ANGULAR UNIVERSITY

PREMIUM QUALITY
ANGULAR TUTORIALS

angular-university.io

Angular Service Worker - Step-By-Step Guide for turning your Application into a PWA

With the Angular Service Worker and the Angular CLI built-in PWA support, it's now simpler than ever to make our web application downloadable and installable, *just like a native mobile application*.

In this post, we will cover how we can configure the Angular CLI build pipeline to generate applications that *in production mode* are downloadable and installable, just like native apps.

We will also add an App Manifest to our PWA, and make the application one-click installable.

I invite you to code along, as we will scaffold an application from scratch using the Angular CLI and we will configure it step-by-step to enable this feature that so far has been exclusive to native apps.

> *We will also see in detail what the CLI is doing so that you can also add the Service Worker to an already existing application if needed.*

Along the way, we will learn about the Angular Service Worker design and how it works under the hood, and see how it works in a way that is quite different than other build-time generated service workers.

Better than current Native Mobile Installation

This service worker download & installation experience that you are about to see in action all happens in the background, without disturbing the user experience, and is actually much better than the current native mobile mechanism that we have for version upgrades.

This PWA-based mechanism even has implicit support for incremental version upgrades - if for example, we change only the CSS, then only the new CSS needs to be reinstalled, instead of having to install the whole application again!

Also, version upgrades can be handled transparently in the background and in a user-friendly way. The user will always see only one version of the application in the multiple tabs that it has opened, but we can also prompt the user and ask if he wants an immediate version upgrade.

Performance Benefits and Offline Support

The performance benefits of installing all our Javascript and CSS bundles on the user browser makes application bootstrapping *much* faster. How much faster? This could go from several times faster to an order of magnitude faster, depending on the application.

Any application, in general, will benefit from the performance boost enabled by this PWA download and installation feature, this is not exclusive to mobile applications.

Having the complete web application downloaded and installed on the user browser is also the **first step** for enabling application offline mode, but note that a complete offline experience requires more than just the download and install feature.

As we can see, the multiple advantages of this new PWA-based application installation feature are huge! So let's go through this awesome feature in detail.

Table Of Contents

In this post, we will cover the following topics:

This post is part of the ongoing Angular PWA Series, here is the complete series:

- Part 1 - Service Workers - Practical Guided Introduction (several examples)
- Part 2 - Angular App Shell - Boosting Application Startup Performance
- Part 3 - Angular Service Worker - Step-By-Step Guide for turning your Application into a PWA

This third part will now focus on the CLI configuration and the Angular Service Worker, for the *specific use case* of Application Download and Installation.

So without further ado, let's get started turning our Angular Application into a PWA!

Step 1 of 7 - Scaffolding an Angular PWA Application using the Angular CLI

With a couple of commands, the CLI will give us a working application that has Download & Installation enabled. The first step to create an Angular PWA is to upgrade the Angular CLI to the latest version:

```
npm install -g @angular/cli@latest
```

If you want to try the latest features, it's also possible to get the next upcoming version:

```
npm install -g @angular/cli@next
```

And with this in place, we can now scaffold an Angular application and add it Angular Service Worker support:

```
ng new angular-pwa-app --service-worker
```

Step 2 of 7 - Understanding How To Add Angular PWA Support Manually

The application scaffolded is almost identical to an application without PWA support. Let's see what this flag includes, in case you need to upgrade an application manually.

We can see that the `@angular/service-worker` package was added to `package.json`. Also, there is a new flag `serviceWorker` set to true in the CLI configuration file `.angular-cli.json`:

```
"apps": [
  {
    "root": "src",
    "outDir": "dist",
    "assets": [
      "assets",
      "favicon.ico"
    ],
    "index": "index.html",
    "main": "main.ts",
    "polyfills": "polyfills.ts",
    "test": "test.ts",
    "tsconfig": "tsconfig.app.json",
    "testTsconfig": "tsconfig.spec.json",
    "prefix": "app",
    "styles": [
      "styles.css"
    ],
    "scripts": [],
    "environmentSource": "environments/environment.ts",
    "environments": {
      "dev": "environments/environment.ts",
      "prod": "environments/environment.prod.ts"
    },
    "serviceWorker": true
  }
]
```

What does the `serviceWorker` flag do?

This flag will cause the production build to include a couple of extra files in the output dist folder:

- The Angular Service Worker file `ngsw-worker.js`
- The runtime configuration of the Angular Service Worker `ngsw.json`

> *Note that `ngsw` stands for Angular Service Worker*

We will cover these two files in detail, right now let's see what else the CLI has added for PWA support.

What does the `ServiceWorkerModule` do?

The CLI has also included in our application root module the Service Worker module:

```

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ServiceWorkerModule.register('/ngsw-worker.js', { enabled: environment.
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}

```

This module provides a couple of injectable services:

- `SwUpdate` for managing application version updates
- `SwPush` for doing server Web Push notifications

More than that, this module registers the Angular Service Worker in the browser (if Service Worker support is available), by loading the `ngsw-worker.js` script in the user's browser via a call to `navigator.serviceWorker.register()`.

The call to `register()` causes the `ngsw-worker.js` file to be loaded in a separate HTTP request. And with this in place, there is only one thing missing to turn our Angular application into a PWA.

The build configuration file `ngsw-config.json`

The CLI has also added a new configuration file called `ngsw-config.json`, which configures the Angular Service Worker runtime behavior, and the generated file comes with some intelligent defaults.

> *Depending on your application, you might not even have to edit this file!*

Here is what the file looks like:

```
{
  "index": "/index.html",
  "assetGroups": [{
    "name": "app",
    "installMode": "prefetch",
    "resources": {
      "files": [
        "/favicon.ico",
        "/index.html"
      ],
      "versionedFiles": [
        "/*.bundle.css",
        "/*.bundle.js",
        "/*.chunk.js"
      ]
    }
  }, {
    "name": "assets",
    "installMode": "lazy",
    "updateMode": "prefetch",
    "resources": {
      "files": [
        "/assets/**"
      ]
    }
  }]
}
```

There is a lot going on here, so let's break it down step-by-step. This file contains the default caching behavior or the Angular Service Worker, which targets the application static asset files: the `index.html`, the CSS and Javascript bundles.

Step 3 of 7 - Understanding the Angular Service Worker runtime caching mechanism

The Angular Service Worker can cache all sorts of content in the browser Cache Storage.

This is a Javascript-based key/value caching mechanism that is not related to the standard browser `Cache-Control` mechanism, and the two mechanisms can be used separately.

The goal of the `assetGroups` section of the configuration file is to configure exactly what HTTP requests get cached in Cache Storage by the Angular Service Worker, and there are two cache configuration entries:

- one entry named `app`, for all single page application files (all the application index.html, CSS and Javascript bundles plus the favicon)

- another entry named `assets`, for any other assets that are also shipped in the dist folder, such as for example images, but that are not necessarily necessary to run every page

Caching static files that *are* the application itself

The files under the app section *are* the application: a single page is made of the combination of its `index.html` plus its CSS and Js bundles. These files are needed for every single page of the application and cannot be lazy loaded.

In the case of these files, we want to cache them as early and permanently as possible, and this is what the `app` caching configuration does.

The `app` files are going to be proactively downloaded and installed in the background by the Service Worker, and that is what the install mode `prefetch` means.

The Service worker will not wait for these files to be requested by the application, instead, it will download them ahead of time and cache them so that it can serve them the next time that they are requested.

This is a good strategy to adopt for the files that together make the application itself (the index.html, CSS and Javascript bundles) because we already know that we will need them all the time.

Caching other auxiliary static assets

On the other hand, the assets files are cached only if they are requested (meaning the install mode is `lazy`), but if they were ever requested once, and if a new version is available then they will be downloaded ahead of time (which is what update mode `prefetch` means).

Again this is a great strategy for any assets that get downloaded in a separate HTTP request such as images because they might not always be needed depending on the pages the user visits.

But if they were needed once then its likely that we will need the updated version as well, so we might as well download the new version ahead of time.

Again these are the defaults, but we can adapt this to suit our own application. In the specific case of the `app` files though, it's unlikely that we would like to use another strategy.

After all, the `app` caching configuration is *the* download and installation feature itself that we are looking for. Maybe we use other files, outside the bundles produced by the CLI? In that case, we would want to adapt our configuration.

It's important to keep in mind that with these defaults, we already have a downloadable and installable application ready to go, so let's try it out!

Step 4 of 7 - Running and Understanding the PWA Production Build

Let's first add something visual to the application that clearly identifies a given version running in the user browser. For example, we can replace the contents of the `app.component.html` file with the following:

```html

<h1>Version V1 is runnning ...</h1>

```

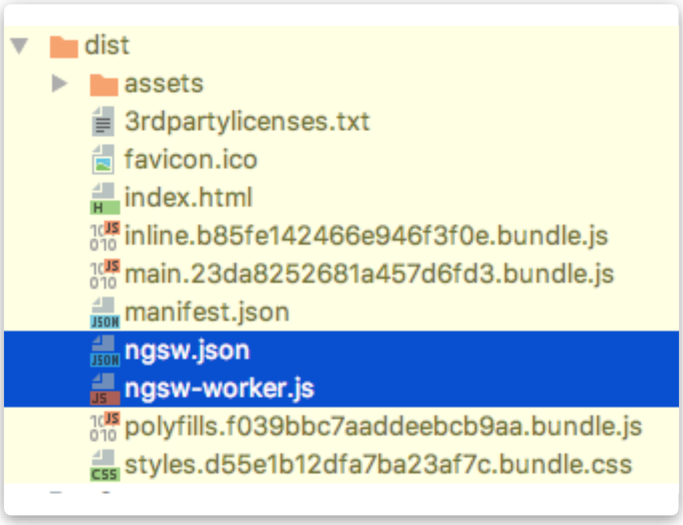
Let's now build this Hello world PWA app. The Angular Service Worker will only be available in production mode, so let's first do a production build of our application:

```
ng build --prod
```

This will take a moment, but after a while, we will have our application build available inside the `dist` folder.

The Production Build Folder

Let's have a look to see what we have in our build folder, here are the most all the files generated:



```
dist
  assets
  3rdpartylicenses.txt
  favicon.ico
  index.html
  inline.b85fe142466e946f3f0e.bundle.js
  main.23da8252681a457d6fd3.bundle.js
  manifest.json
  ngsw.json
  ngsw-worker.js
  polyfills.f039bbc7aaddeebcb9aa.bundle.js
  styles.d55e1b12dfa7ba23af7c.bundle.css
```

As we can see, the `serviceWorker` flag in the `.angular-cli.json` build configuration file has caused the Angular CLI to include a couple of extra files (highlighted in blue).

What is the `ngsw-worker.js` file?

This file is *the* Angular Service Worker itself. Like all Service workers, it get's delivered via its own separate HTTP request so that the browser can track if it has changed, and apply it the Service Worker Lifecycle (covered in detail in this post).

It's the `ServiceWorkerModule` that will trigger the loading of this file indirectly, by calling `navigation.serviceWorker.register()`.

Note that the Angular Service Worker file `ngsw-worker.js` will always be the same with each build, as it gets copied by the CLI straight from `node_modules`.

This file will then remain the same until you upgrade to a new Angular version that contains a new version of the Angular Service Worker.

What is the `ngsw.json` file?

This is the *runtime* configuration file, that the Angular Service worker will use. This file is built based on the `ngsw-config.json` file, and contains all the information needed by the Angular Service Worker to know at runtime about which files it needs to cache, and when.

Here is what the `ngsw.json` runtime configuration file looks like:

```
{
  "configVersion": 1,
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "updateMode": "prefetch",
      "urls": [
        "/favicon.ico",
        "/index.html",
        "/inline.5646543f86fbfdc19b11.bundle.js",
        "/main.3bb4e08c826e33bb0fca.bundle.js",
        "/polyfills.55440df0c9305462dd41.bundle.js",
        "/styles.1862c2c45c11dc3dbcf3.bundle.css"
      ],
      "patterns": []
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "urls": [],
      "patterns": []
    }
  ],
  "dataGroups": [],
  "hashTable": {
    "/inline.5646543f86fbfdc19b11.bundle.js": "1028ce05cb8393bd53706064e3a8
    "/main.3bb4e08c826e33bb0fca.bundle.js": "ae15cc3875440d0185b46b4b73bfa7
    "/polyfills.55440df0c9305462dd41.bundle.js": "c3b13e2980f9515f4726fd262
    "/styles.1862c2c45c11dc3dbcf3.bundle.css": "3318b88e1200c77a5ff691c03ca
    "/favicon.ico": "84161b857f5c547e3699ddfbffc6d8d737542e01",
    "/index.html": "cfdca0ab1cec8379bbbf8ce4af2eaa295a3f3827"
  }
}
```

As we can see, this file is an expanded version of the `ngsw-config.json` file, where all the wilcard urls have been applied and replaced with the paths of any files that matched them.

How does the Angular Service Worker use the `ngsw.json` file?

The Angular Service Worker is going to load these files either proactively in the case of install mode `prefetch`, or as needed in the case of install mode `lazy`, and it will also store the files in Cache Storage.

This loading is going to happen in the background, as the user first loads the application. The next time that the user refreshes the page, then the Angular Service Worker is going to intercept the HTTP requests, and will serve the cached files instead of getting them from the network.

Note that each asset will have a hash entry in the hash table. If we do any modification to any of the files listed there (even if its only one character), we will have a completely different hash in the following Angular CLI build.

The Angular Service Worker will then know that this file has a new version available on the server that needs to be loaded at the appropriate time.

Now that we have a good overview of everything that is going on, let's see this in action!

Step 5 of 7 - Launching an Angular PWA in Production Mode

Let's then start the application in production mode, and in order to do that, we are going to need a small web server. A great choice is `http-server`, so let's install it:

```
npm install -g http-server
```

Let's then go into the `dist` folder, and start the application in production mode:

```
cd dist
http-server -c-1 .
```

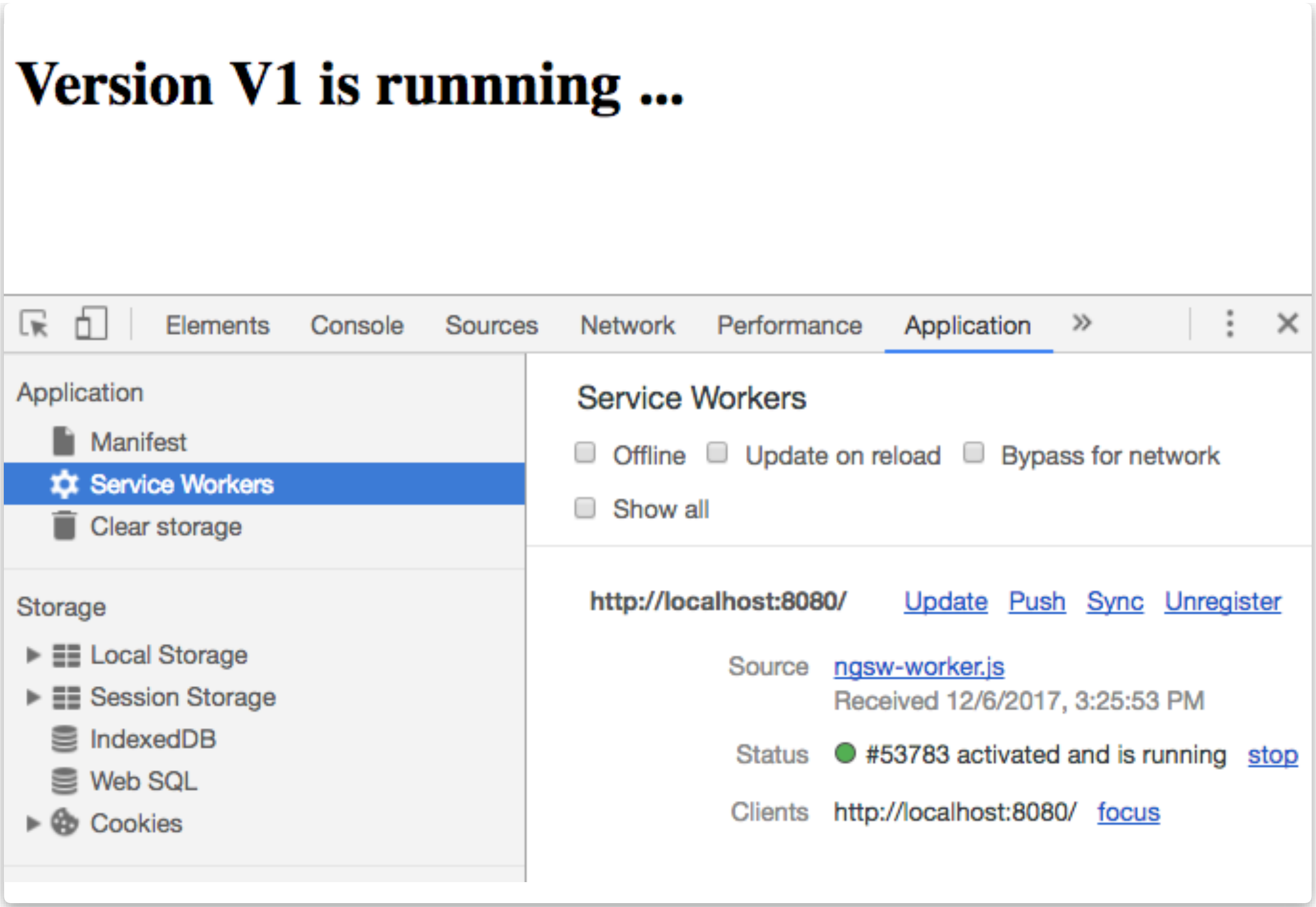
The `-c-1` option will disable server caching, and a server will normally be running on port `8080`, serving the production version of the application.

Note that if you had port `8080` blocked, the application might be running on `8081`, `8082`, etc., the port used is logged in the console at startup time.

If you have a REST API running locally on another server for example in port 9000, you can also proxy any REST API calls to it with the following command:

```
http-server -c-1 --proxy http://localhost:9000 .
```

With the server running, let's then head over to `http://localhost:8080`, and see what we have running using the Chrome Dev Tools:


Version V1 is runnning ...

Service Workers

Offline, Update on reload, Bypass for network, Show all (all unchecked)

http://localhost:8080/ Update Push Sync Unregister

Source ngsw-worker.js
Received 12/6/2017, 3:25:53 PM

Status ● #53783 activated and is running stop

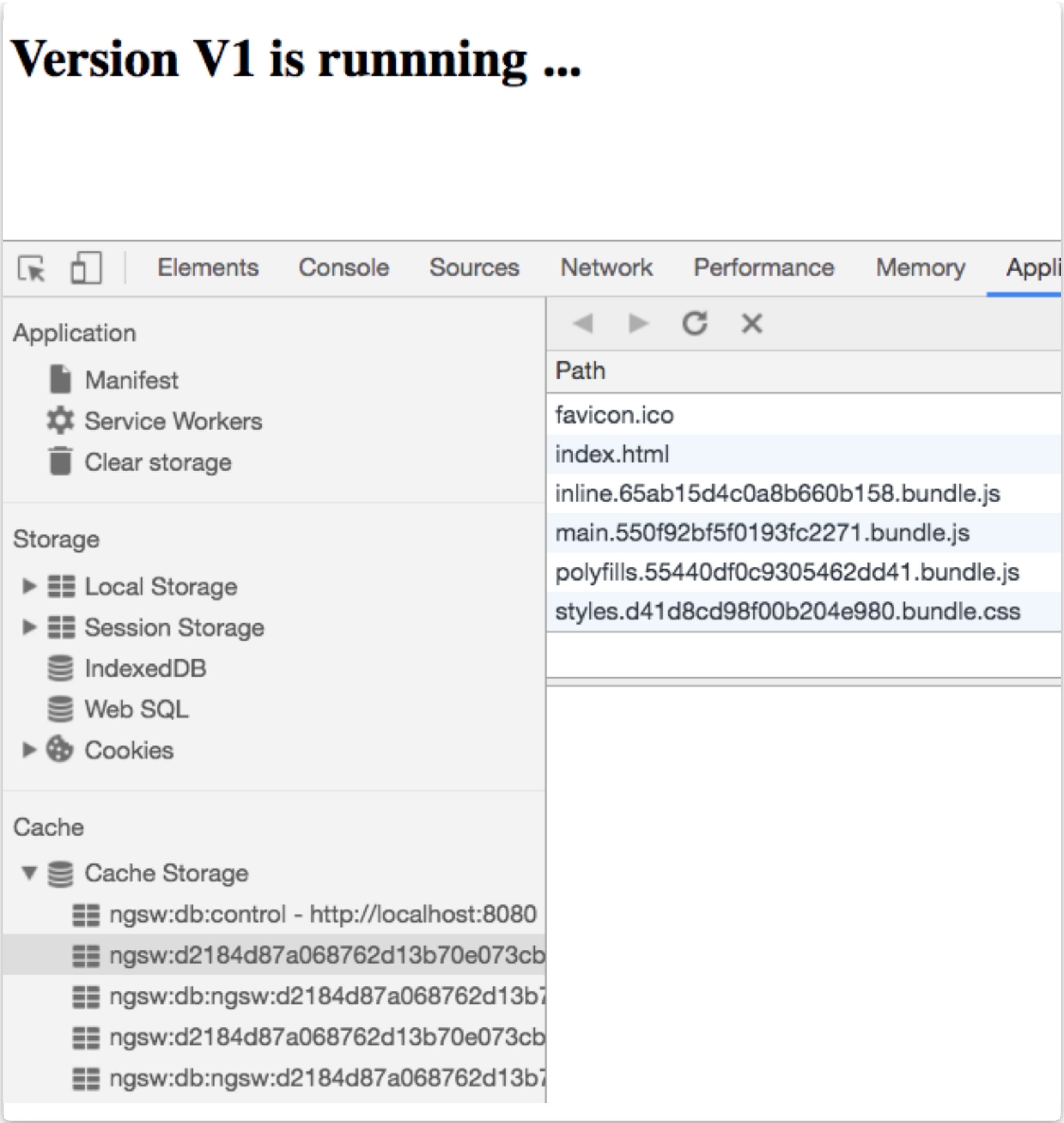
Clients http://localhost:8080/ focus

As we can see, we have now version V1 running and we have installed a Service Worker with source file `ngsw-worker.js`, as expected!

Where are the Javascript and CSS bundles stored?

All the Javascript and CSS files, plus even the `index.html` have all been downloaded in the background and installed in the browser for later use.

These files can all be found in Cache Storage, using the Chrome Dev Tools:


Version V1 is runnning ...

Path
favicon.ico
index.html
inline.65ab15d4c0a8b660b158.bundle.js
main.550f92bf5f0193fc2271.bundle.js
polyfills.55440df0c9305462dd41.bundle.js
styles.d41d8cd98f00b204e980.bundle.css

The Angular Service Worker will start serving the application files the next time you load the page. Try to hit refresh, you will likely notice that the application starts much faster.

> *Note that the performance improvement will be much more noticeable in production than on* `localhost`

Taking the Application Offline

To confirm that the application is indeed downloaded and installed into the user browser, let's do one conclusive test: let's bring the server down by hitting Ctrl+C.

Let's now hit refresh after shutting down the `http-server` process: you might be surprised that the application is still running, we get the exact same screen!

On the console we will find the following message:

```
An unknown error occurred when fetching the script.
ngsw-worker.js Failed to load resource:
net::ERR_CONNECTION_REFUSED
```

It looks like all the Javascript and CSS bundle files that make the application where fetched from somewhere else other than the network because the application is still running.

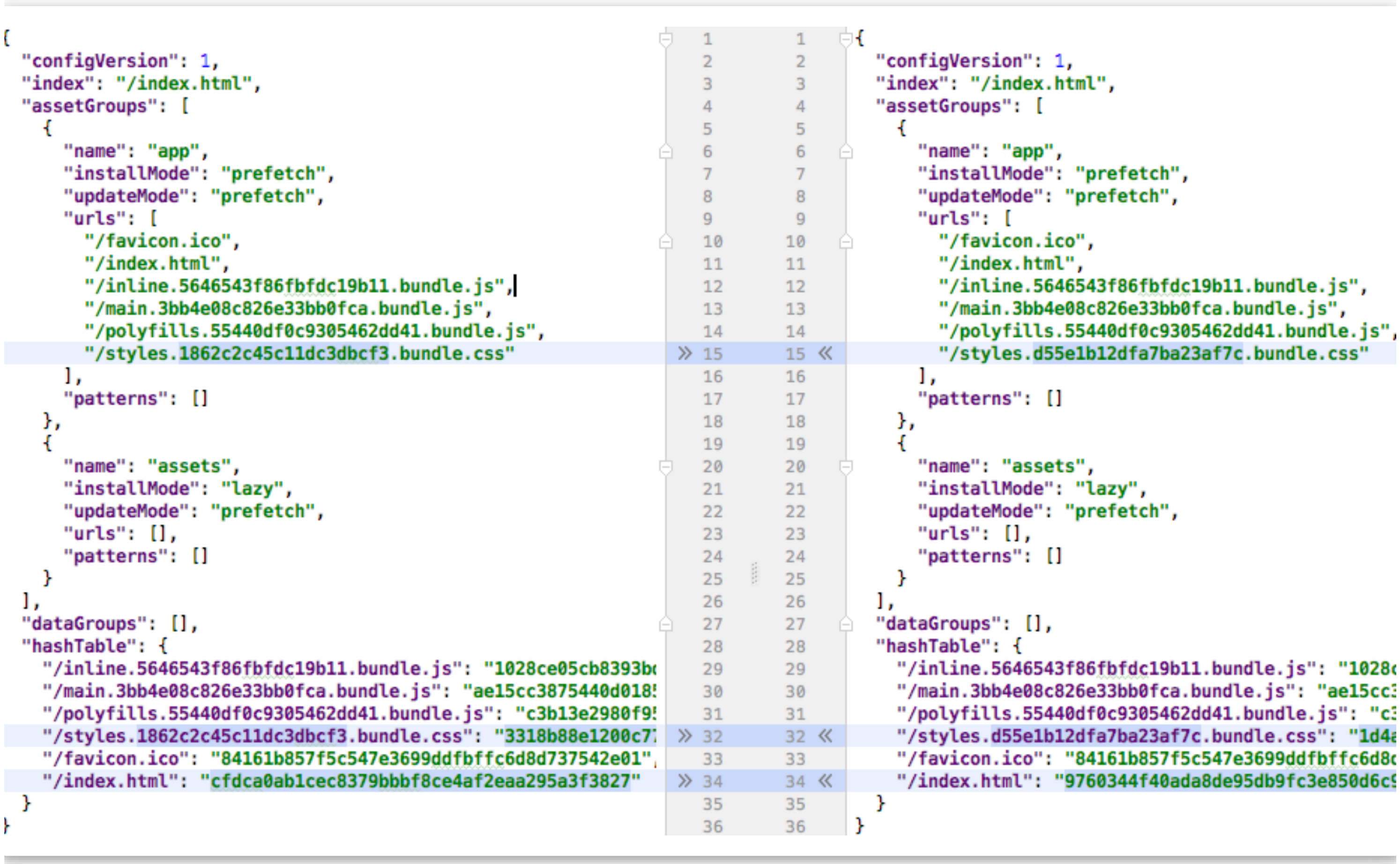
The only file that was attempted to be fetched from the network was the Service Worker file itself, which is expected (more on this in a moment).

Step 6 of 7 - Deploying a new Application Version, Understanding Version Management

This is a great feature, but isn't it a bit dangerous to cache everything, what if there is a bug and we want to ship a new version of the code?

Let's say that we made a small change to the application, like for example editing a global style in the `styles.css` file. Before running the production build again, let's keep the previous version of `ngsw.json`, so that we can see what changed.

Let's now run the production build again, and compare the generated `ngsw.json` file:



```
{
  "configVersion": 1,
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "updateMode": "prefetch",
      "urls": [
        "/favicon.ico",
        "/index.html",
        "/inline.5646543f86fbfdc19b11.bundle.js",
        "/main.3bb4e08c826e33bb0fca.bundle.js",
        "/polyfills.55440df0c9305462dd41.bundle.js",
        "/styles.1862c2c45c11dc3dbcf3.bundle.css"
      ],
      "patterns": []
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "urls": [],
      "patterns": []
    }
  ],
  "dataGroups": [],
  "hashTable": {
    "/inline.5646543f86fbfdc19b11.bundle.js": "1028ce05cb8393b
    "/main.3bb4e08c826e33bb0fca.bundle.js": "ae15cc3875440d018
    "/polyfills.55440df0c9305462dd41.bundle.js": "c3b13e2980f9
    "/styles.1862c2c45c11dc3dbcf3.bundle.css": "3318b88e1200c7
    "/favicon.ico": "84161b857f5c547e3699ddfbffc6d8d737542e01",
    "/index.html": "cfdca0ab1cec8379bbbf8ce4af2eaa295a3f3827"
  }
}
```

```
{
  "configVersion": 1,
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "updateMode": "prefetch",
      "urls": [
        "/favicon.ico",
        "/index.html",
        "/inline.5646543f86fbfdc19b11.bundle.js",
        "/main.3bb4e08c826e33bb0fca.bundle.js",
        "/polyfills.55440df0c9305462dd41.bundle.js",
        "/styles.d55e1b12dfa7ba23af7c.bundle.css"
      ],
      "patterns": []
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "urls": [],
      "patterns": []
    }
  ],
  "dataGroups": [],
  "hashTable": {
    "/inline.5646543f86fbfdc19b11.bundle.js": "1028
    "/main.3bb4e08c826e33bb0fca.bundle.js": "ae15cc
    "/polyfills.55440df0c9305462dd41.bundle.js": "c
    "/styles.d55e1b12dfa7ba23af7c.bundle.css": "1d4
    "/favicon.ico": "84161b857f5c547e3699ddfbffc6d8
    "/index.html": "9760344f40ada8de95db9fc3e850d6c
  }
}
```

As we can see, the only thing that changed in the build output was the CSS bundle, all the remaining files are unchanged except for `index.html` (where the new bundle is being loaded).

How does the Angular Service Worker install new application versions?

Every time that the user reloads the application, the Angular Service Worker will check to see if there is a new `ngsw.json` file available on the server.

This is for consistency with the standard Service Worker behavior, and to avoid having stale versions of the application running for a long time. Stale versions could potentially contain bugs or even be completely broken, so its essential to check frequently if a new application version is available on the server.

In our case, the previous and the new versions of the `ngsw.json` file will be compared, and the new CSS bundle will be downloaded and installed in the background.

The next time the user reloads the page, the new application version will be shown!

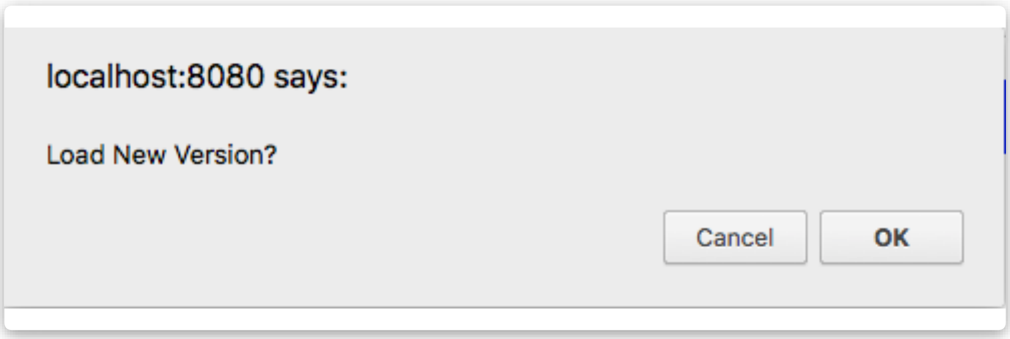
Informing the user a new version is available

For long-running SPA applications that the user might have opened for hours, we might want to check periodically to see if there is a new version in the server, and install it in the background.

For that it's possible to show a dialog to the user and ask if he wants to reload, using the `SwUpdate` service and the `checkForUpdate()` method.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent  implements OnInit {

    constructor(private swUpdate: SwUpdate) {
    }

    ngOnInit() {

        // check back to the server to see if a new ngsw.json is available
    // this can be called inside setInterval, upon router navigation, e
    this.swUpdate.checkForUpdate();

    this.swUpdate.available.subscribe(() => {
      if(confirm("New version available. Load New Version?")) {
        window.location.reload();
      }
    });
  }
}
```

In this case, the first time that the user loads the application after the deployment of the new version, it will see a dialog box with the message:



localhost:8080 says:

Load New Version?

If the user clicks OK, the application will then be reloaded and the new version will be shown! Note that normally we don't need to call `checkForUpdate()` at application startup, as that is done automatically.

And with this in place, we have a downloadable and installable Angular PWA Application, with built-in version management!

The last piece that we are now missing for a complete one-click installation experience, is to ask the user to install the application to its Home Screen.

Step 7 of 7 - One-Click Install with the App Manifest

Let's now make our application one-click installable, and note that this is optional, meaning that we can use the Angular Service Worker *without* an App Manifest.

On the other hand, in order for the App Manifest to work, we need a Service Worker running on the page! By providing a standard `manifest.json` file, the user will be asked to install the application to the Home Screen.

When will the Install to Home Screen button be shown to the user?

There are a couple of conditions for this to work, one of them being that the application needs to run over HTTPS and have a Service Worker.

Also, the option for installing to Home Screen will only be shown if certain *extra conditions* are met.

There is a constantly evolving heuristic that will determine if the button "Install To Home Screen" will be shown or no to the user, that typically has to do with the number of times that the user visited the site, how often, etc.

A Sample App Manifest file

In order to make this functionality work, we need to first create a `manifest.json` file, and we are going to place in the root of our application next to our `index.html`:

```
{
  "dir": "ltr",
  "lang": "en",
  "name": "Angular PWA ",
  "scope": "/",

  "display": "fullscreen",
  "start_url": "http://localhost:8080/",
  "short_name": "Angular PWA",
  "theme_color": "transparent",
  "description": "Sample PWA App",
  "orientation": "any",
  "background_color": "transparent",
  "related_applications": [],
  "prefer_related_applications": false,
  "icons": [
    {
      "src": "/favicon.ico",
      "sizes": "16x16 32x32"
    },
    {
      "src": "/assets/android-icon-36x36.png",
      "sizes": "36x36",
      "type": "image/png",
      "density": "0.75"
    },
    {
      "src": "/assets/android-icon-48x48.png",
      "sizes": "48x48",
      "type": "image.png",
      "density": "1.0"
    },
    {
      "src": "/assets/android-icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png",
      "density": "1.5"
    },
    {
      "src": "/assets/android-icon-96x96.png",
      "sizes": "96x96",
      "type": "image/png",
      "density": "2.0"
    },
    {
      "src": "/assets/android-icon-144x144.png",
      "sizes": "144x144",
      "type": "image/png",
      "density": "3.0"
    },
    {
      "src": "/assets/android-icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png",
      "density": "4.0"
    }
  ]
}
```

This file defines what the icon installed to the Home screen will look like, and it also defines a couple of other native UI parameters.

Setting up the CLI for including the App Manifest

In order to have the App Manifest file in our production build, we are going to configure the CLI to copy this file to the dist folder, together with the complete assets folder.

We can configure that in file `.angular-cli.json`:

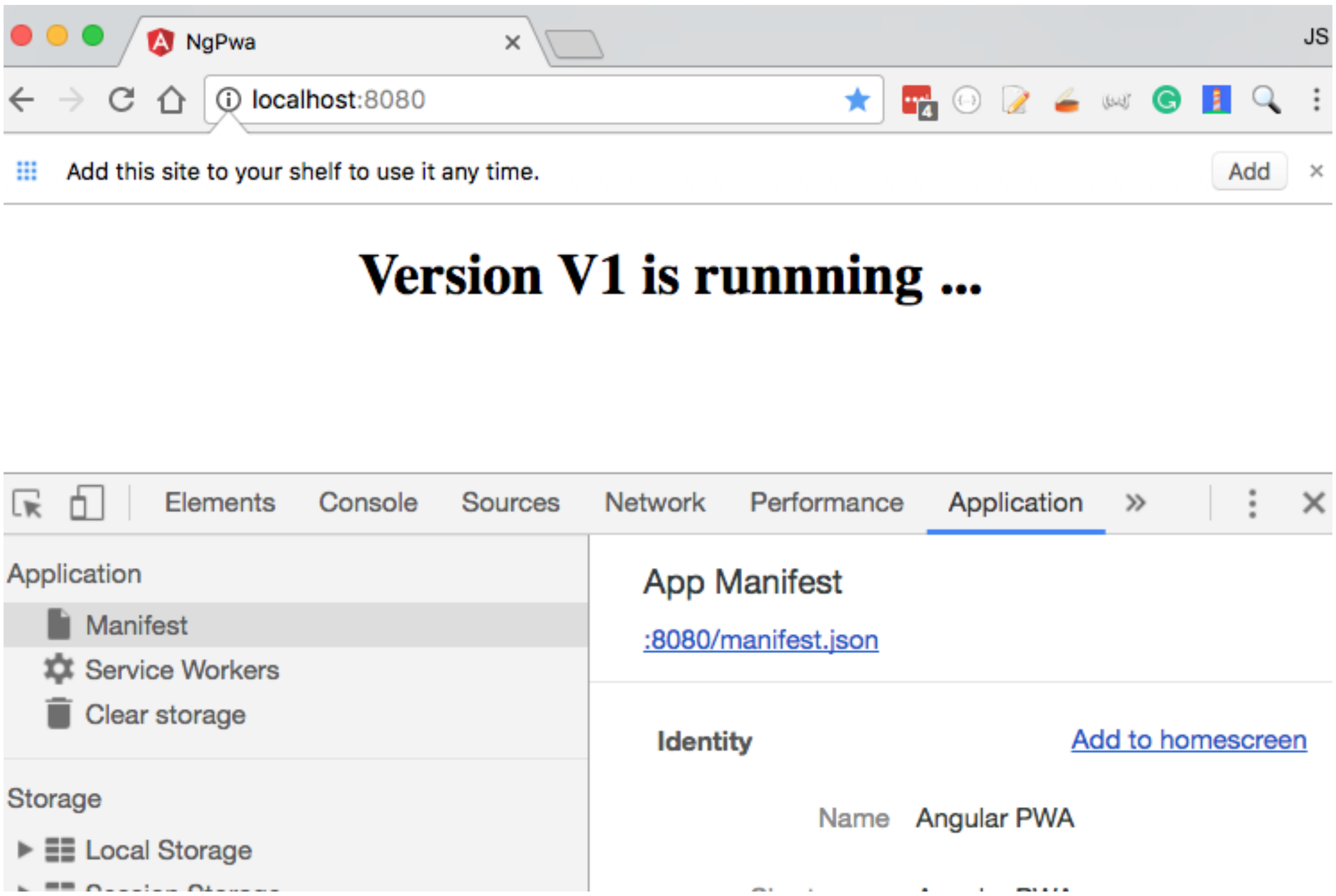
```

  "apps": [
    {
      "root": "src",
      "outDir": "dist",
      "assets": [
        "assets",
        "manifest.json",
        "favicon.ico"
      ],
```

With this in place, we now have a `manifest.json` file in production. but if we now reload the application, most likely, nothing will happen!

Triggering Install to Home Screen

What we mean by that is that most likely no "Install To Home Screen" button will be shown to the user, and this is because the heurestic for displaying this button to the user was not met yet.

But we can trigger the button with the Chrome Dev Tools, in the Manifest tab using the **Add To Home Screen** option:



Version V1 is runnning ...

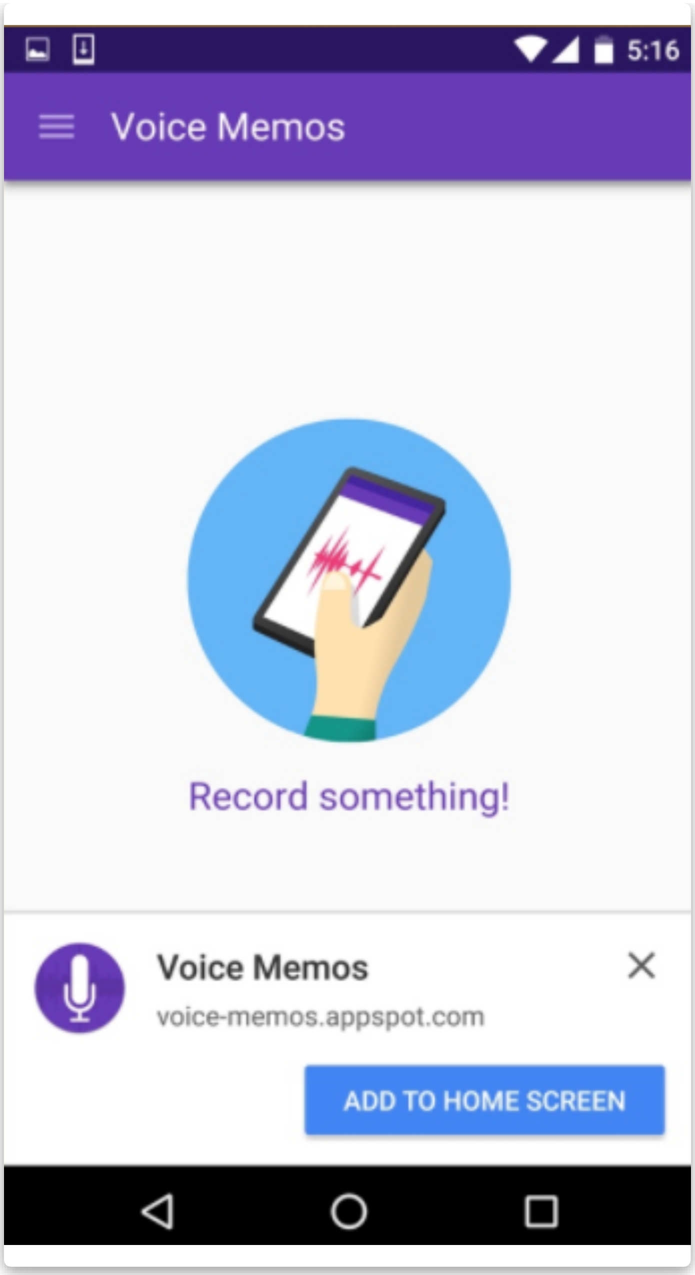
App Manifest

:8080/manifest.json

Identity | Add to homescreen

Name Angular PWA

As we can see, on a Mac the look and feel of the button is still in early stages, but this is what the button should look like on mobile:



And with this in place, we now have a complete one-click download and installation experience for our application.

Summary

Native-like Application Download and Installation is now simpler than ever with the Angular Service Worker and the Angular CLI!

The performance benefits that this feature brings to any application (desktop included) are huge and can be added incrementally to a standard PWA in a progressive way.

Any application (mobile or not) can benefit from a much faster startup time, and this feature can be made to work out of the box with some intelligent defaults provided by the Angular CLI.

I hope that this post helps with getting started with the Angular Service Worker and that you enjoyed it! If you want to learn more about other Angular PWA features, have a look at the other posts of the complete series:

- Part 1 - Service Workers - Practical Guided Introduction (several examples)
- Part 2 - Angular App Shell - Boosting Application Startup Performance
- Part 3 - Angular Service Worker - Step-By-Step Guide for turning your Application into a PWA