Question 1:
Write an Angular service method that fetches a list of products from an API endpoint (https://fakestoreapi.com/products). Use RxJS operators pipe() and map() to transform the response so that it only returns an array of product names.

Example API Response

If the API returns:

```json
[
  { "title": "Smartphone XYZ", "price": 299.99 },
  { "title": "Laptop ABC", "price": 899.99 }
]
```

Then, getProductNames() will return an observable containing:

```typescript
["Smartphone XYZ", "Laptop ABC"]
```

Answer 1 :

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

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private apiUrl = 'https://fakestoreapi.com/products';

  constructor(private http: HttpClient) {}

  getProductNames(): Observable<string[]> {
    return this.http.get<{ title: string }[]>(this.apiUrl).pipe(
      map(products => products.map(product => product.title))
    );
  }
}
```

How It Works in the getProductNames() Method

1. this.http.get<{ title: string }[]>(this.apiUrl)

- Calls the API (https://fakestoreapi.com/products).
- The response is expected to be an array of objects, each containing a title property.

2 .pipe(map(products => products.map(product => product.title)))

- Uses RxJS map() to transform the response.
- Extracts the title property from each product.
- Converts the response from { title: string }[] to string[].

- <{ title: string }[]> ensures the API response is an **array of objects**, each containing a title.
- map() is used to **extract only the titles**, transforming the response into string[].

Alternative Type Definition

Instead of inline typing { title: string }[], you can define an **interface**:

```
interface Product {
  title: string;
}

getProductNames(): Observable<string[]> {
  return this.http.get<Product[]>(this.apiUrl).pipe(
    map(products => products.map(product => product.title))
  );
}
```

Question 2:

Write an example of how to create an Observable that emits the values 1, 2, 3 with a 1-second delay between each emission. Subscribe to it and log the values to the console.

Answer 2 :

```
import { Observable } from 'rxjs';

const numbers$ = new Observable<number>(observer => {
  let count = 1;
  const interval = setInterval(() => {
    observer.next(count++);
    if (count > 3) {
      observer.complete();
      clearInterval(interval);
    }
  }, 1000);
});

numbers$.subscribe(value => console.log(value));
```

Question 3:

Create an Angular component that displays a list of product names using the ProductService (from question 1). Fetch the data inside ngOnInit() and display it in the template.

Answer 3 :

```
import { Component, OnInit } from '@angular/core';
import { ProductService } from '../services/product.service';

@Component({
  selector: 'app-product-list',
  template: `
    <ul>
      <li *ngFor="let product of products">{{ product }}</li>
    </ul>
  `
})
export class ProductListComponent implements OnInit {
  products: string[] = [];

  constructor(private productService: ProductService) {}

  ngOnInit(): void {
    this.productService.getProductNames().subscribe(data => {
      this.products = data;
    });
  }
}
```

Question 4:

Write an Angular component that includes an input field where users can type their name. The name should be displayed in real-time below the input field using two-way data binding.

Answer 4 :

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

@Component({
  selector: 'app-user-input',
  template: `
    <input [(ngModel)]="name" placeholder="Enter your name">
    <p>Hello, {{ name }}!</p>
  `
})
export class UserInputComponent {
  name: string = '';
}
```

Question 5:

Create an Angular service that uses an RxJS Subject to allow components to share a message. Write a method to send messages and another to listen for messages.

Answer 5 :

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

@Injectable({
  providedIn: 'root'
})
export class MessageService {
  private messageSubject = new Subject<string>();

  sendMessage(message: string) {
    this.messageSubject.next(message);
  }

  getMessage() {
    return this.messageSubject.asObservable();
  }
}
```

Question 6:

The following RxJS implementation is supposed to filter out discounted products, but it's broken.

```
this.productService.getProducts()
  .pipe(
    map(products => products.filter(product => product.discount > 0)) // ❌ ERROR
  )
  .subscribe(filteredProducts => this.discountedProducts = filteredProducts);
```

Task:

1. **Fix the error** (Ensure that product.discount exists).
2. **Modify the map() function to filter correctly.**

- Check if product.discount is defined
- Ensure map() filters correctly

Answer 6 :

```
this.productService.getProducts()
  .pipe(
    map(products => products.filter(product => product.discount && product.discount > 0))
  )
  .subscribe(filteredProducts => this.discountedProducts = filteredProducts);
```

Question 7:

Write an Angular HttpInterceptor that attaches a Bearer token to all outgoing HTTP requests. The token should be retrieved from localStorage.

Answer 7 :

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = localStorage.getItem('authToken');

    if (token) {
      request = request.clone({
        setHeaders: { Authorization: `Bearer ${token}` }
      });
    }

    return next.handle(request);
  }
}
```

Question 8:

Create a simple Reactive Form in Angular with a single input field for email and a submit button. The form should be validated to check if the email input is not empty and follows a valid email format.

Answer 8 :

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-email-form',
  template: `
    <form [formGroup]="emailForm" (ngSubmit)="onSubmit()">
      <input formControlName="email" placeholder="Enter your email">
      <button type="submit" [disabled]="emailForm.invalid">Submit</button>
    </form>
    <p *ngIf="emailForm.controls.email.invalid && emailForm.controls.email.touched">
      Please enter a valid email.
    </p>
  `
})
export class EmailFormComponent {
  emailForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.emailForm = this.fb.group({
      email: ['', [Validators.required, Validators.email]]
    });
  }

  onSubmit() {
    console.log(this.emailForm.value);
  }
}
```

Question 9:

When the user selects a **theme**, it should update the UI dynamically.

Task:

1. Bind selectedTheme to a <select> dropdown.
2. When the user selects a theme, update the variable.

```
<label for="theme">Choose Theme:</label>
<select [(ngModel)]="selectedTheme">
  <option *ngFor="let theme of themes" [value]="theme">{{ theme }}</option>
</select>

<p>Selected Theme: {{ selectedTheme }}</p>
```

- Use [(ngModel)] for two-way binding
- Ensure the UI updates when the user selects a theme

Answer 9 :

Component (theme-selector.component.ts)

typescript

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

@Component({
  selector: 'app-theme-selector',
  templateUrl: './theme-selector.component.html',
  styleUrls: ['./theme-selector.component.css']
})
export class ThemeSelectorComponent {
  themes = ['Light', 'Dark', 'Blue', 'Green'];
  selectedTheme = 'Light';
}
```

Template (theme-selector.component.html)

html

```html
<label for="theme">Choose Theme:</label>
<select [(ngModel)]="selectedTheme">
  <option *ngFor="let theme of themes" [value]="theme">{{ theme }}</option>
</select>

<p>Selected Theme: {{ selectedTheme }}</p>
```

Question 10:

The following **NavigationService** does not update the component when the navigation menu is updated.

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

@Injectable({
  providedIn: 'root'
})
export class NavigationService {
  private menuItems = new Subject<string[]>();

  addMenuItem(item: string) {
    this.menuItems.value.push(item); // ❌ ERROR: 'value' is not correct
  }
}
```

Task:

1. **Fix the error** in addMenuItem().
2. **Ensure the new menu item updates the observable correctly.**

- Use next() instead of value.push()
- Make sure menuItems emits updated values

Answer 10 :

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

@Injectable({
  providedIn: 'root'
})
export class NavigationService {
  private menuItems = new BehaviorSubject<string[]>([]);
  menuItems$ = this.menuItems.asObservable();

  addMenuItem(item: string) {
    const updatedMenu = [...this.menuItems.getValue(), item];
    this.menuItems.next(updatedMenu);
  }
}
```

or

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

@Injectable({
  providedIn: 'root'
})
export class NavigationService {
  private menuItems = new BehaviorSubject<string[]>([]);
  menuItems$ = this.menuItems.asObservable();

  addMenuItem(item: string) {
    const currentMenu = this.menuItems.getValue(); // Get the current menu items
    const updatedMenu = currentMenu.concat(item); // Append the new item
    this.menuItems.next(updatedMenu); // Update the observable with the new menu
  }
}
```

1. **Replaces [...this.menuItems.getValue(), item] with .concat(item)** – The concat() method ensures that a **new array** is created while adding the new menu item.
2. **Prevents direct modification** – We **retrieve** the current menu using getValue() and then append the new item safely. **Ensures immutability** –
3. **Ensures Observer Subscription Works** – The updated array is emitted to all subscribers correctly.

Question 11:

The **product filter** component should allow users to **filter products by category**, but **it's not working**.

```
filterProducts(category: string) {
  return this.products.filter(p => p.category = category); // ❌ ERROR
}
```

Task:

1. Fix the filter() condition to compare correctly.
2. Ensure that the filtered products are updated in the UI.

- Use === (strict equality operator) instead of = in the filter condition
- Ensure the filter method correctly updates the product list

Answer 11:

```
filterProducts(category: string) {
  return this.products.filter(p => p.category === category);
}
```

Question 12:

The user should be able to **subscribe to navigation updates** when they add a new menu item.

Task:

1. Implement a NavigationService using BehaviorSubject.
2. Ensure that components **subscribe** to navigation updates dynamically.

- **Use** BehaviorSubject **instead of** Subject (We use BehaviorSubject so **new subscribers get the latest menu immediately**.)
- **Ensure navigation updates are reflected in real-time**

Answer 12:

BehaviorSubject:
1.Holds the latest value and emits it immediately to new subscribers.
2.Ensures that when a component subscribes, it gets the current menu.
3. Unlike Subject, which only emits values when data changes, BehaviorSubject stores the latest state.

```
// navigation.service.ts
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class NavigationService {
  private menuItems = new BehaviorSubject<string[]>(['Home', 'Shop', 'Contact']); // Initial
  menuItems$ = this.menuItems.asObservable(); // Observable for components

  addMenuItem(item: string) {
    const currentMenu = this.menuItems.getValue(); // Get current menu state
    const updatedMenu = currentMenu.concat(item); // Add new item
    this.menuItems.next(updatedMenu); // Notify subscribers of the updated menu
  }
}
```

```typescript
// navigation.component.ts
import { Component, OnInit } from '@angular/core';
import { NavigationService } from '../navigation.service';

@Component({
  selector: 'app-navigation',
  templateUrl: './navigation.component.html',
  styleUrls: ['./navigation.component.css']
})
export class NavigationComponent implements OnInit {
  menuItems: string[] = []; // Store menu items

  constructor(private navigationService: NavigationService) {}

  ngOnInit() {
    // Subscribe to menu updates and update menuItems dynamically
    this.navigationService.menuItems$.subscribe(items => {
      this.menuItems = items;
    });
  }
}
```

Navigation Template to Display Dynamic Menu

The **menu items should update instantly** when a new item is added.

```html
<!-- navigation.component.html -->
<ul>
  <li *ngFor="let item of menuItems">{{ item }}</li>
</ul>
```

Add Menu Items from Another Component

A separate **Admin Panel Component** allows users to **add new menu items dynamically**.

```typescript
// add-menu.component.ts
import { Component } from '@angular/core';
import { NavigationService } from '../navigation.service';

@Component({
  selector: 'app-add-menu',
  templateUrl: './add-menu.component.html'
})
export class AddMenuComponent {
  newItem = '';

  constructor(private navigationService: NavigationService) {}

  addItem() {
    if (this.newItem.trim()) {
      this.navigationService.addMenuItem(this.newItem);
      this.newItem = ''; // Clear input field after adding
    }
  }
}
```