Dependency Injection

SD 555 – Web Application Development III

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Services

- Common layer to interact with APIs
- Communication channel between components
- Provide access to shared global state
- Provide access to shared functionality
- Remove redundant data duplication
- Remove redundant code duplication

Dependency Injection

Dependency Injection **decouples the creation of a dependency from using that dependency**. It promotes loose coupling within our code - a foundation for creating well-architected software.

Practicing good software design patterns yields flexible, maintainable software that allows our applications to grow with new features more quickly.

This way we can change the dependent code without changing the consumer! This technique of injecting the dependencies relies on the **Inversion of Control (IoC)** principle, which is also called informally the Hollywood principle (don’t call us, we’ll call you).

Dependency Injection Parts

The dependency injection framework in Angular has these parts:

- **The Provider** maps a token (**Type**) to a list of dependencies. It tells Angular **what and how to create** an instance.
- **The Injector** holds a set of bindings and is responsible for **resolving dependencies** and injecting them when creating objects.
- **The Injectable** is what's being injected.

In Angular when you want to access an injectable you ask for its **Type**, Angular's dependency injection framework will locate it, create an instance and provide it to you.

Injectors

Every Angular application has one **Root Injector**, which is globally created for the application at the beginning.

Every Component has its own **Local Injector**.

Every route has its own **Route Injector**.

Global Provider

The global injector exists in the **`app.config.ts`** file.

The **`ApplicationConfig`** interface has one property, **`providers`**, which is used to provide **injection tokens**.

```
export const appConfig: ApplicationConfig = {
  providers: [
    // this is root injector
  ]
};
```

Create Custom Services from CLI

To create a new service from Angular CLI:

```
ng g s serviceName
```

By default, services are provided at root, when you provide the service at the root level, Angular creates a single, shared instance of service, and injects it when you ask for it.

How Do We Provide Services?

Globally: We could add it to the root-level **`providers`** array, or use the preferred way of using **`@Injectable({ providedIn: 'root' })`** as it will be tree-shakable, meaning if it is not being used, it will not be part of the production bundle.

Locally: We could add it to an element **`providers`** array of any **component** or **route**.

What Do We Provide?

```
export const PET_NAME = new InjectionToken<string>('Pet_Name');
export const PET_FACTORY = new InjectionToken<{ name: string; }>('Pet_Factory');
export class PET_CLASS { name = 'Theo'; }

const pet_factory = () => {
  const pet_name = inject(PET_NAME);
  return { name: pet_name };
};

export const appConfig: ApplicationConfig = {
  providers: [
    { provide: PET_NAME, useValue: 'Theo' },
    { provide: PET_FACTORY, useFactory: pet_factory },
    // or PET_CLASS (only use for local injectors)
    { provide: PET_CLASS, useClass: PET_CLASS }
  ]
};
```

Asking for a Service

You may ask for a service by **Type**, Angular knows where to find them and provide a singleton instance for you.

```
readonly #pet_name = inject(PET_NAME); // Theo
readonly #pet_factory = inject(PET_FACTORY); // {name: 'Theo'}
readonly #pet_class = inject(PET_CLASS); // {name: 'Theo'}
```

Injection Context

It is where you have access to the **`inject()`** function. The injection context is available in these situations:

- The constructor of a class
- A property of a class
- In a factory function passed to **`useFactory`**
- Any class with an **`@Injectable`**

Dependency Injection Algorithm

When resolving the dependency of a component, Angular will start with the injector of the component itself. Then, if it is unsuccessful, it will climb up to the injector of the parent component, and, finally, will move up to until it reaches the root injector.

```
let inj = this;
while (inj) {
    if (inj.has(requestedDependency)) {
       return inj.get(requestedDependency);
    } else {
        inj = inj.parent;
    }
}
throw new NoProviderError(requestedDependency);
```

Hierarchical Injector

Angular Dependency Injector is **hierarchical**.

The most common pattern is that you provide all your services at root so your application can share the **same instance**.

If providers are specified on each individual component, **different instances** will be created.

Custom Injector

```
// look only in local provider, skip the higher levels
#service = inject(DataService, { self: true })


// could be provided at higher level, if not: null is returned.
#service = inject(DataService, { optional: true })


// it is definitely provided at higher level
#service = inject(DataService, { skipSelf: true})


// service is provided either here or on it's host
#service = inject(DataService, { host: true })
```

APP_INITIALIZER

Angular asks for **APP_INITIALIZER** when the application is initialized. Angular suspends the app initialization until all the functions provided by the **APP_INITIALIZER** are run.

```
providers: [
    { provide: APP_INITIALIZER, useFactory: bootstrap, multi: true}
],
```

```
function bootstrap() {
  const dataService = inject(DataService);
  return () => {
     console.log(`Welcome: `, dataService.fullname)
  }
}
```

Main Points

Dependency injection is a key component of Angular.

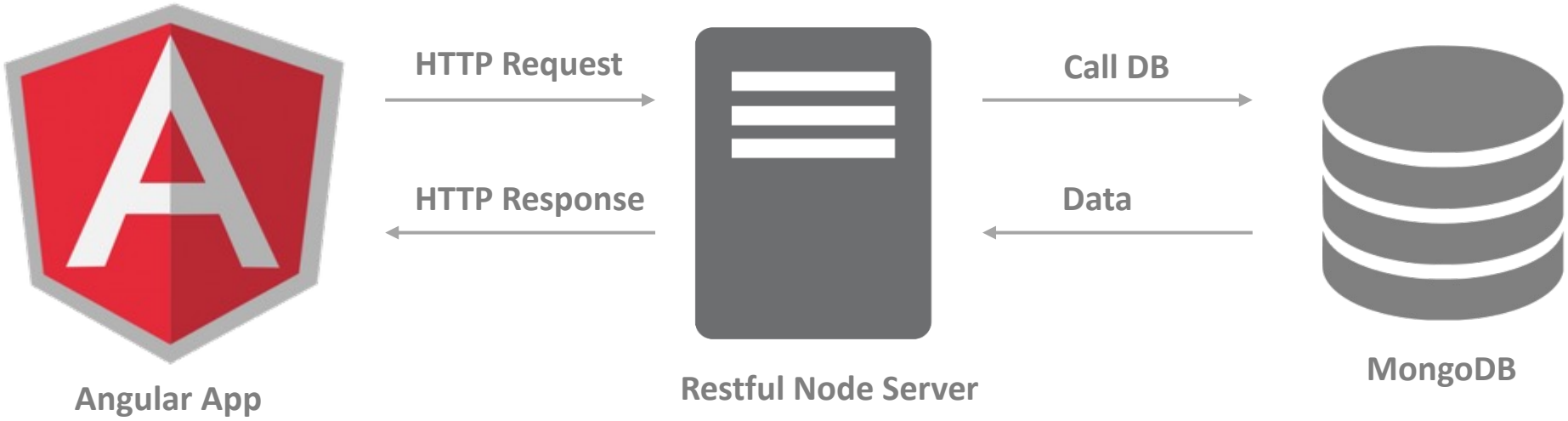
You can configure dependency injection locally at the component level, the route level, or globally at the root level.

Dependency injection allows us to depend on interfaces rather than concrete types. This results in more decoupled code and improves testability.

Making HTTP requests from Angular

Angular simplifies application programming with the XHR and JSONP APIs.

All requests return async Observable.



Why Observables instead of Promise?

Configure the app to use `HttpClient`

1. Provide the **`HttpClient`** service globally as follows:

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

2. Use the **`HttpClient`** service at any injection context as follows:

```
readonly #http = inject(HttpClient)
```

HttpClient API

An instance of the service has the following signature:

```
HttpClient.verb<T>(URL, {optional body}) // returns an observable of T
```

Service Layers

```
@Injectable({ providedIn: 'root' })
class BookService{
   readonly #http = inject(HttpClient)

   getBooks$ = this.#http.get<Book[]>('http://..')

}
```

Example

```
@Component({ /* ... */
    template: `
        @for(book of $books(); track book._id){
            <div>{{book.name}}</div>
        } `
})
export class BooksComponent {
    $books = signal<Book[]>([]);
    #bookService = inject(BookService)

    constructor() {
       this.#bookService.getBooks$.subscribe(books => this.books.set(books));
    }
}
```

This particular returned Observable is categorized as finite, it is always unsubscribed automatically after getting the result. You will need to unsubscribe from any infinite observable to avoid memory leak.

Example

```
@Component({ /* ... */
   imports: [AsyncPipe],
   template: `
      @for(book of books$ | async; track book._id){
            <div>{{book.name}}</div>
       }`
})
export class BooksComponent {
   books$!: Observable<Book[]>;
   readonly #bookService = inject(BookService)

   constructor() {
      this.books = this.#bookService.getBooks$;
   }
}
```

This async pipe takes care of subscribing and unsubscribing automatically, regardless if it's a finite or infinite observable.

Converting Observables to Signals

The **`toSignal()`** function from the **`rxjs-interop`** package will take any Observable, and return a signal that always contains the latest value from that source Observable.

Because signals are synchronous and Observables can potentially be asynchronous and not have an initial value, we can specify an initial value until the Observable emits.

The **`toSignal()`** function returns a **`Signal`**, not a **`WritableSignal`**.

The **`toSignal()`** function also only works in an **injection context**. You need to pass an instance of the **`Injector`** if you want to use it outside an Injection Context.

Example

```
import { toSignal } from '@angular/core/rxjs-interop';

@Component({
  selector: 'app-root',
  standalone: true,
  template: ` {{$quote().quote}} `,
})
export class AppComponent {
  #data = inject(DataService);
  $quote = toSignal(this.#data.get_quote$, { initialValue: { quote: '' } });
}
```

```
export class DataService {
  #http = inject(HttpClient);
  get_quote$ = this.#http.get<{ quote: string; }>('https://api.kanye.rest/');
}
```

HTTP Interceptors

An interceptor sits between the application and a backend API. With interceptors, we can manipulate **ALL requests** going out from our application before they are submitted.

At the same time, all arriving responses from the backend can be intercepted before they are read by the consumer.

You may use the CLI to generate a new interceptor:

```
ng g interceptor interceptorName
```

*Class-based guards/resolvers/interceptors are deprecated as per v15 in favor of functional ones.

Example: Create & Provide Your Interceptor

```
export const addTokenInterceptor = (req, next) => {
  const reqWithToken = req.clone(
    { headers: req.headers.set('Authorization', 'Bearer ...') });

  return next(reqWithToken)
};


providers: [
   provideHttpClient(withInterceptors([addTokenInterceptor]))
]
```