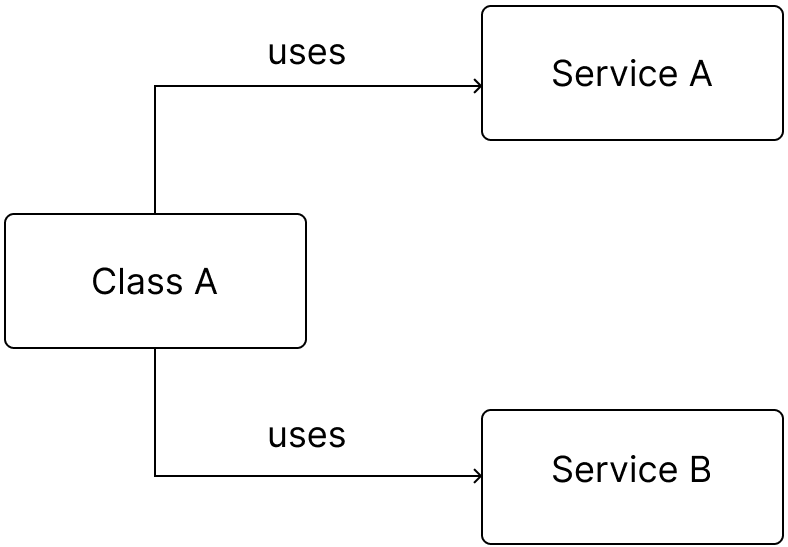
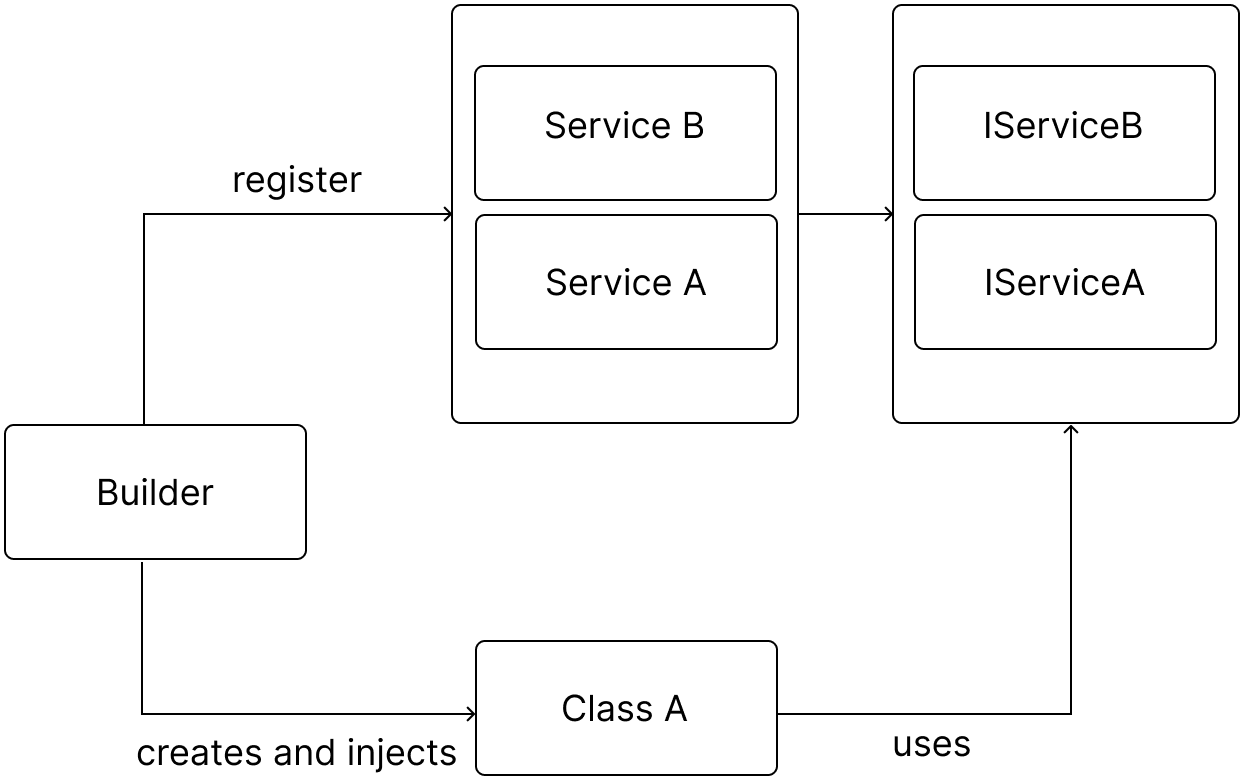
Dependency Injection

Какво е Dependency Injection (DI) ?

Dependency Injection (DI) е софтуерна концепция, при който зависимостите на даден клас или компонент се предоставят от външен източник (обикновено injector), вместо класът сам да създава или да е отговорен за тях.

"Зависимост" се отнася до всеки обект, който даден клас или компонент използва.





Ползи

- Повишена тестируемост
- Намалява свързаността между класовете
- Повишава модулността и гъвкавостта

Негативи

- Learning Curve
- Повишена сложност
- Усложнява процеса на debugging

DI в Angular

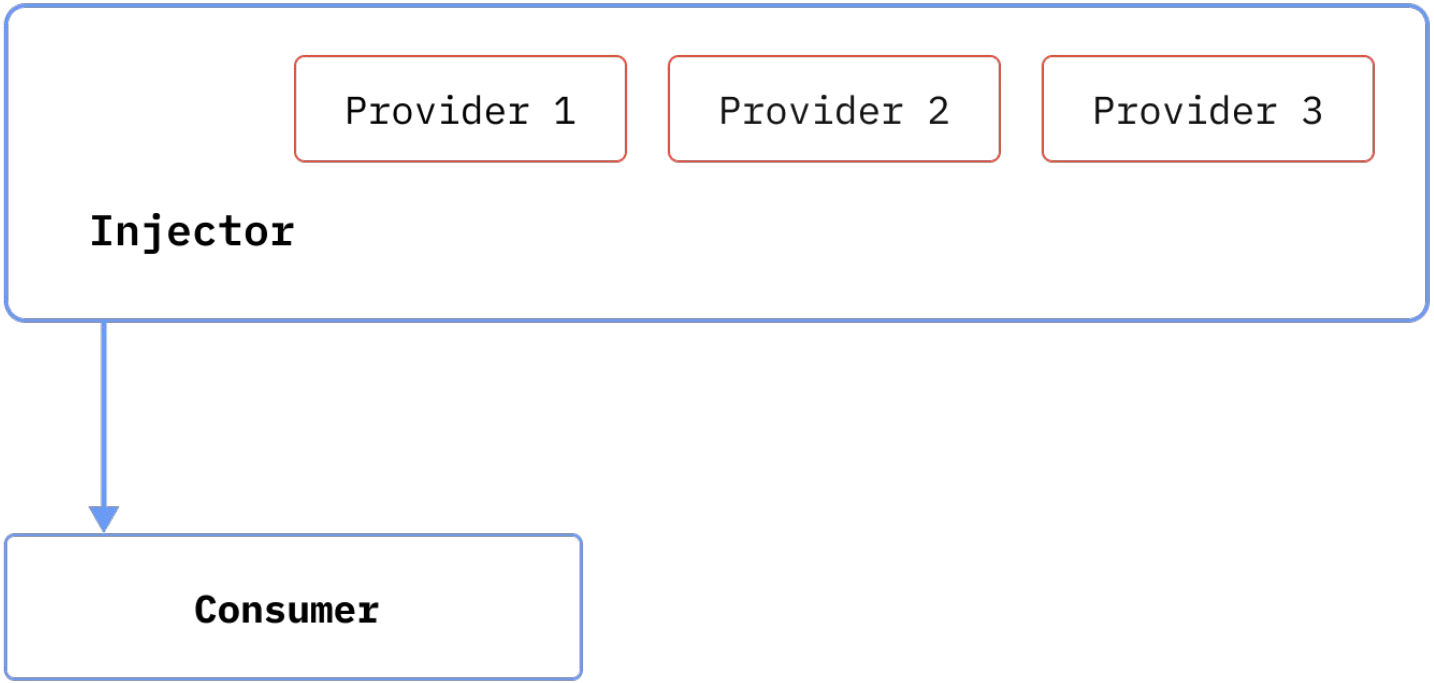
- Как работи DI в Angular ?
- Как да използваме DI в Angular ?

Как работи в Angular ?

В DI системата на Angular има 2 главни роли:

- Dependency consumer
- Dependency provider

Взаимодействието между двете роли се осъществява с помощта на Injector.



Как да използваме DI в Angular ?

Първа стъпка от използването на DI е да създадем dependency provider

```
@Injectable()
export class ProductService {}
```

Следващата стъпка е да го направим видим за dependency consumer

Как да създадем dependency provider ?

- С provideIn
- На компонентно ниво
- В конфигурацията на приложението
- В приложения базирани на ngModule

Използвайки provideIn

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

На компонентно ниво

```
@Component({
  standalone: true,
  selector: 'app-product',
  providers: [ProductService]
})
class ProductComponent {}
```

В конфигурацията на приложението

```
export const appConfig: ApplicationConfig = {
  providers: [{ provide: ProductService }],
};
```

В приложения базирани на ngModule

```
@NgModule({
  declarations: [ProductComponent],
  providers: [ProductService],
})
export class ProductModule {}
```

Как да използваме dependency provider ?

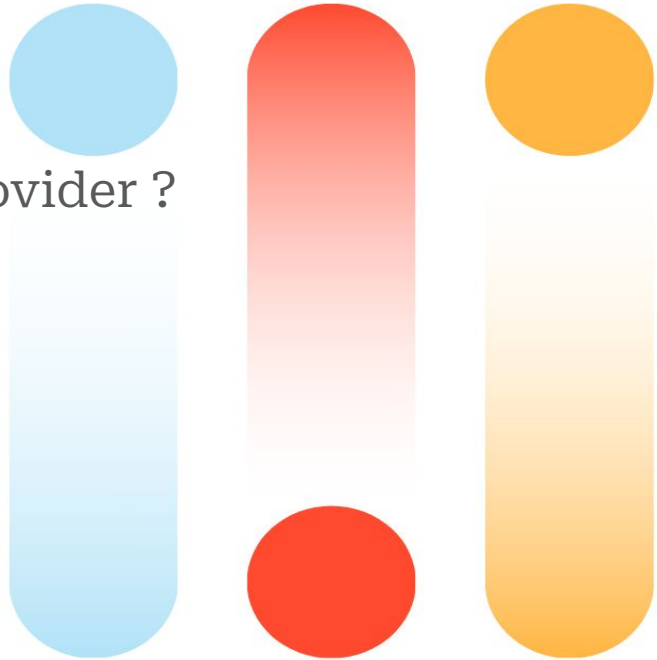
Най-простият начин да използваме dependency provider е като го декларираме в конструктора на класа.

```
@Component({
  standalone: true,
  selector: 'app-product',
  template: `<p>Product</p>`,
  providers: [ProductService],
})
export class ProductComponent {
  constructor(private productSvc: ProductService) {}
}
```

```
@Component({
  standalone: true,
  selector: 'app-product',
  template: `<p>Product</p>`,
  providers: [ProductService],
})
export class ProductComponent {
  private productSvc = inject(ProductService);
  constructor() {}
}
```

Dependency Providers

- Какво е dependency provider ?
- Как да конфигурираме dependency provider ?



Какво е dependency provider ?

Начин за определяне как дадена зависимост (dependency) трябва да бъде създадена, конфигурирана и използвана в приложението.

Как да конфигурираме dependency provider ?

Конфигурацията на dependency provider се състои от :

- Начин за намиране на конкретна зависимост (dependency) - provider token
- Начин за създаване на стойността, която ще бъде върната
  - useClass
  - useExisting
  - useFactory
  - useValue

useClass

Определя класа, който ще бъде създаден и използван за dependency.

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

```
providers: [{ provide: ProductService, useClass: BetterProductService }],
```

```
providers: [ProductService],
```

=

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

useExisting

Позволява ни да свържем един токен към вече съществуващ такъв.
В този случай новия токен играе ролята на псевдоним.

```
providers: [
  BetterProductService,
  { provide: ProductService, useExisting: BetterProductService },
],
```

useFactory

Позволява ни да създадем dependency, чрез извикване на функция. Тази функция може да приема параметри, чрез които може да променим стойността която се връща.

```
providers: [
  {
    provide: ProductService,
    useFactory: productServiceFactory,
    deps: [UserService],
  },
],
```

```
const productServiceFactory = (userSvc: UserService) => {
  if (userSvc.isAdmin) {
    return new BetterProductService();
  }
  return new ProductService();
};
```

```
providers: [
  {
    provide: ProductService,
    useFactory: productServiceFactory,
    deps: [UserService],
  },
],
```

useValue

Позволява ни да използваме статична стойност заедно с DI токен

```
providers: [
  {
    provide: ProductService,
    useValue: {
      getProducts() {
        return [];
      },
    },
  },
],
```

Provider token

Идентификатор, който се използва за намиране на зависимост (dependency) в DI системата.

Токенът, може да бъде клас, низ (string) или персонализиран обект

Създаване на dependency token

```
interface AppConfig {
  title: string;
  version: string;
}

export const APP_CONFIG = new InjectionToken<AppConfig>(
  'app config (description)'
);
```

```
@Component({
  standalone: true,
  selector: 'app-product',
  template: `<p>Product</p>`,

  providers: [
    { provide: APP_CONFIG, useValue: { title: 'Product', version: '1.0.0' } },
  ],
})
export class ProductComponent {
  constructor(@Inject(APP_CONFIG) private appConfig: AppConfig) {}
}
```

Упражнение

DI context

- Какво представлява DI context ?
- Как да се възползваме от DI context ?

Какво представлява DI context ?

Това е контекстът, в който се намира текущия Injector
Без него няма как да използваме DI.

DI контекста е достъпен в следните ситуации:

- По време на създаването на даден клас (constructor)
- При инициализирането на полетата на класа
- Във factory функции
- runInInjectionContext

```
class CatalogComponent {
  private megaProductSvc: MegaProductService;
  private productSvc = inject(ProductService); // in context

  constructor(private betterProductSvc: BetterProductService) { // in context

    this.megaProductSvc = inject(MegaProductService); // In context
  }
}
```

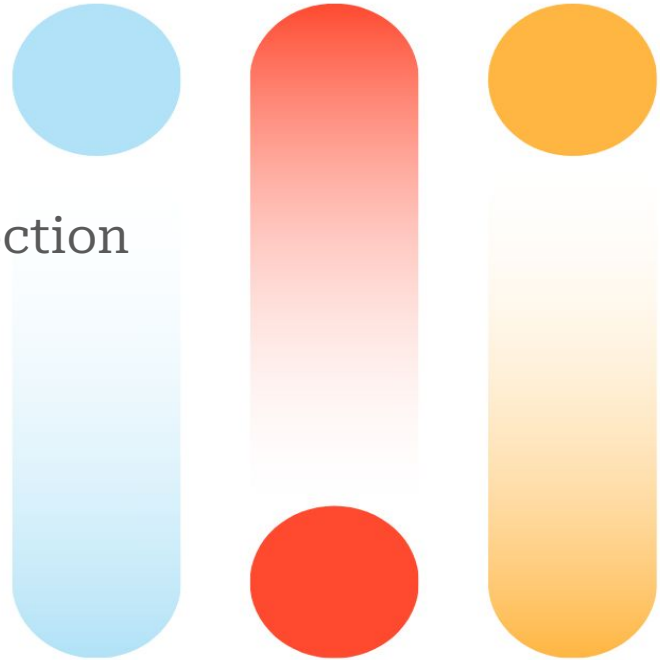
Как да се възползваме от DI context ?

runInInjectionContext - функция, която ни позволява да изпълним даден код в DI контекст, без вече да сме в него.

```
export class CatalogService {
  private environmentInjector = inject(EnvironmentInjector);
  someMethod() {
    runInInjectionContext(this.environmentInjector, () => {
      inject(ProductService);
    });
  }
}
```

Hierarchical injection

- Какво е Hierarchical injection ?
- Типове Hierarchical injection
- Как да модифицираме Hierarchical injection
- Упражнение



Какво е Hierarchical injection ?

Система, която позволява на различни компоненти и услуги в Angular да имат свои собствени инстанции на зависимости (dependency) или да споделят инстанции в зависимост от това къде и как се дефинират доставчиците (providers).

Типове Hierarchical injection

Има 2 типа Hierarchical injection:

- EnvironmentInjector
- ElementInjector

EnvironmentInjector

Injector, който е достъпен за цялата апликация.

EnvironmentInjector може да бъде конфигуриран по 3 начина:

- @Injectable() -> providedIn = root || platform
- ApplicationConfig -> providers - ако приложението ни е базирано на Standalone Component API
- @NgModule() -> providers - ако приложението ни е базирано на NgModule.

@Injectable() -> providedIn

TS logger.service.ts

```ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class LoggerService {
  constructor() {}

  log(message: string, data?: any) {
    console.log(
      `%c${message}`,
      'background: blue; color: white; display: block; padding: 0px 5px;',
      data
    );
  }
}
```

ApplicationConfig -> providers

TS app.config.ts

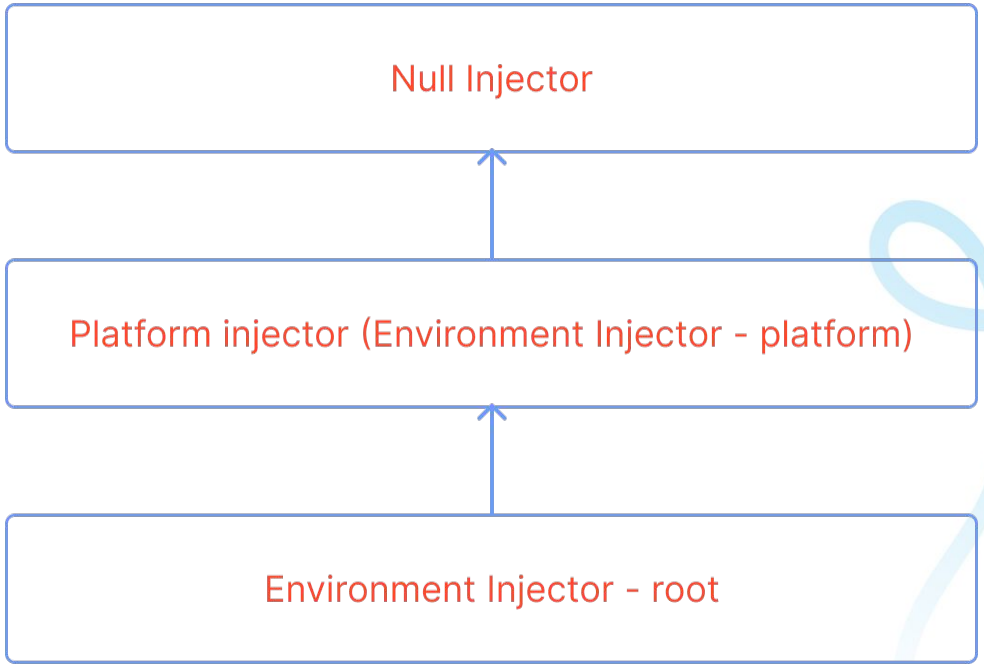
```ts
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { LoggerService } from './services/logger.service';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes), LoggerService],
};
```

@NgModule() -> providers

TS app.module.ts

```ts
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [LoggerService],

  bootstrap: [AppComponent],
})
export class AppModule {}
```

EnvironmentInjector hierarchy



EnvironmentInjector - root

Функцията bootstrapApplication създава нов EnvironmentInjector, който е дете на Platform injector, този нов injector се нарича - **root**

В него се намират само зависимисти (dependencies) специфични за текущото приложението.

TS main.ts

```ts
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

Platform injector

Функцията platformBrowserDynamic() създава EnvironmentInjector, който съдържа специфични зависимости (dependencies) за платформата. Този injector се нарича **platform**

Platform injector ни позволява да имаме зависимости, които се споделят от много приложения по едно и също време.

Можем да добавяме зависимости в Platform injector, когато зададем стойността на providedIn = **platform**

TS logger.service.ts

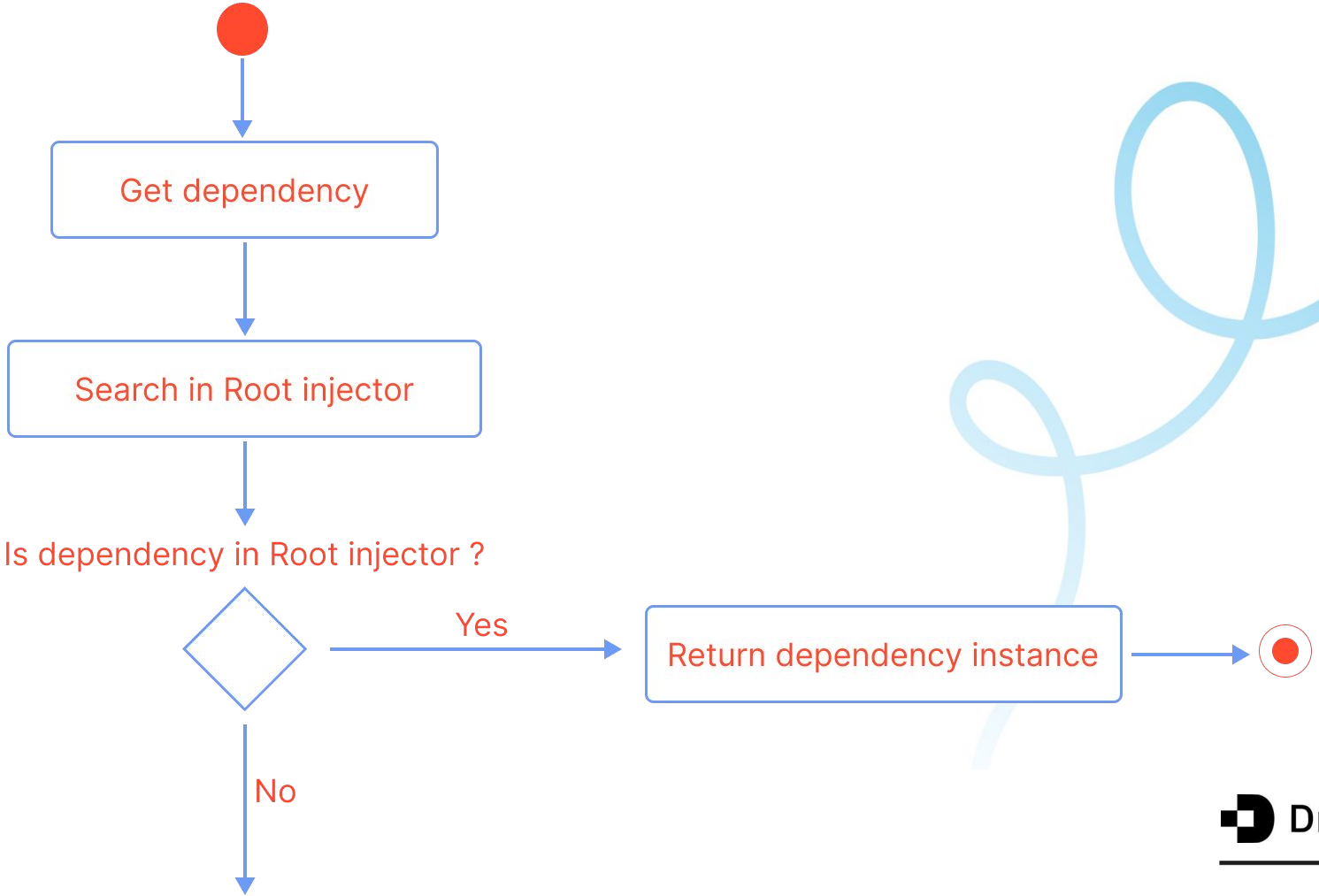
```ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'platform',
})
export class LoggerService {
  constructor() {}

  log(message: string, data?: any) {
    console.log(
      `%c${message}`,
      'background: blue; color: white; display: block; padding: 0px 5px;',
      data
    );
  }
}
```

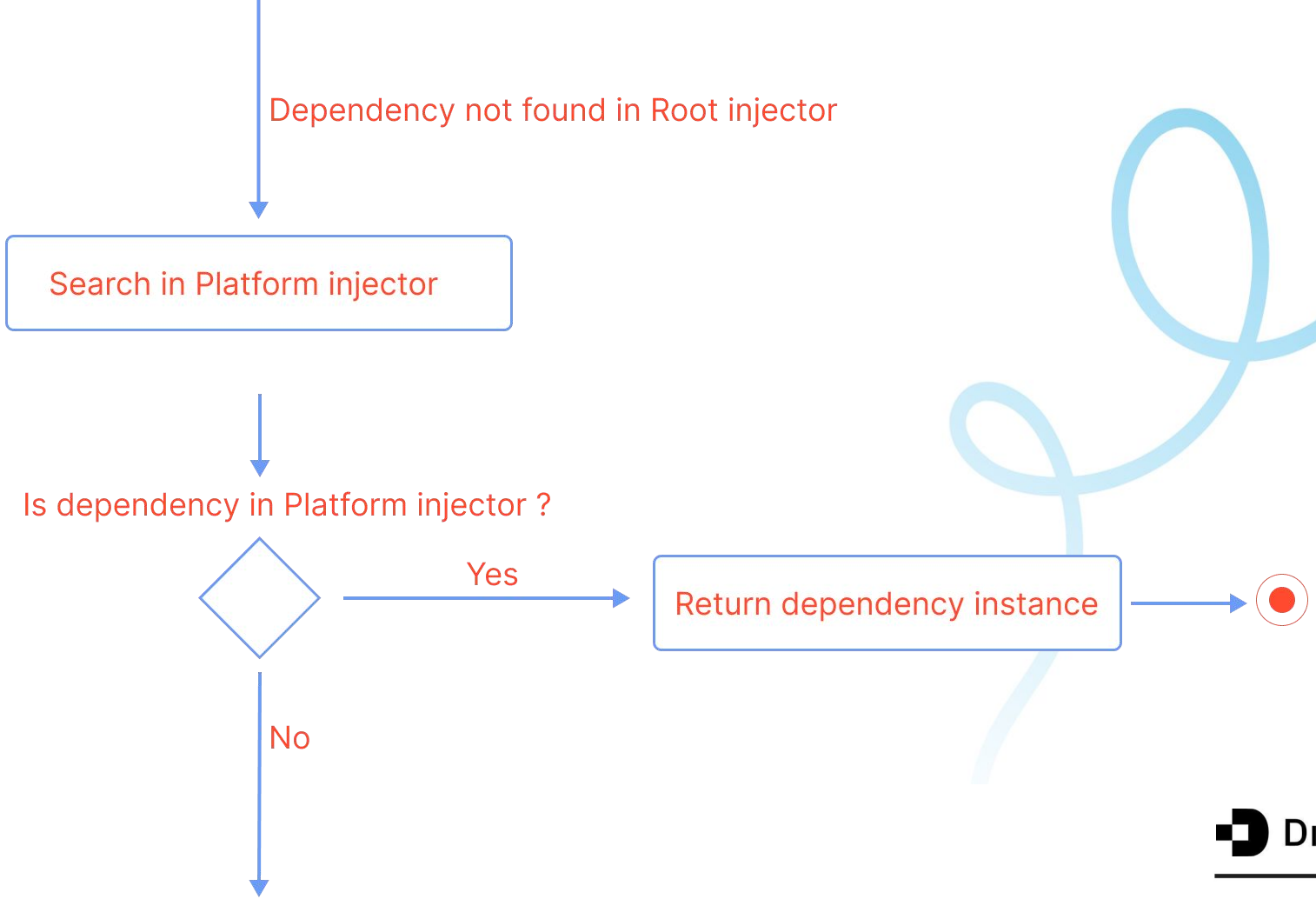
NullInjector

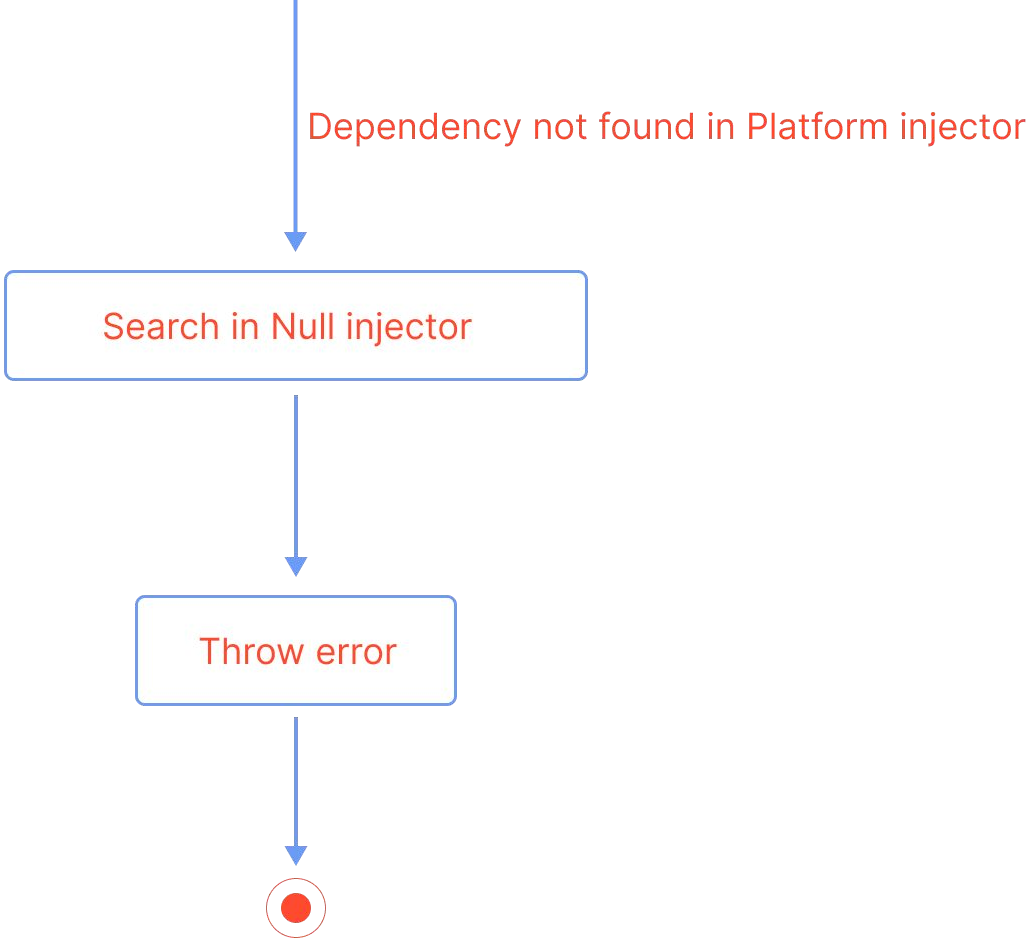
Последният Injector в йерархията.

Ако при търсене на зависимост (dependency) сме стигнали до него значи тя не съществува и ще получим грешка.

```
ERROR NullInjectorError: R3InjectorError(Environment Injector)[_LoggerService -> _LoggerService]:
NullInjectorError: No provider for _LoggerService!
```







ElementInjector

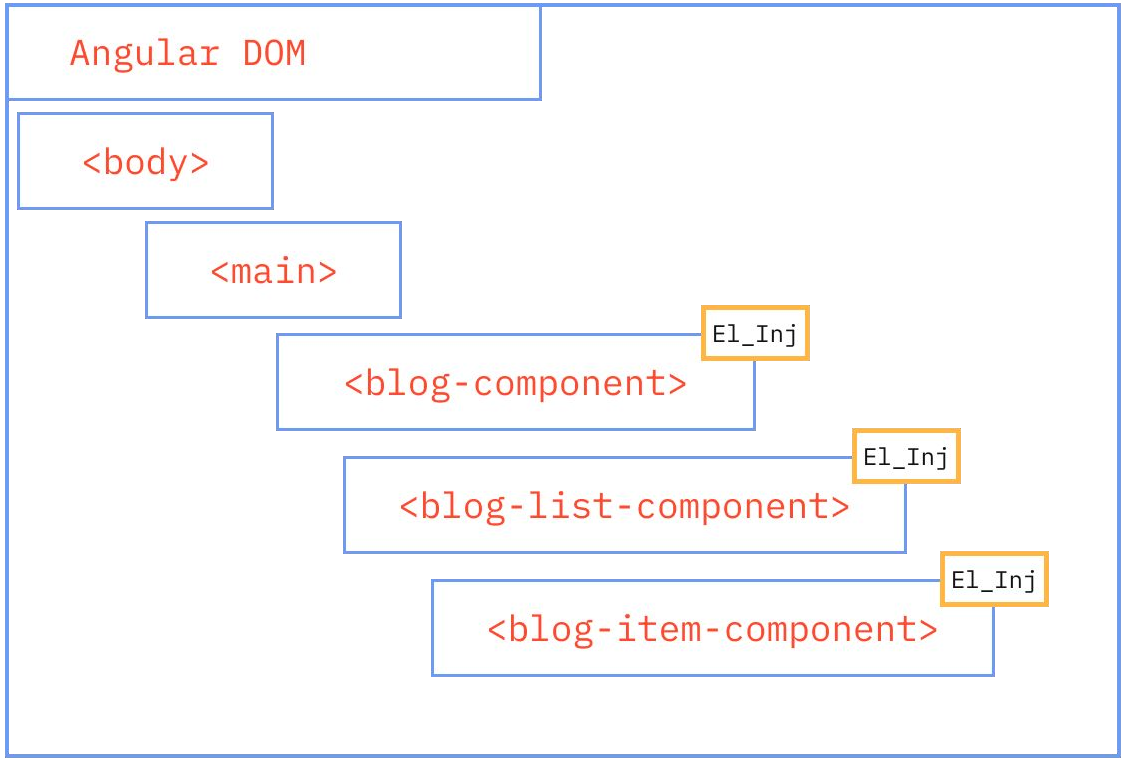
ElementInjector се създава за всеки Angular компонент и е свързан с конкретен DOM елемент.

Всеки компонент има свой собствен injector, той е отговорен за предоставянето на зависимости за този компонент и неговите подкомпоненти/директиви.

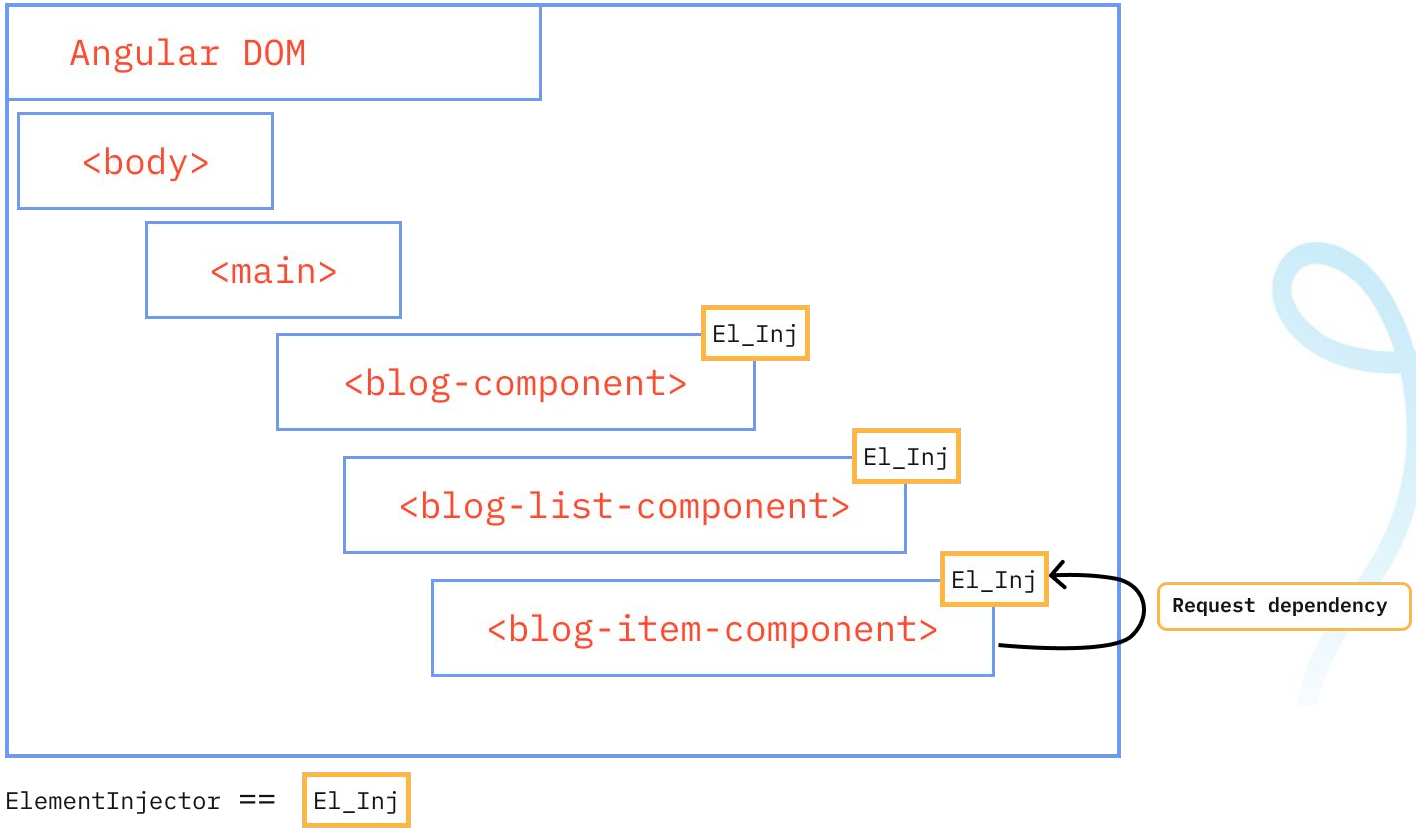
TS app.config.ts

```ts
import { Component } from '@angular/core';
import { LoggerService } from '../services/logger.service';

@Component({
  selector: 'app-user',
  standalone: true,
  providers: [LoggerService],
  templateUrl: './user.component.html',
})
export class UserComponent {
  constructor(private logger: LoggerService) {}
}
```

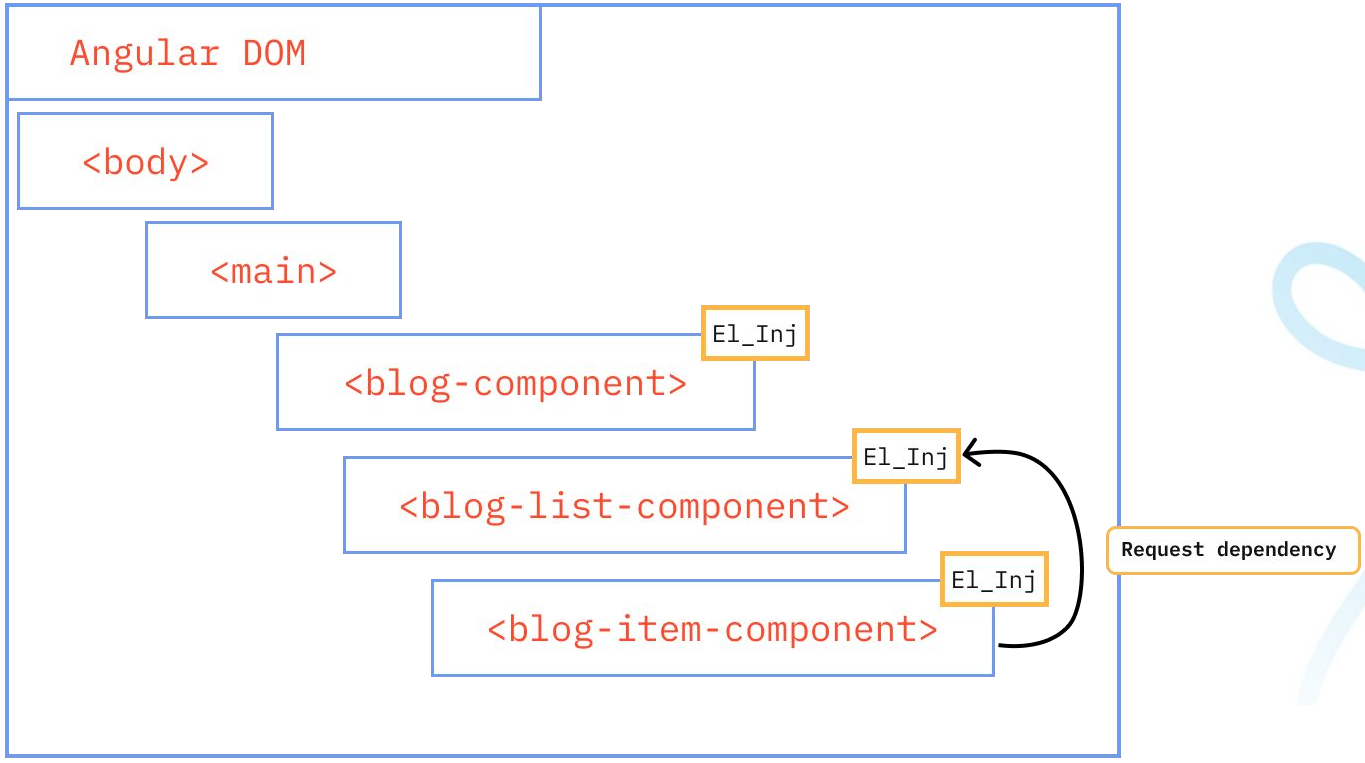
ElementInjector hierarchy



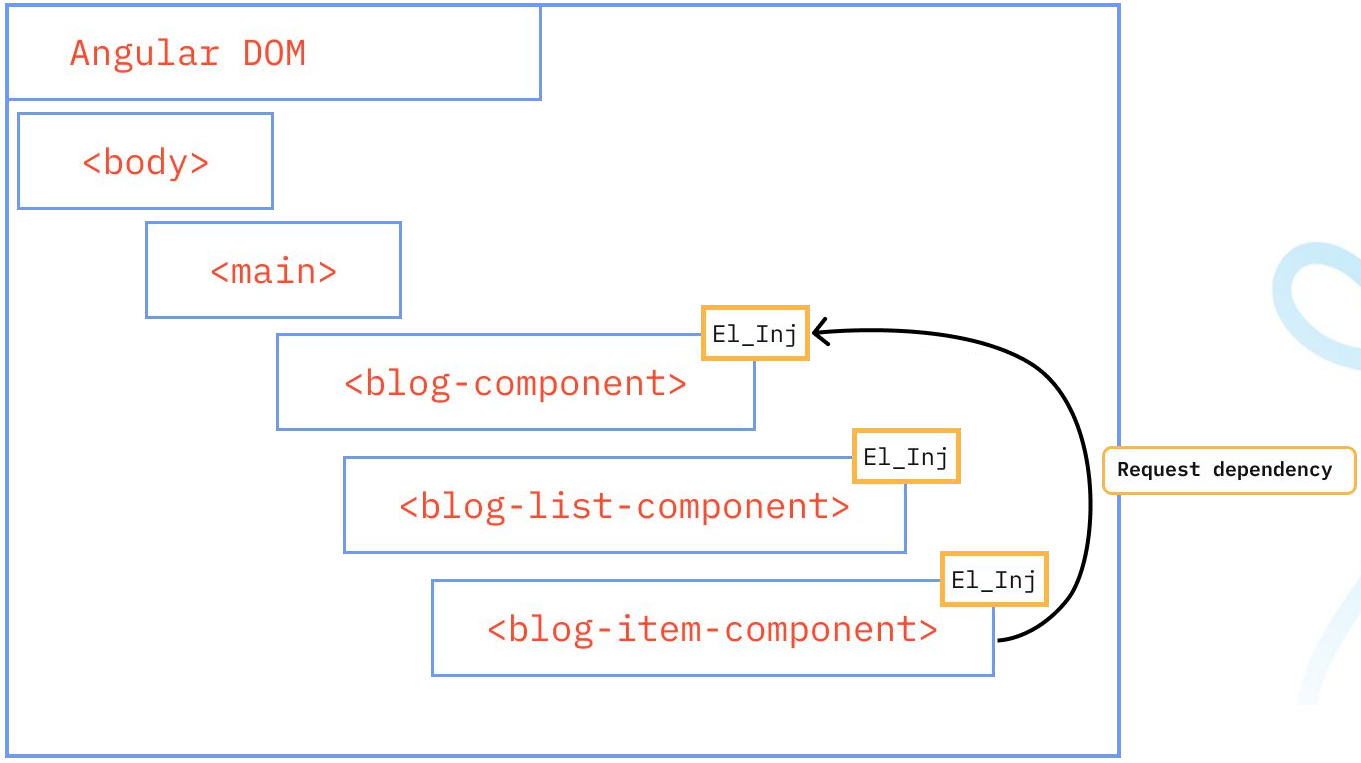
ElementInjector == El_Inj



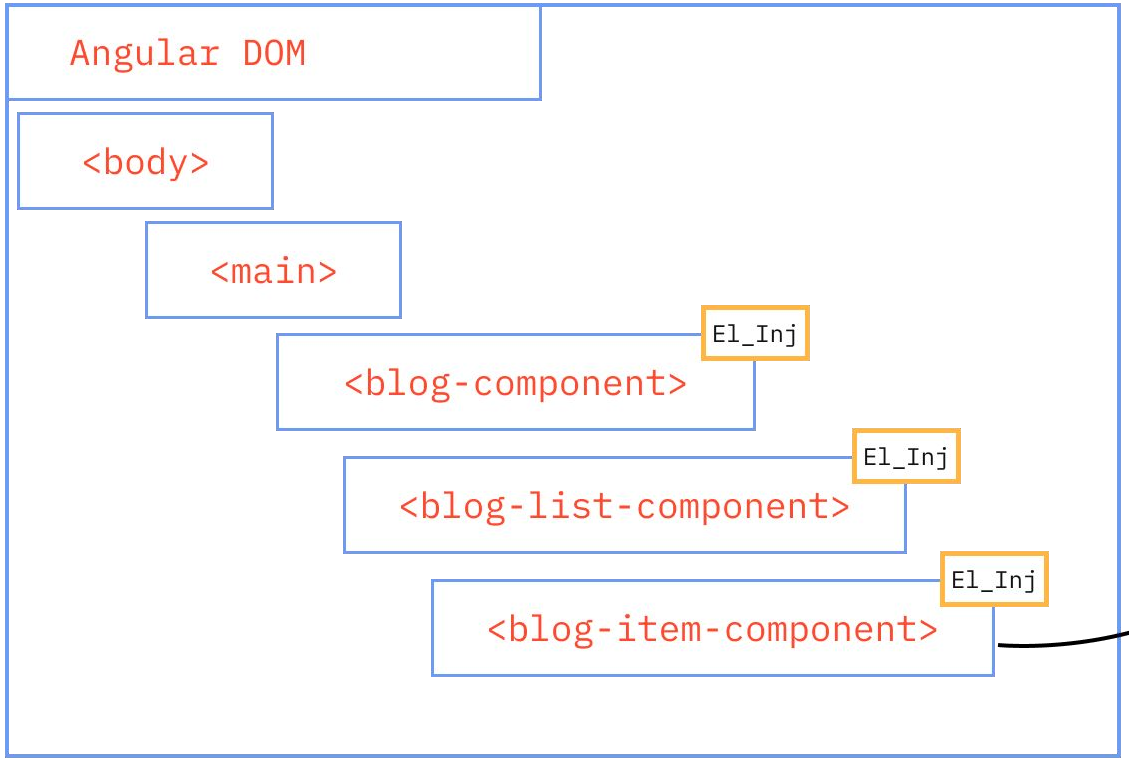
ElementInjector == El_Inj



ElementInjector == El_Inj



ElementInjector == El_Inj

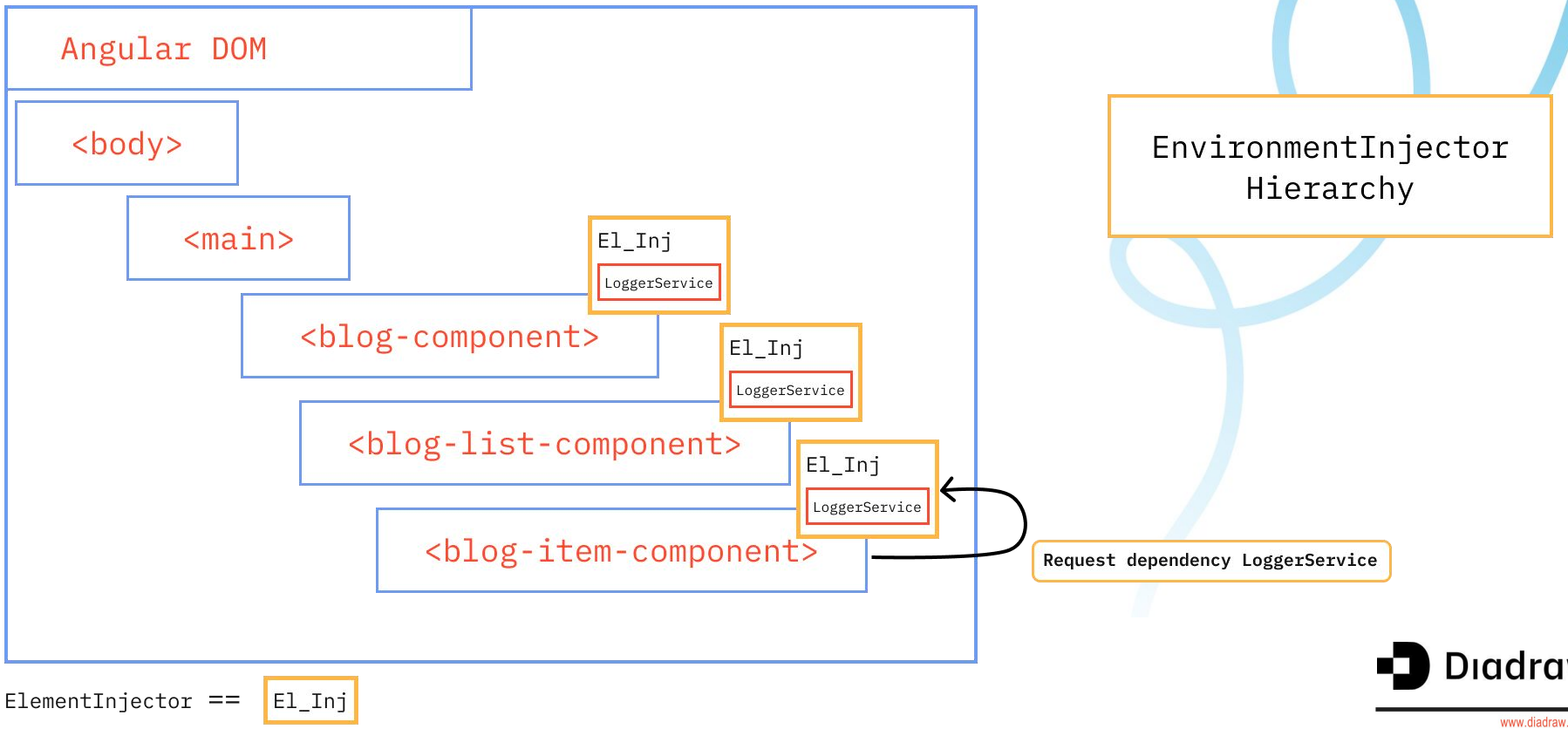


EnvironmentInjector Hierarchy

Request dependency

ElementInjector == El_Inj

Ако регистрираме едни и същи токени на различни нива, Angular DI ще върне инстанцията на първия който срещне.



Resolution modifiers

Какви са Resolution modifiers?

Специални декоратори, които позволяват на Angular да има по-голяма гъвкавост при определянето на това откъде да вземе необходимите зависимости (dependency) за даден компонент или директива.

По подразбиране Angular винаги започва търсенето на зависимост от текущия ElementInjector.

Модификаторите ни позволяват да променяме началната и крайната точка на търсене.

Типове модификатори:

- Какво да прави Angular когато не може да намери зависимост ?
- Откъде да започне да търси за зависимост ?
- Къде да спре да търси за зависимост ?

Какво да прави Angular когато не може да намери зависимост ?

- @Optional()

@Optional()

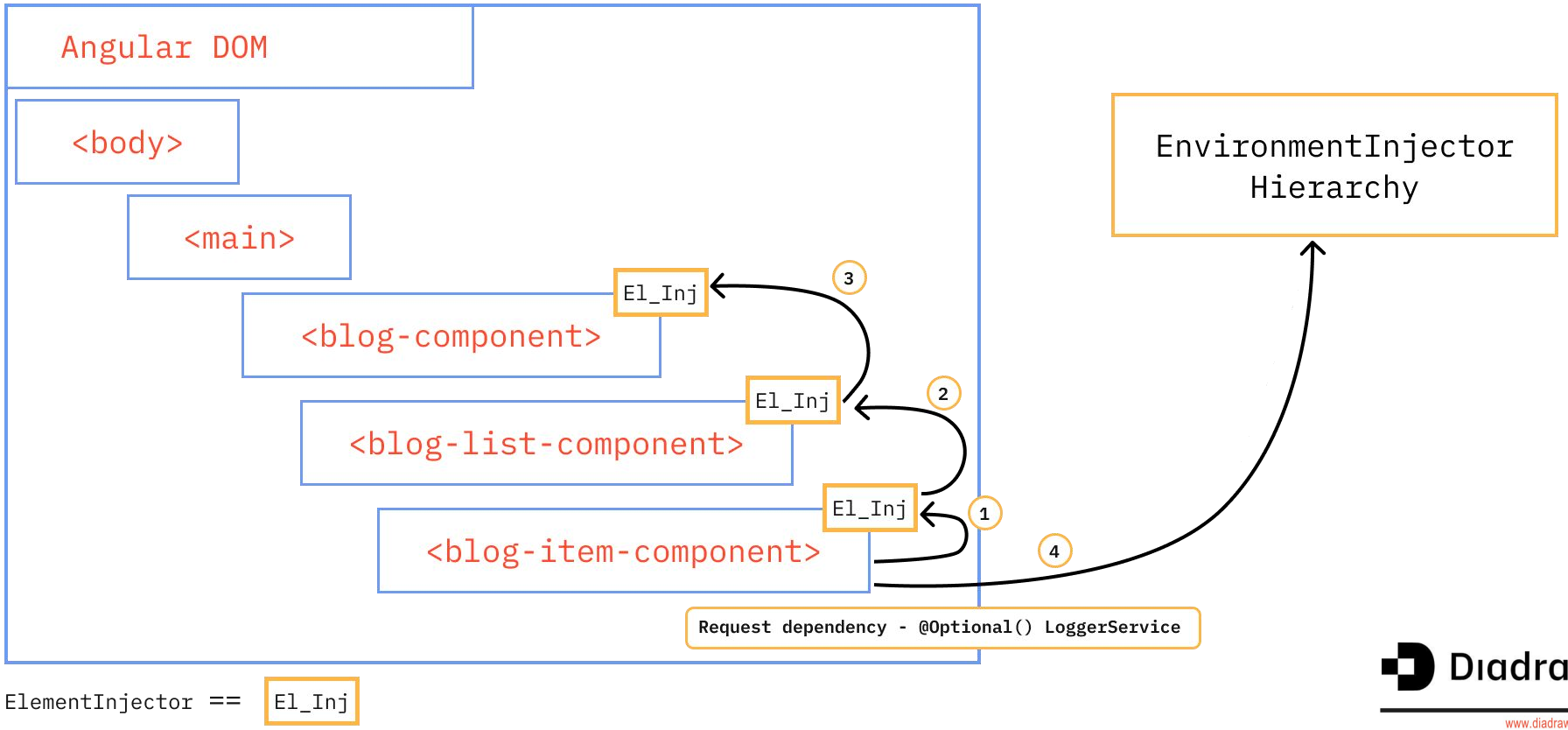
Позволява да смятаме някои зависимости за опционални, което значи, че NullInjector няма да хвърли грешка когато не ги открие.

Този декоратор е полезен, когато зависимостта е по избор и не е задължителна за правилната работа на компонента/директивата.

Ако зависимостта липсва, стойността ѝ ще бъде **null.**

TS user.component.ts

```ts
import { Component, Optional } from '@angular/core';
import { LoggerService } from '../services/logger.service';

@Component({
  selector: 'app-user',
  standalone: true,
  providers: [LoggerService],
  templateUrl: './user.component.html',
})
export class UserComponent {
  constructor(@Optional() private logger: LoggerService) {}
}
```



ElementInjector == El_Inj

Откъде да започне да търси за зависимост ?

- @SkipSelf()

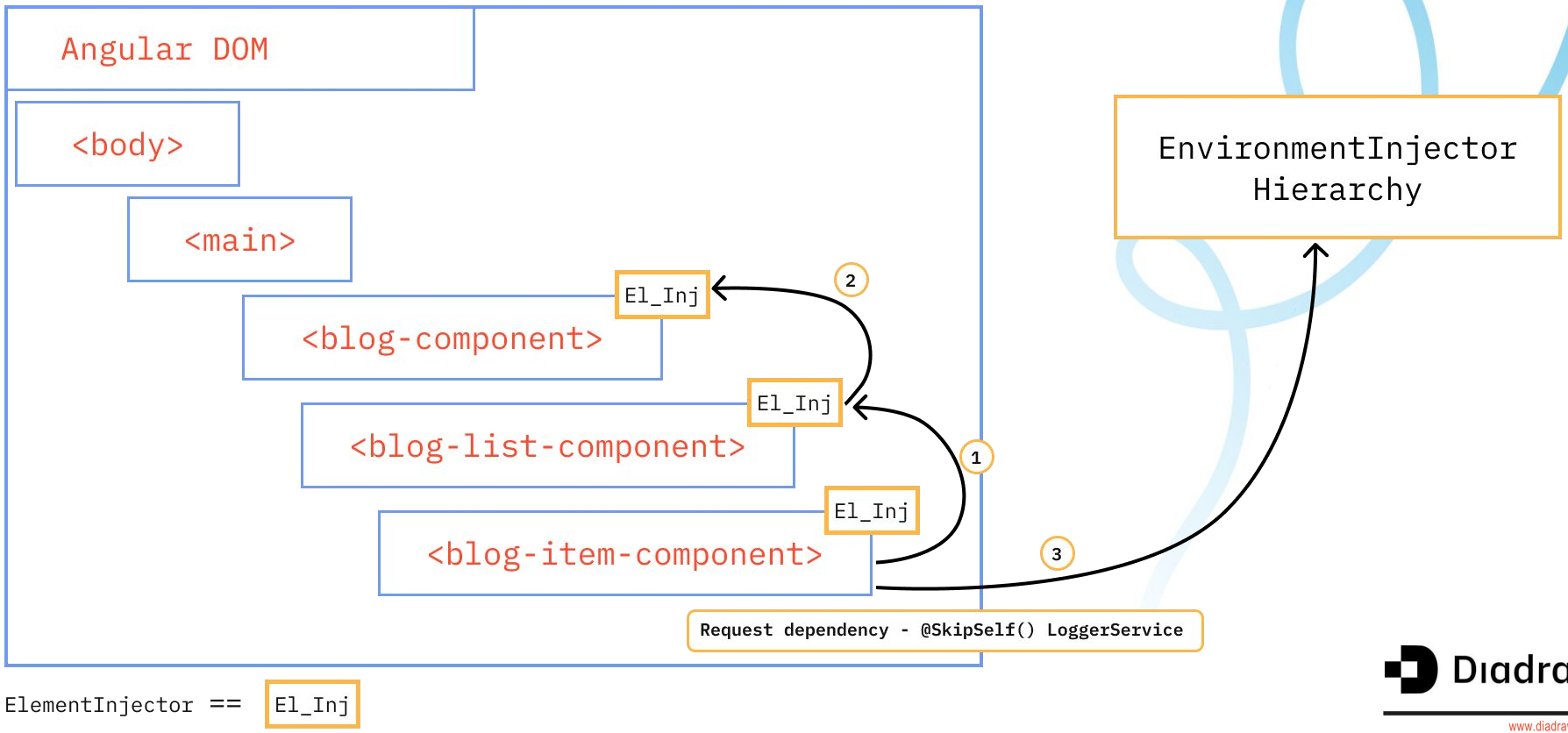
@SkipSelf()

Този декоратор показва на Angular, че трябва да започне търсенето на зависимост от родителския ElementInjector, вместо от текущия.

Позволява ни да игнорираме локална инстанция на зависимост.

TS user.component.ts

```ts
import { Component, Optional, SkipSelf } from '@angular/core';
import { LoggerService } from '../services/logger.service';

@Component({
  selector: 'app-user',
  standalone: true,
  providers: [LoggerService],
  templateUrl: './user.component.html',
})
export class UserComponent {
  constructor(@SkipSelf() private logger: LoggerService) {}
}
```



ElementInjector == El_Inj

Къде да спре да търси за зависимост ?

- @Host()
- @Self()

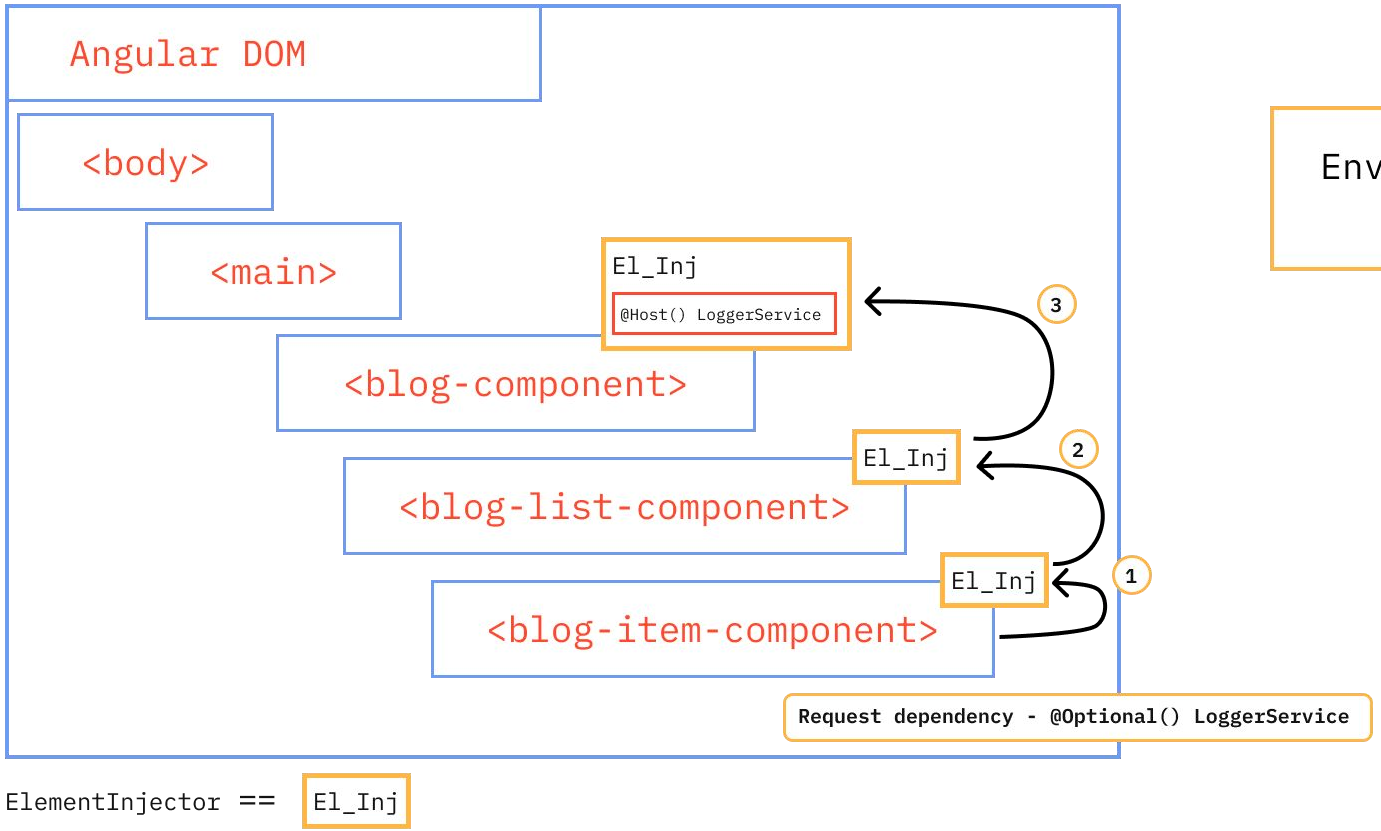
@Host()

Позволява ни да определим крайна точка на търсенето за зависимост (dependency).

Даже и да има инстанция на зависимост по-нагоре по дървото, Angular няма да продължи да търси.

TS user.component.ts

```ts
import { Component, Host } from '@angular/core';
import { LoggerService } from '../services/logger.service';

@Component({
  selector: 'app-user',
  standalone: true,
  providers: [LoggerService],
  templateUrl: './user.component.html',
})
export class UserComponent {
  constructor(@Host() private logger: LoggerService) {}
}
```



EnvironmentInjector Hierarchy

ElementInjector == El_Inj

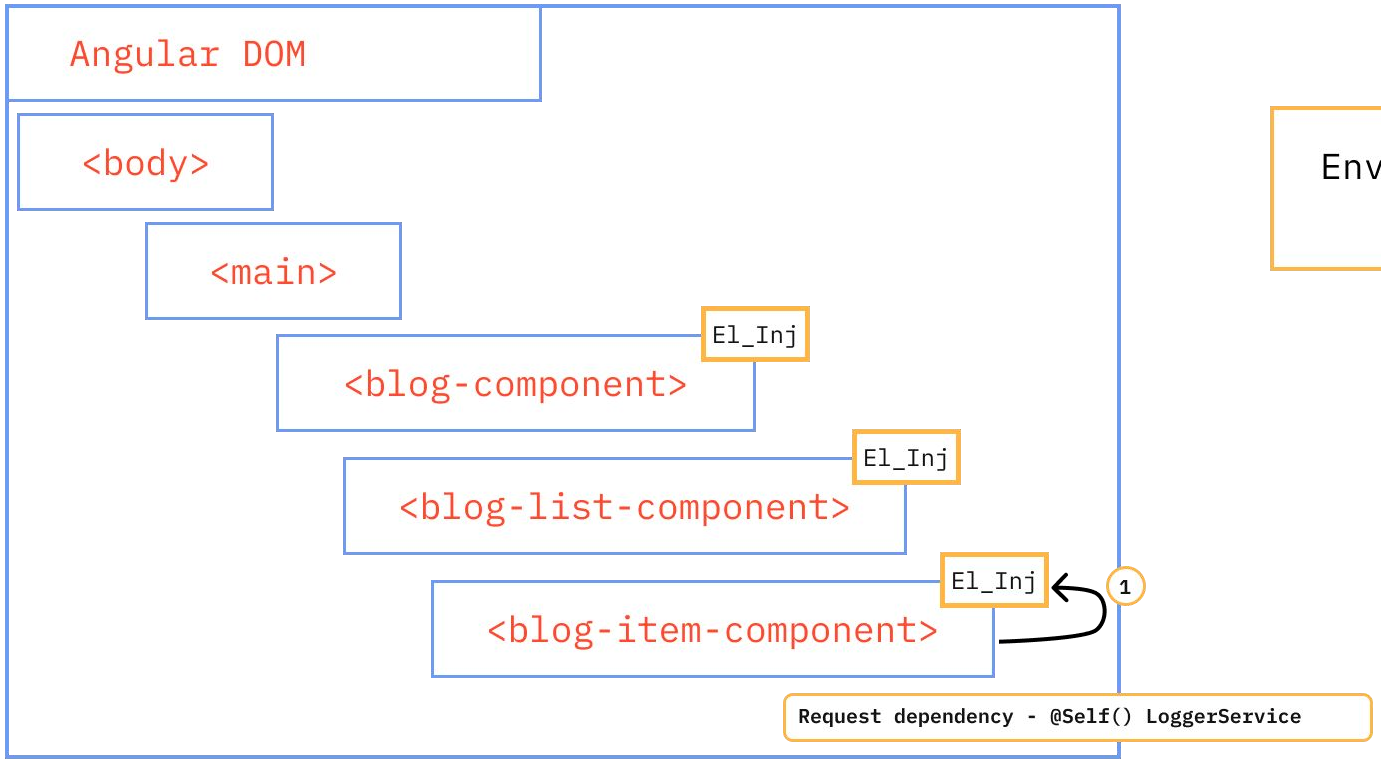
@Self()

@Self() декораторът е противоположен на @SkipSelf. При него Angular търси зависимостта само в текущия ElementInjector.

Ако зависимостта не е намерена ще хвърли грешка.

TS user.component.ts

```ts
import { Component, Self } from '@angular/core';
import { LoggerService } from '../services/logger.service';

@Component({
  selector: 'app-user',
  standalone: true,
  providers: [LoggerService],
  templateUrl: './user.component.html',
})
export class UserComponent {
  constructor(@Self() private logger: LoggerService) {}
}
```



EnvironmentInjector Hierarchy

ElementInjector == El_Inj

Комбиниране на модификатори

Angular ни позволява да комбинираме всички модификатори освен:

- @Host() и @SkipSelf()
- @SkipSelf() и @Self()

TS user.component.ts

```ts
import { Component, Optional, Self } from '@angular/core';
import { LoggerService } from '../services/logger.service';

@Component({
  selector: 'app-user',
  standalone: true,
  providers: [LoggerService],
  templateUrl: './user.component.html',
})
export class UserComponent {
  constructor(@Self() @Optional() private logger: LoggerService) {}
}
```

TS user.component.ts

```ts
import { Component, Host, Optional } from '@angular/core';
import { LoggerService } from '../services/logger.service';

@Component({
  selector: 'app-user',
  standalone: true,
  providers: [LoggerService],
  templateUrl: './user.component.html',
})
export class UserComponent {
  constructor(@Host() @Optional() private logger: LoggerService) {}
}
```

Благодаря за вниманието!