By Marcus Hellberg

PART OF TUTORIAL SERIES
Framework independent PWA basics

Production PWA webpack setup

In the previous tutorial, the focus was on learning the basic principles of how Progressive Web Apps work. Some of the issues we identified with the application were:

- We need to remember to update a timestamp in the ServiceWorker any time we update static resources
- Our initial caching can easily fail if network errors cause a single file to fail
- There is no way to update only changed static resources to save bandwidth
- The dynamic caching never expires and can potentially grow large

In this second part of the tutorial, we will use Webpack and Workbox to create an application that is both easier to develop and more effient in production.

Goals

Goals While having a production ready PWA is our end goal, we want to also use the introduction of tooling to make our developer experience nicer.

- Update ServiceWorker automatically based on our resources
- Only require users to download updated assets to minimize bandwidth usage
- Have a development server with automatic reloading
- Make it easy to toggle the ServiceWorker generation on/off

Prerequisites

Node & NPM installed

Download starting point

This tutorial continues from where we left off in the previous one. If you did not complete the previous tutorial, you can download the starting point here.

Install dependencies and restructure the code

Start by initializing NPM:

```
npm init -y
```

The `-y` flag will skip all the questions and just select the defaults. If you want to change defaults, omit the flag.

Once NPM is initialized, add all the dependencies we'll need:

```
npm install --save-dev clean-webpack-plugin copy-webpack-plugin
css-loader html-webpack-plugin mini-css-extract-plugin style-
loader webpack webpack-cli webpack-dev-server webpack-merge
workbox-webpack-plugin
```

⏱ Wait while the entire internet is downloaded.

In order to conform with Webpack's defaults, we need to restructure our code a bit. Create a src folder and move everything except the readme.md and package(-lock).json files into it.

```
|-src
|   |-img
|   |-index.html
|   |-index.js (rename app.js)
|   |-manifest.webmanifest (rename manifest.json)
|   |-schedule.json
|   |-spearkers.json
|   |-styles.css
|   |-sw.js
|-readme.md
|-package.json
|-package-lock.json
```

> Remember to rename `app.js` to `index.js` and `manifest.json` to `manifest.webmanifest`

Configure webpack

Now that we have all the NPM dependencies installed and the folder structure in place, we can start configuring Webpack. Here's an outline of what we are going to do:

- Create separate configs for production and development builds
- Use the Workbox Webpack plugin to generate a ServiceWorker automatically based on the assets we use in Webpack
- Create scripts for production and development

Create a new file, `webpack.config.js`, in the root of the project.

```
webpack.config.js
```

```
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = ({ mode }) => {
  return {
    mode,
    plugins: [
      new CleanWebpackPlugin(),
      new webpack.ProgressPlugin(),
      new HtmlWebpackPlugin({
        filename: 'index.html',
        template: './src/index.html'
      }),
      new CopyWebpackPlugin(
        [{ from: 'src/img', to: 'img/' },
          'src/manifest.webmanifest'],
        { ignore: ['.DS_Store'] })
    ]
  };
};
```

Here’s what we’re doing:

- We take in the `mode` (either `development` or `production`) as a parameter so we can change it on the fly
- We use `CleanWebpackPlugin` to clear out the output folder between runs
- `HtmlWebpackPlugin` copies over our index.html and automatically includes the built assets (JavaScript and CSS)
- `CopyWebpackPlugin` is used to copy over assets that aren’t explicitly referenced from our JavaScript, namely our manifest file and the speaker images.

Now that HtmlWebpackPlugin injects our CSS and JavaScript, we can delete our own imports from `index.html`:

index.html

```
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<title>PWAConf</title>

-<link rel="stylesheet" href="./styles.css">
<link rel="manifest" href="./manifest.json">

...

</footer>
-<script src="./app.js">
```

Finally, we’ll add NPM scripts to run our builds. Open `package.json` and change the `scripts` section to:

package.json

```
"scripts": {
  "webpack": "webpack",
  "webpack-dev-server": "webpack-dev-server",
  "prod": "npm run webpack -- --env.mode production",
  "dev": "npm run webpack-dev-server -- --env.mode development --hot"
},
```

We now have two scripts:

- `npm run dev` will start a development server with hot deployed changes for a super fast develpoment turnaround
- `npm run prod` will create a production build and output it in the dist folder

Before we can run the scripts, we need to update the way we are fetching our JSON. Instead of using the fetch API, we can use Webpack's built in support for loading JSON and that way get them lazy loaded and included in our ServiceWorker.

Remove the fetchJSON method:

index.js

```
-  async fetchJSON(url) {
-    const res = await fetch(url);
-    return res.json();
-  }
```

Then change the calls to use import() instead:

index.js

```
- this.speakers = await this.fetchJSON('./speakers.json');
+ this.speakers = (await import('./speakers.json')).default;
```

and

index.js

```
- const rawSchedule = await this.fetchJSON('./schedule.json');
+ const rawSchedule = (await import('./schedule.json')).default;
```

The one thing to note here is that Webpack returns the JSON as a Module, with the content exported as the default export, so we need to call `.default` to get the content.

Add environment specific configuration to Webpack

Now that we have a basic Webpack setup working, let's add some environment specific configuration so we can easily change how development and production builds work.

Create a new directory `build-utils` and add two files to it: `webpack.development.js` and `webpack.production.js`

```
|-build-utils
  |-webpack.development.js
  |-webpack.production.js
|-src
|-readme.md
|-package.json
|-package-lock.json
```

We'll start by configuring our development build. Open `webpack.development.js` and add the following:

```
webpack.development.js
```

```
const CopyWebpackPlugin = require('copy-webpack-plugin');

module.exports = () => ({
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  plugins: [
    // Copy empty ServiceWorker so install doesn't blow up
    new CopyWebpackPlugin(['src/sw.js'])
  ],
  devtool: 'source-map'
});
```

Here we:

- Tell Webpack to use style-loader and css-loader to process our css file
- Use CopyWebpackPlugin to copy over our empty ServiceWorker to the output directory
- Enable source maps for easier debugging

There are a couple of changes needed in our project to make this work:

At the top of `index.js`, add:

index.js

```
import './styles.css';
```

We also need to change our ServiceWorker. We are going to autogenerate it during the build, but we need to provide a place for the plugin to inject the assets that should be cached:

Replace the contents of `sw.js` with the following:

sw.js

```
if ('workbox' in self) {
  workbox.precaching.precacheAndRoute(self.__precacheManifest || []);
}
```

With this configuration, our ServiceWorker won't do anything in development as `workbox` isn't configured. We'll configure the ServiceWorker build shortly.

Next, let's configure our production build to minify the CSS by putting the following in `webpack.production.js`

```
webpack.production.js
```

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = () => ({
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader']
      }
    ]
  },
  plugins: [new MiniCssExtractPlugin()]
});
```

Finally, we need to merge these into our Webpack configuration based on the mode we are running in. Open `webpack.config.js` and add the following:

```
webpack.config.js
```

```
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');
+ const webpackMerge = require('webpack-merge');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const CleanWebpackPlugin = require('clean-webpack-plugin');

+ const modeConfig = env => require(`./build-utils/webpack.${env.mode}.js

module.exports = ({ mode }) => {
-   return {
+   return webpackMerge(
    {
      mode,
      plugins: [
        new CleanWebpackPlugin(['dist']),
        new webpack.ProgressPlugin(),
        new HtmlWebpackPlugin({
          filename: 'index.html',
          template: './src/index.html'
        }),
        new CopyWebpackPlugin(
          [{ from: 'src/img', to: 'img/' }, 'src/manifest.webmanifest'],
          { ignore: ['.DS_Store'] }
        )
      ]
-   };
+    },
+    modeConfig({ mode })
  );
};
```

If all went well, you should now be able to run both `npm run dev` for development with source maps and autoreload, or `npm run prod` for a production build.

Generating a ServiceWorker with Workbox

The final step is generating the ServiceWorker. We could have included that in the production config if we only wanted to create the ServiceWorker for production. However, it would be nice to also have the ability to develop with the ServiceWorker generation enabled, especially if we are building custom handling for different routes or asset types.

In order to support this, we'll create a way of loading presets based on a parameter we pass to the script. Start by creating a `presets` folder in the `build-utils` folder. Then create `webpack.serviceworker.js` in the `presets` folder and `loadPresets` in the `build-utils` folder. The structure of the new files and folders should look like this:

```
|-build-utils
  |-presets
    |-webpack.serviceworker.js
  |-loadPresets.js
  |-webpack.development.js
  |-webpack.production.js
```

We'll use `webpack.serviceworker.js` to configure the ServiceWorker generation using the `sw.js` template we created earlier.

webpack.serviceworker.js

```
const WorkboxPlugin = require('workbox-webpack-plugin');

module.exports = () => ({
  plugins: [
    new WorkboxPlugin.InjectManifest({
      swSrc: './src/sw.js'
    })
  ]
});
```

Then, update `loadPresets.js` to load presets based on properties we pass in to the script. This is a bit overkill for only the ServiceWorker, but the functionality will be handy later on for other similar functionalities.

webpack.serviceworker.js

```
const webpackMerge = require('webpack-merge');

const loadPresets = (env = { presets: [] }) => {
  const presets = env.presets || [];

  const mergedPresets = [].concat(...[presets]);
  const mergedConfigs = mergedPresets.map(presetName => {
    return require(`./presets/webpack.${presetName}`)(env);
  });

  return webpackMerge({}, ...mergedConfigs);
};

module.exports = loadPresets;
```

This code loads presets from the `presets` folder based on the argument we pass to our script, and uses `webpackMerge` to merge them together.

We then need to tell our `webpack.config.js` to load these presets:

```
webpack.config.js
```

```
const modeConfig = env => require(`./build-utils/webpack.${env.mode}.js`)(
+ const loadPresets = require('./build-utils/loadPresets');

- module.exports = ({ mode }) => {
+ module.exports = ({ mode, presets }) => {
  return webpackMerge(
    {
      mode,
      plugins: [
        new CleanWebpackPlugin(),
        new webpack.ProgressPlugin(),
        new HtmlWebpackPlugin({
          filename: 'index.html',
          template: './src/index.html'
        }),
        new CopyWebpackPlugin(
          [{ from: 'src/img', to: 'img/' }, 'src/manifest.webmanifest'],
          { ignore: ['.DS_Store'] }
        )
      ]
    },
    modeConfig({ mode, presets }),
+   loadPresets({ mode, presets })
  );
};
```

Finally, we'll update the scripts in our `package.json` to use the new ServiceWorker preset.

package.json

```
"scripts": {
  "webpack": "webpack",
  "webpack-dev-server": "webpack-dev-server",
  "prod": "npm run webpack -- --env.mode production --env.presets servicew
  "dev": "npm run webpack-dev-server -- --env.mode development --hot",
  "dev:sw": "npm run webpack-dev-server -- --env.mode development  --env.p
}
```

We added `--env.presets serviceworker` to the prod script, and created a new `dev:sw` script which allows us to run the development server with ServiceWorker generation. For the development target, we removed the `--hot` flag to have the browser refresh and load the new service worker on each change.

Conclusion

This tutorial ended up being more of a Webpack tutorial than a PWA tutorial, but we now have a really nice setup for developing a PWA and taking it to production.

During development, we get changes hotswapped to our browser and do not generate a ServiceWorker to avoid any issues with cached resources interfering.

The production build uses Workbox to generate our ServiceWorker based on the assets Webpack processes. Workbox will also take care of providing those assets when we're offline, so we were able to simplify the ServiceWorker considerably from the previous step, while getting more functionality 💪

1

Share

Comments (4)

Please, login or signup to start new discussion.

Syed Peer 5 days ago

Hi Marcus, Great follow up article. Please share the final project on a git repo. I was getting a number of errors trying to run and unable to resolve.

Hugo Guzman 3 months ago

Hi Marcus. Excellent post! Thank to you I was able to make Webpack work with service workers for my PWA project. The configuration that you did is so flexible. I'm kind of new to Webpack. I was wondering where would you add Babel in this project?

Zpaulo Carraca 4 months ago

Excellent post. I followed the 2 parts and faced minor typos: npm install serve and then serve . I succeeded doing npx serve instead of serve. webpackcli should be webpack-cli loadPresets.js, on code window, gets the label webpack.serviceworker.js

Your work allowed me to put together workbox and webpack. Thanks a lot.

↳ Marcus Hellberg 3 months ago

Yes, you will need `npx serve` if you don't have it installed locally.

Thanks for letting me know about the typos, I'll get them fixed!