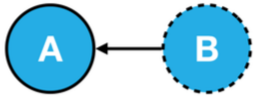
5

Dependency Injection

Angular DI, Injector and Providers

Dependency Injection

- Dependency Injection is the art of decoupling a piece of code of its dependencies
  - It’s a design pattern
  - Improves maintainability
  - Improves testability
  - Improves code design



- Dependencies are components the dependent object depends upon

Dependency Injection

- Decouple the creation of the dependency from the dependent object
  - By passing in the dependency as parameter

```
class Account {
  //tight coupling
  logger: Logger = new Logger();
}

class Account {
  //loose coupling
  constructor(public logger: Logger) {}
}
```

Dependency Injection in Angular

- Angular’s dependency injection mechanism
  - A component simply has to ask for a service
  - An injector is responsible for delivering the service and managing it’s life cycle

- The injector
  - The core of DI Framework
  - Manage the responsibility of dependency creation
  - Supplies the dependency to the dependent object

Dependency Injection in Angular

Three steps

1. Create a service
   - Use the **@Injectable** decorator

```
@Injectable()
export class AuthenticationService {

}
```

Dependency Injection in Angular

Three steps

2. Register the service in a module providers

- In a module → application wide
- In a component → locally

```
// In a module
@NgModule({
  providers: [AuthService]
})
export class AppModule { }
```

```
// In a component
@Component({
    selector: ‘my-app',
    templateUrl: ` `,
    providers: [AuthService]
})
```

Dependency Injection in Angular

Three steps

3. Use the service

- Create a constructor in component in-need
- Add the service as parameter

```
@Component({
  ...
})
export class AppComponent {

    constructor( private  _authService: AuthService) {
        this.user = _ authService.getUser();
    }
}
```

Service registration

- Registration can be handled inside the @Injectable decoarator

```
@Injectable({
         providedIn: 'root'
})
```

- providedIn:
  - ‘root’ : singleton injection for your application
  - ‘{modulename}’ : injected into specific module

Optional dependencies

- By default every service needs to be registered

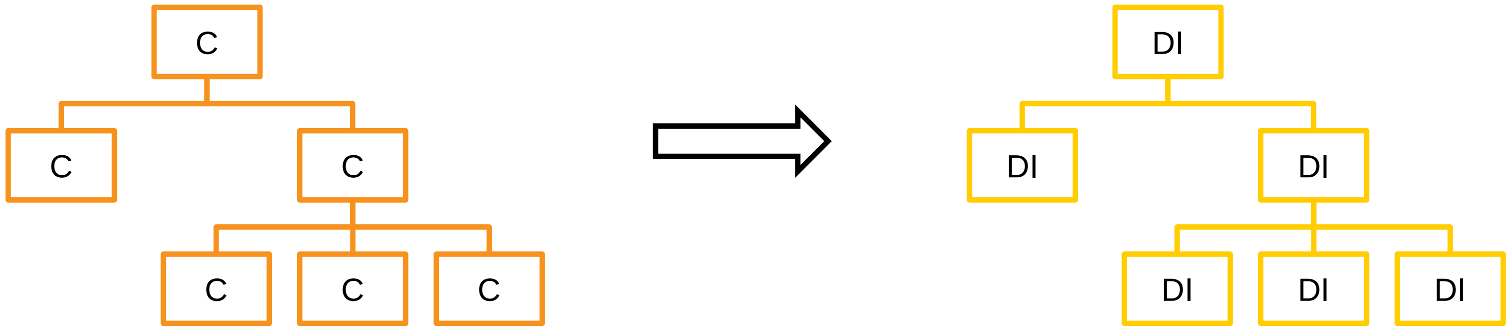
```
EXCEPTION: No provider for LoggerService! (GameListComponent ->
GameService -> LoggerService)
```

- Optional dependencies can be used
  - Need the @Optional() decorator in dependent object
  - Pass a null when service not found

```
constructor(@Optional() private _logger: LoggerService) { }
```

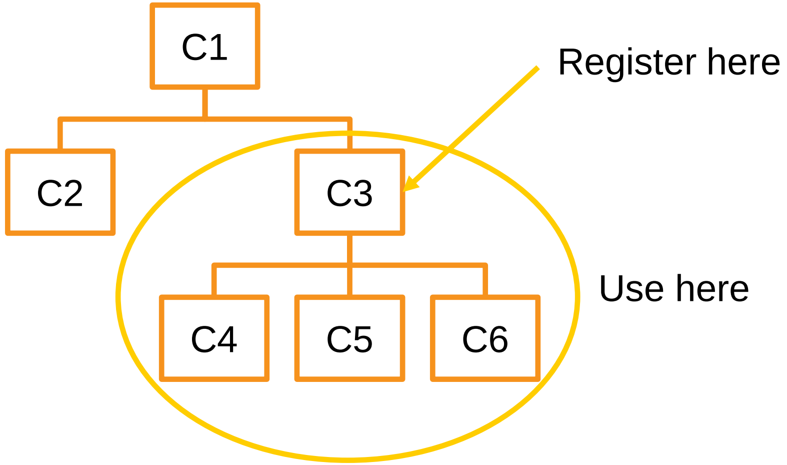
Hierarchical Injection

- Each service is created as singleton
  - In the scope of the injector
- A hierarchy of components leads to a hierarchy of injector



Hierarchical Injection

- When registered in a component, the singleton can be injected into the components and all its children



- Parent (C3) and children (C4,C5,C6) uses the same singleton

Hierarchical Injection

Example

```
// ArticlesListComponent
@Component({
 providers: [ RestoreService, LoggerService ]
})
// ArticleDetailComponent
@Component({
 providers: [ RestoreService ]
})
```

- The ArticlesList component shares a LoggerService with all ArticleDetail components
- The ArticlesList component and each ArticleDetail component has its own RestoreService

Where to register

- **STATELSS** services can easily be shared
  - One instance for all
  - Register in AppModule (providedIn root)

- **STATEFUL** services could get messy when shared
  - Components could override eachother’s state
  - Register localy (in a specific module or a specific component)

Providers

- Providers inform how to create a runtime version of the dependency

- The injector can be configured with three types providers
    - Class provider
    - Value provider
    - Factory provider

Providers

- Provide object needs two properties
  - A **token** serving as the key for registering the provider
  - Provider **definition object**

```
providers: [LoggerService]
```

↕

```
providers: [{provide: LoggerService, useClass: LoggerService }]
```

The token

The definition

Class providers

- Using the keyword : **useClass**
- Asking a different class to provide the service

```
providers: [{provide: LoggerService, useClass: MemoryLoggerService }]
```

- Somebody asking for a LoggerService will now get an instance of MemoryLoggerService
- The **token** here is the class itself

Value providers

- Using the keyword : **useValue**
- When needing an object:
  - use a **string** token

```
export const CONFIG = {
    apiEndpoint: 'api.heroes.com',
    title: 'Dependency Injection'
};
```

```
providers: [{provide: ‘app.config’, useValue: CONFIG }]
```

string token

Value providers

- Using the keyword : **useValue**
- When needing an object:
  - use a **injection** token

```
//Create token
export let APP_CONFIG = new InjectionToken('app.config’);
```

```
providers: [{provide: APP_CONFIG, useValue: CONFIG }]
```

injection token

Value providers

- Using the **value** dependency
  - use the decorator **@Inject()**
  - With a string token:

```
// @Inject(token) to inject the dependency
constructor(@Inject('app.config') private _config: Config){ }
```

  - Or injection token :

```
constructor(@Inject(APP_CONFIG) private _config: Config){ }
```

Factory providers

- Sometimes the right providers needs to be decided at runtime ?
  - Depending on certain condition
  - Switch between providers at runtime
- Solution: use factory provider
- A factory is a function that takes parameters build (instantiate) some class or object and return it

```
let loggerServiceFactory = (userService: UserService) => {
          if (userService.user.isAuthorized) {
                    return new UserLoggerService(userService.user);
          } else {
                    return new AnnonymousLoggerService();
          }
};
```

Factory providers

- Using the keyword : **useFactory**

```
@NgNodule({
  providers: [{
    provide: LoggerService,
    useFactory: loggerServiceFactory,
    deps: [UserService]
  }]
})
```

- When using facotories, extra property is needed : **deps**
  - Dependecies to inject for the factory

Summary

DI basics

- Create service using @Injectable
- Register in providers
- Inject in constructor

Providers

- Class providers (useClass)
- Value providers (useValue)
- Factory providers (useFactory)

Hierarchical Injection

- Application wide (providedIn: ‘root’)
- In the module
- Component on itself or children

DI Tokens

- Class
- String
- InjectionToken

LAB 5

Creating task service