AHEAD-OF-TIME COMPILATION

Learn how to use Ahead-of-time compilation

This cookbook describes how to radically improve performance by compiling *Ahead of Time* (AOT) during a build process.

Table of Contents

Overview

An Angular application consist largely of components and their HTML templates. Before the browser can render the application, the components and templates must be converted to executable JavaScript by the *Angular compiler*.

> Watch compiler author Tobias Bosch explain the Angular Compiler at AngularConnect 2016.

You can compile the app in the browser, at runtime, as the application loads, using the ***Just-in-Time* (JIT) compiler**. This is the standard development approach shown throughout the documentation. It's great .. but it has shortcomings.

JIT compilation incurs a runtime performance penalty. Views take longer to render because of the in-browser compilation step. The application is bigger because it includes the Angular compiler and a lot of library code that the application won't actually need. Bigger apps take longer to transmit and are slower to load.

Compilation can uncover many component-template binding errors. JIT compilation discovers them at runtime which is later than we'd like.

The ***Ahead-of-Time* (AOT) compiler** can catch template errors early and improve performance by compiling at build time as you'll learn in this chapter.

Ahead-of-time (AOT) vs *Just-in-time* (JIT)

There is actually only one Angular compiler. The difference between AOT and JIT is a matter of timing and tooling. With AOT, the compiler runs once at build time using one set of libraries; With JIT it runs every time for every user at runtime using a different set of libraries.

Why do AOT compilation?

Faster rendering

With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.

Fewer asynchronous requests

The compiler *inlines* external html templates and css style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

Smaller Angular framework download size

There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

Detect template errors earlier

The AOT compiler detects and reports template binding errors during the build step before users can see them.

Better security

AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

Compile with AOT

Prepare for offline compilation

Take the Setup as a starting point. A few minor changes to the lone `app.component` lead to these two class and html files:

<button (click)="toggleHeading()">Toggle Heading</button> <h1 *ngIf="showHeading">Hello Angular</h1> <h3>List of Heroes</h3> <div *ngFor="let hero of heroes">{{hero}} import { Component } from '@angular/core'; @Component({ moduleId: module.id, selector: 'my-app', templateUrl: './app.component.html' }) export class AppComponent { showHeading = true; heroes = ['Magneta', 'Bombasto', 'Magma', 'Tornado']; toggleHeading() { this.showHeading = !this.showHeading; } }

Install a few new npm dependencies with the following command:

```
npm install @angular/compiler-cli @angular/platform-server --save
```

You will run the `ngc` compiler provided in the `@angular/compiler-cli` npm package instead of the TypeScript compiler (`tsc`).

`ngc` is a drop-in replacement for `tsc` and is configured much the same way.

`ngc` requires its own `tsconfig.json` with AOT-oriented settings. Copy the original `src/tsconfig.json` to a file called `tsconfig-aot.json` (on the project root), then modify it to look as follows.

tsconfig-aot.json

```
{ "compilerOptions": { "target": "es5", "module": "es2015", "moduleResolution": "node", "sourceMap": true, "emitDecoratorMetadata": true, "experimentalDecorators": true, "lib": ["es2015", "dom"], "noImplicitAny": true, "suppressImplicitAnyIndexErrors": true }, "files": [ "src/app/app.module.ts", "src/main.ts" ], "angularCompilerOptions": { "genDir": "aot", "skipMetadataEmit" : true } }
```

The `compilerOptions` section is unchanged except for one property. **Set the `module` to `es2015`**. This is important as explained later in the Tree Shaking section.

What's really new is the `ngc` section at the bottom called `angularCompilerOptions`. Its `"genDir"` property tells the compiler to store the compiled output files in a new `aot` folder.

The `"skipMetadataEmit" : true` property prevents the compiler from generating metadata files with the compiled application. Metadata files are not necessary when targeting TypeScript files, so there is no reason to include them.

Component-relative Template URLS

The AOT compiler requires that `@Component` URLS for external templates and css files be *component-relative*. That means that the value of `@Component.templateUrl` is a URL value *relative* to the component class file. For example, an `'app.component.html'` URL means that the template file is a sibling of its companion `app.component.ts` file.

While JIT app URLs are more flexible, stick with *component-relative* URLs for compatibility with AOT compilation.

JIT-compiled applications that use the SystemJS loader and *component-relative* URLs *must set the* `@Component.moduleId` *property to* `module.id`. The `module` object is undefined when an AOT-compiled app runs. The app fails with a null reference error unless you assign a global `module` value in the `index.html` like this:

```
<script>window.module = 'aot';</script>
```

> Setting a global `module` is a temporary expedient.

Compiling the application

Initiate AOT compilation from the command line using the previously installed `ngc` compiler by executing:

```
node_modules/.bin/ngc -p tsconfig-aot.json
```

> Windows users should surround the `ngc` command in double quotes:
>
> "node_modules/.bin/ngc" -p tsconfig-aot.json

`ngc` expects the `-p` switch to point to a `tsconfig.json` file or a folder containing a `tsconfig.json` file.

After `ngc` completes, look for a collection of *NgFactory* files in the `aot` folder (the folder specified as `genDir` in `tsconfig-aot.json`).

These factory files are essential to the compiled application. Each component factory creates an instance of the component at runtime by combining the original class file and a JavaScript representation of the component's template. Note that the original component class is still referenced internally by the generated factory.

> The curious can open the `aot/app.component.ngfactory.ts` to see the original Angular template syntax in its intermediate, compiled-to-TypeScript form.
>
> JIT compilation generates these same *NgFactories* in memory where they are largely invisible. AOT compilation reveals them as separate, physical files.

Do not edit the *NgFactories*! Re-compilation replaces these files and all edits will be lost.

Bootstrap

The AOT path changes application bootstrapping.

Instead of bootstrapping `AppModule`, you bootstrap the application with the generated module factory, `AppModuleNgFactory`.

Make a copy of `main.ts` and name it `main-jit.ts`. This is the JIT version; set it aside as you may need it later.

Open `main.ts` and convert it to AOT compilation. Switch from the `platformBrowserDynamic.bootstrap` used in JIT compilation to `platformBrowser().bootstrapModuleFactory` and pass in the AOT-generated `AppModuleNgFactory`.

Here is AOT bootstrap in `main.ts` next to the original JIT version:

```
import { platformBrowser } from '@angular/platform-browser'; import { AppModuleNgFactory } from '../aot/src/app/app.module.ngfactory'; console.log('Running AOT compiled'); platformBrowser().bootstrapModuleFactory(AppModuleNgFactory); import { platformBrowserDynamic } from '@angular/platform-browser-dynamic'; import { AppModule } from './app/app.module'; console.log('Running JIT compiled'); platformBrowserDynamic().bootstrapModule(AppModule);
```

Be sure to recompile with `ngc`!

Tree Shaking

AOT compilation sets the stage for further optimization through a process called *Tree Shaking*. A Tree Shaker walks the dependency graph, top to bottom, and *shakes out* unused code like dead needles in a Christmas tree.

Tree Shaking can greatly reduce the downloaded size of the application by removing unused portions of both source and library code. In fact, most of the reduction in small apps comes from removing unreferenced Angular features.

For example, this demo application doesn't use anything from the `@angular/forms` library. There is no reason to download Forms-related Angular code and tree shaking ensures that you don't.

Tree Shaking and AOT compilation are separate steps. Tree Shaking can only target JavaScript code. AOT compilation converts more of the application to JavaScript, which in turn makes more of the application "Tree Shakable".

Rollup

This cookbook illustrates a Tree Shaking utility called *Rollup*.

Rollup statically analyzes the application by following the trail of `import` and `export` statements. It produces a final code *bundle* that excludes code that is exported, but never imported.

Rollup can only Tree Shake `ES2015` modules which have `import` and `export` statements.

> Recall that `tsconfig-aot.json` is configured to produce `ES2015` modules. It's not important that the code itself be written with `ES2015` syntax such as `class` and `const`. What matters is that the code uses ES `import` and `export` statements rather than `require` statements.

Install the Rollup dependencies with this command:

npm install rollup rollup-plugin-node-resolve rollup-plugin-commonjs rollup-plugin-uglify --save-dev

Next, create a configuration file (`rollup-config.js`) in the project root directory to tell Rollup how to process the application. The cookbook configuration file looks like this.

rollup-config.js

```
import rollup from 'rollup' import nodeResolve from 'rollup-plugin-node-resolve' import commonjs from 'rollup-plugin-commonjs'; import uglify from 'rollup-plugin-uglify' export default { entry: 'src/main.js', dest: 'src/build.js', // output a single application bundle sourceMap: false, format: 'iife', onwarn: function(warning) { // Skip certain warnings // should intercept ... but doesn't in some rollup versions if ( warning.code === 'THIS_IS_UNDEFINED' ) { return; } // intercepts in some rollup versions if ( warning.indexOf("The 'this' keyword is equivalent to 'undefined'") > -1 ) { return; } // console.warn everything else console.warn( warning.message ); }, plugins: [ nodeResolve({jsnext: true, module: true}), commonjs({ include: 'node_modules/rxjs/**', }), uglify() ] }
```

It tells Rollup that the app entry point is `src/app/main.js` . The `dest` attribute tells Rollup to create a bundle called `build.js` in the `dist` folder. It overrides the default `onwarn` method in order to skip annoying messages about the AOT compiler's use of the `this` keyword.

Then there are plugins.

Rollup Plugins

Optional plugins filter and transform the Rollup inputs and output.

RxJS Rollup expects application source code to use `ES2015` modules. Not all external dependencies are published as `ES2015` modules. In fact, most are not. Many of them are published as *CommonJS* modules.

The *RxJs* observable library is an essential Angular dependency published as an ES5 JavaScript *CommonJS* module.

Luckily there is a Rollup plugin that modifies *RxJs* to use the ES `import` and `export` statements that Rollup requires. Rollup then preserves in the final bundle the parts of `RxJS` referenced by the application.

rollup-config.js (CommonJs to ES2015 Plugin)

```
commonjs({ include: 'node_modules/rxjs/**', }),
```

Minification

Rollup Tree Shaking reduces code size considerably. Minification makes it smaller still. This cookbook relies on the *uglify* Rollup plugin to minify and mangle the code.

rollup-config.js (CommonJs to ES2015 Plugin)

```
uglify()
```

> In a production setting, you would also enable gzip on the web server to compress the code into an even smaller package going over the wire.

Run Rollup

Execute the Rollup process with this command:

```
node_modules/.bin/rollup -c rollup-config.js
```

> Windows users should surround the `rollup` command in double quotes:
>
> "node_modules/.bin/rollup" -c rollup-config.js

Load the Bundle

Loading the generated application bundle does not require a module loader like SystemJS. Remove the scripts that concern SystemJS. Instead, load the bundle file using a single `script` tag *after* the `</body>` tag:

index.html (load bundle)

```
<script src="build.js"></script>
```

Serve the app

You'll need a web server to host the application. Use the same *Lite Server* employed elsewhere in the documentation:

```
npm run lite
```

The server starts, launches a browser, and the app should appear.

AOT QuickStart Source Code

Here's the pertinent source code:

<button (click)="toggleHeading()">Toggle Heading</button> <h1 *ngIf="showHeading">Hello Angular</h1> <h3>List of Heroes</h3> <div *ngFor="let hero of heroes">{{hero}} import { Component } from '@angular/core'; @Component({ moduleId: module.id, selector: 'my-app', templateUrl: './app.component.html' }) export class AppComponent { showHeading = true; heroes = ['Magneta', 'Bombasto', 'Magma', 'Tornado']; toggleHeading() { this.showHeading = !this.showHeading; } } import { platformBrowser } from '@angular/platform-browser'; import { AppModuleNgFactory } from '../aot/src/app/app.module.ngfactory'; console.log('Running AOT compiled'); platformBrowser().bootstrapModuleFactory(AppModuleNgFactory); <!DOCTYPE html> <html> <head> <title>Ahead of time compilation</title> <base href="/"> <meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1"> <link rel="stylesheet" href="styles.css"> <script src="node_modules/core-js/client/shim.min.js"></script> <script src="node_modules/zone.js/dist/zone.js"></script> <script>window.module = 'aot';</script> </head> <body> <my-app>Loading...</my-app> </body> <script src="build.js"></script> </html> { "compilerOptions": { "target": "es5", "module": "es2015", "moduleResolution": "node", "sourceMap": true, "emitDecoratorMetadata": true, "experimentalDecorators": true, "lib": ["es2015", "dom"], "noImplicitAny": true, "suppressImplicitAnyIndexErrors": true }, "files": ["src/app/app.module.ts", "src/main.ts"], "angularCompilerOptions": { "genDir": "aot", "skipMetadataEmit" : true } } import rollup from 'rollup' import nodeResolve from 'rollup-plugin-node-resolve' import commonjs from 'rollup-plugin-commonjs'; import uglify from 'rollup-plugin-uglify' export default { entry: 'src/main.js', dest: 'src/build.js', // output a single application bundle sourceMap: false, format: 'iife', onwarn: function(warning) { // Skip certain warnings // should intercept ... but doesn't in some rollup versions if (warning.code === 'THIS_IS_UNDEFINED') { return; } // intercepts in some

rollup versions if (warning.indexOf("The 'this' keyword is equivalent to 'undefined'") > -1) { return; } // console.warn everything else console.warn(warning.message); }, plugins: [nodeResolve({jsnext: true, module: true}), commonjs({ include: 'node_modules/rxjs/**', }), uglify()] }

Workflow and convenience script

You'll rebuild the AOT version of the application every time you make a change. Those *npm* commands are long and difficult to remember.

Add the following *npm* convenience script to the `package.json` so you can compile and rollup in one command.

package.json (build:aot convenience script)

```
{ "scripts": { "build:aot": "ngc -p tsconfig-aot.json && rollup -c rollup-config.js" } }
```

Open a terminal window and try it.

```
npm run build:aot
```

And JIT too!

AOT compilation and rollup together take several seconds. You may be able to develop iteratively a little faster with SystemJS and JIT. The same source code can be built both ways. Here's one way to do that.

- Make a copy of `index.html` and call it `index-jit.html` .
- Delete the script at the bottom of `index-jit.html` that loads `bundle.js`
- Restore the SystemJS scripts like this:

src/index-jit.html (SystemJS scripts)

```
<script src="node_modules/systemjs/dist/system.src.js"></script> <script src="systemjs.config.js"></script> <script> System.import('main-jit.js').catch(function(err){ console.error(err); }); </script>
```

Notice the slight change to the `system.import` which now specifies `src/app/main-jit` . That's the JIT version of the bootstrap file that we preserved above

Open a *different* terminal window and enter.

```
npm start
```

That compiles the app with JIT and launches the server. The server loads `index.html` which is still the AOT version (confirm in the browser console). Change the address bar to `index-jit.html` and it loads the JIT version (confirm in the browser console).

Develop as usual. The server and TypeScript compiler are in "watch mode" so your changes are reflected immediately in the browser.

To see those changes in AOT, switch to the original terminal and re-run `npm run build:aot` . When it finishes, go back to the browser and back-button to the AOT version in the (default) `index.html` .

Now you can develop JIT and AOT, side-by-side.

Tour of Heroes

The sample above is a trivial variation of the QuickStart app. In this section you apply what you've learned about AOT compilation and Tree Shaking to an app with more substance, the tutorial *Tour of Heroes*.

JIT in development, AOT in production

Today AOT compilation and Tree Shaking take more time than is practical for development. That will change soon. For now, it's best to JIT compile in development and switch to AOT compilation before deploying to production.

Fortunately, the source code can be compiled either way without change *if* you account for a few key differences.

index.html

The JIT and AOT apps require their own `index.html` files because they setup and launch so differently.

Here they are for comparison:

```
<!DOCTYPE html> <html> <head> <base href="/"> <title>Angular Tour of Heroes</title> <meta name="viewport" content="width=device-width, initial-scale=1"> <link rel="stylesheet" href="styles.css"> <script src="shim.min.js"></script> <script src="zone.min.js"></script> <script>window.module = 'aot';
</script> </head> <body> <my-app>Loading...</my-app> </body> <script src="dist/build.js"></script> </html> <!DOCTYPE html> <html> <head> <base href="/"> <title>Angular Tour of Heroes</title> <meta name="viewport" content="width=device-width, initial-scale=1"> <link rel="stylesheet" href="styles.css"> <!-- Polyfills --> <script src="node_modules/core-js/client/shim.min.js"></script> <script src="node_modules/zone.js/dist/zone.js">
</script> <script src="node_modules/systemjs/dist/system.src.js"></script> <script src="systemjs.config.js"></script> <script>
System.import('main.js').catch(function(err){ console.error(err); }); </script> </head> <body> <my-app>Loading...</my-app> </body> </html>
```

The JIT version relies on `SystemJS` to load individual modules. Its scripts appear in its `index.html`.

The AOT version loads the entire application in a single script, `aot/dist/build.js`. It does not need `SystemJS`, so that script is absent from its `index.html`

main.ts

JIT and AOT applications boot in much the same way but require different Angular libraries to do so. The key differences, covered in the Bootstrap section above, are evident in these `main` files which can and should reside in the same folder:

```
import { platformBrowser } from '@angular/platform-browser'; import { AppModuleNgFactory } from '../aot/src/app/app.module.ngfactory';
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory); import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module'; platformBrowserDynamic().bootstrapModule(AppModule);
```

TypeScript configuration

JIT-compiled applications transpile to `commonjs` modules. AOT-compiled applications transpile to *ES2015*/*ES6* modules to facilitate Tree Shaking. AOT requires its own TypeScript configuration settings as well.

You'll need separate TypeScript configuration files such as these:

```
{ "compilerOptions": { "target": "es5", "module": "es2015", "moduleResolution": "node", "sourceMap": true, "emitDecoratorMetadata": true,
"experimentalDecorators": true, "lib": ["es2015", "dom"], "noImplicitAny": true, "suppressImplicitAnyIndexErrors": true, "typeRoots": [
"../../node_modules/@types/" ] }, "files": [ "src/app/app.module.ts", "src/main-aot.ts" ], "angularCompilerOptions": { "genDir": "aot", "skipMetadataEmit" :
true } } { "compilerOptions": { "target": "es5", "module": "commonjs", "moduleResolution": "node", "sourceMap": true, "emitDecoratorMetadata": true,
"experimentalDecorators": true, "lib": [ "es2015", "dom" ], "noImplicitAny": true, "suppressImplicitAnyIndexErrors": true, "typeRoots": [
"../../../node_modules/@types/" ] }, "compileOnSave": true, "exclude": [ "node_modules/*", "**/*-aot.ts" ] }
```

@TYPES AND NODE MODULES

In the file structure of *this particular sample project*, the `node_modules` folder happens to be two levels up from the project root. Therefore, `"typeRoots"` must be set to `"../../node_modules/@types/"`.

In a more typical project, `node_modules` would be a sibling of `tsconfig-aot.json` and `"typeRoots"` would be set to `"node_modules/@types/"`. Edit your `tsconfig-aot.json` to fit your project's file structure.

Tree Shaking

Rollup does the Tree Shaking as before.

rollup-config.js

```
import rollup from 'rollup' import nodeResolve from 'rollup-plugin-node-resolve' import commonjs from 'rollup-plugin-commonjs'; import uglify from 'rollup-plugin-uglify' //paths are relative to the execution path export default { entry: 'src/main-aot.js', dest: 'aot/dist/build.js', // output a single application bundle sourceMap: true, sourceMapFile: 'aot/dist/build.js.map', format: 'iife', onwarn: function(warning) { // Skip certain warnings // should intercept ... but doesn't in some rollup versions if ( warning.code === 'THIS_IS_UNDEFINED' ) { return; } // intercepts in some rollup versions if ( warning.indexOf("The 'this' keyword is equivalent to 'undefined'") > -1 ) { return; } // console.warn everything else console.warn( warning.message ); }, plugins: [ nodeResolve({jsnext: true, module: true}), commonjs({ include: ['node_modules/rxjs/**'] }), uglify() ] }
```

Running the application

The general audience instructions for running the AOT build of the Tour of Heroes app are not ready.

The following instructions presuppose that you have cloned the angular.io github repository and prepared it for development as explained in the repo's README.md.

The *Tour of Heroes* source code is in the `public/docs/_examples/toh-6/ts` folder.

Run the JIT-compiled app with `npm start` as for all other JIT examples.

Compiling with AOT presupposes certain supporting files, most of them discussed above.

```
<!DOCTYPE html> <html> <head> <base href="/"> <title>Angular Tour of Heroes</title> <meta name="viewport" content="width=device-width, initial-scale=1"> <link rel="stylesheet" href="styles.css"> <!-- Polyfills --> <script src="node_modules/core-js/client/shim.min.js"></script> <script src="node_modules/zone.js/dist/zone.js"></script> <script src="node_modules/systemjs/dist/system.src.js"></script> <script src="systemjs.config.js"></script> <script> System.import('main.js').catch(function(err){ console.error(err); }); </script> </head> <body> <my-app>Loading...</my-app> </body> </html> var fs = require('fs'); var resources = [ 'node_modules/core-js/client/shim.min.js', 'node_modules/zone.js/dist/zone.min.js', 'src/styles.css' ]; resources.map(function(f) { var path = f.split('/'); var t = 'aot/' + path[path.length-1]; fs.createReadStream(f).pipe(fs.createWriteStream(t)); }); import rollup from 'rollup' import nodeResolve from 'rollup-plugin-node-resolve' import commonjs from 'rollup-plugin-commonjs'; import uglify from 'rollup-plugin-uglify' //paths are relative to the execution path export default { entry: 'src/main-aot.js', dest: 'aot/dist/build.js', // output a single application bundle sourceMap: true, sourceMapFile: 'aot/dist/build.js.map', format: 'iife', onwarn: function(warning) { // Skip certain warnings // should intercept ... but doesn't in some rollup versions if ( warning.code === 'THIS_IS_UNDEFINED' ) { return; } // intercepts in some rollup versions if ( warning.indexOf("The 'this' keyword is equivalent to 'undefined'") > -1 ) { return; } // console.warn everything else console.warn( warning.message ); }, plugins: [ nodeResolve({jsnext: true, module: true}), commonjs({ include: ['node_modules/rxjs/**'] }), uglify() ] } { "compilerOptions": { "target": "es5", "module": "es2015", "moduleResolution": "node", "sourceMap": true, "emitDecoratorMetadata": true, "experimentalDecorators": true, "lib": ["es2015", "dom"], "noImplicitAny": true, "suppressImplicitAnyIndexErrors": true, "typeRoots": [ "../../node_modules/@types/" ] }, "files": [ "src/app/app.module.ts", "src/main-aot.ts" ], "angularCompilerOptions": { "genDir": "aot", "skipMetadataEmit" : true } }
```

Extend the `scripts` section of the `package.json` with these npm scripts:

package.json (convenience scripts)

```
{ "scripts": { "build:aot": "ngc -p tsconfig-aot.json && rollup -c rollup-config.js", "serve:aot": "lite-server -c bs-config.aot.json" } }
```

Copy the AOT distribution files into the `/aot` folder with the node script:

```
node copy-dist-files
```

You won't do that again until there are updates to `zone.js` or the `core-js` shim for old browsers.

Now AOT-compile the app and launch it with the `lite` server:

```
npm run build:aot && npm run serve:aot
```

Inspect the Bundle

It's fascinating to see what the generated JavaScript bundle looks like after Rollup. The code is minified, so you won't learn much from inspecting the bundle directly. But the source-map-explorer tool can be quite revealing.

Install it:

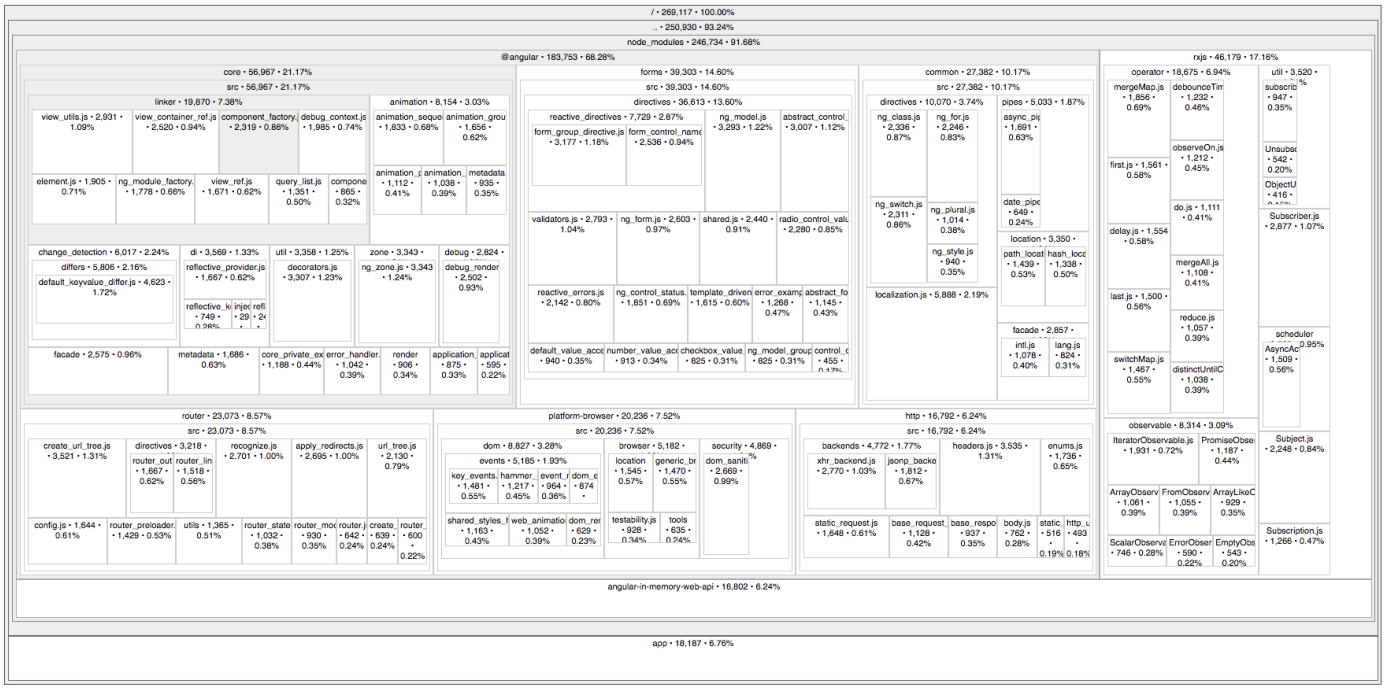
```
npm install source-map-explorer --save-dev
```

Run the following command to generate the map.

```
node_modules/.bin/source-map-explorer aot/dist/build.js
```

The `source-map-explorer` analyzes the source map generated with the bundle and draws a map of all dependencies, showing exactly which application and Angular modules and classes are included in the bundle.

Here's the map for *Tour of Heroes*.



RESOURCES

About

Books & Training