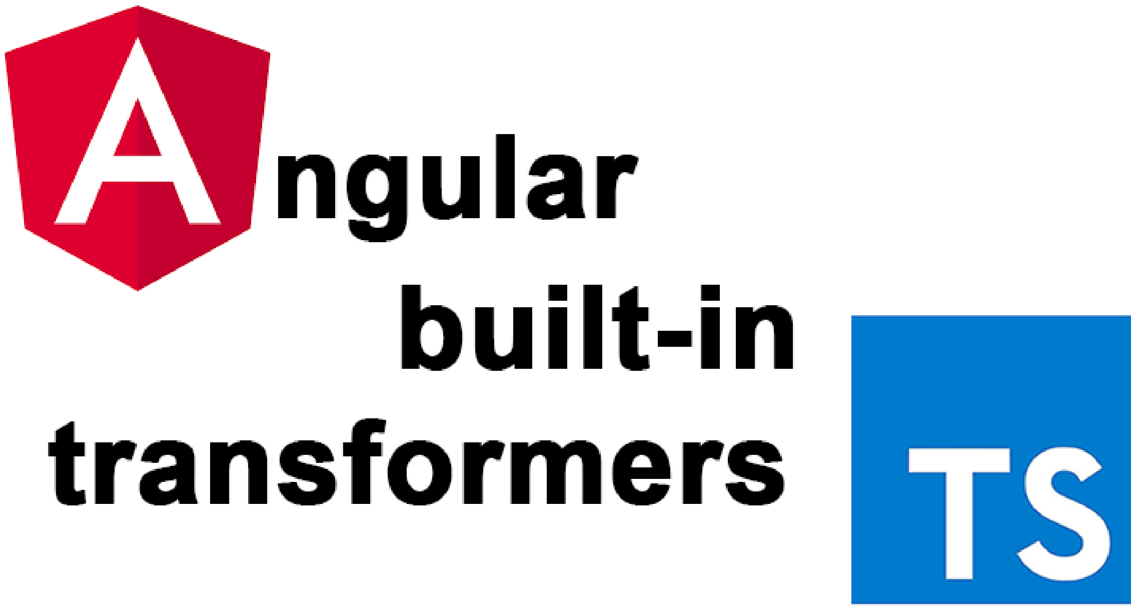
Do you know how Angular transforms your code?

A quick overview of Angular 7 typescript transformers

Alexey Zuev Follow
Jan 23 · 5 min read



Typescript added support for custom transformers in 2.3 version. This type of extensibility allowed developers to go beyond the scope of the base compiler. After that, almost everyone, who had a project with typescript compilation, wanted to turn his ideas into reality. Angular also could not stand aside.

The first pull request on switching to transformer based compilation was born on 10 Jun 2017. A lot of time has passed since that day and now Angular can't live without these custom transformers.

In this article, I will briefly explore many of the Angular transformers that are mostly required for AOT. I divide them into two types:

transformers from the **@angular/compiler-cli** package and **@angular/cli** specific transformers.

> *I use Angular 7.2.1 version here*

. . .

Angular compiler-cli transformers

Once we run the **ngc** command we will send our code to Angular wrapper over typescript Program that can make the following transformations:

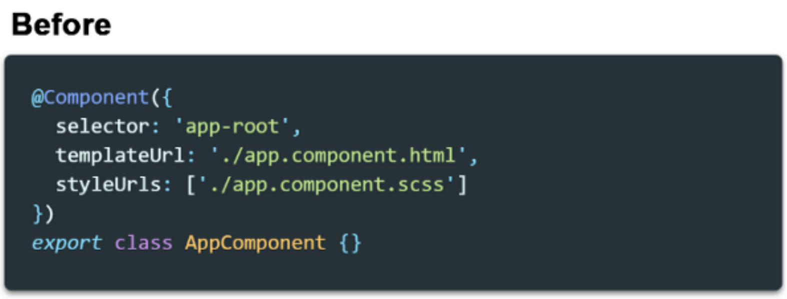
# Inline resource

You must know this transformer if you're a library author.

It replaces `templateUrl` / `styleUrls` properties in `@Component` with `template` / `styles` respectively.

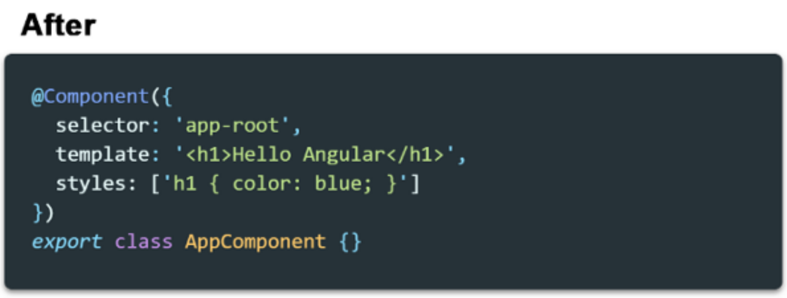
You can enable this transformer via `tsconfig.json` :

```
angularCompilerOptions {
  enableResourceInlining: true
}
```



Before

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



After

```
@Component({
  selector: 'app-root',
  template: '<h1>Hello Angular</h1>',
  styles: ['h1 { color: blue; }']
})
export class AppComponent {}
```

> **Source**: *@angular/compiler-cli/src/transformers/inline_resources.js*

# Lower expressions

I believe we all met such kind of errors:

> *Error: Error encountered resolving symbol values statically. Function calls are not supported. Consider replacing the function or lambda with a reference to an exported function.*

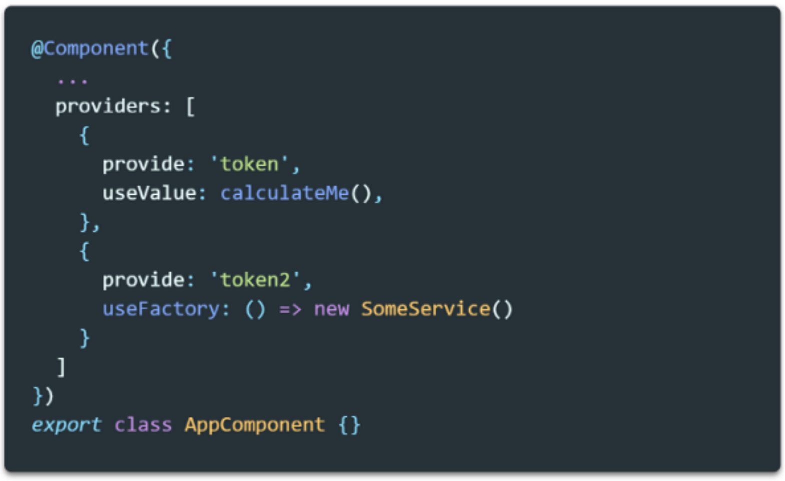
when were writing metadata like:

```
providers: [{provide: Token, useFactory: () => new
SomeClass()}]
```

Lower expressions transformer will rewrite `() => new SomeClass()` expression to a variable exported from the module allowing the compiler to import the variable without needing to understand the expression.

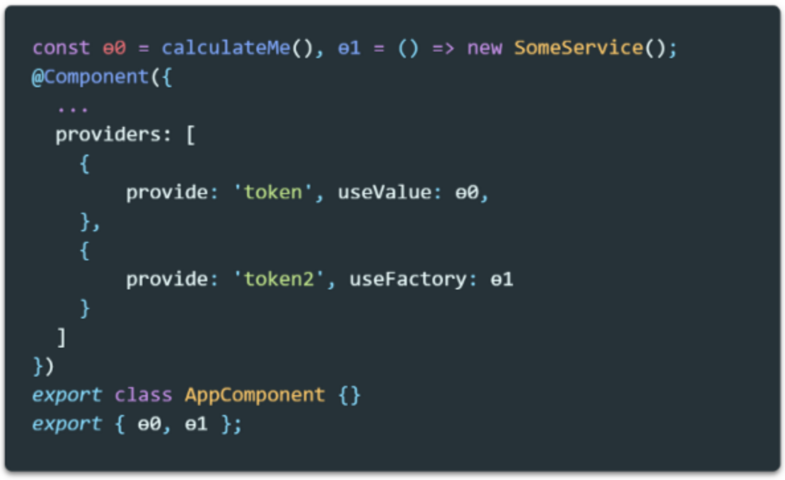
The transformation only works for a strictly defined set of fields: **useValue**, **useFactory**, **data**, **id** and **loadChildren**.

Before



```
@Component({
  ...
  providers: [
    {
      provide: 'token',
      useValue: calculateMe(),
    },
    {
      provide: 'token2',
      useFactory: () => new SomeService()
    }
  ]
})
export class AppComponent {}
```

After



```
const ɵ0 = calculateMe(), ɵ1 = () => new SomeService();
@Component({
  ...
  providers: [
    {
        provide: 'token', useValue: ɵ0,
    },
    {
        provide: 'token2', useFactory: ɵ1
    }
  ]
})
export class AppComponent {}
export { ɵ0, ɵ1 };
```

For a detailed explanation on this, please refer to the documentation.

Lower expression transformer is enabled by default. To disable it use the disableExpressionLowering option:

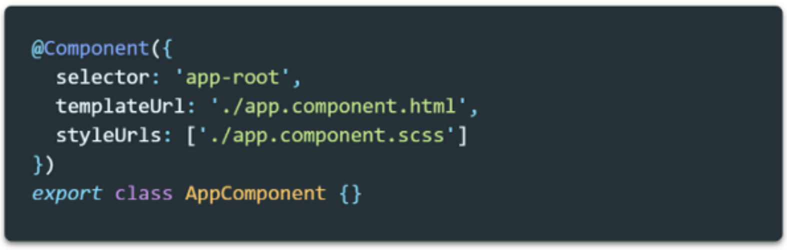
```
angularCompilerOptions {
  disableExpressionLowering: true
}
```

> **Source**: *@angular/compiler-cli/src/transformers/lower_expressions.js*

# Node emitter transformer

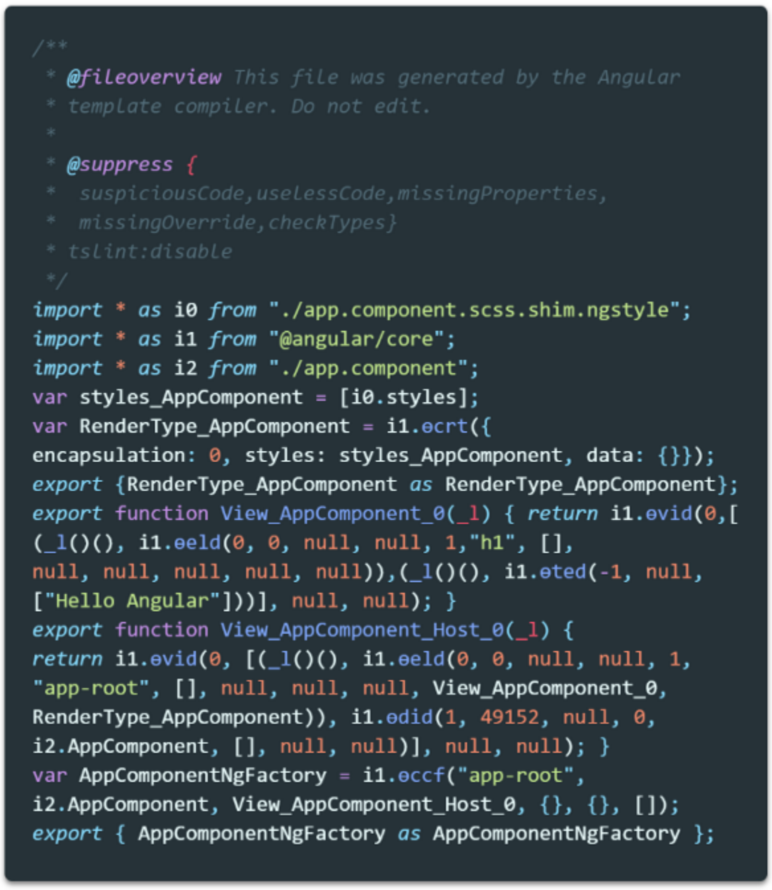
This is the main ngc transformer which takes generated by AOT compiler SourceFiles and adds a file overview JSDoc comment containing Closure Compiler specific “suppress”ions in JSDoc.

Before



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

After



```
/**
 * @fileoverview This file was generated by the Angular
 * template compiler. Do not edit.
 *
 * @suppress {
 *  suspiciousCode,uselessCode,missingProperties,
 *  missingOverride,checkTypes}
 * tslint:disable
 */
import * as i0 from "./app.component.scss.shim.ngstyle";
import * as i1 from "@angular/core";
import * as i2 from "./app.component";
var styles_AppComponent = [i0.styles];
var RenderType_AppComponent = i1.ɵcrt({
encapsulation: 0, styles: styles_AppComponent, data: {}});
export {RenderType_AppComponent as RenderType_AppComponent};
export function View_AppComponent_0(_l) { return i1.ɵvid(0,[
(_l()(), i1.ɵeld(0, 0, null, null, 1,"h1", [],
null, null, null, null, null)),(_l()(), i1.ɵted(-1, null,
["Hello Angular"]))], null, null); }
export function View_AppComponent_Host_0(_l) {
return i1.ɵvid(0, [(_l()(), i1.ɵeld(0, 0, null, null, 1,
"app-root", [], null, null, null, View_AppComponent_0,
RenderType_AppComponent)), i1.ɵdid(1, 49152, null, 0,
i2.AppComponent, [], null, null)], null, null); }
var AppComponentNgFactory = i1.ɵccf("app-root",
i2.AppComponent, View_AppComponent_Host_0, {}, {}, []);
export { AppComponentNgFactory as AppComponentNgFactory };
```

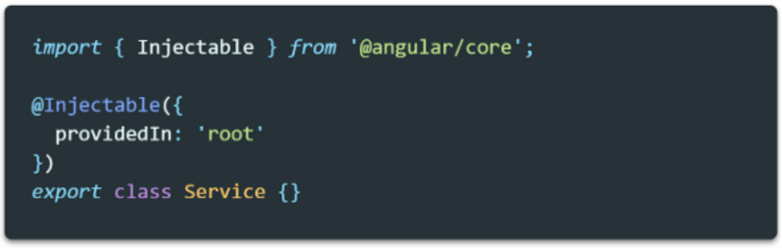
> **Source**: *@angular/compiler-cli/src/transformers/node_emitter_transform.js*

# Angular class transformer

It adds the requested static methods specified by partial modules.

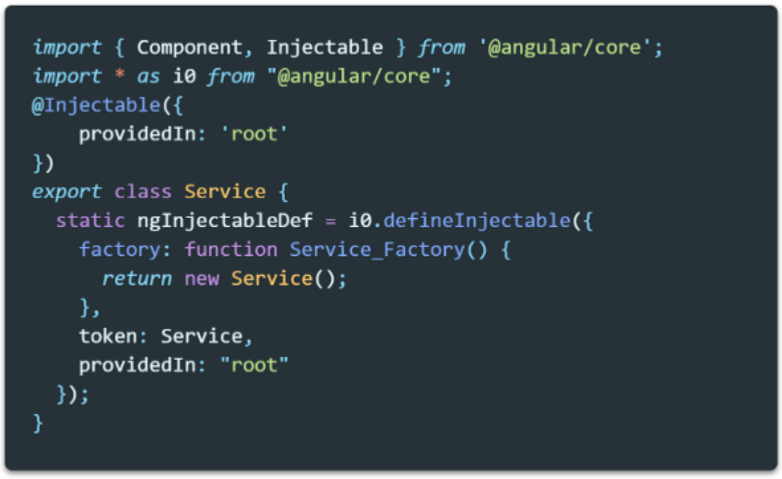
This transformer uses PartialModules system introduced in Ivy renderer. Render2 uses this transformer to convert `Injectable` 's to static `ngInjectableDef` fields.

Before



```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class Service {}
```

After



```
import { Component, Injectable } from '@angular/core';
import * as i0 from "@angular/core";
@Injectable({
    providedIn: 'root'
})
export class Service {
  static ngInjectableDef = i0.defineInjectable({
    factory: function Service_Factory() {
      return new Service();
    },
    token: Service,
    providedIn: "root"
  });
}
```

> **Source**: *@angular/compiler-cli/src/transformers/r3_transform.js*

# DecoratorStripTransformer (ivy only)

Removes decorators like:

```
['Component', 'Directive', 'Injectable', 'NgModule', 'Pipe',
];
```

since they were "reified" to `ngComponentDef` , `ngDirectiveDef` , `ngInjectableDef` etc.

Used in legacy ngtsc mode(enableIvy=true)

> **Source**: *@angular/compiler-cli/src/transformers/r3_strip_decorators.js*

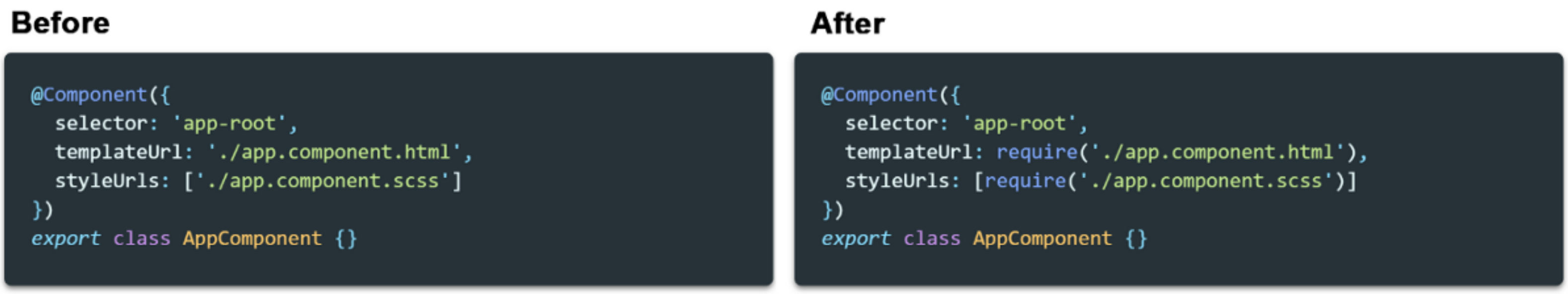
. . .

Cli ngtools/webpack transformers

ng cli command also hides a bunch of transformers:

# Replace resources (Jit only)

Replaces resources with webpack's `require` version so that all resources will be included at runtime.



Before

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

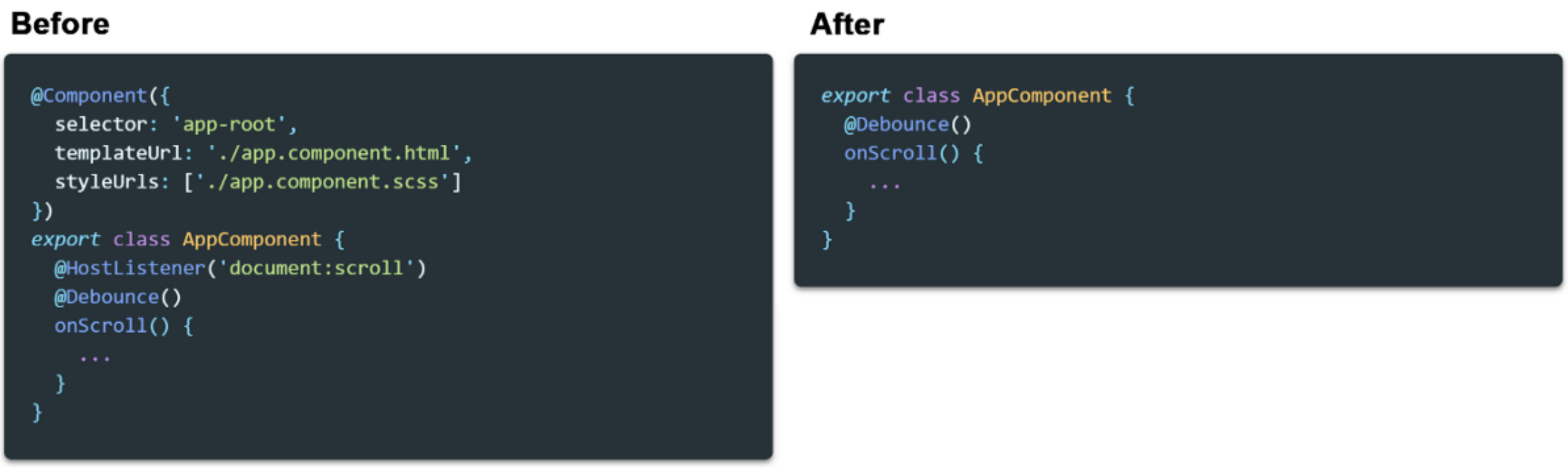
After

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

> **Source**:
> *@ngtools/webpack/src/transformers/replace_resources.js*

# Remove decorators

Removes all decorators originated from `@angular/core` module.



Before

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  @HostListener('document:scroll')
  @Debounce()
  onScroll() {
    ...
  }
}
```

After

```
export class AppComponent {
  @Debounce()
  onScroll() {
    ...
  }
}
```

There is an interesting fact. If we import decorator from a different module then it will be survived.

proxy-core-decorators.ts

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

module-jit.ts

```
import { Component } from './proxy-core-decorators'
```

```
@Component({ // won't be removed
  ...
})
export class SomeComponent {}
```

So you can use that fact to make some trick to save angular decorators for Jit compilation.

> **Source**:
> *@ngtools/webpack/src/transformers/remove_decorators.js*

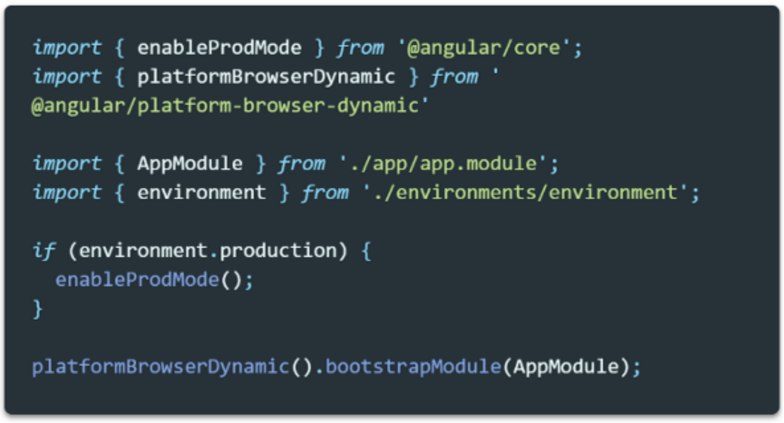
# Register Locale Data (browser only)

Imports locales to the main entry point.

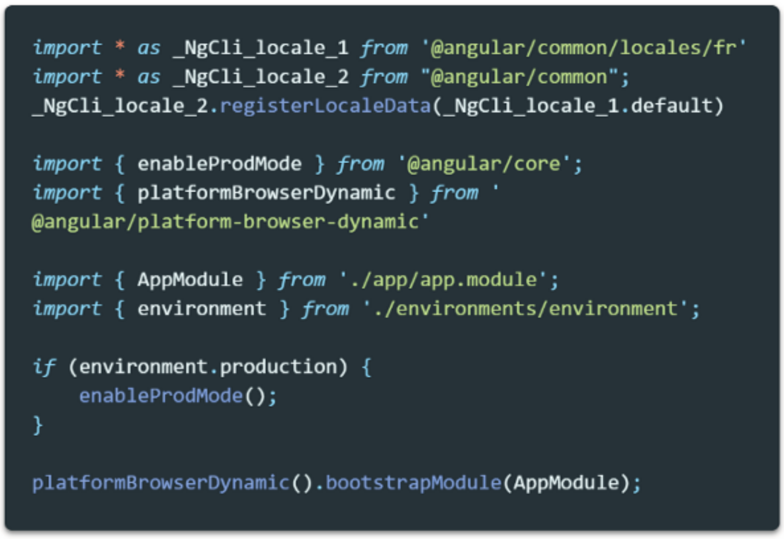
Basically, it works when we provide locale to cli parameters like

```
--locale=fr
```

Before


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

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

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

After

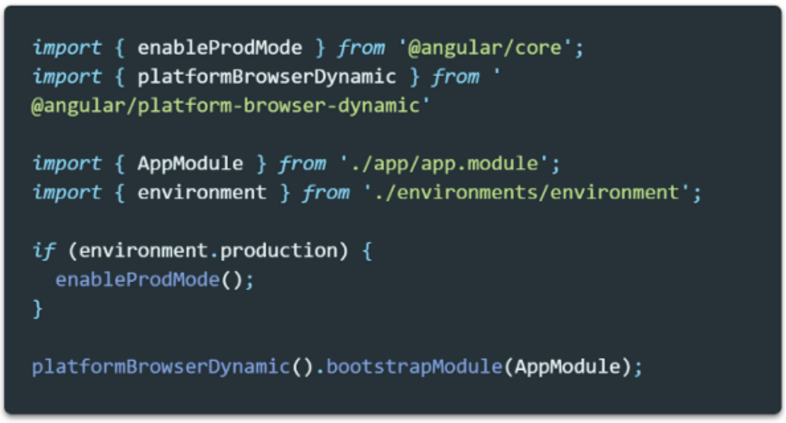

```
import * as _NgCli_locale_1 from '@angular/common/locales/fr'
import * as _NgCli_locale_2 from "@angular/common";
_NgCli_locale_2.registerLocaleData(_NgCli_locale_1.default)

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

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

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

> **Source**:
> *@ngtools/webpack/src/transformers/register_locale_data.ts*

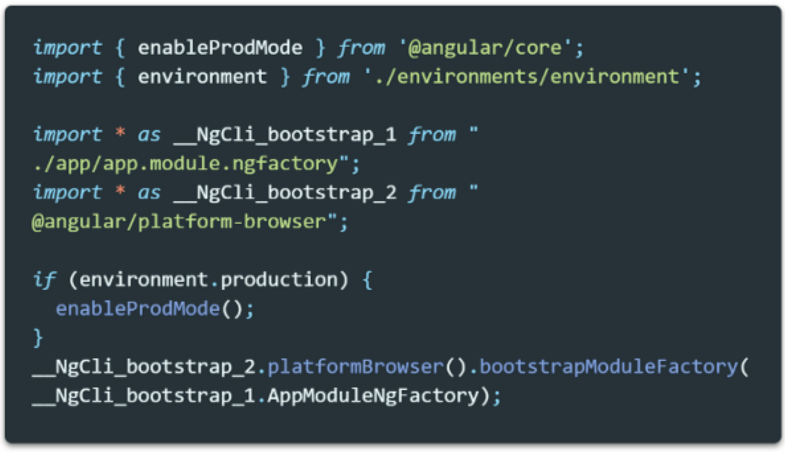
# Replace bootstrap(aot only)

Replaces `platformBrowserDynamic().bootstrapModule(AppModule)` with `platformBrowser().bootstrapModuleFactory(AppModuleNgFactory)`

Before


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

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

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

After


```
import { enableProdMode } from '@angular/core';
import { environment } from './environments/environment';

import * as __NgCli_bootstrap_1 from "
./app/app.module.ngfactory";
import * as __NgCli_bootstrap_2 from "
@angular/platform-browser";

if (environment.production) {
  enableProdMode();
}
__NgCli_bootstrap_2.platformBrowser().bootstrapModuleFactory(
__NgCli_bootstrap_1.AppModuleNgFactory);
```

Gotchas:

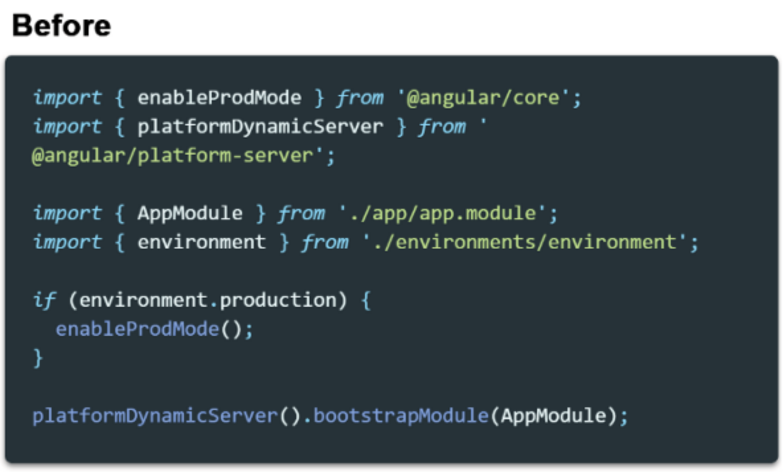
We have to only write `platformBrowserDynamic().bootstrapModule(AppModule);` once otherwise it won't be replaced. Also, we can't split this call in two statements like:

```
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

> **Source**:
> *@ngtools/webpack/src/transformers/replace_bootstrap.ts*

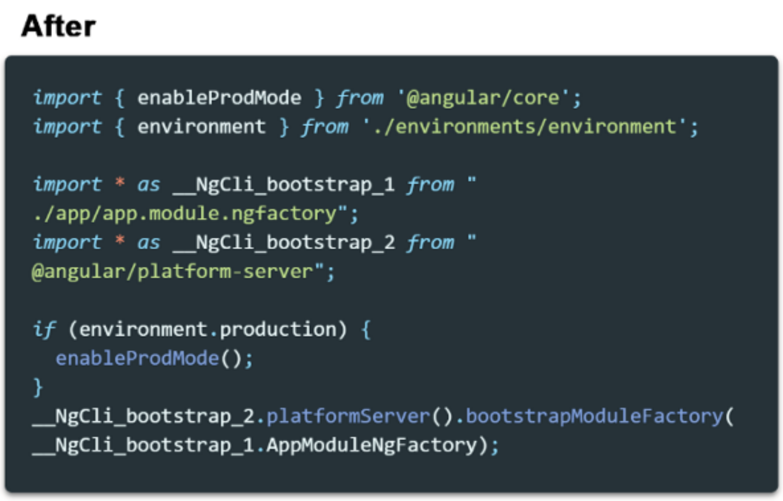
# Replace server bootstrap (server aot only)

The same as above but for server.



Before

```
import { enableProdMode } from '@angular/core';
import { platformDynamicServer } from '
@angular/platform-server';

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

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

platformDynamicServer().bootstrapModule(AppModule);
```



After

```
import { enableProdMode } from '@angular/core';
import { environment } from './environments/environment';

import * as __NgCli_bootstrap_1 from "
./app/app.module.ngfactory";
import * as __NgCli_bootstrap_2 from "
@angular/platform-server";

if (environment.production) {
  enableProdMode();
}
__NgCli_bootstrap_2.platformServer().bootstrapModuleFactory(
__NgCli_bootstrap_1.AppModuleNgFactory);
```

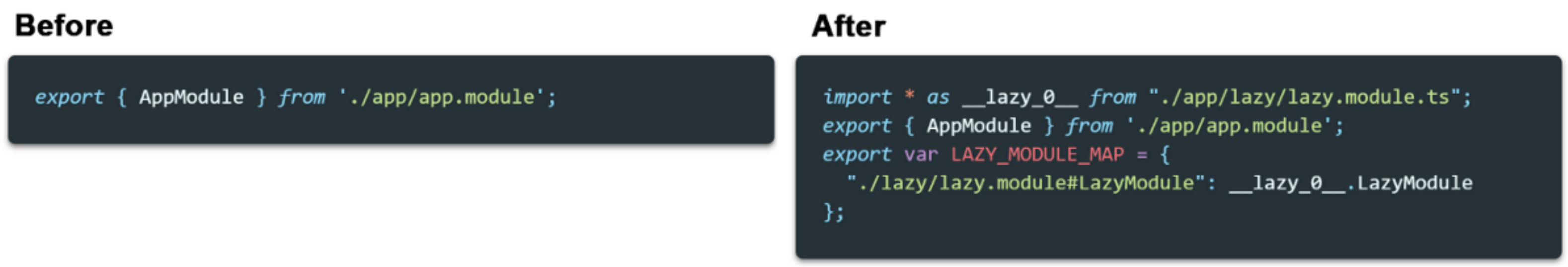
> **Source**:
> *@ngtools/webpack/src/transformers/replace_server_bootstrap.ts*

# Export lazy module map (server only)

If you’re familiar with Angular SSR then you may come across the following code in the **server.ts** file:

```
// * NOTE :: leave this as require() since this file is built
Dynamically from webpack
const {LAZY_MODULE_MAP} = require('./dist/server/main');
```

So, this transformer is responsible for making this working. It is used to support that import.



Before

```
export { AppModule } from './app/app.module';
```

After

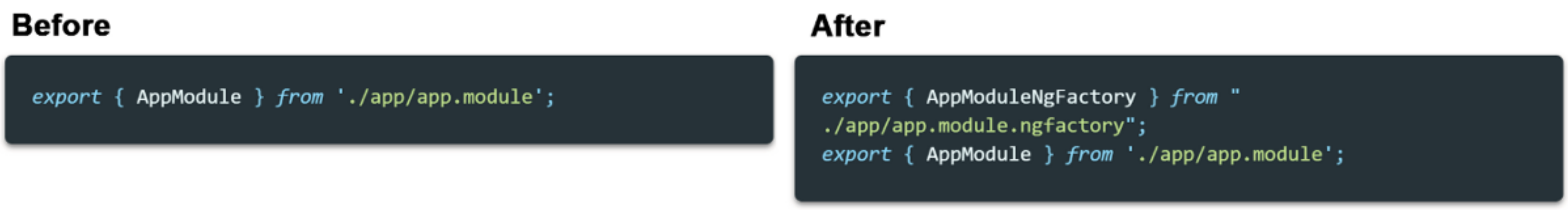
```
import * as __lazy_0__ from "./app/lazy/lazy.module.ts";
export { AppModule } from './app/app.module';
export var LAZY_MODULE_MAP = {
  "./lazy/lazy.module#LazyModule": __lazy_0__.LazyModule
};
```

> **Source**: *@ngtools/webpack/src/transformers/export_lazy_module_map.ts*

# Export ngfactory(server aot only)

Also is used to support import in **server.ts** file.

```
// * NOTE :: leave this as require() since this file is built
Dynamically from webpack
const {AppServerModuleNgFactory, LAZY_MODULE_MAP} =
require('./dist/server/main');
```



Before

```
export { AppModule } from './app/app.module';
```

After

```
export { AppModuleNgFactory } from "
./app/app.module.ngfactory";
export { AppModule } from './app/app.module';
```

> **Source**: *@ngtools/webpack/src/transformers/export_ngfactory.ts*

# PlatformTransformers (public API)

Gives us the ability to provide our own custom transformer.

For example, here is how `native-script` applies its own version of bootstrap replacement:

webpack.config.js

```
new AngularCompilerPlugin({
    platformTransformers: aot ? [nsReplaceBootstrap(() =>
ngCompilerPlugin)] : null,
```

Summary

Angular widely uses typescript custom transformers. It might be useless to know what they do if you are not interested in Angular internals. But, on the other hand, that knowledge can save your time when you're stuck with the build.