ADC
Applaudo
Developers
Conference
2020

Powered by ApplaudoStudios

The Magic of Dependency Injection in Angular

A brief introduction to Dependecy Injection in Angular

Kevin Garcia

Technologies

@khristopg

@khristop

@khristop

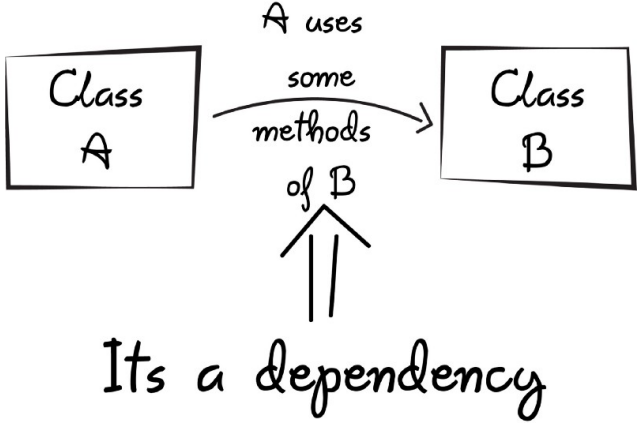
Agenda

1. Que es Dependency Injection
2. Dependency Injection en Angular
3. Tips
4. Ejemplo DI

```
export class LoginComponent implements OnInit {
  ...
  login() {
    if (this.loginForm.invalid) {
       this.snackBar.openSnackBar('error', 'danger');
       return;
    }
    this.auth.login(this.loginForm.value);
  }
}
```

```
export class SnackBar {
  ...
}

export class Auth {
  ...
}
```

Que es una dependencia?



```
export class LoginComponent {
  auth: Auth;
  snackBar: SnackBar;
  constructor() {
     this.auth = new Auth();
     this.snackBar = new SnackBar();
  }
  ...
}
```

```
export class LoginComponent {
  private _auth: Auth;
  private _snackBar: SnackBar;

  set auth(auth: Auth) {
     this._auth = auth;
  }

  set snackBar(snackBar) {
     this._snackBar = snackBar;
  }
  ...
}
```

Como proveer las dependencias

- Constructor-based injection
- Setter injection

```
export class LoginComponent implements OnInit {
  constructor() {
     this.auth = new Auth();
     this.snackBar = new SnackBar();
  }
  ...
}

export class NavbarComponent {
  constructor() {
     this.auth = new Auth();
  }
  ...
}

export class DashboardComponent {
  constructor() {
     this.auth = new Auth();
  }
  ...
}

export class OtherComponent { ... }
```

Problemas

- Se crearán las instancias de las dependencias en cada componente en el que sean utilizadas.

```
export class LoginComponent implements OnInit {
  constructor() {
     this.auth = new Auth(
       new HttpClient(new ...),
       new Storage(...));
     this.snackBar = new SnackBar(new NgZone(...));
  }
  ...
}

export class Auth {
  ...
  constructor(
    private http: HttpClient,
    private storage: Storage) {
  }
}

export class SnackBar {
  ...
  constructor(private ngZone: NgZone) {
  }
}
```

Problemas

- La clase del componente debe saber como instanciar sus dependencias, siguiendo el Single Responsibility Principle de SOLID es mejor que el componente no tenga idea de como instanciar sus dependencias.

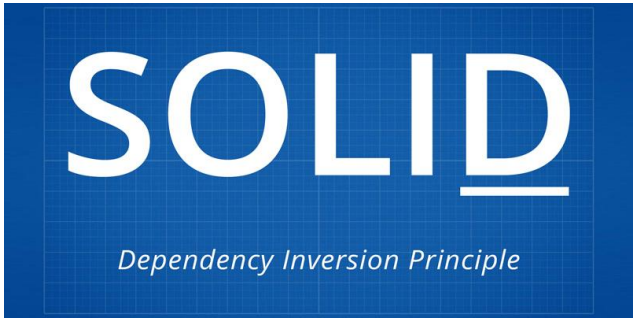
```
export class LoginComponent implements OnInit {
  constructor() {
     this.auth = new Auth(
       new HttpClient(new ...),
       new Storage(...)); // error!
     this.snackBar = new SnackBar(new NgZone(...));
  }
  ...
}


export class Auth {
  ...
  constructor(
    private http: HttpClient,
   // private storage: Storage
  ) { }
}

export class SnackBar {
  ...
  constructor(private ngZone: NgZone) {
  }
}
```

Problemas

- El componente con los servicios está fuertemente acoplado.

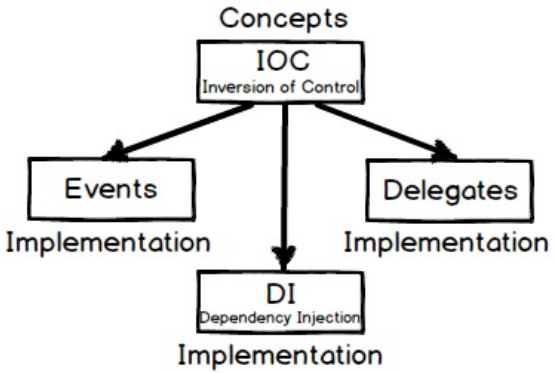
  Por lo que cuando requiera usar el componente debo usar también el servicio, por lo que reemplazar el servicio es imposible.

  Cualquier cambio en la inicialización de algún servicio requerirá cambiar el código de cada clase que lo utilice.



Dependecy Inversion Principle

DIP nos dice que los sistemas mas flexibles son aquellos en los que las dependencias hacen referencia solo a abstracciones, no a concreciones.

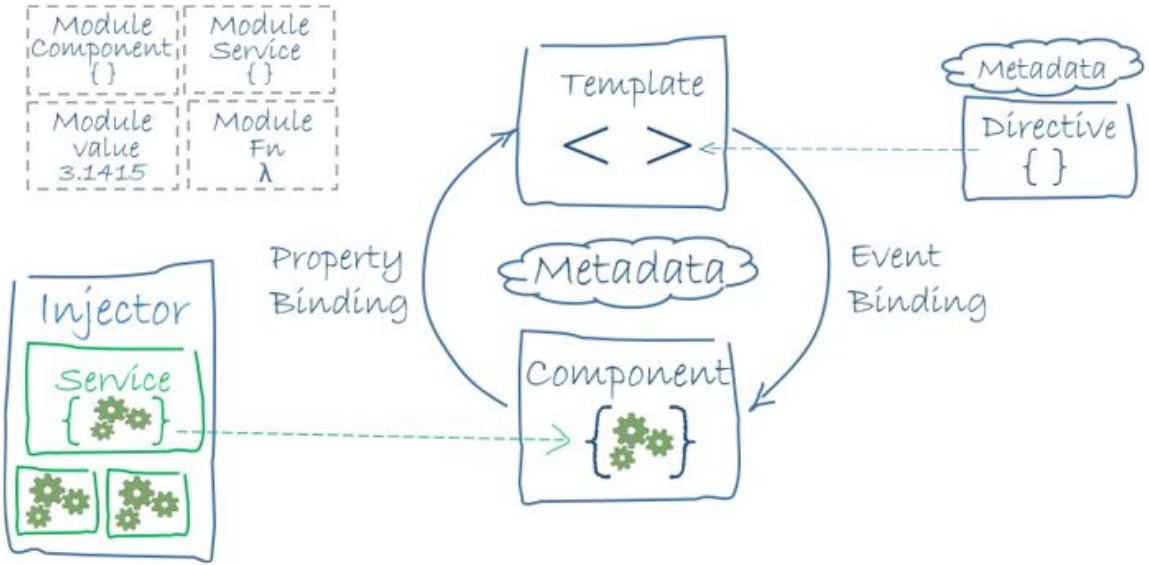


Inversion of Control (IoC)

Es un concepto de diseño usado para representar Dependency Inversion Principle en software.

La lógica que no está relacionada con esa clase se le provee desde una entidad externa y luego la usa, de forma que la implementación de Depency Injection admite la escritura de código débilmente acoplado y hace que el código sea más fácil de probar y reutilizable

Dependency Injection



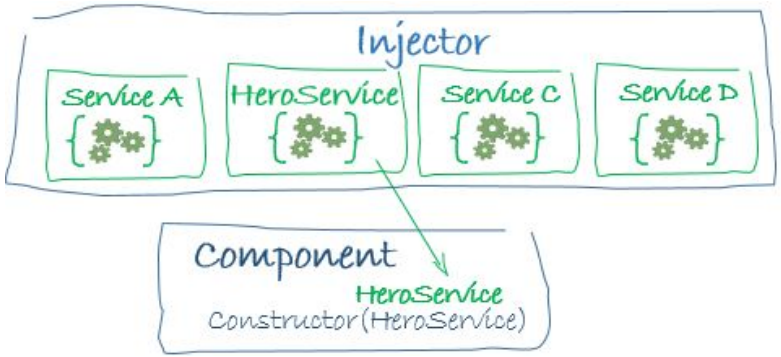
Injector

```
@Directive({ // component || service
  selector: '[appSectionContent]'
})
export class SectionContentDirective {
  constructor(
    private injector: Injector,
    private someService: SomeService,
    @Inject(BaseURL) private baseUrl: string,
  ) {}
  ...
}

// create an injector
const injector: Injector =
    Injector.create({
        providers: [{provide: 'validToken',
                     useValue: 'Value'}]});
injector.get('validToken'); // return 'Value'
```

¿Que es un injector?

Un injector es un diccionario de valores llave/valor, en el cual podemos registrar objetos del tipo Provider así como obtener el valor por medio del InjectionToken.

Cada aplicación, modulo o elemento en angular posee un injector.



Dependency Provider

```
export const APP_CONFIG = new InjectionToken<AppConfig>('app config');
const heroServiceFactory = (logger: Logger, userService: UserService) =>
    { ... };
...

providers: [
    SomeService, // shorthand for
    // { provide: SomeService, useClass: SomeService, multi: false }
    { // add new instance to HTTP_INTERCEPTORS provider array
      provide: HTTP_INTERCEPTORS,
      useClass: ApiInterceptor, multi: true },
    { // add new constant value as provider with APP_CONFIG token
      provide: APP_CONFIG,
      useValue:  {
        apiEndpoint: 'api.heroes.com',
        title: 'Dependency Injection'
    }},
    {  // add new factory provider with an array of provider tokens
      provide: HeroService,
      useFactory: heroServiceFactory,
      deps: [Logger, UserService]
    },
    {  // add new provider using an existing provider
      provide: OldLogger,
      useExisting: NewLogger
    }
]
```

¿Que es un dependecy provider?

El inyector se basa de los Providers para crear instancias de las dependencias que inyecta en components, directives, pipes y services.

¿Que es un InjectionToken?

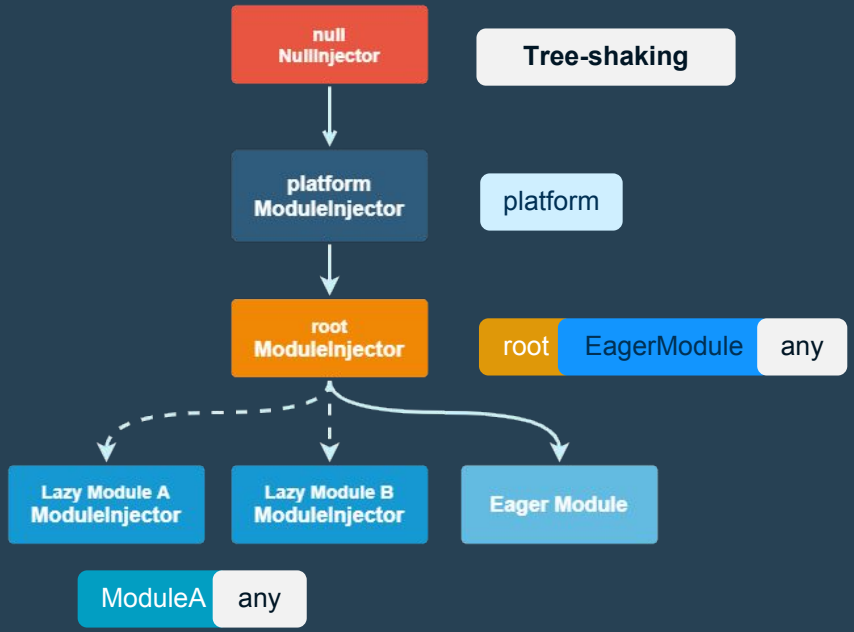
Un token de búsqueda asociado con un Dependency Provider, para usar con el sistema de inyección de dependencia.

@Injectable Decorator

Decorador que marca una clase como disponible para ser proporcionada e inyectada como dependencia.

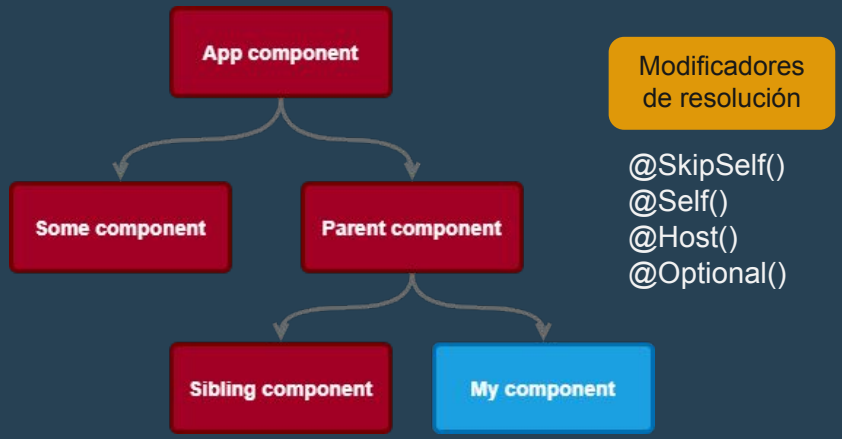
```
@Injectable()
class UsefulService {
}

@Injectable({providedIn: 'root'}
)
class NeedsService {
  constructor(public service: UsefulService) {}
}

...
// tree-shaking alternatives
-> provideIn: 'root' | 'platform' | 'any' | null
```

Hierarchical injectors: Module Injector Tree



- El injector null arroja un error si no se encuentra el provider.
- El platform injector es compartido por todas las aplicaciones de angular en la misma Ventana.
- Cada app tiene un root injector.
- Los modulos lazy crean un injector que es hijo del root.

Hierarchical injectors: Element Injector Tree



Modificadores de resolución

@SkipSelf()
@Self()
@Host()
@Optional()

```
export class MyComponent {
  constructor(private someService: SomeService) {}
}
```

- Cada componente posee su propio injector.
- La resolución inicia desde el injector del propio componente.
- Si el injector del componente no resuelve el token, angular viaja a los nodos padres
- Si después de atravesar hasta el ultimo nodo del injector tree, angular continua con el module injector tree.

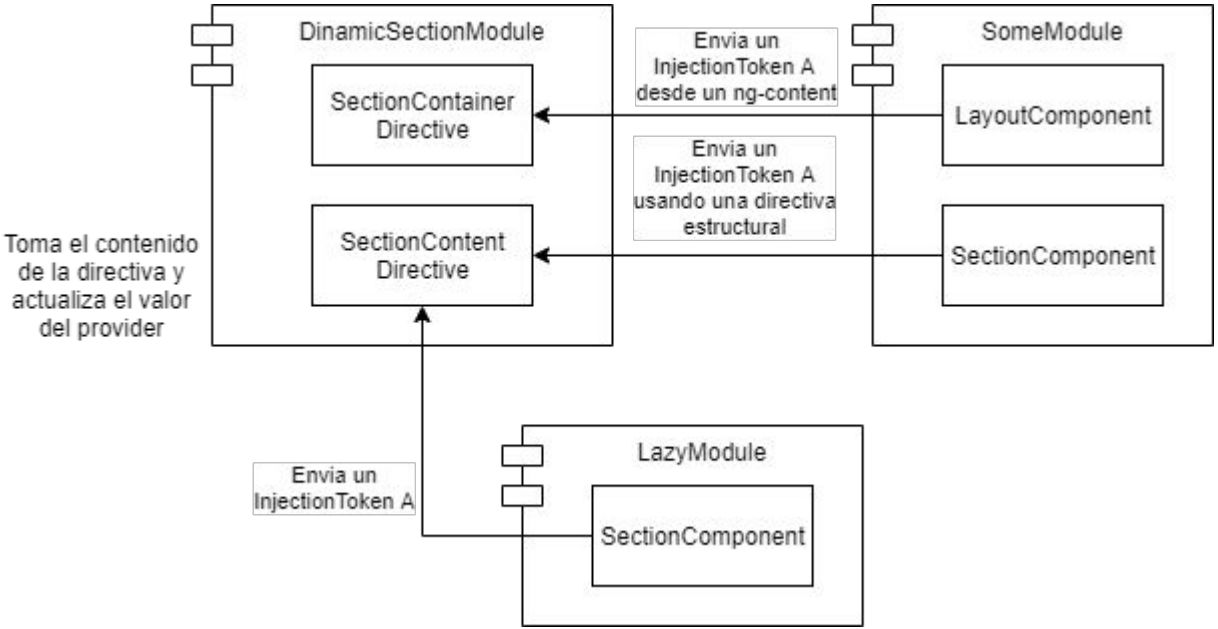
Tree-shakable Injection Tokens

Otra forma de registrar servicios.

Tree-shaking elimina el código muerto removiendo el código sin uso, estos son removidos del bundle final cuando la aplicación no usa estos servicios. Esto no sucede con NgModule providers.

```
// -> non clase base provider
const ApiUrl = new InjectionToken<string>('BaseUrl', {
  providedIn: 'root', // -> "any" | "platform" | ModuleA
  factory: () => 'localhost:3000'
});


// -> clase base provider
@Injectable({
  providedIn: 'root', // -> "any" | "platform" | ModuleA
  // useFactory: () => new Service('someDep'),
})
export class SomeService {
}
```

Demo time



Repo: https://github.com/khristop/applaudoDevConf

https://angular.io/guide/dependency-injection

https://indepth.dev/what-you-always-wanted-to-know-about-angular-dependency-injection-tree

https://dev.to/christiankohler/angular-dependency-injection-infographic-1bjm

Links

Applaudo
Developers
Conference
2020

Powered by ApplaudoStudios