Dependency Injection (DI): Documentation (Advanced Topics)

This document talks about advanced topics related to the DI module and how it is used in Angular. You don't have to know this to use DI in Angular or independently.

Key

Most of the time we do not have to deal with keys.

```
var inj = Injector.resolveAndCreate([
  bind(Engine).toFactory(() => new TurboEngine())  //the passed in token Engine gets mapped
]);
var engine = inj.get(Engine); //the passed in token Engine gets mapped to a key
```

Now, the same example, but with keys.

```
var ENGINE_KEY = Key.get(Engine);

var inj = Injector.resolveAndCreate([
  bind(ENGINE_KEY).toFactory(() => new TurboEngine()) // no mapping
]);
var engine = inj.get(ENGINE_KEY);  // no mapping
```

Every key has an id, which we utilize to store providers and instances. So Injector uses keys internally for performance reasons.

ProtoInjector and Injector

Often there is a need to create multiple instances of essentially the same injector. In Angular, for example, every component element type gets an injector configured in the same way.

Doing the following would be very inefficient.

```
function createComponentInjector(parent, providers: Binding[]) {
  return parent.resolveAndCreateChild(providers);
}
```

This would require us to resolve and store providers for every single component instance. Instead, we want to resolve and store the providers for every component type, and create a set of instances for every component. To enable that DI separates the meta information about injectables (providers and their dependencies), which are stored in `ProtoInjector`, and injectables themselves, which are stored in `Injector`.

```
var proto = new ProtoInjector(providers); // done once
function createComponentInjector(parent, proto) {
  return new Injector(proto, parent);
}
```

`Injector.resolveAndCreate` creates both a `ProtoInjector` and an `Injector`.

Host & Visibility

An injector can have a parent. The parent relationship can be marked as a "host" as follows:

```
var child = new Injector(proto, parent, true /* host */);
```

Hosts are used to constraint dependency resolution. For instance, in the following example, DI will stop looking for `Engine` after reaching the host.

```
class Car {
  constructor(@Host() e: Engine) {}
}
```

Imagine the following scenario:

```
    ParentInjector
       /   \
      /     \ host
   Child1    Child2
```

Here both Child1 and Child2 are children of ParentInjector. Child2 marks this relationship as host. ParentInjector might want to expose two different sets of providers for its "regular" children and its "host" children. providers visible to "regular" children are called "public" and providers visible to "host" children are called "private". This is an advanced use case used by Angular, where components can provide different sets of providers for their children and their view.

Let's look at this example.

```
class Car {
  constructor(@Host() e: Engine) {}
}

var parentProto = new ProtoInjector([
  new BindingWithVisibility(Engine, Visibility.Public),
  new BindingWithVisibility(Car, Visibility.Public)
]);
var parent = new Injector(parentProto);

var hostChildProto = new ProtoInjector([new BindingWithVisibility(Car, Visibility.Public)])
var hostChild = new Injector(hostChildProto, parent, true);

var regularChildProto = new ProtoInjector([new BindingWithVisibility(Car, Visibility.Public
var regularChild = new Injector(regularChildProto, parent, false);
```

```
hostChild.get(Car); // will throw because public dependencies declared at the host cannot be
parent.get(Car); // this works
regularChild.get(Car); // this works
```

Now, let's mark Engine as private:

```
class Car {
  constructor(@Host() e: Engine) {}
}

var parentProto = new ProtoInjector([
  new BindingWithVisibility(Engine, Visibility.Private),
  new BindingWithVisibility(Car, Visibility.Public)
]);
var parent = new Injector(parentProto);

var hostChildProto = new ProtoInjector([new BindingWithVisibility(Car, Visibility.Public)]);
var hostChild = new Injector(hostChildProto, parent, true);

var regularChildProto = new ProtoInjector([new BindingWithVisibility(Car, Visibility.Public)
var regularChild = new Injector(regularChildProto, parent, false);

hostChild.get(Car); // this works
parent.get(Car); // this throws
regularChild.get(Car); // this throws
```

Now, let's mark Engine as both public and private:

```
class Car {
  constructor(@Host() e: Engine) {}
}

var parentProto = new ProtoInjector([
  new BindingWithVisibility(Engine, Visibility.PublicAndPrivate),
  new BindingWithVisibility(Car, Visibility.Public)
]);
var parent = new Injector(parentProto);

var hostChildProto = new ProtoInjector([new BindingWithVisibility(Car, Visibility.Public)]);
var hostChild = new Injector(hostChildProto, parent, true);

var regularChildProto = new ProtoInjector([new BindingWithVisibility(Car, Visibility.Public)
var regularChild = new Injector(regularChildProto, parent, false);

hostChild.get(Car); // this works
parent.get(Car); // this works
regularChild.get(Car); // this works
```

Angular and DI

Now let's see how Angular uses DI behind the scenes.

The right mental model is to think that every DOM element has an Injector. (In practice, only interesting elements containing directives will have an injector, but this is a performance optimization)

There are two properties that can be used to configure DI: providers and viewProviders.

- `providers` affects the element and its children.
- `viewProviders` affects the component's view.

Every directive can declare injectables via `providers`, but only components can declare `viewProviders`.

Let's look at a complex example that shows how the injector tree gets created.

```
<my-component my-directive>
  <needs-service></needs-service>
</my-component>
```

Both `MyComponent` and `MyDirective` are created on the same element.

```
@Component({
  selector: 'my-component',
  providers: [
    bind('componentService').toValue('Host_MyComponentService')
  ],
  viewProviders: [
    bind('viewService').toValue('View_MyComponentService')
  ],
  template: `<needs-view-service></needs-view-service>`,
  directives: [NeedsViewService]
})
class MyComponent {}

@Directive({
  selector: '[my-directive]',
  providers: [
    bind('directiveService').toValue('MyDirectiveService')
  ]
})
class MyDirective {
}
```

`NeedsService` and `NeedsViewService` look like this:

```
@Directive({
  selector: 'needs-view-service'
})
class NeedsViewService {
  constructor(@Host() @Inject('viewService') viewService) {}
}

@Directive({
  selector: 'needs-service'
})
class NeedsService {
  constructor(@Host() @Inject('componentService') service1,
              @Host() @Inject('directiveService') service2) {}
}
```

This will create the following injector tree.

```
      Injector1 [
        {binding: MyComponent,        visibility: Visibility.PublicAndPrivate},
        {binding: 'componentService', visibility: Visibility.PublicAndPrivate},
        {binding: 'viewService',      visibility: Visibility.Private},
        {binding: MyDirective         visibility: Visibility.Public},
        {binding: 'directiveService', visibility: Visibility.Public}
      ]
  /                                                        \
  |                                                         \ host
Injector2 [                                                  Injector3 [
  {binding: NeedsService, visibility: Visibility.Public}            {binding: NeedsViewServic
]                                                            ]
```

As you can see the component and its providers can be seen by its children and its view. The view providers can be seen only by the view. And the providers of other directives can be seen only their children.