Dependency Injection & Services

Siddharth Ajmera **@SiddAjmera**

Content

- Dependency Injection - Why?
- Dependency Injection - As a design pattern
- Dependency Injection - As a framework
- Dependency Injection - What?
- Injectors and Providers
- Hierarchical Dependency Injection
- Services as a Singleton for Data Sharing
- HttpClient
- Observables
- Component Interaction using a Shared Service

DI - Why?

```
class Car {
  constructor() {
    this.engine = new Engine();
    this.tires = Tires.getInstance();
    this.doors = app.get('doors');
  }
}
```

What's wrong with this code?

- It's brittle.
- Tightly coupled with the dependency classes.
- Can't be unit tested.

How do we make it better?

Credits - thoughtram.io

DI - As a pattern?

```
class Car {
  constructor(engine, tires, doors) {
    this.engine = engine;
    this.tires = tires;
    this.doors = doors;
  }
}
```

```
var car = new Car(
  new Engine(),
  new Tires(),
  new Doors()
);
```

```
var car = new Car(
  new MockEngine(),
  new MockTires(),
  new MockDoors()
);
```

This is better, right?

- The responsibility of creating those dependencies was moved to a higher level.
- Easier to Unit Test.

Still some issues with this.

Credits - thoughtram.io

The Issue

```
function main() {
  var engine = new Engine();
  var tires = new Tires();
  var doors = new Doors();
  var car = new Car(engine, tires, doors);

  car.drive();
}
```

We'll have to maintain this!

- We need to maintain a main function now. No issues.
- BUT, what if the Car class has MANY dependencies?

Hmmm, What's the solution?

Credits - thoughtram.io

DI - As a framework

```
function main() {
  var injector = new Injector(...)
  var car = injector.get(Car);

  car.drive();
}
```

THIS, is the solution!

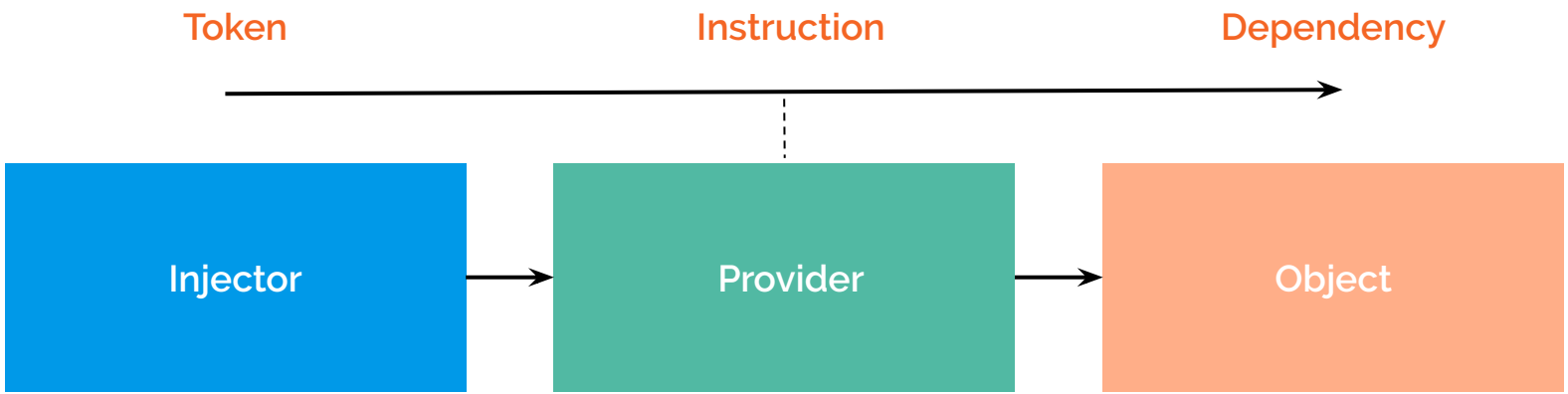
- It's great right? We just get any type of instance we want with this.
- A system will handle everything for us.

It all makes sense now!

Credits - thoughtram.io

DI - What?

It allows us to inject dependencies in different components across our applications, without needing to know, how those dependencies are created, or what dependencies they need themselves.



Credits - thoughtram.io

Injectors & Providers

- Injector - The injector object that exposes APIs to us to create instances of dependencies.
- Provider - A provider is like a recipe that tells the injector how to create an instance of a dependency. A provider takes a token and maps that to a factory function that creates an object.
- Dependency - A dependency is the type of which an object should be created.

```
import { Injector } from '@angular/core';

let injector = Injector.create([
  { provide: Car, deps: [Engine, Tires, Doors] },
  { provide: Engine, deps: [] },
  { provide: Tires, deps: [] },
  { provide: Doors, deps: [] }
]);

let car = injector.get(Car);
```

Credits - thoughtram.io

Types of Providers

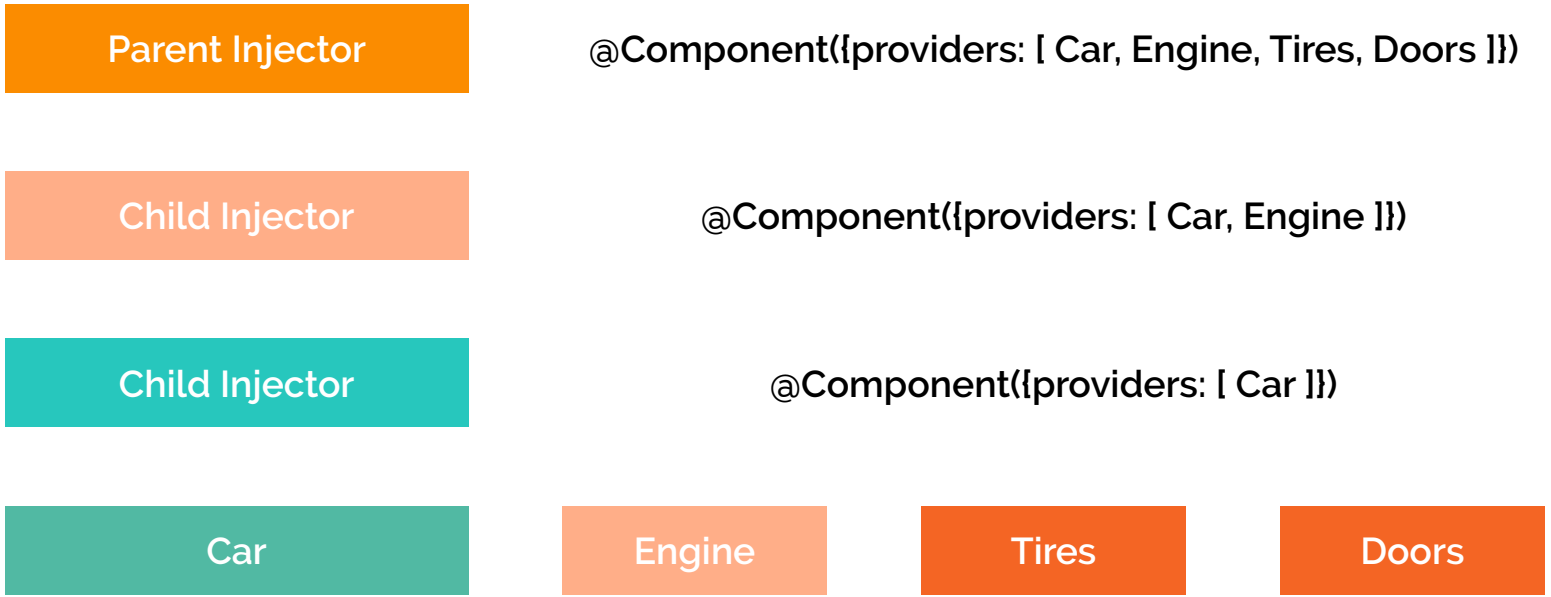
- Normal - providers: [WeatherApiService, AuthService],
- Class Provider - providers: [{ provide: Auth, useClass: UserAuth }]
- Aliased Providers - providers: [{ provide: OldService, useExisting: NewService }]
- Value Providers -

```
const AUTH_CONFIG = {
  apiKey: "...",
  authDomain: "..."
};
providers: [{ provide: AuthConfig, useValue: AUTH_CONFIG }]
```

- Factory Providers -

```
providers: [
    provide: "EmailService",
    useFactory: () => {
     isProd ? return new MandrillService(): return new SendGridService();
    }
]
```

Hierarchical DI



Credits - thoughtram.io

Services

- An angular service is simply a Class that allows you to access it's defined properties and methods
- Services are decorated with @Injectable to allow injection of other services as dependencies
- Services are used to:
  - share the same piece of code across multiple files
  - Hold the business logic
  - Interact with the backend
  - Share data among components
- Services in Angular can be Singletons
- Services are registered on Modules or Components through providers

HttpClient

- Angular Apps can make AJAX Calls to fetch data.
- We use HttpClient for making AJAX Calls. It exposes APIs for all types of AJAX calls like GET, POST, PUT, DELETE, OPTIONS, PATCH, JSONP etc.
- Angular uses RXJS Observables to handle response
- An Observable emit responses overtime, instead of response being a one time event.
- HttpClient can be used with HttpHeaders and HttpParams as well.

```typescript
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';

import { IUser } from '../../interfaces/user';

@Injectable()
export class UserService {

  private _rootUrl: string = 'http://jsonplaceholder.typicode.com/users';

  constructor(private http: HttpClient) { }

  getUsers(): Observable<IUser[]> {
    let headers = new HttpHeaders().set('Authorization', 'Bearer access-token');
    return this.http.get<IUser[]>(this._rootUrl, { headers });
  }

  deleteUser(id: number): Observable<IUser> {
    return this.http.delete<IUser>(`${this._rootUrl}/${id}`);
  }

  createUser(user: IUser): Observable<IUser> {
    user.id = null;
    return this.http.post<IUser>(this._rootUrl, user);
  }

  editUser(user: IUser): Observable<IUser> {
    user.name = 'Sam Kolder';
    user.email = 'Sam.Kolder@domain.com';
    return this.http.put<IUser>(`${this._rootUrl}/${user.id}`, user);
  }

  getPostsByUser(id: number) {
    let url = 'http://jsonplaceholder.typicode.com/posts';
    let httpParams = new HttpParams().set('userId', id.toString());
    return this.http.get<any>(url, { params: httpParams });
  }

}
```

Require imports

Inject HttpClient as a dependency.

Use Methods like get, post, put and delete to perform CRUD Operations. HttpParams and HttpHeaders can also be used to set headers and params in the Request.

Observables

Observable/Observer is a design pattern used for asynchronous programming

- Observable is an object that streams data from some data source
- It streams data to subscribers using push model
- Observers subscribe to these Observables
- Data pushed to subscribers can be transformed on the way from source to subscriber

Observables vs Promise

Multiple values	Single values
Cancellable	Not Cancellable
Operators: Map, filter, reduce, forkJoin	No Operators
Observables can be retried using retry and retryWhen operators	Access the original function to retry

Hot vs Cold Observables

Cold observables start running upon subscription, i.e., the observable sequence only starts pushing values to the observers when Subscribe is called. (…) This is different from hot observables such as mouse move events or stock tickers which are already producing values even before a subscription is active

Cold - Starts streaming data when some code calls subscribe on it. Example playing a pre recorded match.

Hot - Starts streaming data even when no subscriber is interested in receiving data. Example live match.

Subject

Subject is a special observable that act both as observer and observable

- A subject can be subscribed to, just like an observable.
- A subject can subscribe to other observables.

Subject can act as a bridge/proxy between the source Observable and many observers, making it possible for multiple observers to share the same Observable execution.

Subjects can be used for sharing data between components.