**Explain the concept of RESTful web service, Web API & Microservice**

**Features of REST architecture - Representational State Transfer, Stateless, Messages, Concept of Microservice, Difference between WebService & WebAPI, Not restricted to send XML as response.**

A RESTful Web Service follows the principles of Representational State Transfer (REST), an architectural style designed to simplify communication between systems over the web. REST emphasizes stateless communication, meaning each request from the client to the server must contain all the necessary information, as no session state is stored on the server. RESTful services operate around the concept of resources, which are identified via URIs and manipulated using standard HTTP methods like GET, POST, PUT, and DELETE. Messages in REST can be exchanged in multiple formats such as JSON, XML, or HTML, with JSON being the most popular today, highlighting that REST is not restricted to XML responses, unlike traditional web services.

A Web API is a framework—commonly implemented using ASP.NET Core or .NET Framework—that allows developers to build HTTP-based interfaces for accessing services or data. Web APIs adhere to REST principles and are optimized for delivering data, typically in JSON format, making them lightweight and easily consumable by browsers, mobile apps, and other services. Unlike traditional Web Services (which often use SOAP and are tightly coupled to XML and WSDL), Web APIs are simpler, faster, and more flexible, offering better performance for modern applications.

The Microservice architecture takes this a step further by breaking down a large application into a collection of small, independent services, each responsible for a specific business capability. These services are loosely coupled and communicate over lightweight protocols such as HTTP or message queues. Each microservice can be developed, deployed, and scaled independently, enabling faster development cycles and more resilient systems. RESTful principles are often used as the foundation for communication between microservices, making REST, Web API, and microservices highly complementary in modern application development.

Features of REST Architecture

|  |  |
| --- | --- |
| Feature | Explanation |
| Stateless | Server does not store client state. Every request is self-contained. |
| Uniform Interface | Single consistent API design regardless of resource type. |
| Cacheable | Responses must define themselves as cacheable or not. |
| Layered System | Clients can’t tell whether they’re connected directly to the server or an intermediary. |
| Representation | Resources can be sent in JSON, XML, HTML, etc. JSON is most common today. |
| Client-Server | Client and server roles are separated. |

**Explain what is HttpRequest & HttpResponse.**

In web communication, **HttpRequest** and **HttpResponse** are two fundamental components of the **client-server model** that enable data exchange over the internet using the **HTTP protocol**.

An **HttpRequest** is sent by the **client (usually a web browser or an app)** to the **server** when it wants to perform an action, such as retrieving a web page or submitting data. This request includes several key components:

* **HTTP method** (e.g., GET, POST, PUT, DELETE) that defines the type of operation.
* **URL** indicating the resource being accessed.
* **Headers** that carry metadata like content type, authorization tokens, etc.
* **Query parameters** appended to the URL for filtering or searching.
* **Body** (used in POST/PUT) to send data, such as form inputs or JSON payload.

An **HttpResponse**, on the other hand, is the server’s reply to the client’s request. It contains:

* **Status code** (e.g., 200 for success, 404 for not found, 500 for server error) indicating the result of the request.
* **Headers** that describe the response (like content type and caching rules).
* **Body** containing the actual data or message (often in JSON, XML, or HTML format).

Together, **HttpRequest** and **HttpResponse** form the foundation of how data is requested and delivered in any web-based application or API.

**List the types of Action Verbs**

**HttpGet, HttpPost, HttpPut, HttpDelete - Meaning of action verbs and how that should be declared as attributes for Web API**

In Web API, **Action Verbs** (also known as **HTTP methods**) define the type of operation a client wants to perform on a resource. These verbs are mapped to corresponding **controller actions** using attributes. Here are the commonly used action verbs:

|  |  |  |
| --- | --- | --- |
| Verb | Purpose | Attribute in API Controller |
| GET | Retrieve a resource | [HttpGet] |
| POST | Create a new resource | [HttpPost] |
| PUT | Update an existing resource | [HttpPut] |
| DELETE | Remove a resource | [HttpDelete] |

**List the types of HttpStatusCodes used in WebAPI**

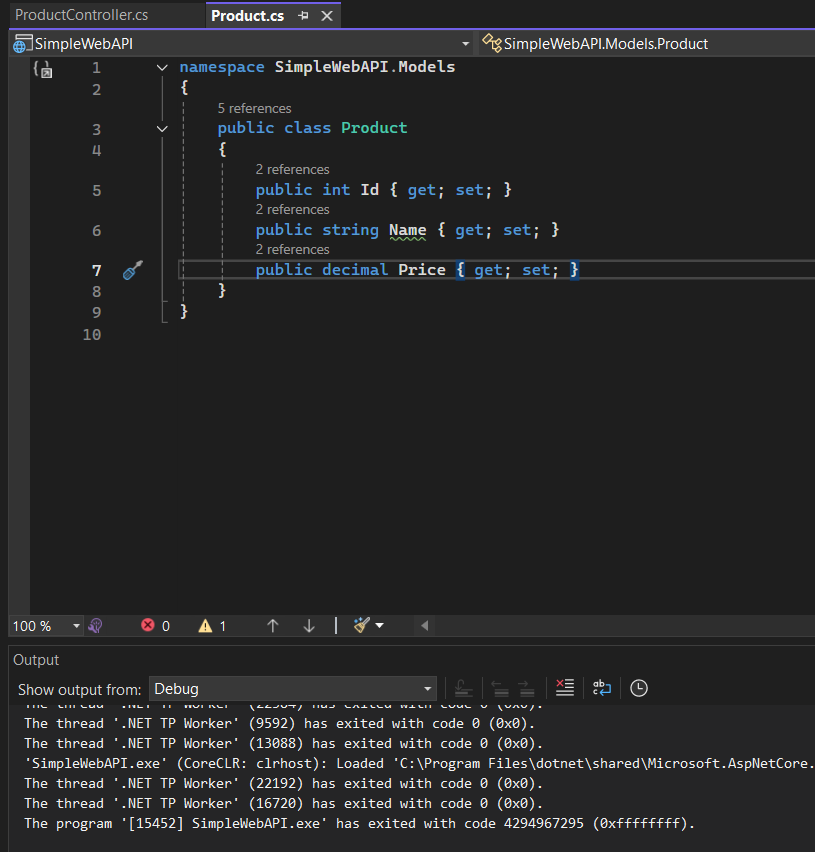
**Ok, InternalServerError, Unauthorized, BadRequest - All thru the action result types**

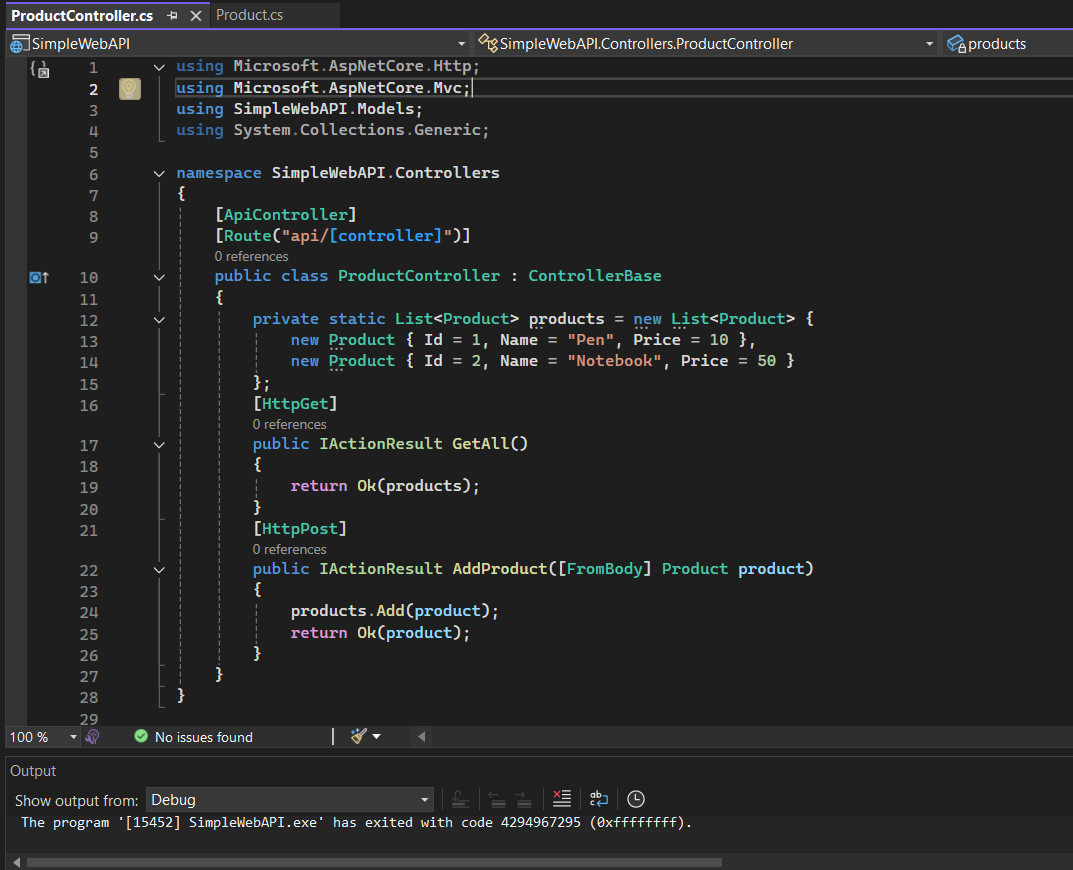
In Web API, HttpStatusCodes are used to communicate the result of an HTTP request from the server to the client. These codes are returned using Action Result types like Ok(), BadRequest(), etc., within controller methods. Here are some commonly used status codes and their corresponding action result methods:

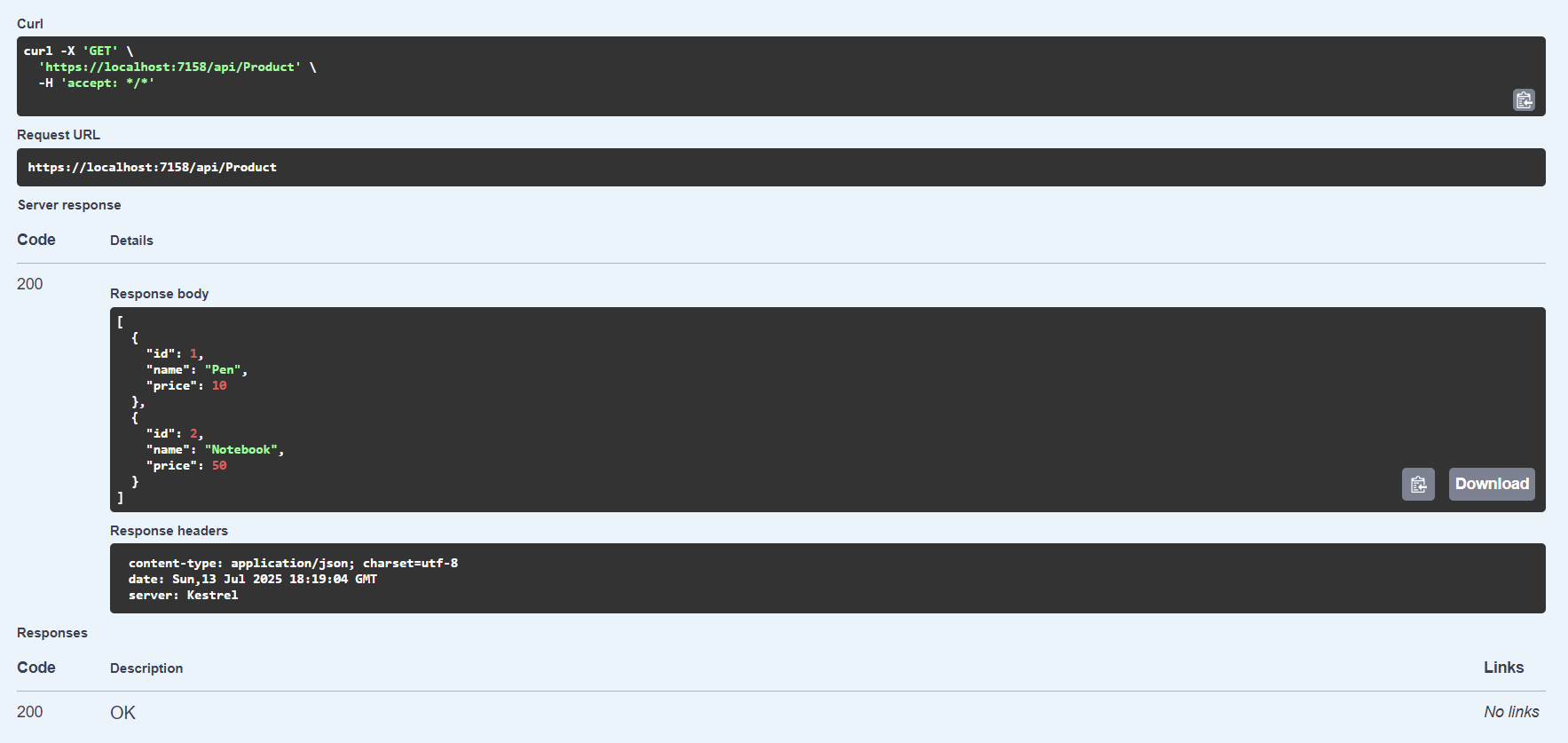
|  |  |  |  |
| --- | --- | --- | --- |
| HttpStatusCode | Numeric Code | Action Result Method | Description |
| OK | 200 | Ok() | Request processed successfully |
| Bad Request | 400 | BadRequest() | Client sent invalid or malformed request |
| Unauthorized | 401 | Unauthorized() | Authentication is missing or failed |
| Internal Server Error | 500 | StatusCode(500) | Server encountered an unexpected error |

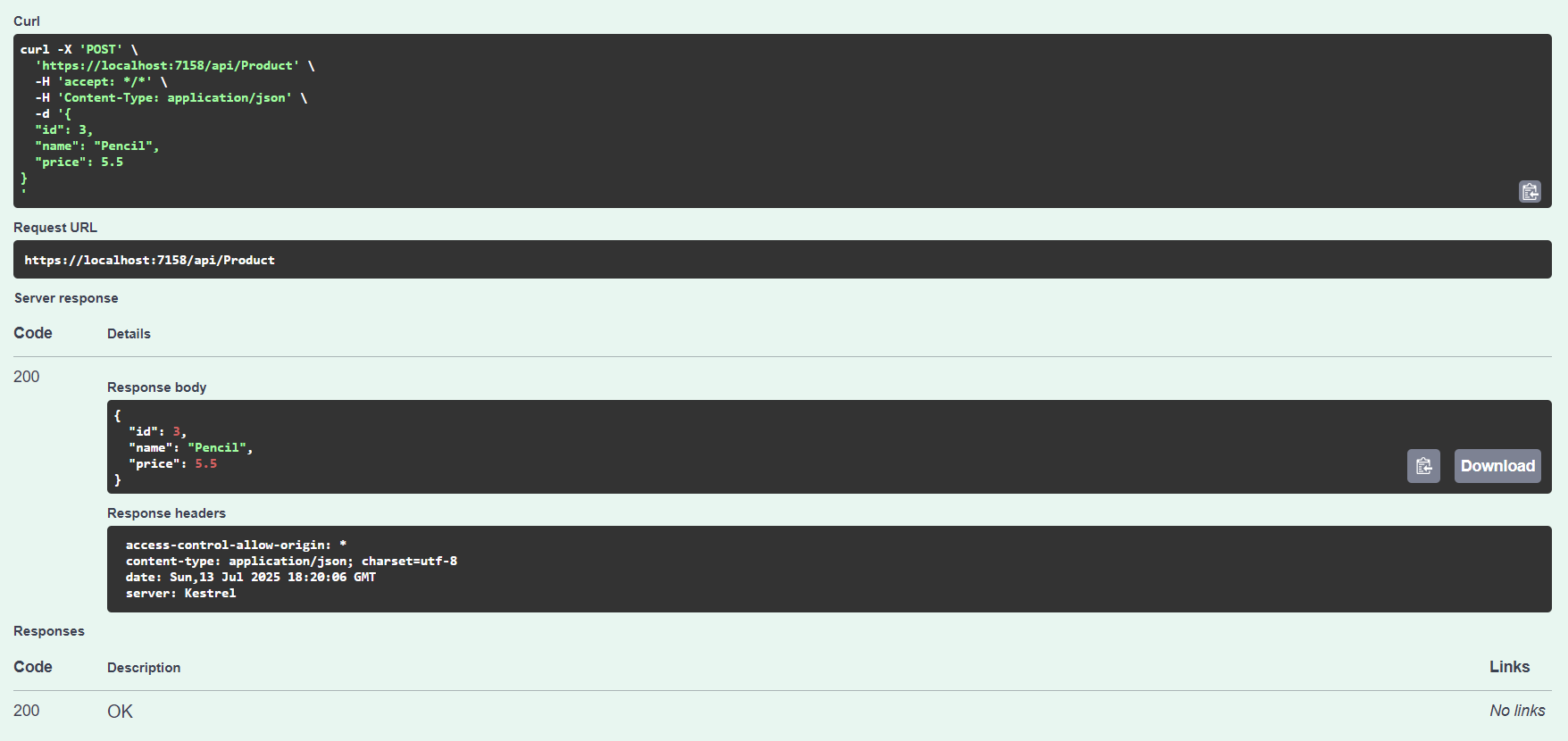
**Demonstrate creation of a simple WebAPI - With Read, Write actions**

**Structure of a web api - Controller & its inheritance from ApiController, Action verbs, Action method.**









**Explain the types of Configuration files of WebAPI**

**Startup.cs with depdency injection, appSettings.json, launchSettings.json, Explain Route.config & WebAPI.config in .Net 4.5 framework**

Startup.cs (ASP.NET Core): This file is the main entry point for configuring your ASP.NET Core application. It defines two methods: ConfigureServices, where you register services and dependency injection, and Configure, where you define the HTTP request pipeline, including routing, middleware, and endpoints.

appsettings.json (ASP.NET Core): This file stores application settings like connection strings, API keys, logging levels, or custom settings. It replaces the traditional web.config and is easy to read and update. You can access its values in code using dependency injection or the Configuration object.

launchSettings.json (ASP.NET Core): This is a development-only file that controls how the app is launched in Visual Studio. It sets the port number, environment (e.g., Development), and whether the browser should open automatically. It supports multiple profiles like Kestrel or IIS Express.

web.config (ASP.NET Web API in .NET Framework 4.5): This is the main configuration file in older .NET Framework projects. It holds settings for authentication, compilation, connection strings, and other system configurations. It is XML-based and must be updated manually.

WebApiConfig.cs (ASP.NET Web API in .NET Framework 4.5): This file defines the routing configuration for Web API in older projects. It maps URL patterns like api/{controller}/{id} to controller actions. It is registered in Global.asax.cs during application startup.