**Dependency Injection**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Reference:** [**https://angular.io/guide/dependency-injection**](https://angular.io/guide/dependency-injection)

**DI, is a design pattern in which a class requests dependencies from external sources rather than creating them itself.**

**@Injector() => It is responsible for creation of a class instance and inject into it constructor of the object.**

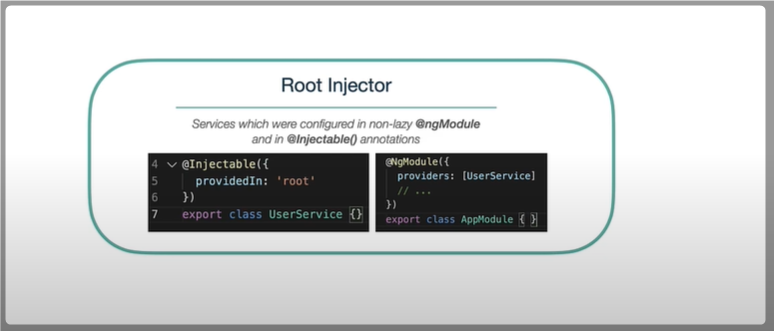
**There are two injector hierarchies in Angular:**

1. **Module Injector hierarchy**
2. **Element Injector hierarchy**

**Module Injector hierarchy:**

**=========================**

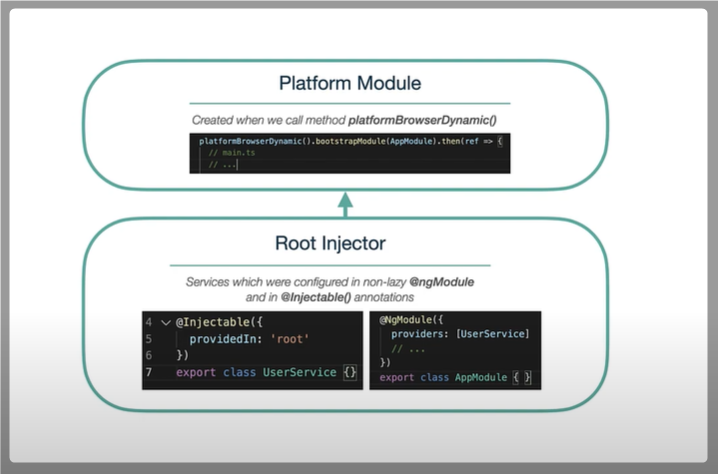
**When Angular starts it creates a Root Injector where will be registered our services which we provided via Injectable Annotations(Provided in Root) and All services provided in providers property in NgModule, If these modules are not lazy loaded.**

****

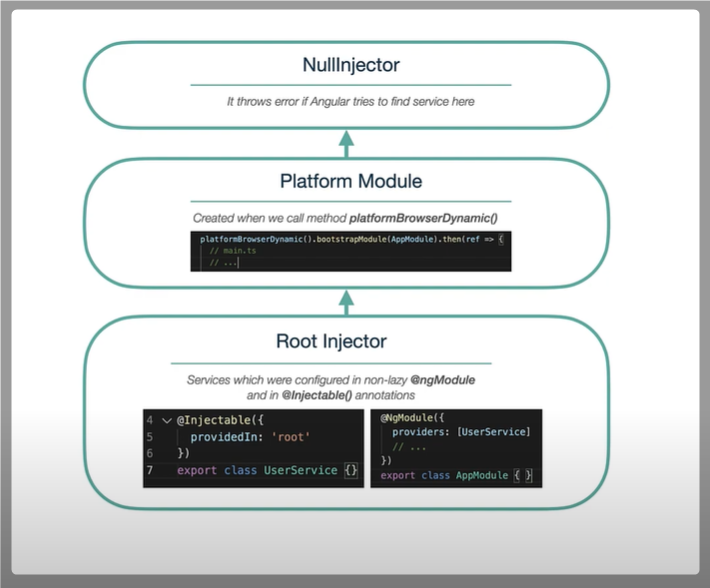
**Angular recursively goes through the all modules which are being used in the application and creates instances for provided services and register them into Root Injector.**

**So if you thought that if you provide some service in some module and it will be available in this module scope it is actually wrong,**

**If you provide some service in early loaded module it will be added to the Root Injector which makes it available to the whole application but this Root Injector is not the highest in this hierarchy during application bootstrapping angular creates few more injectors.**

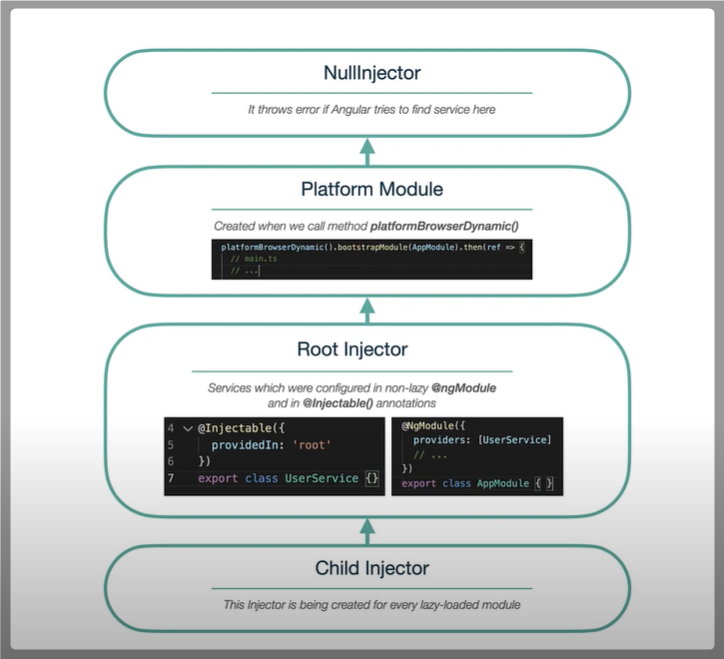
****

**Above the Root Injector we have Platform injector, this one is created by platformBrowserDynamic function inside main.ts and it will provide some platform specific features like Dom Sanitizer, PlatformIdToken ..etc.**

****

**On top of this we have Null Injector, the responsibility of this injector is simply throws the error if any dependency is not available.**

**Another modules which we import in our appModule they are not creating new child Injectors, Providers from them are going to Root Injector but it is only non lazy Modules.**

****

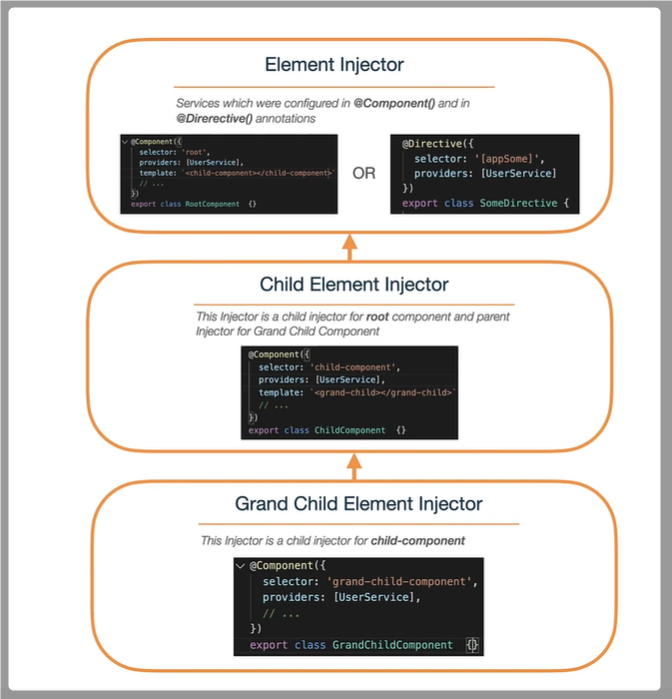
**But Modules which were loaded lazily creates own injector which are child injectors for our Root Injector and few years ago it could be a problem if you wanted to have a singleton service across the application bcz these child injectors creates a new service instance so you would have multiple instances for the same service this problem is resolved by ProvidedIn ‘Root’.**

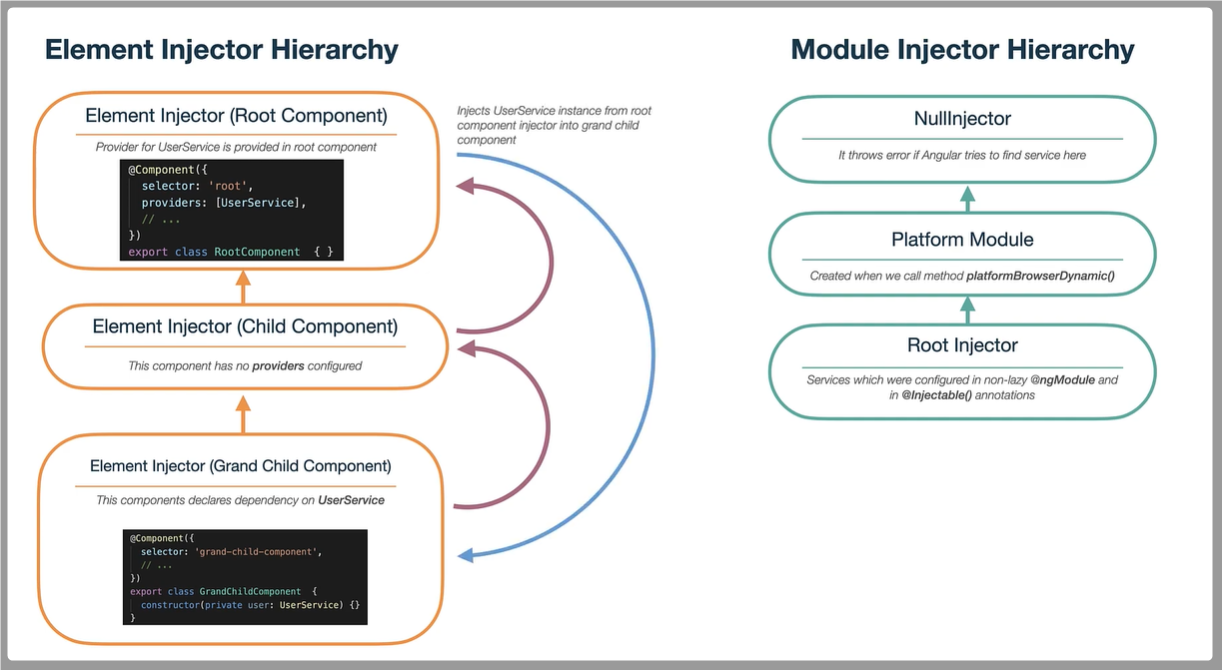
**Element Injector hierarchy :**

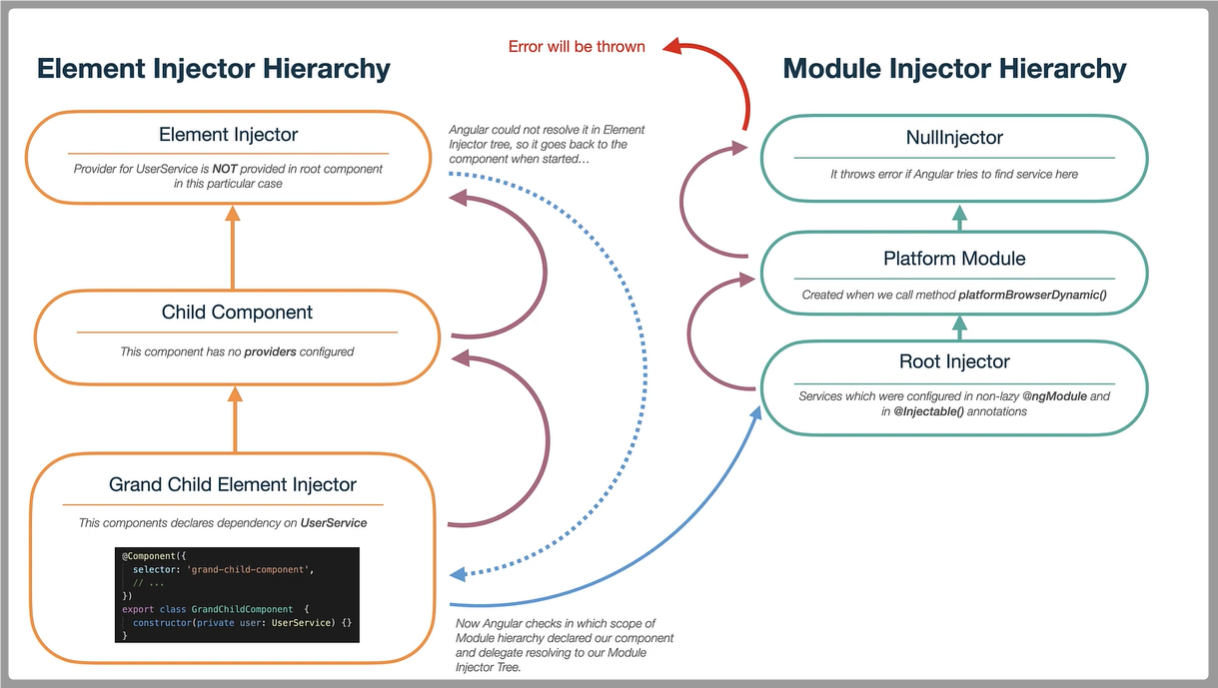
**=========================**

**This is created implicitly at each DOM element. An element injector is empty by default unless you configure it in the providers property on Component or Directive level.**

****

****

****

****

**Resolution Modifiers:**

**====================**

**Angular’s DI resolution can be controlled with @Optional(), @Self(), @SelfSkip() and @Host().**

1. **@Optional() => this allows the injected service as optional. If instance not found it does not throw any error.**
2. **@Self() => Angular will only look at the ElementInjector for the current component or directive(i.e Providers property).**

**(it creates its own instance results multiple instance for the same service if it is injected anywhere in the application)**

1. **@SkipSelf() => Angular starts its search for a service in the parent ElementInjector and above, rather than in the current one.**
2. **@Host() => Every Angular component has its own Host Element(selector), Host resolution modifier works for Host element should be the last stop for searching for Injector.**

**Dependency Providers:**

**=====================**

**This refers how instance creation will be done in Angular DI process.**

1. **useClass => simply creates a new object with the provided class**
2. **useExisting => it uses the existing object instead creating new one based on the Service ProvidedIn property.**
3. **useValue => it needs an object, use cases like config object or injection token.**
4. **useFactory => this provider is very convenient if you don’t know which service to provide in advance and this could be evaluated only at run time.**

**Multiple Providers:**

**==================**

Usually, when we register multiple providers with the same token, the last one wins.

Using multi: true tells Angular that the provider is a multi provider. As mentioned earlier, with multi providers, we can provide multiple values for a single token in DI. That’s exactly what we’re doing. We have two providers, both have the same token but they provide different values. If we ask for a dependency for that token, what we get is a list of all registered and provided values.

With multi providers, we can basically provide **multiple dependencies for a single token.**

This means, with multi providers we can basically **extend** the thing that is being injected for a particular token. Angular uses this mechanism to provide pluggable hooks.

Real use cases like :

1. **when we using** NG\_VALIDATORS
2. **Interceptors**