

A Deep, Deep, Deep, Deep, Deep Dive into the Angular Compiler



Uri Shaked [Follow](#)

Jul 7, 2017 · 14 min read

As you know, I love Angular, and all the magical things you can do with it, and I thought it would be an interesting challenge to take a peek into the compiler in Angular 4, try to reverse engineer it, and simulate some part of the compilation process.

Working through the compiler was a great experience, and I turned a lot of what I learned into my talk at ng-conf 2017: DiY Angular Compiler. Since I enjoyed the learning and tinkering process so much, I thought it would be good to share a little of what I learned in blog-form!

So now I present you with “A Deep, Deep, Deep, Deep, Deep Dive into the Angular Compiler!”



And down we go!

As with many of my posts, I think it's better if you can follow along as I go, and so before we get down to business, there are a few things you'll need to have installed on your machine before we get started if you'd like to follow along:

First, you'll need node.js and npm (or yarn) installed on your system.

You also need the latest Angular CLI (version 1.2.0 or newer). To check your Angular CLI version type:

```
ng -v
```

Your result should look like this (or similar):

```
@angular/cli: 1.2.0
```

Otherwise, install the latest Angular CLI:

```
npm i -g @angular/cli
```

We will also use another great tool, source-map-explorer. If you don't have it, you can install it by running:

```
npm i -g source-map-explorer
```

Isolating the Compiler

To begin our deep (deep, deep, deep...) dive into the Angular Compiler, let's create a new project to play with. Go into some directory and type:

```
ng new compiler-playground
```

This will take a few minutes, but you will end up with a new angular project in the compiler-playground directory. Go into that directory and then type:

```
ng build
```

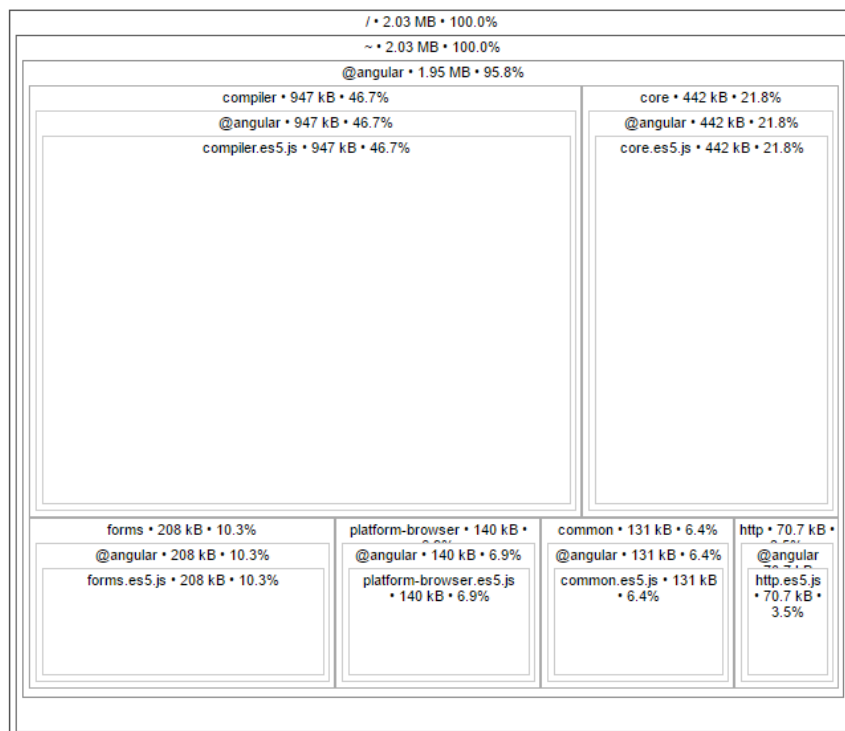
This will create a dist folder with the compiled application. You'll notice though that the size of the resulting JavaScript files is quite big: if we look inside the `dist` folder, we will see a `vendor.bundle.js` file which is about 2 megs in size. This is obviously not ideal!

Taking a peek inside this `vendor.bundle.js` file, we see tons of JavaScript, none of which is minified. We can actually run uglify on this file to get it significantly smaller—about 650kb. But this is still a very big file for just the plain “hello world” app.

This is where `source-map-explorer` comes into play—it allows us to peek into the bundle and find out what makes it big. We can try that by typing:

```
source-map-explorer dist/vendor.bundle.js
```

Wait a few seconds, and we get an output that looks like this:



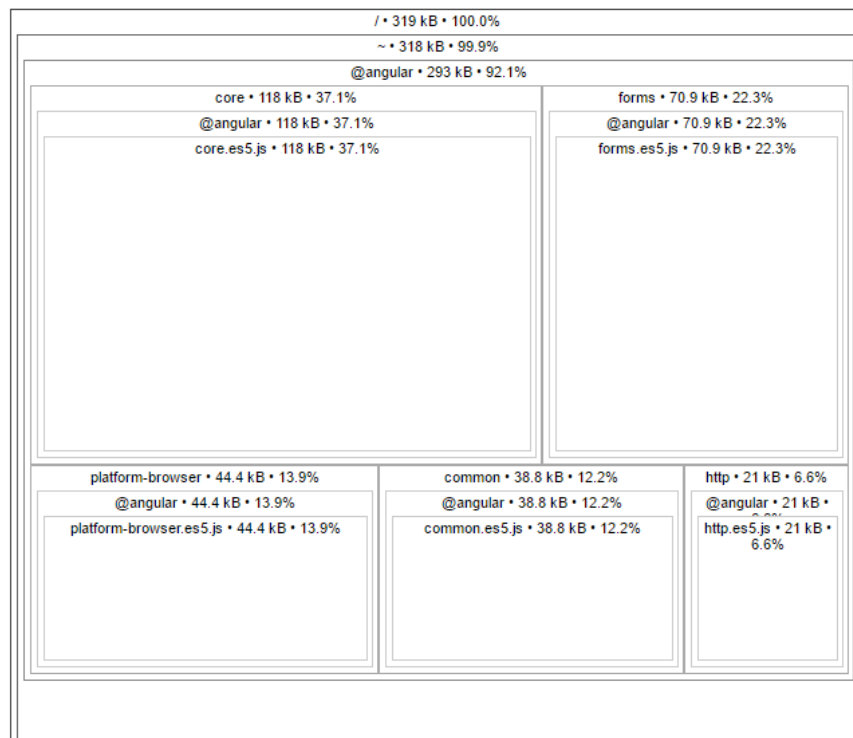
We can see that the “compiler” module accounts for nearly 50% of the bundle size—that is about 1MB (or 320kb when minified) that goes down the wire for every client.

Luckily, it is very easy to get rid of that compiler. Simply run:

```
ng build -prod --sourcemaps
```

and the compiler part will be magically removed, using a feature of angular called AoT (“Ahead of Time” compilation). AoT runs the compile step during the build process instead of inside the browser, so when you build your project for production, the compiler can disappear completely from the output, saving precious CPU cycles when the page loads in the user’s browser.

Now, let’s have a look at the dist directory: the vendor JavaScript file has now been shrunk to 310kb, and using `source-map-explorer` we can see that the big compiler chunk is now gone:



We can also easily shave off another 30% of the bundle size by removing the `forms` and the `http` modules (if we don’t use them)—I hope that in the future the build system will be smart enough to do this for us (the term for removing unused code is “tree-shaking”).

Then, if we actually remove `forms` and `http` (we don't use them) and enable compression, this file gets to be just around 79kb.

Note: the numbers may be slightly different for you, depending on the exact Angular version that you are using and your setup.

So what is this Angular compiler doing there? How comes we can remove it and have the app will still work? Why is it needed at the first place?

To understand the role of the compiler, let's take a tour of some of the inner working of Angular.

Inside Angular: Templates and Views

When we create our templates, we declare what the view should look like. Basically we use HTML language to describe the DOM structure and bind data to it. When your application starts, Angular has to create the DOM tree corresponding to your template, and populate it from data. That is, if you write `<h1>{{title}}</h1>` in your view, Angular has to execute code similar to this (assuming your component controller instance is called `ctrl` in this context):

```
const h1Element = document.createElement('h1');
h1Element.innerText = ctrl.title;
```

In addition, Angular has to monitor the value of the `title` property and update the element whenever that value changes.

In AngularJS (the versions prior to “Angular,” or versions 1.x), the creation of the DOM was delegated to the browser, which parsed your HTML and created the DOM tree (that's its job, after all), and then AngularJS would run over the DOM elements, figure out the directives and text binding expressions and replace them with the actual data (here is the code in AngularJS that actually does this).

This approach introduced several problems.

First of all, browsers can be inconsistent. Different browsers sometimes parse the same HTML input into different DOM structure (example), and Angular has to account for that. Also, browsers are

not very good at dealing with errors—they will often try to cover up for the error by automatically closing elements or moving them around, and even if they do spit an error, they don't tell line numbers. This makes debugging problems much more challenging, usually leading to elimination until we find the error.

In addition, this means that we need a browser just for parsing our templates and rendering them into HTML that can be served to clients and displayed immediately (and also to search engines)—making server-side rendering a complicated and error prone setup (for more see [here](#) or [here](#)).

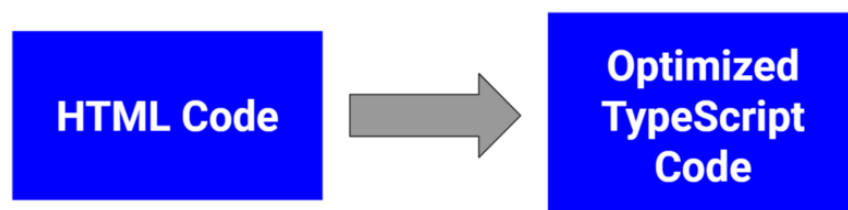
Finally, for some reason, the HTML is case-insensitive when it comes to html tags names and attributes. Not only that, it does not preserve the original case, converting tag names to upper case, and attributes to lower case. You can observe this behavior by running:

```
document.createElement('h1').nodeName
```

And you see that you get uppercase “H1”. This is what led AngularJS to use the famous kebab-case (i.e. `ng-if`, `ng-model`, etc.) in contrast with camelCase, which is a standard in the JavaScript worlds.

So if you use the browser HTML parser, we get different results on different browsers, lacking error information, can't get server rendering and lose attribute case.

That's why we have the compiler. The compiler actually replaces the browser and parses the HTML for you. This gives us consistent parsing across all browsers, and also means it can be run in the server (since it's just a piece of JS code that parses your templates), provide detailed error information, and preserve tag/attribute case. We also get some really awesome tooling, but more on that in a minute.



The magic of the Angular Compiler: transforming your HTML templates into optimized TypeScript code which creates an equivalent DOM structure

The Angular Compiler: Performance, Performance, Performance!

The Angular compiler is an amazing piece of engineering, as we are going to see soon. There is a good reason it is more than 1MB of code, and is the result of more than a year of hard work by the Angular team—not only does it parse the code templates for you, it also creates a highly-performing code, tuned to creating and updating the DOM with minimum CPU and memory overhead.

The goal of adding the compiler was (and still is) to achieve a small memory footprint, quick page load and fast change detection. Here is a link to the research done prior to implementing the compiler in Angular 4: [Generating Less Code](#).

In the meantime, the Angular team is working hard on better tooling and integration with the Closure compiler, a tool that applies aggressive optimizations to JavaScript code, resulting in even smaller bundles and faster execution time. That's what I love about Angular—there is a great, brilliant team behind it that keeps improving the internals all the time, so our apps just get faster and also get better, just like fine wine. And we get to benefit from all this hard work for free!

Now, let's explore the compiler!

Running the compiler

Add the following line inside the “scripts” section of your

`package.json` file:

```
"scripts": {  
  ...,  
  "compile": "ngc"  
}
```

Then run:

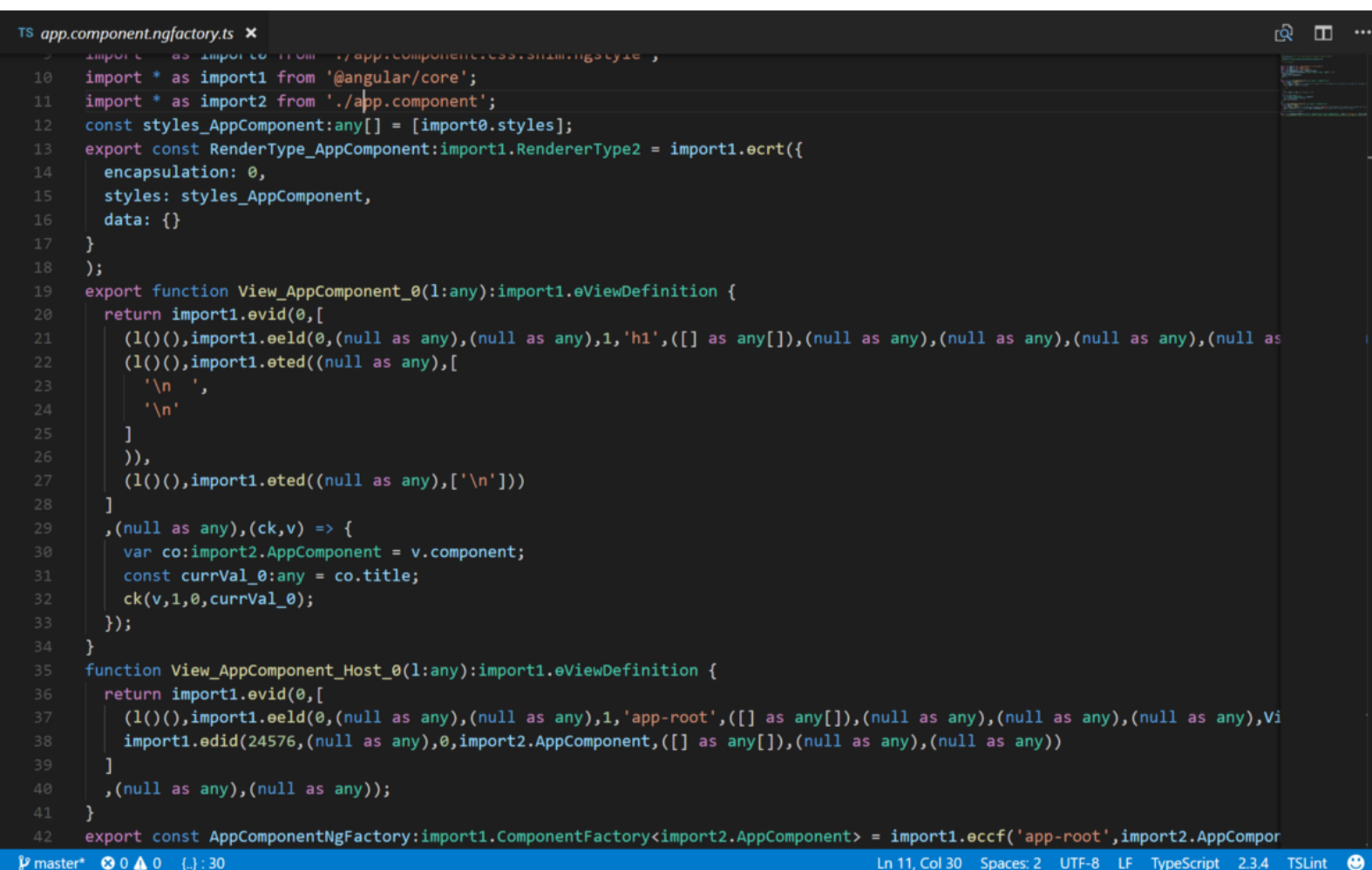
```
npm run compile
```

Wait a few seconds, and you will notice that a lot of files have been created inside your project folder. Your `app.component.html` file has been transformed into `app.component.ngfactory.ts`, your `app.module.ts` resulted in `app.module.ngfactory.ts`, and your CSS files have been turned into shims. We are going to have a look inside each of these now.

Components (View Creation & Change Detection)

🕒: 00:27:00, if you want to follow along :)

The angular compiler transforms our 3-line HTML template into `app.component.ngfactory.ts`. If you look inside this file, you will see a lot of code, which is hard to understand from a quick glance. This code was actually written for machine to read, not for human beings —that's why we need some patience and reverse engineering skills. Fortunately, TypeScript is very helpful here.



```
TS app.component.ngfactory.ts X
import * as import0 from './app.component.css.shim.ngstyle';
10 import * as import1 from '@angular/core';
11 import * as import2 from './app.component';
12 const styles_AppComponent:any[] = [import0.styles];
13 export const RenderType_AppComponent:import1.RendererType2 = import1.e1({
14   encapsulation: 0,
15   styles: styles_AppComponent,
16   data: {}
17 });
18 );
19 export function View_AppComponent_0(l:any):import1.eViewDefinition {
20   return import1.e1(0,[
21     (l())(),import1.e1(0,(null as any),(null as any),1,'h1',([ as any[]]),(null as any),(null as any),(null as any),(null as any),
22     (l())(),import1.e1(0,(null as any),[
23       '\n ',
24       '\n '
25     ])
26   )),
27   (l())(),import1.e1(0,(null as any),['\n']))
28 ];
29 , (null as any),(ck,v) => {
30   var co:import2.AppComponent = v.component;
31   const currVal_0:any = co.title;
32   ck(v,1,0,currVal_0);
33 });
34 }
35 function View_AppComponent_Host_0(l:any):import1.eViewDefinition {
36   return import1.e1(0,[
37     (l())(),import1.e1(0,(null as any),(null as any),1,'app-root',([ as any[]]),(null as any),(null as any),(null as any),Vi
38     import1.e1(24576,(null as any),0,import2.AppComponent,([ as any[]]),(null as any),(null as any))
39   ])
40   , (null as any),(null as any));
41 }
42 export const AppComponentNgFactory:import1.ComponentFactory<import2.AppComponent> = import1.e1('app-root',import2.AppComp
```

Only 3 lines of HTML code result in so much code!

The first thing you may notice is a lot of obscure method names, starting with the letter `ɵ` (Greek Theta) followed by 3 other English letters (e.g. `ɵvid`). The letter `ɵ` is used by the Angular team to indicate that some method is private to the framework and must not be called directly by the user, as the API for these method is not guaranteed to stay stable between Angular versions (in fact, I would say it's almost guaranteed to break).

The reason for using 3 letter shortcut instead of full method name, is simply to save bytes in the final bundle size. But if you Ctrl+click one of those methods (in Visual Studio Code or WebStorm), you will actually see the full method name. For `ɵvid`, that would be `viewDef`, the function that defines a view.

Try changing your view template (`app.component.html`), then run the angular compiler again (`npm run compile`) and see how your changes are reflected in the compiled file. For instance, try to change the template to read:

```
<h1>Hi, {{title + title}}</h1>
```

And see what the compiled look for that looks like.

Basically, most of the magic happens inside the method called `View_AppComponent_ɵ`, which comprises two parts: the top part defines the view—that is, all the elements that are going to be created, their attributes, the text, etc., and the bottom part, does the change detection. This allows Angular to be efficient—the top part runs only once, when the view is created, and only the bottom part is run when Angular performs change detection.

```

export function ViewAppComponent_0(l:any):import1.eViewDefinition {
  return import1.evid(0,[
    (l())(),import1.eeld(0,(null as any),(null as any),1,'h1',([] as any)),
    (l())(),import1.eted((null as any),[
      'Hi, ',
      ''
    ])
  ]),
  (l())(),import1.eted((null as any),['\n']))
],(null as any),(ck,v) => {
  var co:import2.AppComponent = v.component;
  const currVal_0:any = (co.title + co.title);
  ck(v,1,0,currVal_0);
});
}


```

View Creation


Change Detection

The top part runs only once, and only the bottom part is run when Angular performs change detection

Note: I'm going to skip over an explanation of Styles in this article, but if you're curious about how those work, you can check that out in my talk at

: 00:34:18.

Modules

: 00:40:38.

We use modules to organize our applications into components and services. This establishes the context for component resolution and dependency injection: the compiler looks inside the modules to figure out which components are available for other components to use. So unlike AngularJS, pipes and components are not globally available; they are only available in the context of the module that declared them or imported them from another module. This helps preventing naming collisions when building large-scale applications with Angular.

We are going to have a look at how Dependency Injection is implemented by Angular. You would probably imagine that there would be some object or Map that maps each class name or token to the actual implementation. And indeed, AngularJS used objects for this purpose. The disadvantage of using objects is that their indices are always converted to strings, so we were limited to using strings as dependency injection tokens.

With Angular, this is no longer the case—the framework went with a different approach, which allows classes and other objects to be used as dependency injection tokens in addition to strings. So what is it?

When we open the `app.module.ngfactory.ts` file, we can see a very long `getInternal()` method. This is actually how dependency injection is implemented in Angular. My initial thought was that this was done for performance reasons—perhaps a bunch of `if` statement was currently the most efficient way to map between Values in JavaScript?

I asked the Angular team and found out that the main reason for choosing this approach is actually that it allows better dead-code elimination—basically, the Closure compiler can detect unused services this way and remove their implementation from the final bundle.

```
if ((token === import3.HAMMER_GESTURE_CONFIG)) { return this._HAMMER_GESTURE_CONFIG_17; }
if ((token === import3.EVENT_MANAGER_PLUGINS)) { return this._EVENT_MANAGER_PLUGINS_18; }
if ((token === import3.EventManager)) { return this._EventManager_19; }
if ((token === import3.eDomSharedStylesHost)) { return this._eDomSharedStylesHost_20; }
if ((token === import3.eDomRendererFactory2)) { return this._eDomRendererFactory2_21; }
if ((token === import0.RendererFactory2)) { return this._RendererFactory2_22; }
if ((token === import3.eSharedStylesHost)) { return this._eSharedStylesHost_23; }
if ((token === import0.Testability)) { return this._Testability_24; }
if ((token === import3.Meta)) { return this._Meta_25; }
if ((token === import3.Title)) { return this._Title_26; }
return notFoundResult;
}
```

Just a bunch of ``if`` statements


Once a match is found, the relevant `if` statement invokes a getter, which will create the service instance when run for the first time, otherwise it will return the instance previously created. So basically, dependency injection is just a bunch of `if` statements.

```
get __Compiler_11(): import0.Compiler {
  if ((this.__Compiler_11 == null)) {
    (this.__Compiler_11 = new import0.Compiler());
  }
  return this.__Compiler_11;
}
```

Example: the getter for the Compiler service, creating the instance on-demand

In case that some service depends on another service, this fact is already known at compile time, so we can look for the relevant service before we instantiate it, and pass it as a parameter to the constructor:

```
get __Testability_24(): import0.Testability {  
  if ((this.__Testability_24 == null)) {  
    (this.__Testability_24 =  
      new import0.Testability(this.parent.get(import0.NgZone)));  
  }  
  return this.__Testability_24;  
}
```



The Testability service depends on NgZone, so an instance of it will be created and passed to the Testability service constructor

If you want to keep diving in, another great resource for learning about the compiler is Tobias Bosch's ng-conf 2017 talk about the Angular 4.0 Compiler. He did a lot of the compiler engineering work, so surely he knows how it works best :-)

Some Fun with Tooling

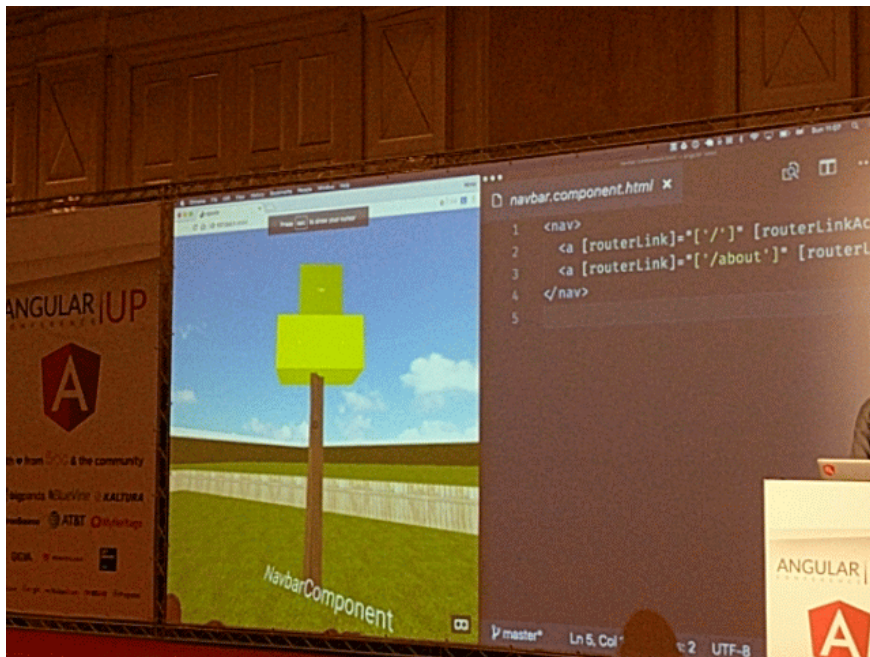
Although we could go on for pages and pages about the possibilities for tooling that you could build around the Angular Compiler, I'd like to call out one such tool in particular: Language Services.

The Angular Language Services allow you to run the compiler inside your favorite IDE (WebStorm, Visual Studio Code), etc., and get benefits such as auto complete and detailed errors while editing the templates. If you haven't used them you definitely should: they will make you a much more efficient Angular developer. If you use VSCode—here is the extension.

Minko Gechev also mentioned some very cool use cases in his ng-conf 2017 talk: Mad Science with the Angular Compiler. In addition to building tools on top of the compiler to automatically migrate between angular versions and visualize app structure, at some point he even builds a 3d-model of the app with all the components rendered as... trees!



Turning your Angular App into a Virtual Reality World



Tree-shaking. Literally!

Playtime: Do It Yourself!

We have just scratched the surface of the Angular Compiler in this post, and there is much more to explore. I'm going to leave you with 3 "do it yourself" exercises that will help you get a good feeling of the inner working of the compiler if you're more of a hands-on-learner. In each exercise, you're basically going to manually try and perform transformations that the compiler do.

Before we start, let's switch the code to consume the compiled code, so you will be able to modify it and see the results.

First of all, run `ng serve` and verify that the app works (<http://localhost:4200>), because after modifying the app entry point, the angular webpack plugin will spit an error (it just happens on webpack init). We can work around this by running `ng eject` and reconfiguring webpack to use the plain typescript plugin instead, but that's not the point here.

After you got the app running, modify `src/main.ts` to import `AppModuleNgFactory` and call `bootstrapModuleFactory` (🐛: 00:52:45). The result should look like:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModuleNgFactory } from
'./app/app.module.ngfactory';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModuleFactory(AppModuleNgFactory);
```

That's it! Angular is now running your compiled code. To verify this, simply modify your `app.component.html` file (e.g. add some text), the app will reload but you will not see your new changes, since you are using the compiled version directly, and the template is no longer being compiled for you in the browser (🐛: 00:54:15). You can also double-verify by making changes to your component factory (🐛: 00:55:10).

When doing these exercises, edit the `.ngfactory` files directly. Don't edit the HTML file and run the compiler—that would be cheating :-)

I recommend taking advantage of the typings (hover/ctrl-click or cmd-click on the different functions called from the compiled files to see their definitions), as this will help you understand what you see there much more quickly.

Exercise 1 — Uppercase Title

Modify the component factory to display the title in uppercase (e.g. APP WORKS!).

Bonus: Display another copy of the title below the heading, this time without uppercase. E.g. the HTML code that will be rendered in the browser would be

```
<h1>APP WORKS!</h1>
app works!
```

Solution 🐼: 1:02:50.

Exercise 2 — Dependency Injection

Create a new Emoji service by running the following CLI command:

```
ng generate service emoji
```

Then, add the following line inside the `emoji-service.ts` file, just above the `constructor() {}` :

```
cat = '🐱';
```

Finally, change the constructor of `app.component.ts` to inject and use this service:

```
constructor(emoji: EmojiService) {
  this.title += emoji.cat;
}
```

(don't forget to import the `EmojiService` class at the beginning of the file).

This will obviously not work— `emoji` will get an undefined value in the component. You'll need to find a way to modify the compiled files

in a way that registers the service as a dependency of the component and also provides it in your module's dependency injection.

Hints:

1. Add the service to the list of component dependencies in the directive definition (`ɵdid`) of the app component (in `app.component.factory.ts` , of course).
2. Add the service to `getInternal()` method inside `app.module.ngfactory.ts`

Solution 🐞: 1:30:05.

Exercise 3 — ngOnInit

Add an `ngOnInit()` method to `AppComponent` :

```
ngOnInit() {  
  this.title = 'onInit was run!';  
}
```

Why doesn't Angular run it? How can we fix it?

Hints:


1. Have a look at the view flags (first argument to `ɵdid` call in the component factory). The available flags are defined here.
2. Add component to change detection cycle (provide a view update function as the 3rd argument to `ɵdid` call inside `View_AppComponent_Host_0` , similar to the function passed to `ɵdid` inside `View_AppComponent_0`).
3. If you are not familiar with bit-wise operations in JavaScript, or just need a few more hints check out 🐞: 1:35:50.

Solution 🐞: 1:49:00.

Takeaways

The angular compiler is an amazing piece of engineering. I hope that this post gave you the opportunity to explore it and understand how it actually works. This is just a small bit—there is much more to

explore, and now you have the knowledge and tools for running the compiler, examining its output and figuring out all the magic that it does.

Thanks to the Angular team for always pushing the limits of what Angular can do and improving performance, and Tobias Bosch and **Igor Minar** for answering my many questions while I was trying to figure out this compiler masterpiece. Special thanks to **Pascal Precht**  for reviewing and offering his feedback on this post.

