Here is what you need to know about dynamic components in Angular

Max Koretskyi aka Wizard Follow
May 30, 2017 · 10 min read

Create Angular components dynamically like a pro

. . .

If you've been programming with AngularJS (first generation of the framework) you probably got used to generating HTML strings on the fly, running them through `$compile` service and linking to a data model (scope) to get two-way data binding.

```
const template = '<span>generated on the fly: {{name}}
</span>'
const linkFn = $compile(template);
const dataModel = $scope.$new();
dataModel.name = 'dynamic'

// link data model to a template
linkFn(dataModel);
```

In AngularJS a directive can modify DOM in any way possible and the framework has no clue what the modifications will be. But the problem with such approach is the same as with any dynamic environment—it's hard to optimize for speed. Dynamic template evaluation is of course not the main culprit of AngularJS being viewed as a slow framework, but it certainly contributed to the reputation.

After studying Angular internals for quite some time it seems to be that the newer framework design was very much driven by the need for speed. You'll find many comments like this in the sources:

```
Attention: Adding fields to this is performance
sensitive!

Note: We use one type for all nodes so that loops that
loop over all nodes of a ViewDefinition stay
monomorphic!

For performance reasons, we want to check and update
the list every five seconds.
```

So Angular guys in the newer framework decided to provide less flexibility in return for a much greater speed. And introduced a JIT and AOT compilers and static templates. And factories. And factory resolver. And many other things that look hostile and unfamiliar to AngularJS community. But no worries. If you've come across these concepts before and is wondering what these are read on and achieve enlightenment.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

. . .

Component factory and compiler

In Angular every component is created from a factory. And factories are generated by the compiler using the data you supply in the `@Component` decorator. If after reading many article on the web you're still not sure what this decorator does read Implementing custom component decorator.

Under the hood Angular uses a concept of a View. The running framework is essentially a tree of views. Each view is composed of different types of nodes: element nodes, text nodes and so on. Each node is narrowly specialized in its purpose so that processing of such nodes takes as little time as possible. There are various providers associated with each node—like ViewContainerRef and TemplateRef. And each node knows how to respond to queries like ViewChildren and ContentChildren.

That's a lot of information for each node. Now, to optimize for speed all this information has to be available when the node is constructed and cannot be changed later. This is what compilation process does—collects all the required information and encapsulates it in the form of a component factory.

Suppose you define a component and its template like this:

```
@Component({
  selector: 'a-comp',
  template: '<span>A Component</span>'
})
class AComponent {}
```

Using this data the compiler generates the following slightly simplified component factory:

```
function View_AComponent_0(l) {
  return jit_viewDef1(0,[
      elementDef2(0,null,null,1,'span',...),
      jit_textDef3(null,['A Component ',...])
    ]
```

It describes the structure of a component view and is used when instantiating the component. The first node is element definition and the second one is text definition. You can see that each node gets the information it needs when being instantiated through parameters list. It's a job of a compiler to resolve all the required dependencies and provide them at the runtime.

If you have access to a factory you can easily create a component instance from it and insert into a DOM using viewContainerRef. I've written about it in Exploring Angular DOM manipulations. This is how it would look:

```
export class SampleComponent implements AfterViewInit
{
    @ViewChild("vc", {read: ViewContainerRef}) vc:
ViewContainerRef;

    ngAfterViewInit() {
        this.vc.createComponent(componentFactory);
    }
}
```

So the question now is how to get access of a component factory and we will see shortly.

. . .

Angular modules and ComponentFactoryResolver

Although AngularJS also had modules it lacked true namespaces for directives. There was always a potential for conflicts and no way to encapsulate utility directives inside a particular module. Luckily, Angular learnt its lessons and now provides proper namespacing for declarative types: directives, components and pipes.

Just as in AngularJS every component in the newer framework is part of some module. Components don't exist by themselves and if you want to use a component from a different module you have to import that module:

```
@NgModule({
    // imports CommonModule with declared directives
like
    // ngIf, ngFor, ngClass etc.
    imports: [CommonModule],
    ...
})
export class SomeModule {}
```

In turn, if a module wants to provide some components to be used by other module components it has to export these components. Here is how `CommonModule` does that:

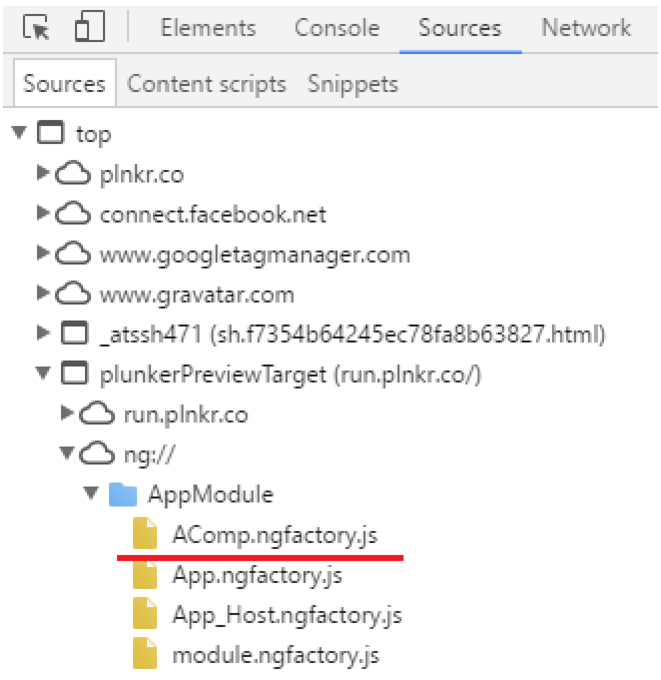
```
const COMMON_DIRECTIVES: Provider[] = [
    NgClass,
    NgComponentOutlet,
    NgForOf,
    NgIf,
    ...
];

@NgModule({
    declarations: [COMMON_DIRECTIVES, ...],
    exports: [COMMON_DIRECTIVES, ...],
    ...
})
export class CommonModule {
}
```

So each component is bound to a particular module and you can't declare the same component in different modules. If you do that you'll get an error:

```
Type X is part of the declarations of 2 modules: ...
```

When Angular compiles an application, it takes components that are defined in `entryComponents` of a module or found in components templates and generates component factories for them. You can see those factories in the `Sources` tab:



```
▼ top
  ▶ plnkr.co
  ▶ connect.facebook.net
  ▶ www.googletagmanager.com
  ▶ www.gravatar.com
  ▶ _atssh471 (sh.f7354b64245ec78fa8b63827.html)
  ▼ plunkerPreviewTarget (run.plnkr.co/)
    ▶ run.plnkr.co
    ▼ ng://
      ▼ AppModule
          AComp.ngfactory.js
          App.ngfactory.js
          App_Host.ngfactory.js
          module.ngfactory.js
```

In previous section we identified that if we had an access to a component factory we could then use it to create a component and insert into a view. Each module provides a convenient service for all its components to get a component factory. This service is ComponentFactoryResolver. So, if you define a `BComponent` on the module and want to get a hold of its factory you can use this service from a component belonging to this module:

```
export class AppComponent {
  constructor(private resolver:
ComponentFactoryResolver) {
    // now the `f` contains a reference to the cmp
factory
    const f =
this.resolver.resolveComponentFactory(BComponent);
  }
```

This only works if both components are defined in the same module or if a module with a resolved component factory is imported.

. . .

Dynamic module loading and compilation

But what if your components are defined on the other module that you don't want to load until the components in it are actually required? We can do that. This will be something similar to what router is doing with `loadChildren` configuration option.

There are two options how to load a module during runtime. The first one is to use the SystemJsNgModuleLoader provided by Angular. It is used by the router to load child routes if you're using SystemJS as a loader. It has one public method `load`, which loads a module to a browser and compile the module and all components declared in it. This method takes a path to a file with a module and export name and returns ModuleFactory:

```
loader.load('path/to/file#exportName')
```

If you don't specify export name, the loader will use the `default` export name. The other thing to note is that `SystemJsNgModuleLoader` requires a DI setup with some injections so you should define it as a provider like that:

```
providers: [
    {
      provide: NgModuleFactoryLoader,
      useClass: SystemJsNgModuleLoader
    }
  ]
```

You can of course specify any token for `provide`, but the router module uses `NgModuleFactoryLoader` so it's probably a good thing to use the same approach.

So, the here is the full code to load a module and get a component factory:

```
@Component({
  providers: [
    {
      provide: NgModuleFactoryLoader,
      useClass: SystemJsNgModuleLoader
    }
  ]
})
export class ModuleLoaderComponent {
  constructor(private _injector: Injector,
              private loader: NgModuleFactoryLoader) {
  }

  ngAfterViewInit() {

this.loader.load('app/t.module#TModule').then((factory
) => {
      const module = factory.create(this._injector);
      const r = module.componentFactoryResolver;
      const cmpFactory =
r.resolveComponentFactory(AComponent);

      // create a component and attach it to the view
      const componentRef =
cmpFactory.create(this._injector);
      this.container.insert(componentRef.hostView);
    })
  }
}
```

But there is a one problem with using `SystemJsNgModuleLoader`. Under the hood it uses compileModuleAsync method of the compiler. This method creates factories only for components declared in `entryComponents` of a module or found in components templates. But what if you don't want to declare your components as entry components? There is a solution—load the module yourself and use compileModuleAndAllComponentsAsync method. It generates factories for all components on the module and returns them as an instance of `ModuleWithComponentFactories`:

```
class ModuleWithComponentFactories<T> {
    componentFactories: ComponentFactory<any>[];
    ngModuleFactory: NgModuleFactory<T>;
```

Here is the full code that shows how to load a module yourself and get access to all component factories:

```
ngAfterViewInit() {
  System.import('app/t.module').then((module) => {

_compiler.compileModuleAndAllComponentsAsync(module.TM
odule)
        .then((compiled) => {
          const m =
compiled.ngModuleFactory.create(this._injector);
          const factory =
compiled.componentFactories[0];
          const cmp = factory.create(this._injector,
[], null, m);
        })
    })
}
```

Keep in mind that this approach makes use of a compiler which is not supported as a Public API. Here is what the docs say:

> *One intentional omission from this list is* `@angular/compiler`*, which is currently considered a low level api and is subject to internal changes. These changes will not affect any applications or libraries using the higher-level apis (the command line interface or JIT compilation via* `@angular/platform-browser-dynamic`*). Only very specific use-cases require direct access to the compiler API (mostly tooling integration for*

> *IDEs, linters, etc). If you are working on this kind of integration, please reach out to us first.*

. . .

Creating components on the fly

From the previous sections you found out how the dynamic components can be created in Angular. You know that this process requires an access to component factories which are placed on a module. Until now I've used modules that are defined before the runtime and can be loaded eagerly or lazily. But the good thing is that you don't have to define modules beforehand and then load them. You can create a module and a component on the fly just like in AngularJS.

Let's take the example I showed in the beginning and see how we can achieve the same in Angular. So, here is the example again:

```
const template = '<span>generated on the fly: {{name}}
</span>'
const linkFn = $compile(template);
const dataModel = $scope.$new();
dataModel.name = 'dynamic'

// link data model to a template
linkFn(dataModel);
```

The general flow to create and attach a dynamic content to the view is the following:

1. Define a component class and its properties and decorate the class
2. Define a module class, add the component to module declarations and decorate the module class
3. Compile module and all components to get hold of a component factory

The module is simply a class with a decorator applied to it. The same holds for a component. Since decorators are simple functions and available during runtime we can use them to decorate classes whenever we want. Here is the how to create and attach component dynamically on the fly:

```
@ViewChild('vc', {read: ViewContainerRef}) vc:
ViewContainerRef;

constructor(private _compiler: Compiler,
            private _injector: Injector,
            private _m: NgModuleRef<any>) {
}

ngAfterViewInit() {
  const template = '<span>generated on the fly:
{{name}}</span>';

  const tmpCmp = Component({template: template})(class
{
  });
  const tmpModule = NgModule({declarations: [tmpCmp]})
(class {
  });


this._compiler.compileModuleAndAllComponentsAsync(tmpM
odule)
    .then((factories) => {
      const f = factories.componentFactories[0];
      const cmpRef = this.vc.createComponent(f);
      cmpRef.instance.name = 'dynamic';
    })
}
```

You may want to replace anonymous class with a named class in decorators for better debugging information.

. . .

Ahead-of-Time Compilation

The compiler we've been using in the examples above is called Just-In-Time (JIT) compiler. You may have also heard about Ahead-Of-Time (AOT) compiler. Actually, Angular has only one compiler and it is referred to as JIT or AOT depending on when it is used. If it's used during runtime in a browser it is JIT. If you compile your components before the code is executed in a browser, it's AOT compilation. The latter is a preferred method and this manual section mentions good reason to do so—some of them are faster rendering and smaller angular framework download size.

If you use AOT compilation it usually means that you don't have a compiler instance at runtime. The examples above that don't require a compiler and simply use `ComponentFactoryResolver` will still work, but dynamic compilation will not possible. However, there is no reason you cannot load the compiler to a browser and use it. It requires a bit of a setup, but nothing fancy. Here is the code:

```
import { JitCompilerFactory } from
'@angular/compiler';

export function createJitCompiler() {
  return new JitCompilerFactory([{
    useDebug: false,
    useJit: true
  }]).createCompiler();
}

import { AppComponent }  from './app.component';

@NgModule({
  providers: [{provide: Compiler, useFactory:
createJitCompiler}],
  ...
})
export class AppModule {
}
```

Here we use `JitCompilerFactory` function from the `@angular/compiler` package that creates a compiler factory. We configure the compiler to use JIT and then simply create its instance available in the application through the regular token `Compiler` . No other changes are required in other parts of an application.

· · ·

Destroying components

The last thing is that if you've added components manually, don't forget to destroy them when parent component is destroyed:

```
ngOnDestroy() {
  if(this.cmpRef) {
    this.cmpRef.destroy();
  }
}
```

This will remove DOM by detaching component view from the view container and destroy the component view.

. . .

ngOnChanges

For all components that are added dynamically Angular performs change detection as it does for statically added components. It means that ngDoCheck lifecycle hook is called. However, ngOnChanges are not triggered even if dynamically created component declares `@Input` and the parent property changes. This is because the function that performs inputs checks is generated by the compiler during compilation. This function is part of the component factory and at the moment it can be generated only based on the information in the template. Since we don't use dynamic component in the template, the function is not generated by the compiler.

. . .

Github

All code samples used in this article are available in this repository.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story. For more insights follow me on Twitter and on Medium.