code for more granular serving and caching. Treebo wanted to build on this and looked to the **PRPL pattern** for inspiration.

> *PRPL is a pattern for structuring and serving PWAs, with an emphasis on the performance of app delivery and launch.*

PRPL stands for:

- **Push** critical resources for the initial URL route.

- **Render** initial route.
- **Pre-cache** remaining routes.
- **Lazy-load** and create remaining routes on demand.

A PRPL visualization by Jimmy Moon

The “Push” part encourages serving an unbundled build designed for server/browser combinations that support HTTP/2 to deliver the resources the browser needs for a fast first paint while optimizing caching. The delivery of these resources can be triggered efficiently using `<link rel="preload">` or HTTP/2 Push.

Treebo opted to use <link rel=”preload” /> to preload the current route’s chunk ahead of time. This had the impact of dropping their first interactive times since the current route’s chunk was already in the cache when webpack made a call to fetch it after their initial bundles finished executing. It shifted the time down a little bit and so the first interactive happened at the 4.6s mark.

The only con they had with preload is that it's not implemented cross-browser. However, there's an implementation of link rel preload in Safari Tech Preview. I'm hopeful that it's going to land and stick this year. There's also work underway to try landing it in Firefox.

HTML Streaming

> *One difficulty with renderToString() is that it is synchronous, and it can become a performance bottleneck in server-side rendering of React sites. Servers won't send out a response until the entire HTML is created. When web servers stream out their content instead, browsers can render pages for users before the entire response is finished. Projects like react-dom-stream can help here.*

To improve perceived performance and introduce a sense of progressive rendering to their app, Treebo looked to **HTML Streaming**. They would stream the head tag with link rel preload tags set up to early preload in their CSS and their JavaScripts. They then perform their server side rendering and send the rest of the payload down to the browser.

The benefit of this was that resource downloads started earlier on, dropping their first paint to 0.9s and first interactive to 4.4s. The app was consistently interactive around the 4.9/5 second mark.

The downside here was that it kept the connection open for a little bit longer between the client and server, which could have issues if you run into longer latency times. For HTML streaming, Treebo defined an early chunk with the <head> content, then they have the main content and the late chunks. All of these being injected into the page. This is what it looks like:

```
earlyChunk(route) {
  return `
    <!doctype html>
    <html lang="en">
      <head>
        <link rel="stylesheet" href="${assets.main.css}">
        <link rel="preload" as="script" href="${assets.webpackManifest.js}">
        <link rel="preload" as="script" href="${assets.vendor.js}">
        <link rel="preload" as="script" href="${assets.main.js}">
        ${!assets[route.name] ? '' : `<link rel="preload" as="script" href="${assets[route.name
      </head>`;
},
lateChunk(app, head, initialState) {
  return `
      <body>
        <div id="root">${app}</div>
        <script>window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}</script>
        <script src="${assets.webpackManifest.js}"></script>
        <script src="${assets.vendor.js}"></script>
        <script src="${assets.main.js}"></script>
      </body>
    </html>
  `;
},
```

html.js hosted with ♡ by GitHub view raw

```
const serverRenderedChunks = async (req, res, renderProps) => {
  const route = renderProps.routes[renderProps.routes.length - 1];
  const store = configureStore();
  //set the content type since you're streaming the response
  res.set('Content-Type', 'text/html');
```

```
  //flush the head with css & js resource tags first so the download starts immediately
  const earlyChunk = html.earlyChunk(route);
  res.write(earlyChunk);
  res.flush();
  //call & wait for api's response, set them into state
  await loadOnServer({ ...renderProps, store });
  //flush the rest of the body once app the server side rendered
  const lateChunk = html.lateChunk(
    renderToString(
      <Provider store={store} key="provider">
        <ReduxAsyncConnect {...renderProps} />
      </Provider>,
    ),
    Helmet.renderStatic(),
    store.getState(),
    route,
  );
  res.write(lateChunk);
  res.flush();
  //let client know the response has ended
  res.end();
};
```

Effectively, the early chunk has got their rel=preload statements for all of their different script tags. The late chunk has got the server rendered html and anything that's going to include state or actually use the JavaScript that's being loaded in.

Inlining critical-path CSS

> *CSS Stylesheets can block rendering. Until the browser has requested, received, downloaded and parsed your stylesheets, the page can remain blank. By reducing the amount of CSS the browser has to go through, and by inlining (critical-path styles) it on the page, thus removing a HTTP request, we can get the page to render faster.*

Treebo added support for **Inlining their critical-path CSS** for the current route and asynchronously loading in the rest of their CSS using loadCSS on DOMContentLoaded.

It had the effect of removing the critical-path render blocking link tag for stylesheets and inlining fewer lines of core CSS, improving first paint times to about 0.4s.

```
import assetsManifest from '../../build/client/assetsManifest.json';
//read the styles into an assets object during server startup
export const assets = Object.keys(assetsManifest)
```

```
  .reduce((o, entry) => ({
    ...o,
    [entry]: {
      ...assetsManifest[entry],
      styles: assetsManifest[entry].css
        ? fs.readFileSync(`build/client/css/${assetsManifest[entry].css.split('/').pop()}`, 'ut
    },
  }), {});
export const scripts = {
  //loadCSS by filamentgroup
  loadCSS: 'var loadCSS=function(e,n,t){func...',
  loadRemainingCSS(route) {
    return Object.keys(assetsManifest)
      .filter((entry) => assetsManifest[entry].css && entry !== route.name && entry !== 'main')
      .reduce((s, entry) => `${s}loadCSS("${assetsManifest[entry].css}");`, this.loadCSS);
  },
};
```

fragments.js hosted with ♡ by GitHub view raw

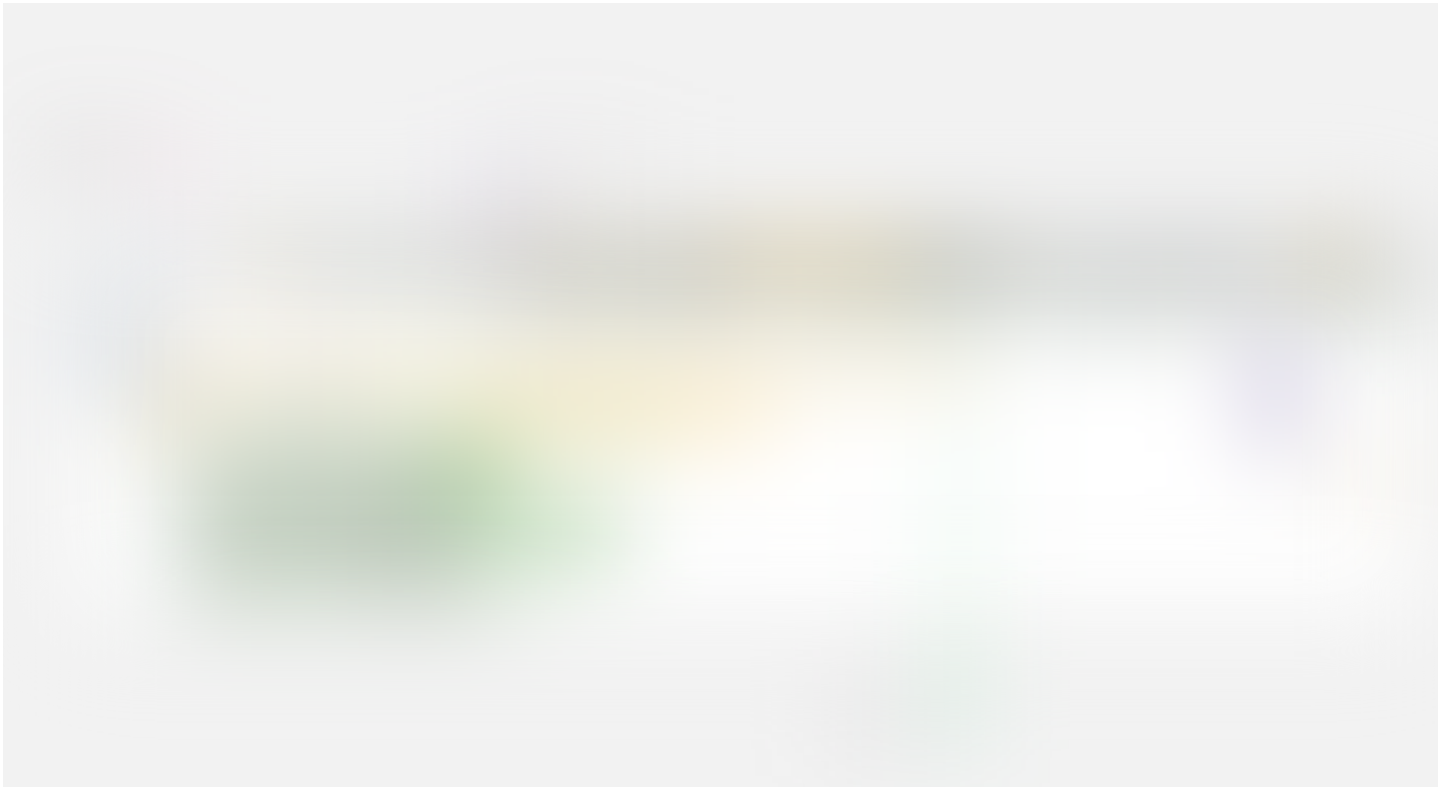
```
//use the assets object to inline styles into your lateChunk template generation logic during r
lateChunk(route) {
  return `
        <style>${assets.main.styles}</style>
        <style>${assets[route.name].styles}</style>
      </head>
      <body>
        <div id="root">${app}</div>
        <script>window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}</script>
        <script src="${assets.webpackManifest.js}"></script>
        <script src="${assets.vendor.js}"></script>
        <script src="${assets.main.js}"></script>
        <script>${scripts.loadRemainingCSS(route)}</script>
      </body>
    </html>
  `;
},
```

html.js hosted with ♡ by GitHub view raw

```
//replace ExtractTextPlugin with ExtractCssChunks from 'extract-css-chunks-webpack-plugin'
module: {
  rules: isProd ? [
    { test: /\.js$/, exclude: /node_modules/, use: ['babel-loader'] },
    { test: /\.css$/, loader: ExtractCssChunks.extract({ use: [{ loader: 'css-loader', options:
    //...

plugins: [
```

```
    new ExtractCssChunks('css/[name].[contenthash:8].css'),
    //this generates a css chunk alongside the js chunk for each dynamic import() call (route-spl
    //main.hash.js, main.hash.css
    //landing.hash.js, landing.hash.css
    //cities.hash.js, cities.hash.css

    //the landing.hash.css and cities.hash.css will contain the css rules for their respective ch
    //but will also contain shared rules between them like button, grid, typography css and so or
    //to extract these shared rules to the main.hash.css use the CommonsChunkPlugin
    //bonus: this also extracts the common js code shared between landing.hash.js and cities.hash
    new webpack.optimize.CommonsChunkPlugin({
      children: true,
      minChunks: 2,
    }),

    //use the assets-webpack-plugin to get a manifest of all the generated files
    new AssetsPlugin({
      filename: 'assetsManifest.json',
      path: path.resolve('./build/client'),
      prettyPrint: true,
    }),
    //...
```

The downside was that time to first interactive went up a bit to 4.6s as the payload size was larger with inline styles and took time to parse before JavaScript could be executed.



Offline-caching static assets

> *A Service Worker is a programmable network proxy, allowing you to control how network requests from your page are handled.*

Treebo added support for **Service Worker** caching of their static assets as well a custom offline page. Below we can see their Service Worker registration and how they used sw-precache-webpack-plugin for resource caching"

```
// register the service worker after the onload event to prevent
// bandwidth resource contention during the main and vendor js downloads
export const scripts = {
  serviceWorker:
    `"serviceWorker" in window.navigator && window.addEventListener("load", function() {
      window.navigator.serviceWorker.register("/serviceWorker.js")
        .then(function(r) {
          console.log("ServiceWorker registration successful with scope: ", r.scope)
        }).catch(function(e) {
          console.error("ServiceWorker registration failed: ", e)
        })
    });`,
};
```

fragments.js hosted with ♡ by GitHub view raw

```
<script src="${assets.webpackManifest.js}"></script>
<script src="${assets.vendor.js}"></script>
<script src="${assets.main.js}"></script>
<script>${scripts.loadRemainingCSS(route)}</script>
//add the serviceWorker script to your html template
<script>${scripts.serviceWorker}</script>
```

html.js hosted with ♡ by GitHub view raw

```
//serve it at the root level scope
app.use('/serviceWorker.js', express.static('build/client/serviceWorker.js'));
```

server.js hosted with ♡ by GitHub view raw

```
new SWPrecacheWebpackPlugin({
  cacheId: 'app-name',
  filename: 'serviceWorker.js',
  staticFileGlobsIgnorePatterns: [/\.map$/, /manifest/i],
  dontCacheBustUrlsMatching: /./,
  minify: true,
}),
```

webpack.js hosted with ♡ by GitHub view raw

Caching static assets like their CSS and JavaScript bundles means pages load up (almost) instantly on repeat visits as they load from the disk cache rather than having to go back out to the network each time. Diligently defined caching headers can have this same effect with respect to disk cache hit-rates, but it's Service Worker that gives us offline support.

Serving JavaScript cached using Service Worker using the Cache API (as we covered in JavaScript Start-up Performance) also has the nice property of early-opting Treebo into V8's code cache so they save a little time on start-up during repeat visits.

Next, Treebo wanted to try getting their vendor bundle-size and JS execution time down, so they switched from React to **Preact** in production.

Switching from React to Preact

Preact is a tiny 3KB alternative to React with the same ES2015 API. It aims to offer high performance rendering with an optional compatibility later (preact-compat) that works with the rest of the React ecosystem, like Redux.

Part of Preact's smaller size comes from removing Synthetic Events and PropType validations. In addition it:

- Diffs Virtual DOM against the DOM
- Allows props like class and for
- Passes (props, state) to render
- Uses standard browser events
- Supports fully async rendering
- Subtree invalidation by default

In a number of PWAs, switching to Preact has led to smaller JS bundle sizes and lower initial JavaScript boot-up times for the application. Recent PWA launches like Lyft, Uber and Housing.com all use Preact in production.

Note: Working with a React codebase and want to use Preact? Ideally, you should use preact and preact-compat for your dev, prod and test builds. This will enable you to discover any interop bugs early on. If you would prefer to only alias preact and preact-compat in Webpack for production builds (e.g if your preference is using Enzyme), make sure to thoroughly test everything works as expected before deploying to your servers.

In Treebo's case, this switch had the impact of dropping their vendor bundle sizes from 140kb all the way down to 100kb. This is all gzipped, by the way. It dropped first

interactive times from **4.6s to 3.9s** on Treebo's target mobile hardware which was a net win.

You can do this in your Webpack config by aliasing react to preact-compat, and react-dom to preact-compat as well.

```
resolve: {
  alias: {
    react: 'preact-compat',
    'react-dom': 'preact-compat',
  },
},
```

webpack.js hosted with ♡ by GitHub view raw

The downside to this approach was that they did have to end up putting together a few workarounds in order to get Preact working exactly with all the different pieces of the React ecosystem that they wanted to use.

Preact tends to be a strong choice for the 95% of cases you would use React; for the other 5% you may end up needing to file bugs to work around edge-cases that are not yet factored in.

Notes: As WebPageTest does not currently offer a way to test real Moto G4s directly from India, performance tests were run under the "Mumbai — EC2 — Chrome — Emulated Motorola G (gen 4) — 3GSlow — Mobile" setting. Should you wish to look at these traces they can be found here.

Skeleton screens

> *"A skeleton screen is essentially a blank version of a page into which information is gradually loaded."*
>
> *~Luke Wroblewski*

Treebo like to implement their skeleton screens using preview enhanced components (a little like skeleton screens for each component). The approach is basically to enhance any atomic component (Text, Image etc) to have a preview version, such that if the source data that is required for the component is not present, it shows the preview version of the component instead.

For example, if you look at the hotel name, city name, price etc in the list items above, they're implemented using Typography components like <Text /> which take two extra props, preview and previewStyle which is used like so.

```
<Text
  preview={!hotel.name}
  previewStyle={{width: 80%}}
>
  {hotel.name}
</Text>
```

Text.js hosted with ♡ by GitHub view raw

Basically, if the hotel.name does not exist then the component changes the background to a greyish color with the width and other styles set according to the previewStyle passed down (width defaults to 100% if no previewStyle is passed).

```
.text {
  font-size: 1.2rem;
  color: var(--color-secondary);

  &--preview {
    opacity: 0.1;
    height: 13px;
    width: 100%;
    background: var(--color-secondary);
  }

  @media (--medium-screen) {
    font-size: 1.4rem;

    &--preview {
      height: 16px;
    }
  }
}
```

text.css hosted with ♡ by GitHub view raw

```
import React, { PropTypes } from 'react';
import cn from 'classnames';

const Text = ({
  className,
  tag,
  preview,
  previewStyle,
  children,
  ...props
}) =>
  React.createElement(tag, {
    style: preview ? previewStyle : {}
```

```
      style: preview ? previewStyle : {},
      className: cn('text', {
        'text--preview': preview,
      }, className),
      ...props,
    }, children);

Text.propTypes = {
  className: PropTypes.string,
  tag: PropTypes.string.isRequired,
  preview: PropTypes.bool.isRequired,
  previewStyle: PropTypes.object,
  children: PropTypes.node,
};

Text.defaultProps = {
  tag: 'p',
  preview: false,
};

export default Text;
```

To tie hosted with ♡ by GitHub

Treebo likes this approach because the logic to switch to the preview mode is independent of the data actually being shown which makes it flexible. If you look at the “Incl. of all taxes” part, it’s just static text, which could have been shown right at the start but that would’ve looked very confusing to the user since the prices are still loading during the api call.

So to get the static text “Incl. of all prices” into a preview mode alongside the rest of the ui they just use the price itself as the logic for the preview mode.

```
<Text preview={!price.sellingPrice}>
  Incl. of all taxes
</Text>
```

TextPreview.js hosted with ♡ by GitHub view raw

This way while the prices are loading you get a preview UI and once the api succeeds you get to see the data in all its glory.

Webpack-bundle-analyzer

At this point, Treebo wanted to perform some bundle analysis to look at what other low-hanging fruit they could optimize.

Note: If you're using a library like React on mobile, it's important to be diligent about the other vendor libraries you are pulling in. Not doing so can negatively impact performance. Consider better chunking your vendor libraries so that routes only load those that are needed

Treebo used webpack-bundle-analyzer to keep track of their bundle size changes and to monitor what modules are contained in each route chunk. They also use it to find areas where they can optimize to reduce bundle sizes such as stripping moment.js' locales and reusing deep dependencies.

Optimizing moment.js with webpack

Treebo relies heavily on moment.js for their date manipulations. When you import moment.js and bundle it with Webpack, your bundle will include all of moment.js and it's locales by default which is ~61.95kb gzipped. This seriously bloats your final vendor bundle size.

To optimize the size of moment.js, there are two webpack plugins available: IgnorePlugin, ContextReplacementPlugin

Treebo opted to remove all locale files with the IgnorePlugin since they didn't need any of the them.

new webpack.IgnorePlugin(/^\.\/locale$/, /moment$/)

With the locales stripped out, the moment.js' bundled size dropped to ~16.48kb gzipped.

The biggest improvement as a side effect of stripping out moment.js' locales was that the vendor bundle size dropped from ~179kb to ~119kb. That's a massive 60kb drop from a critical bundle that has to be served on first load. All this translates to a considerable decrease in first interaction times. You can read more about optimizing moment.js here.

Reusing existing deep dependencies

Treebo was initially using the "qs" module to perform query string operations. Using the webpack-bundle-analyzer output they found that "react-router" included the "history" module which in-turn included the "query-string" module.

Since there were two different modules both accomplishing the same operations, replacing “qs” with this version of “query-string” (by installing it explicitly) in their source code, dropped their bundle size by a further 2.72kb gzipped (size of the “qs” module).

Treebo have been good open source citizens. They’ve been using a lot of open source software. In return, they’ve actually open sourced most of their Webpack configuration, as well as a boilerplate that contains a lot of the set up they’re using in production. You can find that here: https://github.com/lakshyaranganath/pwa

They’ve also committed to trying to keep that up to date. As they evolve you can take advantage of them as another PWA reference implementation.

Conclusions and the future

Treebo knows that no application is perfect, they actively explore many methods to keep improving the experience they deliver to their users. Some of which are:

Lazy Loading Images

Some of you might have figured out from the network waterfall graphs before that the website image downloads are competing for bandwidth with the JS downloads.

Since image downloads are triggered as soon as the browser parses the img tags, they share the bandwidth during JS downloads. A simple solution would be lazy loading images only when they come into the user's viewport, this will see a good improvement in our time to interactive.

Lighthouse highlights these problems well in the offscreen images audit:

Dual Importing

Treebo also realise that while they are asynchronously loading the rest of the CSS for the app (after inlining the critical css), this approach is not viable for their users in the long run as their app grows. More features and routes means more CSS, and downloading all of that leads to bandwidth hogging and wastage.

Merging approaches followed by loadCSS and babel-plugin-dual-import, Treebo changed their approach to loading CSS by using an explicit call to a custom

implemented importCss('chunkname') to download the CSS chunk in parallel to their import('chunkpath') call for their respective JS chunk.

```
import assetsManifest from '../../build/client/assetsManifest.json';

lateChunk(app, head, initialState, route) {
    return `
          <style>${assets.main.styles}</style>
          // inline the current route's css and assign an id to it
          ${!assets[route.name] ? '' : `<style id="${route.name}.css">${assets[route.name].sty
        </head>
        <body>
          <div id="root">${app}</div>
          <script>window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}</script>
          <script>window.__ASSETS_MANIFEST__ = ${JSON.stringify(assetsManifest)}</script>
          <script src="${assets.webpackManifest.js}"></script>
          <script src="${assets.vendor.js}"></script>
          <script src="${assets.main.js}"></script>
        </body>
      </html>`;
  },
```

html.js hosted with ♡ by GitHub view raw

```
export default (chunkName) => {
  if (!__BROWSER__) {
    return Promise.resolve();
  } else if (!(chunkName in window.__ASSETS_MANIFEST__)) {
    return Promise.reject(`chunk not found: ${chunkName}`);
  } else if (!window.__ASSETS_MANIFEST__[chunkName].css) {
    return Promise.resolve(`chunk css does not exist: ${chunkName}`);
  } else if (document.getElementById(`${chunkName}.css`)) {
    return Promise.resolve(`css chunk already loaded: ${chunkName}`);
  }

  const head = document.getElementsByTagName('head')[0];
  const link = document.createElement('link');
  link.href = window.__ASSETS_MANIFEST__[chunkName].css;
  link.id = `${chunkName}.css`;
  link.rel = 'stylesheet';

  return new Promise((resolve, reject) => {
    let timeout;

    link.onload = () => {
      link.onload = null;
      link.onerror = null;
```

```
      clearTimeout(timeout);
      resolve(`css chunk loaded: ${chunkName}`);
    };

    link.onerror = () => {
      link.onload = null;
      link.onerror = null;
      clearTimeout(timeout);
      reject(new Error(`could not load css chunk: ${chunkName}`));
    };

    timeout = setTimeout(link.onerror, 30000);
    head.appendChild(link);
  });
};
```

importCss.js hosted with ♡ by GitHub view raw

```
<IndexRoute
  name="landing"
  getComponent={(_, cb) => {
    Promise.all([
      import('./views/LandingPage/LandingPage' /* webpackChunkName: 'landing' */),
      importCss('landing'),
    ]).then(([module]) => cb(null, module.default));
  }}
/>

<Route
  name="search"
  path="/search/"
  getComponent={(_, cb) => {
    Promise.all([
      import('./views/SearchResultsPage/SearchResultsPage' /* webpackChunkName: 'search' */),
      importCss('search'),
    ]).then(([module]) => cb(null, module.default));
  }}
/>
```

With this new approach, a route transition results in two parallel asynchronous requests, one for JS and the other for CSS unlike the previous approach where all of the CSS was being downloaded on DOMContentLoaded. This is more viable since a user will only ever download the required CSS for the routes they are visiting.

A/B Testing

Treebo are currently implementing an AB testing approach with server side rendering

and code splitting so as to only push down the variant that user needs during both server and client side rendering. (Treebo will follow up with a blog post on how they tackled this).

Eager Loading

Treebo ideally don't want to always load all the split chunks of the app on load of the initial page since they want to avoid the bandwidth contention for critical resource downloads — this also wastes precious bandwidth for mobile users especially if you're not caching it with service-worker for their future visits. If we look at how well Treebo is doing on metrics like consistently interactive, there's still much room for improvement:

This is an area they're experimenting with improving. One example is eager loading the next route's chunk during the ripple animation of a button. onClick Treebo make a webpack dynamic import() call to the next route's chunk entry and delay the route transition with a setTimeout. They also want to make sure that the next route's chunk is small enough to be downloaded within the given 400ms timeout on a slow 3g network.

That's a wrap.

It's been fun collaborating on this write-up. There's obviously more work to be done, but we hope you found Treebo's performance journey an interesting read :) You can find us over on twitter at @addyosmani and @__lakshya (yep, double underscore xD) we would love to hear your thoughts.

With thanks to @_zouhir, @_developit and @samcccone for their reviews and input.

If you're new to React, React for Beginners by Wes Bos is a comprehensive overview for getting started.

Thanks to Jason Miller and Lakshya Ranganath.

JavaScript React Reactjs Progressive Web App Pwa