R.M.K GROUP OF ENGINEERING INSTITUTIONS

R.M.K
GROUP OF INSTITUTIONS

R.M.K
GROUP OF
INSTITUTIONS

R.M.K GROUP OF INSTITUTIONS

Please read this disclaimer before proceeding:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

R.M.K GROUP OF INSTITUTIONS

22CS402

Web Development Frameworks

Department : CSE & AI&DS

Batch / Year : 2022 - 2026 / II

Created by : Dr. K Manikannan &

Mr. S Vijayakumar,

Dr. Siva Subramaniam

Ms. Saranya

Mr. L. Maria MichaelVisuwasam

Ms. J. Bhuvaneswari

Date: 03.03.2024

1. Contents

Content – Unit 3

S.No.	Contents
1	Dependency injection in Angular
2	Reactive programming in Angular
3	Laying out pages with Flex Layout
4	Implementing component communications
5	Implementing component communications
6	Change detection
7	Component lifecycle

2. Course Objectives

The Course will enable learners to:

- Simplify website development using Spring Boot as server-side technologies.
- Build single page applications using REACT as a reusable UI component technology as client-side technology.
- Assemble REACT as a front end technology and Node js as a server side technology to develop enterprise applications
- Develop a scalable and responsive web application using Angular development Frameworks
- Develop an industry ready application web enterprise feature using Angular framework

3. Prerequisites

R.M.K GROUP OF INSTITUTIONS

4. Syllabus

22CS301	Web Development Frameworks	L	T	P	C
		3	0	2	4

OBJECTIVES:

The Course will enable learners to:

- Simplify website development using Spring boot as server-side technologies.
- Build single page applications using REACT as a reusable UI component technology as client-side technology.
- Assemble REACT as a front end technology and Node js as a server side technology to develop enterprise applications
- Develop a scalable and responsive web application
- Develop an industry ready application web enterprise feature

UNIT I	SPRING BOOT AND STRUTS	9 +6

Spring Boot: Introducing Spring Boot, getting started with spring Boot, Common Spring Boot task-Managing configuration, creating custom properties, executing code on Spring Boot application startup, Database access with Spring data, Securing Spring Boot application.

List of Exercise/Experiments:

1. Use Spring Boot to build a Web Application
2. Create REST Service for an Education Site

UNIT II	JAVA REACT	9 +6

React: Introduction to React, Pure React- The Virtual DOM, React Elements, React with JSX, Props, State, and the Component Tree, Enhancing Components- Flux.

List of Exercise/Experiments:

1. Build Search filter in React
2. Display a list in React
3. Create Simple Login form in React

UNIT III	Node JS	9 +6

Node JS: Introduction to Node JS, Setting up Node.js, Node.js Modules- Finding and loading CommonJS and JSON modules using require, Hybrid CommonJS/Node.js/ES6 module scenarios, npm - the Node.js package management system.

List of Exercise/Experiments:

1. Write a node.js program for making external http calls
2. Write a program in node.js to parse the given url.

4. Syllabus Contd...

UNIT IV	WEB FRAMEWORK (ANGULAR) – I	9+6
Introduction- Angular First App, Angular UI with Bootstrap CSS Authentication, Authentication Service, Unsubscribe, Logout and Route Guard Cleanup, Customer Service ,Http Service, Token Interceptor, Multi Provider, Compile-time Configuration, Runtime Configuration, Error Handling. **List of Exercise/Experiments:** 1. Create a Dropdown using Angular UI bootstrap 2. Modify existing components and generating new components using Angular		
UNIT V	**WEB FRAMEWORK (ANGULAR) – II**	**9+6**
Dependancy injection in Angular,Reactive programming in Angular, Laying out pages with Flex Layout, Implementing component communications, Change detection and component lifecycle. **List of Exercise/Experiments:** 1. Launching your app with Angular root module		

4. Syllabus Contd...

OUTCOMES:

Upon completion of the course, the students will be able to:

CO1: Write Web API/RESTful API application programming interface to communicate with Spring boot as a serverside technology.

CO2: Build single page applications using REACT as a reusable UI component technology as client side technology

CO3: Build applications using Node Js as server side technologies

CO4: Able to develop a web application using latest Angular Framework

CO5: Apply various Angular features including directives, components, and services.

TEXT BOOK:

1. Somnath Musib, Spring Boot in Practice, Manning publication, June 2022 (https://www.manning.com/books/spring-boot-in-practice)
2. Alex Banks, Eve Porcello , "Learning React", May 2017, O'Reilly Media, Inc. ISBN: 9781491954621. (https://www.oreilly.com/library/view/learning- react/9781491954614/)
3. David Herron ,"Node.js Web Development - Fourth Edition",2018, Packt Publishing,ISBN: 9781788626859
4. Sukesh Marla, "A Journey to Angular Development Paperback ", BPB Publications.(https://in.bpbonline.com/products/a-journey-to-angular-development?_pos=1&_sid=0a0a0e9fb&_ss=r)
5. Yakov Fain Anton Moiseev, "Angular Development with TypeScript", 2nd Edition.(https://www.manning.com/books/angular-development-with-typescript-Second Edition.

REFERENCES:

1. Sue Spielman, The Struts Framework 1: A Practical guide for Java Programmers||, 1st Edition. Elsevier 2002

LIST OF EQUIPMENTS:

VSCode, Angular JS, React JS, Node JS, Ruby, Django

5. Course Outcomes

Upon completion of the course, the students will be able to:

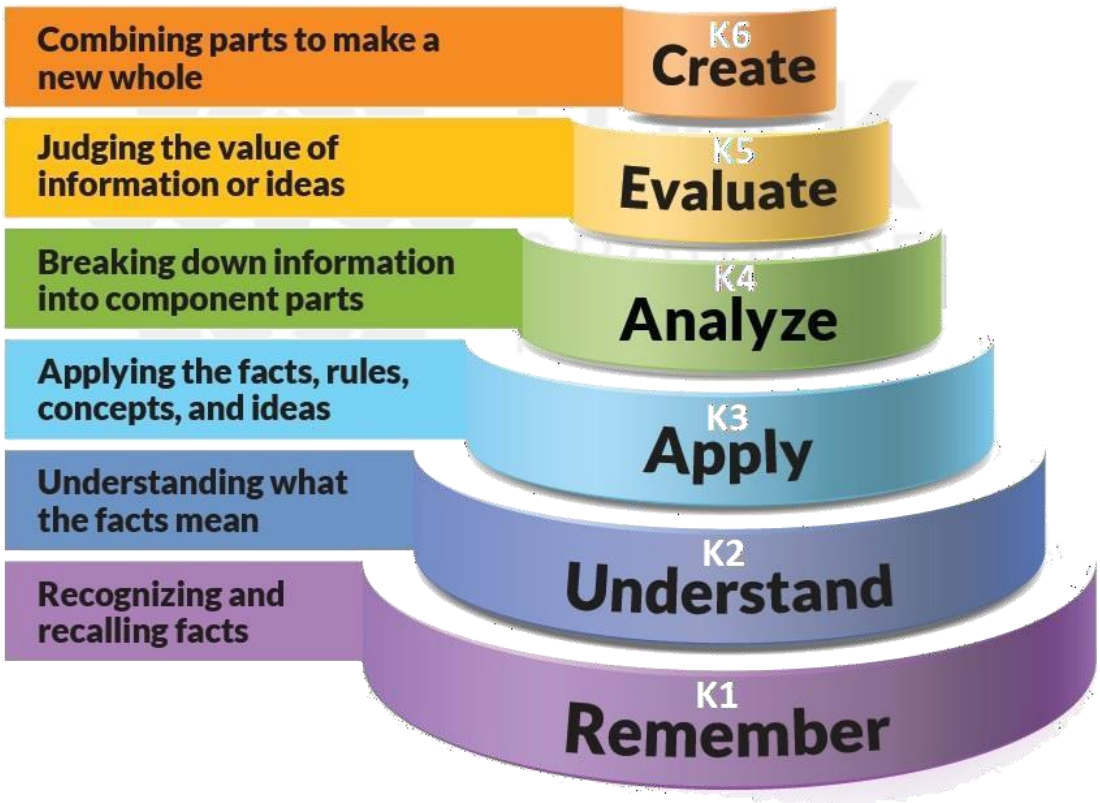
CO1: Write Web API/RESTful API application programming interface to communicate with Spring boot as a serverside technology.

CO2: Build single page applications using REACT as a reusable UI component technology as client side technology

CO3: Build applications using Node Js as server side technologies

CO4: Able to develop a web application using latest Angular Framework

CO5: Apply various Angular features including directives, components, and services.



6. CO - PO Mapping

	POs and PSOs														
COs	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
CO1	2	3	2	2	2	3	2	1	1	3	2	2	3	2	2
CO2	2	3	2	2	2	3	1			3	2	2	3	2	2
CO3	2	3	2	2	2	1	2	1	1	3	2	2	3	2	2
CO4	2	3	2	2	1	3	2	1	1	3	2	2	3	2	2
CO5	2	3	2	2	2	3	2	1	1	3	2	2	3	2	2

7. Lecture Plan - Unit V

S. No.	Topic	No. of Periods	Proposed Date	Actual Lecture Date	Pertaining CO	Taxonomy Level	Mode of Delivery
1	Dependency injection in Angular	1			CO5	K2	Chalk & Talk
2	Reactive programming in Angular	1			CO5	K3	Chalk & Talk
3	Laying out pages with Flex Layout	1			CO5	K2	Chalk & Talk
4	Implementing component communications	1			CO5	K2	Chalk & Talk
5	Implementing component communications	1			CO5	K2	Chalk & Talk
6	Change detection	1			CO5	K2	Chalk & Talk
7	Change detection	1			CO5	K3	Chalk & Talk
8	Component lifecycle	1			CO5	K2	Chalk & Talk
9	Component lifecycle	1			CO5	K2	Chalk & Talk

8. Activity Based Learning

Learning Method	Activity
Learn by Solving Problems	Tutorial Sessions available in iamneo Portal
Learn by Questioning	Quiz / MCQ Using RMK Nextgen App and iamneo Portal
Learn by doing Hands-on	Practice available in iamneo Portal

iamneo RMK Nextgen

9. Lecture Notes

Dependency injection in Angular, Reactive programming in Angular, Laying out pages with Flex Layout,
Implementing component communications,
Change detection and component lifecycle

9.LECTURE NOTES

Dependency injection in Angular

Injectors

Injectors are data structures that store instructions detailing where and how services form. They act as intermediaries within the Angular DI system.

Module, directive, and component classes contain metadata specific to injectors. A new injector instance accompanies every one of these classes. In this way, the application tree mirrors its hierarchy of injectors.

The providers: [] metadata accepts services that then register with the class' injector. This provider field adds the instructions necessary for an injector to function. A class (assuming it has dependencies) instantiates a service by taking on its class as its data type. The injector aligns this type a creates an instance of that service on the class' behalf.
Of course, the class can only instantiate what the injector has instructions for. If the class' own injector does not have the service registered, then it queries its parent. So on and so forth until either reaching an injector with the service or the application root.

Services can register at any injector within the application. Services go in the providers: [] metadata field of class modules, directives, or components. The class' children can instantiate a service registered in the class' injector. Child injectors fallback on parent injectors after all.

Dependency Injection

Take a look at the skeletons for each class: service, module, directive, and component.

```
// service

        constructor() { }
}
    // module
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
       imports: [
    CommonModule
  ],
  declarations: [],
  providers: [ /* services go here */ ]
    })
    export class TemplateModule { }
    // directive

    import { Directive } from '@angular/core';

    @Directive({
      selector: '[appTemplate]',
      providers: [ /* services go here */ ]
    })
    export class TemplateDirective {
  constructor() { }
    }
    //component

import { Component } from '@angular/core';

@Component({
  selector: 'app-template',
  templateUrl: './template.component.html',
  styleUrls: ['./template.component.css'],
       providers: [ /* services go here */ ]
})
export class TemplateComponent {
  // class logic ...
```

}

Service

The providedIn: string metadata of @Injectable specifies which injector to register with. Using this method, and depending on if the service gets used, the service may or may not register with the injector. Angular calls this *tree-shaking*.

By default the value is set to 'root'. This translates to the root injector of the application. Basically, setting the field to 'root' makes the service available anywhere.

Module, Directive, and Component

Modules and components each have their own injector instance. This is evident given the providers: [] metadata field. This field takes an array of services and registers them with the injector of the module or component class. This approach happens in the @NgModule, @Directive, or @Component decorators.

This strategy omits *tree-shaking*, or the optional removal of unused services from injectors. Service instances live on their injectors for the life of the module or component.

Instantiating References

References to the DOM can instantiate from any class. Keep in mind that references are still services. They differ from traditional services in representing the state of something else. These services include functions to interact with their reference.

Directives are in constant need of DOM references. Directives perform mutations on their host elements through these references. See the following example. The directive's injector instantiates a reference of the host element into the class' constructor.

```
// directives/highlight.directive.ts

import { Directive, ElementRef, Renderer2, Input } from
'@angular/core';

@Directive({
  selector: '[appHighlight]'
})
    export class HighlightDirective {
  constructor(
    private renderer: Renderer2,
```

```
    private host: ElementRef
  ) { }

  @Input() set appHighlight (color: string) {
    this.renderer.setStyle(this.host.nativeElement,  'background-color',
color);
      }
    }
// app.component.html

    <p [appHighlight]="'yellow'">Highlighted Text!</p>
```

Renderer2 also gets instantiated. Which injector do these services come from? Well, each service's source code comes from @angular/core. These services must then register with the application's root injector.

```
    import { BrowserModule } from '@angular/platform-browser';
    import { NgModule } from '@angular/core';
    import { AppComponent } from './app.component';
    import { HighlightDirective } from './directives/highlight.directive';

@NgModule({
      declarations: [
        AppComponent,
        HighlightDirective
],
      imports: [
    BrowserModule
      ],
      providers: [],
  bootstrap: [
    AppComponent
  ]
})
    export class AppModule { }
```

Instantiating Services:

The Services and Injectors article explains this section to an extent. Though, this section rehashes the previous section or the most part. Services will often provide references to something else. They may just as well provide an interface extending a class' capabilities.
The next example will define a logging service that gets added to a component's injector via its providers: [] metadata.

```
// services/logger.service.ts

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

@Injectable()
export class LoggerService {
  callStack: string[] = [];

  addLog(message: string): void {
    this.callStack = [message].concat(this.callStack);
    this.printHead();
  }

  clear(): void {
    this.printLog();
    this.callStack = [];
    console.log("DELETED LOG");
  }

  private printHead(): void {
    console.log(this.callStack[0] || null);
  }

  private printLog(): void {
    this.callStack.reverse().forEach((log) => console.log(message));
  }
}
// app.component.ts
```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [LoggerService]
})
export class AppComponent {
  constructor(private logger: LoggerService) { }

  logMessage(event: any, message: string): void {
    event.preventDefault();
    this.logger.addLog(`Message: ${message}`);
  }

  clearLog(): void {
    this.logger.clear();
  }
}
// app.component.html

<h1>Log Example</h1>
<form (submit)="logMessage($event, userInput.value)">
  <input #userInput placeholder="Type a message...">
  <button type="submit">SUBMIT</button>
</form>

<h3>Delete Logged Messages</h3>
<button type="button" (click)="clearLog()">CLEAR</button>
```

Focus on the AppComponent constructor and metadata. The component injector receives instructions from the provider's metadata field containing LoggerService. The injector then knows what to instantiate LoggerService from requested in the constructor.

The constructor parameter loggerService has the type LoggerService which the injector recognizes. The injector follows through with the instantiation as mentioned.

REACTIVE PROGRAMMING IN ANGULAR

Reactive programming is a programming paradigm dealing with data streams and the propagation of changes. Data streams may be static or dynamic. An example of static data stream is an array or collection of data. It will have an initial quantity and it will not change. An example for dynamic data stream is event emitters. Event emitters emit the data whenever the event happens. Initially, there may be no events but as the time moves on, events happens and it will gets emitted.

Reactive programming enables the data stream to be emitted from one source called **Observable** and the emitted data stream to be caught by other sources called **Observer** through a process called subscription. This Observable / Observer pattern or simple **Observer** pattern greatly simplifies complex change detection and necessary updating in the context of the programming.

JavaScript does not have the built-in support for Reactive Programming. **RxJs** is a JavaScript Library which enables reactive programming in JavaScript. Angular uses **RxJs** library extensively to do below mentioned advanced concepts –

- Data transfer between components.
- HTTP client.
- Router.
- Reactive forms.

Observable

As learn earlier, **Observable** are data sources and they may be static or dynamic. **Rxjs** provides lot of method to create **Observable** from common JavaScript Objects. Let us see some of the common methods.

of – Emit any number of values in a sequence and finally emit a complete notification.

```
const numbers$ = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Here,

- **numbers$** is an **Observable** object, which when subscribed will emit 1 to 10 in a sequence.
- **Dollar sign ($)** at the end of the variable is to identify that the variable is Observable.

range – Emit a range of number in sequence.

```
const numbers$ = range(1,10)
```

from – Emit array, promise or iterable.

```
const numbers$ = from([1,2,3,4,5,6,7,8,9,10]);
```

ajax – Fetch a url through AJAX and then emit the response.

```
const api$ = ajax({ url: 'https://httpbin.org/delay/1', method: 'POST', headers: { 'Content-Type': 'application/text' }, body: "Hello" });
```

Here,

https://httpbin.org is a free REST API service which will return the supplied body content in the JSON format as specified below –


```
{
  "args": {},
  "data": "Hello",
  "files": {},
  "form": {},
  "headers": {
    "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
    "Accept-Encoding": "gzip, deflate, br",
    "Accept-Language": "en-US,en;q=0.9",
    "Host": "httpbin.org", "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "none",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.106
```

```
Safari/537.36",
      "X-Amzn-Trace-Id": "Root=1-5eeef468-015d8f0c228367109234953c"
   },
   "origin": "ip address",
   "url": "https://httpbin.org/delay/1"
}
```

fromEvent – Listen to an HTML element's event and then emit the event and its property whenever the listened event fires.

```
const clickEvent$ = fromEvent(document.getElementById('counter'), 'click');
```

Angular internally uses the concept extensively to provide data transfer between components and for reactive forms.

Subscribing process

Subscribing to an Observable is quite easy. Every Observable object will have a method, subscribe for the subscription process. Observer need to implement three callback function to subscribe to the Observable object. They are as follows –

- **next** – Receive and process the value emitted from the Observable
- **error** – Error handling callback
- **complete** – Callback function called when all data from Observable are emitted.

Once the three callback functions are defined, Observable's subscribe method has to be called as specified below –

```
const numbers$ = from([1,2,3,4,5,6,7,8,9,10]);
// observer
const observer = {
   next: (num: number) => {       this.numbers.push(num); this.val1 += num },
      error: (err: any) => console.log(err),
      complete: () => console.log("Observation completed")
};
numbers$.subscribe(observer);
```

Here,

- **next** – method get the emitted number and then push it into the local variable, **this.numbers**.
- **next** – method also adding the number to local variable, **this.val1**.
- **error** – method just writes the error message to console.
- **complete** – method also writes the completion message to console.

We can skip **error** and **complete** method and write only the **next** method as shown below –

```
number$.subscribe((num: number) => { this.numbers.push(num); this.val1 += num; });
```

Operations

Rxjs library provides some of the operators to process the data stream. Some of the important **operators** are as follows –

filter – Enable to filter the data stream using callback function.

```
const filterFn = filter( (num : number) => num > 5 );
const filteredNumbers$ = filterFn(numbers$);
filteredNumbers$.subscribe( (num : number) => {
this.filteredNumbers.push(num); this.val2 += num } );
```

map – Enables to map the data stream using callback function and to change the data stream itself.

```
const mapFn = map( (num : number) => num + num ); const mappedNumbers$ = mappedFn(numbers$);
```

pipe – Enable two or more operators to be combined.

```
const filterFn = filter( (num : number) => num > 5 );
const mapFn = map( (num : number) => num + num ); const processedNumbers$ = numbers$.pipe(filterFn, mapFn);
processedNumbers$.subscribe( (num : number) => { this.processedNumbers.push(num); this.val3 += num } );
```

Let us create a sample application to try out the reaction programming concept learned in this chapter.

Create a new application, reactive using below command –

ng new reactive

Change the directory to our newly created application.

cd reactive

Run the application.

ng serve

Change the AppComponent component code (src/app/app.component.ts) as specified below –

```
import { Component, OnInit } from '@angular/core'; import { Observable,
of, range, from, fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { filter, map, catchError } from 'rxjs/operators';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
   title = 'Reactive programming concept';
   numbers : number[] = [];
   val1 : number = 0;
   filteredNumbers : number[] = [];
   val2 : number = 0;
   processedNumbers : number[] = [];
   val3 : number = 0;
   apiMessage : string;
   counter : number = 0;
   ngOnInit() {
      // Observable stream of data Observable<number>
      // const numbers$ = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
      // const numbers$ = range(1,10);
      const numbers$ = from([1,2,3,4,5,6,7,8,9,10]);
      // observer
      const observer = {
         next: (num: number) => {this.numbers.push(num); this.val1 +=
```

```
num },
        error: (err: any) => console.log(err),
        complete: () => console.log("Observation completed")
      };
      numbers$.subscribe(observer);
      const filterFn = filter( (num : number) => num > 5 );
      const filteredNumbers = filterFn(numbers$);
      filteredNumbers.subscribe(        (num        :        number)        =>
{this.filteredNumbers.push(num); this.val2 += num } );
      const mapFn = map( (num : number) => num + num );
      const processedNumbers$ = numbers$.pipe(filterFn, mapFn);
      processedNumbers$.subscribe(       (num       :       number)       =>
{this.processedNumbers.push(num); this.val3 += num } );
      const api$ = ajax({
        url: 'https://httpbin.org/delay/1',
        method: 'POST',
        headers: {'Content-Type': 'application/text' },
        body: "Hello"
      });
      api$.subscribe(res => this.apiMessage = res.response.data );
      const  clickEvent$  =  fromEvent(document.getElementById('counter'),
'click');
      clickEvent$.subscribe( () => this.counter++ );
   }
}
```

Here,

- Used of, range, from, ajax and fromEvent methods to created Observable.
- Used filter, map and pipe operator methods to process the data stream.
- Callback functions catch the emitted data, process it and then store it in component's local variables.

Change
the **AppComponent** template **(src/app/app.component.html)** as specified below –

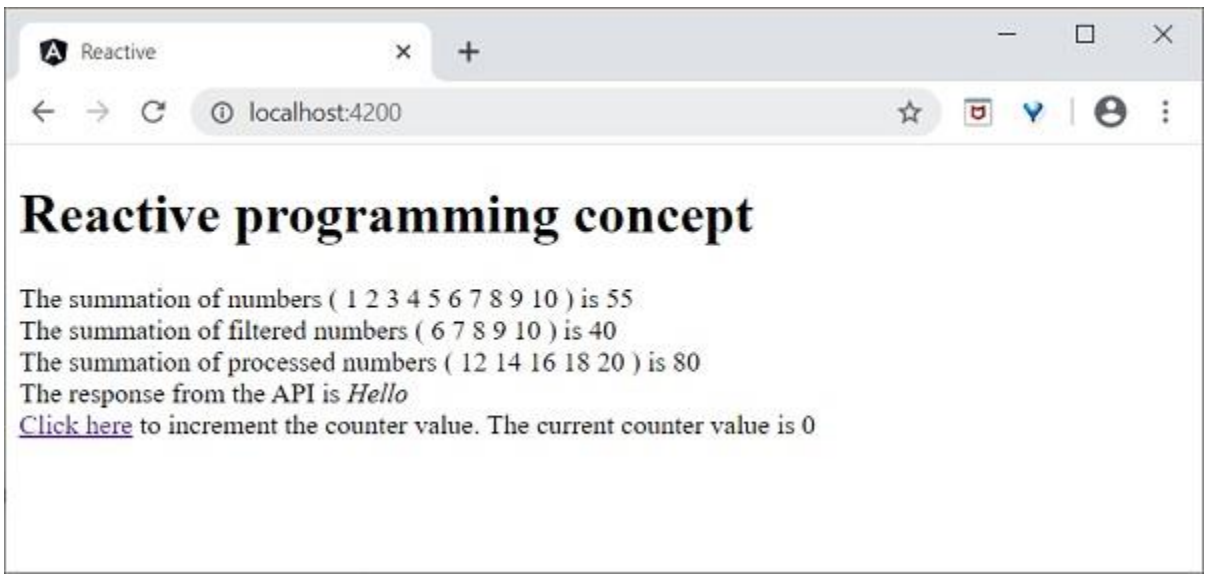
```
<h1>{{ title }}</h1>
<div>
  The summation of numbers ( <span *ngFor="let num of numbers"> {{
num }} </span> ) is {{ val1 }}
</div>
<div>
```

```
  The summation of filtered numbers ( <span *ngFor="let num of filteredNumbers"> {{ num }} </span> ) is {{ val2 }}
</div>
<div>
  The summation of processed numbers ( <span *ngFor="let num of processedNumbers"> {{ num }} </span> ) is {{ val3 }}
</div>
<div>
  The response from the API is <em>{{ apiMessage }}</em> </div>
<div>
  <a id="counter" href="#">Click here</a> to increment the counter value. The current counter value is {{ counter }}
<div>
```

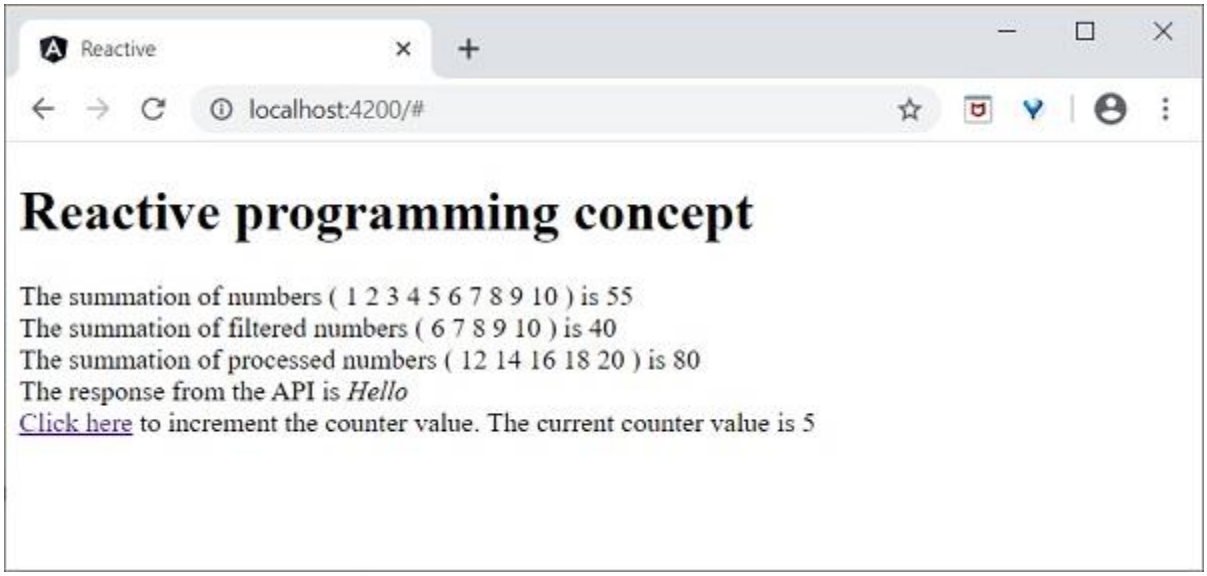
Here,

Shown all the local variable processed by **Observer** callback functions.

Open the browser, http://localhost:4200.



Click the **Click here** link for five times. For each event, the event will be emitted and forward to the **Observer**. Observer callback function will be called. The callback function increments the counter for every click and the final result will be as shown below –



LAYING OUT PAGES WITH FLEX LAYOUT

Angular Flex layout

Angular flex-layout is a stand-alone library developed by the Angular team for designing sophisticated layouts. Angular Layout provides a sophisticated API using Flexbox. The module provides Angular developers with component layout features using a custom layout API.

When creating HTML pages in Angular, we can easily create flexbox-based page layouts using angular flex-layout, which has a set of instructions available for use in your template. It eliminates the need to write a separate CSS style.

Responsive layouts are created using complex CSS code and media queries in normal CSS Flexbox or CSS Grid.

Powerful layout functions

Flex-Layout is a TypeScript-based UI layout engine that uses HTML markup to specify layout configurations.

Flex Layout Instructions for Flexbox Containers

Below are the APIs or Directives used on Flexbox Containers,

- fxlayout
- fxlayout gap
- fxLayoutAlign

- fxlayout

fxLayout is a directive used to define the layout of HTML elements. i.e., it decides the flow of child elements within the flexbox container and should be applied to the parent DOM element i.e. the flexbox container. This directive is case sensitive and the allowed values of fxLayout are row, column, row-reverse, and column-reverse.

```
<div fxLayout="row">
<div class="sub-section-1"></div>
<div class="sub-section-2"></div>
</div>
```

fxLayoutAlign

fxLayoutAlign directive defines the alignment of children elements within the flexbox parent container.

Syntax:

1. **<div** fxLayout="row" fxLayoutAlign="<main-axis> <cross-axis>" **> **

Example:

1. **<div** fxLayout="row" fxLayoutAlign="center start"**> **

fxLayoutGap

fxLayoutGap is a directive that defines the gap between the children items within a flexbox container

Example:

1. **<div** fxLayoutGap="20px"**> **

Angular flex layout directives for flexbox children

The following directives are applicable to flexbox children elements

- fxFlex
- fxFlexOrder
- fxFlexOffset
- fxFlexAlign
- fxFlexFill

What is fxFlex?

fxFlex is one of the most useful and powerful APIs in Angular Flex Layout. It must be used on children elements inside the fxLayout container. It is responsible for resizing elements along the main-axis of the layout.

```
<div fxLayout="row" fxLayoutAlign="start center">
<div fxFlex="30"></div>
<div fxFlex="30"></div>
</div>
```

fxFlexOrder

Defines the order of a flexbox item,

```
<div fxFlexOrder="2"></div>
```

fxFlexOffset

Offset a flexbox item in its parent container flow layout.

```
<div fxFlexOffset="20px"></div>
```

fxFlexAlign

Works like *fxLayoutAlign,* but applies only to a single flexbox item, Instead of all of them.

```
<div fxFlexAlign="center"></div>
```

fxFlexFill

Maximizes the width and height of an element in a layout container.

1. **<div** fxFlexFill**> **

Flex with Align-Self

'flex-align' can change the alignment for a single element only.

1. **<div** fxFlexAlign="baseline"**>**

Understanding of Layout with 'layout-align'

Flex items are laid out in horizontal rows or vertical columns.

Layout Direction

We can defined layout direction as a container *{ fxLayout: row | row-reverse | column | column-reverse; } row (default):* left to right in ltr; right to left in rtl;, which has four possible values:

- row
- row-reverse
- column
- column-reverse

Alignment in Layout Direction (Horizontal)

We can use layout alignment attributes value to adjust item's horizontally, and it's defined at main-axisfxLayoutAlign="". These have seven possible values,

- none
- start (default)
- end
- center
- space-around
- space-between
- space-evenly

Alignment in Perpendicular Direction (Vertical)

The same concept applies to vertical layout alignment attributes:

- none
- start
- center
- end
- stretch (default)

Use the following code to align the Layout vertically and horizontally.

Layout Direction: row

Alignment in LayoutDirection(Horizontal): start

Alignment in Perpendicular Direction(Vertical): center

```
<div fxLayout="row" fxLayoutAlign="start center">
  <div class="child-1"></div>
  <div class="child-2"></div>
</div>
```

Another example:

Layout Direction: column

Alignment in LayoutDirection(Horizontal): end

Alignment in Perpendicular Direction(Vertical): center

```
<div fxLayout="column" fxLayoutAlign="end center">
  <div class="child-1"></div>
 <div class="child-2"></div>
```

Another example:

Layout Direction: column

Alignment in LayoutDirection(Horizontal): end

Alignment in Perpendicular Direction(Vertical): center

```
<div fxLayout="column" fxLayoutAlign="end center">
<div class="child-1"></div>
<div class="child-2"></div>
 </div>
```

Responsive Layout Directions

Angular Flex-Layout responsive API

1. An Angular service is just a JavaScript function. All we have to do is create a class and add methods and properties. Then we can creat e an instance of this class in our component and call its methods.
2. One of the best uses of services is to get data from a data source. Let us create a simple service, which receives product data and sends it to our c omponent.

```
<div fxLayout="row" fxLayout.xs="column" fxLayoutGap="20px">
 <div fxFlex="35"> Child - One </div>
 <div fxFlex="30" fxHide.lt-md> Child - Two </div>
     <div fxFlex="35" fxLayoutAlign="space-around center">
    <div  fxFlex="50" fxFlex.lt-lg="80"> Sub-child - One</div>
     </div>
    </div>
```

Installation of Angular Flex-Layout

Use the following command to use Angular Flex layouts in your projects.

1. npm install --save @angular/flex-layout @angular/cdk

After installing flex layout we need to import flexLayoutModule in our *app.module.ts* file as shown below.

```
1. import { NgModule } from '@angular/core';
2. import { FlexLayoutModule } from '@angular/flex-layout';
3. import {BrowserAnimationsModule} from '@angular/platform-
   browser/animations';
4. import { AppComponent } from './app.component';
5. import { BrowserModule } from '@angular/platform-browser';
6. import { CardLayoutComponent } from './card-layout/card-
   layout.component';
7. @NgModule({
8. imports: [ FlexLayoutModule ],
9. });
```

IMPLEMENTING COMPONENT COMMUNICATIONS

There are few ways in which components can communicate or share data between them. And methods depend on whether the components have a Parent-child relationship between them are not.

Here are the three Possible scenarios

1. Parent to Child Communication
2. Child to Parent Communication
3. Interaction when there is no parent-child relation

Parent to Child Communication

If the Components have a parent-child relationship then, then the parent component can pass the data to the child using the @input Property.

Using @Input Decorator to Pass Data

Create a property (`someProperty`) in the Child Component and decorate it with `@Input()`. This will mark the property as input property

```
1
2export class ChildComponent {
3    @Input() someProperty: number;
```

```
4}
5
```
And in the Parent Component Instantiate the Child Component. Pass the `value` to the `someProperty` using the Property Bind Syntax
```
1
2<child-component [someProperty]=value></child-component>`
3
```
In this way, Child Component will receive the data from the parent

Listen for Input Changes

The Child Component can get the values from the `someProperty`. But it also important for the child component to get notification when the values changes.
There are two ways in which we can achieve that.

1. Using `OnChanges` life Cycle hook or
2. Using a Property Setter on Input Property

Child to Parent Communication

The Child to Parent communication can happen in three ways.

1. Listens to Events from Child
2. Uses Local Variable to access the child in the Template
3. Uses a @ViewChild to get a reference to the child component

Listens to Child Event

This is done by the child component by exposing an EventEmitter Property. We also decorate this Property with @Output decorator. When Child Component needs to communicate with the parent it raises the emit event of the EventEmitter Property. The Parent Component listens to that event and reacts to it.

Uses Local Variable to access the child

Using Local Variable is to refer to the child component is another technique. For Example, Create a reference variable `#child` to the Child Component.
```
1
```

```
2<child-component #child></child-component>
3
```

You can use the `child` (note without #) to access a property of the Child Component. The Code below displays `count` of the Child Component and displays it on screen

```
1
2<p> current count is {{child.count}} </p>
3
```

Uses a @ViewChild to get the reference to the child component

```
1
2<child-component></child-component>
3
```

Another way to get the reference of the child component is using the ViewChild query in the component class

```
1
2@ViewChild(ChildComponent) child: ChildComponent;
3
```

You can call any method in the Child component.

```
1
2  increment() {
3    this.child.increment();
4  }
5
```

Communication when there is no relation

If the Components do not share the Parent-child relationship, then the only way they can share data is by using the services and observable.

The advantageous of using service is that

1. You can share data between multiple components.
2. Using observable, you can notify each component, when the data changes.

Create a Service and create an Angular Observable in that service using either BehaviorSubject or Subject.

```
export class TodoService {

  private _todo = new BehaviorSubject<Todo[]>([]);
  readonly todos$ = this._todo.asObservable();
  ...
}
```

The `_todo` observable will emit data, whenever it is available or changes using the `next` method of the Subject.

```
this._todo.next(Object.assign([], this.todos));
```

In the component class, you can listen to the changes just by subscribing to the observable

```
    this.todoService.todos$.subscribe(val=> {
      this.data=val;
      //do whatever you want to with it.
    })
```

Change Detection and Component Life Cycle

```
@Component({
  selector: 'app-root',
  template: `
  <h2>{{count}}</h2>
  <button (click)='incCount()'>Increment</button>
  `
})
export class AppComponent implements OnInit {


  count: number = 10;
  incCount(): void{
    this.count = this.count +1;
```

```
  }
  ngOnInit() {

  }

}
```

The above component uses interpolation and event binding to display data and call a function on the click event, respectively. Each time the button is clicked, the value of count increases by 1, and the view gets updated to display the updated data. So, here you can see that Angular can detect data changes in the component, and then automatically re-render the view to reflect the change.

The syncing gets complex when the data model gets updated at runtime. Let's take a look at the next code listing:

```
@Component({
  selector: 'app-root',
  template: `
  <h2>{{count}}</h2>
  `
})
export class AppComponent implements OnInit {

  count: number = 10;
  ngOnInit() {
    setInterval(() => {
      this.count = this.count + 1;
    },100)

  }
}
```

The above component simply updates the value of count in every 100 milliseconds. Here, the count is a data model that is getting updated at runtime, but still the Angular change detector displays the updated value of the count in every 100 milliseconds by re-rendering the view.

So, the part of the Angular framework that makes sure the view and the data model are in sync with each other is known as the **change detector**.

The change detector checks the component for the data change and re-renders the view to project the updated data.

When Change Detector Runs

Angular assumes that the data in the component or the whole application state changes due to the following reasons, hence it runs the change detector when either of the following happens:

1. An event, such as click or submit, gets fired
2. An XHR is call to work with an API
3. An asynchronous JavaScript function, such as setTimeOut() or setInterval(), gets executed

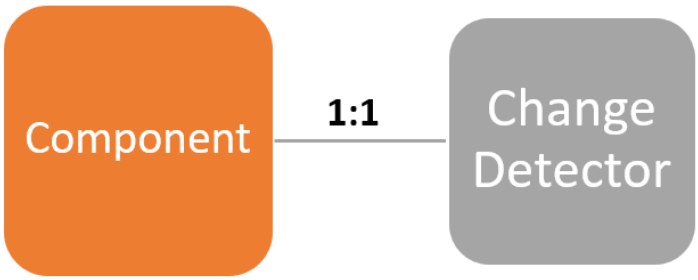
In the last code example, the component uses a setInterval() asynchronous JavaScript method, which updates the values of the count. Since it's an asynchronous method, Angular runs the change detector to update the view with the latest value of the count.

Now the question arises: What notifies Angular of these asynchronous operations?

So, there is something called **ngZone** in Angular whose responsibility is to inform Angular about any asynchronous operations. We won't get into the details of ngZone in this article, but you should know it exists.

Change Detector Tree

Each component in Angular has its own change detector.



The change detector can be referred inside the component using the **ChageDetectorRef** service, and if required you can inject the ChageDetectorRef in a component by making a reference of it in the constructor as shown in next code listing:

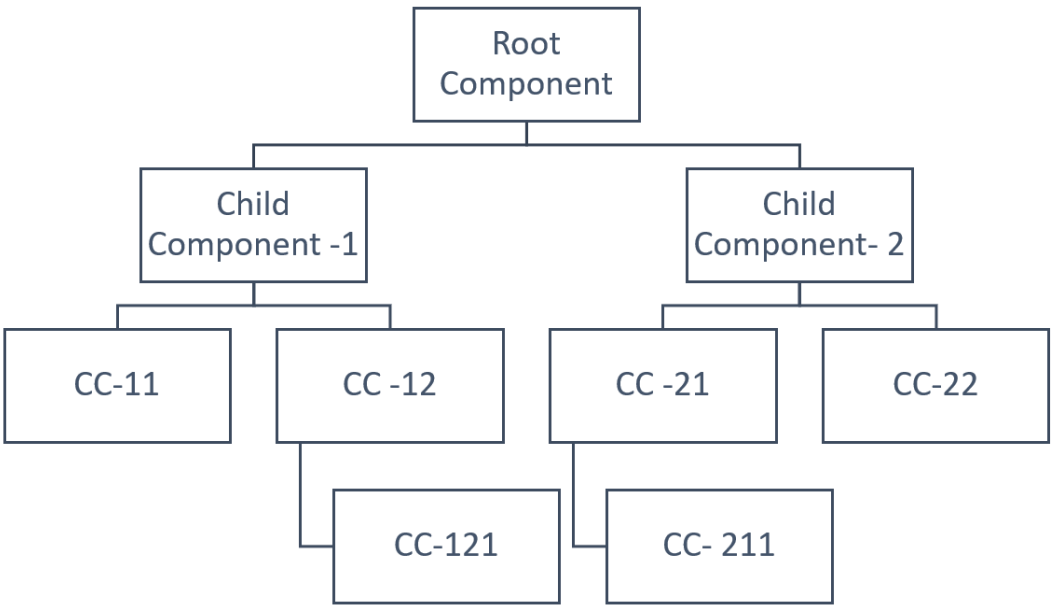
```
export class AppComponent implements OnInit {

  constructor(private cd: ChangeDetectorRef) {
    console.log(this.cd);
  }

  ngOnInit() {
    console.log('init life cycle hook');
  }
}
```

TypeScript

The ChangeDetectorRef provides various APIs to work with the change detector, but before working with them effectively, you need to understand

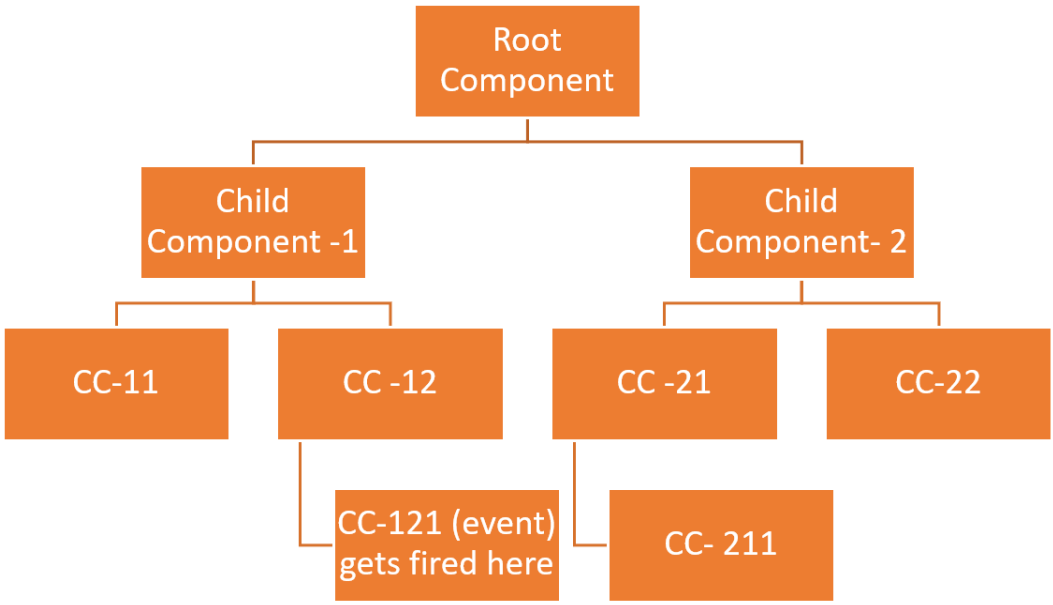
the component tree.

Each component in Angular has its own change detector, and you can see the whole Angular application as a component tree. A component tree is a directed graph, and Angular runs the change detector from top to bottom in the tree.



Logically you can also view the component tree as a change detector tree because each component has its own change detector.

The change detector works from top to bottom in the component tree, and even if an event gets fired in any child node component, Angular always runs the change detector from the root component. For example, in the above change detector tree, if an event gets fired in the component CC-121, which is the bottom node component in the tree, Angular still runs the change detector from the root component node and for all the components.



It may come to your mind that, if for a single event somewhere in the application, Angular runs the change detector for all the components, then perhaps it may have some performance issues. However, that is not true, because of the following reasons:

1. Angular component tree is a directed graph, which means there is a unidirectional flow of the change detector from root to bottom. Angular knows in which direction the tree has to be traversed, and there is no circular or bidirectional traversing of the change detector tree.
2. After a single pass, the change detection tree gets stable.
3. Unlike AngularJS, in Angular, there is no generic function to update the view. Since here every component has its own change detector, JavaScript VM can optimize it for better performance.

So, in Angular, there is no generic function to perform binding, and it generates the change detector class for each component individually at runtime. The definition of the generated change detector class is very particular for a specific component; hence JavaScript VM can optimize it for better performance.

Reducing the Number of Checks

By default, Angular checks each component in the application after any events, asynchronous JavaScript functions, or XHR calls, and, as you have

seen earlier, a single event raised somewhere in the tree could cause each node in the component tree to be checked. But there is a way to reduce the number of checks, and you can avoid running the change detector for the whole subtree.

To optimize the number of checks, Angular provides two change detection strategies:

1. Default strategy
2. onPush strategy

In the **Default strategy**, whenever any data to @Input() decorated properties are changed, Angular runs the change detector to update the view. In the **onPush** strategy, Angular runs change detector only when a **new reference** is passed to the @Input() decorated properties.

Let us understand by having a look at CountComponent:

```
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-count',
  template :`
  <h3>Count in child = {{Counter.count}}</h3>
  `
})
export class CountComponent implements OnInit {

  @Input() Counter;
  constructor() { }

  ngOnInit(): void {
  }
}
```

The CountComponent has one @Input() decorated property Counter, which accepts data from the parent component. Also, the CountComponent is used inside AppComponent, as shown in the next code listing:

```
@Component({
  selector: 'app-root',
  template:`
  <h2>Change Detector Demo</h2>
  <app-count [Counter]='Counter'></app-count>
  <button (click)='incCount()'>Increase Count Value</button>`
})
export class AppComponent implements OnInit {

  Counter = {
    count: 1
  }

  incCount(){

    this.Counter.count = this.Counter.count+ 1;
  }
  ngOnInit() {
    console.log('init life cycle hook');
  }
}
```

AppComponent is using CountComponent as a child and increasing the value of the count on the button click. So, as soon as the click event gets fired, Angular runs the change detector for the whole component tree; hence you get an updated value of the count in the child node CountComponent.

Also, whenever @Input() decorated properties' values change, the Angular change detector runs from the root component and traverses all child components to update the view.

So, for the default change detection strategy, you get the output as expected, but the challenge is, even for one event, Angular runs the change detector for the whole tree. If you wish, you can avoid it for a particular component and its subtree by setting **ChangeDetectionStrategy** to **onPush**.

The CountComponent is modified to use onPush strategy as shown in next code listing:

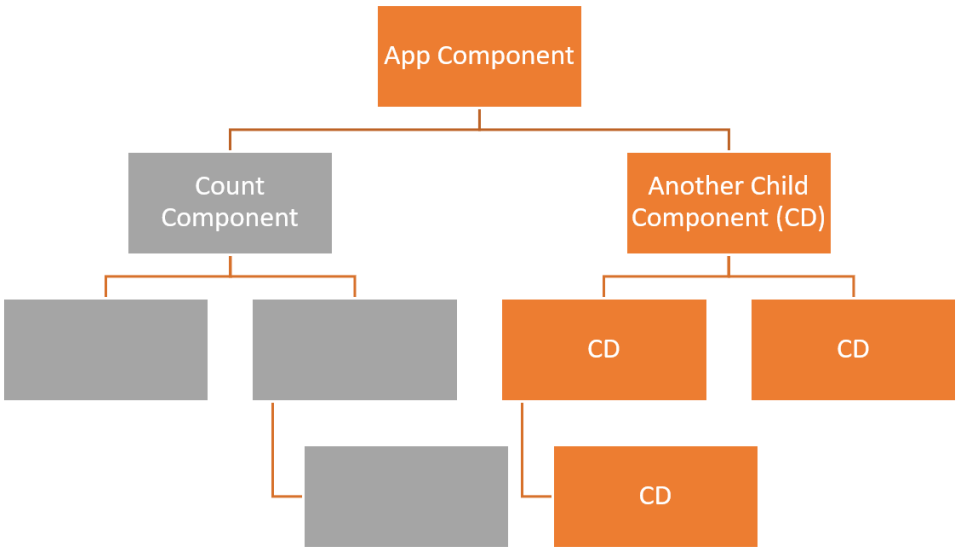
```
@Component({
  selector: 'app-count',
  template :`
  <h3>Count in child = {{Counter.count}}</h3>
  `,
  changeDetection:ChangeDetectionStrategy.OnPush
})
export class CountComponent implements OnInit {

  @Input() Counter;
  constructor() { }

  ngOnInit(): void {
  }
}
```

TypeScript

The onPush change detection strategy instructs Angular to run change detector on the component and its subtree only when a new reference is passed to the @Input decorated properties.

As of now, AppComponent does not pass a new reference of the Counter object—it just changes the property values in it, so Angular would not run the change detector for the CountComponent; hence view would not show the updated value of the count.

You can understand the above scenario with the below diagram:



The above diagram assumes that for "Another Child Component" the change detection strategy is set to Default. Hence, due to the button click in the AppComponent, Angular runs the change detector for each node of Another Child Component subtree.

However, for the CountComponent, change detection strategy is set to onPush, and AppComponent is not passing new reference for the Counter property; hence Angular does not run change detection for Count Component and its subtree.

As Angular is not checking CountComponent, the view is not getting updated. To instruct Angular to check CountComponent and run the change detector, AppComponent has to pass a new reference of count as shown in the next code listing:

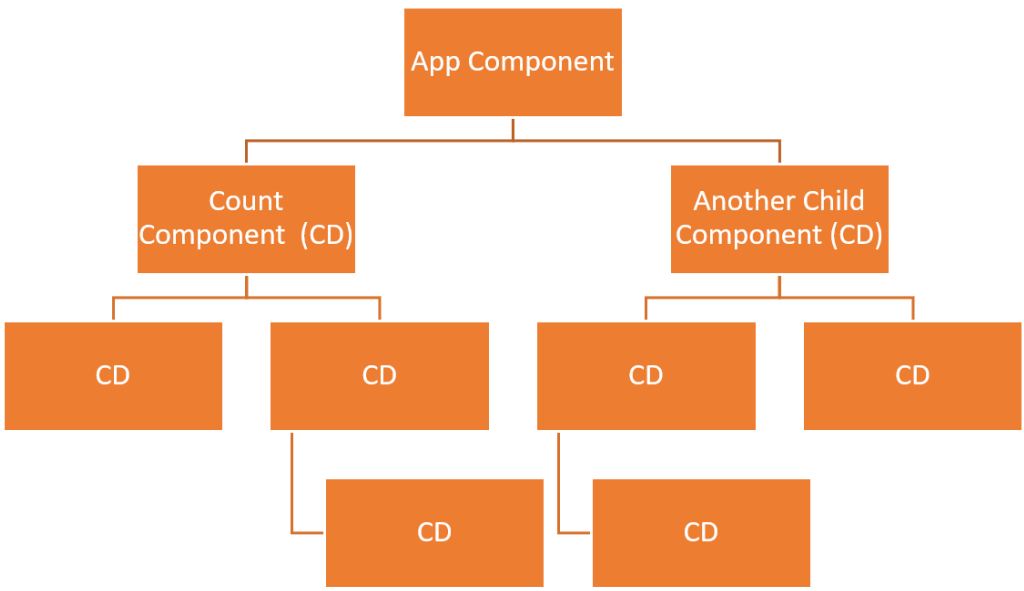
```
incCount(){

  //this.Counter.count = this.Counter.count+ 1;
  this.Counter =  {
    count:  this.Counter.count + 1
  }
 }
```

TypeScript

Now the characteristics of the CountComponent are as follows:

- Its change detection strategy is set to onPush
- Its @Input() decorated property is receiving a new reference of the data

So, Angular runs the change detector for the CountComponent and its subtree, and you get updated data on the view. You can understand the above scenario with the below diagram:



You can opt for either the Default or onPush change detection strategy depending on your requirement. One essential thing you must keep in mind is that even if a component is set to onPush and a new reference is not being passed to it, Angular will still run change detector for it if either of the following happens:

1. An event, such as click or submit, gets fired
2. XHR call to work with an API
3. An asynchronous JavaScript function, such as setTimeOut() or setInterval(), gets executed

9. Assignment Questions

Category - 1

S.No.	Write Programs for the following	K - Level	COs
1	To implement a NGX-charts which is a declarative charting frameworks for Angular	K4	CO5
2	To implement admin panel framework using Angular	K3	CO5

Category - 2

S.No.	Write Programs for the following	K - Level	COs
1	To illustrate a Angular CLI tool for initializing,developing and maintaining angular applications	K5	CO5
2	Write a Reactive Programming for creating a real time applications using Angular	K3	CO5

Category - 3

S.No.	Write Programs for the following	K - Level	COs
1	To describe how to access a browser's local storage with cookies fallback.	K5	CO5
2	For creating a Angular Design patterns that are Used inside Angular Frameworks.	K5	CO5

10. Assignment Questions

Category - 4

S.No.	Write Programs for the following	K - Level	COs
1	To implement Angular HelloWorld project	K3	CO5
2	how the data binding happens in the Angular.	K5	CO5

Category - 5

S.No.	Write Programs for the following	K - Level	COs
1	To explain about the dependency injection with any one applications.	K5	CO5
2	To create a first angular app with your own lists.	K5	CO5

11. Part A
Question & Answer

Part A

1.What is the dependency injection in Angular?

Dependency injection (DI) is the part of the Angular framework that provides components with access to services and other resources. Angular provides the ability for you to inject a service into a component to give that component access to the service.

2.How to use dependency injection in AngularJS?

To inject a value into AngularJS controller function, add a parameter with the same when the value is defined.

```
var myModule = angular.module("myModule", []);
    myModule.value("numberValue", 100);
    myModule.controller("MyController", function($scope, numberValue) {
    console.log(numberValue);
    });
```

3.How does dependency injection work?

Dependency injection is **a pattern to allow your application to inject objects on the fly to classes that need them, without forcing those classes to be responsible for those objects**. It allows your code to be more loosely coupled, and Entity Framework Core plugs in to this same system of services.

4.What is meant by Reactive Programming?

Reactive programming is a programming paradigm dealing with data streams and the propagation of changes. Data streams may be static or dynamic.

5.What is observable and observer in Reactive Programming?

Reactive programming enables the data stream to be emitted from one source called Observable and the emitted data stream to be caught by other sources called Observer through a process called subscription. This Observable / Observer pattern or simple Observer pattern greatly simplifies complex change detection and necessary updating in the context of the programming.

6.What are the three callback function to subscribe to the observable object?

Observer need to implement three callback function to subscribe to the Observable object. They are as follows –

next – Receive and process the value emitted from the Observable

error – Error handling callback

complete – Callback function called when all data from Observable are emitted.

7.what are the important operators that Rxjs library provides fo process the data stream?

Rxjs library provides some of the operators to process the data stream. Some of the important operators are as follows –

filter – Enable to filter the data stream using callback function.

map – Enables to map the data stream using callback function and to change the data stream itself.

pipe – Enable two or more operators to be combined.

8.How to create a sample applications in the Reactive Programming?

Create a new application, reactive using below command –

ng new reactive

Change the directory to our newly created application.

cd reactive

Run the application.

ng serve

Change the AppComponent component code (src/app/app.component.ts)

Change the AppComponent template (src/app/app.component.html)

Shown all the local variable processed by Observer callback functions.

Open the browser, http://localhost:4200.

9.What is meant by Angular Flex Layout?

Angular flex-layout is a stand-alone library developed by the Angular team for designing sophisticated layouts. Angular Layout provides a sophisticated API using Flexbox. The module provides Angular developers with component layout features using a custom layout API.

10.Explain fxLayout and fxLayoutAlign Direxctive.

fxLayout is a directive used to define the layout of HTML elements. i.e., it decides the flow of child elements within the flexbox container and should be applied to the parent DOM element i.e. the flexbox container. This directive is case sensitive and the allowed values of fxLayout are row, column, row-reverse, and column-reverse.

fxLayoutAlign directive defines the alignment of children elements within the flexbox parent container.

Syntax:<div fxLayout="row" fxLayoutAlign="<main-axis> <crossaxis>" >

11.What is meant by fyFlex?

fxFlex is one of the most useful and powerful APIs in Angular Flex Layout. It must be used on children elements inside the fxLayout container. It is responsible for resizing elements along the main-axis of the layout.

12.Give one example for Layout Direction: column.

Alignment in Perpendicular Direction(Vertical): center

```
<div fxLayout="column" fxLayoutAlign="end center">
    <div class="child-1"></div>
<div class="child-2"></div>
```

13.What are the three possible scenarios in implementing component communications?

Parent to Child Communication
Child to Parent Communication
Interaction when there is no parent-child relation.

14.What are the three ways can happen in the child to parent communication?

The Child to Parent communication can happen in three ways.

Listens to Events from Child
Uses Local Variable to access the child in the Template
Uses a @ViewChild to get a reference to the child component.

15.What is change detector? When it runs?

The change detector checks the component for the data change and re-renders the view to project the updated data.

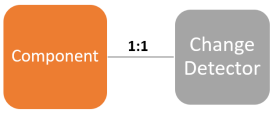
When Change Detector Runs

Angular assumes that the data in the component or the whole application state changes due to the following reasons, hence it runs the change detector when either of the following happens:

An event, such as click or submit, gets fired
An XHR is call to work with an API
An asynchronous JavaScript function, such as setTimeOut() or setInterval(), gets executed.

16. Explain Change Detector tree.

Each component in Angular has its own change detector.



The change detector can be referred inside the component using the ChangeDetectorRef service, and if required you can inject the ChangeDetectorRef in a component by making a reference of it in the constructor.

17.What are the two change detection strategies that the Angular provides?

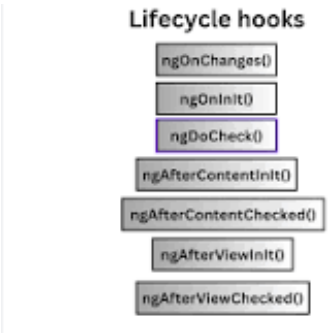
To optimize the number of checks, Angular provides two change detection strategies:

Default strategy

onPush strategy

In the Default strategy, whenever any data to @Input() decorated properties are changed, Angular runs the change detector to update the view. In the onPush strategy, Angular runs change detector only when a new reference is passed to the @Input() decorated properties.

18.What is the life cycle of components in Angular?



In Angular, a component has a lifecycle that goes through a series of stages or hooks during its creation, update, and destruction. These hooks are methods that Angular calls at specific points in the component's lifecycle, and they allow you to perform custom logic or actions at those points.

19.What is the difference between blur and Onchange in Angular?

onchange gets triggered when an element changes, for example, when the value of an input box changes. onblur gets triggered when an object goes out of focus; for example, when you have the cursor in one input box and the cursor goes to another input box, the onblur event of the first input box will get triggered.

20.What are the types of Dependency Injection in angular?

There are three types of Dependency Injections in Angular, they are as follows:
Constructor injection: Here, it provides the dependencies through a class constructor.
Setter injection: The client uses a setter method into which the injector injects the dependency.
Interface injection: The dependency provides an injector method that will inject the dependency into any client passed to it. On the other hand, the clients must implement an interface that exposes a setter method that accepts the dependency.

21.What is the difference between @Component and @Directive in Angular?

A component is *a directive with a template* and the **@Component** decorator is actually a **@Directive** decorator extended with template-oriented feature.

12.PART – B Questions

.

1.Explain in detail about the Dependency Injection in Angular with any one example program.CO5,K3

2.What are the services and its Instantiating in Angular?Explain .CO5,K2

3.What is meant by Reactive Programming and create an sample application using it? .CO5,K3

4.Explain about Laying out pages with flexlayout and its installation in Angular? .CO5,K3

5.What are the ways in implementing Component Communication in Angular? Explain with a program. CO5,K2

6.Explain in detail about the change detector and component life cycle in Angular? .CO5,K3

7.How to reduce the number of checks in Angular? CO5,K2

13. Supportive online courses

Online courses

1. https://www.udemy.com/topic/spring-boot/
2. https://www.guvi.in/mlp/join-full-stack-program
3. https://www.codingninjas.com/careercamp/professionals/
4. https://www.coursera.org/learn/spring-repositories
5. https://www.coursera.org/learn/google-cloud-java-spring

External Links for Additional Resources

1. https://spring.io/guides/gs/spring-boot/
2. https://www.baeldung.com/spring-boot-start
3. https://www.interviewbit.com/spring-boot-interview-questions/

14.Real Time Applications

Online Shopping mart
Social Media Dashboard
Document Editing in various platform
Weather app
Own portfolio blog using angular

Building a template-driven form

How to create a template-driven form. The control elements in the form are bound to data properties that have input validation. The input validation helps maintain data integrity and styling to improve the user experience.

Template-driven forms use two-way data binding to update the data model in the component as changes are made in the template and vice versa.

We can build almost any kind of form with an Angular template —login forms, contact forms, and pretty much any business form. We can lay out the controls creatively and bind them to the data in your object model. You can specify validation rules and display validation errors, conditionally allow input from specific controls, trigger built-in visual feedback, and much more.

15. Assessment Schedule

Tentative schedule for the Assessment During 2023-2024 Even Semester

S. No.	Name of the Assessment	Start Date	End Date	Portion
1	Unit Test 1			Unit 1
2	IAT 1	12.02.2024	17.02.2024	Unit 1 & 2
3	Unit Test 2			Unit 3
4	IAT 2	01.04.2024	06.04.2024	Unit 3 & 4
5	Revision 1			Unit 5, 1 & 2
6	Revision 2			Unit 3 & 4
7	Model	20.04.2024	30.04.2024	All 5 Units

16. Text Books & References

TEXT BOOKS:

1. Somnath Musib, Spring Boot in Practice, Manning publication, June 2022 (https://www.manning.com/books/spring-boot-in-practice)
2. Alex Banks, Eve Porcello , "Learning React", May 2017, O'Reilly Media, Inc. ISBN: 9781491954621. (https://www.oreilly.com/library/view/learning-react/9781491954614/)
3. David Herron ,"Node.js Web Development - Fourth Edition",2018, Packt Publishing,ISBN: 9781788626859
4. Sukesh Marla, "A Journey to Angular Development Paperback ", BPB Publications.(https://in.bpbonline.com/products/a-journey-to-angular-development?_pos=1&_sid=0a0a0e9fb&_ss=r)
5. Yakov Fain Anton Moiseev, "Angular Development with TypeScript", 2nd Edition.(https://www.manning.com/books/angular-development-with-typescript-Second Edition).

Reference Books:

REFERENCES:

1. Sue Spielman, The Struts Framework 1: A Practical guide for Java Programmers‖, 1st Edition. Elsevier 2002

WEB REFERENCES:

1. https://www.manning.com/books/spring-boot-in-practice
2. https://www.oreilly.com/library/view/learning- react/9781491954614
3. https://in.bpbonline.com/products/a-journey-to-angular-development?_pos=1&_sid=0a0a0e9fb&_ss=r
4. https://in.bpbonline.com/products/a-journey-to-angular-development?_pos=1&_sid=0a0a0e9fb&_ss=r
5. https://www.manning.com/books/angular-development-with-typescript-Second Edition

17. Mini Project Suggestions

Category 1:

1.Discuss with the Real life Example of an Online Shopping Cart application to illustrate Dependency Injection in Angular. CO5,K6

2. To create a Google Maps integration for our current apps using Angular . CO5,K6

Category 2:

1.Explain in detail to create a weather app using dynamic UI with RXjs. CO5,K5

2.Building our own portfolio blog using angular and some basic HTML and CSS styles. CO5,K5

Category 3:

1.To create a social network platform which allows users to create profiles, connect with friends and share content using angular. CO5, K4

2.To implement the database to store user data, and creating social features such as commenting and liking using angular. CO5, K4.

Category 4:

1. To create a fitness tracker app that allows users to track their exercise routines and progress. It involves creating a user interface that allows users to log their workouts, view their progress, and set fitness goals. CO5, K3

2. To implement a movie database app that allows users to search for and view information about movies. CO5, K3

Category:5

1. A to-do list app is a great starting project for beginner Angular Developers. The app allows users to add, delete and mark tasks as complete. CO5, K 2

2. To create a first app on angular JS with the basic concepts. CO5, K2

Thank you