Build your first progressive web app with React

Scott Domes
Apr 25, 2017 · 8 min read

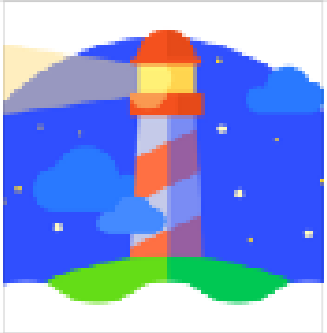
Progressive Web Apps are the much-hyped future of the web. Let's build one!

Google has been pushing PWA's hard as the solution to many of the problems of the modern web — particularly issues for mobile users.

Web | Google Developers

A service worker, written in JavaScript, is like a client-side proxy and puts you in control of the...

developers.google.com



PWA are essentially fast, performance-focused web applications that are streamlined for mobile. They also can be saved to your smartphone's home screen and, from there, look and feel like a native app (including features like offline access and push notifications).

Big players like Twitter and Flipboard have recently launched PWA's, which you can try out by going to either http://flipboard.com or https://lite.twitter.com/ on your phone.

In this tutorial, you'll build a simple PWA using React — giving you a boilerplate from which to construct more complex applications.

Getting Set Up

To start, let's generate a basic React application with create-react-app.

To do so, switch to the directory in which you want to save your app, and run the following:

```
npm install -g create-react-app
create-react-app pwa-experiment
```

Then, let's install React Router:

```
cd pwa-experiment
npm install --save react-router@3.0.5
```

Finally, copy this gist into your App.js. This will give us a simple layout with navigation:

```
import React, { Component } from 'react';
import { Router, browserHistory, Route, Link } from 'react-router';
import logo from './logo.svg';
import './App.css';

const Page = ({ title }) => (
    <div className="App">
      <div className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <h2>{title}</h2>
      </div>
      <p className="App-intro">
        This is the {title} page.
      </p>
      <p>
        <Link to="/">Home</Link>
      </p>
      <p>
        <Link to="/about">About</Link>
      </p>
      <p>
        <Link to="/settings">Settings</Link>
      </p>
    </div>
);

const Home = (props) => (
  <Page title="Home"/>
);
```

```
const About = (props) => (
  <Page title="About"/>
);

const Settings = (props) => (
  <Page title="Settings"/>
);

class App extends Component {
  render() {
    return (
      <Router history={browserHistory}>
        <Route path="/" component={Home}/>
        <Route path="/about" component={About}/>
        <Route path="/settings" component={Settings}/>
      </Router>
    );
  }
}

export default App;
```

pwa-starter is hosted with ♡ by GitHub view raw

Run *npm start* to test our your app. Not much to look at, but it'll serve our purposes just fine. Let's get started converting this into a PWA.

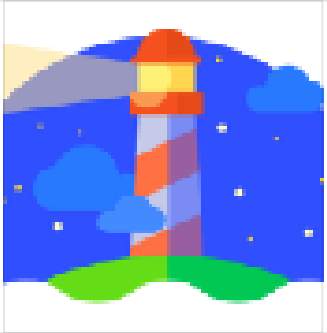
Step 1: Install Lighthouse

Lighthouse is a free tool from Google that evaluates your app based on their PWA checklist.

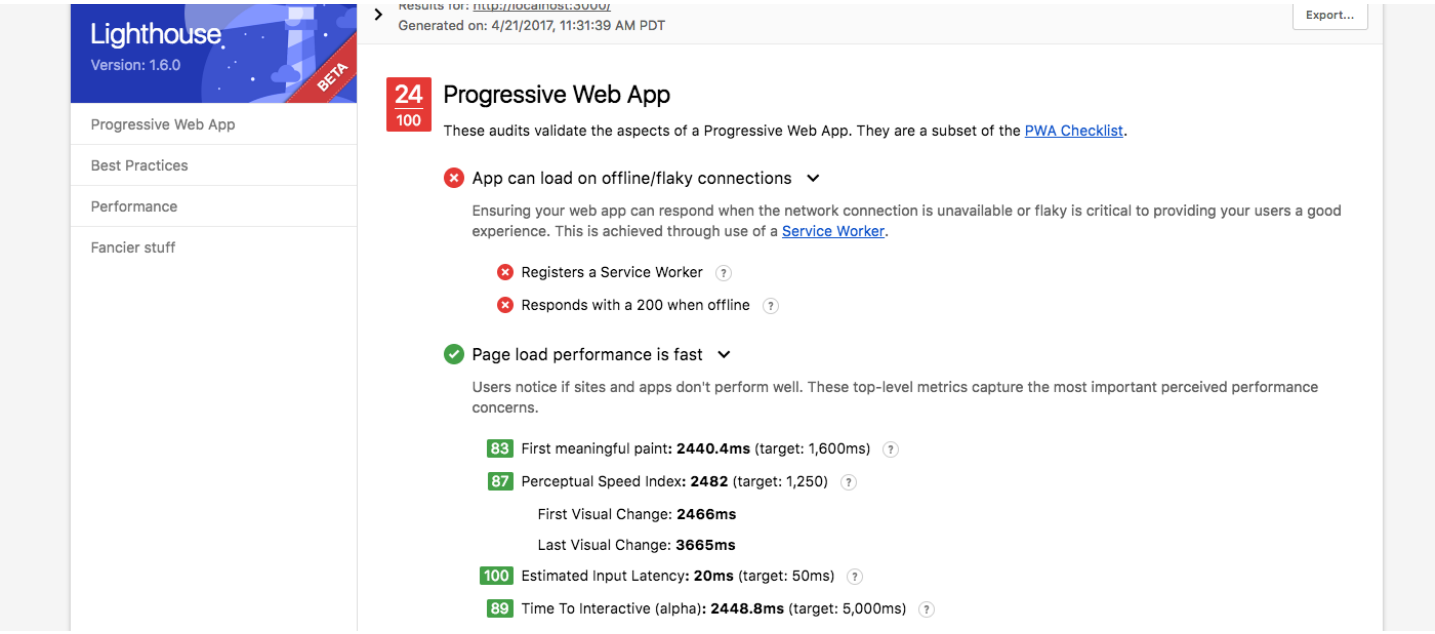
Lighthouse | Web | Google Developers

You can run Lighthouse as a Chrome Extension, from the command line, or as a Node module. Yo...

developers.google.com



Let's install it to Chrome, and then evaluate our app. You can start a Lighthouse audit by clicking the Lightouse icon in the top right corner of Chrome, and then clicking Generate Report.



Ouch.

So far, our app is fast (since we have very little content) but fails in a number of key areas.

You can view the checklist that Lighthouse uses here:

Progressive Web App Checklist | Web | Google Developers

To help teams create the best possible experiences we've put together this checklist which breaks down all the things...

developers.google.com

Let's work through the problems, one by one.

Step 2: Set Up A Service Worker

A service worker is a bit of JavaScript that sits between our application and the network. We're going to use it to intercept network requests and serve up cached files — this will allow our app to work offline.

To get started with a service worker, we need to do three things:

- Create a service-worker.js file in our public folder
- Register the worker via our index.html
- Set up caching

Let's get to it.

First step is pretty self-explanatory. In pwa-experiment/public, create a blank JavaScript file named service-worker.js.

Second step is a bit more involved. We want to check if the browser supports service workers, and then register one by loading in service-worker.js.

To do so, let's add a script tag to our public/index.html.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <script>
      if ('serviceWorker' in navigator) {
        window.addEventListener('load', function() {
          navigator.serviceWorker.register('service-
worker.js').then(function(registration) {
            // Registration was successful
            console.log('ServiceWorker registration successful with
scope: ', registration.scope);
          }, function(err) {
            // registration failed :(
            console.log('ServiceWorker registration failed: ', err);
          }).catch(function(err) {
            console.log(err)
          });
        });
      } else {
        console.log('service worker is not supported');
      }
    </script>
  </body>
</html>
```

The code is straightforward- if the navigator supports it, we wait for the page to load and then register our worker by loading in the service-worker.js file.

Last step: set up caching!

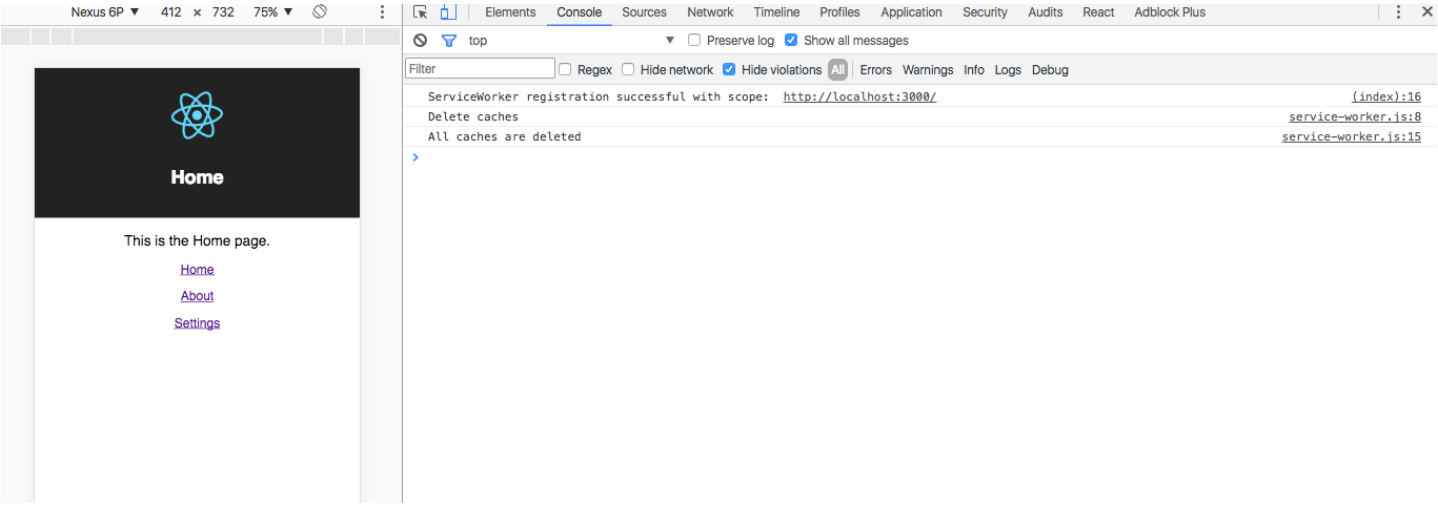
We're going to copy Addy Osmani's service worker configuration from here, but also disable the cache for development purposes (and take the precautionary step of deleting all caches when the service worker initializes).

In public/service-worker.js:

```
// Set this to true for production
var doCache = false;

// Name our cache
var CACHE_NAME = 'my-pwa-cache-v1';

// Delete old caches that are not our current one!
self.addEventListener("activate", event => {
  const cacheWhitelist = [CACHE_NAME];
  event.waitUntil(
    caches.keys()
      .then(keyList =>
        Promise.all(keyList.map(key => {
          if (!cacheWhitelist.includes(key)) {
            console.log('Deleting cache: ' + key)
            return caches.delete(key);
          }
        }))
      )
  );
});

// The first time the user starts up the PWA, 'install' is triggered.
self.addEventListener('install', function(event) {
  if (doCache) {
    event.waitUntil(
      caches.open(CACHE_NAME)
        .then(function(cache) {
          // Get the assets manifest so we can see what our js file is named
          // This is because webpack hashes it
          fetch("asset-manifest.json")
            .then(response => {
              response.json()
            })
            .then(assets => {
              // Open a cache and cache our files
              // We want to cache the page and the main.js generated by webpack
              // We could also cache any static assets like CSS or images
              const urlsToCache = [
                "/",
```

```
                  assets["main.js"]
                ]
                cache.addAll(urlsToCache)
                console.log('cached');
              })
          })
      );
    }
});

// When the webpage goes to fetch files, we intercept that request and serve up the matching fi
// if we have them
self.addEventListener('fetch', function(event) {
    if (doCache) {
      event.respondWith(
          caches.match(event.request).then(function(response) {
              return response || fetch(event.request);
          })
      );
    }
});
```

Restart your app with *npm run start* and you should see the following in the console:

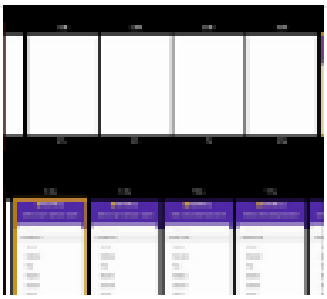


For a deeper discussion into service workers and scope, see the below:

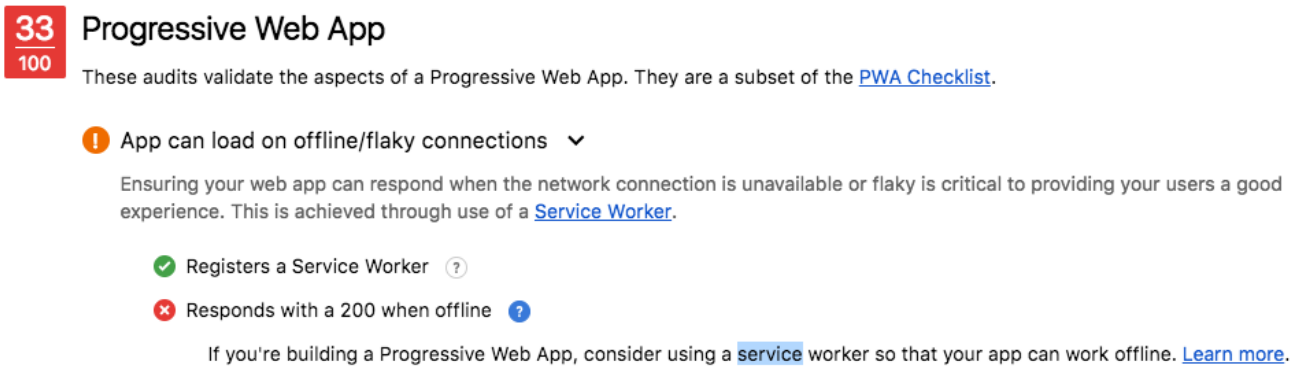
Progressive Web Apps with React.js: Part 3 — Offline support and network resilience

Part 3 of a new series walking through tips for shipping mobile web apps optimized using...

medium.com



Let's close DevTools and run the Lighthouse audit again:



We're making progress! We now have a registered service worker. Since we disabled the caching, we haven't ticked the second box yet, but once we enable caches (when we go live) it will work!

Step 3: Add Progressive Enhancement

Progressive enhancement basically means your site will work without any JavaScript loading.

Right now, our index.html just renders an empty div (#root), which our React app then hooks into.

We want to instead display some basic HTML and CSS before the React app initializes.

The easiest way to do so is to move some of our basic HTML structure *within* that div#root. This HTML will be overwritten as soon as ReactDOM renders our App component, but will give the user something other than a blank page to stare at as the bundle.js loads.

Here's our new index.html. Note both the styles in the head, and the HTML within div#root.

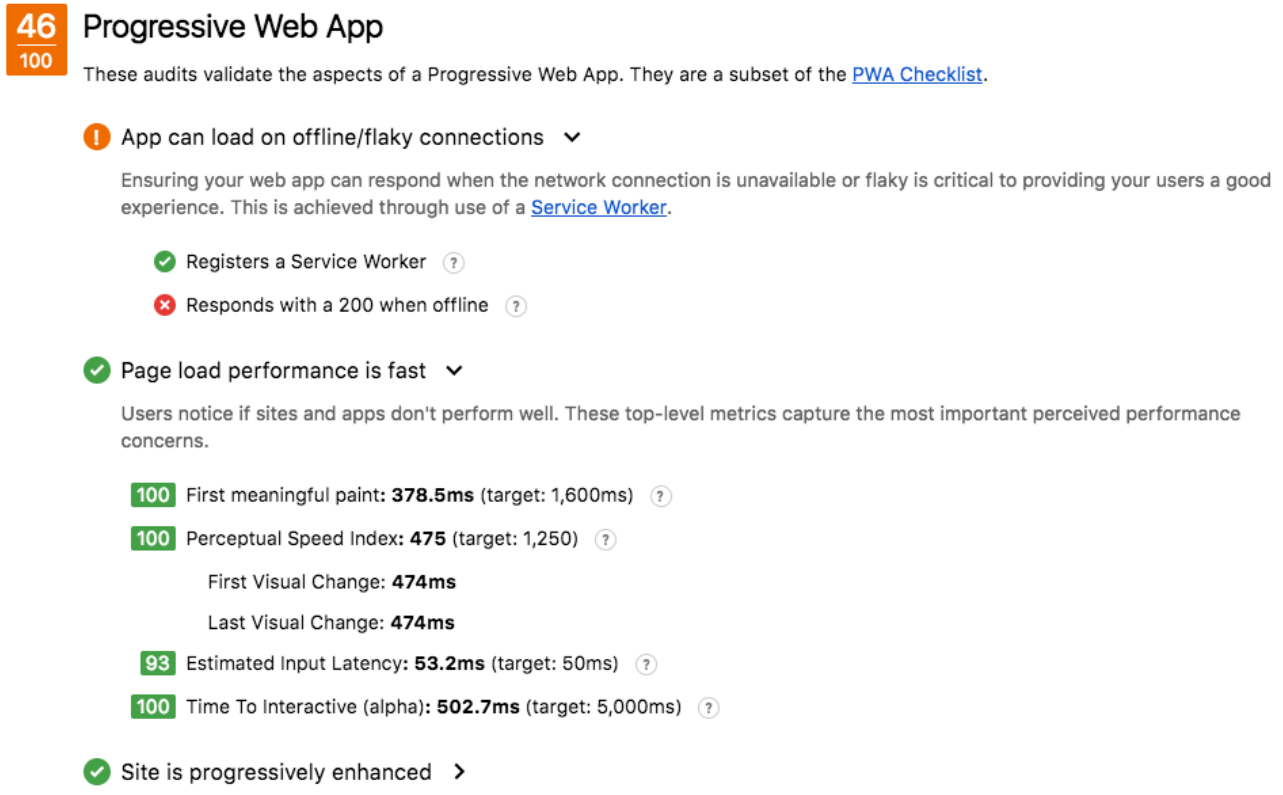
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
```

```
    <title>React App</title>
    <!-- Add in some basic styles for our HTML -->
    <style type="text/css">
      body {
        margin: 0;
        padding: 0;
        font-family: sans-serif;
      }

      .App {
        text-align: center;
      }

      .App-header {
        background-color: #222;
        height: 150px;
        padding: 20px;
        color: white;
      }

      .App-intro {
        font-size: large;
      }
    </style>
  </head>
  <body>
    <!-- Filler HTML as our app starts up -->
    <div id="root">
      <div class="App">
      <div class="App-header">
        <h2>Home</h2>
      </div>
      <p class="App-intro">
        Loading site...
      </p>
    </div>
    <script>
      if ('serviceWorker' in navigator) {
        window.addEventListener('load', function() {
          navigator.serviceWorker.register('service-worker.js').then(function(registration) {
            // Registration was successful
            console.log('ServiceWorker registration successful with scope: ', registration.scop
          }, function(err) {
            // registration failed :(
            console.log('ServiceWorker registration failed: ', err);
          }).catch(function(err) {
            console.log(err)
          })
```

```
54            });
55          });
56        } else {
57          console.log('service worker is not supported');
58        }
59      </script>
60    </body>
61  </html>
```

(As an aside, we can now delete duplicate styles in App.css and index.css — just to be clean.)

Does Lighthouse approve?



Yep!

Step 4: Add To Home Screen Capability

We can skip the requirements about https — that will be taken care of once we deploy.

Now, onto the feature that makes PWA's particularly exciting: the ability for the user to save them to their home screen, and then open them like an app.

To do, we need to add a manifest.json file to our public directory (copy the uncommented version below this one).

```
{
  // Short name is what appears on home screen
  "short_name": "My First PWA",
  // Name is what appears on splash screen
  "name": "My First Progressive Web App",
  // What appears on splash screen & home screen
  "icons": [
    {
      "src":"icon.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ],
  // So your site can tell it was opened from home screen
  "start_url": "/?utm_source=homescreen",
  // Match our app header background
  "background_color": "#222",
  // What the URL bar will look like
  "theme_color": "#222",
  // How the app will appear when it launches (see link below)
  "display": "standalone"
  // Read more: https://developer.mozilla.org/en-US/docs/Web/Manifest
}
```

Here's the file without comments:

```
{
  "short_name": "My First PWA",
  "name": "My First Progressive Web App",
  "icons": [
    {
      "src":"icon.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ],
  "start_url": "/?utm_source=homescreen",
  "background_color": "#222",
  "theme_color": "#222",
  "display": "standalone"
}
```

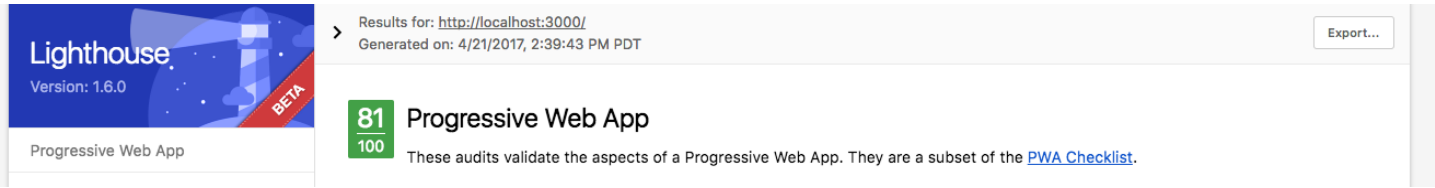
manifest.json hosted with ♡ by GitHub view raw

Here's the icon if you need one (courtesy of my company, MuseFind), or create your own (must be 192 by 192 pixels):

Add the icon.png and manifest.json into your public folder, and then add the following lines to your index.html:

```
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <!-- Add manifest -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <!-- Tell the browser it's a PWA -->
    <meta name="mobile-web-app-capable" content="yes">
    <!-- Tell iOS it's a PWA -->
    <meta name="apple-mobile-web-app-capable" content="yes">
    <!-- Make sure theme-color is defined -->
    <meta name="theme-color" content="#536878">
    <title>React App</title>
  </head>
```

Alright, *now* how are we doing?



All we're missing is caching and https. Let's push this live!

Step 5: Deploy Via Firebase

First, let's turn on caching. Change *doCache* to **true** in your service-worker.js.

Then, in the Firebase console, create a new project called pwa-experiment.

Back in your project folder, run the following:

```
npm install -g firebase-tools
firebase login
firebase init
```

After you complete the login and start the initiation, answer the following questions:

When it asks **What Firebase CLI features do you want to setup for this directory?**, use the spacebar to deselect all but Hosting.

Hit enter, then select pwa-experiment as the project.

When it asks **What do you want to use as your public directory?**, type *build* and then hit enter.

For the single page app question, say no.

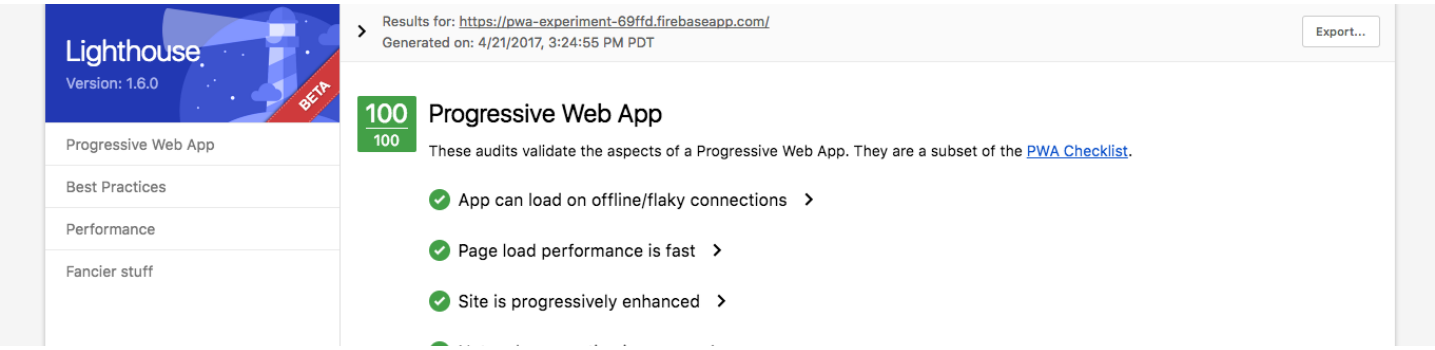
That should complete. Then, you can run the following:

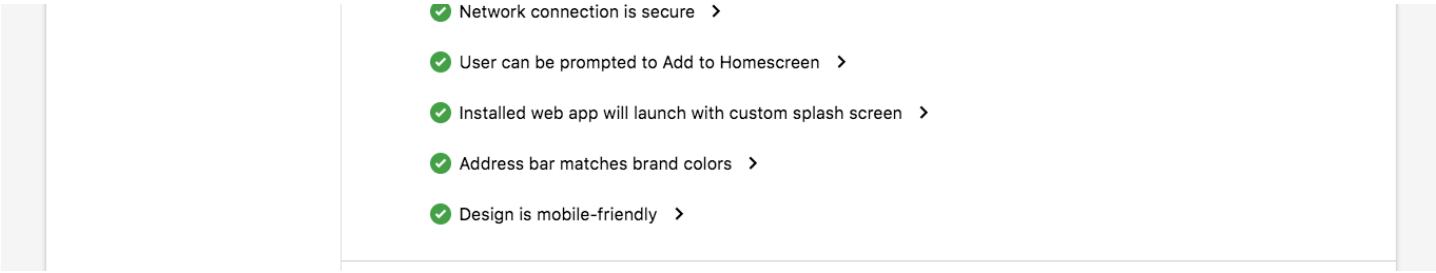
```
npm run build && firebase deploy
```

This tells create-react-app to build our project into the build/ folder, which Firebase then deploys.

Firebase will give you back a URL. Let's open that in Chrome, and then run our Lighthouse audit one last time.

This time, we'll be using HTTPS instead of HTTP, and take advantage of caching.





We did it!

As a last test, open it up on your phone and try saving it to your home screen. Once opened from the home screen, it should feel like a native app.

Where To Go From Here

The essence of a PWA is speed. In this tutorial, we skipped a lot of the performance enhancement, since our app was so barebones.

As your app grows, however, our main.js file is going to grow and grow, and Lighthouse will be less and less pleased with us.

Stay tuned (AKA follow me: Scott Domes or my Twitter) for an in-depth article on optimizing performance with React and React Router that will work for both PWA and old-fashioned web apps.

For now, we have a working skeleton of PWA to build on — we're ready for the future of web apps.

Done!

If this article has been helpful, recommend it by hitting the green heart or (even better) share it.

Want to stay up to date/learn best practices for the future of web development? Say hello to Progressive Web App Newsletter — subscribe below.

Progressive Web App Newsletter

A free, bi-weekly newsletter on all things PWA-news, tutorials, interesting projects. Three links an issue, every two...

pwa-newsletter.com

Here's the final repo:

scottdomes/pwa-experiment

Contribute to pwa-experiment development by creating an account on GitHub.

github.co

Thanks to Stephen Shen.

Web Development JavaScript Progressive Web App React ES6