Practical No 9

Aim: Design a page to implement Add to Cart functionality using decorators, custom properties, custom events of component communication.

A. Objective:

- Modify the behaviour of a class, property, method, or parameter
- Pass data from a parent component to a child component
- Modify the behaviour of a child component based on data passed from the parent component and Pass data from the child component to the parent component.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

C. Expected Skills to be developed based on competency:

Learner candevelop a deeper understanding of how decorators work, how to encapsulate and share data across different parts of your application,help you to create more powerful and dynamic applications.

D. Expected Course Outcomes(Cos)

Utilize angular template driven and reactive forms in different problem solutions.

E. Practical Outcome(PRo)

Learner will be able to design module using decorators, custom properties, custom events of component communication to make real time user interactive application.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

In Angular, a decorator is a function that is used to modify a class, method, property, or parameter. Decorators provide a way to add metadata to your code, which can be used by the Angular framework to configure your application.

Angular provides several built-in decorators that you can use to modify the behavior of your components, services, directives, and other classes.

By using decorators, you can easily add features such as dependency injection, event handling, and custom directives to your components and services.

@Input decorator

In Angular, the @Input decorator is used to pass data from a parent component to its child component. Here's an example of how to use @Input decorator and custom properties:

Create a child component called child-component using the Angular CLI command: ng generate component child-component.

In the child component's TypeScript file (child-component.ts), import the Input decorator from @angular/core.

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

@Component({
selector: 'app-child-component',
templateUrl: './child-component.html',
styleUrls: ['./child-component.css']
})
export class ChildComponent {
@Input() customProperty: string;
}
```

In this example, we create a custom property called customProperty using the @Input decorator. This property can be set by the parent component.

In the child component's HTML file (child-component.html), display the value of the custom property using interpolation syntax.

```
<p>Custom property value: {{ customProperty }}</p>
```

In the parent component's TypeScript file (parent-component.ts), set the value of the custom property.

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

@Component({
selector: 'app-parent-component',
templateUrl: './parent-component.html',
styleUrls: ['./parent-component.css']
})
export class ParentComponent {
customPropertyValue = 'Hello, world!';
}
```

In this example, we define the customPropertyValue variable and set it to a string.

In the parent component's HTML file (parent-component.html), add the child component and bind to the custom property using the [customProperty] syntax.

```
<app-child-component [customProperty]="customPropertyValue"></app-child-component>
```

In this example, we bind to the customProperty using the [customProperty] syntax and pass the customPropertyValue variable as the value.

That's how you can use @Input decorator and custom properties in Angular to pass data from a parent component to its child component.

@Output decorator

In Angular, the @Output decorator is used to create custom events that can be emitted from a child component to its parent component. Here's an example of how to use @Output decorator and custom events:

Create a child component called child-component using the Angular CLI command: ng generate component child-component.

In the child component's TypeScript file (child-component.ts), import the Output decorator from @angular/core and create an instance of the EventEmitter class.

```
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
selector: 'app-child-component',
templateUrl: './child-component.html',
styleUrls: ['./child-component.css']
})
export class ChildComponent {
@Output() customEvent = new EventEmitter<string>();

handleClick() {
this.customEvent.emit('Custom event was triggered!');
  }
}
```

In this example, we create a custom event called customEvent using the @Output decorator. We also create an instance of the EventEmitter class, which is used to emit the custom event when a button is clicked. The handleClick() method is called when the button is clicked, which triggers the custom event using this.customEvent.emit().

In the child component's HTML file (child-component.html), add a button that calls the handleClick() method when clicked.

```
<button (click)="handleClick()">Click me!</button>
```

In the parent component's TypeScript file (parent-component.ts), import the ViewChild decorator from @angular/core and the child component.

```
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child-component/child-component.component';

@Component({
selector: 'app-parent-component',
templateUrl: './parent-component.html',
styleUrls: ['./parent-component.css']
})
```

```
export class ParentComponent {
@ViewChild(ChildComponent) childComponent: ChildComponent;

handleCustomEvent(event: string) {
console.log(event);
  }
}
```

In this example, we use the ViewChild decorator to get a reference to the child component. We also define the handleCustomEvent() method to handle the custom event.

In the parent component's HTML file (parent-component.html), add the child component and bind to the custom event using the (customEvent) syntax.

```
<app-child-component (customEvent)="handleCustomEvent($event)"></app-child-component>
```

In this example, we bind to the customEvent using the (customEvent) syntax and call the handleCustomEvent() method when the custom event is triggered.

That's how you can use @Output decorator and custom events in Angular to emit events from a child component to its parent component.

@input and @output decorator

In Angular, you can use both @Input and @Output decorators to create a two-way data binding between a parent component and its child component. Here's an example of how to use @Input and @Output decorators together:

Create a child component called child-component using the Angular CLI command: ng generate component child-component.

In the child component's TypeScript file (child-component.ts), import the Input and Output decorators from @angular/core.

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

@Component({
selector: 'app-child-component',
templateUrl: './child-component.html',
styleUrls: ['./child-component.css']
```

```
})
export class ChildComponent {
@Input() customProperty: string;
@Output() customEvent = new EventEmitter<string>();

handleClick() {
this.customEvent.emit('Custom event was triggered!');
  }
}
```

In this example, we create a custom property called customProperty using the @Input decorator. We also create a custom event called customEvent using the @Output decorator and an instance of the EventEmitter class. The handleClick() method is called when a button is clicked, which triggers the custom event using this.customEvent.emit().

In the child component's HTML file (child-component.html), display the value of the custom property using interpolation syntax and add a button that calls the handleClick() method when clicked.

```
<p>Custom property value: {{ customProperty }}</p>
<button (click)="handleClick()">Click me!</button>
```

In the parent component's TypeScript file (parent-component.ts), define the custom property value and the method that handles the custom event.

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

@Component({
selector: 'app-parent-component',
templateUrl: './parent-component.html',
styleUrls: ['./parent-component.css']
})
export class ParentComponent {
customPropertyValue = 'Hello, world!';

handleCustomEvent(event: string) {
console.log(event);
  }
}
```

In this example, we define the customPropertyValue variable and set it to a string. We also define the handleCustomEvent() method to handle the custom event.

In the parent component's HTML file (parent-component.html), add the child component and bind to the custom property using the [customProperty] syntax. Also, bind to the custom event using the (customEvent) syntax.

```
<app-child-component [customProperty]="customPropertyValue" (customEvent)="handleCustomEvent($event)"></app-child-component>
```

In this example, we bind to the customProperty using the [customProperty] syntax and pass the customPropertyValue variable as the value. We also bind to the customEvent using the (customEvent) syntax and call the handleCustomEvent() method when the custom event is triggered.

Practical No 10

Aim: Design an e-commerce product page and product details page that displays product details when clicking on any particular product.

A. Objective:

Services are an important part of Angular development, providing a way to encapsulate functionality, manage dependencies, and promote code reusability. the Router, ActivatedRoute, and param parameters allowing for the implementation of dynamic routing, state management, access to route parameters and enabling URL management.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering*tools and appropriate technique to conduct standard tests and measurements.

C. Expected Skills to be developed based on competency:

Learner can manage Service, router and Active Router facilities and to pass data from one component to another via the URL that can handle different user inputs and scenarios.

D. Expected Course Outcomes(Cos)

Utilize angular template driven and reactive forms in different problem solutions.

E. Practical Outcome(PRo)

Learner will be able to encapsulate and modularize functionality that can be used across components and modules, making it easier to reuse code and avoid

duplication, enables navigation from one component to another, without reloading the page, managing URLs and handling redirects to make real time user interactive application.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Service Concept:

Service is a class that is used to encapsulate functionality that can be shared across components. Services are typically used to handle business logic, API calls, and other non-UI related tasks. Here's an example of a service with two methods and how it can be used in a component:

Service: UserService

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

@Injectable({
  providedIn: 'root'
})
export class UserService {
  private users = [
    { id: 1, name: 'abc'', email: 'abc@example.com' },
    { id: 2, name: 'pqr'', email: 'pqr@example.com' },
    { id: 3, name: 'xyz', email: 'xyz@example.com' },
  ];

  getUsers() {
    return this.users;
  }

  getUserById(id: number) {
    return this.users.find(user => user.id === id);
  }
}
```

This UserService has two methods, getUsers() and getUserById(), which respectively return all the users and a specific user by ID.

Component: UserComponent

```
import { Component, OnInit } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-user',
  templateUrl: './user.component.html',
  styleUrls: ['./user.component.css']
})
export class UserComponent implements OnInit {
  users = [];

  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.users = this.userService.getUsers();
  }

  getUserById(id: number) {
    return this.userService.getUserById(id);
  }
}
```

In this UserComponent, the UserService is injected into the constructor using Dependency Injection. In the ngOnInit() method, the getUsers() method of the UserService is called to get all the users, which are then assigned to the users property of the component. The getUserById() method is also defined in the component, which calls the getUserById() method of the UserService to get a specific user by ID.

To summarize, the UserService is a reusable class that encapsulates functionality related to user management, and it can be used in multiple components by injecting it using Dependency Injection. In this example, the UserComponent uses the getUsers() and getUserById() methods of the UserService to get all users and a specific user by ID.

In Angular, the Router is a built-in service that allows for navigation between different views of the application based on the URL. The ActivatedRoute provides information about the currently activated route, while the router.navigate() method is used to navigate to a different route programmatically. Here's an example of how to use these concepts in Angular:

Routing Configuration: app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

In this routing configuration, we define three routes: the home route (''), the about route ('about'), and the contact route ('contact'). Each route corresponds to a component: HomeComponent, AboutComponent, and ContactComponent, respectively.

Component: home.component.ts

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

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  constructor(private router: Router) { }

  ngOnInit(): void {
  }
```

```
  goToAboutPage() {
    this.router.navigate(['/about']);
  }

  goToContactPage() {
    this.router.navigate(['/contact']);
  }

}
```

In this HomeComponent, we inject the Router service into the constructor using Dependency Injection. We define two methods, goToAboutPage() and goToContactPage(), which use the router.navigate() method to navigate to the about and contact pages, respectively.

Component: home.component.html

```
<div>
  <h1>Welcome to the Home Page!</h1>
  <button (click)="goToAboutPage()">Go to About Page</button>
  <button (click)="goToContactPage()">Go to Contact Page</button>
</div>
```

In this HomeComponent HTML template, we use two buttons that are bound to the goToAboutPage() and goToContactPage() methods defined in the TypeScript file. When a user clicks one of these buttons, the corresponding method is called, which uses the router.navigate() method to navigate to the specified route.

Here ContentComponent contain the same thing as HomeComponent except the description of it.

Overall, the Router, ActivatedRoute, and router.navigate() method are essential components of Angular development, allowing for dynamic routing, state management, and navigation guards. In this example, we demonstrated how to use these concepts to navigate between different views of the application based on the URL.

Practical No 11

Aim: Develop page to demonstrate different methods of angular component lifecycle.

A. Objective: Main objective of angular component lifecycle are component initialization to initialize component before it renders to page, Change Detection mechanism to detect changes in the component's properties and update the view accordingly, Content Projection used to project content into their views using content projection, View Initialization and Update and Destruction of component.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

C. Expected Skills to be developed based on competency:

D. Expected Course Outcomes (Cos)

Utilize angular template driven and reactive forms in different problem solutions.

E. Practical Outcome (PRo)

Learner will be able to utilize different methods of the angular component lifecycle.

F. Expected Affective domain Outcome (ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Angular components have a series of lifecycle events that are executed in a specific order during their creation, rendering, and destruction. These lifecycle events allow developers to control and respond to changes in the state of the component.

The following is the order of the Angular component lifecycle events:

ngOnChanges: This event is executed when the input properties of a component change. It receives a SimpleChanges object that contains the current and previous values of the input properties.

ngOnInit: This event is executed once, immediately after the first ngOnChanges event. It is used to initialize the component's properties.

ngDoCheck: This event is executed every time Angular performs change detection. It allows developers to detect and respond to changes that Angular may have missed.

ngAfterContentInit: This event is executed after the component's content has been projected into its view. It is used to perform initialization tasks that depend on the component's content.

ngAfterContentChecked: This event is executed every time Angular checks the content of the component. It allows developers to perform additional checks or updates after the content has been checked.

ngAfterViewInit: This event is executed after the component's view has been initialized. It is used to perform initialization tasks that depend on the component's view.

ngAfterViewChecked: This event is executed every time Angular checks the view of the component. It allows developers to perform additional checks or updates after the view has been checked.

ngOnDestroy: This event is executed just before the component is destroyed. It is used to perform cleanup tasks such as unsubscribing from observables or removing event listeners.

By using these lifecycle events, developers can build more robust and responsive Angular applications.

Below example for implementing the Angular component lifecycle:

Create a new Angular project using the Angular CLI.

Create a new component called "LifecycleComponent" using the CLI command: ng generate component Lifecycle
In the "LifecycleComponent" class, implement the following lifecycle hooks:
ngOnInit
ngOnChanges
ngDoCheck
ngAfterContentInit
ngAfterContentChecked
ngAfterViewInit
ngAfterViewChecked
ngOnDestroy
In each lifecycle hook method, add a console.log statement to output a message indicating which hook has been called.
Add the "LifecycleComponent" to the "AppComponent" template.
Test the component by running the application and observing the console output.
Here's an example implementation of the "LifecycleComponent" class:

```
import { Component, OnInit, OnChanges, DoCheck, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy } from '@angular/core';

@Component({
 selector: 'app-lifecycle',
 templateUrl: './lifecycle.component.html',
 styleUrls: ['./lifecycle.component.css']
})
export class LifecycleComponent implements OnInit, OnChanges, DoCheck, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy {

 constructor() { }

 ngOnInit(): void {
  console.log('ngOnInit');
 }

 ngOnChanges(): void {
  console.log('ngOnChanges');
 }

 ngDoCheck(): void {
```

```
    console.log('ngDoCheck');
  }

  ngAfterContentInit(): void {
    console.log('ngAfterContentInit');
  }

  ngAfterContentChecked(): void {
    console.log('ngAfterContentChecked');
  }

  ngAfterViewInit(): void {
    console.log('ngAfterViewInit');
  }

  ngAfterViewChecked(): void {
    console.log('ngAfterViewChecked');
  }

  ngOnDestroy(): void {
    console.log('ngOnDestroy');
  }

}
```

And here's an example implementation of the "AppComponent" template:

```
<div>
  <h1>Angular Component Lifecycle Exercise</h1>
  <app-lifecycle></app-lifecycle>
</div>
```

Practical No.12

Aim: Design a page to display student information using dependency Injection.

A. **Objective:** GET and POST web APIs is important task to develop any dynamic single page application so here students details will be displayed on page using web APIs by students.

B. **Expected Program Outcomes (POs)**

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.
2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.
3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.
5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes in field of engineering.

C. **Expected Skills to be developed based on competency:**

1. Develop service to use dependency injection.
2. Design a page to display student details using angular component and service.

D. **Expected Course Outcomes(Cos)**

Design pages to make HTTP GET/POST calls to perform CRUD operations using different server-side APIs.

E. **Practical Outcome(PRo)**

Design a page to display student information using dependency Injection.

F. **Expected Affective domain Outcome (ADos)**

1. Maintain tools and equipment.
2. Follow Coding standards and practices.

3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory: Dependency Injection (DI) in Angular is a design pattern that allows us to create loosely coupled components by providing dependencies to a component at the time of creation. In other words, DI is a way of providing objects that a class needs (its dependencies) from an external source.

In Angular, we use DI to inject the required services or dependencies into a component, directive, or any other class that needs them. This is done by adding the dependencies as parameters to the class constructor. When the component is instantiated, Angular creates and injects an instance of the required dependencies into the component.

Using DI in Angular has several benefits, including:

Loosely coupled components: DI helps to create components that are independent of each other and can be easily reused.

Testability: Since the dependencies are passed in as parameters, it's easy to provide mock or stub implementations of the dependencies for unit testing.

Code maintainability: DI makes it easy to manage the dependencies of a component and update them if needed.

Scalability: DI allows for easy scaling of an application by making it possible to easily add or remove dependencies as needed.

To use DI in Angular, we first need to define the dependencies as services and then inject them into the components that need them. This is done using the @Injectable() decorator for services and adding the required services as parameters to the constructor of the component that needs them. Below is example of student info display with dependency injection.

StudentService.ts

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


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


  public students = [
```

```
    { id: 1, name: 'Alpesh Thaker', major: 'Computer Engineering },
    { id: 2, name: 'Umang Shah', major: 'Biology' },
    { id: 3, name: 'Yagnik Tank', major: 'Mathematics' }
  ];

getStudents() {
    return this.students;
  }

getStudentById(id: number) {
    return this.students.find(student => student.id === id);
  }
}
```

student-information.component.ts

```
import { Component, OnInit } from '@angular/core';
import { StudentService } from '../student.service';

@Component({
  selector: 'app-student-information',
templateUrl: './student-information.component.html',
styleUrls: ['./student-information.component.css']
})
export class StudentInformationComponent implements OnInit {

  public students:any;

  constructor(private studentService: StudentService) { }
```

```
ngOnInit() {
this.students = this.studentService.getStudents();
 }}
```

student-information.component.html

```
<p>student-information works!</p>
<div *ngFor="let student of students">
<h3>{{ student.name }}</h3>
<p>Major: {{ student.major }}</p>
</div>
```

Practical No.13

Aim: Develop a page for product listing and search-as-you type and web APIs from database.

A. Objective: The objective of implementing a search-as-you-type feature using observables and web APIs from a database in Angular is to enhance the user experience by providing real-time search results as the user types into the search bar. The use of observables in Angular allows for asynchronous data streams to be handled in a reactive manner, ensuring that the search results are always up-to-date and responsive to user input.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.
2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.
3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.
5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes in field of engineering.

C. Expected Skills to be developed based on competency:

1. Knowledge of components, modules, services, and observables.
2. Developing search-as-you-type functionality requires a strong understanding of RxJS operators and how to use them effectively.

D. Expected Course Outcomes (Cos)

Design pages to make HTTP GET/POST calls to perform CRUD operations using different server-side APIs.

E. Practical Outcome (PRo)

Develop a page for product listing and search-as-youtype using observables and web APIs from database.

F. Expected Affective domain Outcome (ADos)

1. Maintain tools and equipment.
2. Follow Coding standards and practices.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Observables: In Angular, observables are a powerful feature that allow developers to work with asynchronous data streams. An observable is a representation of a stream of events that can be subscribed to by one or more observers. It provides a way to handle asynchronous operations and to propagate changes across an application.

Observables are used extensively in Angular for handling events such as HTTP requests, user interactions, and data updates. They can be used to manage the flow of data between components, services, and other parts of an application.

Some key features of observables in Angular include:

1. Asynchronous data handling: Observables allow developers to work with asynchronous data streams, such as HTTP requests, in a more efficient and reliable manner.
2. Easy to compose: Observables can be composed and combined to create more complex data streams, which makes it easier to handle complex scenarios and data flows.
3. Cancellation support: Observables can be cancelled, which helps to prevent memory leaks and reduce unnecessary processing.
4. Error handling: Observables have built-in error handling support, which makes it easier to handle errors and exceptions that may occur during asynchronous operations.
5. Support for multiple values: Observables can emit multiple values over time, which makes them well-suited for handling real-time data streams.

In summary, observables in Angular are a powerful feature that enable developers to handle asynchronous data streams in a more efficient and reliable manner. They provide a flexible and composable way to manage the flow of data in an application, and are used extensively in Angular for handling events and data updates.

RXJS Operators: DebounceTime(), distinctUntilChanged(), and switchMap() are three RxJS operators commonly used together in Angular for implementing search-as-you-type functionality using Observables.

1. debounceTime(): This operator delays the emission of items from an Observable for a specified period of time after the last emission. It is commonly used to

prevent unnecessary network requests when a user is typing quickly in a search input field. For example, if you specify a debounce time of 300 milliseconds, the operator will wait for 300 milliseconds after the last keystroke before emitting the value.

2. distinctUntilChanged(): This operator filters out duplicate values emitted by an Observable. It compares the current value emitted with the previous value emitted, and if they are the same, it suppresses the emission of the current value. This is useful in search-as-you-type scenarios to avoid making unnecessary network requests when the user types the same value repeatedly.

3. switchMap(): This operator maps each value emitted by an Observable to another Observable, and then flattens the resulting Observable into a single stream of values. This is useful in search-as-you-type scenarios because it allows you to cancel previous network requests and only return the results for the most recent search term entered by the user.

Now search-as-you-type functionality can be implemented using observables. The basic idea is to listen for changes to the search input field and use an observable to update the search results in real-time.

Here are the steps to implement search-as-you-type functionality using observables in Angular:

1. Set up the search input field: Create a text input field in your template and bind it to a property in your component using [(ngModel)].

2. Create an Observable: Use the RxJS library to create an Observable that listens to changes in the search input field.

3. Handle the Observable: In the Observable's subscribe() method, use the value of the search input field to make an HTTP request to your server-side API and retrieve the search results.

4. Update the search results: Use Angular's change detection mechanism to update the search results in the template as soon as new results are retrieved from the server.

Here's some sample code to help illustrate how this works In your component:

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

import { Observable, Subject } from 'rxjs';

import { debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operators';

import { ApiService } from './api.service';
```

```
@Component({
  selector: 'app-search',
  template: `
    <input type="text" [(ngModel)]="searchTerm" placeholder="Search">
    <ul>
      <li *ngFor="let result of searchResults">{{result}}</li>
    </ul>
  `,
})
export class SearchComponent {
  searchTerm: string;
  searchResults: string[];
  private searchTerms = new Subject<string>();

  constructor(private apiService: ApiService) {}

  ngOnInit(): void {
    this.searchTerms.pipe(
      debounceTime(300),
      distinctUntilChanged(),
      switchMap((term: string) => this.apiService.search(term))
    ).subscribe((results: string[]) => {
      this.searchResults = results;
    });
  }

  search(term: string): void {
    this.searchTerms.next(term);
```

```
  }
}
```

In your API service:

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

@Injectable({
  providedIn: 'root',
})
export class ApiService {
  constructor(private http: HttpClient) {}

  search(term: string): Observable<string[]> {
    return this.http.get<string[]>(`/api/search?term=${term}`);
  }
}
```

In the example above, the searchTerms property is a Subject that emits new search terms whenever the search input field changes. These search terms are then piped through a series of operators, including debounceTime(), distinctUntilChanged(), and switchMap(). The switchMap() operator makes an HTTP request to the server-side API to retrieve the search results, and then the results are updated in the searchResults property of the component.

Practical No.14

Aim: Design web page to display student data in table using HTTP GET/POST Calls from web APIs.

A. Objective: GET and POST web APIs is important task to develop any dynamic single page application so here students details will be displayed on page using web APIs by students.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the engineering problems.
2. **Problem analysis**: Identify and analyse well-defined engineering problems using codified standard methods.
3. **Design/ development of solutions:** Design solutions for engineering well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern engineering tools and appropriate technique to conduct standard tests and measurements.
5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes in field of engineering.

C. Expected Skills to be developed based on competency:

1. Develop GET and POST web APIs in PHP and MYSQL.
2. Design a page to display student details using angular component and service.

D. Expected Course Outcomes(Cos)

Design pages to make HTTP GET/POST calls to perform CRUD operations using different server-side APIs.

E. Practical Outcome(PRo)

Design web page to display student data in table using HTTP GET/POST Calls from web APIs.

F. Expected Affective domain Outcome(ADos)

1. Maintain tools and equipment.
2. Follow Coding standards and practices.
3. Follow safety practices.

4. Follow ethical practices

G. Prerequisite Theory: Web APIs are also called RESTFUL APIs. REST (Representational State Transfer) is a popular architectural style for creating web services, and it uses HTTP methods to perform various CRUD (Create, Read, Update, Delete) operations on resources. In this answer, I'll provide a basic theory for creating POST and GET RESTful APIs using PHP and MySQL.

To start with, we need to create a MySQL database and a table to store the data that we want to expose through our APIs. For example, let's assume that we want to create an API to manage a list of users, so we can create a MySQL database with a table named "users" having columns like "id", "name", "email", "phone", etc.

Now let's see how we can create a POST API to add a new user to our database using PHP.

POST API:

The HTTP POST method is used to create a new resource. To create a new user, we need to send a POST request to our API endpoint with the user data in the request body. Here's an example of how we can implement this in **students.php**:

```
<?php
$dbhost = "localhost";
$dbname = "angular";
$username = "root";
$password = "";

$conn = mysqli_connect($dbhost, $username, $password, $dbname);

// Check connection
if (!$conn) {
  die("Connection failed: " . mysqli_connect_error());
}
// Get all students
function getstudents($conn) {
  $sql = "SELECT * FROM students";
  $result = mysqli_query($conn, $sql);
```

```
  $students = array();

  if (mysqli_num_rows($result) > 0) {
    while($row = mysqli_fetch_assoc($result)) {
      $students[] = $row;
    }
  }

  return $students;
}

// Get a single student by ID
function getStudent($conn, $id) {
  $sql = "SELECT * FROM students WHERE id = $id";
  $result = mysqli_query($conn, $sql);

  $students = mysqli_fetch_assoc($result);

  return $students;
}
$method = $_SERVER['REQUEST_METHOD'];
echo $_SERVER['PATH_INFO'];
$request = explode('/', trim($_SERVER['PATH_INFO'],'/'));

switch ($method) {
  case 'GET':
    if ($request[0] == 'Students') {
```

```
        if (isset($request[1])) {
          $response = getStudent($conn, $request[1]);
        } else {
          $response = getstudents($conn);
        }
      } else {
        // Handle other endpoints
      }
      break;
    case 'POST':
      // Handle POST requests
      break;
    case 'PUT':
      // Handle PUT requests
      break;
    case 'DELETE':
      // Handle DELETE requests
      break;
  }

  echo json_encode($response);
```

In the above example, GET method demonstrated to generate response of students in json format and below details describe angular example.

Sample Angular code for designing a web page to display student data in a table using HTTP GET/POST calls from web APIs:

1. First, create a new Angular component using the Angular CLI:

   Css code

   ng generate component student-table

2. In the **student-table.component.html** file, create a table to display the student data:

```
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Email</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr *ngFor="let student of students">
<td>{{ student.id }}</td>
<td>{{ student.name }}</td>
<td>{{ student.email }}</td>
<td>{{ student.phone }}</td>
</tr>
</tbody>
</table>
```

3. In the **student-table.component.ts** file, import the **HttpClient** module from **@angular/common/http**, inject it into the constructor, and create a method to fetch the student data using a GET call to a web API:

Typescript code

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-student-table',
```

```
templateUrl: './student-table.component.html',
styleUrls: ['./student-table.component.css']
})
export class StudentTableComponent implements OnInit {
 students: any[];

 constructor(private http: HttpClient) {}

ngOnInit() {

this.http.get<any[]>('http://localhost/api/students.php/students').subscribe
(
    data => {
this.students = data;
    },
    error => {
      console.log(error);
    }
  );
 }
}
```

In the **app.module.ts** file, import the **HttpClientModule** module from **@angular/common/http** and add it to the **imports** array:

typescript code

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
 selector: 'app-student-table',
```

```
templateUrl: './student-table.component.html',
styleUrls: ['./student-table.component.css']
})
export class StudentTableComponent implements OnInit {
 students: any[];

 constructor(private http: HttpClient) {}

ngOnInit() {

this.http.get<any[]>('http://localhost/api/students.php/student').subscribe(
    data => {
this.students = data;
    },
    error => {
      console.log(error);
    }
  );
 }
}
```

In the **app.component.html** file, add the **app-student-table** selector to display the **StudentTableComponent**:

htmlCopy code

```
<app-student-table></app-student-table>
```

6. Finally, start the Angular development server using **ng serve** and navigate to **http://localhost:4200** to view the student data in the table.

   Note: You can also use a POST call to a web API to add new student data to the table. To do this, create a form in the **student-table.component.html** file to capture the student data and use the **HttpClient** module to make a POST call to the web API when the form is submitted.

Practical No.15

Aim: Design a page to implement Multiview component with login, logout functionalities using different routing options.

A. Objective: Routing is important option of angular and students will implement it with Multiview using Login, Home Screen and Logout screens.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.
2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.
3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.
5. **Engineering practices for society, sustainability and environment:** Apply appropriate technology in context of society, sustainability, environment and ethical practices.
6. **Project Management:** Use engineering management principles individually, as a team member or a leader to manage projects and effectively communicate about well-defined engineering activities.
7. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes in field of engineering.

C. Expected Skills to be developed based on competency:

1. Design a page to implement Multiview components for login, home screen and logout functionality using routing option.

D. Expected Course Outcomes(Cos)

Develop single page dynamic applications using Angular framework and APIs.

E. Practical Outcome(PRo)

Design a page to implement Multiview component with login, logout functionalities using different routing options.

F. Expected Affective domain Outcome(ADos)

1. Maintain tools and equipment.
2. Follow Coding standards and practices.

3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

To implement the Multiview component with login and logout functionalities using different routing options in Angular, we need to create the following components:

- HomeComponent
- LoginComponent
- DashboardComponent

The HomeComponent will contain a basic welcome message and a login button, which will redirect the user to the LoginComponent. The LoginComponent will have a form to enter the username and password, and once the user clicks on the login button, the user will be redirected to the DashboardComponent.

The DashboardComponent will contain the main content of the application and will be accessible only after the user logs in. Once the user logs out, the user will be redirected back to the HomeComponent.

Here are the steps to implement this in Angular:

Step 1: Create a new Angular project

ng new multiview-component-demo

Step 2: Create the necessary components using the Angular CLI

ng generate component home

ng generate component login

ng generate component dashboard

Step 3: Add the necessary routes to the app-routing.module.ts file

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { LoginComponent } from './login/login.component';
import { DashboardComponent } from './dashboard/dashboard.component';

const routes: Routes = [
```

```
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'dashboard', component: DashboardComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Step 4: Update the HomeComponent template to include a login button.

```
<h1>Welcome to the Multiview Component Demo</h1>
<button routerLink="/login">Login</button>
```

Step 5: Update the LoginComponent template to include a login form

```
<h1>Login</h1>
<form>
  <label>
    Username:
    <input type="text">
  </label>
  <br>
  <label>
    Password:
    <input type="password">
  </label>
  <br>
  <button routerLink="/dashboard">Login</button>
</form>
```

Step 6: Update the DashboardComponent template to include a logout button

<h1>Welcome to the Dashboard</h1>

<button routerLink="/">Logout</button>

Step 7: Update the AppComponent template to include the router outlet

<router-outlet></router-outlet>

Step 8: Run the application using the following command

ng serve

Now you should be able to see the HomeComponent with a login button. Clicking on the login button should redirect you to the LoginComponent with a login form. Once you enter the username and password and click on the login button, you should be redirected to the DashboardComponent. Clicking on the logout button in the DashboardComponent should redirect you back to the HomeComponent.

Practical No.16

Aim: Develop a page to demonstrate page navigation of product list using routing concepts.

A. Objective: Routing is important option of angular and students will implement it for page navigation of products listing.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.
2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.
3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.
5. **Engineering practices for society, sustainability and environment:** Apply appropriate technology in context of society, sustainability, environment and ethical practices.
6. **Project Management:** Use engineering management principles individually, as a team member or a leader to manage projects and effectively communicate about well-defined engineering activities.
7. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes in field of engineering.

C. Expected Skills to be developed based on competency:

1. Design Effective page navigation.
2. Apply page routing concepts in angular.

D. Expected Course Outcomes (Cos)

Develop single page dynamic applications using Angular framework and APIs.

E. Practical Outcome (PRo)

Develop a page to demonstrate page navigation of product list using routing concepts.

F. Expected Affective domain Outcome (ADos)

1. Maintain tools and equipment.
2. Follow Coding standards and practices.
3. Follow safety practices.

4. Follow ethical practices

G. Prerequisite Theory: to design a product list with pagination navigation using routing concepts in Angular. First, create a new Angular project by running the following command in your terminal:

ng new product-list-app

Next, create a new component to display the product list. Run the following command in your terminal:

ng generate component product-list

Now, open the product-list.component.ts file and add the following code to define the product list:

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


interface Product {
 id: number;
 name: string;
}


@Component({
 selector: 'app-product-list',
 templateUrl: './product-list.component.html',
 styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {


 products: Product[] = [];


 constructor() { }


 ngOnInit(): void {
  // Populate the product list with some dummy data
```

```
    for (let i = 1; i <= 100; i++) {
      this.products.push({
        id: i,
        name: `Product ${i}`
      });
    }
  }
}
```

This code defines an interface for the product object, and initializes an empty array for the product list. In the ngOnInit method, we populate the product list with some dummy data.

Next, open the product-list.component.html file and add the following code to display the product list:

```
<h2>Product List</h2><ul>
  <li *ngFor="let product of products">{{ product.name }}</li>
</ul>
```

This code simply uses the *ngFor directive to loop through the products array and display each product's name in a list item.

Now, we'll create a pagination component to navigate between pages of the product list. Run the following command in your terminal:

ng generate component pagination

Open the pagination.component.ts file and add the following code to define the pagination component:

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


@Component({
  selector: 'app-pagination',
  templateUrl: './pagination.component.html',
  styleUrls: ['./pagination.component.css']
})
```

```
export class PaginationComponent implements OnInit {

  @Input() totalItems: number = 0;
  @Input() itemsPerPage: number = 10;
  @Input() currentPage: number = 1;
  @Input() totalPages: number = 0;

  constructor() { }

  ngOnInit(): void {
    this.setTotalPages();
  }

  setTotalPages() {
    this.totalPages = Math.ceil(this.totalItems / this.itemsPerPage);
  }
}
```

This code defines a pagination component that takes four input properties: totalItems, itemsPerPage, currentPage, and totalPages. In the ngOnInit method, we call the setTotalPages method to calculate the total number of pages based on the total number of items and the number of items per page.

Next, open the pagination.component.html file and add the following code to display the pagination links:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

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

```
export class ProductService {

  private apiUrl = 'https://example.com/api/products';

  constructor(private http: HttpClient) { }

  getProducts(page: number, pageSize: number) {
    const url = `${this.apiUrl}?page=${page}&pageSize=${pageSize}`;
    return this.http.get<any[]>(url);
  }

}
```

In this service, we're using HttpClient to make a GET request to the backend API with the current page and page size parameters.

Finally, let's create a template for the ProductListComponent that will display the list of products and paging navigation. Here's an example implementation:

```
<div *ngFor="let product of products">
  {{ product.name }}
</div>

<div>
  <button *ngIf="page > 1" [routerLink]="['/products']" [queryParams]="{ page: page - 1, pageSize: pageSize }">Previous</button>
  <button *ngIf="products?.length === pageSize" [routerLink]="['/products']" [queryParams]="{ page: page + 1, pageSize: pageSize }">Next</button>
</div>
```

In this template, we're using *ngFor to display each product in the products array. We're also using *ngIf to conditionally display the "Previous" and "Next" buttons based on whether we're on the first page or whether there are more products to display.

To use this component, you can add it to your routing configuration like this:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.component';

const routes: Routes = [
  { path: 'products', component: ProductListComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Now you can navigate to the /products URL in your app and see the product list with paging navigation.