### 1. **Tree-shakable Injection with `providedIn`**

```ts
@Injectable({
  providedIn: 'root'
})
export class MyService {}
```

* **`providedIn: 'root'`**:

  * Registers the service at the root injector level.
  * Creates a **singleton** service accessible application-wide.
  * Tree-shakable: the service is only included in the final bundle if it is used.

Other options:

* `providedIn: 'any'`: Creates a new instance in each lazy-loaded module.
* `providedIn: SomeModule`: Only available within that specific module.

### 2. **Providing Services in Components**

```ts
@Component({
  providers: [CustomService]
})
```

* A new instance of `CustomService` will be created **only** for this component and its children.
* Used when the component needs a different behavior/state from the global one.

### 3. **Provider Configuration Options**

#### a. `useClass`

```ts
{ provide: MyInterface, useClass: MyService }
```

* Provides an instance of `MyService` when `MyInterface` is injected.

#### b. `useValue`

```ts
{ provide: 'API_ENDPOINT', useValue: 'https://api.example.com' }
```

* Injects a static value or object.

#### c. `useFactory`

```ts
{
  provide: CustomService,
  useFactory: (http: HttpClient) => new CustomService(http),
  deps: [HttpClient]
}
```

* Dynamically creates instances with custom logic.

#### d. Object Injection

```ts
{ provide: 'APP_CONFIG', useValue: { apiUrl: '...', debug: true } }
```

* Helpful for injecting settings/configurations.

### 4. **Injection Parameter Decorators**

#### a. `@Optional()`

```ts
constructor(@Optional() config?: ConfigService) {}
```

* Prevents error if the dependency is not found.

#### b. `@Self()`

```ts
constructor(@Self() service: ServiceFromComponentOnly) {}
```

* Only looks in the current injector.

#### c. `@SkipSelf()`

```ts
constructor(@SkipSelf() parentService: ParentLevelService) {}
```

* Skips the current injector and looks in parent injectors.

#### d. `@Host()`

```ts
constructor(@Host() hostService: ConfigService) {}
```

* Restricts the search to the host component/directive only.

### 5. **InjectionToken Usage**

```ts
export const CUSTOM_TOKEN = new
InjectionToken<CustomService>('CUSTOM_TOKEN');

providers: [
  {
    provide: CUSTOM_TOKEN,
    useClass: CustomService
  }
]
```

* Enables DI with interfaces or abstract types.

### 6. **Directive and Component Example with `@Host()`**

#### Summary Component:

```html
<app-summary appTableHighlighter></app-summary>
```

#### Directive:

```ts
@Directive({ selector: '[appTableHighlighter]' })
export class TableHighlighterDirective {
  constructor(@Host() config: ConfigService) {}
}
```

* Ensures `ConfigService` is resolved from the host component
(`SummaryComponent`).

### Summary

These DI techniques are essential for:

* Code modularity and flexibility
* Testing and mocking dependencies
* Optimizing bundle size (tree-shaking)
* Managing service scopes

* Handling complex service instantiation