

1. Introduction to Web API

Basics of Web API

- **Question:** What is ASP.NET Web API, and how does it differ from ASP.NET MVC?

Answer: ASP.NET Web API is a framework for building HTTP-based services that can be consumed by a wide variety of clients, including browsers and mobile devices. While ASP.NET MVC is primarily used to return views (HTML), Web API is used to return data (typically JSON or XML). It focuses on RESTful services and is stateless, unlike ASP.NET MVC.

RESTful Services

- **Question:** What are RESTful services, and how are they implemented in Web API?

Answer: RESTful services follow the principles of Representational State Transfer (REST), where each resource (like users, orders) is identified by a URI and operations are performed using HTTP methods such as GET, POST, PUT, and DELETE. Web API maps these HTTP methods to controller actions.

2. HTTP Methods in Web API

Mapping HTTP Methods to Controller Actions

- **Question:** How does Web API map HTTP methods (GET, POST, PUT, DELETE) to controller actions?

Answer: In Web API, HTTP methods are mapped to controller actions based on method names or attributes like `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, and `[HttpDelete]`. Web API automatically maps actions with common names like `Get()`, `Post()`, `Put()`, and `Delete()` to the respective HTTP methods.

csharp

 Copy code

```
[HttpGet] public IEnumerable<Product> GetProducts() { return _productService.GetAll(); }
```

Custom HTTP Method Mapping

- **Question:** How can you map custom HTTP methods to controller actions in Web API?

Answer: You can map custom HTTP methods using attributes like `[AcceptVerbs("PATCH", "HEAD")]`. This allows you to handle non-standard HTTP methods or map multiple methods to a single action.

csharp

 Copy code

```
[AcceptVerbs("PATCH")] public IHttpActionResult UpdatePartially(int id, JsonPatchDocument<Product> patchDoc) { // Handle partial update }
```

3. Routing in Web API

Attribute Routing

- **Question:** How is attribute routing implemented in Web API?

Answer: Attribute routing allows defining routes directly on controller actions using attributes like `[Route]` and `[RoutePrefix]`. It provides greater control over the URI structure and is more flexible than conventional routing.

csharp


 Copy code

```
[Route("api/products/{id:int}")] public IActionResult GetProductById(int id) {  
    var product = _productService.GetById(id); return Ok(product); }
```

Conventional Routing

- **Question:** What is conventional routing in Web API, and how is it configured?

Answer: Conventional routing in Web API is defined in the `WebApiConfig.cs` file using the `MapHttpRoute` method. It follows a pattern like `api/{controller}/{id}`, where the controller and action are inferred from the URL.


```
csharp  Copy code  
  
config.Routes.MapHttpRoute( name: "DefaultApi", routeTemplate:  
    "api/{controller}/{id}", defaults: new { id = RouteParameter.Optional } );
```

4. Action Results and Responses

Returning Data in Web API

- **Question:** How do you return data in Web API from a controller action?


Answer: Data in Web API is returned using `IHttpActionResult` or directly as a type like `IEnumerable<T>`. Common responses include `Ok()` (HTTP 200), `NotFound()` (HTTP 404), and `BadRequest()` (HTTP 400). These methods help return both data and HTTP status codes.

```
csharp  Copy code  
  
public IActionResult GetProduct(int id) { var product =  
    _productService.GetById(id); if (product == null) { return NotFound(); } return  
    Ok(product); }
```

JSON and XML Formatting

- **Question:** How does Web API handle JSON and XML formatting of responses?

Answer: By default, Web API returns JSON or XML based on the `Accept` header sent by the client. It uses media type formatters to convert objects into JSON or XML. You can configure which formatters to use in `HttpConfiguration`.


```
csharp  Copy code  
  
config.Formatters.JsonFormatter.SerializerSettings.Formatting =  
    Formatting.Indented;
```

5. Model Binding in Web API

Model Binding from HTTP Requests

- **Question:** How does Web API bind data from HTTP requests to action method parameters?

Answer: Web API automatically binds data from the request body (for POST and PUT) or query strings (for GET) to action method parameters. This process is called model binding and works for primitive types, complex types, and collections.

```
csharp  Copy code  
  
public IActionResult CreateProduct([FromBody] Product product) { if  
    (ModelState.IsValid) { _productService.Create(product); return
```


```
CreatedAtRoute("DefaultApi", new { id = product.Id }, product); } return  
BadRequest(ModelState); }
```

Binding Complex Types

- **Question:** How do you bind complex types in Web API from HTTP requests?

Answer: Complex types are typically bound from the request body for POST and PUT requests. The request body is automatically deserialized into the specified complex type. You can use `[FromBody]` explicitly to bind from the request body.

```
csharp
```

 Copy code


```
public IActionResult UpdateProduct([FromBody] Product product) {  
    _productService.Update(product); return Ok(); }  
}
```

6. Error Handling in Web API

Global Exception Handling

- **Question:** How do you implement global exception handling in Web API?

Answer: Global exception handling is implemented using `ExceptionHandler` or `ExceptionHandler`. You can create a custom exception handler by inheriting from `ExceptionHandler` and registering it in `HttpConfiguration`.

```
csharp  Copy code
```

```
public class GlobalExceptionHandler : ExceptionHandler { public override void
Handle(ExceptionHandlerContext context) { context.Result = new
ResponseMessageResult(context.Request.CreateErrorResponse(HttpStatusCode.InternalServe
"An error occurred")); } } config.Services.Replace(typeof(IExceptionHandler), new
GlobalExceptionHandler());
```

Action-Specific Error Handling

- **Question:** How do you handle exceptions within individual Web API actions?

Answer: You can handle exceptions within specific Web API actions using try-catch blocks. Additionally, you can return specific error responses like `BadRequest()` or `NotFound()`.

```
csharp
```

```
public IActionResult GetProduct(int id) { try { var product =  
_productService.GetById(id); if (product == null) { return NotFound(); } return  
Ok(product); } catch (Exception ex) { return InternalServerError(ex); } }
```

7. Authorization and Security in Web API

Basic Authentication

- **Question:** How do you implement basic authentication in Web API?

Answer: Basic authentication can be implemented using a custom `AuthorizeAttribute` or `AuthenticationFilter` that checks the `Authorization` header for valid credentials.

```
csharp
```


 Copy code

```
public class BasicAuthenticationAttribute : AuthorizationFilterAttribute { public
override void OnAuthorization(HttpContext actionContext) { var authHeader =
actionContext.Request.Headers.Authorization; if (authHeader == null ||
authHeader.Scheme != "Basic") { actionContext.Response =
actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized); return; } //
Check credentials } }
```

Token-Based Authentication (JWT)

- **Question:** How do you implement JWT (JSON Web Token) authentication in Web API?

Answer: JWT-based authentication is implemented by validating the token in the Authorization header of HTTP requests. You can use libraries like `System.IdentityModel.Tokens.Jwt` to validate and issue JWT tokens.

```
csharp  Copy code

var tokenHandler = new JwtSecurityTokenHandler(); var tokenDescriptor = new
SecurityTokenDescriptor { Subject = new ClaimsIdentity(new Claim[] { new
Claim(ClaimTypes.Name, user.Name) }), Expires = DateTime.UtcNow.AddDays(7),
SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature) }; var token =
tokenHandler.CreateToken(tokenDescriptor); return tokenHandler.WriteToken(token);
```

8. CORS (Cross-Origin Resource Sharing)

Enabling CORS

- **Question:** How do you enable CORS in Web API?

Answer: CORS (Cross-Origin Resource Sharing) allows web applications from one domain to make requests to APIs on another domain. In Web API, you enable CORS using the `EnableCorsAttribute` or through configuration in `HttpConfiguration`.

```
csharp  Copy code

config.EnableCors(new EnableCorsAttribute("https://example.com", "*", "*"));
```

9. Versioning in Web API

API Versioning

- **Question:** How do you implement versioning in Web API?

Answer: API versioning can be implemented using various approaches such as:

- **URL Versioning:** Adding the version number to the URL (e.g., `api/v1/products`).
- **Query String Versioning:** Passing the version number as a query parameter (e.g., `api/products?version=1`).
- **Header Versioning:** Including the version in a custom HTTP header.

For URL versioning, the route template is updated to include the version:

```
[Route("api/v1/products")] public class ProductsV1Controller : ApiController {
    public IEnumerable<Product> GetProducts() { return _productService.GetAll(); } }
```

Versioning via Header

- **Question:** How do you implement header-based versioning in Web API?

Answer: Header-based versioning can be implemented by checking for a custom version header in the request and routing the request to the appropriate controller.

```
public class ApiVersionHandler : DelegatingHandler {
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken) {
        if (request.Headers.Contains("api-version")) {
            var version = request.Headers.GetValues("api-version").FirstOrDefault();
            // Logic for routing based on version
        }
        return await base.SendAsync(request, cancellationToken);
    }
}
```

10. Testing in Web API

Unit Testing Controllers

- **Question:** How do you unit test Web API controllers?

Answer: Unit testing Web API controllers involves testing the action methods to ensure they return the correct `HttpResponseMessage` or `IHttpActionResult` . Mocking frameworks like Moq can be used to mock dependencies and isolate the controller logic.

```
[TestMethod] public void GetProduct_ReturnsOk() {
    var mockService = new Mock<IProductService>();
    mockService.Setup(service => service.GetById(1)).Returns(new Product { Id = 1 });
    var controller = new ProductsController(mockService.Object);
    var result = controller.GetProduct(1) as OkNegotiatedContentResult<Product>;
    Assert.IsNotNull(result);
    Assert.AreEqual(1, result.Content.Id);
}
```

Integration Testing

- **Question:** How do you perform integration testing for a Web API?

Answer: Integration testing involves testing the Web API endpoints in a real or in-memory environment, ensuring the components work together. You can use `HttpClient` to send requests to the API and verify responses.

```
[TestMethod] public async Task GetProducts_ReturnsProducts() {
    var client = new HttpClient { BaseAddress = new Uri("http://localhost:5000") };
    var response = await client.GetAsync("api/products");
    response.EnsureSuccessStatusCode();
    var products = await response.Content.ReadAsAsync<IEnumerable<Product>>();
    Assert.IsTrue(products.Any());
}
```

11. Throttling and Rate Limiting

Implementing Rate Limiting

- **Question:** How do you implement rate limiting in Web API?

Answer: Rate limiting can be implemented by restricting the number of requests a client can make to the API in a given time window. You can use third-party libraries like `AspNetCoreRateLimit` or implement it manually by tracking client requests and applying logic to throttle excessive requests.

```
public class RateLimitHandler : DelegatingHandler { private static
Dictionary<string, int> _requestCounts = new Dictionary<string, int>(); protected
override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
Cancellation token cancellationToken) { var clientId = request.GetClientIpAddress();
if (_requestCounts.ContainsKey(clientId) && _requestCounts[clientId] >= 100) {
return request.CreateResponse(HttpStatusCode.TooManyRequests, "Rate limit
exceeded"); } _requestCounts[clientId] = _requestCounts.ContainsKey(clientId) ?
_requestCounts[clientId] + 1 : 1; return await base.SendAsync(request,
cancellationToken); } }
```

12. Logging and Monitoring

Logging in Web API

- **Question:** How do you implement logging in Web API?

Answer: Logging can be implemented using logging frameworks like NLog, Serilog, or the built-in `ILogger` in ASP.NET Core. You can log important events, errors, and request details by injecting a logger into your controllers or using middleware to log each request.

```
public class ProductsController : ApiController { private readonly
ILogger<ProductsController> _logger; public
ProductsController(ILogger<ProductsController> logger) { _logger = logger; } public
IHttpActionResult GetProducts() { _logger.LogInformation("Fetching all products");
return Ok(_productService.GetAll()); } }
```

Monitoring with Application Insights

- **Question:** How do you integrate Application Insights for monitoring in Web API?

Answer: Application Insights can be integrated into a Web API project to monitor application performance, request statistics, exceptions, and more. You install the Application Insights SDK and configure it to track telemetry data.

```
public class ProductsController : ApiController { private readonly TelemetryClient
_telemetry; public ProductsController() { _telemetry = new TelemetryClient(); }
public IHttpActionResult GetProducts() { _telemetry.TrackEvent("GetProducts
called"); return Ok(_productService.GetAll()); } }
```

13. Dependency Injection (DI) in Web API

Implementing Dependency Injection

- **Question:** How do you implement dependency injection in Web API?

Answer: Dependency Injection in Web API can be implemented using libraries like Unity, Autofac, or Ninject, or by using the built-in DI support in ASP.NET Core. Services are registered in `Startup.cs` and injected into controllers via constructors.

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

Registering Services in ASP.NET Core

- **Question:** How do you register services for dependency injection in ASP.NET Core Web API?

Answer: Services are registered in the `Startup.ConfigureServices` method using `services.AddScoped()`, `services.AddSingleton()`, or `services.AddTransient()`. These define the lifetime of the service.

```
public void ConfigureServices(IServiceCollection services) {
services.AddScoped<IProductService, ProductService>(); }
```

14. Caching in Web API

Implementing Response Caching

- **Question:** How do you implement response caching in Web API?

Answer: Response caching is implemented to store a copy of the response and reuse it for subsequent requests. In ASP.NET Core, this can be done using the `ResponseCache` attribute or `Cache-Control` headers.

```
[ResponseCache(Duration = 60)] public IActionResult GetProducts() { return
Ok(_productService.GetAll()); }
```

Using In-Memory Caching

- **Question:** How do you implement in-memory caching in Web API?

Answer: In-memory caching is used to store frequently accessed data in memory to avoid querying the database repeatedly. You can use `IMemoryCache` in ASP.NET Core for this purpose.

```
public class ProductsController : ApiController { private readonly IMemoryCache
_cache; public ProductsController(IMemoryCache cache) { _cache = cache; } public
IActionResult GetProducts() { if (!_cache.TryGetValue("products", out
List<Product> products)) { products = _productService.GetAll();
_cache.Set("products", products, TimeSpan.FromMinutes(5)); } return Ok(products); }
}
```

15. OData in Web API

Implementing OData Queries

- **Question:** How do you implement OData queries in Web API?

Answer: OData is implemented in Web API to enable advanced querying options such as filtering, sorting, and paging. You can install the OData package and decorate controller actions with `[EnableQuery]` to allow OData query options.

```
[EnableQuery] public IQueryable<Product> GetProducts() { return  
    _productService.GetAll().AsQueryable(); }
```
