

# 1. Introduction to .NET Core Web API

## Basics of .NET Core Web API

- **Question:** What is a .NET Core Web API, and how is it different from a traditional web application?  
**Answer:** A .NET Core Web API is a framework for building HTTP services that can be consumed by web browsers, mobile apps, or other applications. Unlike traditional web applications, which return HTML pages, Web APIs return data in formats like JSON or XML, enabling communication between client and server applications.


## RESTful Services

- **Question:** What does it mean to build a RESTful Web API, and what are the key principles?  
**Answer:** REST (Representational State Transfer) is an architectural style for creating scalable web services. A RESTful Web API adheres to key principles like:
    - **Statelessness:** Each request from a client must contain all the necessary information.
    - **Use of HTTP methods:** Methods like GET, POST, PUT, DELETE are used according to their intended purpose.
    - **Uniform interface:** Resource-based URLs (e.g., `/api/products`).
    - **Client-Server separation.**
- 

# 2. Routing in .NET Core Web API

## Attribute Routing vs Convention-Based Routing


- **Question:** What is the difference between attribute-based routing and convention-based routing in .NET Core Web API?  
**Answer:**
  - **Convention-based routing** (defined in `Startup.cs`) uses a centralized route template for all controllers.
  - **Attribute-based routing** (defined directly in controllers) allows more flexibility by specifying routes at the action or controller level using `[Route]` attributes.

```
csharp  Copy code

[Route("api/[controller]")] public class ProductsController : ControllerBase {
    [HttpGet("{id}")] public IActionResult GetProduct(int id) { } }
```

## Route Constraints

- **Question:** What are route constraints in .NET Core Web API, and how are they used?  
**Answer:** Route constraints restrict the parameters that can be passed in the route. For example, restricting a route to accept only integer values:

```
csharp  Copy code

[Route("api/products/{id:int}")] public IActionResult GetProduct(int id) { }
```

---

# 3. HTTP Methods and Status Codes

## Using HTTP Methods

- **Question:** How are HTTP methods like GET, POST, PUT, and DELETE mapped to actions in .NET Core Web API?

**Answer:** HTTP methods are mapped to controller actions using attributes:

- **GET** for retrieving data ( `[HttpGet]` ).
- **POST** for creating resources ( `[HttpPost]` ).
- **PUT** for updating resources ( `[HttpPut]` ).
- **DELETE** for removing resources ( `[HttpDelete]` ).

## Status Codes

- **Question:** How do you return proper HTTP status codes in .NET Core Web API?

**Answer:** You can return appropriate status codes using predefined helper methods like:

- `Ok()` for 200 (Success),
- `NotFound()` for 404 (Resource not found),
- `BadRequest()` for 400 (Client error),
- `Created()` for 201 (Resource created),
- `NoContent()` for 204 (Successful but no content returned).

csharp



```
if (product == null) return NotFound(); return Ok(product);
```

---

## 4. Model Binding and Validation

### Model Binding

- **Question:** How does model binding work in .NET Core Web API?

**Answer:** Model binding in .NET Core Web API automatically maps data from HTTP requests (query strings, form data, or JSON bodies) to action method parameters or model classes. It simplifies the process of extracting data from HTTP requests into strongly-typed objects.

### Model Validation

- **Question:** How do you perform model validation in .NET Core Web API?

**Answer:** You can use data annotations like `[Required]` , `[StringLength]` , and `[Range]` on model properties. Model validation is triggered automatically, and validation errors can be checked using `ModelState.IsValid` . If the model is invalid, you can return a `BadRequest()` response with validation errors.

csharp



```
[HttpPost] public IActionResult CreateProduct([FromBody] Product model) { if (!ModelState.IsValid) return BadRequest(ModelState); return CreatedAtAction("GetProduct", new { id = model.Id }, model); }
```

---

## 5. Dependency Injection in .NET Core Web API

### Built-in Dependency Injection

- **Question:** How does dependency injection work in .NET Core Web API, and why is it important?

**Answer:** Dependency injection (DI) is built into .NET Core, allowing services to be injected into controllers or other services through constructor injection. It promotes loose coupling and testability. Services are registered in `Startup.cs` using `AddTransient()`, `AddScoped()`, or `AddSingleton()` for different lifetimes.

csharp

 Copy code

```
public class ProductsController : ControllerBase { private readonly IProductService
_productService; public ProductsController(IProductService productService) {
_productService = productService; } }
```

### Service Lifetimes

- **Question:** What are the different service lifetimes in .NET Core DI?

**Answer:**

- **Transient:** Services are created every time they are requested.
- **Scoped:** Services are created once per request.
- **Singleton:** Services are created once and reused throughout the application's lifetime.

---

## 6. Middleware in .NET Core Web API

### Custom Middleware

- **Question:** How do you create custom middleware in .NET Core Web API?

**Answer:** Custom middleware is a component that handles HTTP requests and responses. Middleware is added to the pipeline in `Startup.cs`. You create custom middleware by implementing a class with an `Invoke` or `InvokeAsync` method that takes an `HttpContext` as a parameter.

csharp

 Copy code

```
public class CustomMiddleware { private readonly RequestDelegate _next; public
CustomMiddleware(RequestDelegate next) { _next = next; } public async Task
InvokeAsync(HttpContext context) { // Custom logic await _next(context); // Call
the next middleware in the pipeline } }
```

### Adding Middleware to the Pipeline

- **Question:** How do you add middleware to the request pipeline in .NET Core?

**Answer:** Middleware components are added to the pipeline using `app.UseMiddleware<T>()` or predefined methods like `app.UseAuthentication()`, `app.UseRouting()`, etc., in the `Configure` method of `Startup.cs`.

csharp

 Copy code

```
public void Configure(IApplicationBuilder app) {
app.UseMiddleware<CustomMiddleware>(); app.UseRouting(); app.UseEndpoints(endpoints
=> { endpoints.MapControllers(); }); }
```


---

## 7. Authentication and Authorization

### JWT Authentication

- **Question:** How is JWT authentication implemented in .NET Core Web API?

**Answer:** JWT (JSON Web Token) authentication is implemented by configuring the authentication middleware in `Startup.cs` to validate tokens. Tokens are usually issued by an identity provider and are included in the request headers ( `Authorization: Bearer <token>` ).


```
csharp  Copy code

public void ConfigureServices(IServiceCollection services) {
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options => { options.TokenValidationParameters = new
            TokenValidationParameters { ValidateIssuer = true, ValidateAudience = true,
            ValidateLifetime = true, ValidateIssuerSigningKey = true, ValidIssuer =
            "yourIssuer", ValidAudience = "yourAudience", IssuerSigningKey = new
            SymmetricSecurityKey(Encoding.UTF8.GetBytes("yourSecretKey")) }; }); };
```

### Role-Based Authorization

- **Question:** How do you implement role-based authorization in .NET Core Web API?

**Answer:** Role-based authorization is implemented by decorating controller actions with `[Authorize(Roles = "Admin, Manager")]` . This ensures that only users in the specified roles can access the endpoint.

```
csharp  Copy code

[Authorize(Roles = "Admin")] [HttpPost("create")] public IActionResult
CreateProduct(Product model) { }
```

---


## 8. Error Handling in .NET Core Web API

### Global Error Handling

- **Question:** How do you implement global error handling in .NET Core Web API?

**Answer:** You can implement global error handling using exception-handling middleware.

In `Startup.cs` , you can add middleware to catch exceptions globally and return appropriate HTTP responses.


```
csharp  Copy code

public void Configure(IApplicationBuilder app) { app.UseExceptionHandler(errorApp
=> { errorApp.Run(async context => { context.Response.StatusCode = 500; await
context.Response.WriteAsync("An unexpected error occurred."); }); }); }
```

### Returning Custom Error Responses

- **Question:** How do you return custom error responses in .NET Core Web API?

**Answer:** Custom error responses can be returned using the `Problem()` or `BadRequest()` methods, which allow you to format error messages with relevant information. For more control, custom exceptions can be created and handled using middleware or filters.

```
csharp  Copy code

if (!ModelState.IsValid) { return BadRequest(new { message = "Invalid data", errors
= ModelState.Values.SelectMany(e => e.Errors) }); }
```

---

## 9. Versioning in .NET Core Web API

### API Versioning

- **Question:** How is API versioning implemented in .NET Core Web API?

**Answer:** API versioning allows multiple versions of an API to coexist. It can be implemented using the `Microsoft.AspNetCore.Mvc.Versioning` NuGet package. You can configure versioning via query strings, URL segments, or headers. Once added, you can annotate your controllers with version attributes.

```
[ApiVersion("1.0")] [Route("api/v{version:apiVersion}/products")] public class
ProductsV1Controller : ControllerBase { [HttpGet] public IActionResult GetV1() {
return Ok("Version 1"); } } [ApiVersion("2.0")]
[Route("api/v{version:apiVersion}/products")] public class ProductsV2Controller :
ControllerBase { [HttpGet] public IActionResult GetV2() { return Ok("Version 2"); }
}
```

### Configuring API Versioning

- **Question:** How do you configure API versioning in .NET Core Web API?

**Answer:** You configure API versioning in the `Startup.cs` by registering the versioning service:

```
public void ConfigureServices(IServiceCollection services) {
services.AddApiVersioning(options => { options.DefaultApiVersion = new
ApiVersion(1, 0); options.AssumeDefaultVersionWhenUnspecified = true;
options.ReportApiVersions = true; }); }
```

---

## 10. CORS (Cross-Origin Resource Sharing)

### Enabling CORS

- **Question:** How do you enable CORS in .NET Core Web API, and why is it necessary?

**Answer:** CORS allows resources to be requested from another domain, which is typically restricted by web browsers for security reasons. To enable CORS in .NET Core Web API, you need to configure CORS policies in `Startup.cs`.

```
public void ConfigureServices(IServiceCollection services) {
services.AddCors(options => { options.AddPolicy("AllowAll", builder =>
builder.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader()); }); } public void
Configure(IApplicationBuilder app) { app.UseCors("AllowAll"); }
```

### CORS Policies

- **Question:** What is the significance of configuring CORS policies, and how do you apply them selectively?

**Answer:** CORS policies can be configured to allow or restrict access to specific domains,

methods, or headers. You can apply CORS selectively by applying a policy at the controller or action level.

```
[EnableCors("AllowSpecificOrigins")] [HttpGet] public IActionResult GetProduct() {  
    return Ok(); }
```

---

## 11. Asynchronous Programming in .NET Core Web API

Using `async` and `await`

- **Question:** How do you implement asynchronous operations in .NET Core Web API?  
**Answer:** Asynchronous programming improves scalability by freeing up threads while awaiting external resources like database calls or API requests. In .NET Core Web API, you use the `async` keyword on actions, and the `await` keyword for I/O-bound operations like database queries.

```
[HttpGet("{id}")] public async Task<IActionResult> GetProduct(int id) { var product  
    = await _productService.GetProductAsync(id); if (product == null) return  
    NotFound(); return Ok(product); }
```

Benefits of Asynchronous Programming

- **Question:** What are the benefits of using asynchronous programming in .NET Core Web API?  
**Answer:** Asynchronous programming allows more efficient use of resources, improves scalability, and reduces thread blocking during I/O-bound operations, such as database access or external API calls. This leads to better performance and responsiveness in high-load applications.

---

## 12. Logging and Monitoring

Built-in Logging

- **Question:** How does logging work in .NET Core Web API, and what are some best practices?  
**Answer:** .NET Core provides built-in support for logging through `ILogger` and third-party providers (e.g., Serilog, NLog). Logging can be used to track errors, warnings, and informational messages.

```
private readonly ILogger<ProductsController> _logger; public  
ProductsController(ILogger<ProductsController> logger) { _logger = logger; } public  
IActionResult GetProduct(int id) { _logger.LogInformation("Fetching product with ID  
{id}", id); // Other logic }
```

Structured Logging

- **Question:** What is structured logging, and why is it important in .NET Core Web API?  
**Answer:** Structured logging captures log data as key-value pairs, enabling better querying, filtering, and analysis of log entries. This is useful for diagnostics and monitoring in production environments, where logs may need to be analyzed for specific data points like user IDs or request paths.

---

## 13. Health Checks in .NET Core Web API

### Implementing Health Checks

- **Question:** How do you implement health checks in .NET Core Web API?

**Answer:** Health checks monitor the health of the application, such as database connectivity or service dependencies. In .NET Core, health checks can be implemented using the `Microsoft.Extensions.Diagnostics.HealthChecks` package.

```
public void ConfigureServices(IServiceCollection services) {  
    services.AddHealthChecks().AddCheck<CustomHealthCheck>("CustomCheck")  
        .AddSqlServer(Configuration.GetConnectionString("DefaultConnection")); } public  
void Configure(IApplicationBuilder app) { app.UseHealthChecks("/health"); }
```

### Creating Custom Health Checks

- **Question:** How do you create custom health checks in .NET Core Web API?

**Answer:** Custom health checks can be created by implementing the `IHealthCheck` interface. This allows you to define custom logic to verify the health of any service or resource.

```
public class CustomHealthCheck : IHealthCheck { public Task<HealthCheckResult>  
CheckHealthAsync(HealthCheckContext context, CancellationToken cancellationToken =  
default) { bool healthCheckResultHealthy = true; if (healthCheckResultHealthy) {  
return Task.FromResult(HealthCheckResult.Healthy("A healthy result.")); } return  
Task.FromResult(HealthCheckResult.Unhealthy("An unhealthy result.")); } }
```

---

## 14. Rate Limiting in .NET Core Web API

### Implementing Rate Limiting

- **Question:** How can rate limiting be implemented in .NET Core Web API?

**Answer:** Rate limiting controls the number of requests a user or client can make within a specific time window, preventing abuse or overuse of resources. This can be implemented using third-party libraries like `AspNetCoreRateLimit` or through custom middleware that tracks client request counts.

### Using Middleware for Rate Limiting

- **Question:** How would you create custom middleware to limit the number of requests in .NET Core Web API?

**Answer:** You can create custom middleware that tracks the number of requests from each client (identified by IP address or API key) and throttles requests if the limit is exceeded.

```
public class RateLimitingMiddleware { private readonly RequestDelegate _next;  
private static Dictionary<string, int> _requestCounts = new Dictionary<string, int>  
( ); public RateLimitingMiddleware(RequestDelegate next) { _next = next; } public  
async Task Invoke(HttpContext context) { var clientId =  
context.Connection.RemoteIpAddress.ToString(); if  
(_requestCounts.ContainsKey(clientId)) { _requestCounts[clientId]++; } else {  
_requestCounts[clientId] = 1; } if (_requestCounts[clientId] > 100) { // Example  
limit context.Response.StatusCode = 429; // Too Many Requests await
```

```
context.Response.WriteAsync("Rate limit exceeded."); return; } await  
_next(context); } }
```

---