A Simple Progressive Web App Tutorial

Create a Hello World PWA with HTML and Javascript

James Johnson · Follow
Published in Design Notes
8 min read · Aug 27, 2018

After searching the web for a simple PWA tutorial, everything I found was either too complicated or required one 3rd party library/framework/platform or another. Personally, when learning a new technology, I'd rather not get sidetracked with unnecessary details. So drawing from a number of sources I wrote a simple tutorial myself that doesn't require any 3rd party content: The classic "Hello World" app, PWA style.

Hello World as a Progressive Web App

If you're not familiar with Progressive Web Apps the basic idea is to use browser

technologies to build a web application that works offline and has the look and feel of a native application. In this tutorial I'll show you how to use a manifest and service workers to create just about the simplest app possible, one that works without an internet connection, and can be added to your home screen.

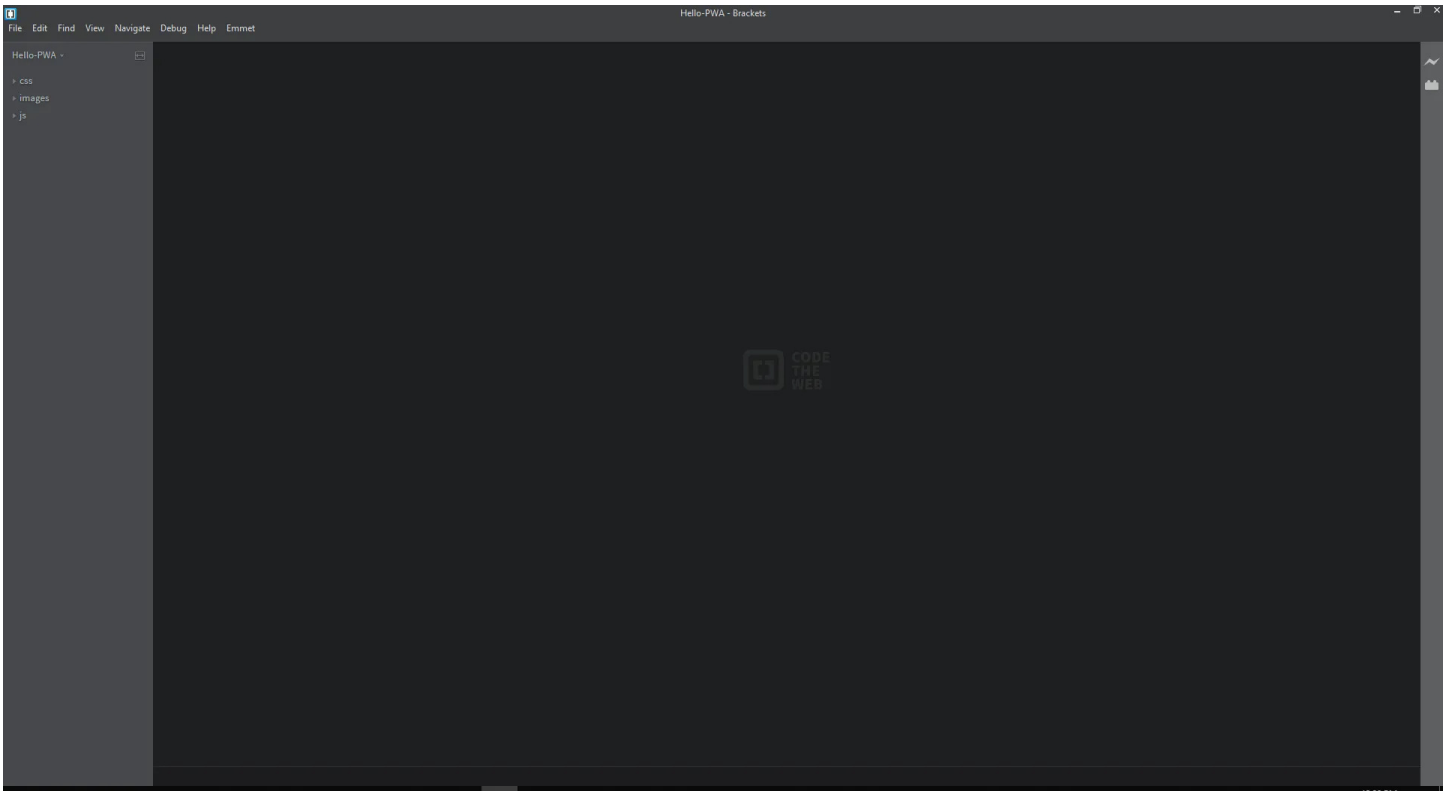
To get the most out of this tutorial you should be familiar with HTML, CSS and JavaScript. If you can code a web page and use plain-vanilla JavaScript to add some interactivity you should be able to follow along. To build this app you'll need a text editor, the latest version of Chrome and a local web server. I've used Adobe's Brackets in this tutorial, because it has a built in web server, but you can use any text editor and server combo you're comfortable with.

The Setup

Create a directory for the app and add *js*, *css*, and *images* subdirectories. It should look like this when you're finished:

```
/Hello-PWA   # Project Folder
    /css     # Stylesheets
    /js      # Javascript
    /images  # Image files.
```

Open your project folder in Brackets to get started.



Brackets

Writing the App Interface

When writing the markup for a Progressive Web App there are 2 requirements to keep in mind:

1. The app should display some content even if JavaScript is disabled. This prevents users from seeing a blank page if their internet connection is bad or if they are using an older browser.
2. It should be responsive and display properly on a variety of devices. In other words, it needs to be mobile friendly.

For our little app, we'll tackle the first requirement by simply hard coding the content and the second by adding a `viewport` meta tag.

To do this, create a file named *index.html* in your project root folder and add the following markup:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
  <link rel="stylesheet" href="css/style.css">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
<body class="fullscreen">
  <div class="container">
    <h1 class="title">Hello World!</h1>
  </div>
</body>
</html>
```

index.html hosted with ❤️ by **GitHub** **view raw**

index.html

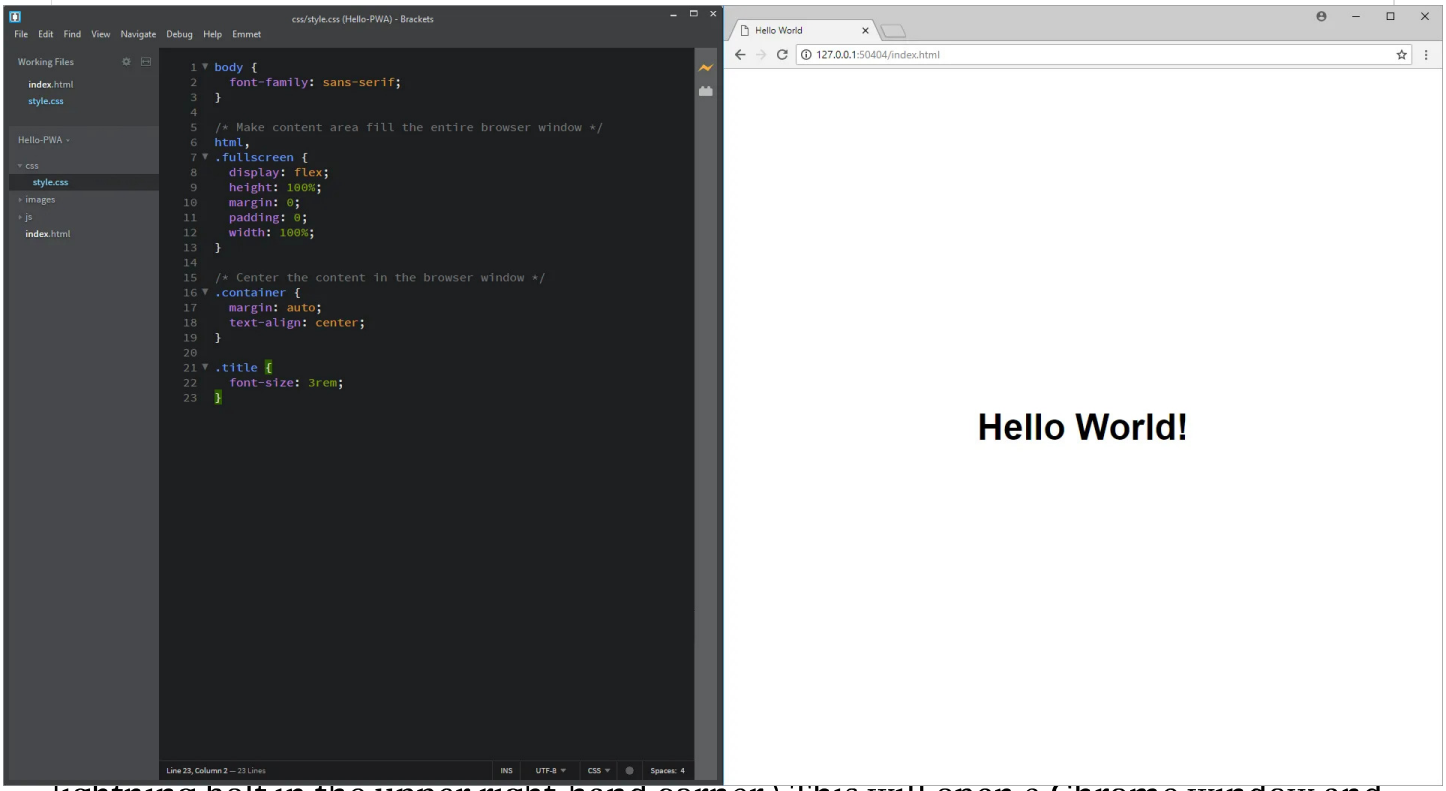
All this markup does is load in a stylesheet and set the viewport width & scale to their defaults. The “hello” text is hard coded into the body’s `h1` element which is wrapped in a container div to make styling easier.

Next, create a file named *style.css* in the *css* folder and add this code:

```css
body {
  font-family: sans-serif;
}

/* Make content area fill the entire browser window */
html,
.fullscreen {
  display: flex;
  height: 100%;
  margin: 0;
  padding: 0;
  width: 100%;
}

/* Center the content in the browser window */
.container {
  margin: auto;
  text-align: center;
}
```

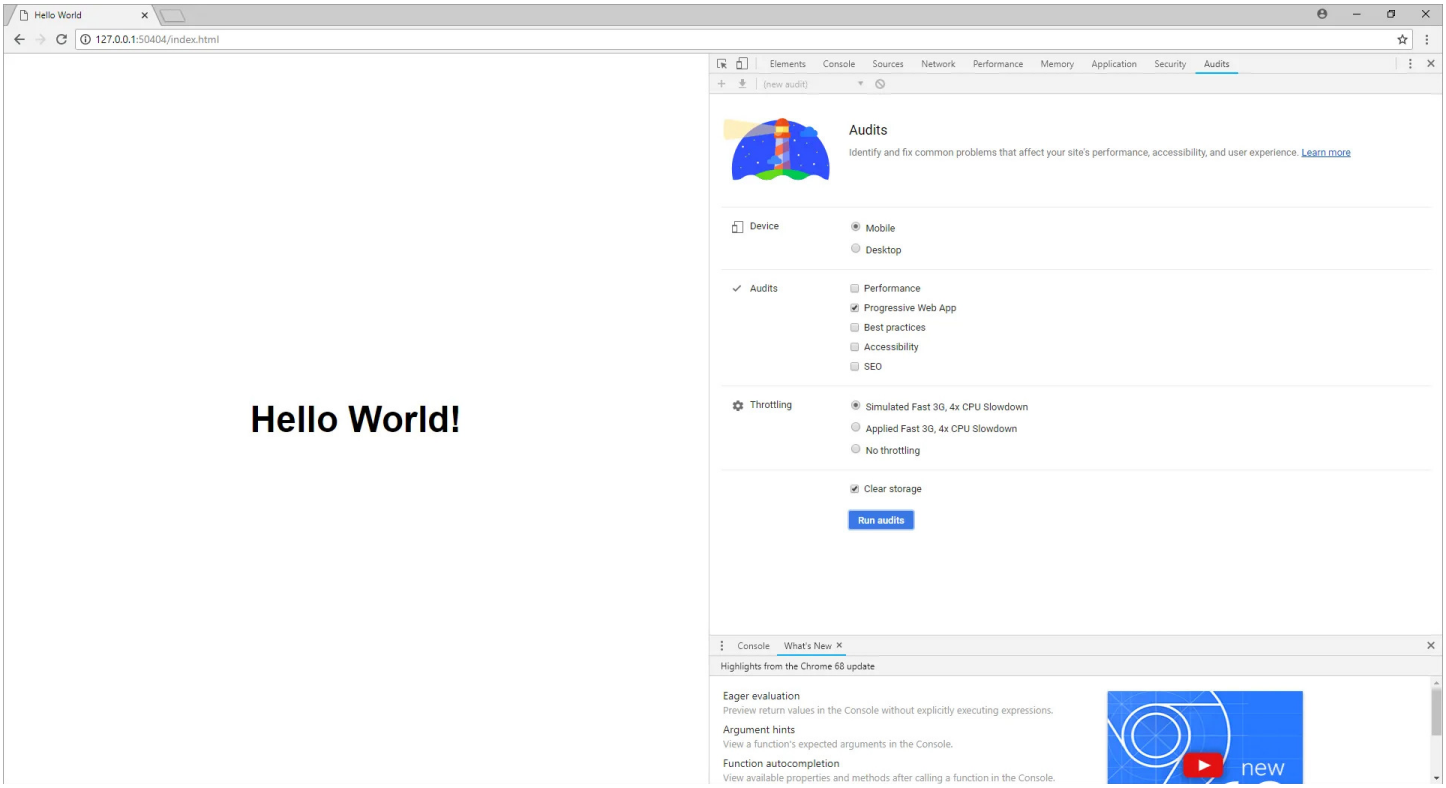


Previewing the app in Chrome

lightning bolt in the upper right-hand corner.) This will open a Chrome window and serve up your page.

Testing the App

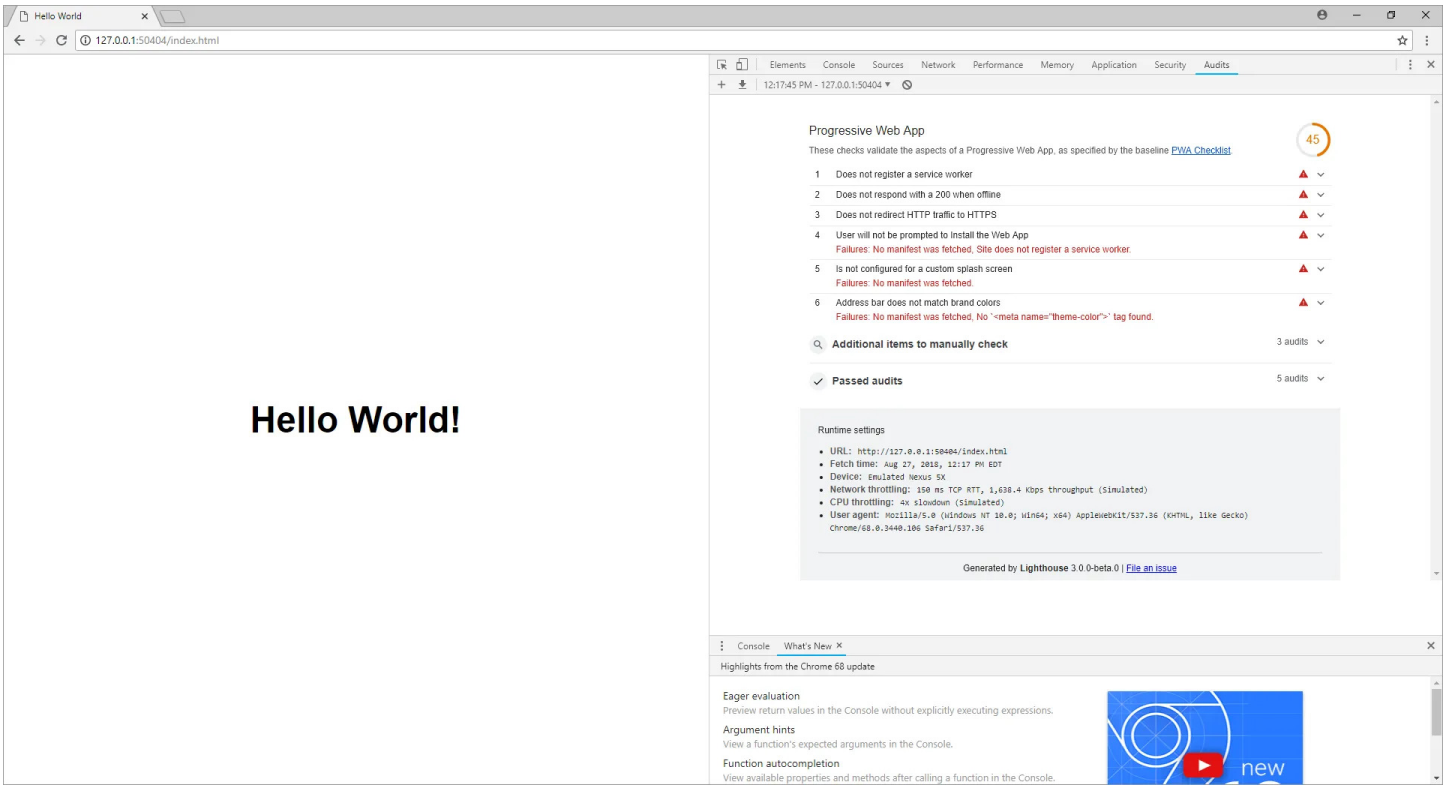
Now that we've got something in the browser, we'll use Google's Lighthouse to test the app and see how well it conforms to PWA standards. Press F12 to open the developer panel in Chrome and click on the audits tab to open Lighthouse.



Google Lighthouse

Make sure the "Progressive Web App" option is checked. You can uncheck the others for now. Then click on the "run tests" button. After a minute or two Lighthouse should give you a score and list of audits that the app passed or failed. The app should score around 45 at this point. If everything was coded properly, you'll notice that most of the tests it passes are related to the requirements we outlined at the beginning:

1. Contains some content when JavaScript is not available.
4. Has a `<meta name="viewport>` tag with width or initial-scale.
5. Content is sized correctly for the viewport.



Audit results

As you build your app you can periodically run these audits to make sure development is on track and detect any problems that need to be fixed.

Add a Service Worker

The next requirement for our app is to register a service worker. Service workers are essentially scripts that run in the background to perform tasks that don't require user interaction. This frees up the main app for your users while the service worker takes care of the boring stuff.

For our app we'll use one to download and cache our content and then serve it back up from the cache when the user is offline.

Create a file named *sw.js* in your root folder and enter the contents of the script below. The reason it's saved in the app root is to give it access to all of the app's files. This is becauses service workers only have permission to access files in the same directory and sub-directories.

```
var cacheName = 'hello-pwa';
var filesToCache = [
  '/',
  '/index.html',
  '/css/style.css',
  '/js/main.js'
];

/* Start the service worker and cache all of the app's content */
self.addEventListener('install', function(e) {
  e.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(filesToCache);
    })
  );
});

/* Serve cached content when offline */
self.addEventListener('fetch', function(e) {
  e.respondWith(
    caches.match(e.request).then(function(response) {
      return response || fetch(e.request);
    })
  );
});
```

sw.js hosted with ❤️ by **GitHub** **view raw**

sw.js

The first lines of the script declares two variables: `cacheName` and `filesToCache`. `cacheName` is used to create an offline cache in the browser and give us access to it from Javascript. `filesToCache` is an array containing a list of all of the files that need to be cached. These files should be written in the form of URLs. Notice that the first one is simply "/", the base URL. This is so the browser caches *index.html* even if the user doesn't directly type in that file name.

Next, we add a function to install the service worker and create the browser cache using `cacheName`. Once the cache is created it adds all of the files listed in the `filesToCache` array. (Please note that while this code works for demonstration

purposes it is not intended for production as it will stop if it fails to load even one of the files.)

Finally, we add a function to load in the cached files when the browser is offline.

Now that the service worker script is created we need to register it with our app. Create a file named *main.js* in the *js* folder and enter the following code:

```
window.onload = () => {
  'use strict';

  if ('serviceWorker' in navigator) {
    navigator.serviceWorker
             .register('./sw.js');
  }
}
```

main.js hosted with ❤ by **GitHub** **view raw**

main.js

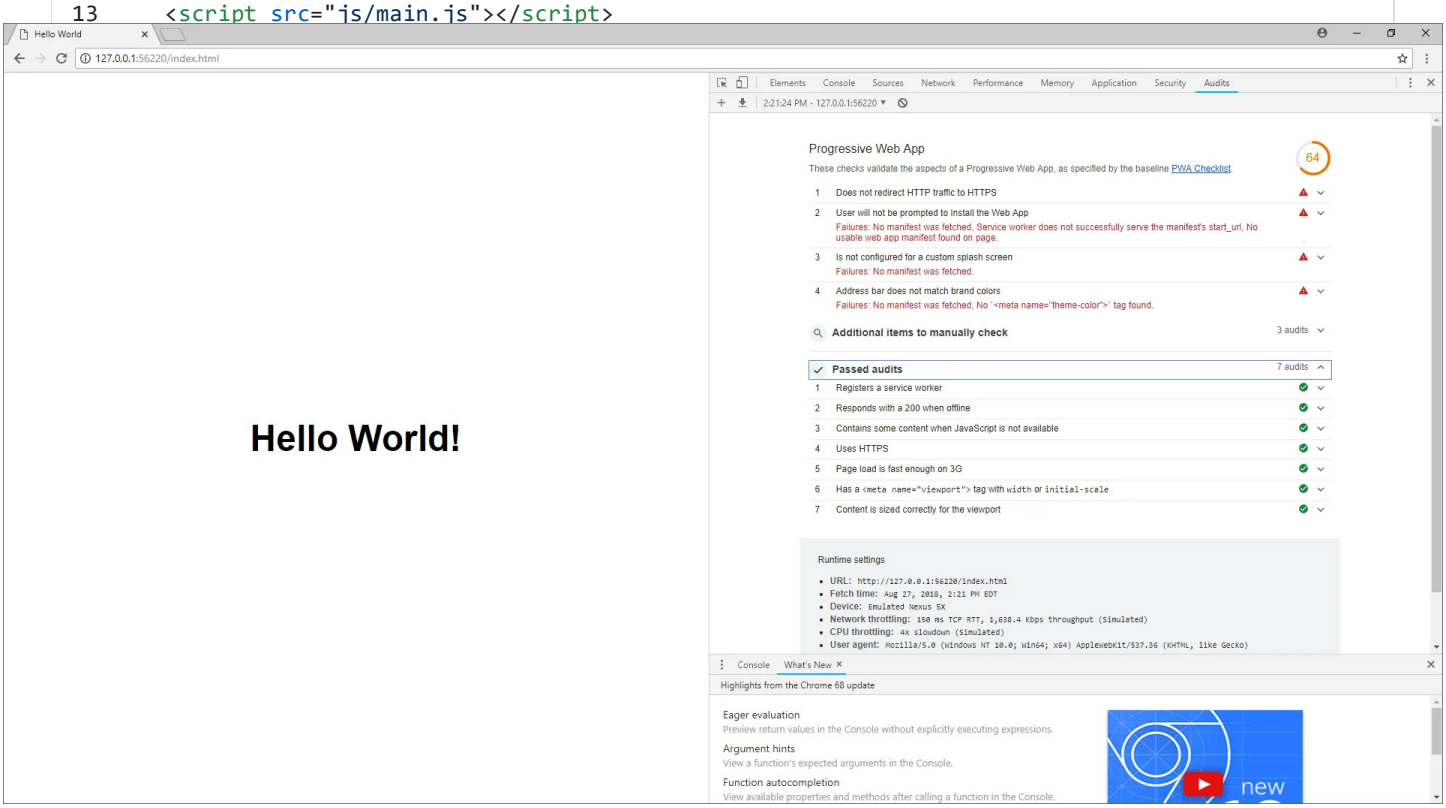
This code simply loads up the service worker script and gets it started.

Add the code to your app by including the script just before the closing `</body>` tag in *index.html*.

```
...
</div>
  <script src="js/main.js"></script>
</body>
```

The revised *index.html* should look like this:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
  <link rel="stylesheet" href="css/style.css">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body class="fullscreen">
  <div class="container">
    <h1 class="title">Hello World!</h1>
  </div>
  <script src="js/main.js"></script>
```



Passing more audits

Add a Manifest

The final requirement for a PWA is to have a manifest file. The manifest is a *json* file that is used to specify how the app will look and behave on devices. For example, you can set the app's orientation and theme color.

Save a file named *manifest.json* in your root folder and add the following content:

```
{
  "name": "Hello World",
  "short_name": "Hello",
  "lang": "en-US",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "white",
  "theme_color": "white"
}
```

manifest.json hosted with ❤️ by **GitHub** **view raw**

manifest.json

For our app we set the title, its background and theme colors, and tell the browser it should be treated as a standalone app without the browser chrome.

Line-by-line the fields are as follows:

name
The title of the app. This is used when prompting the user to install the app. It should be the full title of the app.

short_name
Is the name off the app as it will appear on the app icon. This should be short and to the point.

lang
The default language the app is localized in. In our case, English.

start_url
Tells the browser which page to load up when the app is launched. This will usually be *index.html* but it doesn't need to be.

display
The type of shell the app should appear in. For our app, we are using *standalone* to make it look and feel like a standard native app. There are other settings to make it full screen or include the browser chrome.

background_color
The color of the splash screen that opens when the app launches.

theme_color
Sets the color of the tool bar and in the task switcher.

To add the manifest to your app, link to it inside the *index.html* `head` tag like this:

```
<head>
...
<link rel="manifest" href="/manifest.json">
...
</head>
```

You should also declare the theme color to match the one set in your manifest by adding a meta tag inside the `head`:

```
<head>
...
<meta name="theme-color" content="white"/>
...
</head>
```

If you preview the app and run Lighthouse the score should go up to around 73.

App Icons

After the previous step, you may have noticed that Lighthouse complains about missing app icons. While not strictly necessary for the app to work offline, they do allow your users to add the app to their home screen.

To properly add this feature, you'll need an app icon that's been sized for the browser, Windows, Mac/iPhone and Android. That's a minimum of 7 different sizes: 128x128 px, 144x144 px, 152x152 px, 192x192 px, 256x256 px, 512x512px and a 16x16px favicon. Rather than creating your own, you can download the ones I

created for this tutorial from Github. Save them in the *images* folder and place *favicon.ico* in the project root folder.

Add the icons to your manifest file after the *short_name* property like this:

```
{
  "name": "Hello World",
  "short_name": "Hello",
  "icons": [{
    "src": "images/hello-icon-128.png",
      "sizes": "128x128",
      "type": "image/png"
    }, {
      "src": "images/hello-icon-144.png",
      "sizes": "144x144",
      "type": "image/png"
    }, {
      "src": "images/hello-icon-152.png",
      "sizes": "152x152",
      "type": "image/png"
    }, {
      "src": "images/hello-icon-192.png",
      "sizes": "192x192",
      "type": "image/png"
    }, {
      "src": "images/hello-icon-256.png",
      "sizes": "256x256",
      "type": "image/png"
    }, {
      "src": "images/hello-icon-512.png",
      "sizes": "512x512",
      "type": "image/png"
    }],
  "lang": "en-US",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "white",
  "theme_color": "white"
}
```

manifest.json hosted with ❤ by **GitHub** **view raw**

manifest.json

Finally, add them to *index.html* in the `head` tag:

```
<head>
...
<link rel="icon" href="favicon.ico" type="image/x-icon" />
<link rel="apple-touch-icon" href="images/hello-icon-152.png">
<meta name="theme-color" content="white"/>
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
<meta name="apple-mobile-web-app-title" content="Hello World">
<meta name="msapplication-TileImage" content="images/hello-
icon-144.png">
<meta name="msapplication-TileColor" content="#FFFFFF">
...
</head>
```

If all goes well you should see your Lighthouse score go up to ~91.

Finishing Up

At this point, the coding is finished and the last thing to do is upload the app to a web server. The final requirement for a Progressive Web App is that it must be served via *https*. Setting up a secure web server is out of scope for this tutorial but I've hosted the app on Github as an example:

https://jamesjohnson280.github.io/hello-pwa/

You can also get the full source code to this example on Github:

https://github.com/jamesjohnson280/hello-pwa

Feel free to fork it and make something cool. If you do, send me a link and I'll post it here.

I hope you enjoyed this tutorial and found it useful. Please let me know what you think in the comments below.

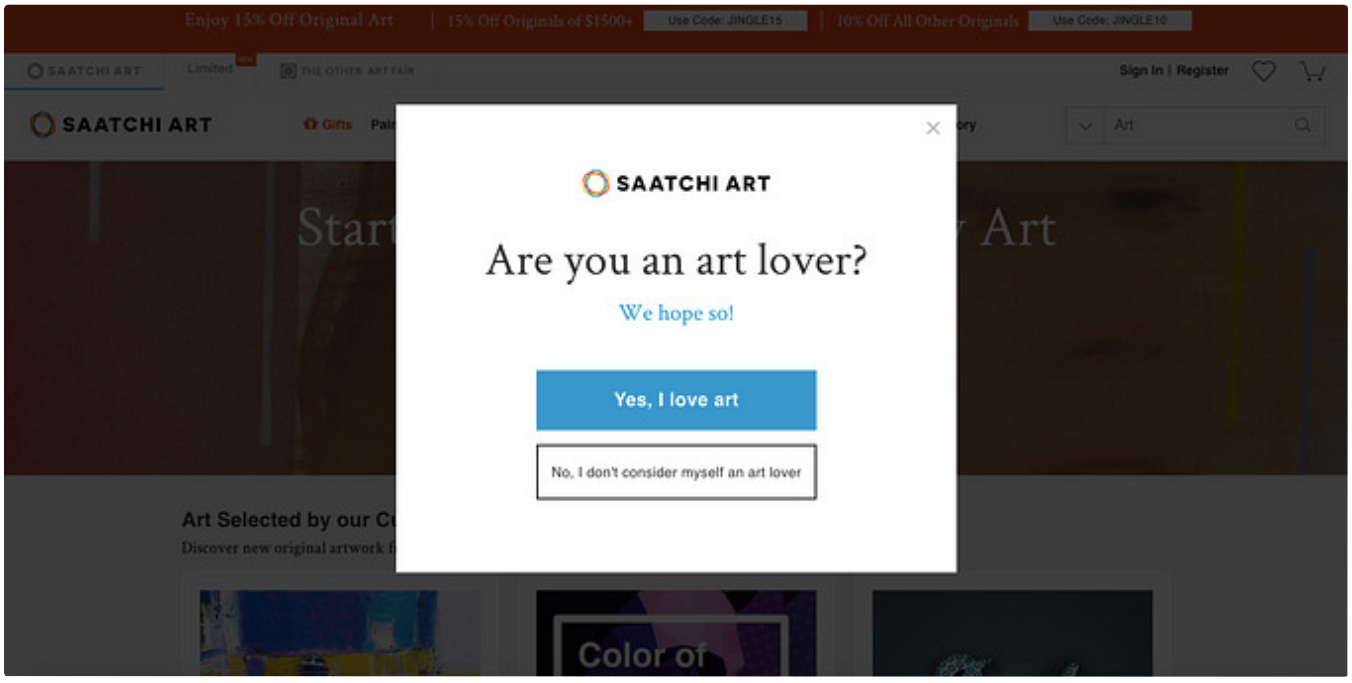
Web Development HTML CSS JavaScript Pwa

Follow

Written by James Johnson

160 Followers · Editor for Design Notes

More from James Johnson and Design Notes



James Johnson in Design Notes

UX for Editorial Design on the Web

Recent high profile pivots aside, the written word is still the most important form of content on the internet.

6 min read · Dec 28, 2017

116 2

James Johnson in Design Notes

My Current Side Project

About a month ago I started working on Lognotes. A journaling app for the web.

2 min read · Sep 14, 2018

8 1

James Johnson in Design Notes

Editorial Design Walkthrough

As a followup to my previous article on Editorial UX, I've written a short walkthrough of how I approach designing an article page for the...

5 min read · Jan 15, 2018

10

See all from James Johnson

See all from Design Notes

Recommended from Medium