**Introduction to services and dependency injection**

*Service* is a broad category encompassing any value, function, or feature that an application needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Angular distinguishes components from services to increase modularity and reusability.

Ideally, a component's job is to enable only the user experience. A component should present properties and methods for data binding to mediate between the view and the application logic. The view is what the template renders and the application logic is what includes the notion of a *model*.

A component should use services for tasks that don't involve the view or application logic. Services are good for tasks such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an *injectable service class*, you make those tasks available to any component. You can also make your application more adaptable by injecting different providers of the same kind of service, as appropriate in different circumstances.

Angular doesn't *enforce* these principles. Instead, Angular helps you *follow* these principles by making it easy to factor your application logic into services. In Angular, *dependency injection* makes those services available to components.

Service examples

Here's an example of a service class that logs to the browser console.

src/app/logger.service.ts (class)

export class Logger {

log(msg: any) { console.log(msg); }

error(msg: any) { console.error(msg); }

warn(msg: any) { console.warn(msg); }

}

Services can depend on other services. For example, here's a HeroService that depends on the Logger service, and also uses BackendService to get heroes. That service in turn might depend on the [HttpClient](https://v17.angular.io/api/common/http/HttpClient) service to fetch heroes asynchronously from a server.

src/app/hero.service.ts (class)

export class HeroService {

private heroes: Hero[] = [];

constructor(

private backend: BackendService,

private logger: Logger) { }

getHeroes() {

this.backend.getAll(Hero).then( (heroes: Hero[]) => {

this.logger.log(`Fetched ${heroes.length} heroes.`);

this.heroes.push(...heroes); // fill cache

});

return this.heroes;

}

}

Dependency injection (DI)



Dependency injection (DI) is the part of the Angular framework that provides components with access to services and other resources. Angular provides the ability for you to *inject* a service into a component to give that component access to the service.

Two main roles exist in the DI system: dependency consumer and dependency provider.

Angular facilitates the interaction between dependency consumers and dependency providers using an abstraction called [Injector](https://v17.angular.io/guide/glossary#injector). When a dependency is requested, the injector checks its registry to see if there is an instance already available there. If not, a new instance is created and stored in the registry. Angular creates an application-wide injector (also known as "root" injector) during the application bootstrap process, as well as any other injectors as needed. In most cases you don't need to manually create injectors, but you should know that there is a layer that connects providers and consumers.

Add the @[Injectable](https://v17.angular.io/api/core/Injectable)() decorator to a service class so that Angular can inject it into a component as a *dependency*; the optional argument tells Angular where to register this class by default.

@[Injectable](https://v17.angular.io/api/core/Injectable)({providedIn: 'root'})

export class HeroService {

* Something *injectable* must be registered with an *injector* before it can be created and used.
* Register an injectable with a *provider*, an object that tells an injector how to obtain or create a dependency. For a service class, the provider is typically the class itself.
* You don't have to create injectors. Under the hood Angular creates an application-wide *root injector* for you during the bootstrap process. It creates additional child injectors as needed.

An injectable dependency doesn't have to be a class — it could be a function, for example, or a value.

When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types. For example, the constructor of HeroListComponent needs HeroService.

src/app/hero-list.component.ts (constructor)

content\_copyconstructor(private service: HeroService) { }

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector makes one using the registered provider and adds it to the injector before returning the service to Angular.

When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments.

The process of HeroService injection looks something like this.



Providing services

You must register at least one *provider* of any service you are going to use. The provider can be part of the service's own metadata, making that service available everywhere, or you can register providers with specific components. You register providers in the metadata of the service (in the @[Injectable](https://v17.angular.io/api/core/Injectable)() decorator) or @[Component](https://v17.angular.io/api/core/Component)() metadata

* By default, the Angular CLI command [ng generate service](https://v17.angular.io/cli/generate) registers a provider with the root injector for your service by including provider metadata in the @[Injectable](https://v17.angular.io/api/core/Injectable)() decorator. The tutorial uses this method to register the provider of HeroService class definition.

hero.service.ts (provide in root)

@[Injectable](https://v17.angular.io/api/core/Injectable)({providedIn: 'root'})

export class HeroService {

When you provide the service at the root level, Angular creates a single, shared instance of HeroService and injects it into any class that asks for it. Registering the provider in the @[Injectable](https://v17.angular.io/api/core/Injectable)() metadata also allows Angular to optimize an app by removing the service from the compiled application if it isn't used, a process known as *tree-shaking*.

* When you register a provider at the component level, you get a new instance of the service with each new instance of that component. At the component level, register a service provider in the providers property of the @[Component](https://v17.angular.io/api/core/Component)() metadata.

src/app/hero-list.component.ts (component providers)

@[Component](https://v17.angular.io/api/core/Component)({

standalone: true,

selector: 'app-hero-list',

templateUrl: './hero-list.component.html',

imports: [ [NgFor](https://v17.angular.io/api/common/NgFor), [NgIf](https://v17.angular.io/api/common/NgIf), HeroDetailComponent ],

providers: [ HeroService ]

})

When you register a provider at the component level, you get a new instance of the service with each new instance of that component.

For [NgModule](https://v17.angular.io/api/core/NgModule) based applications, use the providers field of the @[NgModule](https://v17.angular.io/api/core/NgModule) decorator to provide a service or other [Injectable](https://v17.angular.io/api/core/Injectable) available at the application level.

**What is Dependency Injection ?**

**Dependency Injection** is a **design pattern** in which **components or services are provided with their dependencies** instead of creating or locating them internally. In Angular, the Dependency Injection system manages the dependencies between various parts of an application, providing loose coupling and modular development.

**Key Concepts of Dependency Injection in Angular**

1. **Providers:**
   * Providers are responsible for registering dependencies with the Angular Dependency Injection system.
   * They define how instances of services or values are created and made available throughout the application.
   * Providers are typically registered at the module level using the providers array in the module metadata or at the component level using the providers property in the component metadata.
2. **Injection Tokens:**
   * Injection tokens serve as keys for looking up dependencies in Angular's Dependency Injection system.
   * They are typically classes or values that act as unique identifiers for specific dependencies.
   * Angular provides built-in injection tokens for commonly used services like HttpClient, RouterModule, etc.
3. **Injection Mechanism:**
   * Components, services, or other Angular constructs declare dependencies in their constructors by specifying the corresponding injection tokens as parameters.
   * When Angular creates an instance of a component or service, it resolves the dependencies by looking up the providers registered in the current injector hierarchy.
   * Angular automatically injects the appropriate dependencies into the constructor parameters based on the injection tokens.
4. **Hierarchical Nature:**
   * Angular's Dependency Injection system is hierarchical, meaning that each component has its own injector that can access dependencies provided by its parent component or any ancestor component.
   * This hierarchical nature allows dependencies to be scoped at different levels of the application, promoting encapsulation and reusability.

Singleton means Angular creates only one object of the service, regardless of how many places you use it. However, if you re-provide the service in another component or a module and use it inside, Angular will create separate objects for them.