Dependency Injection in Angular 8

Dependency Injection

A hierarchy of injectors

- If what is sought is not in current injector, look at its parent

Platform is root of injector tree

- An important duty of the platform is setting the initial providers (e.g. which renderer to use)
- Look at *https://github.com/angular/angular/blob/master/packages/platform-browser/src/browser.ts*

Angular Comes with built-in DI

You should use it!!

- Mocking & testing
- Re-purposing
- Flexibility & extensibility
- Component and service composition

Advertising slogan For DI

- “Change you mind – not your code - later ”

What is DI?

Injection

- Something that comes from somewhere else in a controlled manner

Dependency

- A used component or service
- Dependencies are delivered (e.g. as function parameters) to where needed, not “new-ed” there

Why use DI?

Application Composition

How do we build up combinations of components and services

- Varying dependency needs for testing, different backends, differing client enviornments,
- Importantly, to achieve all this without changing stable application code

What is an Injector?

From Angular source code:

- "An `Injector` is a replacement for a `new` operator, which can automatically resolve the constructor dependencies." [link]
- What is an injectable object?
  - Providers
  - ViewProviders

What is an Injectable?

An injector injects an injectable

- The injectable is what gets injected into the consumer
- `export Injectable(options : InjectableMetadata) : InjectableMetadataFactory`

Good practice

- Add the `Injectable()` decorator to injectables
- Should add to ALL services
- Injectable actually needed for hierarchical injection

Hierarchical dependency injection

Hierarchy of components can have hierarchy of injectors

- Important to avoid unexpected duplicate injectors for same token

Types

Inject (const)

- Exported instance of InjectMetadataFactory

Injectable (decorator)

Injector (class)

Injector

Maps from tokens to instances

- ```
  class Injector {
    get(token: any, notFoundValue?: any) : any;
  }
  ```
- `get()` method takes a token and looks up injector for appropriate result
- Note token is `any`, as is return value
  ```

ReflectiveInjector

- resolve
- fromResolvedProviders
- resolveAndCreate
  [this calls `resolve` and then `fromResolvedProviders`, so where app needs multiple injected objects, more efficient for app to call `resolve` once, and `fromResolvedProviders` many times]
- get

Providers

A provider is used by an injector to instantiate something, based on a token

- Rich flexibility in how this happens
- Often your app code just needs to supply either a custom-created list of providers, or use supplied list

Provider Class

Main properties of Provider class

- token : any
- useClass : Type
- useValue : any
- useExisting : any
- useFactory : Function
- dependencies : Object[]
- multi : boolean

Providers

Provider

ProvideRouter (Function)

SERVER_PLATFORM_PROVIDERS

HTTP_PROVIDERS

APPLICATION_COMMON_PROVIDERS

ResolvedReflectiveProvider

Provider Config and A Resolved Provider

In application code we supply provider configuration

- The Angular Framework needs to do some internal housework to make a provider useable
- To create a resolved provider
- Sometimes more performant to do such work once

Using Providers

Provide

UseValue

UseClass

UseExisting

UseFactory

Injectors and NgModule

Platform is root injector

- Feature modules eagerly loaded do not have their own injector
- Lazy loaded modules do have their own injector

Each created component has its own injector

Arranged in a hierarchy

@

@Optional

@Self

@SkipSelf

@Host

Error Handling

Angular DI has rich descriptive exceptions

- In error situations, do examine exception

Types include:

- CyclicDependencyError, InstantiationError, InvalidProviderError, NoAnnotationError, OutOfBoundsError

Avoid Unexpected Duplicate Providers

If the same provider is presented at different component levels

- May create multiple instances

Maybe what is required

- But often not, and leads to unexpected behavior

Reflective Key

Represents the key associated with a provider

- A key is actually a token (object) and an id (number)

A key registry is maintain internally

- A map of tokens to keys

A forward ref allows use of a declared token that is not yet defined

- export interface ForwardRefFn { (): any; }

Create the reference:

- export function forwardRef(forwardRefFn: ForwardRefFn): Type {
- (<any>forwardRefFn).__forward_ref__ = forwardRef;
- (<any>forwardRefFn).toString = function() { return stringify(this()); };
- return (<Type><any>forwardRefFn); }

Resolve the reference:

- export function resolveForwardRef(type: any): any {
- if (isFunction(type) && type.hasOwnProperty('__forward_ref__') &&
- type.__forward_ref__ === forwardRef) {
- return (<ForwardRefFn>type)();
- } else { return type; } }

Forward Ref

Injection Token

```
/**
 * Creates a token that can be used in a DI Provider.
 *
 * Use an `InjectionToken` whenever the type you are injecting is not reified (does not have a
 * runtime representation) such as when injecting an interface, callable type, array or
 * parameterized type.
 * ..
 */
export class InjectionToken<T> {
  readonly ngMetadataName = 'InjectionToken';
  readonly ngInjectableDef: never|undefined;
  constructor(protected _desc: string, options?: {  .. }
  toString(): string { return `InjectionToken ${this._desc}`; }
}
```

Summary of DI Types (from source)

- export * from './metadata';
- export {InjectFlags} from './interface/injector';
- export {ɵɵdefineInjectable, defineInjectable, ɵɵdefineInjector, InjectableType, InjectorType} from './interface/defs';
- export {forwardRef, resolveForwardRef, ForwardRefFn} from './forward_ref';
- export {Injectable, InjectableDecorator, InjectableProvider} from './injectable';
- export {Injector} from './injector';
- export {ɵɵinject, inject, INJECTOR} from './injector_compatibility';
- export {ReflectiveInjector} from './reflective_injector';
- export {StaticProvider, ValueProvider, ConstructorSansProvider, ExistingProvider, FactoryProvider, Provider, TypeProvider, ClassProvider} from './interface/provider';
- export {ResolvedReflectiveFactory, ResolvedReflectiveProvider} from './reflective_provider';
- export {ReflectiveKey} from './reflective_key';
- export {InjectionToken} from './injection_token'; [LINK]