**Json Web Token**

JWT, or JSON Web Token, is an open standard used to share security information between two parties — a client and a server. Each JWT contains encoded JSON objects, including a set of claims. JWTs are signed using a cryptographic algorithm to ensure that the claims cannot be altered after the token is issued.

private string GenerateToken(string username)

{

var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_configuration["Jwt:Key"]));

var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

var claims = new[]

{

new Claim(ClaimTypes.NameIdentifier,username),

new Claim(ClaimTypes.Role,"User")

};

var token = new JwtSecurityToken(\_configuration["Jwt:Issuer"],

\_configuration["Jwt:Audience"],

claims,

expires: DateTime.Now.AddMinutes(15),

signingCredentials: credentials);

return new JwtSecurityTokenHandler().WriteToken(token);

}

**How JWT Works**

JWTs differ from other web tokens in that they contain a set of claims. Claims are used to transmit information between two parties.

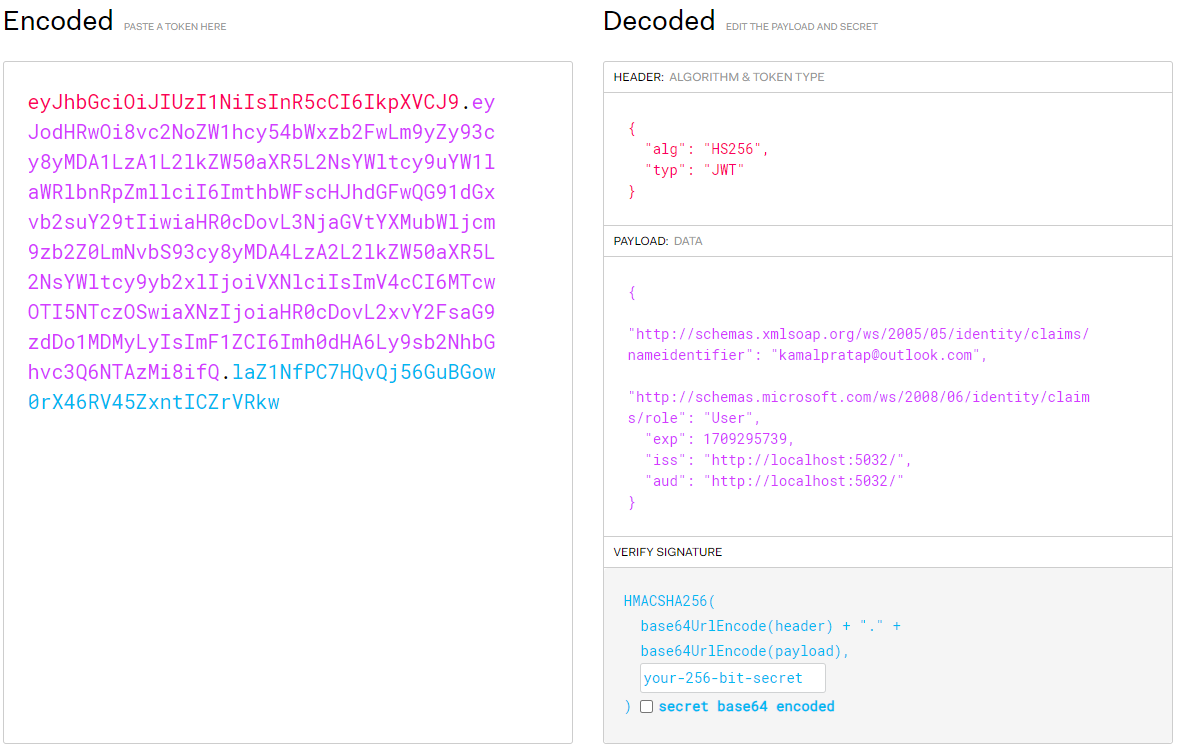
What these claims are depends on the use case at hand. For example, a claim may assert who issued the token, how long it is valid for, or what permissions the client has been granted.

A JWT is a string made up of three parts, separated by dots (.), and serialized using base64. In the most common serialization format, compact serialization, the JWT looks something like this:

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZW1hcy54bWxzb2FwLm9yZy93cy8yMDA1LzA1L2lkZW50aXR5L2NsYWltcy9uYW1laWRlbnRpZmllciI6ImthbWFscHJhdGFwQG91dGxvb2suY29tIiwiaHR0cDovL3NjaGVtYXMubWljcm9zb2Z0LmNvbS93cy8yMDA4LzA2L2lkZW50aXR5L2NsYWltcy9yb2xlIjoiVXNlciIsImV4cCI6MTcwOTI5NTczOSwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDMyLyIsImF1ZCI6Imh0dHA6Ly9sb2NhbGhvc3Q6NTAzMi8ifQ.laZ1NfPC7HQvQj56GuBGow0rX46RV45ZxntICZrVRkw**

Once decoded, you will get two JSON strings:

* The header and the payload.
* The signature.



The **header** typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

The second part of the token is the **payload**, which contains the claims. Claims are statements about an entity (typically, the user) and additional data.

To create the **signature** part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. The signature ensures that the token hasn’t been altered.

**HMACSHA256(**

**base64UrlEncode(header) + "." +**

**base64UrlEncode(payload),**

**secret)**

**How do JSON Web Tokens work?**

In **authentication**, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema. The content of the header should look like the following:

**Authorization: Bearer <token>**

**Why Use JWT?**

In short, JWTs are used as a secure way to authenticate users and share information.

Typically, a private key, or secret, is used by the issuer to sign the JWT. The receiver of the JWT will verify the signature to ensure that the token hasn’t been altered after it was signed by the issuer. It is difficult for unauthenticated sources to guess the signing key and attempt to change the claims within the JWT.

**Below filter user to authorize the API in asp.Net Core**

[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]

##### **What is Web API?**

API stands for Application Programming Interface. Web API is a concept (not a technology) that works on the HTTP Protocol and is used to extend the functionality of an application.

A Web API, or Web Application Programming Interface, is a set of rules and protocols that allows different software applications to communicate with each other over the internet or a network. It enables the exchange of data and functionality between various systems, using HTTP (Hypertext Transfer Protocol) as the communication protocol.

##### **What is ASP.NET Core Web API?**

The term API stands for Application Programming Interface. ASP.NET Core Web API is a framework provided by Microsoft that makes it easy to build Web APIs, i.e., HTTP-based services. The ASP.NET Core Web API is ideal for building Restful services on top of the .NET Platform, commonly known as the .NET Core Platform. These Web API services can then be consumed by a variety of clients, such as

1. Browsers
2. Mobile applications
3. Desktop applications
4. IOTs, etc.

##### **What is Rest?**

REST stands for **Representational State Transfer**. This is an architectural pattern used for exchanging data over a distributed environment. At rest, there is something called a Client and a Server, and the data will be exchanged between the client and server over a distributed environment.

A Distributed Environment means the client can be on any platform like Java, .NET, PHP, etc.; the server can also be on any platform like Java, .NET, PHP, etc. The REST architectural pattern treats each **service as a resource,** and a client can access those using resources by HTTP Protocol methods such as GET, POST, PUT, PATCH, and DELETE.

##### **What are the Differences Between REST and SOAP Services?**

Let us discuss the difference between the REST and SOAP service:

* SOAP stands for **Simple Object Access Protocol**, whereas REST stands for **Representational State Transfer**.
* SOAP is an XML-based protocol, whereas REST is not a protocol. Rather, it is an architectural pattern, i.e., resource-based architecture.
* SOAP has stateless and state-full implementation specifications, whereas REST is completely stateless.
* SOAP enforces the message format XML, whereas REST does not enforce the message format XML or JSON.
* The SOAP message consists of an envelope with SOAP headers and a body to store the information we want to send. In contrast, REST uses the HTTP build-in headers (with various media types) to store the information and uses HTTP Methods such as GET, POST, PUT, PATCH, and DELETE to perform CRUD operations.
* SOAP is operation-based, where services are exposed as operations, while REST is resource-based, where services are exposed as resources identified by URLs.
* REST is more flexible and offers better performance due to its stateless nature and support for a broader range of data formats. SOAP, with its rigid structure and XML format, is slower. SOAP performance is slow compared to REST.
* REST is generally considered easier to work with and is more commonly used in modern web service applications, especially for public APIs.

##### **What is HTTP?**

HTTP stands for **Hypertext Transfer Protocol**. It is the foundation for exchanging information between a web server and a client on the Web. HTTP defines the format for messages (requests and responses) between the Clients and Server. Here are the key characteristics of HTTP.

#### **HTTP Request Components**

**Request Headers:** Each HTTP Request can contain one or more Request Headers. The Request Header will be in the form of key-value pairs that provide additional information about the request. Some common headers include:

* **Host**: Specifies the domain name of the server.
* **User-Agent:** Identifies the client software initiating the request (e.g., the browser or application).
* **Accept:**  Tells the server what content types the client can handle.
* **Content-Type:**  When the request includes a body (like a POST, PUT, or PATCH request), this header indicates the media type of the body.
* **Authorization:** Contains credentials for authentication purposes.
* **Cookie:** Includes any cookies that the client has for this domain. This is used for state management.
* **Cache-Control:** Directives for caching mechanisms in both requests and responses.

**Request Body (Optional):** The body of an HTTP request is optional and is used when sending additional data to the server, like in POST, PUT, or PATCH requests. It could contain data from a form submission, file uploads, or JSON/XML data in the case of API requests.

#### **HTTP Response Components:**

Whatever the client receives from the web server is called HTTP Response. The HTTP response contains the following components.

* **HTTP Status Code:** It must have a Status Code indicating the status of the HTTP Request. 200 Indicates successful, 500 indicates internal server error, 404 indicates resource not found, etc.
* **Response Headers:** It can have one or more response headers.
* **Data:**Response can have data, i.e., return to the client.

**Routing in Asp.net Core**

Routing in ASP.NET Core Web API application is the process of mapping the incoming HTTP Request (URL) to a particular resource, i.e., controller action method. Here is an overview of How Routing works in ASP.NET Core Applications:

[Route("api/[controller]")]

##### **Configuring the Routing Middleware in ASP.NET Core**

To enable Routing in ASP.NET Core, we must add the following two middleware components to the HTTP Request Processing Pipeline.

**app.UseRouting();** This middleware enables routing capabilities in your ASP.NET Core application. It is responsible for matching incoming HTTP requests to routes that have been defined in your application.

**app.MapControllers();** This extension method is used to map attribute-routed controllers. It essentially tells the application to look for controllers in your project and creates routes for them based on the attributes you’ve defined (like [Route], [HttpGet], etc.). This is typically used when you have an API-centric application with controllers handling various HTTP requests.

##### **Adding Attribute Routing in ASP.NET Core Web Application**

Now, let us add two action methods within the EmployeeController class. Now, please don’t concentrate on the return type and the data that we are returning from the action method; rather, concentrate on the Routing concept.

We want to invoke the GetAllEmployees method with the URL **/Emp/All** and the GetEmployeeById method with the URL **/Emp/ById/102**.

To achieve this, we need to use the Route Attribute and decorate the action GetAllEmployees and GetEmployeeById method as **[Route(“Emp/All”)]**  and **[Route(“Emp/ById/{Id}”)]** respectively.

So, modify the EmployeeController class as shown below.

namespace RoutingInASPNETCoreWebAPI.Controllers

{

[ApiController]

public class EmployeeController : ControllerBase

{

[Route("Emp/All")]

[HttpGet]

public string GetAllEmployees()

{

return "Response from GetAllEmployees Method";

}

[Route("Emp/ById/{Id}")]

[HttpGet]

public string GetEmployeeById(int Id)

{

return "Response from GetEmployeeById Method Id: {Id}";

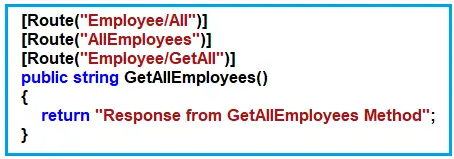
}

}

}

## ****How to set up Multiple URLs for a Single Resource in ASP.NET Core Web API****

We want to access the above resource with three URLs: **Employee/All**, **AllEmployees**, and **Employee/GetAll.**How do we do this? If this is your requirement, you need to decorate the GetAllEmployees action method with three different Route Attributes, as shown in the image below.



##### **ASP.NET Core Web API Attribute Routing with Route Constraints**

##### Route constraints in ASP.NET Core Web API are used to restrict the HTTP requests that can match a particular route. They enable the API to ensure that the parameters of a route are of a certain type, range, or pattern, which can be essential for the API’s logic and security. Implementing route constraints effectively can lead to more robust and error-free applications.

1. **Type: int, double, bool, float, datetime, etc.**
2. **Min: min(number)**
3. **Max: max(number)**
4. **Range: range(10. 15)**
5. **Alpha: alpha**
6. **MinLength: minlength(5)**
7. **MaxLength: maxlength(10)**
8. **Length: length(10)**
9. **Required: required**
10. **Regex: regex(expression)**

Let us understand ASP.NET Core Web API Attribute Routing Route Constraints with Examples. Please modify the Employee Controller class as shown below.

using Microsoft.AspNetCore.Mvc;

namespace RoutingInASPNETCoreWebAPI.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

[Route("{EmployeeId}")]

[HttpGet]

public string GetEmployeeDetails(int EmployeeId)

{

return $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

}

[Route("{EmployeeName}")]

[HttpGet]

public string GetEmployeeDetails(string EmployeeName)

{

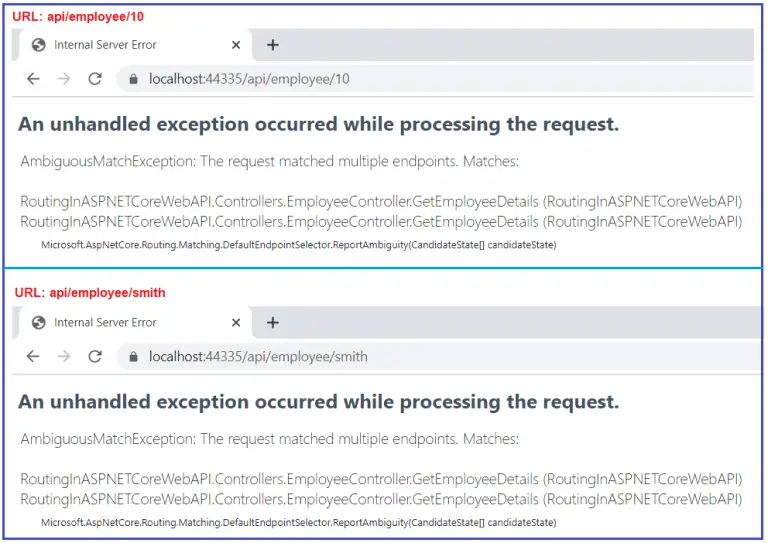
return $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

}

}

}

With the above changes in place, now run the application and navigate to the URL **api/employee/10** and **api/employee/smith**, and in both cases, you will get the following error.



Use below route constraints to resolve the issue.

**[Route(“{EmployeeId:int}”)]**

using Microsoft.AspNetCore.Mvc;

namespace RoutingInASPNETCoreWebAPI.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class EmployeeController : ControllerBase

{

[Route("{EmployeeId:int}")]

[HttpGet]

public string GetEmployeeDetails(int EmployeeId)

{

return $"Response from GetEmployeeDetails Method, EmployeeId : {EmployeeId}";

}

[Route("{EmployeeName}")]

[HttpGet]

public string GetEmployeeDetails(string EmployeeName)

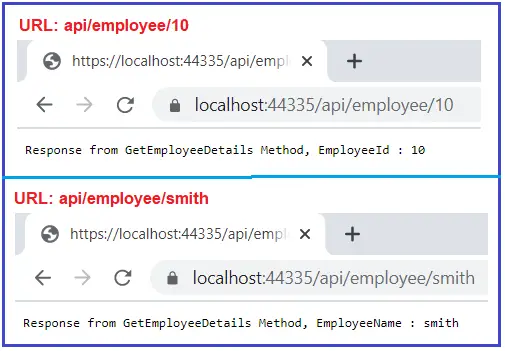
{

return $"Response from GetEmployeeDetails Method, EmployeeName : {EmployeeName}";

}

}

}



##### **Model Binding Techniques in ASP.NET Core Web API**

Understanding the various model binding techniques can greatly enhance your ability to process incoming requests efficiently. Model Binding in ASP.NET Core Web API can extract data from various parts of an HTTP request, including:

* Form Data
* Query Strings
* Route Data
* HTTP Headers
* Request Body

##### **FromForm Example in ASP.NET Core Web API**

##### Model:

namespace ModelBinding.Models

{

public class UserModel

{

public string Name { get; set; }

public string Email { get; set; }

// Add other properties as needed

}

}

Controller:

using Microsoft.AspNetCore.Mvc;

using ModelBinding.Models;

namespace ModelBinding.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class UserController : ControllerBase

{

[HttpPost]

public IActionResult CreateUser(**[FromForm]** UserModel user)

{

// Handle the user data, e.g., save it to a database

var response = new

{

Success = true,

Message = $"User {user.Name} created successfully!",

Code = StatusCodes.Status200OK

};

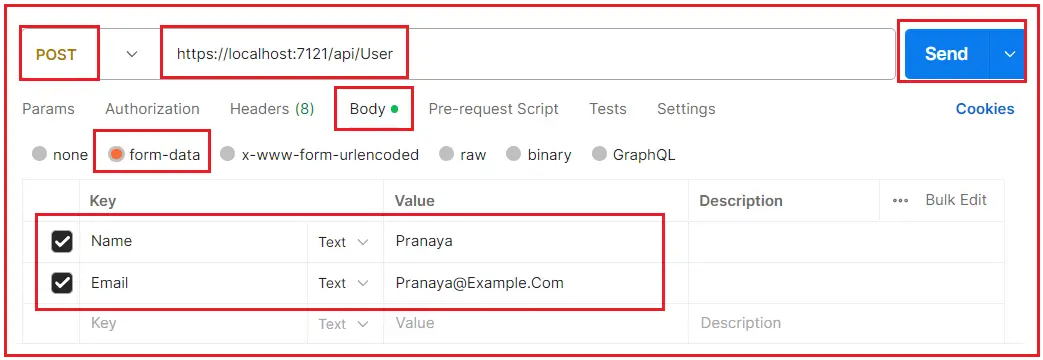
return Ok(response);

}

}

}

Testing API:





##### **Model Binding using FromQuery in ASP.NET Core Web API**

Model:

namespace ModelBinding.Models

{

public class UserModel

{

public int Id { get; set; }

public string Name { get; set; }

public string Department { get; set; }

public string Gender { get; set; }

public int Salary { get; set; }

}

}

Controller:

using Microsoft.AspNetCore.Mvc;

using ModelBinding.Models;

namespace ModelBinding.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class UserController : ControllerBase

{

private static List<UserModel> Users = new List<UserModel>

{

new UserModel { Id = 1, Name = "Rakesh", Department = "IT", Gender = "Male", Salary = 1000 },

new UserModel { Id = 2, Name = "Priyanka", Department = "IT", Gender = "Female", Salary = 2000 },

new UserModel { Id = 3, Name = "Suresh", Department = "HR", Gender = "Male", Salary = 3000 },

new UserModel { Id = 4, Name = "Hina", Department = "HR", Gender = "Female", Salary = 4000 },

new UserModel { Id = 5, Name = "Pranaya", Department = "HR", Gender = "Male", Salary = 35000 },

new UserModel { Id = 6, Name = "Pooja", Department = "IT", Gender = "Female", Salary = 2500 },

};

[HttpGet]

public IActionResult GetProducts(**[FromQuery]** string Department)

{

// Implementation to retrieve employees based on the Department

var FilteredUsers = Users.Where(emp => emp.Department.Equals(Department, StringComparison.OrdinalIgnoreCase)).ToList();

if (FilteredUsers.Count > 0)

{

return Ok(FilteredUsers);

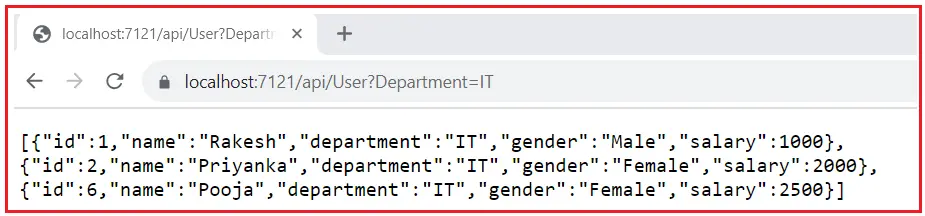
}

return NotFound($"No Users Found with Department: {Department}");

}

}

}



##### **FromRoute in ASP.NET Core Web API**

Model:

namespace ModelBinding.Models

{

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public int Quantity { get; set; }

public string Categogy { get; set; }

public int Price { get; set; }

}

}

Controller:

using Microsoft.AspNetCore.Mvc;

using ModelBinding.Models;

namespace ModelBinding.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

private static List<Product> Products = new List<Product>

{

new Product { Id = 1, Name = "Laptop", Categogy = "Electronics", Price = 1000, Quantity = 10 },

new Product { Id = 2, Name = "Desktop", Categogy = "Electronics", Price = 2000, Quantity = 20 },

new Product { Id = 3, Name = "Mobile", Categogy = "Electronics", Price = 3000, Quantity = 30 },

new Product { Id = 4, Name = "Casual Shirts", Categogy = "Apparel", Price = 500, Quantity = 10 },

new Product { Id = 5, Name = "Formal Shirts", Categogy = "Apparel", Price = 600, Quantity = 30 },

new Product { Id = 6, Name = "Jackets & Coats", Categogy = "Apparel", Price = 700, Quantity = 20 },

};

[HttpGet("{id}")]

public IActionResult GetProductById(**[FromRoute]** int id)

{

// Logic to retrieve the user by ID

var product = Products.FirstOrDefault(prd => prd.Id == id);

if (product != null)

{

return Ok(product);

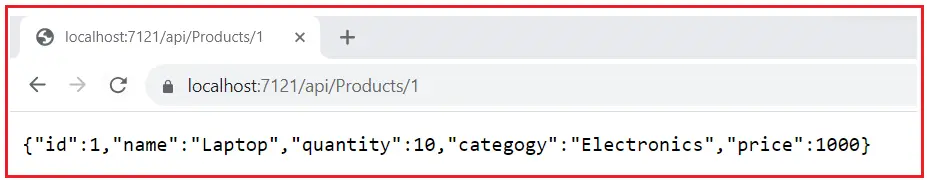
}

return NotFound($"No Product Found with Product Id: {id}");

}

}

}



## ****Model Binding Using FromHeader in ASP.NET Core Web API****

using Microsoft.AspNetCore.Mvc;

namespace ModelBinding.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class UserController : ControllerBase

{

[HttpGet]

public IActionResult GetResource([FromHeader] string Authorization)

{

// Implementation

if (Authorization == null)

{

return BadRequest("Authorization Token is Missing");

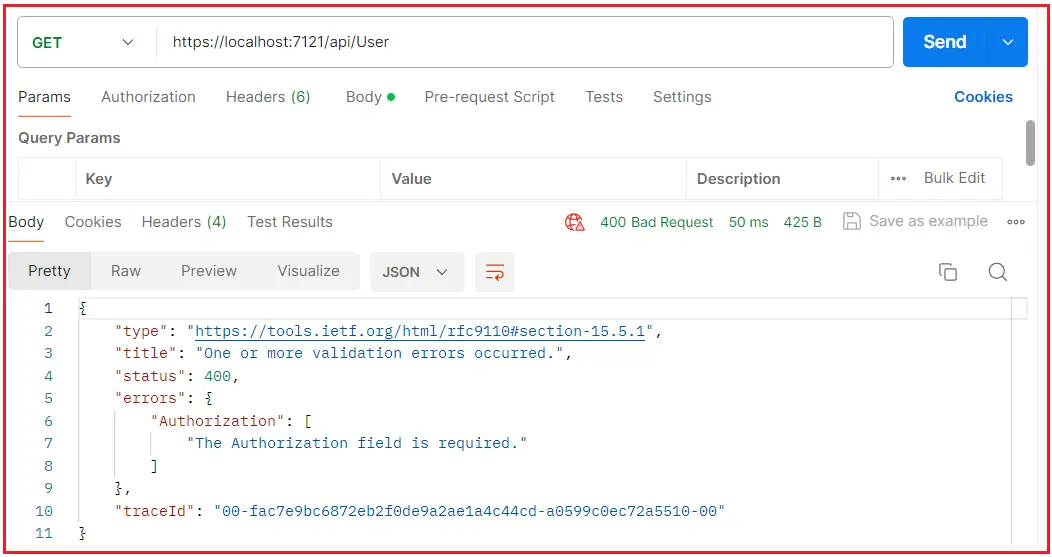
}

return Ok("Request Processed Successfully");

}

}

}



##### **Model Binding Example using FromBody**

Model:

namespace ModelBinding.Models

{

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public int Quantity { get; set; }

public int Price { get; set; }

}

}

Controller:

using Microsoft.AspNetCore.Mvc;

using ModelBinding.Models;

namespace ModelBinding.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

[HttpPost]

public IActionResult CreateProduct([FromBody] Product product)

{

// Add the product to the database or in-memory store

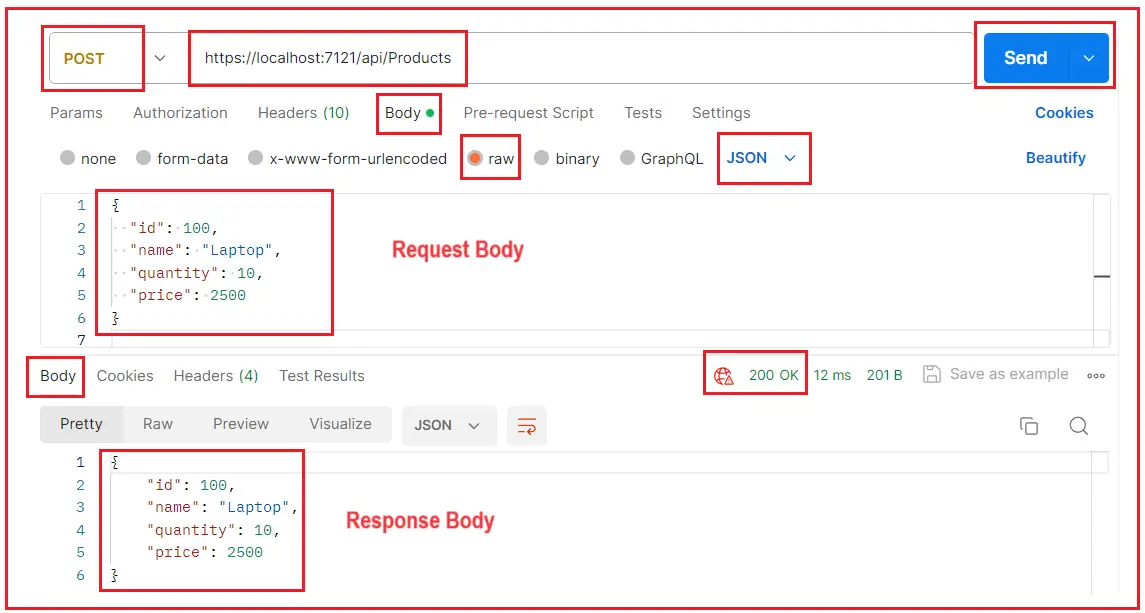
// For demonstration, let's return the product back

return Ok(product);

}

}

}



## ****How to Apply Binding Attributes to Model Properties in ASP.NET Core Web API****

using Microsoft.AspNetCore.Mvc;

namespace ModelBinding.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class BookController : ControllerBase

{

[HttpPost("{Id}")]

public IActionResult CreateBook([FromRoute] int Id, [FromQuery] string Author, [FromHeader] string Title)

{

// Your logic to Store the Data into the database

//Here, we are simply creating an Anonymous Object and returning the Book details

var response = new

{

BookId = Id,

BookTitle = Title,

AuthorName = Author

};

return Ok(response);

}

}

## }

## 

##### **Why Content Negotiation in Rest Services?**

We know that there are three pillars of the Restful Web Service and they are:

* The Resource
* The URL
* The Representation

The first two (i.e., the Resource and the URL) are very straightforward, but the last one (i.e., the Representation) is a little confusing. Representation is very important in the Modern Application. Why? Because, people are currently not only using desktop computers to browse web applications but are also using various types of devices (Tab, Mobile, etc.) to consume web applications. The important and interesting fact is that these devices expect the data in multiple formats.

For example, a few clients want the data in standard HTML, while some want it in a normal text format. Others may need the data in JSON format, and others want the data in XML format. This is where Content **Negotiation comes** into the picture.

**How Does the Content Negotiation Work?**

Content Negotiation primarily involves two HTTP headers: Accept and Content-Type.

**Accept Header:** The "Accept" header field can be used by **client** to specify **response** media types that are acceptable. A client can use the Accept header to indicate that it prefers to receive data in the JSON format (application/json) or XML format (application/xml).

**Content-Type Header:** Used by the server to specify the media type of the response content it sends back to the client. It tells the client in what format the response data is encoded, ensuring that the client knows how to parse and process the data received.

Model:

namespace ModelBinding.Models

{

public class Employee

{

public int Id { get; set; }

public string Name { get; set; }

public string Gender { get; set; }

public int Age { get; set; }

public string Department { get; set; }

}

}

Controller:

using Microsoft.AspNetCore.Mvc;

using ModelBinding.Models;

namespace ModelBinding.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class EmployeeController : ControllerBase

{

[HttpGet]

public ActionResult<List<Employee>> GetEmployees()

{

var listEmployees = new List<Employee>()

{

new Employee(){ Id = 1001, Name = "Anurag", Age = 28, Gender = "Male", Department = "IT" },

new Employee(){ Id = 1002, Name = "Pranaya", Age = 28, Gender = "Male", Department = "IT" },

};

return Ok(listEmployees);

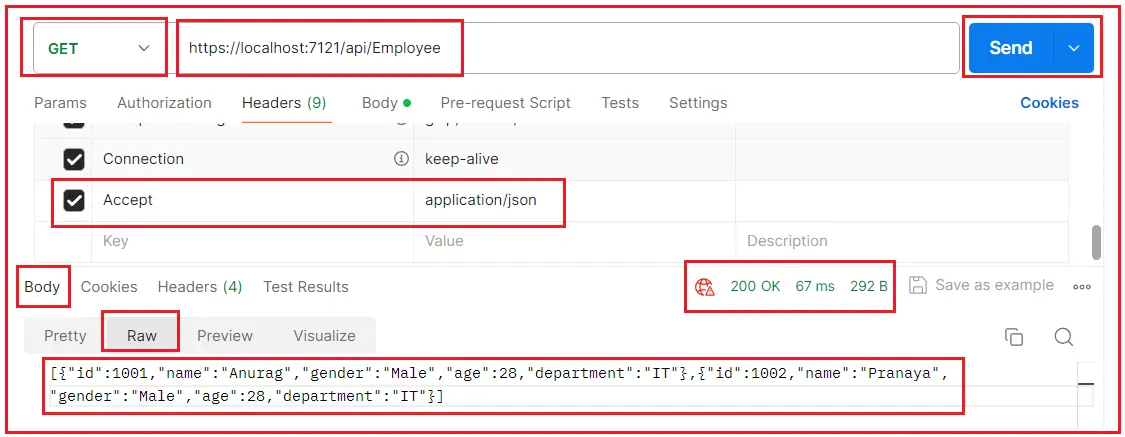
}

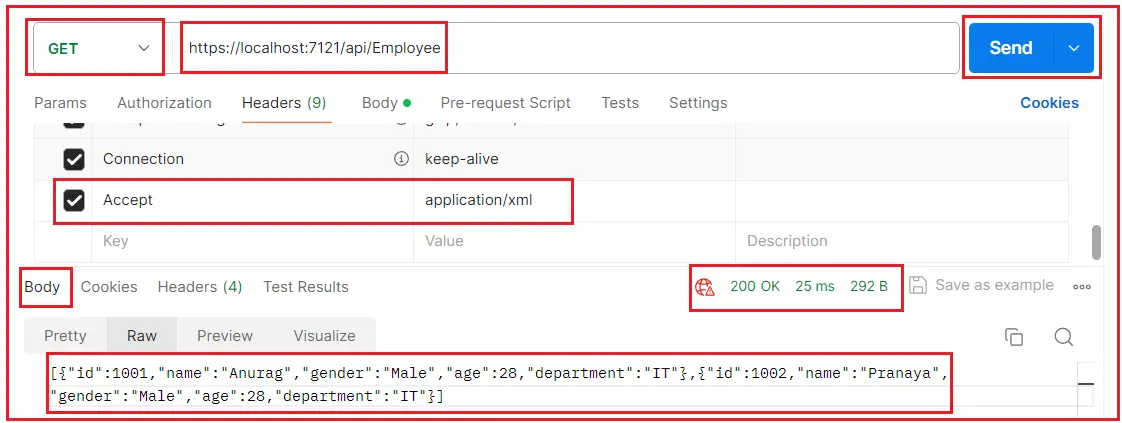
}

}

**Testing the API using Accept Header:**

Now, let us send an HTTP GET Request to the above endpoint using the **Accept** header and setting its value to **application/json**, telling the server to send the response in JSON format, as shown in the image below. Please change the Port number where your application is running:



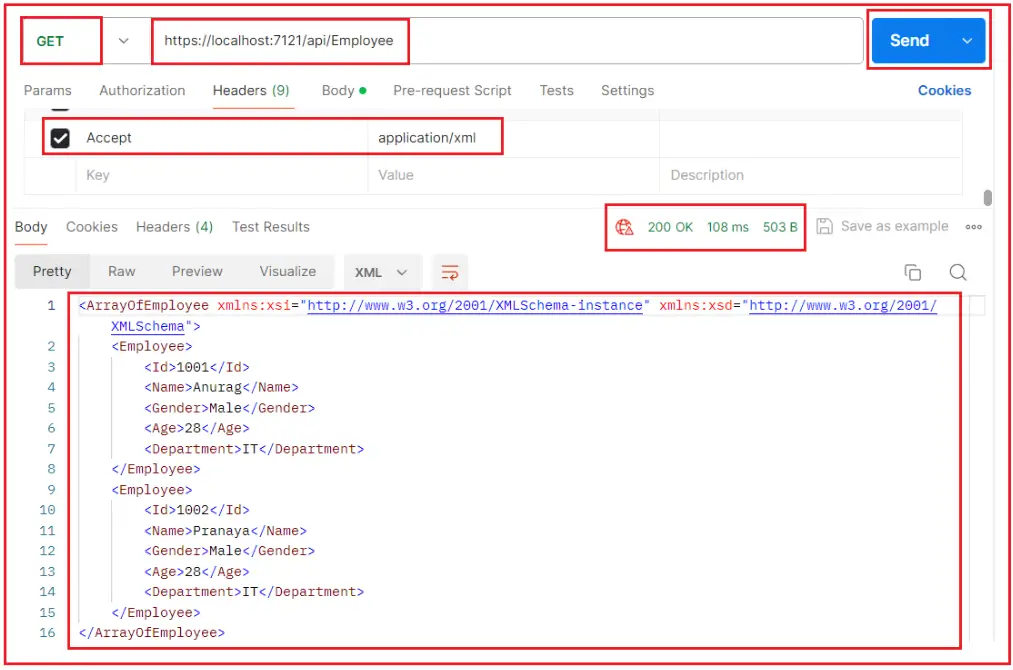


As you can see in the above image, even if we set the Accept header value to application/xml, we did not get the response in XML format. This is because we have not yet enabled the XML formatter in our application. The only Formatter enabled by default is JSON Formatter, and that JSON Formatter is used to send the response by default irrespective of the Accept header value.

**How do you enable the XML Formatter in ASP.NET Core Web API?**

We need to register the XML Formatter service to the dependency injection container, and we can do so in the Program class as follows:

builder.Services.AddControllers().AddXmlSerializerFormatters();



**What happens if we specify both application/json and application/xml in the Accept Header?**

While sending an HTTP Request, it is possible to set multiple values separated by a comma for the Accept header. In that case, the first value will take the priority. For example, suppose we set the Accept header value as **application/xml,application/json. In that case,** the server will give priority to XML and will return the data in XML Format.

**Middleware in ASP.NET Core Web API**

Middleware is a piece of code that is used in the HTTP Request Pipeline. An ASP.NET Core Web API Application can have n numbers of middleware. So, depending upon the requirement, we can configure n numbers of middleware in the application request processing pipeline.

**Middleware Examples**

**Routing**: If you want to implement Routing in your application, then you need to use Routing Middleware in the HTTP Request Processing pipeline.

**Authentication**: If you want to authenticate the user then you need to use Authentication Middleware.

**Authorize**: The Authorize Middleware is used to Authorize the users while accessing a specific resource.

**Log**: If you want to log request and response while processing, then you need Middleware.

**Exception Middleware:** You can also use Middleware to handle the exception globally.

**How to Configure Middleware Components in ASP.NET Core application?**

In the ASP.NET Core Web API application, the Middleware components are configured in program.cs class. The Program class is the class that is going to run when the application starts.

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

app.UseDeveloperExceptionPage();

}

app.UseCors("corsapp");

app.UseAuthentication();

app.UseAuthorization();

app.MapControllers();

##### **Creating Custom Middleware in ASP.NET Core**

While working with the real-time applications in ASP.NET Core Web API, it is a common requirement to create Custom Middleware Components. So, let us add a new class file to our project. It is this class file that is going to contain the logic.

namespace TutorialAPI.UserClasses

{

public class MyCustomMiddleware

{

private readonly RequestDelegate \_next;

public MyCustomMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

var startTime = DateTime.Now;

await \_next(context);

var endTime = DateTime.Now;

var elapsedTime = endTime - startTime;

var logMessage = $"{context.Request.Method} {context.Request.Path} {context.Response.StatusCode} {elapsedTime.TotalMilliseconds}ms";

Console.WriteLine(logMessage);

}

}

}

**Note:**While calling the next method from any custom middleware components, we need to pass the context object and that you can see in the above code.

Our Custom Middleware component is ready. Now we need to use it in our HTTP Request Processing pipeline. Now it is a one process to use this custom middleware component.

Registering the Custom Middleware in the HTTP Request Processing Pipeline

**app.UseMiddleware<MyCustomMiddleware>();**

**HTTP Status Codes in ASP.NET Core Web API:**

Here are some of the most frequently used HTTP status codes in ASP.NET Core Web API, along with their significance:

**Successful Responses**

* **200 OK** – The HTTP 200 OK success status response code indicates that the request has succeeded. This is often used for GET and POST requests that are processed successfully.
* **201 Created** – The request has succeeded, and a new resource has been created as a result. This is typically used in response to a POST request.
* **202 Accepted**– The 202 Status Code indicates that the request has been accepted for processing, but the processing has not been completed.
* **204 No Content** – The server successfully processed the request but is not returning any content. This is often used for DELETE requests.

**Redirection Messages**

* **301 Moved Permanently –** This response code indicates that the URI of the requested resource has been changed permanently. Future requests should use the new URI.
* **302 Found:** This response code indicates that the resource is temporarily under a different URI. As the redirection might be altered occasionally, the client should continue using the original URI for future requests.

**Client Error Responses**

* **400 Bad Request –** The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax).
* **401 Unauthorized** – Although the HTTP standard specifies “unauthorized”, semantically, this response means “unauthenticated”. That is, the client must authenticate itself to get the requested response.
* **403 Forbidden** – The client does not have access rights to the content; that is, it is unauthorized, so the server refuses to give the requested resource. Unlike 401, the client’s identity is known to the server.
* **404 Not Found** – The server can not find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid, but the resource itself does not exist.
* **405 Method Not Allowed**– The 405 Method Not Allowed response status code indicates that the server knows the request method but is not supported by the target resource. For example, we have one method which is a POST method in the server and we trying to access that method from the client using GET Verb, then, in that case, you will get a 405-status code.

**Server Error Responses**

* **500 Internal Server Error –** The server has encountered a situation it doesn’t know how to handle.
* **501 Not Implemented –** The server either does not recognize the request method or lacks the ability to fulfil the request.
* **503 Service Unavailable –** The server is not ready to handle the request. Common causes include a server that is down for maintenance or overloaded.
* **504 Gateway Timeout.**The 504 Gateway Timeout server error response code indicates that the server while acting as a gateway or proxy, did not get a response in time from the upstream server that is needed to complete the request.

##### **Microservices using ASP.NET Core**

##### In this growing fast-paced world, the amount of data and internet usage are proportionally increasing, and so more reliable and fast responding software systems are required, Unlike the older way of application development in Monolithic architecture which causes high maintenance cost, more downtime during upgrades made to existing monolithic architected software is not reliable. So, the Microservices Architecture of developing applications came into the picture.

Earlier software architecture build contains all business functionalities, Database calls, and UI designed in a single bundle. Like Asp.Net Webforms, MVC as a collection of single projects.

It has its disadvantages, the larger the application grows, the harder it is to quickly resolve the technical bugs/problems and to update the app with the new features. The Microservice architecture-based approach for building applications helps solve these real-time issues and provides more space for agile development methods and faster response from applications.

##### **What are Microservices?**

Microservices are the architectural approach to build applications from small to large scale applications. With this architectural approach, an application is broken down into the smallest components, independent of each other. Unlike Monolithic architecture, where all the functionalities are targeted to build into a single project/application, Microservices helps to separate functionalities to develop in a more modular way and all modules work together to accomplish the specific targeted tasks.

##### **Advantages of Microservices**

Microservices give development teams and testers a faster approach through distributed development. Provides the ability to develop multiple Microservices simultaneously. This means more developers working on the same app but different functional modules, at the same time, this results in a less deliverable time of application to the client. Below are some critical points you can observe positively in Microservices.

###### **Faster Development**

Since the application is designed into smaller modules development time is shortened, a microservices architecture supports a more agile development process. Multiple teams can work on each module/feature to develop and deliver separately.

###### **High scalability**

As user demand for some services in application grows, you can deploy those services across multiple servers, and infrastructures, to meet your needs. As much as the usage of services increases by end-users, easily the deployment and extending of Microservice can be handled by load balancing.

###### **Resilient & Independency**

The independent services, after properly developed and orchestrated to not impact one another. This means that one service failed to work the whole app won’t go down, unlike the monolithic application model.

###### **Easy of Deployment**

Because the microservice-based applications are more modular than traditional monolithic applications, the problems that came with application deployments and application downtimes are reduced.

###### **Accessibility for Development**

Since the larger app is broken down into smaller modules, developers can easily understand, update, and enhance the new changes, which results in fast development cycles, if planned with agile development methodologies.

###### **More open**

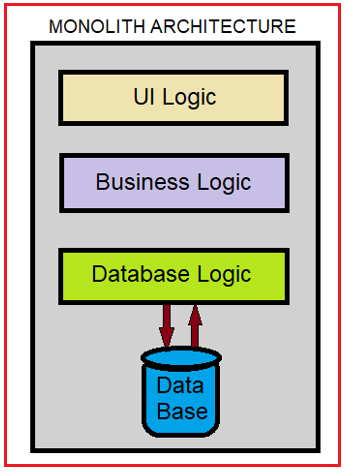
Microservices are not a particular language/technology-dependent, developers are free to choose any best language and technology. In our current article, we will discuss more the .Net technology stack.

##### **Monolith Architecture**

First, we will understand the traditional way of building applications called Monolith Architecture, which is the widely used architectural pattern while developing applications. Still, we can see these architecture applications exist in the real world, it is designed in a way that the entire application built is ultimately a single piece, no matter how much you try to de-couple them by using Patterns and tiers – 1/2/3.

All the services or business functionalities would be tightly coupled within the single Solution. The major thing to note is that while publishing the application, you would have to deploy them to a single server only.

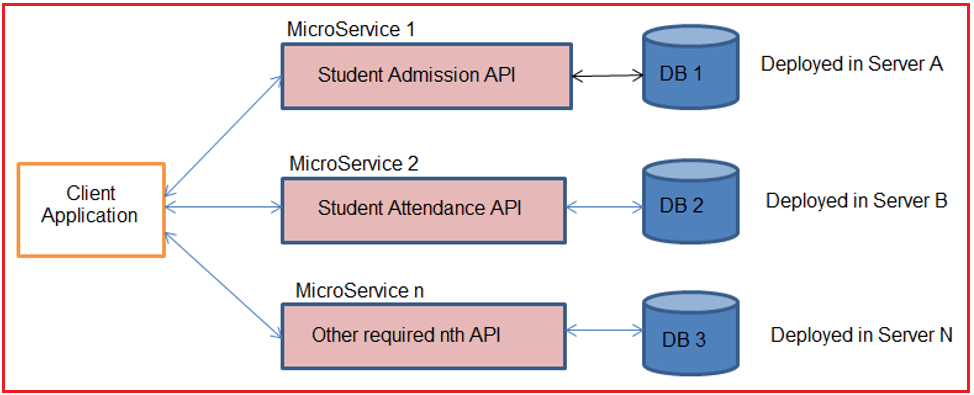
While it is still an effective way to build applications, there are some drawbacks associated with Monolith Architecture. Any small to midscale Applications would do just fine with this Architecture, but when you scale up further to extend the capabilities of existing software, you would have to make a few compromises as well as face whole application downtimes while deploying new versions/bug fixes.



From the above diagram, you can observe different logics or layers you want to integrate with your application are consolidated into a part of the single application itself. Thus, it is a big problem to maintain and add a new feature and deployment of the application in this architecture.

##### **Microservice Architecture**

As we mentioned using Microservice architecture the application is divided into various components or modules, with each module serving a particular purpose. And these components are called Microservices all to gather. These components are not dependent on the application itself. Each of these components is truly independent in all technical manners. Because of this robust separation, you can have separately dedicated Databases for each component i.e., Microservice as well as can deploy them to separate Hosts & Servers.



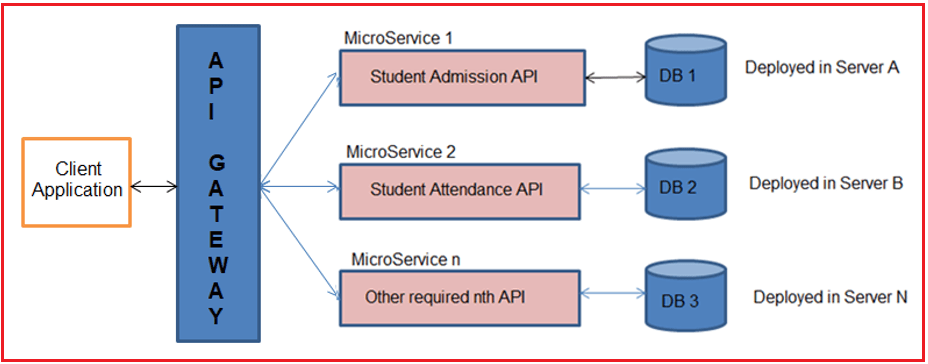
In the above picture, we can see the Client UI application either developed in .Net technology or Android or IOS or Angular, etc., will interact with our decoupled APIs called Microservices developed in .NET CORE technology, and they are deployed on multiple servers.

##### **Microservice Vs Monolith Architecture**

|  |  |
| --- | --- |
| **Monolithic** | **Microservices** |
| Dependent & compounded project development of User Interface, Business functional Components, and Database Logics. | Multiple Services are developed with a single purpose each. |
| Single Database due to complex coupled development | Dedicated Database for each microservice due to modularity. |
| Single programming Language dependency for the Backend (Business logic) | Ability to choose multiple technologies/languages for each microservice. |
| The solution has to be deployed into a single Server. Physical Separation of deployment and maintenance is complex. | Each of the Microservice can be deployed anywhere on the web with ease. |
| Different business functionalities technically called modules or components will be tightly coupled to the application itself. | Loosely Coupled Architecture because of the modular approach. |

##### **Understanding API Gateway (Ocelot Gateway with example)**

API Gateway is nothing but a middleware layer of directing incoming HTTP request calls from Client applications to specific Microservice without directly exposing the Microservice details to the Client and returning the responses generated from the respective Microservice.



Ocelot is an Open-Source API Gateway for the .NET/Core Platform which is officially supported by Microsoft. What it does is simple. It mimics masking multiple microservices existing behind that the client does not have to worry about the location of each and every Microservice.

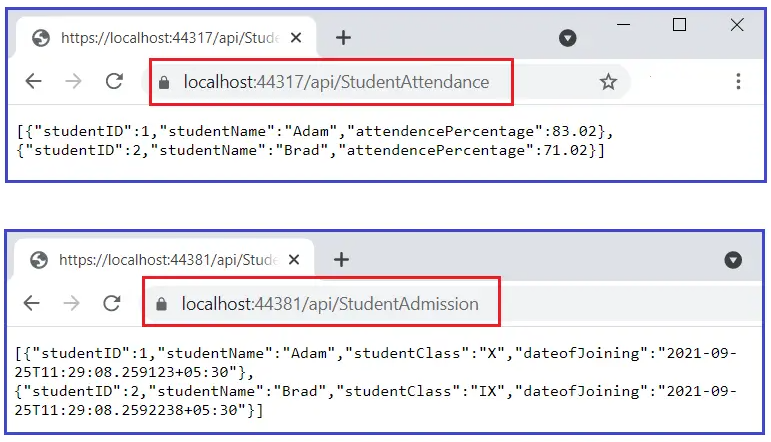
Ocelot is widely used by Microsoft and other tech giants as well for Microservice Management. The latest version of ocelot supports only .NET Core applications build on the 3.1 version and above.

##### **Features of API Gateway**

**API virtualization:**API Gateways acts as a single point entry for all the microservices configured, and avoids direct availability of microservices to clients, and hides versioning details of microservices.

**Serves as an additional layer of security microservices:**API gateways prevent malicious attacks by providing an additional layer of protection from attack vectors and hackers like SQL Injection, XML Parser exploits, and denial-of-service (DoS) attacks, and forged form data submissions.

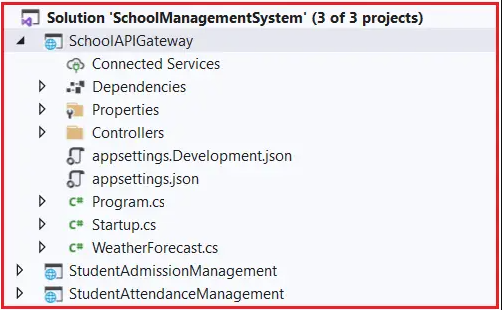
**Decreased microservice complexity**: Authorization techniques such as JWT and other concerns of development can constitute more time for the development of each microservices. An API gateway can handle these concerns on its own and removes the development burden from your API code.



They returned the details as expected. So, this is how microservices are developed, and our two microservices are available to serve the user requests over HTTP requests and responses. To test Microservices with the Postman application, please refer to the Testing section provided at the end of this learning path.

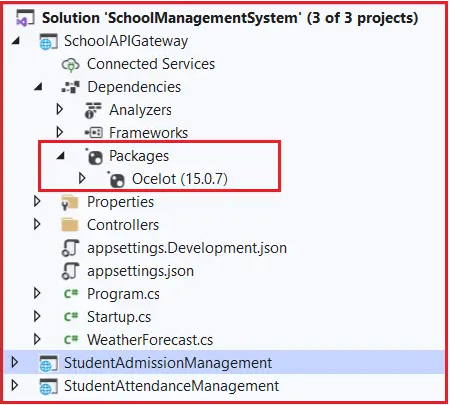
**Creating API Gateway Project using Ocelot**

Let’s create a separate ASP.NET Core Web API project named SchoolAPIGateway, as we created projects in the solution earlier. Once the project is created, it will look like the one below in the Solution Explorer window.



Installing Ocelot package

Let’s install the Ocelot Package. This makes the project behave as an API gateway. To do so, click on Tools => NuGet Package Manager => Package Manager Console option from the context menu. After successful installation, you can see the Ocelot package in the packages folder, as shown in the below image.



Configure API Gateway and Integrate Microservices.

let’s configure the project and define the Microservices endpoints.

Configure Ocelot as a middleware

using Ocelot.DependencyInjection;

using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

//Ocelet Configuration Start

builder.Configuration.AddJsonFile("ocelot.json", optional: false, reloadOnChange: true);

builder.Services.AddOcelot(builder.Configuration);

//Ocelet Configuration End

var app = builder.Build();

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

//Ocelet Configuration Start

app.MapControllers();

app.UseOcelot();

//Ocelet Configuration End

app.Run();

**Configuring Ocelot Routes in ASP.NET Core**

This is the most important part of configuring Gateway to reroute the incoming requests to a definite API service. In the Ocelot.json file, you have to configure the Upstream / Downstream routes for the API Gateways, which makes Ocelot understand the routes to redirect the requests to microservices. Let’s understand the following terms.

Create a new Ocelot.json file in the root directory of the SchoolAPIGateway Project. This file carries the routing configurations needed for Ocelot. To do so, add a JSON file with the name Ocelot.json into the root directory of the SchoolAPIGateway Project and then copy and paste the following code into it.

{

"GlobalConfiguration": {

"BaseUrl": "https://localhost:5194"

},

"Routes": [

{

"DownstreamPathTemplate": "/api/Account",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 5242

}

],

"UpstreamPathTemplate": "/gateway/Account",

"UpstreamHttpMethod": [ "Post" ]

},

{

"DownstreamPathTemplate": "/api/AdminLogin",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 5056

}

],

"UpstreamPathTemplate": "/gateway/AdminLogin",

"UpstreamHttpMethod": [ "Post" ]

},

{

"DownstreamPathTemplate": "/api/BlogDetails",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 5036

}

],

"UpstreamPathTemplate": "/gateway/BlogDetails",

"UpstreamHttpMethod": [ "Get" ]

}

]

}

**DownstreamPathTemplate** is the path of the actual endpoint in the Microservice to which the API Gateway redirects the incoming request from the client.

**DownstreamScheme** is the scheme used by Microservice, which is HTTPS.

**DownstreamHostAndPorts** defines the location of the Microservice, where we will add the host details and port number details.

**UpstreamPathTemplate** is the path directed to the Ocelot API Gateway exposed to send requests from the client.

**UpstreamHttpMethod** is the supported HTTP Method by the API Gateway, such as GET/PUT/POST/DELETE. Based on the Incoming HTTP Method, Ocelot forwards a similar HTTP method request to the microservice as well.

We have the URL for Student Admission Microservice, which we built earlier and tested in Browser

**https://localhost:44381/api/StudentAdmission ->** Downstream URL

**https://localhost:44317/api/StudentAttendance ->** Downstream URL

