What you always wanted to know about Angular Dependency Injection tree

Alexey Zuev Follow
Mar 21, 2018 · 10 min read

If you didn't dive deep into angular dependency injection mechanism, your mental model should be that in angular application we have some root injector with all merged providers, every component has its own injector and lazy loaded module introduces new injector.

But maybe there is some more you should be aware of?

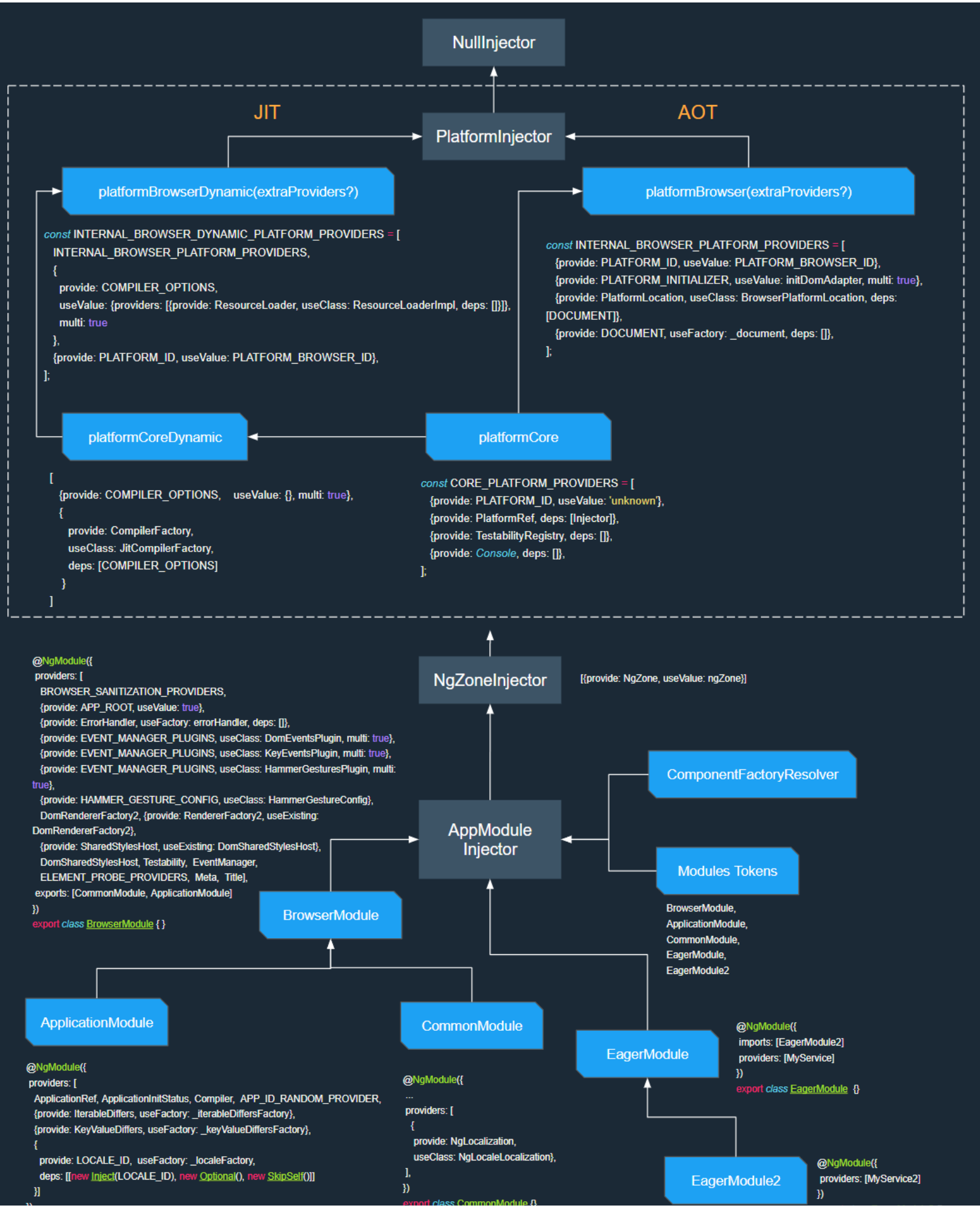
Also a while ago, so-called **Tree-Shakeable Tokens** feature was merged into master branch. If you are like me, you probably want to know what has changed.

So it's time to examine all these things and maybe find something new...

The Injector Tree

Most of angular developers know that angular creates root injector with singleton providers. But seems there is another injector which is higher that injector.

As a developer I want to understand how angular builds injector tree. Here is how I see the top part of Angular injector tree:


NullInjector
JIT
AOT
PlatformInjector
platformBrowserDynamic(extraProviders?)
platformBrowser(extraProviders?)
const INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS = [
INTERNAL_BROWSER_PLATFORM_PROVIDERS,
{
provide: COMPILER_OPTIONS,
useValue: {providers: [{provide: ResourceLoader, useClass: ResourceLoaderImpl, deps: []}]},
multi: true
},
{provide: PLATFORM_ID, useValue: PLATFORM_BROWSER_ID},
];
const INTERNAL_BROWSER_PLATFORM_PROVIDERS = [
{provide: PLATFORM_ID, useValue: PLATFORM_BROWSER_ID},
{provide: PLATFORM_INITIALIZER, useValue: initDomAdapter, multi: true},
{provide: PlatformLocation, useClass: BrowserPlatformLocation, deps: [DOCUMENT]},
{provide: DOCUMENT, useFactory: _document, deps: []},
];
platformCoreDynamic
platformCore
[
{provide: COMPILER_OPTIONS, useValue: {}, multi: true},
{
provide: CompilerFactory,
useClass: JitCompilerFactory,
deps: [COMPILER_OPTIONS]
}
]
const CORE_PLATFORM_PROVIDERS = [
{provide: PLATFORM_ID, useValue: 'unknown'},
{provide: PlatformRef, deps: [Injector]},
{provide: TestabilityRegistry, deps: []},
{provide: Console, deps: []},
];
@NgModule({
providers: [
BROWSER_SANITIZATION_PROVIDERS,
{provide: APP_ROOT, useValue: true},
{provide: ErrorHandler, useFactory: errorHandler, deps: []},
{provide: EVENT_MANAGER_PLUGINS, useClass: DomEventsPlugin, multi: true},
{provide: EVENT_MANAGER_PLUGINS, useClass: KeyEventsPlugin, multi: true},
{provide: EVENT_MANAGER_PLUGINS, useClass: HammerGesturesPlugin, multi: true},
{provide: HAMMER_GESTURE_CONFIG, useClass: HammerGestureConfig},
DomRendererFactory2, {provide: RendererFactory2, useExisting: DomRendererFactory2},
{provide: SharedStylesHost, useExisting: DomSharedStylesHost},
DomSharedStylesHost, Testability, EventManager,
ELEMENT_PROBE_PROVIDERS, Meta, Title],
exports: [CommonModule, ApplicationModule]
})
export class BrowserModule { }
NgZoneInjector
[{provide: NgZone, useValue: ngZone}]
ComponentFactoryResolver
AppModule Injector
Modules Tokens
BrowserModule,
ApplicationModule,
CommonModule,
EagerModule,
EagerModule2
BrowserModule
ApplicationModule
CommonModule
EagerModule
@NgModule({
providers: [
ApplicationRef, ApplicationInitStatus, Compiler, APP_ID_RANDOM_PROVIDER,
{provide: IterableDiffers, useFactory: _iterableDiffersFactory},
{provide: KeyValueDiffers, useFactory: _keyValueDiffersFactory},
{
provide: LOCALE_ID, useFactory: _localeFactory,
deps: [[new Inject(LOCALE_ID), new Optional(), new SkipSelf()]]
}]
})
@NgModule({
...
providers: [
{
provide: NgLocalization,
useClass: NgLocaleLocalization},
],
})
export class CommonModule {}
@NgModule({
imports: [EagerModule2]
providers: [MyService]
})
export class EagerModule {}
EagerModule2
@NgModule({
providers: [MyService2]
})

Top part of Angular Injector Tree

This is not the entire tree. For now, there aren't any components here. We'll continue drawing later. But now let's start with AppModule Injector since it's most used part of angular.

Root AppModule Injector

Well known angular application root injector is presented as **AppModule Injector** in the picture above. As it has already been said, this injector collects all providers from transitive modules. It means that:

> *If we have a module with some providers and import this module directly in* `AppModule` *or in any other module, which has already been imported in* `AppModule` *, then those providers become application-wide providers.*

According to this rule, `MyService2` from `EagerModule2` will be included into the root injector.

ComponentFactoryResolver is also added to the root module injector by Angular. This resolver is responsible for dynamic creation components since it stores factories of `entryComponents` .

It is also worth noting that among all other providers we can see **Module Tokens** which are actually types of all merged NgModules. We will come back to this later when will be exploring tree-shakeable tokens.

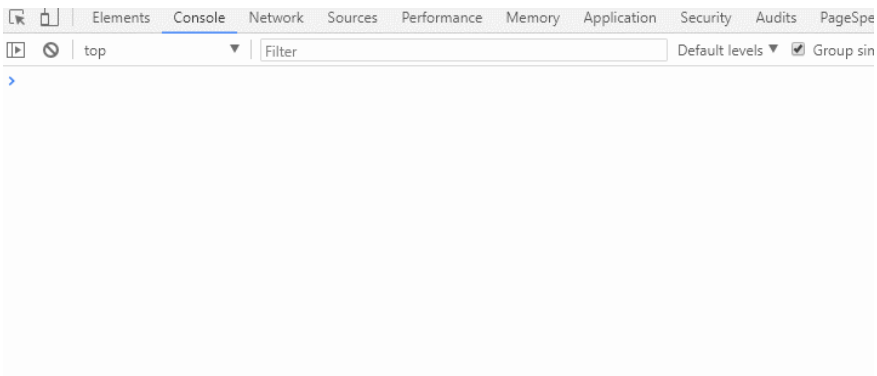
In order to initialize NgModule injector Angular uses `AppModule` factory, which is located in so-called `module.ngfactory.js` file.

AppModule factory

We can see that the factory returns the module definition with all merged providers. It should be well known by many developers.

> ***Tip:*** *If you have angular application in dev mode and want to see all providers from root AppModule injector then just open devtools console and write:*

```
ng.probe(getAllAngularRootElements()[0]).injector.view.
```



There are also a lot of well known facts which I won't describe here because they are well covered in angular documentation:

- https://angular.io/guide/ngmodule-faq
- https://angular.io/guide/hierarchical-dependency-injection

. . .

Platform Injector

As it turned out, the `AppModule` root injector has a parent **NgZoneInjector** , which is a child of **PlatformInjector**.

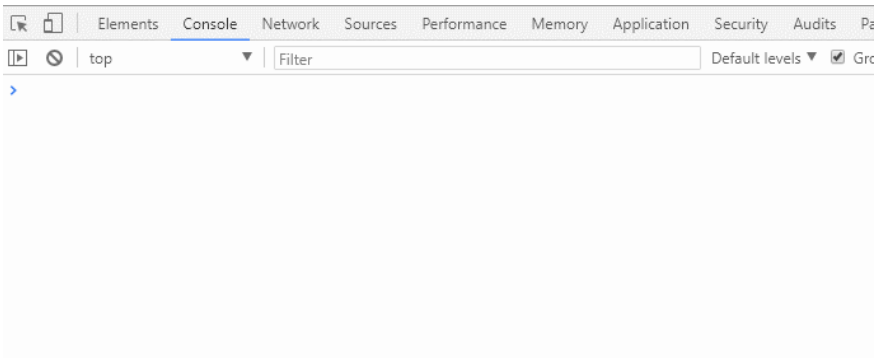
Platform injector usually includes built-in providers but we can provide our own when creating platform:

```
const platform = platformBrowserDynamic([ {
  provide: SharedService,
  deps:[]
}]);
platform.bootstrapModule(AppModule);
```

Extra providers, which we can pass to the platform, must be **StaticProviders**. If you're not familiar with the difference between `StaticProvider` and `Provider` , then follow this SO answer .

> ***Tip:*** *If you have angular application in dev mode and want to see all providers from Platform injector then just open devtools console and write:*

```
ng.probe(getAllAngularRootElements()[0]).injector.view.

// to see stringified value use
ng.probe(getAllAngularRootElements()[0]).injector.view.
```



Even though it's quite clear how angular resolves dependency on AppModule injector level and higher, I found out that it is very confusing thing on a components level. So I started my investigation.

EntryComponent and RootData

When I was talking about ComponentFactoryResolver, I mentioned entryComponents. These types of components are usually passed either in `bootstrap` or `entryComponents` array of NgModule. Angular router also creates component dynamically.

Angular creates host factories for all entryComponents and they are **root views** for all others. This means that:

> *Every time we create dynamic component angular creates **root view** with **root data,** that contains references to **elInjector** and **ngModule injector.***

```
function createRootData(
    elInjector: Injector, ngModule: NgModuleRef<any>,
    projectableNodes: any[][], rootSelectorOrNode: any
  const sanitizer = ngModule.injector.get(Sanitizer);
  const errorHandler = ngModule.injector.get(ErrorHand
  const renderer = rendererFactory.createRenderer(null
  return {
    ngModule,
    injector: elInjector, projectableNodes,
```

Now assume we run an angular application.

What happens when the following code is being executed?

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

In fact, a lot of things occur in the background, but we are interested in the part where angular creates entry component.

```
1   const compRef = componentFactory.create(Injector.NULL,
```

That's the place where angular **injector tree is bifurcated into parallel trees**.

Element Injector vs Module Injector

Some time ago, when lazy loaded modules started to be widely used, one strange behavior was reported on github: dependency injection system caused doubled instantiation of lazy loaded modules. As a result, a new design was introduced. So, starting from that moment we've had two parallel trees: one for elements and other for modules.

The main rule here is that:

When we ask some dependency in component or in directive angular uses **Merge Injector** to go through **element injector** tree and then, if dependency won't be found, switch to module injector tree to resolve dependency.

> ***Please note*** *I don't use phrase "component injector" but rather "element injector".*

What is the Merge Injector?

Have you ever written such a code?

```
1   @Directive({
2     selector: '[someDir]'
3   }
4   export class SomeDirective {
5    constructor(private injector: Injector) {}
```

So, the injector here is a merge injector (Similarly, we can inject Merge injector in component constructor).

Merge injector has the following definition:

```
class Injector_ implements Injector {
  constructor(private view: ViewData, private elDef: N
  get(token: any, notFoundValue: any = Injector.THROW_
    const allowPrivateServices =
        this.elDef ? (this.elDef.flags & NodeFlags.Com
    return Services.resolveDep(
        this.view, this.elDef, allowPrivateServices,
```

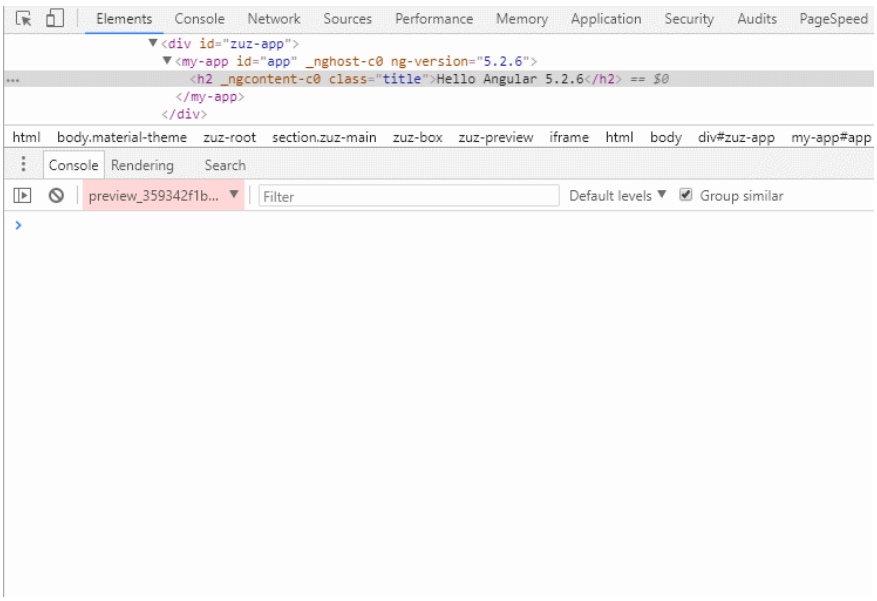
As we can see in the preceding code Merge injector is just combination of **view** and **element definition**. This injector works like a bridge between element injector tree and module injector tree when angular resolves dependencies.

Merge injector can also resolve such built-in things as `ElementRef`, `ViewContainerRef`, `TemplateRef`, `ChangeDetectorRef` etc. And more interestingly, it can return merge injector.

Basically every element can have merge injector even if you didn't provide any token on it.

> ***Tip:*** *to get merge injector just open console and write:*

```
ng.probe($0).injector
```



But you may ask what is the element injector then?

As we all know angular parses template to create factory with view definition. View is just representation of template, which contains different types of nodes such as `directive`, `text`, `provider`, `query` etc. And among others there is **element node**. Actually, the element injector resides on this node. Angular keeps all information about providers on an element node with the following properties:

```
export interface ElementDef {
  ...
  /**
   * visible public providers for DI in the view,
   * as see from this element. This does not include p
   */
  publicProviders: {[tokenKey: string]: NodeDef}|null;
  /**
   * same as visiblePublicProviders, but also includes
```

Let's see how element injector resolves dependency:

```
const providerDef =
  (allowPrivateServices ? elDef.element!.allProviders
    elDef.element!.publicProviders)![tokenKey];
if (providerDef) {
  let providerData = asProviderData(searchView, provid
  if (!providerData) {
    providerData = { instance: _createProviderInstance
    searchView.nodes[providerDef.nodeIndex] = provider
```

It's just checks `allProviders` or `publicProviders` properties depending on a privacy.

This injector contains the component/directive instance and all the providers registered by the component or directives.

These providers are filled during view instantiation, but the main source comes from ProviderElementContext, which is a part of Angular compiler. If we'll dive deep into this class, we can find there some interesting things.

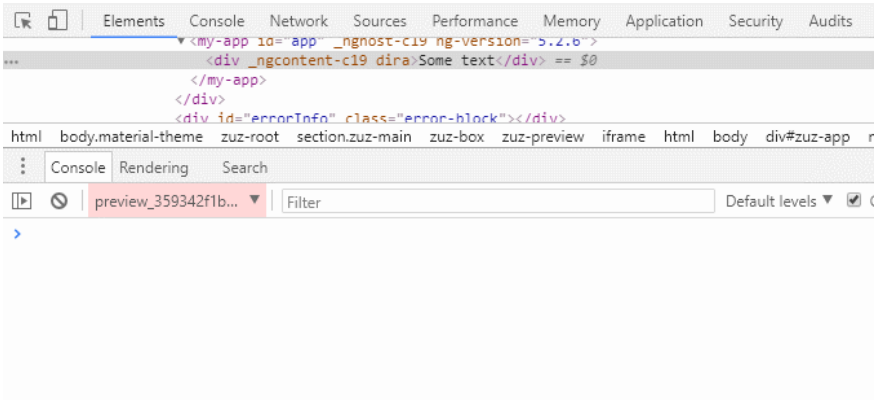
For example, Angular has some restriction when using Host decorator. `viewProviders` on the host element might help here. (See

also https://medium.com/@a.yurich.zuev/angular-nested-template-driven-form-4a3de2042475).

Another case is that if we have element with component and directive applied on it, and we provide the same token on the component and on the directive, then directive's provider wins.

> ***Tip:*** *to get element injector just open console and write:*

```
1  ng.probe($0).injector.elDef.element
```



Resolution algorithm

The code that describes Angular dependency resolution algorithm within view can be found here. And that's exactly what merge injector uses in `get` method (`Services.resolveDep`). To understand how dependency resolution algorithm work we need to be familiar with concepts of view and view parent element.

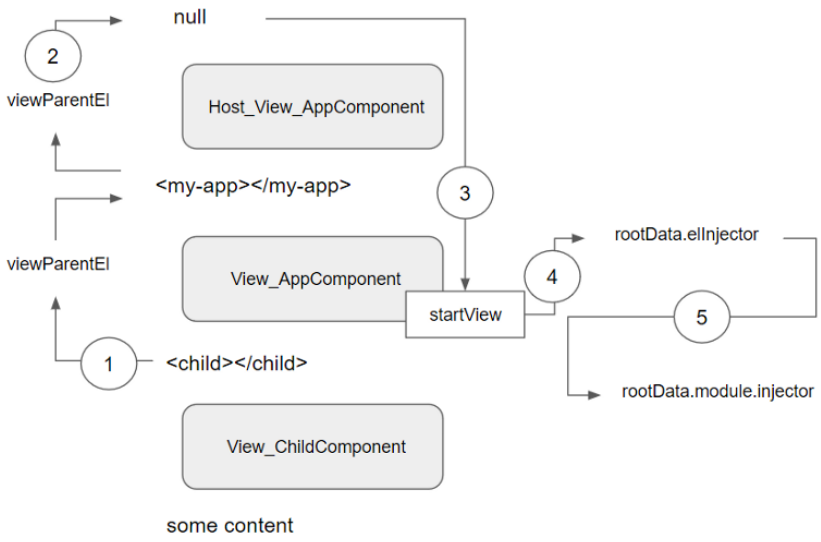
If we have root AppComponent with template `<child></child>` , then we have three views:

```
HostView_AppComponent
    <my-app></my-app>
```

```
View_AppComponent
    <child></child>
```

```
View_ChildComponent
    some content
```

The resolution algorithm is based on view hierarhy:



If we ask for some token in child component it will first look at child element injector, where checks `elRef.element.allProviders|publicProviders` , then goes up through all **parent view elements**(1) and also checks providers in element injector. If the next parent view element equals null(2) then it returns to **startView**(3), checks **startView.rootData.elnjector**(4) and only then, if token won't be found, checks **startView.rootData module.injector**(5).

That is, Angular searches for **parent element of particular view not for parent of particular element** when walking up through components to resolve some dependency. To get **view parent element** Angular uses the following function:

```
/**
 * for component views, this is the host element.
 * for embedded views, this is the index of the parent
 * that contains the view container.
 */
export function viewParentEl(view: ViewData): NodeDef|
  const parentView = view.parent;
  if (parentView) {
    return view.parentNodeDef !.parent;
```

For instance, let's imagine the following small angular app:

```
@Component({
  selector: 'my-app',
  template: `<my-list></my-list>`
})
export class AppComponent {}

@Component({
  selector: 'my-list',
  template: `
    <div class="container">
      <grid-list>
        <grid-tile>1</grid-tile>
        <grid-tile>2</grid-tile>
        <grid-tile>3</grid-tile>
      </grid-list>
    </div>
  `
})
export class MyListComponent {}

@Component({
  selector: 'grid-list',
```

Assume that we are inside `grid-tile` component and asking for `GridListComponent.` We will be able to get that component instance successfully. But how?

What's the view parent element at this point?

Here are the steps, I follow, to answer this question:

1. Find **starting element**. Our `GridTileComponent` has `grid-tile` element selector, therefore we need to find element that matches `grid-tile` selector. It's `grid-tile` element.
2. Find **template,** which `grid-tile` element belongs to (`MyListComponent` template).
3. Determine **view** for this element. If it has't any parent embedded view then it is component view otherwise it's embedded view. (We don't have any `ng-template` or `*structuralDirective` above `grid-tile` element so it's `View_MyListComponent` in our case).

4. Find **view parent element**. That is, **parent element for view not for element.**

There are two cases here:

- For embedded view this is the parent node, that contains the view container.

For instance, let's imagine we applied a structural directive on `grid-list`:

```
@Component({
  selector: 'my-list',
  template: `
    <div class="container">
      <grid-list *ngIf="1">
        <grid-tile>1</grid-tile>
        <grid-tile>2</grid-tile>
        <grid-tile>3</grid-tile>
      </grid-list>
```

View parent element for `grid-tile` will be `div.container` in this case.

- For component view this is the host element

This is what we have in our original small application. So view parent element will be `my-list` element not `grid-list`.

Now, you may wonder how angular can resolve `GridListComponent` if it bypassed `grid-list` ?

The key to understanding this is how angular collects providers for elements: **it uses prototypical inheritance**.

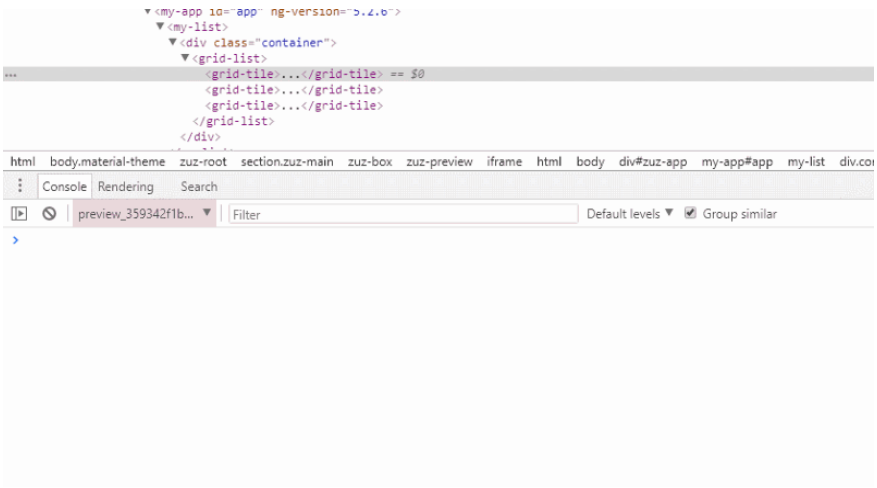
> *Each time we provide any token on an element, angular creates new `allProviders` and `publicProviders` array inherited from parent node, otherwise it just shares the same array with parent node.*

It means that `grid-tile` has already known about all providers that were registered on all parent elements within current view.

Basically, here's how angular collects providers for elements within template:

```
<div class="container">          []
  <grid-list>                    [GridListComponent] + []
    <grid-tile>1</grid-tile>     [GridTileComponent] + [GridListComponent]
    ...
  </grid-list>
</div>


<div class="container">          []
  <grid-list>                    [GridListComponent]
    <grid-tile>1</grid-tile>     [GridTileComponent, GridListComponent]
    ...
  </grid-list>
</div>
```

As we can see above, `grid-tile` can successfully get `GridListComponent` from its element injector through `allProviders` because grid-tile element injector contains providers from parent element.



More on this here in this SO answer.

Prototypical inheritance providers on elements is one of the reason why we can't use `multi` option to provide token on multiple levels. But since dependency injection is very flexible system there is a way to workaround it.
https://stackoverflow.com/questions/49406615/is-there-a-way-how-to-use-angular-multi-providers-from-all-multiple-levels

With all this in mind, it's time to continue drawing our injector tree.

Simple my-app->child->grand-child application

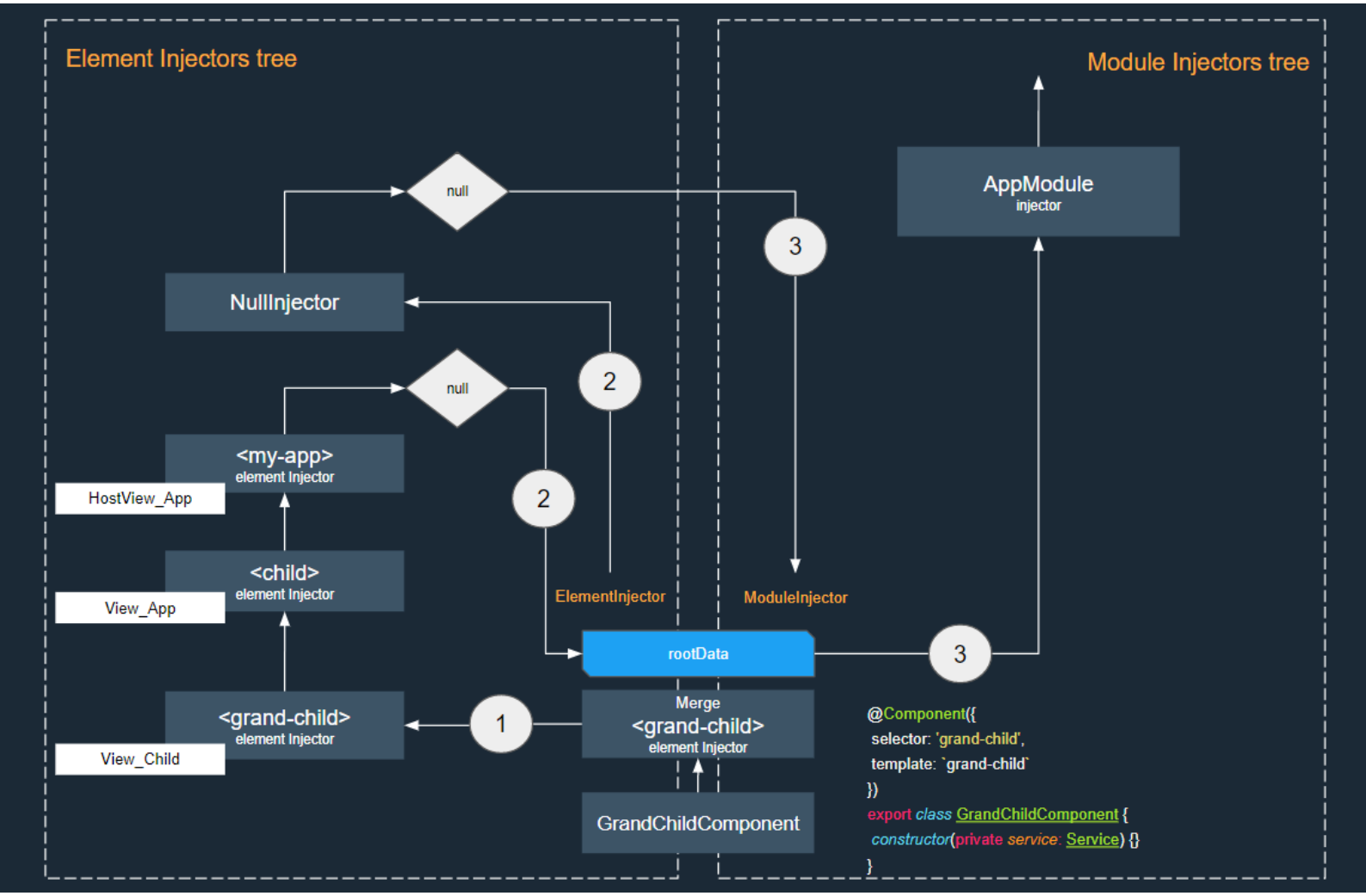
Let's consider the following simple application:

```
@Component({
  selector: 'my-app',
  template: `<child></child>`,
})
export class AppComponent {}

@Component({
  selector: 'child',
  template: `<grand-child></grand-child>`
})
export class ChildComponent {}

@Component({
  selector: 'grand-child',
  template: `grand-child`
})
export class GrandChildComponent {
  constructor(private service: Service) {}
}

```

We have tree levels of components and ask `Service` in `GrandChildComponent`.

```
my-app
   child
      grand-child(ask for Service dependency)
```

Here is how angular will resolve `Service` dependency.



In the picture above we start with `grand-child` element, which is located on `View_Child` (1). Angular will walk up through all view parent elements. When there is no view parent element(in our case `my-app` doesn't have any view parent elements) it first looks at the root `elInjector` (2):

```
1   startView.root.injector.get(depDef.token, NOT_FOUND_CHE
```

`startView.root.injector` is a `NullInjector` in our case. Since `NullInjector` doesn't keep any tokens, the next step is to switch to the module injector (3):

```
1   startView.root.ngModule.injector.get(depDef.token, notF
```

So now angular will attempt to resolve dependency the following way:

```
AppModule Injector
        ||
        \/
    ZoneInjector
        ||
        \/
  Platform Injector
        ||
        \/
    NullInjector
        ||
        \/
       Error
```

. . .

Simple routed application

Let's modify our application and add router to `ChildComponent`.

```
@Component({
  selector: 'my-app',
  template: `<router-outlet></router-outlet>`,
})
export class AppComponent {}
...
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      { path: 'child', component: ChildComponent },
      { path: '', redirectTo: '/child', pathMatch: 'fu
    ])
  ],
  declarations: [
```

After that we have something like:

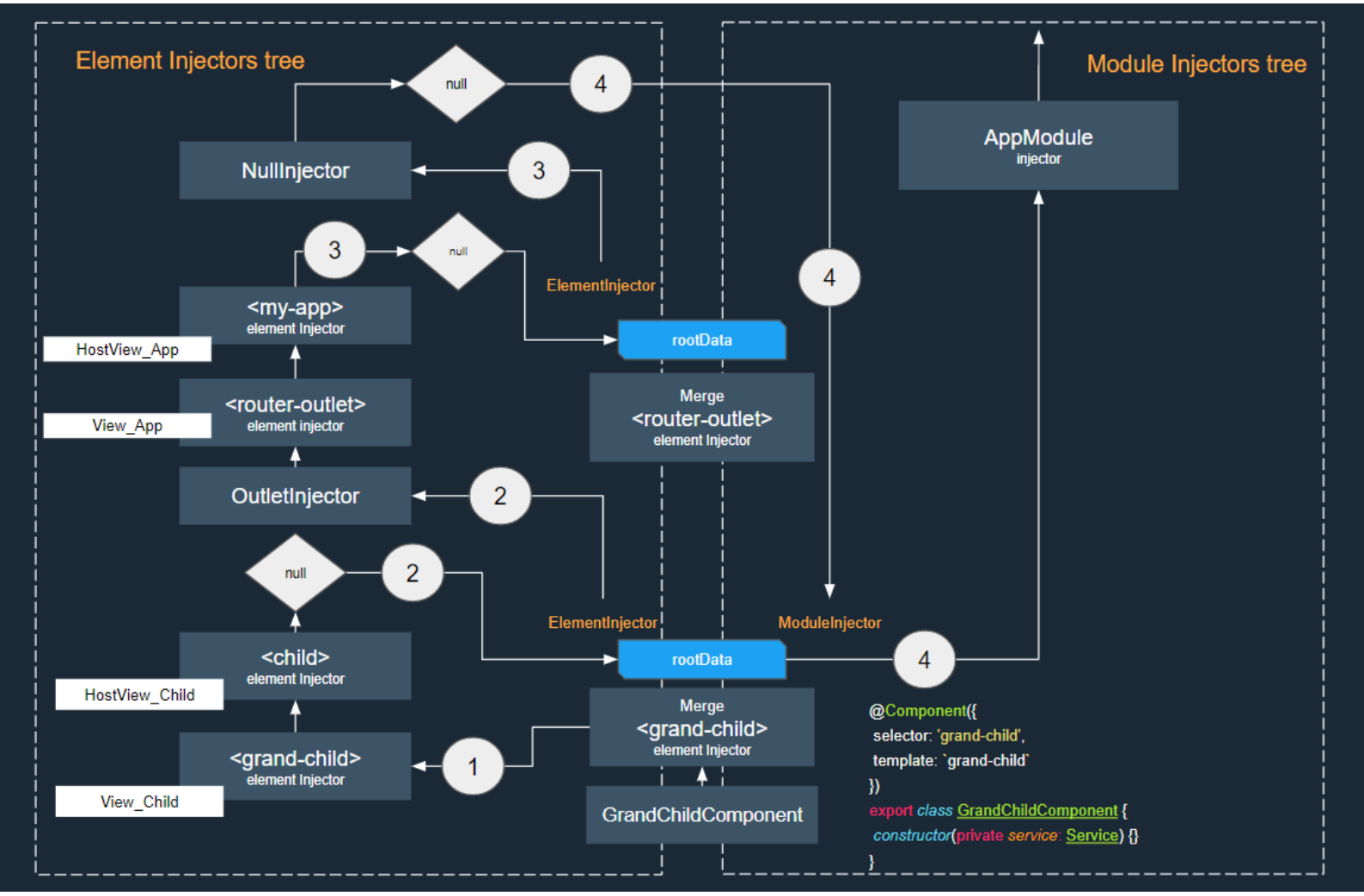
```
my-app
   router-outlet
   child
      grand-child(dynamic creation)
```

Now, let's look at the place where router creates dynamic components:

```
1   const injector = new OutletInjector(activatedRoute, chi
2   this.activated = this.location.createComponent(factory,
```

At this point angular creates a new root view with new **rootData** object. We can see that angular passes **OutletInjector** as root `elInjector`. `OutletInjector` is created with parent `this.location.injector` which is the injector for `router-outlet` element.

OutletInjector is a special kind of injector, which acts like reference between routed component and parent `router-outlet` element and can be found here.



• • •

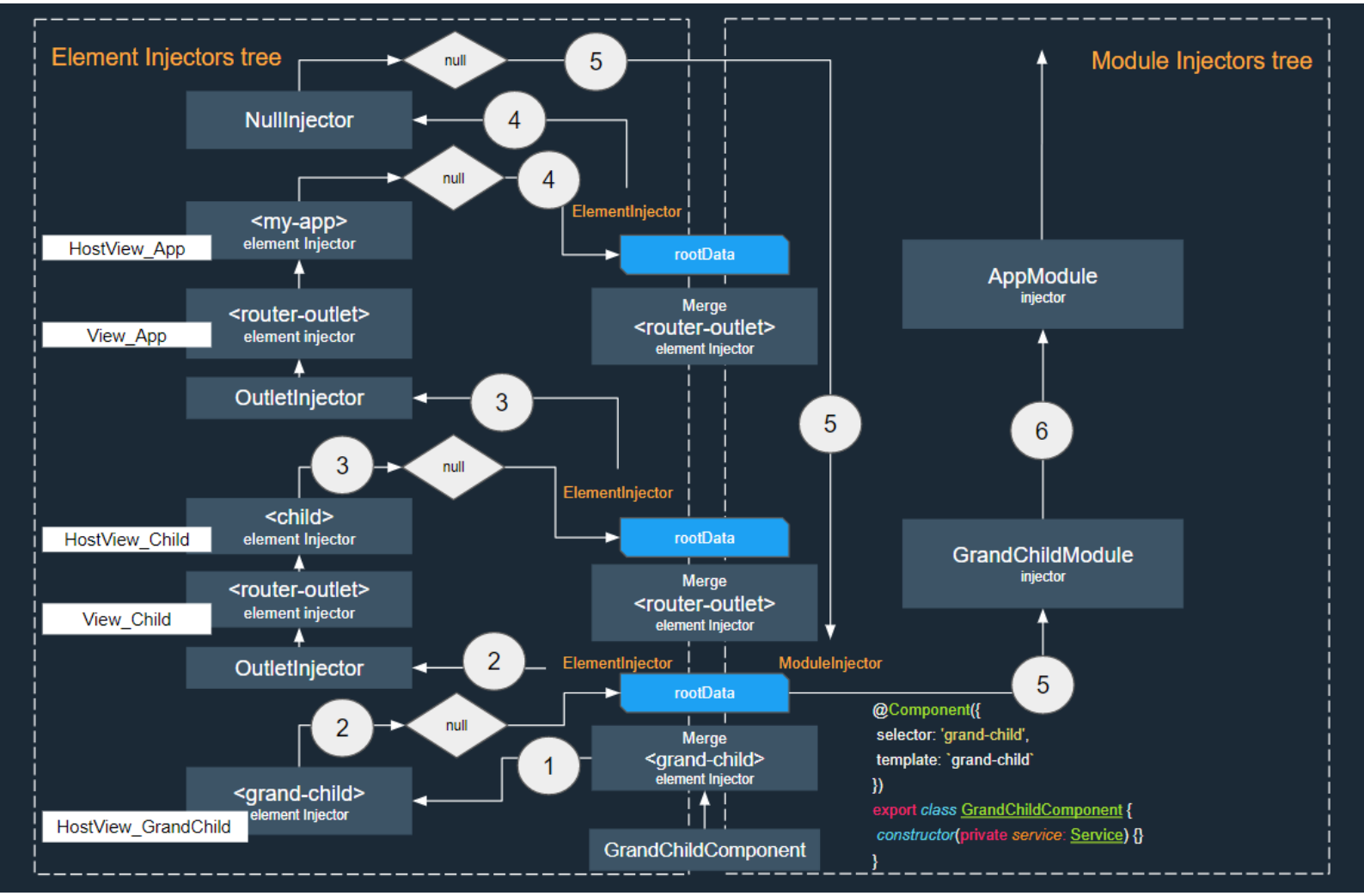
Simple application with lazy loading

Finally, let's move `GrandChildComponent` to lazy loaded module. To do that we need to add `router-outlet` to the child component view and change router configuration as shown below:

```
@Component({
  selector: 'child',
  template: `
    Child
    <router-outlet></router-outlet>
  `
})
export class ChildComponent {}
...
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {
        path: 'child', component: ChildComponent,
        children: [
          {
             path: 'grand-child',
             loadChildren: './grand-child/grand-child.
        ]
      }
```

```
my-app
   router-outlet
   child (dynamic creation)
       router-outlet
         +grand-child(lazy loading)
```

Now let's draw two separate trees for our application with lazy loading:



. . .

Tree-shakeable tokens are on horizon

Angular continues working on making framework smaller and since **version 6** it is going to support another way of registering providers.

Injectable

Before, a class with `Injectable` decorator didn't indicate that it could have dependency, it was not related to how it would be used in other parts. So, if a service does not have any dependency, `@Injectable()` can be removed without causing any issue.

As soon as API becomes stable, we can configure Injectable decorator to tell angular which module it belongs to and how it should be instantiated:

```
export interface InjectableDecorator {
  (): any;
  (options?: {providedIn: Type<any>| 'root' | null}&Inj
  new (): Injectable;
  new (options?: {providedIn: Type<any>| 'root' | null}
}

```

Here's an simple example of how we can use it:

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

@Injectable({
  providedIn: 'root',
  useClass: MyService,
```

This way, instead of including all providers in ngModule factory angular stores information about provider in Injectable metadata. That's what we need to make our libraries smaller. If we use Injectable to register providers and consumers don't import our providers then it won't be included into final bundle. So,

> *Prefer registering providers in Injectables over NgModule.providers over Component.providers*

Early I mentioned Modules Tokens, which are added to the root module injector. So angular can distinguish which modules are presented in a particular module injector.

Resolver uses this information to check whether tree-shakeable token belongs to the module injector.

InjectionToken

In case of InjectionToken we also will be able to define how a token will be constructed by the DI system and in which injectors it will be available.

```
export class InjectionToken<T> {
  constructor(protected _desc: string, options?: {
    providedIn?: Type<any>| 'root' | null,
    factory: () => T
  }) {}
```

So it's supposed to be used as follows:

```
export const apiUrl = new InjectionToken('tree-shakeabl
  providedIn: 'root',
  factory: () => 'someUrl'
});
```

Conclusion

Dependency injection model is quite complex topic in angular. Knowing how it works internally makes you confident in what you do. So I strongly suggest you looking into angular source code from time to time...

. . .