## Provider

**Learning Objectives**

* Know how we can configure injectors with providers.
* Know the 4 different types of dependencies we can configure provider to provide.

**Providers**

As mentioned in the previous lecture we can *configure* injectors with *providers* and a provider links a *token* to a *dependency*.

But so far we seem to be configuring our injector with just a list of classes, like so:

let injector = ReflectiveInjector.resolveAndCreate([

MandrillService,

SendGridService

])

;

The above code is in fact a shortcut for:

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: MandrillService, useClass: MandrillService },

{ provide: SendGridService, useClass: SendGridService },

]);

The real configuration for a provider is an object which describes a *token* and configuration for how to create the associated *dependency*.

The provide property is the *token* and can either be a *type*, a *string* or an instance of something called an InjectionToken.

The other properties of the provider configuration object depend on the *kind* of dependency we are configuring, since we are configuring classes in this instance we the useClass property.

 If all we want to do is providing classes, it’s such a common pattern that Angularlets you define a list of class names as the providers.

### Switching dependencies

The above is an excellent example of how we can use the DI framework to architect our application for *ease of re-use*, *ease of testing* and less *brittle* code.

If we wanted to *re-use* our application and move from Mandrill to SendGrid without using DI we would have to search through all the code for where we have requested MandrillService to be injected and replace with SendGridService.

A better solution is to configure the DI framework to return either MandrilService or SendGridService depending on the context, like so:

import { ReflectiveInjector } from '@angular/core';

class MandrillService {};

class SendGridService {};

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: "EmailService", useClass: MandrillService }

①

])

;

let emailService = injector.get("EmailService");

console.log(emailService); // new MandrillService()

① The token is "EmailService" and the dependency is the class MandrillService

The above is configured so when code requests the token "EmailService" it returns an instance of the class MandrillService.

To switch to using the SendGridService throughout our application we can just configure our injector with a different provider, like so:

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: "EmailService", useClass: SendGridService }

①

])

;

① The token is "EmailService" and the dependency is the class SendGridService

Now the DI framework just returns instances of SendGridService whenever the token "EmailService" is requested.

### Provider configurations

So far we’ve only seen how we can configure a provider to provide classes however there are 4 types of dependencies providers can provide in Angular.

**useClass**

We can have a provider which maps a *token* to a *class*, like so:

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: Car, useClass: Car },

])

;

The above is so common that there is a shortcut, if all we want to provide is a class we can simply pass in the class name, like so:

let injector = ReflectiveInjector.resolveAndCreate([Car]);

**useExisting**

We can make two tokens map to the same *thing* via aliases, like so:

import { ReflectiveInjector } from '@angular/core';

class MandrillService {}; class SendGridService {}; class GenericEmailService {};

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: GenericEmailService, useClass: GenericEmailService }, ①

{ provide: MandrillService, useExisting: GenericEmailService }, ②

{ provide: SendGridService, useExisting: GenericEmailService } ③

]);

let emailService1 = injector.get(SendGridService); ④ console.log(emailService1); // GenericEmailService {} let emailService2 = injector.get(MandrillService); ⑤ console.log(emailService2); // GenericEmailService {} let emailService3 = injector.get(GenericEmailService); ⑥ console.log(emailService3); // GenericEmailService {} console.log(emailService1 === emailService2 === emailService3); // false ⑦

1. The token GenericEmailService resolves to an instance of GenericEmailService.
2. This provider maps the token MandrillService to whatever the existing GenericEmailService provider points to.
3. This provider maps the token SendGridService to whatever the existing GenericEmailService provider points to.
4. Requesting a resolve of SendGridService, MandrillService or GenericEmailService return an instance of GenericEmailService.
5. All three instances of GenericEmailService returned are *the same instance*.

Whenever anyone requests MandrillService or SendGridService we return an instance of GenericEmailService instead.

**useValue**

We can also provide a simple *value*, like so:

import { ReflectiveInjector } from '@angular/core';

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: "APIKey", useValue: 'XYZ1234ABC' }

])

;

let apiKey = injector.get("APIKey");

console.log(apiKey); // "XYZ1234ABC"

Or if we wanted to pass in an object we can, like so:

import { ReflectiveInjector } from '@angular/core';

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: "Config",

useValue: {

'APIKey': 'XYZ1234ABC',

'APISecret': '555-123-111'

}

}

])

;

let config = injector.get("Config");

console.log(config); // Object {APIKey: "XYZ1234ABC", APISecret: "555-123-111"}

If the intention however is to pass around *read-only* constant values then passing an object is a problem since any code in your application will be able to *change* properties on that object. What Config points to *can’t* be changed but the properties of Config *can* be changed.

So when passing in an object that you intend to be *immutable* (unchanging over time) then use the Object.freeze method to stop client code from being able to *change* the config object, like so:

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: "Config",

useValue: Object.freeze({

①

'APIKey': 'XYZ1234ABC',

'APISecret': '555-123-111'

})

}

])

;

① By using Object.freeze that objects values can’t be changed, in effect it’s read-only.

**useFactory**

We can also configure a provider to call a function every-time a token is requested, leaving it to the provider to figure out *what* to return, like so:

① When the injector resolves to this provider, it calls the useFactory function and returns whatever is returned by this function as the dependency.

Just like other providers the result of the call is *cached*. So even though we are using a factory and creating an instance with new ourselves calling injector.get(EmailService) again will still return the *same* instance of MandrillService that was created with the first call.

let emailService1 = injector.get(EmailService); let emailService2 = injector.get(EmailService); console.log(emailService2 === emailService2); // true ①

① Returns the same instance

import { ReflectiveInjector } from '@angular/core';

class MandrillService {};

class SendGridService {};

const isProd = true;

let injector = ReflectiveInjector.resolveAndCreate([

{

provide: "EmailService",

useFactory: () => {

①

if (isProd) {

return new MandrillService();

} else {

return new SendGridService();

}

}

},

])

;

let emailService1 = injector.get("EmailService");

console.log(emailService1); // MandrillService {}

**Summary**

We can configure providers to return 4 different kinds of dependencies: classes, values, aliases and factories.

In the next lecture we will look at the different ways we can define *tokens*.

**Listing**

*script.ts*

import {ReflectiveInjector} from '@angular/core'; import {OpaqueToken} from '@angular/core';

// Switching Dependencies Example

{

console.log("Switching Dependencies Example");

class MandrillService {

}

class SendGridService {

}

let injector = ReflectiveInjector.resolveAndCreate([

{provide: "EmailService", useClass: MandrillService}

]);

let emailService = injector.get("EmailService");

console.log(emailService); // new MandrillService()

{

let injector = ReflectiveInjector.resolveAndCreate([

{provide: "EmailService", useClass: SendGridService}

]);

let emailService = injector.get("EmailService");

console.log(emailService); // new SendGridService()

}

}

// useClass Provider

{

console.log("useClass");

class EmailService {

}

class MandrillService extends EmailService {

}

class SendGridService extends EmailService {

}

let injector = ReflectiveInjector.resolveAndCreate([

{provide: EmailService, useClass: SendGridService}

]);

let emailService = injector.get(EmailService);

console.log(emailService);

}

// useExisting

{

console.log("useExisting");

class MandrillService {

}

class SendGridService {

}

class GenericEmailService {

}

let injector = ReflectiveInjector.resolveAndCreate([

{provide: GenericEmailService, useClass: GenericEmailService},

{provide: MandrillService, useExisting: GenericEmailService},

{provide: SendGridService, useExisting: GenericEmailService}

]);

let emailService1 = injector.get(SendGridService);

console.log(emailService1); // GenericEmailService {}

let emailService2 = injector.get(MandrillService);

console.log(emailService2); // GenericEmailService {}

let emailService3 = injector.get(GenericEmailService);

console.log(emailService3); // GenericEmailService {}

console.log(emailService1 === emailService2 && emailService2 === emailService3); //

true

}

// useValue

{

console.log("useValue");

let injector = ReflectiveInjector.resolveAndCreate([

{

provide: "Config",

useValue: Object.freeze({

'APIKey': 'XYZ1234ABC',

'APISecret': '555-123-111'

})

}

]);

let config = injector.get("Config");

console.log(config); // Object {APIKey: "XYZ1234ABC", APISecret: "555-123-111"}

}

// useFactory

{

console.log("useFactory");

class MandrillService {

}

class SendGridService {

}

const isProd = true;

let injector = ReflectiveInjector.resolveAndCreate([

{

provide: "EmailService",

useFactory: () => {

if (isProd) {

return new MandrillService();

} else {

return new SendGridService();

}

}

},

]);

let emailService1 = injector.get("EmailService");

console.log(emailService1); // MandrillService {}

}