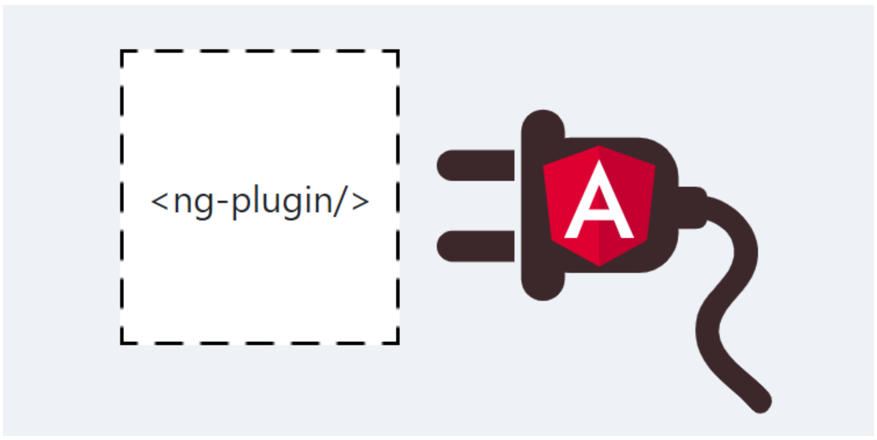
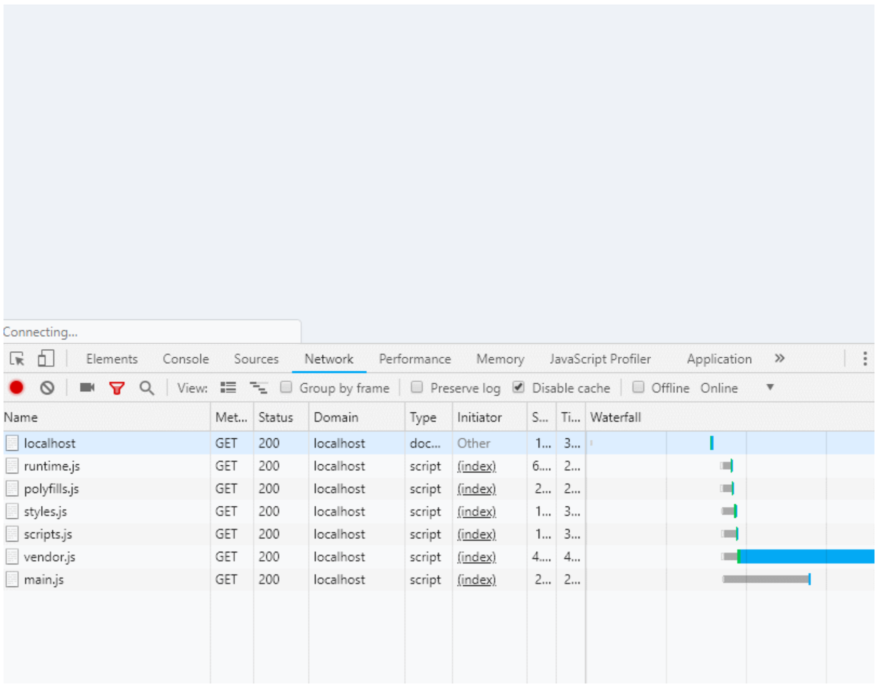
Building an extensible Dynamic Pluggable Enterprise Application with Angular

Alexey Zuev Follow
Apr 2 · 11 min read



In this article, we will talk about how we can leverage Angular CLI build tools to create an **AOT precompiled Angular plugin**, which can share common code with other plugins and even work with Angular universal. This is an unofficial solution, but it works well for our case.

Here's a simple representation of what we're building:



Name	Met...	Status	Domain	Type	Initiator	S...	Ti...	Waterfall
localhost	GET	200	localhost	doc...	Other	1...	3...	
runtime.js	GET	200	localhost	script	(index)	6....	2...	
polyfills.js	GET	200	localhost	script	(index)	2...	2...	
styles.js	GET	200	localhost	script	(index)	1...	3...	
scripts.js	GET	200	localhost	script	(index)	1...	3...	
vendor.js	GET	200	localhost	script	(index)	4....	4...	
main.js	GET	200	localhost	script	(index)	2...	2...	

Pluggable architecture example

Here we have a simple page where we get plugins configuration from a `plugins-config.json` file. Then we first lazy load the AOT compiled plugin (`plugin1.js`) which has a dependency to `shared.js` . The shared library contains a Tabs component and Tabs ng factories. Sometime later we load the second plugin(`plugin2.js`) that reuses the code from the previously loaded `shared.js` library.

Here's the angular-plugin-architecture Github repository if you want to take a look at the source code.

> *We use Angular CLI 7.3.6 in the demo*

Let's to not wait for Ivy but instead deal with the current ViewEngine today

· · ·

What are we going to discuss here?

- Why Angular?
- The goal
- What's a plugin?

Why Angular?

The Angular team and community are continuing the rapid growth of its ecosystem.

Angular helps us to structure our code in a consistent manner so every new developer can easily be involved in a project.

We like the fact that we're following best web practices with Angular, just think of typescript, observables, server-side rendering, web workers, differential loading, progressive web application(PWA), lazy loading, etc. All of this helps us to adopt those features in a fast manner.

There are also more features that Angular offers us, like built-in dependency injection system, reactive forms, schematics and so on.

That is why we usually choose Angular when building an enterprise application.

The goal

One day our client asked us to add a new feature to his existing Angular Universal application. He wanted to have a pluggable

Content Management System(CMS). The goal was to add the possibility to extend the functionality of the current app so that a 3rd party developer could easily develop a new module on his own and upload it. Then, the Angular app should pick it up without having to recompile the whole application and redeploy.

Simply put, we need to develop **a plugin system**.

What is a plugin?

> *Plugin systems allow an application to be extended without modification of the core application code.*

It sounds simple but writing a plugin with Angular is always a challenge.

Why is it so hard to create a plugin with Angular?

One of my colleagues, who worked with AngularJS long time ago, said that he saw an Angular 2 application written in plain es5 (hmm.. maybe he found my jsfiddle or my old repo). So he suggested creating an Angular module in es5, putting it in a folder and that the main app should make it work somehow.

Don't get me wrong but I'm with Angular since 2 alpha and Angular 2 (or just Angular) is a completely new thing.

Of course, the main pitfall here is Ahead Of Time (AOT) compilation. The main advantage of using AOT is better performance for your application.

I saw lots of examples out there that use JitCompiler for building a pluggable architecture. This is not the way we want to go. We have to keep our app fast and not include compiler code in the main bundle. That's why we shouldn't use es5 because only TypeScript code can be AOT precompiled. Also, the current implementation of Angular AOT is based on the `@NgModule` transitive scope and requires that all should be compiled together. All of this makes things harder.

Another pitfall is that we need to share code between plugins to avoid code duplication. What kind of duplication can we consider? We distinguish two types of code that can be duplicated:

- The code that we write or the code that we take from `node_modules`
- The code produced by AOT, think of component and module factories (`component.ngfactory.js` and `module.ngfactory.js`). Actually, this is a huge amount of code.

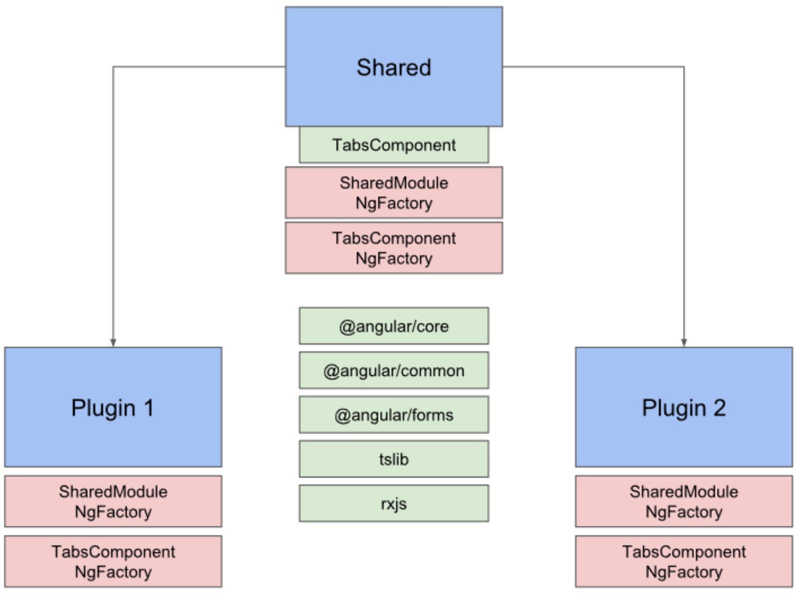
In order to avoid these code duplications, we need to deal with the fact of how ViewEngine generates a factory.

> *If you don't know, the ViewEngine is the current Angular rendering engine*

The problem here is that code generated by the Angular compiler can point to ViewFactory from another piece of generated code. For instance, here's how element definition is linked to ViewDefinitionFactory (Github source code)

```
elementDef(…, componentView?: null | ViewDefinitionFactory,
componentRendererType?: RendererType2 | null)
```

So this results in getting duplicates of all the factories from the shared library.



Duplicates of ngFactories in non-optimized plugin

So when we were discussing the Angular plugin system, we should keep the following in mind:

Requirements

- AOT
- Avoid duplicated code (packages like `@angular/core{common,forms,router},rxjs,tslib`)
- Use a shared library in all plugins. But, DO NOT SHIP generated factories from that shared library in each plugin. Rather, reuse library code and factories.
- For importing the external modules we just need to know one thing: their bundle file path.
- Our code should recognize the module and place the plugin into the page.
- Support server-side rendering
- Load the module only when needed
- Support the same level of optimization that Angular CLI gives us

All of these considerations led us to our own solution.

Similar solutions

There are different approaches out there, but they lack the crucial parts: support for AOT, optimized code, and non-duplicated code.

And one of the solutions that is close to our needs is:
https://github.com/iwnow/angular-plugin-example

It uses rollup to produce the plugin bundle in umd format.

However, here are the drawbacks I see with this approach:

- ❌ it doesn't use the optimization techniques that Angular CLI offers us: i.e. it doesn't remove Angular decorators and doesn't run buildOptimizer.
- ❌ it duplicates factories if we use shared components in each plugin.

Towards a Solution

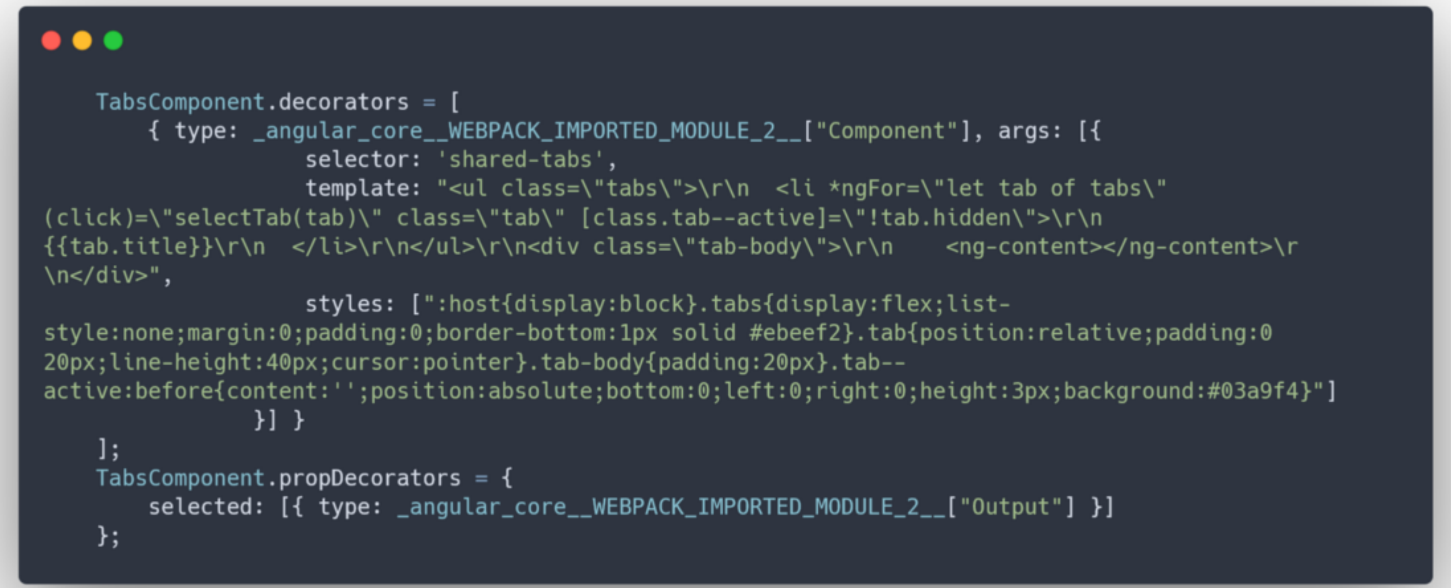
Fortunately, Angular is very extendable through custom scripts.

Since Angular 6, there's the possibility to hook into the compilation process using builders. It allows us to add a custom webpack configuration, including all the benefits you can imagine from a vanilla webpack setup.

So I thought that it could be a good idea to write a custom builder for building plugins.

Angular CLI supports the generation of libraries. But ng-packagr, that is used to build that library, generates many artifacts that we don't need. (Yeah, I know that it follows the Angular Package Format(APF)). And those artifacts can be only consumed by other Angular applications.

And here I thought that I could use an Angular application to build my plugins. By application, I mean an application which is generated by using a CLI command like `ng generate application` . This approach is beneficial because it gives us the same level of optimization that Angular CLI does. For example, we won't get Angular specific decorators after AOT.



```
    TabsComponent.decorators = [
        { type: _angular_core__WEBPACK_IMPORTED_MODULE_2__["Component"], args: [{
                    selector: 'shared-tabs',
                    template: "<ul class=\"tabs\">\r\n  <li *ngFor=\"let tab of tabs\"
(click)=\"selectTab(tab)\" class=\"tab\" [class.tab--active]=\"!tab.hidden\">\r\n
{{tab.title}}\r\n  </li>\r\n</ul>\r\n<div class=\"tab-body\">\r\n    <ng-content></ng-content>\r
\n</div>",
                    styles: [":host{display:block}.tabs{display:flex;list-
style:none;margin:0;padding:0;border-bottom:1px solid #ebeef2}.tab{position:relative;padding:0
20px;line-height:40px;cursor:pointer}.tab-body{padding:20px}.tab--
active:before{content:'';position:absolute;bottom:0;left:0;right:0;height:3px;background:#03a9f4}"]
                }] }
    ];
    TabsComponent.propDecorators = {
        selected: [{ type: _angular_core__WEBPACK_IMPORTED_MODULE_2__["Output"] }]
    };
```

Decorators in non-optimized bundle

Now, if only I could find a way to produce a single bundle with all exports I need:

Single bundle

So, I started with:

```
ng generate application plugins —minimal
```

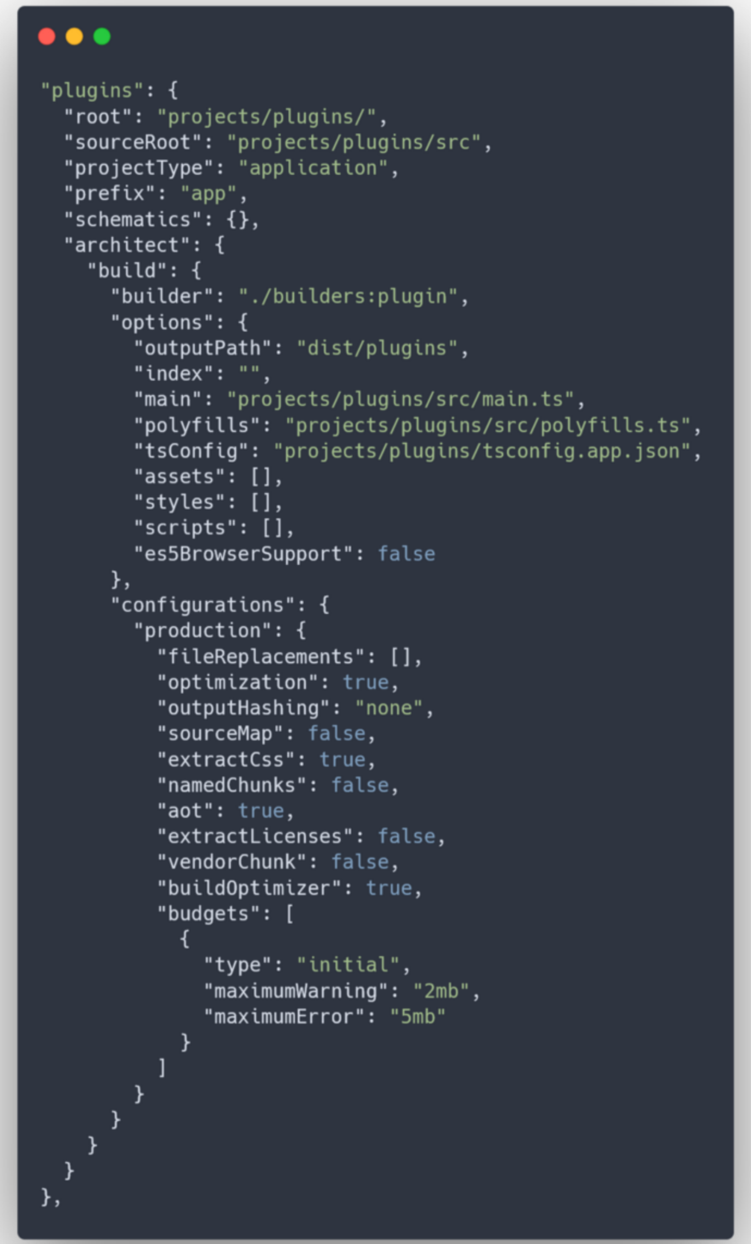
which gave me a simple new Angular application in `projects` folder

```
projects
   |_ plugins
src
angular.json
```

Next, I removed all unnecessary files so that it looked like:

```
projects
   |_ plugins
        |_ src
            |_ plugin1
            |_ plugin2
            main.ts
        tsconfig.app.json
src
angular.json
```

Then I removed all the redundant stuff from `angular.json` for that project:



```
"plugins": {
  "root": "projects/plugins/",
  "sourceRoot": "projects/plugins/src",
  "projectType": "application",
  "prefix": "app",
  "schematics": {},
  "architect": {
    "build": {
      "builder": "./builders:plugin",
      "options": {
        "outputPath": "dist/plugins",
        "index": "",
        "main": "projects/plugins/src/main.ts",
        "polyfills": "projects/plugins/src/polyfills.ts",
        "tsConfig": "projects/plugins/tsconfig.app.json",
        "assets": [],
        "styles": [],
        "scripts": [],
        "es5BrowserSupport": false
      },
      "configurations": {
        "production": {
          "fileReplacements": [],
          "optimization": true,
          "outputHashing": "none",
          "sourceMap": false,
          "extractCss": true,
          "namedChunks": false,
          "aot": true,
          "extractLicenses": false,
          "vendorChunk": false,
          "buildOptimizer": true,
          "budgets": [
            {
              "type": "initial",
              "maximumWarning": "2mb",
              "maximumError": "5mb"
            }
          ]
        }
      }
    }
  }
},
```

plugins project configuration in angular.json

As you can see, I removed `index`, all `assets`, `scripts`, `fileReplacements` etc. and I also created a custom PluginBuilder which is located in `./builders:plugin`.

The builder should produce **a single umd bundle**:

```
// Make sure we are producing a single bundle
delete config.entry.polyfills;
delete config.optimization.runtimeChunk;
delete config.optimization.splitChunks;
delete config.entry.styles;

config.output.library = pluginName;
config.output.libraryTarget = 'umd';
```

Also, it should **provide externals** and **generate the appropriate exports** depending on which plugin we're building. This is a vital part of the builder.

Externals

Webpack allows us to provide externals. So they might look like:

```
config.externals = {
  rxjs: 'rxjs',
  '@angular/core': 'ng.core',
  '@angular/common': 'ng.common',
  '@angular/forms': 'ng.forms',
  '@angular/router': 'ng.router',
  tslib: 'tslib'
  // put here other common dependencies
};
```

There's another option which I'll show a bit later in the **Shared library** section.

Dynamic exports

Here is the interesting part which gives us the ability to build different plugins with the same build command.

First, the `tsconfig.app.json` of the `plugins` application only looks at the `main.ts` file so that we only compile what we provide in `main.ts`.

tsconfig.app.json

```
{
  "extends": "../../tsconfig.json",
  "compilerOptions": {
    ...
  },
  "files": ["./src/main.ts"]
}
```

Next, the builder takes the `modulePath` parameter:

```
modulePath=./plugin1/plugin1.module#Plugin1Module
```

You may notice that it looks like a path to a lazy module when we're working on lazy loading. The builder analyzes that path and generates dedicated code in `main.ts`. For example, in the `modulePath` above we get:

```
export * from './plugin1/plugin1.module';
export * from './plugin1/plugin1.module.ngfactory';
import { Plugin1ModuleNgFactory } from
'./plugin1/plugin1.module.ngfactory';
export default Plugin1ModuleNgFactory;
```

The key point here is to be able to export not only the code, but also the factories generated by Angular compiler.

I keep `main.ts` empty so that I can update it dynamically before the build and so it will export what we want.

Build plugin

Here's an example of the command I use to build the plugin:

```
"build:plugin1": "ng build --project plugins --prod --
modulePath=./plugin1/plugin1.module#Plugin1Module --
pluginName=plugin1 --outputPath=./src/assets/plugins",
```

What is going here?

- `ng build — project plugins — prod`
  Build the above created `plugins` application in prod mode.
- `modulePath=./plugin1/plugin1.module#Plugin1Module`
  Provide a path to the module we want to be a plugin.

- `pluginName=plugin1`
  Define the name for the bundle.
- `– outputPath=./src/assets/plugins`
  Specify the directory where the generated bundle should be placed.

Externals and Shared Angular libraries

We also can use a shared library for plugins. To be able to expose all factories from the shared library we need to have the library code in one file. Otherwise, it would be hard to figure out what's the path for those ngfactories.

And that's what ng-packagr generates for us.

We can build generated by ng-packagr file. So, first, I create a library:

```
ng generate library shared
```

and then build it for the plugin system:

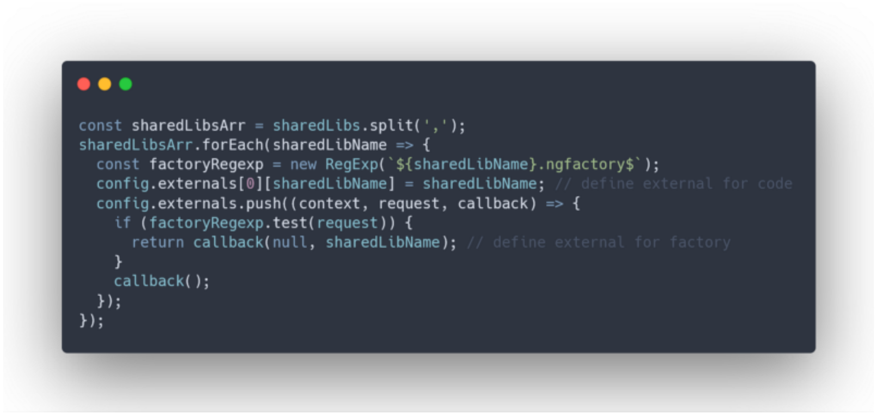
```
"build:shared": "ng build shared && ng build --project
plugins --prod --modulePath=shared#SharedModule --
pluginName=shared  --outputPath=./src/assets/plugins",
```

Now, when I create a `shared` library I can reuse it in my plugins:

```
"build:plugin1": "ng build --project plugins --prod --
modulePath=./plugin1/plugin1.module#Plugin1Module --
pluginName=plugin1 --sharedLibs=shared --
outputPath=./src/assets/plugins",
```

Note `--sharedLibs=shared`

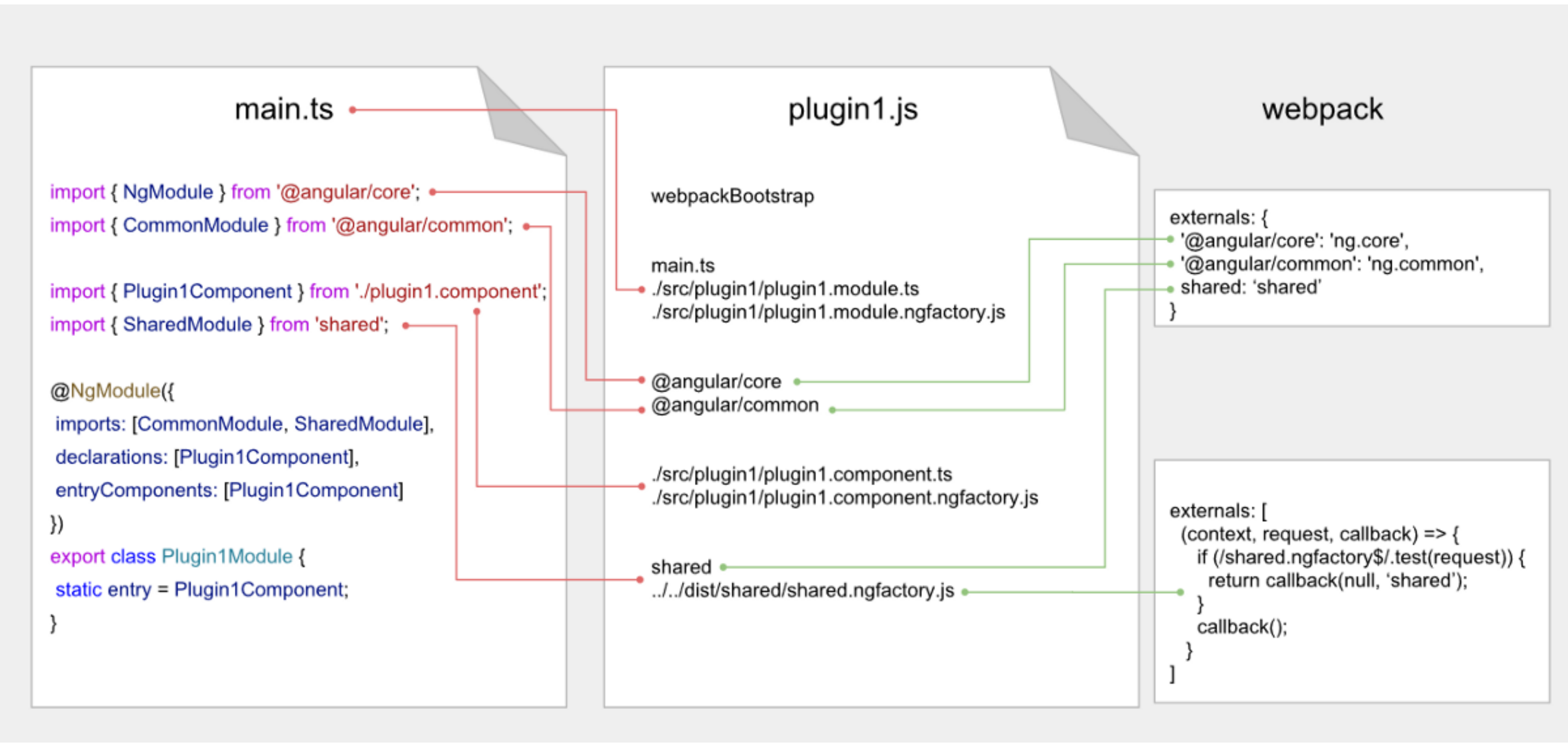
I'm controlling the factory externals via the webpack `externals` options where I'm providing a special function to catch all ngfactory imports.



```
const sharedLibsArr = sharedLibs.split(',');
sharedLibsArr.forEach(sharedLibName => {
  const factoryRegexp = new RegExp(`${sharedLibName}.ngfactory$`);
  config.externals[0][sharedLibName] = sharedLibName; // define external for code
  config.externals.push((context, request, callback) => {
    if (factoryRegexp.test(request)) {
      return callback(null, sharedLibName); // define external for factory
    }
    callback();
  });
});
```

Webpack externals to catch factories imports

This way we cover all possible externals in our bundle:



```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { Plugin1Component } from './plugin1.component';
import { SharedModule } from 'shared';

@NgModule({
 imports: [CommonModule, SharedModule],
 declarations: [Plugin1Component],
 entryComponents: [Plugin1Component]
})
export class Plugin1Module {
 static entry = Plugin1Component;
}
```

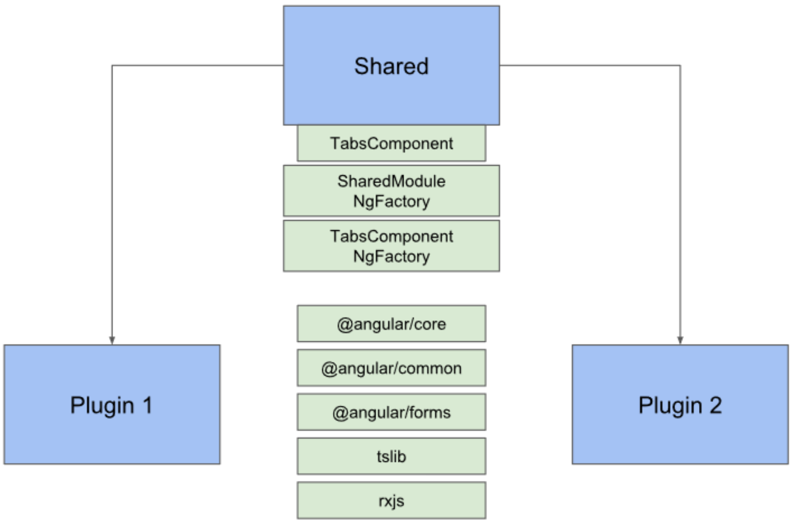
```
webpackBootstrap

main.ts
./src/plugin1/plugin1.module.ts
./src/plugin1/plugin1.module.ngfactory.js

@angular/core
@angular/common

./src/plugin1/plugin1.component.ts
./src/plugin1/plugin1.component.ngfactory.js

shared
../../dist/shared/shared.ngfactory.js
```

```
externals: {
  '@angular/core': 'ng.core',
  '@angular/common': 'ng.common',
  shared: 'shared'
}
```

```
externals: [
  (context, request, callback) => {
    if (/shared.ngfactory$/.test(request)) {
      return callback(null, 'shared');
    }
    callback();
  }
]
```

Webpack externals cover all possible external imports

And finally, get what we wanted:

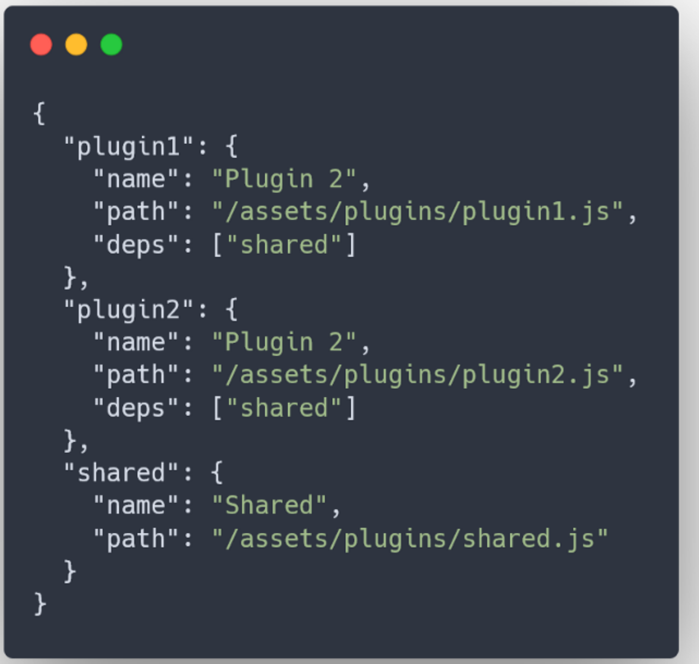


Plugins share all common code

Consuming the plugin

First of all, we need to load our configuration from the server.

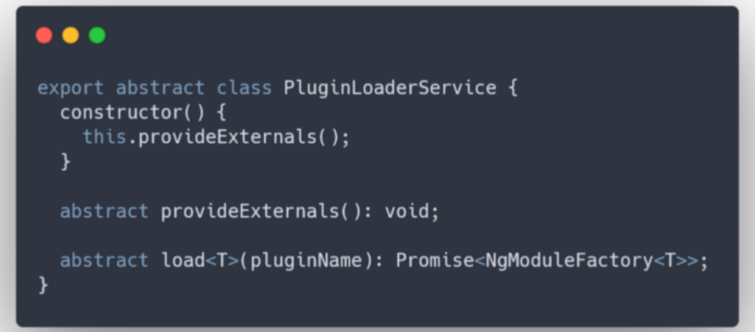
For the sake of simplicity, we load a simple JSON file, that looks like this:



```json
{
  "plugin1": {
    "name": "Plugin 2",
    "path": "/assets/plugins/plugin1.js",
    "deps": ["shared"]
  },
  "plugin2": {
    "name": "Plugin 2",
    "path": "/assets/plugins/plugin2.js",
    "deps": ["shared"]
  },
  "shared": {
    "name": "Shared",
    "path": "/assets/plugins/shared.js"
  }
}
```

plugins-config.json

Then, in order to consume the built plugin, we create two services for the client and server sides which extend `PluginLoaderService` abstract class:



```
export abstract class PluginLoaderService {
  constructor() {
    this.provideExternals();
  }

  abstract provideExternals(): void;

  abstract load<T>(pluginName): Promise<NgModuleFactory<T>>;
}
```

Base interface for LoaderService

Client Side

We use Angular Dependency Injection to provide a client specific loader only for the client code:

app.module.ts

```
providers: [
  { provide: PluginLoaderService, useClass:
ClientPluginLoaderService }
],
```

In order to load the plugin on the client-side, we use a minimal `systemjs@3.0.2` build that supports loading AMD modules.

```
"scripts": [
  "node_modules/systemjs/dist/s.js",
  "node_modules/systemjs/dist/extras/named-register.js",
  "node_modules/systemjs/dist/extras/amd.js"
],
```

Externals are provided by the `define` global function:

```
Object.keys(PLUGIN_EXTERNALS_MAP).forEach(externalKey =>
  window.define(externalKey, [], () =>
PLUGIN_EXTERNALS_MAP[externalKey])
);
```

And the load function looks like

```
load<T>(pluginName): Promise<NgModuleFactory<T>> {
  return SystemJs.import(config[pluginName].path).then(
    module => module.default.default
  );
}
```

Server Side

The Server code has its own implementation of the loader service:

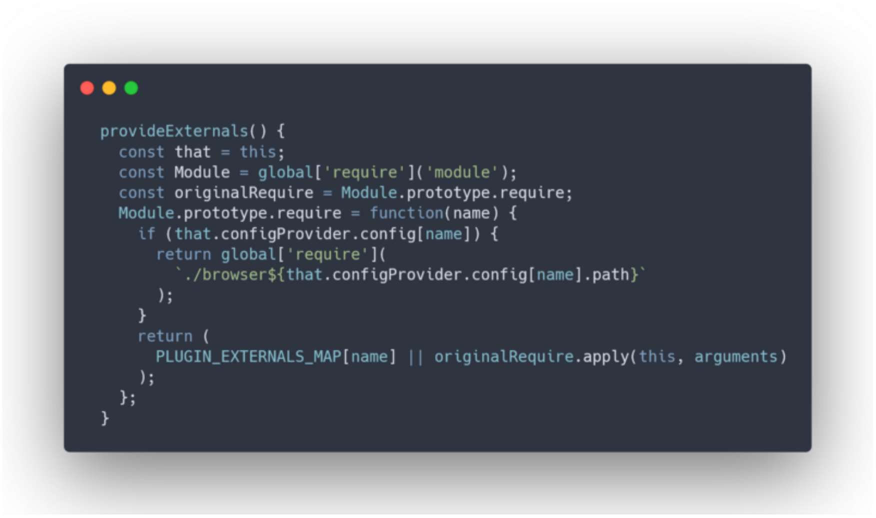
app.server.module.ts

```
providers: [
  { provide: PluginLoaderService, useClass:
ServerPluginLoaderService }
],
```

The service uses the nodejs require method to get the module:

```
load<T>(pluginName): Promise<NgModuleFactory<T>> {
  const factory = global['require']
(`./browser${config[pluginName].path}`)
    .default;
  return Promise.resolve(factory);
}
```

We also override the native `require` method in order to provide externals:

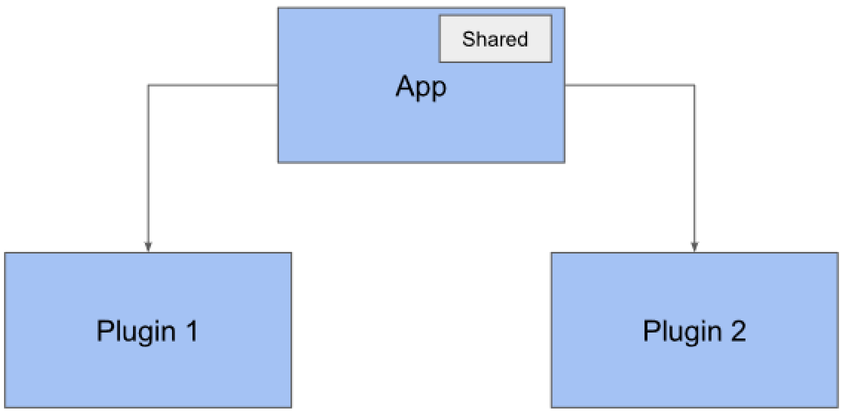


```
provideExternals() {
  const that = this;
  const Module = global['require']('module');
  const originalRequire = Module.prototype.require;
  Module.prototype.require = function(name) {
    if (that.configProvider.config[name]) {
      return global['require'](
        `./browser${that.configProvider.config[name].path}`
      );
    }
    return (
      PLUGIN_EXTERNALS_MAP[name] || originalRequire.apply(this, arguments)
    );
  };
}
```

Externals on the server-side

The case when the main application uses the same shared components as plugins

Let’s now imagine that our shared library should be also used in our main application:



Plugins and the main app are shared the same library

What can we do to not duplicate the code?

Since we have the `shared` library already built in the main app we can define it as externals to other plugins.

plugin-externals.prod.ts

```
import * as shared from 'shared';

export const PLUGIN_EXTERNALS_MAP = {
  'ng.core': core,
  'ng.common': common,
  'ng.forms': forms,
  'ng.router': router,
  rxjs,
  tslib,
  shared: { ...shared, ...require('shared/shared.ngfactory') }
}
};
```

This way we share both the library code and the code produced by aot.

You may note that I have `prod` postfix in the file above. That's because we define such external only for production build.

```
"configurations": {
  "production": {
    "fileReplacements": [
      ...
      {
        "replace": "src/app/services/plugin-loader/plugin-
externals.ts",
        "with": "src/app/services/plugin-loader/plugin-
externals.prod.ts"
      }
    ],
```

In dev mode we still load `shared` library through Systemjs as other plugins.

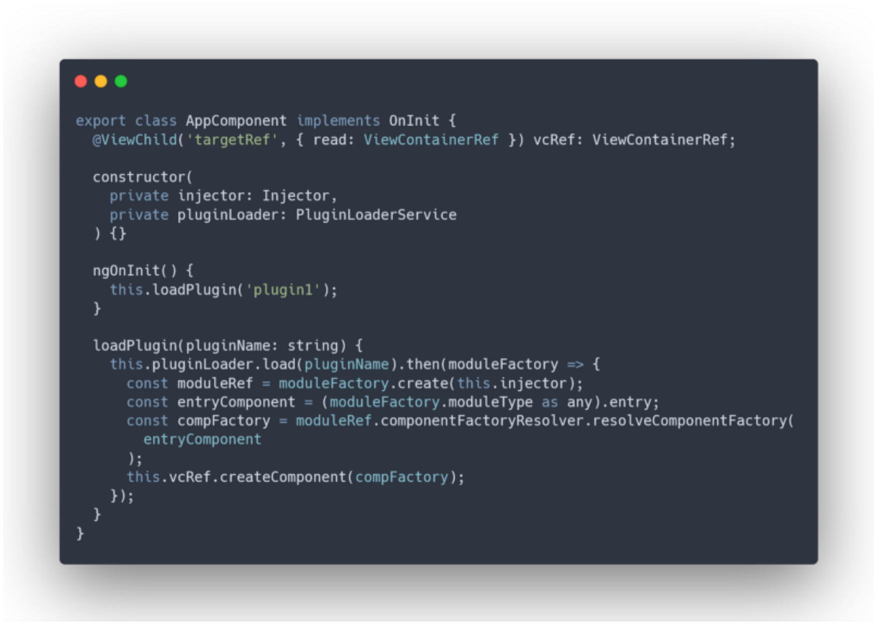
An example of the code for this case can be found in share-lib-between-app-and-plugins branch.

How do we render the plugin?

Now that we have all we need to get NgModuleFactory, we can render our plugins with the help of one of the well-known Angular API methods for creating a dynamic component:

- `componentFactory.create`
- `viewContainerRef.createComponent`

In the example below, we used the second option:



```
export class AppComponent implements OnInit {
  @ViewChild('targetRef', { read: ViewContainerRef }) vcRef: ViewContainerRef;

  constructor(
    private injector: Injector,
    private pluginLoader: PluginLoaderService
  ) {}

  ngOnInit() {
    this.loadPlugin('plugin1');
  }

  loadPlugin(pluginName: string) {
    this.pluginLoader.load(pluginName).then(moduleFactory => {
      const moduleRef = moduleFactory.create(this.injector);
      const entryComponent = (moduleFactory.moduleType as any).entry;
      const compFactory = moduleRef.componentFactoryResolver.resolveComponentFactory(
        entryComponent
      );
      this.vcRef.createComponent(compFactory);
    });
  }
}
```

Low-level API to render plugins

Isolating the main app from errors in a plugin

One of the problems with a pluggable architecture is that exceptions in plugins can sink the entire app.

Fortunately, there are a few different ways for dealing with such errors:

- Global ErrorHandler
- Zone.onError.subscribe
- try catch

This lets us gracefully handle any errors when creating a plugin, and safely isolate, kill, and report bad components without impacting the rest of our app.

Summary

Angular evolves each quarter. But, it still doesn’t have a simple way to deal with dynamic templating. It’s the most required part when we are working with a big application and clients demand that all should be dynamic, lazy and changeable at runtime.

Luckily, Angular CLI provides us the tools to configure a build for our specific needs. We showed in this article one option for how to build an AOT precompiled plugin. It might not exactly work for your needs, but possibly you can borrow some of the ideas in this article. So, I hope you found them useful.

Thank you for reading!

. . .

The code can be found on Github: **angular-plugin-architecture**.