Automatically upgrade lazy-loaded Angular modules for Ivy!

Craig Spence Follow
Mar 11 · 7 min read

Lazy-loading in Angular! 💾

If you've ever created a lazy-loaded module in an Angular app, then the following code might look pretty familiar to you:

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

@NgModule({
    imports: [
        RouterModule.forChild([{
            path: '',
            loadChildren: './lazy/lazy.module#LazyModu
        }])
```

Magical configuration for a lazy-loaded module in an Angular 7 app

It does the job, but it's fairly magical, since it relies on a special string syntax, and some compiler wizardry in the Angular CLI...

Luckily, web standards have evolved since this syntax was introduced, and there's now a "better" way to split our app and load each parts on demand!

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

@NgModule({
    imports: [
        RouterModule.forChild([{
            path: '',
            loadChildren: () => import('./lazy/lazy.mo
        }])
```

Less magical configuration for a lazy-loaded module in an Angular 8 app

Thanks to the changes coming with Ivy in Angular 8, we will soon be able to use the `import()` operator to fetch a module as we navigate around our application.

• • •

Want it right now? Cool! 🎉

If you're already playing with Ivy, you can install a **TSLint** rule with a fixer to upgrading this automatically:

```
npm install @phenomnomnominal/angular-lazy-routes-fix -D
```

Add the following to your `tslint.json`:

```
{
    "extends": [
        "@phenomnomnominal/angular-lazy-routes-fix"
    ],
    "//": "either",
    "no-lazy-module-paths": [true],
    "//": "or",
    "no-lazy-module-paths": [true, "async"]
}
```

And then run:

```
ng lint --fix
```

Voilà! All your lazy-loaded routes should be upgraded! 🎉

> ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ ⚠️ *The upgraded code will only work with if the Ivy renderer is enabled, or if your app is running in JIT mode.*

A picture of a cute little sloth.

What stopped us from doing this up until now?

> *This next section goes pretty deep into how lazy-loading works in Angular right now, and how it's going to work in Angular in the future! Most of this is going away, but it's still pretty interesting!*

Lazy-loaded routes in Angular 7.x.x

We use `RouterModule.forChild()` and `RouterModule.forRoot()` to tell Angular about the route structure of our application. But how does it work? Let's check out the Angular source and find out!

If we dig into the implementation of `RouterModule.forChild()` and `RouterModule.forRoot()`, we can see that when we pass in the array

of routes, they are registered as a multi provider against the `ROUTES` `InjectionToken`:

```
@NgModule({declarations: ROUTER_DIRECTIVES, exports: R
export class RouterModule {
  // ...

  static forRoot(routes: Routes, config?: ExtraOptions
    return {
      ngModule: RouterModule,
      providers: [
        // ...
        provideRoutes(routes),
        // ...
      ],
    };
  }

  static forChild(routes: Routes): ModuleWithProviders
    return {ngModule: RouterModule, providers: [provid
```

The RouterModule (from the Angular 7.x.x source) takes the given route configuration and attaches them to the **ROUTES InjectionToken.**

This means that at runtime we're going to have an injectable array of route configuration objects! But how does Angular use these `ROUTES`? Again, let's use the source:

```
export const ROUTES = new InjectionToken<Route[][]>('R

export class RouterConfigLoader {
  // ...
  load(parentInjector: Injector, route: Route): Observ
    // ...
    const moduleFactory$ = this.loadModuleFactory(rout
    // ...
  }

  private loadModuleFactory(loadChildren: LoadChildren
    if (typeof loadChildren === 'string') {
      return from(this.loader.load(loadChildren));
    } else {
      return wrapIntoObservable(loadChildren()).pipe(m
        if (t instanceof NgModuleFactory) {
```

The RouterConfigLoader (from the Angular 7.x.x source) is responsible for figuring out how to load a module based on its config.

The `ROUTES` are injected into the application's `Router` when it is created. When Angular encounters a route with a `loadChildren` property on it, it uses the `RouterConfigLoader` to try and figure out how to do that loading. We can see that the `RouterConfigLoader` does something differently based on if `typeof loadChildren` is a `string` or not... but doesn't `loadChildren` *have* to be a string?

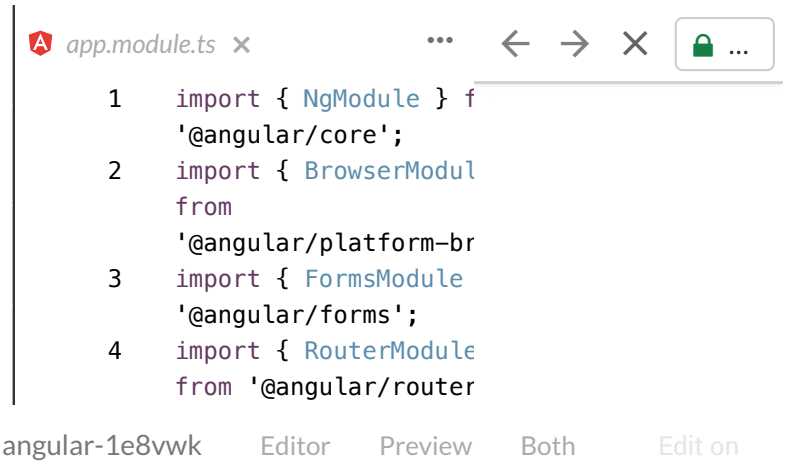
Let's have a look at the `LoadChildren` type:

```
/**
 * A function that is called to resolve a collection c
 */
export type LoadChildrenCallback = () => Type<any>| Ng

/**
 * A string of the form `path/to/file#exportName` that
```

TypeScript types for **LoadChildren**, showing that you can use either a **string** or an **async function**.

Isn't that interesting! Even in a pre-Ivy world, `loadChildren` can be a `string` or an `async function`! So that should mean that our fancy `import()` syntax will already work? Let's try it out:



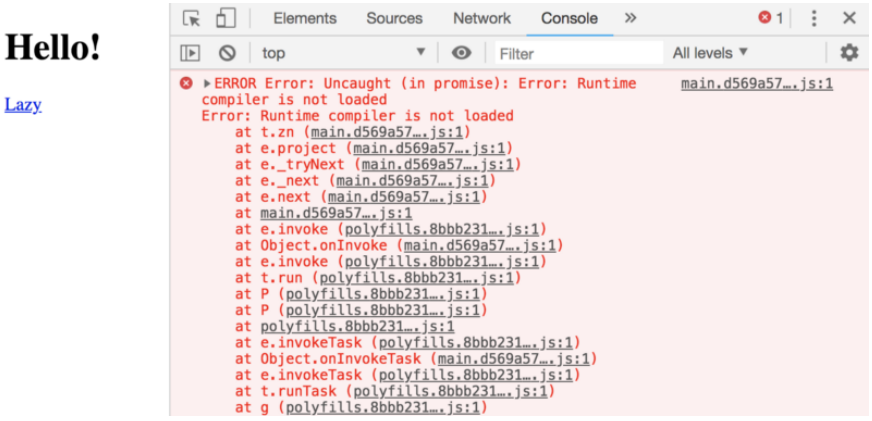
StackBlitz showing the alternative **loadChildren** syntax working in Angular 7.x.x

What? It does work! But how does this work?! Why have we been using the magic string syntax all along?!?!

The answer is **there's a catch...** 🎣

Lazy-loaded routes in Angular 7.x.x with Ahead of Time compilation

If we were to take our above application and build it with the prod flag (`ng build --prod`), everything appears to work! But when we try to navigate to our lazy-loaded route, we get a big red error:



When running our app in AOT mode, we get an error: Runtime compiler is not loaded

This error makes sense! We used the `--prod` flag to enable the "Ahead-of-time" **(AOT)** compiler, which means we opted out of the "Just-in-time" **(JIT)** runtime compiler. If we look at where the error

comes from, we can see it's caused by the call to `compileModuleAsync()` in the **`RouterConfigLoader`**:

```
export class RouterConfigLoader {
  // ...
  private loadModuleFactory(loadChildren: LoadChildren
    if (typeof loadChildren === 'string') {
      // ...
    } else {
      return wrapIntoObservable(loadChildren()).pipe(m
        if (t instanceof NgModuleFactory) {
          return of (t);
        } else {
          return from(this.compiler.compileModuleAsync
```

RouterConfigLoader#loadModuleFactory throws in AOT mode because it doesn't have access to the JIT compiler at run time.

We end up down that **`else`** path because the **`instanceof`** check fails! When we use the **`import()`** operator with AOT, the object that we import from the lazy-loaded module is an **`NgModule`** instead of a **`NgModuleFactory`**. So how do we make sure that we are loading an **`NgModuleFactory`**?

From NgModule to NgModuleFactory with the AOT Compiler:

The Angular compiler's job is to statically analyse all of the code in our entire application, and to efficiently compile all of our templates and styles. It takes our **`NgModule`** files, and turns them into **`NgModuleFactory`** files, which contain the generated code that will create our views at runtime.

The compiler is able to start at a given file, and navigate through all of the import statements (e.g. `import { Thing } from './path/to/thing';`) and build up a tree of all of the referenced modules. In order to split our application into chunks, we have to change our code to explicitly break this tree of references apart, while also making sure that the compiler knows about all the split parts of our application. The way we do this in an Angular application is with the **`loadChildren`** property, specifically with the magic string format:

```
export function listLazyRoutes(
    moduleMeta: CompileNgModuleMetadata, reflector: St
  const allLazyRoutes: LazyRoute[] = [];
  for (const {provider, module} of moduleMeta.transiti
    if (tokenReference(provider.token) === reflector.R
      const loadChildren = _collectLoadChildren(provid
      for (const route of loadChildren) {
        allLazyRoutes.push(parseLazyRoute(route, refle
      }
    }
  }
  return allLazyRoutes;
}

export function parseLazyRoute(
    route: string, reflector: StaticReflector, module?
  const [routePath, routeName] = route.split('#');
```

Angular AOT compiler code for finding lazy-loaded routes with the **ROUTES InjectionToken**

The Angular AOT compiler finds all the `ROUTES` by using the `InjectionToken` and then looks for any strings using the `./path/to/my.module#MyModule` format. Each time it finds one, the compiler will start from the given path, build up the tree of referenced files, and compile each `NgModule` into an `NgModuleFactory`. If we don't use that format, we don't end up with the `NgModuleFactory` that the runtime needs. If we do use that format, then we end up with a generated file with an unknown path containing the `NgModuleFactory`, which means we can't reference it with `import()` ...

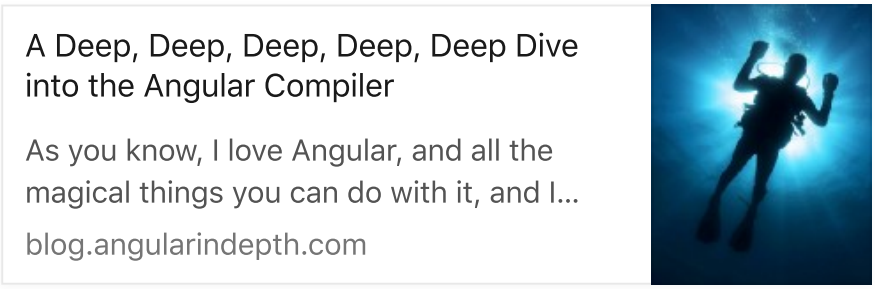
Altogether, this means that even though the types in Angular 7.x.x allow us to specify an async function for `loadChildren` it will never work in a production build of our application 😭 😭 😭. But why does the `import()` operator work in JIT mode?

The `import()` operator is another way to declare that we want to lazily reference another part of our application. Modern tooling can detect it, mark the referenced path as another entry point, and lazily load the reference at runtime. Unfortunately, only the Angular CLI knows how to turn a `NgModule` into an `NgModuleFactory`, and it doesn't know about `import()`. We saw it working because JIT mode only needs an uncompiled `NgModule`.

This is where we hit a bit of a dead end in Angular 7.x.x. For us to be able to use `import()`, something needs to change with how Angular works. Luckily for us, that change is just around the corner!

. . .

> *You can learn more about how the Angular compiler works in this incredible article by Uri Shaked ❤️:*



. . .

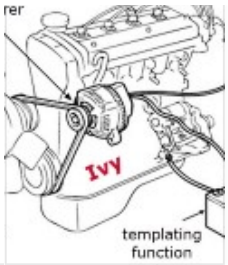
Lazy-loaded routes in Angular 8.x.x with Ivy:

One of the main design goals of the new Ivy renderer is to remove the differences between the JIT and AOT modes based on the principle of locality. Each file knows about everything that it needs to know about, without extra metadata files—this means no more `NgModuleFactory` classes!

That means that we no longer need to run a separate AOT compile, no longer have to worry about generated files with unknown paths, and we can use our `import()` operator!

. . .

> *You can learn more about the changes in Ivy in this post by Max Koretskyi aka Wizard ❤️:*



. . .

Upgrading from magic strings to nice async functions:

Now we know we have a cool new tool that we will be able to use soon! But we also have a lot of existing code that uses the magic string syntax. Wouldn't it be great if there was an automatic way to upgrade all of our old code?

We can write a custom TSLint rule and fixer to do all this for us! Let's look at the whole rule first, and then break it down:

```
// Dependencies:
import { tsquery } from '@phenomnomnominal/tsquery';
import { Replacement, RuleFailure, Rules } from 'tslin
import { SourceFile } from 'typescript';

// Constants:
const LOAD_CHILDREN_SPLIT = '#';
const LOAD_CHILDREN_VALUE_QUERY = `StringLiteral[value
const LOAD_CHILDREN_ASSIGNMENT_QUERY = `PropertyAssign

const FAILURE_MESSAGE = 'Found magic `loadChildren` st

export class Rule extends Rules.AbstractRule {
    public apply(sourceFile: SourceFile): Array<RuleFa
        const options = this.getOptions();
        const [preferAsync] = options.ruleArguments;

        return tsquery(sourceFile, LOAD_CHILDREN_ASSIG
            const [valueNode] = tsquery(result, LOAD_C
            let fix = preferAsync === 'async' ? this._

            // Try to fix indentation in replacement:
            const { character } = sourceFile.getLineAn
            fix = fix.replace(/\n/g, `\n${' '.repeat(c

            const replacement = new Replacement(valueN
            return new RuleFailure(sourceFile, result.
        });
    }

    private promiseReplacement (loadChildren: string)
```

First things first, we have a TSQuery selector to choose the part of the code we want to modify:

```
PropertyAssignment
:not(:has(Identifier[name="children"]))
:has(Identifier[name="loadChildren"])
:has(StringLiteral[value=/.*#*/])
```

We use this selector in our rule to give us access to the right parts of our code:

```
// Constants:
const LOAD_CHILDREN_VALUE_QUERY = `StringLiteral[value
const LOAD_CHILDREN_ASSIGNMENT_QUERY = `PropertyAssign

export class Rule extends Rules.AbstractRule {
    public apply(sourceFile: SourceFile): Array<RuleFa
        const options = this.getOptions();
        const [preferAsync] = options.ruleArguments;

        return tsquery(sourceFile, LOAD_CHILDREN_ASSIG
            const [valueNode] = tsquery(result, LOAD_C
```

We can parse out the magic string, and create the replacement code. The fixer can generate code that uses either a raw `Promise` or `async/await`:

```
export class Rule extends Rules.AbstractRule {
    public apply(sourceFile: SourceFile): Array<RuleFa
        // ...
        let fix = preferAsync === 'async' ? this._asyn
        // ...
    }

    private _promiseReplacement (loadChildren: string)
        const [path, moduleName] = this._getChunks(loa
        return `() => import('${path}').then(m => m.${
    }

    private _asyncReplacement (loadChildren: string):
        const [path, moduleName] = this._getChunks(loa
        return `async () => {
    const { ${moduleName} } = await import('${path}');
```

Finally, we need to (somewhat clumsily) handle any indentation in the source code, and apply our fix:

```
const FAILURE_MESSAGE = 'Found magic `loadChildren` st

export class Rule extends Rules.AbstractRule {
    public apply(sourceFile: SourceFile): Array<RuleFa
        // ...
        // Try to fix indentation in replacement:
        const { character } = sourceFile.getLineAndCha
        fix = fix.replace(/\n/g, `\n${' '.repeat(chara

```

And there we have it! One nice new sparkly TSLint fixer. If anyone wants to show me/help me how to turn this into an ESLint rule, then that'd be awesome 😇 .

The End!

Phew! How's that for a brain dump of soon to be obsolete knowledge! I hope you learned a thing or two, maybe feel a little bit less scared about reading Angular source code, and maybe feel a bit inspired to write your own automation for upgrading your apps. Please reach out to me with any questions, and I'd love your feedback!

❤️ 🦄

P.S. Big thanks to Thomas Burleson for "encouraging" me to write stuff down!