SOLID Principles. RxJS. Services

S.O.L.I.D.

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#angular

Table of Contents

Change Detection Strategy

Change Detection Strategy

- Angular performs change detection on all components (from top to bottom) every time something changes
- Change detection is very performant, but as an app gets more complex and the amount of components grows, change detection will have to perform more and more work

Change Detection Strategy

- The strategy that the default change detector uses to detect changes

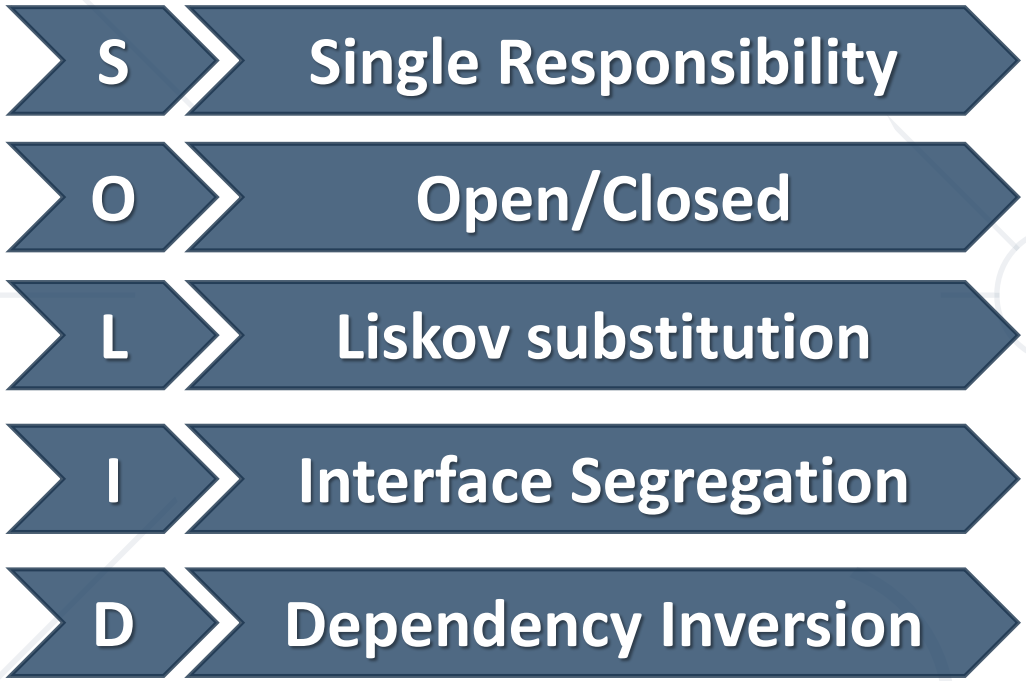
```
enum ChangeDetectionStrategy {
  OnPush: 0,
  Default: 1
}
```

- When set, takes effect the next time change detection is triggered

Change Detection Strategy - Members

Software University

- **OnPush: 0** - CheckOne strategy
  - Automatic change detection is deactivated until reactivated by setting the strategy to Default
  - This strategy applies to all child directives and cannot be overridden
- **Default: 1** - CheckAlways strategy
  - Use the default CheckAlways strategy
  - Change detection is automatic until explicitly deactivated



SOLID Principles

Single Responsibility Principle

- A **responsibility** can be defined as a **reason to change**
- Every class should have **only one** responsibility
  - The responsibility should be entirely **encapsulated** by the class
- This principle leads to
  - **Stronger cohesion** and **looser coupling**
  - Better **readability**
  - **Lower** complexity

Open-Closed Principle

- Software entities like **classes**, **modules** and **functions** should be **open** for **extension**, but **closed** for **modification**
- **Open** for extension
  - Adding new behavior **doesn't require** changes over existing source code
- **Closed** for modification
  - Changing the source code is **not allowed**

Liskov Substitution Principle

- Derived types must be completely **substitutable** for their base types
- Derived classes
  - Only **extend** functionalities of the base class
  - Must **not** remove **base** class **behavior**

Interface Segregation Principle

- Classes that implement **interfaces**, should **not** be **forced** to implement **methods** they **do not use**
- "**Fat**" interfaces need to be divided into "**role**" interfaces (**small** and more **specific**)
- It is **better** to have many **smaller** interfaces, than fewer, **fatter** ones

Dependency Inversion Principle

Software University

- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend on details. Details should depend on abstractions

Dependency Inversion Principle

- The design principle does not just change the direction of the dependency
- Splits the dependency between the high-level and low-level
  - The high-level module depends on the abstraction
  - The low-level depends on the same abstraction

Dependency Injection

- Dependency is another **object** that your class **needs**
  - Examples (**Framework**, **Database**, **File System**, **Providers**)
- Classes that **dependent** on each other are called **coupled**
- Dependencies are **bad** because they **decrease** reuse

```
public class Customer {
  customerService = new CustomerService('Service');
}
```

Customer class is dependent on concrete service

Dependency Injection

- Dependency Injection is a popular **design pattern**
- Inversion of Control (**IoC**)
  - Dependencies are **pushed** in the class from the **outside**
  - The class does **not** instantiate its dependencies

```
public class Customer {
  private customerService;
  constructor(cService: CustomerService) {
    this.customerService = cService;
  }
}
```

The service comes from outside

Classic Violations

- Using the **new** keyword
- Using **static** methods/properties

```
public class Laptop {
  public battery: Battery;
  public videoCard: VideoCard;

  constructor() {
    this.battery = new Battery('Acer battery');
    this.videoCard = new VideoCard('Nvidia 960 GTX');
  }
}
```

The class is **brittle**, **inflexible** and **hard** to test

How to Fix?

- Add the dependencies **through** the **constructor**

```
constructor(
      public videoCard: VideoCard,
      public battery: Battery)
```

- Create whatever **model** you like

```
let firstLaptop = new Laptop(
  new VideoCard('Nvidia 940m'),
  new Battery('Acer Battery'));
```

```
let secondLaptop = new Laptop(
  new VideoCard('Radeon 280x'),
  new Battery('Toshiba Battery'));
```

General Requirements

- A class should **receive** its dependencies from **external** sources rather than **creating** them **itself**
- **Decouple** dependencies through **constructor injection**
- Your **code** should be **easier** to test
- Additional information
  https://angular.io/guide/dependency-injection-overview

Services

Constructor Injection, Providers, Injectable

Why We Need Services ?

- Components **shouldn't** fetch or save **data** directly
- They should focus on **presenting data** and delegate data access to a service
- Services are a great way to
  - Share information among classes that **don't know** about each other
  - Avoid **code duplication**

Create a Service

- Services in Angular are just normal **TypeScript classes** that handle data manipulation

```
export class BooksService {
  booksData: Book[];

  addBook(b: Book) {
    this.booksData.push(b);
  }
}
```

Injecting into Components

- Services are injected into components via **constructor injection**
- Before that they should be **provided** from inside the **decorator**

```
@Component({
  providers: [ BooksService ]
})
export class BookListComponent {
  constructor(
    private booksService: BooksService
  ) {  }
}
```

The **same instance** will be provided for **child components**

Injectable Decorator

- In order to inject **one** service **into another** use the **@Injectable** decorator

```
@Injectable()
export class BooksService {
  booksData: Book[];

  constructor (
    private loggingService: LoggingService
  ) { }
}
```

```
@Injectable({
    providedIn: 'root'
})
```

Provided in 'app.module.ts'

RxJS and Observables

Intro to FRP

Functional Programming

- Functional Programming is used a lot in JavaScript
  - Easier **array** manipulation using (**map**, **filter**, **reduce**, etc.)
- Front-end programming is **asynchronous**
- Using a **stream** to handle **asynchronous** operations

```
0, 1, 2, 3, 4
```

A stream of numeric values

The Observable

- In Angular we **handle** streams using **Observables**
  - **Create** Streams
  - **Subscribe** to Streams
  - **React** to new values
  - **Combine** streams to build new ones

```
[ 0, 1, 2, 3, 4 ]
```

Handle a stream as an array

Function Reactive Programming

- FRP is a **paradigm** for software development
  - Entire **programs** can be built **uniquely** around the notion of **streams**
  - Create, combine and subscribe to streams
- The core **goal** of FRP
  - Build programs in a **declarative** way
  - Lack of application **state** variables

Introducing RxJS

Software University

- Stands for **R**eactive E**x**tensions for **J**ava**S**cript
  - Install Library

```
npm install rxjs
```

  - Use with CommonJS

```
const { range } = require('rxjs')
const { map, filter } = require('rxjs/operators')
```

  - Use with import/export

```
import { range } from 'rxjs'
import { map, filter } from 'rxjs/operators'
```

Observables Side Effect (Hot vs Cold)

Software University

- Using the **tap** operator

```
const obs = range(1, 10)
  .pipe(
    tap(i => console.log(`Hello: ${i}`))
  );
```

- Observables are either **hot** or **cold**
  - **Cold observables** are observables where the data producer is created by the observable itself

of, **from**, **range**, **interval** and **timer**

  - **Hot observables** have their data producer outside the observable itself

fromEvent

Observables Side Effect

- Observables are **not shared** by **default**
  - Creating a **subscriber** sets up a whole **new** separate processing **chain**

```
obs.subscribe(i => console.log(`first sub ${i}`));
obs.subscribe(i => console.log(`second sub ${i}`));
```

- **Two** things to **keep** in mind:
  - Is the observable **hot** or **cold**?
  - Is the observable **shared** or **not**?

Commonly Used RxJS Operators

Software University

- The **map** operator

```
const obs = range(1, 10).pipe(map(i => i ** 2));
```

- The **filter** operator

```
const obs = range(1, 10).pipe(filter(i => i % 2 === 0));
```

- The **reduce** operator

```
const obs = range(1, 10).pipe(
  reduce((prevVal, val) => prevVal + val, 0)
))
```

RxJS and FRP Overview

- RxJS and FRP are powerful concepts
- Multiple choice to **structure** an **Angular** app
  - Go **full** reactive (extensive use of RxJS)
  - Via **parts** (Forms or Http)
- More on observables here: Click
- More RxJS operators here: Click

HTTP Client

Fetching Data from a Remote API

The HTTP Client Module

- Before using the **HttpClient** to **fetch** data, import the **HttpClientModule** in "**app.module.ts**"

```
import { HttpClientModule } from '@angular/common/http'
```

- Add the module in **imports** array

```
@NgModule({
declarations:[ // App Components ],
imports:[
    BrowserModule,
    HttpClientModule
],
```

From now on **HttpClient** can be **injected** in Services

Using the HTTP Client in Services

```
@Injectable()
export class PostsService {
  constructor(
    private http : HttpClient
  ) { }

  getAllPosts() : Observable<Post[]> {
    const url = 'https://jsonplaceholder.typicode.com/posts';

    return this.http.get<Post[]>(url);
  }
}
```

Inject the HttpClient and use it as a service

The Client works with generic types

Subscribe to the Observable

- **Inject** a service and **subscribe** to observables

```
export class PostsComponent implements OnInit {
  posts: Posts[];
  constructor(
   private postsService : PostsService
  ) { }

  ngOnInit(): void {
   this.postsService.getAllPosts()
       .subscribe(data => {
         this.posts = data;
       });
  }
}
```

Always **subscribe** to observables

Type Checking the Response

- It is recommended to **cast** the response

```
getAllPosts() : Observable<Post[]> {
 const url = 'https://jsonplaceholder.typicode.com/posts';

 return this.http.get<Post[]>(url);
}
```

Should be an interface

```
interface Post {
  userId: number;
  id: number;
  title: string;
  body: string;
}
```

Handling Errors

- To handle **errors,** add an error **handler** to **subscribe** call

```
ngOnInit(): void {
this.postsService.getAllPosts()
  .subscribe(
    data => { // Attach data to prop },
    err => {
      console.log(`${JSON.stringify(err)}`)
    }
  )
}
```

Summary

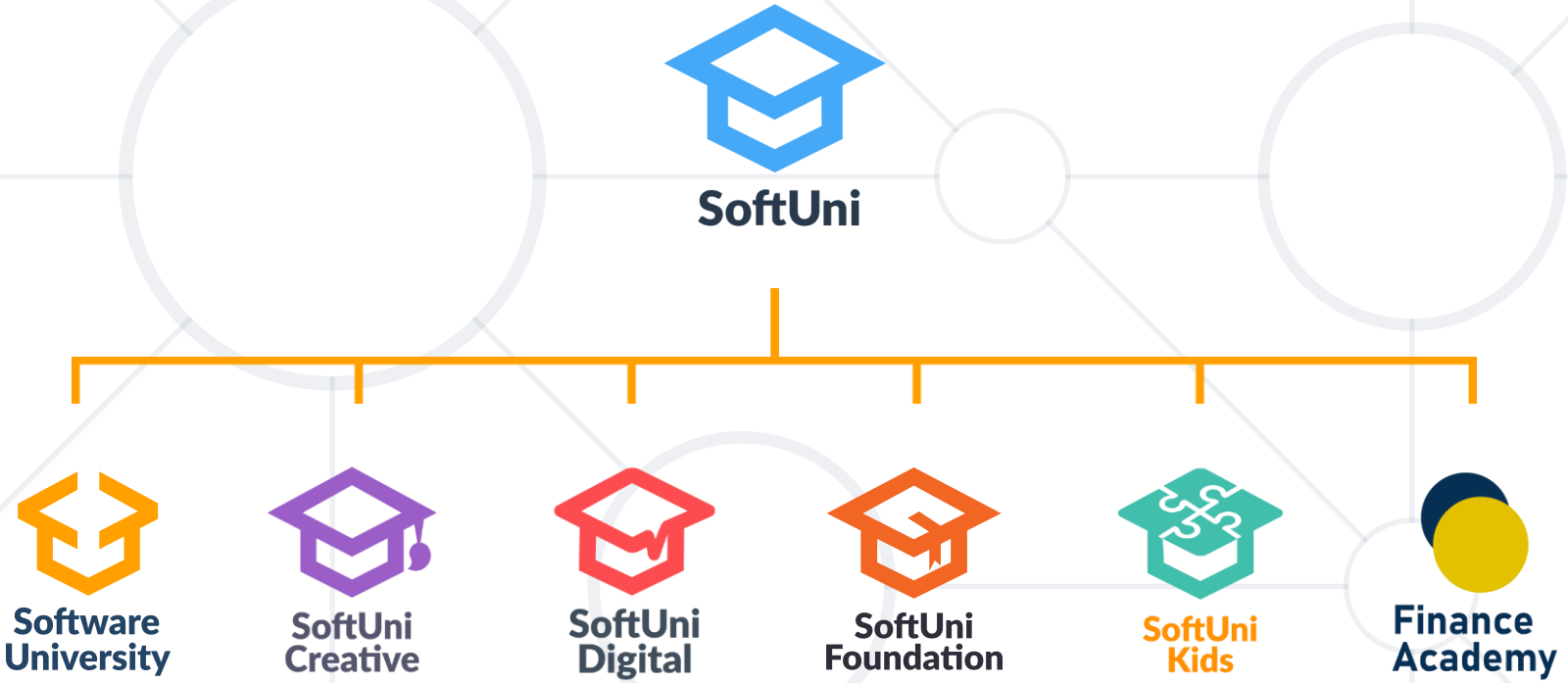
- DI is a popular design pattern
- Using services in Angular is recommended

```
@Component({
  providers: [ UsersService ]
})
```

- RxJS and FRP are powerful concepts
- Use the HttpClient to fetch data from an API

```
this.http.get(url).retry().pipe().subscribe()
```

Questions?



SoftUni Diamond Partners

Software University

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg