Lambda expressions

Lambda expressions in C# are a **concise way to represent anonymous methods (methods with no names) or expressions**.

They are especially useful for writing short pieces of code that can be passed as arguments or returned from other methods, commonly used with LINQ (Language Integrated Query) and when working with events or callback methods.

Syntax of Lambda Expressions

```
(argument-list) => expression
```

lub

```
(argument-list) => { statement(s); }
```

Simple example

```
var method= () => 42
```

```
var method2= (x, y) =>
{
    int result = x + y;
    return result;
}
```

Action delegate

The Action delegate type is used to represent a method that **has a void return type and may take zero or more arguments**. It is defined in the System namespace.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Action with no parameters
        Action printHello = () => Console.WriteLine("Hello");
        printHello();

        // Action with one parameter
        Action<string> greet = name =>
Console.WriteLine($"Hello, {name}!");
        greet("World");

        // Action with two parameters
        Action<int, int> addAndPrint = (x, y) =>
Console.WriteLine($"{x} + {y} = {x + y}");
        addAndPrint(5, 7);
    }
}
```

Function

The Func delegate type is used to represent a method that returns a value and may take zero or more input parameters. The last type parameter specifies the return type.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Func with no input parameters, returning an integer
        Func<int> getRandomNumber = () => new Random().Next(1,
100);
        Console.WriteLine(getRandomNumber());

        // Func with one input parameter, returning a string
        Func<int, string> numberToString = number => $"Number
is: {number}";
        Console.WriteLine(numberToString(42));

        // Func with two input parameters, returning their sum
        Func<int, int, int> add = (x, y) => x + y;
        Console.WriteLine(add(5, 3));
    }
}
```

Different kinds of applications - shift to web development

The transition from desktop-based to cloud-based applications represents a **significant shift in the way software is developed, deployed, and utilized**.

This evolution can be understood through various dimensions, including technological advancements, changing user expectations, and economic factors.

Desktop-Based Applications

- **Definition**: Software that is installed and runs on a single computer or on a local network.
- **Characteristics**: High performance for specific tasks, works offline, and data security is managed individually.
- **Development Era**: Predominant from the early days of personal computing until the late 2000s.

Cloud-Based Applications

- **Definition**: Software that is hosted on remote servers and accessed over the internet.
- **Characteristics**: Accessible from anywhere with internet, scalable, updates are seamless, and often subscription-based.
- **Development Era**: Began gaining prominence in the late 2000s and became the norm in the 2010s.

Drivers of the Shift

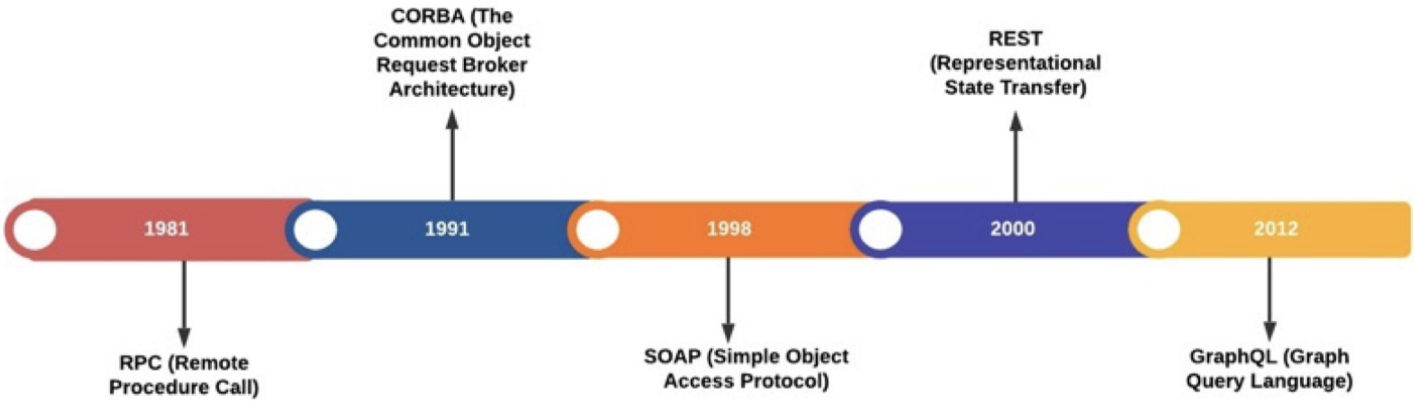
1. **Technological Advancements**: The rise of high-speed internet and improvements in web technologies (HTML5, JavaScript, cloud computing infrastructure) made cloud-based applications feasible and reliable.
2. **Changing User Expectations**: Users started expecting more flexibility, such as the ability to access their data and applications from anywhere, on any device. Cloud applications met these expectations perfectly.

3. **Economic Factors**: For developers and companies, cloud-based models offer recurring revenue through subscriptions and lower distribution costs. For users, it offers lower upfront costs and the ability to scale usage as needed.
4. **Collaboration and Integration**: Cloud-based applications facilitate easier collaboration in real-time and integration with other services, enhancing productivity and workflow.
5. **Security and Compliance**: While initially considered a concern, cloud providers have significantly invested in security, offering levels of data protection and compliance that can be superior to what individual organizations can achieve.
6. **Scalability and Maintenance**: Cloud apps can be quickly scaled to accommodate growing numbers of users, and updates are managed by the service provider, ensuring all users have the latest version without needing to manually update the software.

Implications for Development

- **Skill Set Shift**: Developers have had to adapt to new programming languages, frameworks, and architectures suited to web and cloud development (e.g., microservices architecture, containerization).
- **Development and Deployment Cycle**: The cycle has become more agile, with continuous integration and deployment (CI/CD) enabling faster updates and improvements.
- **Security Considerations**: Developers now need to consider data privacy and security in a shared environment, adhering to various compliance standards.

Communication between heterogeneous systems



RPC (Remote Procedure Call)

- **=Concept=**: Allows a program on one computer to execute a procedure (function) on another computer without needing to understand the network's details.
- **=Key Features=**: Procedure call abstraction, hides the complexities of the network.
- **=Use Case=**: Ideal for internal communications within the same organization, where you have control over both ends of the communication.

CORBA (Common Object Request Broker Architecture)

- **=Concept=**: A standard defined by the Object Management Group (OMG) that allows objects to communicate with one another regardless of the programming language, operating system, or hardware they run on.
- **=Key Features=**: Language and platform independence, supports complex object-oriented communication.
- **Use Case**: Suited for large-scale enterprise applications requiring robust cross-platform communication.

SOAP (Simple Object Access Protocol)

- **Concept**: A protocol for exchanging structured information in the implementation of web services in computer networks. It relies on XML for its message format.
- **Key Features**: Protocol independence (can be used over HTTP, SMTP, TCP, etc.), high security (supports WS-Security), extensive standards support for message integrity and authentication.
- **Use Case**: Suitable for enterprise-level web services where security and transaction compliance are crucial

```
<?xml version="1.0"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:usr="http://example.com/userService">
   <soapenv:Header/>
   <soapenv:Body>
      <usr:GetUserDetailsRequest>
         <usr:UserID>12345</usr:UserID>
      </usr:GetUserDetailsRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<?xml version="1.0"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:usr="http://example.com/userService">
   <soapenv:Header/>
   <soapenv:Body>
      <usr:GetUserDetailsResponse>
         <usr:User>
            <usr:UserID>12345</usr:UserID>
            <usr:Name>John Doe</usr:Name>
            <usr:Email>john.doe@example.com</usr:Email>
         </usr:User>
      </usr:GetUserDetailsResponse>
```

```
    </soapenv:Body>
</soapenv:Envelope>
```

REST (Representational State Transfer)

- **Concept**: An architectural style for designing networked applications, where web services are viewed as resources and can be identified by their URLs.
- **Key Features**: Uses standard HTTP methods (GET, POST, PUT, DELETE), lightweight (no extensive processing like SOAP), stateless.
- **Use Case**: Ideal for web APIs used by mobile apps, web apps, and other client-side applications that require efficient, stateless communication.

```
GET /users/12345 HTTP/1.1
Host: example.com
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 123

{
    "userID": "12345",
    "name": "John Doe",
    "email": "john.doe@example.com"
}
```

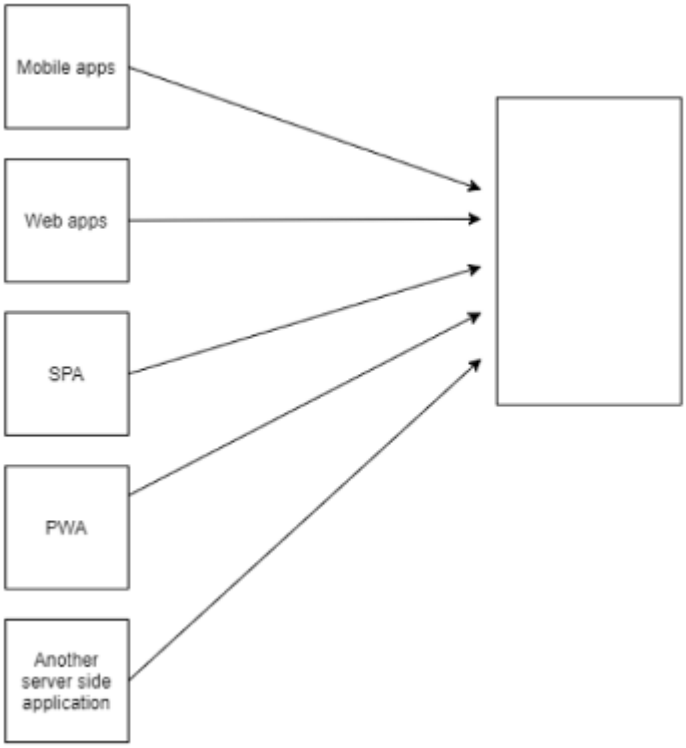
GraphQL

- **Concept**: A query language for your API, and a server-side runtime for executing queries by using a type system you define for your data.
- **Key Features**: Clients can request exactly what they need, easy to evolve APIs over time, enables powerful developer tools.
- **Use Case**: Perfect for complex systems and mobile applications where bandwidth efficiency and flexible data retrieval are key.

```
query {
  book(id: "123") {
    title
    author {
      name
    }
  }
}
```

```
{
  "data": {
    "book": {
      "title": "The Great Gatsby",
      "author": {
        "name": "F. Scott Fitzgerald"
      }
    }
  }
}
```

Representational State Transfer (REST)



Original dissertation:
https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

REST, which stands for Representational State Transfer, is an architectural style for designing networked applications. It was defined by Roy Fielding in his PhD dissertation at the University of California, Irvine, in 2000. Fielding is one of the principal authors of the HTTP specification versions 1.0 and 1.1.

The REST architectural style was developed as a way to harness the existing capabilities of the Web yet make web services and web APIs more efficient and standardized.

Creation of REST

Fielding's goal with REST was to capture the characteristics that made the web successful and formalize those characteristics into an architectural style.

Before REST, there was a **lack of standards** in how to use HTTP for anything beyond simple document retrieval, which led to inefficient designs and implementations. Fielding identified several key principles underlying the web's success, such as statelessness, the use of resources identified by URLs, and the use of standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on these resources.
By formalizing these principles, Fielding aimed to guide the design of systems that could enjoy the same scalability, generality, and independence from the technologies being used that the web did.

The REST architectural style is defined by **six guiding constraints**, which, when applied as a whole, are intended to create systems that are performant, scalable, simple, modifiable, visible, portable, and reliable.

1. Client-server architecture

The client-server constraint separates concerns between the user interface and data storage, which improves the portability of the user interface across multiple platforms and scalability by simplifying the server components.

2. Stateless

Each client request to the server must contain all the information the server needs to understand and fulfil the request.

The server does not store any state about the client session on the server side. This makes the server more scalable by reducing its resource consumption and improves reliability by simplifying the server design.

3. Cacheable

As on the **World Wide Web, clients and intermediaries can cache responses**. Responses must, therefore, implicitly or explicitly label themselves as cacheable or not, to prevent clients from reusing stale or inappropriate data in response to further requests. Caching can eliminate some client-server interactions, further improving scalability and performance.

4. Uniform Interface

The uniform interface constraint simplifies the architecture, which makes the overall system easier to understand and interact with. It also decouples the architecture, which allows each part to evolve independently. The four guiding principles of the uniform interface are:

- **Resource Identification in Requests**: Individual resources are identified in requests, for example, using URIs in web-based REST systems.
- **Resource Manipulation through Representations**: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
- **Self-descriptive Messages**: Each message includes enough information to describe how to process the message.
- **Hypermedia as the Engine of Application State (HATEOAS)**: Clients interact with a RESTful service entirely through hypermedia provided dynamically by application servers – a concept known as HATEOAS. Clients make state transitions only through actions that are dynamically identified within hypermedia.

5. Layered System

A **client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way**. Intermediary servers can improve system scalability by enabling load-balancing and providing shared caches. Layers can also enforce security policies.

6. Code on Demand (optional)

Servers can temporarily extend or customize the functionality of a client by transferring executable code. For example, compiled components such as Java applets, or client-side scripts such as JavaScript, can be provided to the client. This is the only optional constraint of REST.

Resource

In REST (Representational State Transfer), the notion of "**resources**" is fundamental and central to its architectural approach. **Resources represent the key abstraction of information and data in RESTful designs.**

A resource is **any piece of information** that can be named and is considered significant enough to be referenced as a whole. In practical terms, a resource can be a document or image, a temporal service (e.g., "today's weather in Boston"), a collection of other resources, a non-virtual object (e.g., a person), and so on.
Each resource is identified by a Uniform Resource Identifier (URI), which is used to uniquely address that resource. The URI is a critical component in RESTful designs because it provides a global addressing space for resource and service discovery.

Characteristics

- **Addressability**: Every resource is uniquely addressable using a URI.
- **State and Representation**: While a resource represents a conceptual entity, its state at any given moment can be conveyed via representations. A representation is a concrete, transferable encapsulation of the state of a resource, such as an HTML document, a JSON object, or an XML file.
- **Manipulation through Representations**: Clients interact with resources by exchanging representations. For instance, a client may retrieve a resource's current state (via a GET request), modify that state (via a PUT or POST request), or delete the resource (via a DELETE request), all through the exchange of representations.

HTTP != REST

REST does not specify the protocol that should be used for transporting the data.

Attribution to REST by Default: The reason HTTP is often attributed to REST by default is largely because of the natural compatibility between REST principles and HTTP features. Many of the key concepts of RESTful design map directly to features available in HTTP, making it a convenient choice for implementing RESTful services. However, this association has led to the misconception that REST is inherently tied to, or dependent on, HTTP.

Another important thing associated with REST are resource methods to be used to perform the desired transition. A large number of people wrongly relate resource methods to HTTP GET/PUT/POST/DELETE methods.

Roy Fielding has never mentioned any recommendation around which method to be used in which condition. All he emphasizes is that it should be uniform interface. If you decide HTTP POST will be used for updating a resource – rather than most people recommend HTTP PUT – it's alright and application interface will be RESTful.

ASP.NET

ASP.NET is a free, open-source web framework developed by Microsoft that allows developers to build dynamic web sites, web applications, and web services. It is part of the .NET framework.

Key Features

- **Model-View-Controller (MVC)**: ASP.NET includes support for the MVC pattern, enabling clean separation of concerns, making web applications easier to manage and test.
- **Web API**: ASP.NET facilitates the creation of RESTful services with its Web API framework, allowing for the development of back-end services for web and mobile applications.
- **SignalR**: It supports real-time web functionality, enabling server-side code to push content to clients instantly as it becomes available.
- **Razor Pages**: A simpler syntax for combining server code with HTML to produce dynamic web content.
- **Entity Framework**: ASP.NET integrates well with Entity Framework, an ORM that simplifies data access by allowing developers to work with data as strongly typed objects.

Program.cs - builder pattern

```
var builder = WebApplication.CreateBuilder(args);
```

Initiates and configures the construction of a web application, taking arguments passed to the application and using them for configuration.

```
builder.Services.AddEndpointsApiExplorer();
```

Adds services required for exploring the API through Swagger/OpenAPI, enabling the generation of API documentation.

```
builder.Services.AddSwaggerGen();
```

Adds a Swagger generator, which is used to create OpenAPI objects describing the API. This facilitates the automatic creation of API documentation.

```
builder.Services.AddControllers();
```

Registers the services required for MVC (Model-View-Controller) controllers, enabling the handling of HTTP requests by the controllers.

```
var app = builder.Build();
```

Completes the configuration process and builds the web application. In this case, the Builder pattern is used.

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

If the application is running in a development environment, it enables Swagger and the Swagger UI, which facilitates testing and documenting the API.

```
app.UseHttpsRedirection();
```

Enables redirection from HTTP to HTTPS, increasing the security of the application.

```
app.MapControllers();
```

Maps routes to MVC controllers, enabling the handling of HTTP requests by action methods in controllers.

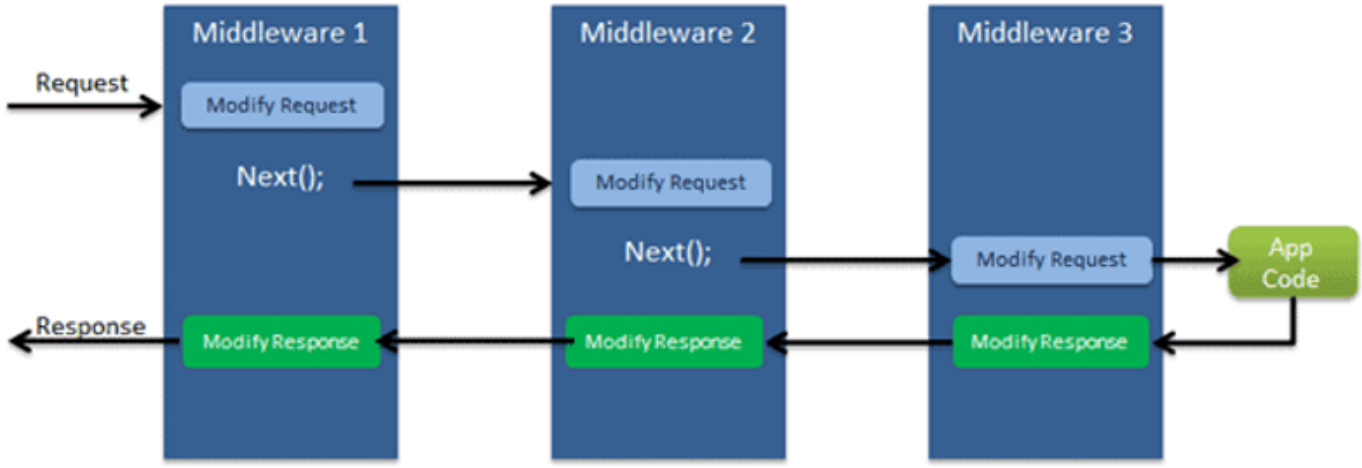
```
app.Run();
```

Launches the application, starting to listen for incoming connections.

ASP.NET - request pipeline

Middleware are software components that are assembled into an application pipeline to handle requests and responses. Each middleware component can perform operations before and after the next component in the pipeline. Middleware can:

- **Process a request and pass it to the next middleware in the sequence.**
- **Decide not to call the next middleware, effectively short-circuiting the pipeline.**
- **Perform some operations, then pass the request to the next middleware, and do more work when the request comes back in the response phase.**



Endpoint design guidelines

- **Use nouns instead of verbs in endpoint paths**
- **Name collections with plural nouns**
- **Nesting resources for hierarchical objects**
- **Handle errors gracefully and return standard error codes**
- **Allow filtering, sorting, and pagination**

```
+GET /companies/3/employees should get the list of all
employees from company 3

+GET /companies/3/employees/45 should get the details of
employee 45, which belongs to company 3

+DELETE /companies/3/employees/45 should delete employee 45,
which belongs to company 3

+POST /companies should create a new company and return the
details of the new company created
```

HTTP methods

- **GET method requests data from the resource and should not produce any side effect**. E.g /companies/3/employees returns list of all employees from company 3.
- **POST method requests the server to create a resource in the database**, mostly when a web form is submitted. E.g /companies/3/employees creates a new Employee of company 3.
- ==**POST is non-idempotent** ==which means multiple requests will have different effects.
- **PUT method requests the server to update resource** or create the resource, if it doesn't exist. E.g. /companies/3/employees/john will request the server to update, or create if doesn't exist, the john resource in employees collection under company 3.
- **PUT is idempotent** which means multiple requests will have the same effects.
- **DELETE method requests that the resources**, or its instance, should be removed from the database. E.g /companies/3/employees/john/ will request the server to delete john resource from the employees collection under the company 3.

Sorting, filtering, searching, pagination

=Sorting= In case, the client wants to get the sorted list of companies, the GET /companies endpoint should accept multiple sort params in the query. E.g GET /companies?sort=rank_asc would sort the companies by its rank in ascending order.

=Filtering= For filtering the dataset, we can pass various options through query params. E.g GET /companies?category=banking&location=india would filter the companies list data with the company category of Banking and where the location is India.

=Searching= When searching the company name in companies list the API endpoint should be GET /companies?search=Digital Mckinsey

=Pagination= When the dataset is too large, we divide the data set into smaller chunks, which helps in improving the performance and is easier to handle the response. Eg. GET /companies?page=23&pageSize=10 means get the list of companies on 23rd page.

HTTP status codes

- **1xx informational response** – the request was received, continuing proces
- **2xx successful** – the request was successfully received, understood, and accepted
- **3xx redirection** – further action needs to be taken in order to complete the request
- **4xx client error** – the request contains bad syntax or cannot be fulfilled
- **5xx server error** – the server failed to fulfil an apparently valid request
  https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

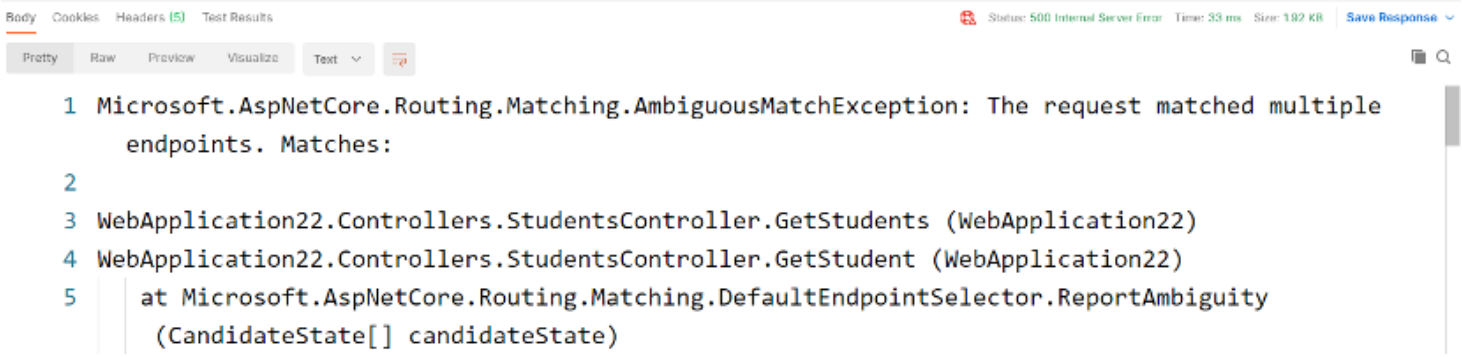
Different method for returning results

```
[HttpPost]
public IActionResult GetStudents(Student student)
{
    var list = new List<Student>();
    list.Add(new Student
    {
        FirstName = "Jan",
        LastName = "Kowalski"
    });


    //return StatusCode((int)HttpStatusCode.OK, "data");
    //return Created("uri", "data");
    //return ValidationProblem("message");
    //return Forbid("message");
    //return Challenge();
    //return Accepted("data or message");
    //return Unauthorized("Message");
    //return NotFound("Message");
    //return BadRequest("Error description");
    return Ok(list);
}
```

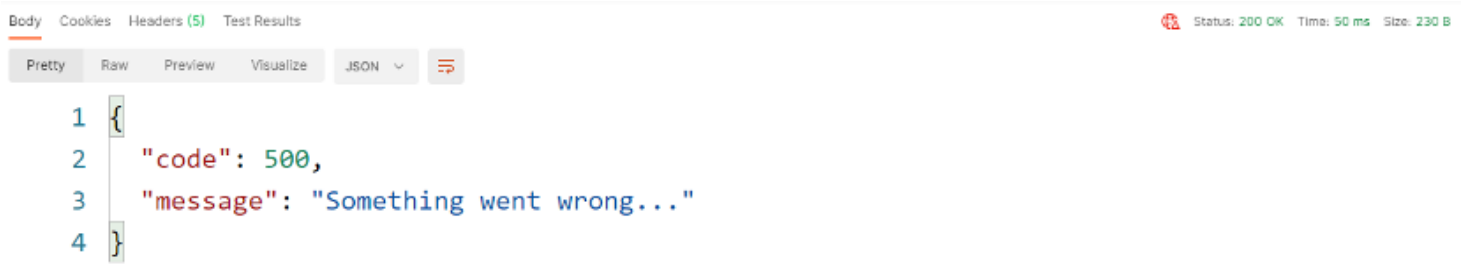
Multiple methods matching the same endpoint

```
[Route("api/students")]
[ApiController]
public class StudentsController : ControllerBase
{

    [HttpGet]
    public IActionResult GetStudent()
    {
        return Ok();
    }

    [HttpGet]
    public IActionResult GetStudents()
    {
        var list = new List<Student>();
        list.Add(new Student
        {
            FirstName = "Jan",
            LastName = "Kowalski"
        });

        return Ok(list);
    }
}
```

This will cause a runtime exception when routing the request.



Bad practice when returning the data



Content negotiation

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {

        services.AddControllers().AddXmlSerializerFormatters();
        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo { Title = "WebApplication22", Version = "v1" });
        });
    }
```

Attribute routing

Attribute routing in C# ASP.NET Core allows you to specify routes directly on controllers and actions by using attributes.

This approach gives you more control over the URIs in your web application. It enables you to define the **routes by decorating controllers and methods with route attributes, making the routes closely associated with their actions**. This can lead to more readable and organized code, especially for APIs with complex routing schemes.

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    [HttpGet]
    public IActionResult GetAllProducts()
    {
        // Implementation
    }

    [HttpGet("{id}")]
    public IActionResult GetProductById(int id)
    {
        // Implementation
    }
}
```

ASP.NET - minimal API

Minimal APIs in ASP.NET Core provide a streamlined approach to building HTTP APIs with less boilerplate code compared to traditional MVC controllers.

Introduced in .NET 6, minimal APIs are designed to make it easier and quicker to create simple and focused microservices or web services by reducing the complexity and amount of code required to set up endpoints.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/greet", () => "Hello, World!");

app.Run();
```

Passing data to web API

URL Segment

URL segments are parts of the URL path that represent a specific resource or hierarchy in your API. They are commonly used for identifying resources.

- **Use Case**: Ideal for RESTful operations where you're targeting a specific resource. For example, accessing, updating, or deleting a particular user in a system.
- **Example**: `GET /api/users/123` retrieves the user with ID 123.

```
app.MapGet("/api/users/{id}", (int id) => $"User ID: {id}");
```

Request body

The request body is used to send data to the server. It is primarily used with POST and PUT requests to create or update resources, respectively.

- **Use Case**: Best for operations that need to send complex data or large amounts of information that cannot be easily or securely passed via URL segments or query strings. It's commonly used in POST and PUT requests to create or update resources.
- **Example**: Using `POST /api/users` with a JSON body containing user details to create a new user.

```
app.MapPost("/api/users", (User user) => $"Create user:
{user.Name}");
```

Query string

Query strings are appended to the URL and provide a flexible way to pass optional or filtering data to the server. They start with a ? and can include multiple key-value pairs separated by &.

- **Use Case**: Suited for scenarios where you need to filter resources or pass optional parameters that don't pertain to a specific resource's identity. They are commonly used with GET requests.
- **Example**: `GET /api/users?age=25` might retrieve all users who are 25 years old.

```
app.MapGet("/api/users", (HttpContext context) => {
    var age = context.Request.Query["age"];
    return $"Users with age: {age}";
});
```

When designing RESTful APIs:

- **URL Segments** are used for identifying specific resources.
- **Request Body** is used for passing complex data for creating or updating resources.
- **Query Strings** are used for filtering lists/manipulating the resource.