**1. What is a RESTful Web Service, Web API, and Microservice?**

A **RESTful web service** is an architectural style that uses standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources identified by URIs. It follows principles such as statelessness and uniform interface.

A **Web API** is a framework provided by .NET (or other platforms) for building HTTP-based services that can be accessed by clients like browsers or mobile apps. It allows communication between systems using JSON or XML.

A **Microservice** refers to a small, self-contained unit of software designed to perform a single business function. Microservices are independently deployable and often communicate using RESTful APIs.

**2. Features of REST Architecture**

REST (Representational State Transfer) architecture includes the following key features:

* **Stateless**: Each request from the client must contain all the information needed; the server does not retain session state.
* **Resource-based**: Resources are identified using URIs, and operations are performed using standard HTTP methods.
* **Flexible message formats**: REST is not limited to XML; JSON is commonly used.
* **Layered system**: REST allows for layered architecture for scalability and security.
* **Microservice-friendly**: REST’s lightweight nature makes it ideal for use in microservice-based systems.

**3. What is HttpRequest and HttpResponse?**

An **HttpRequest** represents the incoming request from a client. It includes information such as the HTTP method, headers, query strings, and body content.

An **HttpResponse** represents the server’s reply to the client. It includes status codes, headers, and the body content (often in JSON or XML).

**4. Types of Action Verbs**

Web API uses HTTP action verbs to indicate the intended operation:

* **HttpGet**: Retrieves data from the server.
* **HttpPost**: Submits new data to the server.
* **HttpPut**: Updates existing data.
* **HttpDelete**: Removes data from the server.

These are declared using attributes like [HttpGet], [HttpPost], etc., on the respective controller action methods.

**5. Common HTTP Status Codes Used in Web API**

Web API controllers commonly return these status codes:

* **Ok (200)**: The request succeeded and the server returned the requested data.
* **BadRequest (400)**: The server could not process the request due to invalid input.
* **Unauthorized (401)**: Authentication is required and has failed or not been provided.
* **InternalServerError (500)**: A server-side error occurred during processing.

These codes are typically returned using helper methods like Ok(), BadRequest(), Unauthorized(), and StatusCode(500).

**6. Structure of a Web API (Read/Write Actions)**

A Web API typically includes:

* A **controller** class that inherits from ControllerBase or ApiController.
* **Action methods** within the controller that correspond to HTTP methods (GET, POST, PUT, DELETE).
* **Attributes** like [HttpGet], [HttpPost], etc., used to define the routing behavior.

This structure enables the API to handle CRUD (Create, Read, Update, Delete) operations efficiently.

**7. Configuration Files in .NET Core Web API**

In a .NET Core Web API project, several configuration files are used:

* **Startup.cs**: This file configures services, middleware, routing, and dependency injection.
* **appsettings.json**: Stores application-level settings such as connection strings and API keys.
* **launchSettings.json**: Defines how the application should launch during development, including environment variables and ports.

In the older .NET Framework (like 4.5):

* **Web.config**: Used for configuration settings such as authentication, database connections, and logging.
* **Route.config**: Contains route mappings to control how URLs are interpreted and matched to controller actions.

**8. Creating the First Web API in .NET Core**

When a developer creates a new .NET Core Web API project using the "API" template, the default setup includes a ValuesController with predefined action methods such as Get(), Post(), Put(), and Delete().

Once the project is created and executed, the developer can navigate to the endpoint (e.g., /api/values) to test the GET action. If everything is configured correctly, the API returns a JSON response, confirming that the GET method works as expected.

using System;

using System.Collections.Generic;

public class ValuesController

{

static List<string> items = new List<string> { "Apple", "Banana" };

public IEnumerable<string> Get() => items;

public string Post(string value) { items.Add(value); return "Item added"; }

public string Delete(int id) { if (id < 0 || id >= items.Count) return "Item not found"; items.RemoveAt(id); return "Item deleted"; }

}

public class Program

{

public static void Main()

{

var controller = new ValuesController();

Console.WriteLine("GET:");

foreach (var item in controller.Get()) Console.WriteLine(item);

Console.WriteLine("POST:");

Console.WriteLine(controller.Post("Orange"));

Console.WriteLine("GET After POST:");

foreach (var item in controller.Get()) Console.WriteLine(item);

Console.WriteLine("DELETE:");

Console.WriteLine(controller.Delete(1));

Console.WriteLine("GET After DELETE:");

foreach (var item in controller.Get()) Console.WriteLine(item);

}

}

