### Section 1: Theoretical Questions

1. **Explain the role of Angular services in an application.**

Angular services help manage data and logic separately from components, making the app easier to maintain and scale. Here’s why they are important:

* Centralized Data Sharing: Services store data in one place, ensuring all components access the same information.
* Reusable Code: Functions in services can be used across multiple components, reducing duplication.
* Dependency Injection: Angular Dependency Injection allows components to easily access services, making the code more modular and testable.
* Separation of Concerns: Components focus on the user interface, while services handle data and logic.
* Scalability: Services keep the app organized as it grows, simplifying updates and maintenance.
* Flexibility for Future Changes: Static data in a service can later be replaced with API calls without impacting components.

In short, Angular services keep components lightweight, promote reuse, and ensure smooth growth and maintenance of the app.

1. **Describe how Angular handles HTTP requests and responses**

Angular provides a robust system for handling HTTP communication between the frontend and backend using its HttpClient module. This module helps manage tasks such as retrieving data, making API calls, handling errors, and supporting cross-domain requests, ensuring seamless interaction with web services.

* Fetching Data from APIs

With Angular, retrieving data from a server becomes simple. Using the HttpClient, developers can make GET requests to access information from remote servers. This data can then be displayed or processed in the application. It allows smooth integration of dynamic content and backend systems.

* Making Cross-Domain Requests (JSONP)

In situations where a server does not support regular cross-origin requests, Angular offers JSONP support. JSONP, or JSON with Padding, allows data to be fetched from a different domain without running into browser restrictions. This is useful when interacting with third-party APIs that permit this type of request.

* Handling Errors Gracefully

Angular provides tools to manage both client-side and server-side errors. If an issue occurs—such as a network failure or server error—the application can catch the error and respond appropriately. For example, it can display a meaningful error message to the user or log the issue for further investigation. This ensures the application remains stable and user-friendly even when problems arise.

Angular HttpClient simplifies HTTP communication by streamlining requests, supporting cross-origin solutions, and enabling error management. This makes the development of reliable, data-driven applications easier, as it ensures smooth communication between the client and backend services.

1. **What are the benefits of using RxJS in Angular applications?**

RxJS (Reactive Extensions for JavaScript) is a library that simplifies reactive programming by working with observables—streams of asynchronous data. It helps compose async or callback-based code and is especially useful for managing data streams, events, and API calls.

Benifits:

* Creating Observables:

From promises, events, timers, or AJAX requests.

Example: fromEvent() listens to mouse events, and interval() emits values on time intervals.

* Subjects:

A Subject act as both an observable and an observer, allowing values to be emitted and shared with multiple subscribers. This enables:

* Multicasting: All subscribers receive the same data.
* Message Service Pattern: Components send and receive messages via a centralized service.
* Operators

RxJS provides operators to transform or manage observable data. Some common operators include:

* map(): Transforms data (e.g., squaring numbers).
* filter(): Filters values based on a condition.
* catchError(): Handles errors gracefully and keeps streams running.
* shareReplay(): Converts unicast streams (independent for each subscriber) to multicast, sharing the latest value with all subscribers.
* Handling Errors

Use catchError() to handle errors during observable streams and provide fallback values.

* Multiple Subscribers:

Most observables are unicast, meaning each subscriber gets a fresh execution. To avoid repeated API calls, RxJS can multicast observables (e.g., using shareReplay()), so all subscribers share the same data.

RxJS makes it easy to build loose-coupling in applications, where one component produces data streams, and others consume them independently through observables and operators. This approach ensures flexibility and better management of asynchronous data flows.

1. **Explain how dependency injection works in Angular**

Dependency Injection (DI) is a core feature of Angular that simplifies managing dependencies between classes, such as components, directives, pipes, and services. It allows these elements to request the resources they need without having to create or manage them directly. Angular DI system promotes modularity, reusability, and testability by delegating object creation to a specialized mechanism known as the injector.

1. Core Concepts of Angular DI

Dependency Consumer:

This is the class (like a component or service) that needs access to some external resource or functionality.

Dependency Provider:

This refers to the service, object, or resource being provided to the consumer. The provider is registered with Angular to make the dependency available when required.

Injector:

The injector is responsible for managing instances of dependencies. It creates and provides them to the consumer classes when needed. Angular maintains an application-wide injector during the bootstrap process, along with other injectors scoped to individual components or modules.

2. How Dependency Injection Works in Angular

When a class declares a dependency in its constructor, Angular injector checks if an instance of the requested service or object is available. If the instance exists, it provides that instance to the class. If not, the injector creates a new instance using the registered provider and stores it for future use.

Angular uses a root injector to maintain shared services across the entire app, while also supporting hierarchical injectors for more specific scoping, such as at the component level. This ensures efficient resource management by sharing instances where appropriate and creating new ones only when necessary.

3. Providing Dependencies in Angular

To inject a class or service as a dependency, it must be registered with Angular. This process involves marking the class with the @Injectable decorator and specifying where the dependency should be provided.

Component-Level Providers

A service can be registered at the component level, meaning each new instance of that component will receive its own service instance.

Example

@Component({

selector: 'hero-list',

template: '...',

providers: [HeroService]

})

class HeroListComponent {}

This ensures that every instance of HeroListComponent has its own copy of HeroService.

Root-Level Providers

For services that need to be shared across the entire application, you can register them with providedIn: 'root' in the @Injectable decorator.

Example

@Injectable({

providedIn: 'root'

})

class HeroService {}

This approach ensures that a single instance of HeroService is shared throughout the app. Additionally, unused services are removed during the build process (tree-shaking), which improves performance.

Standalone Applications (ApplicationConfig)

In standalone apps, services can be registered using an ApplicationConfig object passed to the bootstrapApplication function.

Example

export const appConfig: ApplicationConfig = {

providers: [{ provide: HeroService }]

};

bootstrapApplication(AppComponent, appConfig);

This makes the service available throughout the application, similar to the root injector.

4. Injecting Dependencies

Angular provides two ways to inject dependencies into a class:

Constructor Injection

The most common method is to declare the dependency in the constructor. Angular uses the type information to determine which service to inject.

Example

@Component({ ... })

class HeroListComponent {

constructor(private heroService: HeroService) {}

}

Using the inject() Function

Angular also offers the inject() function, which can be used within the class body to obtain the service.

Example

@Component({ ... })

class HeroListComponent {

private heroService = inject(HeroService);

}

5. Managing Service Instances

Component-Level Instances:

When a service is provided at the component level, each component instance will receive a new copy of the service.

Root-Level Instance:

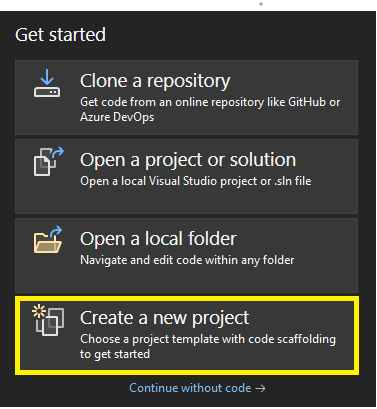
Services provided at the root level (with providedIn: 'root') share a single instance across the application, ensuring consistency and efficient resource use.

In Angular, Dependency Injection (DI) ensures that components, directives, and services can easily access the resources they need without creating them manually. By using an injector system, Angular efficiently manages the creation, reuse, and scope of dependencies. Developers can provide services at different levels—component, module, or root—based on the required scope, promoting flexibility and modular architecture. With tools like constructor injection and the inject() function, Angular makes it easy to manage dependencies and improve code maintainability.

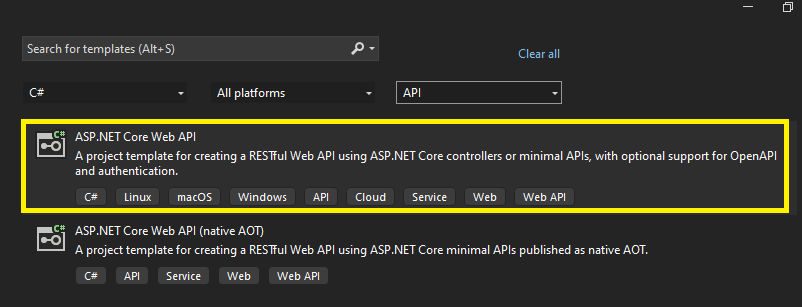
1. **Describe the process of setting up a C# Web API project.**

Create a New Web API Project

1. Open Visual Studio.
2. Click on Create a new project.



1. In the search box, type ASP.NET Core Web API and select it.



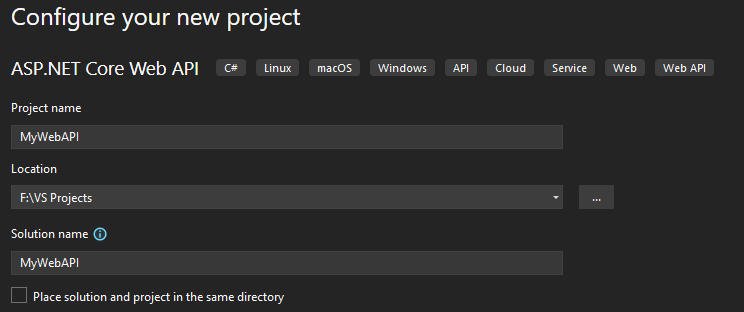
Click Next.

1. Configure the Project

Project Name: Enter a name, such as MyWebAPI.

Location: Choose a folder where the project will be saved.

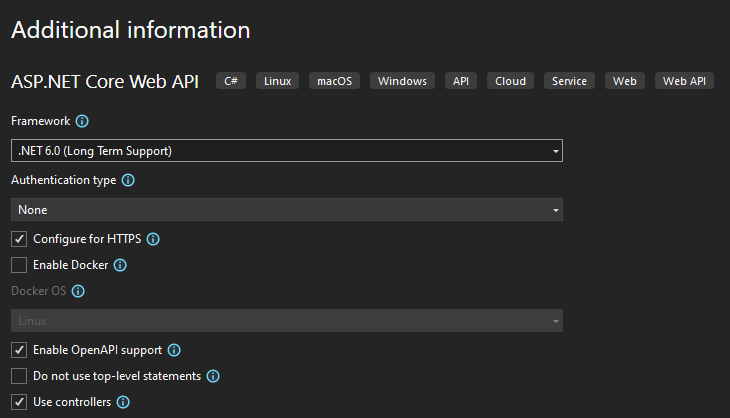
Solution Name: (Optional) Set a custom solution name if needed.



Click Next.

1. Set Target Framework and API Type

Framework: Select .NET 6.0 (Long-term support) or .NET 7.0 (or the latest available version).



Authentication Type: Select None for a simple API (you can configure it later if needed).

Check Use controllers (optional: leave enabled for easy setup).

Click Create.

6: Run the Web API

Once the project is created, press F5 or click the Run button to start the Web API.

The API will launch in a browser at a URL like:

https://localhost:5001

You will see the default WeatherForecast API in action.

7: Explore the Default Controller

In Solution Explorer, expand the Controllers folder.

Open the WeatherForecastController.cs file to see the example code provided by the template.

8: Add a New Controller

Right->click the Controllers folder → Add → Controller.

Choose API Controller - Empty and click Add.

Name your controller (e.g., ProductsController).

Inside the controller, add the following example

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

[HttpGet]

public IEnumerable<string> Get()

{

return new string[] { "Product1", "Product2" };

}

}

9: Test the API Using Swagger

Visual Studio automatically configures Swagger for API testing.

Run the project, and you'll see Swagger at:

https://localhost:5001/swagger/index.html

We can test API endpoints directly in the browser using Swagger’s UI.

10: Debugging and Modifications

You can set breakpoints in controller code and use F5 to debug.

Modify your API logic as needed by editing the controller files.

Optional: Install Additional Packages

In Tools → NuGet Package Manager → Manage NuGet Packages for Solution....

Search and install any necessary packages for project.

This guide provides everything need to quickly set up and run a C# Web API project using Visual Studio.

### Section 2: Practical Coding

Task 1

Task 2

Task 3

#### Task 2: Implementing the Front-End

#### Task 3: Implementing the Back-End

### Section 3: Debugging

1. **Identify and fix any issues in the provided Angular service code:**

Identified Issues:

* Missing Import for throwError:

The throwError function from rxjs is used in the handleError method but is not imported.

* Undefined Item Type:

The Item type is used in the getItems method's return type (Observable<Item[]>), but it's neither defined nor imported.

* Incorrect this Context in catchError:

Passing this.handleError directly to catchError can lead to an incorrect this context, causing runtime errors.

Step-by-Step Fixes:

* Import throwError:

Add throwError to the imports from rxjs.

* Define or Import the Item Interface:

If Item is defined elsewhere, ensure it's imported. If not, define it within the service or in a separate file.

* Bind the Correct this Context in catchError:

Use an arrow function to preserve the ‘this’ context when passing the error handler to catchError.

Fixed code

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

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

import { Observable, throwError } from 'rxjs'; // Imported throwError

import { catchError } from 'rxjs/operators';

export interface Item {

id: number;

name: string;

}

@Injectable({

providedIn: 'root'

})

export class ItemService {

private apiUrl = 'http://localhost:5000/api/items';

constructor(private http: HttpClient) {}

getItems(): Observable<Item[]> {

return this.http.get<Item[]>(this.apiUrl).pipe(

catchError(error => this.handleError(error)) // Preserved 'this' context

);

}

private handleError(error: any): Observable<never> {

console.error('An error occurred:', error);

return throwError('Something went wrong; please try again later.');

}

}

2. **Identify and fix any issues in the provided C# Web API controller code:**

Identified Issues:

* Missing GetItem Method:

The PostItem method uses CreatedAtAction referencing GetItems, which does not accept an id parameter. This should reference a GetItem method that retrieves a single item by its id.

* Missing Import for Item Model:

The Item model is used but not defined or imported. This will cause a compilation error unless Item is defined in the same file or appropriately imported.

* Thread Safety Concerns with static List:

Using a static List<Item> can lead to thread safety issues in a multi-threaded environment like a web server. Concurrent access can cause race conditions.

* Lack of Data Validation:

The PostItem and PutItem methods do not perform any validation on the incoming Item object, which can lead to inconsistent or invalid data being stored.

* No GetItem Method Implementation:

Typically, APIs provide a method to retrieve a single item by id. Its absence limits the API's functionality and causes issues with CreatedAtAction.

* No Error Handling for Duplicate IDs:

When adding a new item, there's no check to prevent adding an item with an existing Id, which can lead to data inconsistencies.

Fixed code

using Microsoft.AspNetCore.Mvc;

using System.Collections.Concurrent;

using System.Collections.Generic;

using System.Linq;

using MyApi.Models;

namespace MyApi.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ItemsController : ControllerBase

{

// Using ConcurrentDictionary for thread-safe operations

private static ConcurrentDictionary<int, Item> items = new ConcurrentDictionary<int, Item>();

// GET: api/Items

[HttpGet]

public ActionResult<IEnumerable<Item>> GetItems()

{

return Ok(items.Values);

}

// GET: api/Items/5

[HttpGet("{id}", Name = "GetItem")]

public ActionResult<Item> GetItem(int id)

{

if (items.TryGetValue(id, out var item))

{

return Ok(item);

}

return NotFound(new { Message = $"Item with Id = {id} not found." });

}

// POST: api/Items

[HttpPost]

public ActionResult<Item> PostItem([FromBody] Item item)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

if (items.ContainsKey(item.Id))

{

return Conflict(new { Message = $"An item with Id = {item.Id} already exists." });

}

if (items.TryAdd(item.Id, item))

{

return CreatedAtRoute("GetItem", new { id = item.Id }, item);

}

return StatusCode(500, new { Message = "An error occurred while creating the item." });

}

// PUT: api/Items/5

[HttpPut("{id}")]

public IActionResult PutItem(int id, [FromBody] Item item)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

if (id != item.Id)

{

return BadRequest(new { Message = "ID in URL does not match ID in the body." });

}

if (!items.ContainsKey(id))

{

return NotFound(new { Message = $"Item with Id = {id} not found." });

}

items[id] = item;

return NoContent();

}

// DELETE: api/Items/5

[HttpDelete("{id}")]

public IActionResult DeleteItem(int id)

{

if (items.TryRemove(id, out var removedItem))

{

return NoContent();

}

return NotFound(new { Message = $"Item with Id = {id} not found." });

}

}

}