


# Building an Angular 2 Application for Production

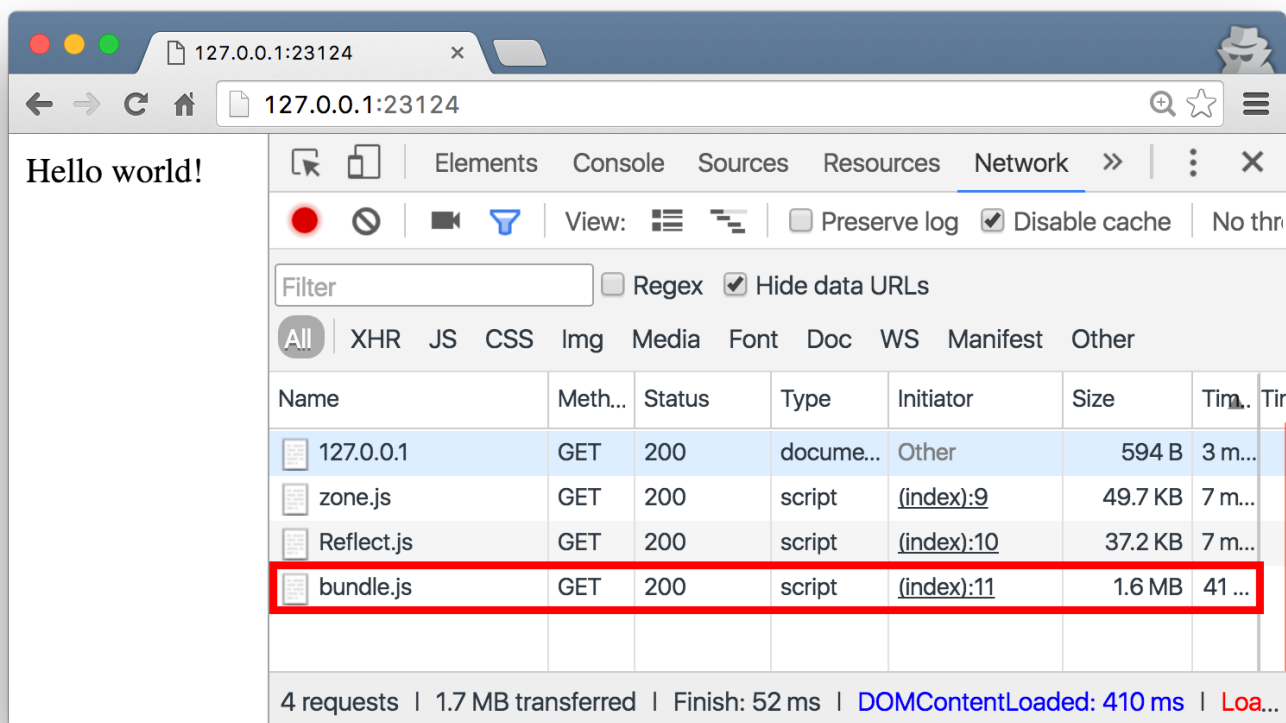
 [blog.mgechev.com/2016/06/26/tree-shaking-angular2-production-build-rollup-javascript/](http://blog.mgechev.com/2016/06/26/tree-shaking-angular2-production-build-rollup-javascript/)

Building an Angular 2 Application for Production was published on June 26, 2016 .

[Progressive Web Applications](#) help us build native-like web apps, thanks to amazing tools such as Service Workers, IndexedDB, App Shell etc. Once the browser downloads all the static assets required by our app, the active Service Worker can cache all of them locally.

This way the user may experience slowdown during the initial page load, but once the page has been successfully loaded for first time, each next time the load will be instant!

In order to help developers take advantage of the technologies behind the PWA as easy as possible, the Angular team is working on the Angular [mobile-toolkit](#). However, a big concern for developing high-performance Angular 2 is the framework size itself. For instance, a simple non-optimized “Hello world!” Angular 2 application, bundled with [browserify](#) is 1.6MB! This is suicidal when your users are supposed to download it via an unreliable 3G connection.



This is main reason Angular is (was) criticized for. During the keynote of [ng-conf](#), [Brad Green](#) (manager of the Angular team) mentioned that the core team managed to drop the size of the “Hello world!” app to **less than 50K!**

In this blog post we'll explain all the steps we need to go through in order to achieve such results!

*The experiments from this blog posts are located here:*

- [Simple build with minification](#).

- [Tree-shaking and minification.](#)
- [ngc, tree-shaking and minification.](#)

## Sample Application

In order to get a better understanding of the optimizations explained below, let's first describe the sample application that we're going to apply them on.

Our app is going to consist of the following two files:

```
// app.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: 'Hello world!'
})
export class AppComponent {}

..and

// main.ts

import { AppComponent } from './app.component';
import { bootstrap } from '@angular/platform-browser-dynamic';

bootstrap(AppComponent);
```

In `app.component.ts` we have a single component with a template which is going to render the text "Hello world!". `main.ts` is responsible for bootstrapping the application, using the `bootstrap` method exported by the `@angular/platform-browser-dynamic` package.

Our `index.html` page looks like:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <my-app></my-app>
  <script src="/node_modules/zone.js/dist/zone.js"></script>
  <script src="/node_modules/reflect-metadata/Reflect.js"></script>
  <script src="dist/bundle.js"></script>
</body>
</html>
```

And here's the directory layout:

```
.
├── app
```

```
|   ├── app.component.ts
|   └── main.ts
├── dist
├── index.html
├── package.json
├── tsconfig.json
├── node_modules
├── typings
└── typings.json
```

`dist` is where the output is going to live (i.e. application bundles), we're using `typings` and `npm` in order to manage, respectively, the ambient TypeScript type definitions and the dependencies of our application.

## Step 1 - Minification and Compression

The two most obvious optimizations that we can apply are minification and compression. You can find the example explained in this section [in my GitHub profile](#). I'd recommend you to clone the repository and run:

```
npm install
```

Now we can explore the build process by taking a look at `package.json`:

```
"scripts": {
  "clean": "rm -rf dist",
  "typings": "typings install",
  "serve": "http-server . -p 5556",
  "postinstall": "npm run typings",
  "build": "npm run clean && tsc",
  "build_prod": "npm run build && browserify -s main dist/main.js > dist/bundle.js && npm run minify",
  "minify": "uglifyjs dist/bundle.js --screw-ie8 --compress --mangle --output dist/bundle.min.js"
}
```

We have a simple `clean` script which removes the `dist` directory. In order to install all the required typings once the `npm install` command completes its execution, we have `typings install` as `postinstall` script.

Our `build` script first cleans the `dist` directory and after that compiles our application by using the TypeScript compiler. This will produce two files - `main.js` and `app.component.js`, in the `dist` directory.

By using `SystemJS` we can already run them in the browser, but since we want to reduce the number of HTTP requests made by the browser in the process of loading the app, we can create a single bundle. This is what we do in `build_prod`:

```
npm run build && browserify -s main dist/main.js > dist/bundle.js && npm run minify
```

In the script above, first we use the TypeScript compiler. Once the app has been compiled, all we need to do is to create a “[standalone](#)” bundle with entry point the `dist/main.js` file, and output the bundles content to `bundle.js` within the `dist` directory.

In order to try the app you can use:

```
npm run build_prod
npm run serve
```

open `http://localhost:5556`

## Size Analysis

Now lets see what the bundle size is:

```
$ ls -lah bundle.js
-rw-r--r--  1 mgechev  staff   1.6M Jun 26 12:01 bundle.js
```

Wow...so we reached the disastrous point we described above! However, the bundle contains a bunch of useless content such as:

- Unused functions, variables...
- A lot of whitespace.
- Comments.
- Non-mangled variables.

In order to reduce the size of the bundle we can now use the minify script:

```
uglifyjs dist/bundle.js --screw-ie8 --compress --mangle --output dist/bundle.min.js
```

It takes the `bundle.js` file, and optimizes it. The output is now produced in `dist/bundle.min.js`, and its size is:

```
$ ls -lah bundle.min.js
-rw-r--r--  1 mgechev  staff   702K Jun 26 12:01 bundle.min.js
```

So, we reduced the size of the bundle to 702K only by applying a simple minification!

## Compression of the Application

Most HTTP servers support compression of the content, with gzip. The requested by the browser resources are compressed by the web server and sent through the network. Responsibility of the client is to decompress them.

Lets find out what the size of the compressed bundle is:

```
$ gzip bundle.min.js
$ ls -lah bundle.min.js.gz
-rw-r--r--  1 mgechev  staff   152K Jun 26 12:01 bundle.min.js.gz
```

We reduced the size of the bundle with another ~78% only by applying compression! But we can do even better!

## Step 2 - Tree-Shaking

In this section we'll use very important property of the ES2015 modules - they are tree-shakable!

What does tree-shaking means:

| *Tree-shaking is excluding unused exports from bundles.*

Because the ES2015 modules are static, by performing a static code analysis over them we can decide which exports are used and which are not used in our application. In contrast, CommonJS modules are not always tree-



shakable because of the dynamic nature of their format.

For instance:

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Algorithm you want to use for sorting the numbers? ', answer => {
  const sort = require(answer);
  sort([42, 1.618, 4]);
});
```

It is impossible to guess which algorithm will be chosen by the user by performing a static code analysis.

With ES2015 we can do something like:

```
import {Graphs} from './graphs';
import {Algorithms} from './algorithms';

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Algorithm you want to use for sorting the numbers? ', answer => {
  const sort = Algorithms[answer];
```

```
    sort([42, 1.618, 4]);
  });
```

In this way, we will be able to perform tree-shaking and remove the `./graphs` module from the final bundle because since we're not using `Graphs` anywhere. On the other hand, we need to import all sorting algorithms because we have a level of non-determinism - we're not sure what the input of the user is going to be so we can't get rid of anything from `Algorithms`.

## Applying Tree-Shaking with Rollup

Rollup.js is a module bundler which is optimized for ES2015 modules. It is a great, pluggable tool which allows us to perform tree-shaking over ES2015 and CommonJS (in most cases) by using a plugin.

We're going to integrate Rollup in the example above, trying to achieve even smaller bundle size! The example from this section is [available here](#).

Now, let's take a look at the `scripts` section in our `package.json` in order to explore the build process:

```
"scripts": {
  "clean": "rm -rf dist",
  "typings": "typings install",
  "serve": "http-server . -p 5557",
  "postinstall": "npm run typings",
  "build": "tsc -p tsconfig.json",
  "rollup": "rollup -f iife -c -o dist/bundle.es2015.js",
  "es5": "tsc --target es5 --allowJs dist/bundle.es2015.js --out dist/bundle.js",
  "minify": "uglifyjs dist/bundle.js --screw-ie8 --compress --mangle --output dist/bundle.min.js",
  "build_prod": "npm run clean && npm run build && npm run rollup && npm run es5 && npm run minify"
}
```

We have the same `clean`, `typings`, `serve`, `postinstall`, `minify` and `build` scripts like above. The new things here are `rollup`, `es5`, `build_prod`.

`rollup` is responsible for bundling our app and perform tree-shaking in the process.

TypeScript supports ES2015 modules, which means that we can apply tree-shaking directly over our non-transpiled app. This is further simplified by the [TypeScript plugin for Rollup](#) which allows us to perform the transpilation as the part of the bundling. This would work great if the dependencies of our application were distributed as TypeScript as well. However, Angular 2 is distributed as ES5 and ES2015 (within the `esm` directory), and RxJS is distributed as ES5 and ES2015 (in the `rxjs-es` package).

Since we can't apply tree-shaking directly over the original TypeScript files of our app, we'll first need to transpile it to TypeScript, after that create an ES2015 bundle by using `rollup`, and in the end transpile it to ES5.

Because of these changes in the flow of the build process, there's a very important difference between the `tsconfig.json` presented in the example above, and the one used in this section:

```
{
  "compilerOptions": {
    "target": "es2015",
    "module": "es2015",
    // ...
  }
}
```

```

    },
    "compileOnSave": false,
    "files": [
      "app/main.ts"
    ]
  }
}

```

Our target version here is es2015 in order to transpile the TypeScript application to an ES2015 one, with ES2015 modules.

Now we can explore the rollup script:

```
rollup -f iife -c -o dist/bundle.es2015.js
```

Above we tell rollup to bundle the modules as IIFE (Immediately-Invoked Function Expression), use the configuration file provided in the root of the project (rollup.config.js) and output the bundle as bundle.es2015.js in dist.

Now lets take a look at the rollup.config.js file:

```

import nodeResolve from 'rollup-plugin-node-resolve';

class RollupNG2 {
  constructor(options){
    this.options = options;
  }
  resolveId(id, from){
    if (id.startsWith('rxjs/')){
      return `${__dirname}/node_modules/rxjs-es/${id.replace('rxjs/', '')}.js`;
    }
  }
}

const rollupNG2 = (config) => new RollupNG2(config);

export default {
  entry: 'dist/main.js',
  sourceMap: true,
  moduleName: 'main',
  plugins: [
    rollupNG2(),
    nodeResolve({
      jsnext: true, main: true
    })
  ]
};

```

Great thing about this configuration file is that it is pure JavaScript! We export the configuration object, and declare inside of it the entry point of the application, the module name (required if we use IIFE bundling), we also declare that we want to have sourceMaps, and the set of plugins that we want to use.

We use the nodeResovle plugin for rollup in order to hint the bundler that we want to use node-like module resolution. Once the bundler finds import like @angular/core, for instance, it'll go to node\_modules/@angular/core



and read the `package.json` file there. Once it finds property called `main:jsnext`, the bundler will use the file set as its value. If such property is not found, the bundler will use the file pointed by `main`.

A problem comes that RxJS is distributed as ES5 by default. In order to deal with this problem, for bundling the required RxJS operations we'll use the package `rxjs-es`, which is already [available in package.json](#). After installing this module, we need to make sure that the module bundler will use `rxjs-es` instead of `rxjs` from `node_modules`. This is exactly what the purpose of the `rollupNG2` plugin is - to translate all the `rxjs/*` imports to `rxjs-es/*` ones.

Alright, now if we run:

```
npm run clean && npm run build && npm run rollup
```

We'll get the bundle `bundle.es2015.js`. Now let's transpile this bundle to ES5.

This can be easily achieved by using the `es5` script:

```
tsc --target es5 --allowJs dist/bundle.es2015.js --out dist/bundle.js
```

We use the TypeScript compiler and output the ES5 bundle to `dist/bundle.js` (note that this script may throw an error for `Duplicate identifier _subscribe`, which is not problematic at this point).

In order to get our final optimized, bundle we need to invoke `npm run minify`. To verify that the application still works use:

```
npm run serve
```

## Size Analysis

Let's find out what the size of our ES5 bundle is:

```
$ ls -lah bundle.js
-rw-r--r--  1 mgechev  staff   1.4M Jun 26 13:00 bundle.js
```

Great! This is 200K smaller than the previous section! Now let's see what the bundle size will be after minification:

```
$ ls -lah bundle.min.js
-rw-r--r--  1 mgechev  staff   484K Jun 26 13:01 bundle.min.js
```

702K to 484K...not bad at all! After gzipping we get:

```
$ ls -lah bundle.min.js.gz
-rw-r--r--  1 mgechev  staff   115K Jun 26 13:01 bundle.min.js.gz
```

About 25% reduction of the bundle size! But I'm sure we can do even better!

*Credits: Igor Minar published similar experiments in the official Angular 2 repository. They can be found [here](#).*

## Using ngc

As [static code analysis enthusiast](#), I'm following the progress around the Angular compiler (`ngc`).

The core idea of `ngc` is to process the templates of the components in our applications and [generate VM friendly](#), tree-shakable code. This can happen either run-time or build-time, but since in run-time compilation the application is already loaded in the browser we can't take advantage of tree-shaking. The project is [located here](#).



Although in the previous example we already applied decent tree-shaking we still can do better! Why? Well, having an HTML template rollup is not completely sure what parts of Angular we can get rid of from the final bundle since HTML is not something that rollup can analyze at all. That's why we can:

- Compile our application (including templates) to TypeScript with ngc.
- Perform tree-shaking with rollup (this way we will get *at least* as small bundle as above).
- Transpile the bundle to ES5.
- Minify the bundle.
- Gzip it!

Alright, lets begin! The code explained in the paragraphs below can be [found here](#).

Lets take a look at the scripts in package.json:

```
"scripts": {
  "typings": "typings install",
  "serve": "http-server . -p 5558",
  "postinstall": "npm i -f @angular/tsc-wrapped@latest && rm -rf
node_modules/@angular/compiler-cli/node_modules && npm run typings",
  "clean": "rm -rf dist && rm -rf app/*.ngfactory.ts && cd compiled && ls | grep -v main-ngc.ts
| xargs rm && cd ..",
  "build": "tsc -p tsconfig-tsc.json",
  "rollup": "rollup -f iife -c -o dist/bundle.es2015.js",
  "es5": "tsc --target es5 --allowJs dist/bundle.es2015.js --out dist/bundle.js",
  "minify": "uglifyjs dist/bundle.js --screw-ie8 --compress --mangle --output
dist/bundle.min.js",
  "ngc": "ngc -p . && cp app/* compiled",
  "build_prod": "npm run clean && npm run ngc && npm run build && npm run rollup && npm run es5
&& npm run minify"
}
```

build\_prod just confirms the order into which the individual actions need to be performed. Lets take a look at the clean method, since it looks quite complex this time:

```
rm -rf dist && rm -rf app/*.ngfactory.ts && cd compiled && ls | grep -v main-ngc.ts | xargs rm
&& cd ..
```

What we do here is to remove the dist directory, all files which match app/\*.ngfactory.ts and also everything except main-ngc.ts from the compiled directory. ngc produces \*.ngfactory.ts files. Since they are artifacts from the build process we'd want to remove them before the next build. But why we remove everything except main-ngc.ts from the compiled directory? Lets take a look at the file's content:

```
import {ComponentResolver, ReflectiveInjector, coreBootstrap} from '@angular/core';
import {BROWSER_APP_PROVIDERS, browserPlatform} from '@angular/platform-browser';

import {AppComponentNgFactory} from './app.component.ngfactory';

const appInjector = ReflectiveInjector.resolveAndCreate(BROWSER_APP_PROVIDERS,
browserPlatform().injector);
coreBootstrap(AppComponentNgFactory, appInjector);
```

This is what the process of bootstrapping a precompiled app at the moment of writing is. Notice that we bootstrap the app by using `AppComponentNgFactory`, and import it from `app.component.ngfactory`, i.e. a generated by `ngc` file. So, once we compile our app with `ngc`, we want to move everything in the `compiled` directory, and after that invoke the TypeScript compiler, in order to make it produce ES2015 code. That is why our `tsconfig.json` is slightly changed as well:

```
// tsconfig-tsc.json
{
  "compilerOptions": {
    "target": "es2015",
    "module": "es2015",
    // ...
  },
  "compileOnSave": false,
  "files": [
    "compiled/main-ngc.ts"
  ]
}
```

We are using `compiled/main-ngc.ts` as entry file. Also notice that we have two `tsconfig` files: one for `ngc` and one for `tsc`.

Alright...now lets run `npm run ngc`. Once the scripts completes its execution, here's the directory structure of the app:

```
.
├── README.md
├── app
│   ├── app.component.ngfactory.ts
│   ├── app.component.ts
│   └── main.ts
├── compiled
│   ├── app.component.ngfactory.ts
│   ├── app.component.ts
│   ├── main-ngc.ts
│   └── main.ts
├── dist
├── index.html
├── package.json
├── rollup.config.js
├── tsconfig-tsc.json
├── tsconfig.json
├── typings
└── typings.json
```

Now we can transpile the application to ES2015:

```
npm run build
```

*Notice that the `postinstall` script is quite complex as well. This is due to [this bug](#) introduced in `@angular/compiler-cli` by RC.3*

At this point we already have the ES2015 version of our app located in `dist`. The only steps left are:

- Tree-shaking.
- Transpilation from ES2015 to ES5.
- Minification.
- Gzipping.

This is process we're already familiar with so let's invoke the individual scripts one by one without providing further explanation:

```
# Bundle the app
npm run rollup

# Notice that this command may fail with "Duplicate _identifier"
# This will not have any impact over the end result.
npm run es5

# Minify the app
npm run minify
```

In order to make sure that everything works you can use:

```
npm run serve
```

## Size Analysis

Let's see how big is our precompiled, tree-shaked app!

```
$ ls -lah bundle.min.js
-rw-r--r--  1 mgechev  staff   210K Jun 26 14:22 bundle.min.js
```

The application got more than twice smaller than it was without ngc!

If we gzip it, we'll get the following results:

```
$ ls -lah bundle.min.js.gz
-rw-r--r--  1 mgechev  staff    49K Jun 26 14:22 bundle.min.js.gz
```

The final result is 49K!

*Credits: Rob Wormald who did experiments with [ngc here](#).*

## Conclusion

As we can see from the chart above, by applying a set of optimizations over our production bundle we can reduce the size of our application up to 33 times!

This is thanks to a couple of techniques:

- Optimization by performing static-code analysis, more specifically tree-shaking.
- Minification (including mangling).
- Compression with gzip.

Definitely there's some overhead at first, before being able to get all the things going on in this process. Luckily, in the end, all this is going to happen automatically with tools like [angular-cli](#), and [angular2-seed](#).

As a matter of fact, we are already planning to add automated production build, which performs all the listed steps above in [angular2-seed](#). This is definitely going to happen in near future, once we make sure that all tools explained above are mature enough.

By then, we can easily reduce the bundle size to ~150K only by applying minification and compression!

---