**Ocelot APIGateway**

* An API Gateway is a server or service that acts as an entry point for a group of microservices or backend services. It serves as an intermediary layer between clients (which can be web or mobile applications) and the various microservices that make up a larger software application. API Gateways are an essential component of modern software architectures, especially in microservices-based systems.
* In simple term , when we are having multiple APIs running , it asks for multiple ports and it is very hectic to get data from each API using different host and ports . So by using Ocelot APIGateways, we can map all the port to an unified port and can fetch all the APIs by its names mentioned in ocelot.json file .

The features that I have implemented in this project are

1. Configuration
2. Routing
3. Authentication
4. Rate Limiting
5. Caching
6. Header Transformation
7. HTTP Method Transformation
8. Logging
9. Request Aggregation
10. Request ID
11. Quality Of Service
12. Tracing
13. Method Transformation
14. Service Discovery
15. Load Balancer
16. WebSockets

Now lets talk about each feature that I have implemented.

**1.Configuration:-**

* Configuration refers to the configuration settings or setup that you define to control how the API Gateway behaves. It's like a set of rules and instructions that tell the API Gateway how to route requests, handle security, and perform various tasks.
* To implement this, we need to create one ocelot.json file , in which we can add all properties related to Configuration.
* There are two sections in Config . Routes and GlobalConfiguration
* The Routes are the objects that tell Ocelot how to treat an upstream request.
* GlobalConfiguration is the unified port address.
* **Example:-**

**{**

"Routes": [

{

"DownstreamPathTemplate": "/WeatherForecast",

"DownstreamScheme": "https",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 7246

},

"UpstreamPathTemplate": "/api/Weather",

"UpstreamHttpMethod": [ "Get" ],

**],**

"GlobalConfiguration": {

"RequestIdKey": "OcRequestId",

"BaseUrl": "[https://localhost:5021](https://localhost:5021/)"

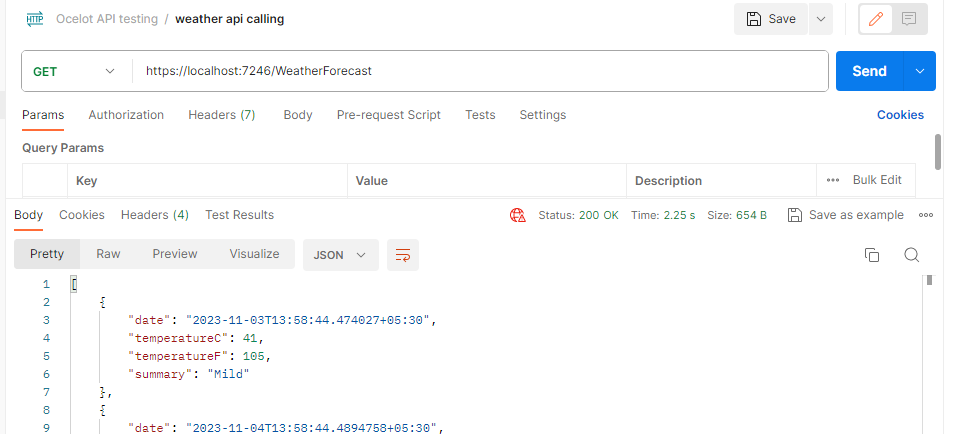
}

In the above example , DownstreamPathTemplate means the API Controller and method name . DownstreamScheme means which kind of request it is . DownstreamHostAndPorts contains the name of host and ports . BaseUrl in GlobalConfiguration contains the BaseUrl which is the unique port for all APIs. UpstreamPathTemplate is the new name of that API .

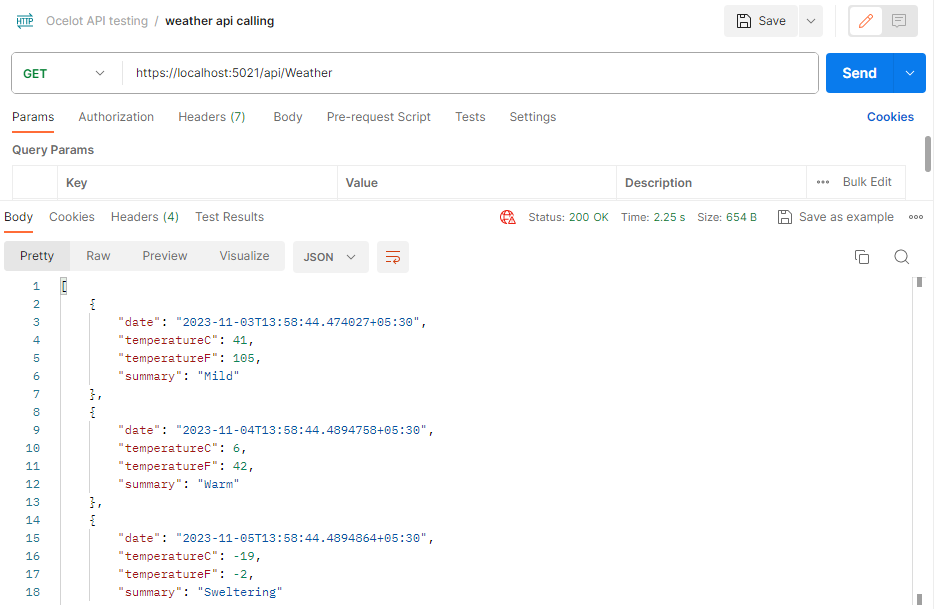
**2.Routing:-**

Routing is already explained in the above example. Below are some live examples.

**By using usual path.**



**Through APIGateway Path**

****

**3. Authentication:-**

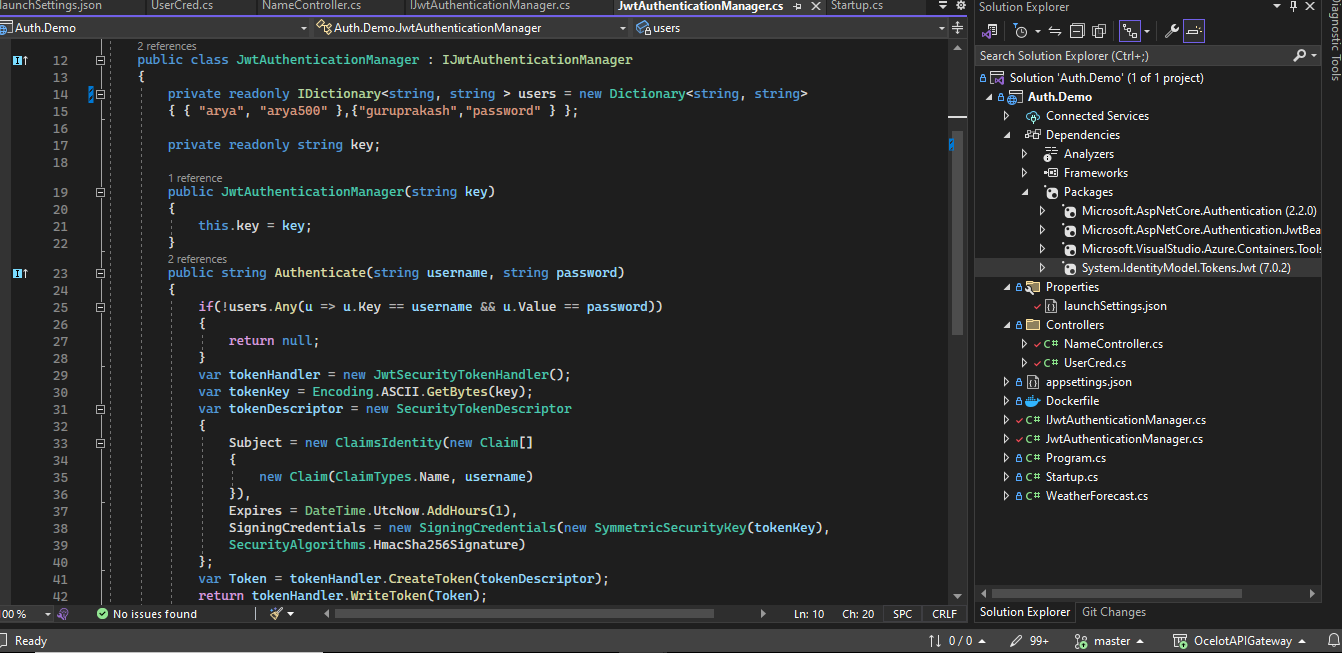
* Authentication in API gateways means verifying the identity of clients or users seeking access to APIs. It involves methods like API keys, JWT or OAuth2 and is essential for enforcing access control and security in a centralized manner.
* Here I used JWT Authentication method . For that I need some NuGet packages installed in my project . They are

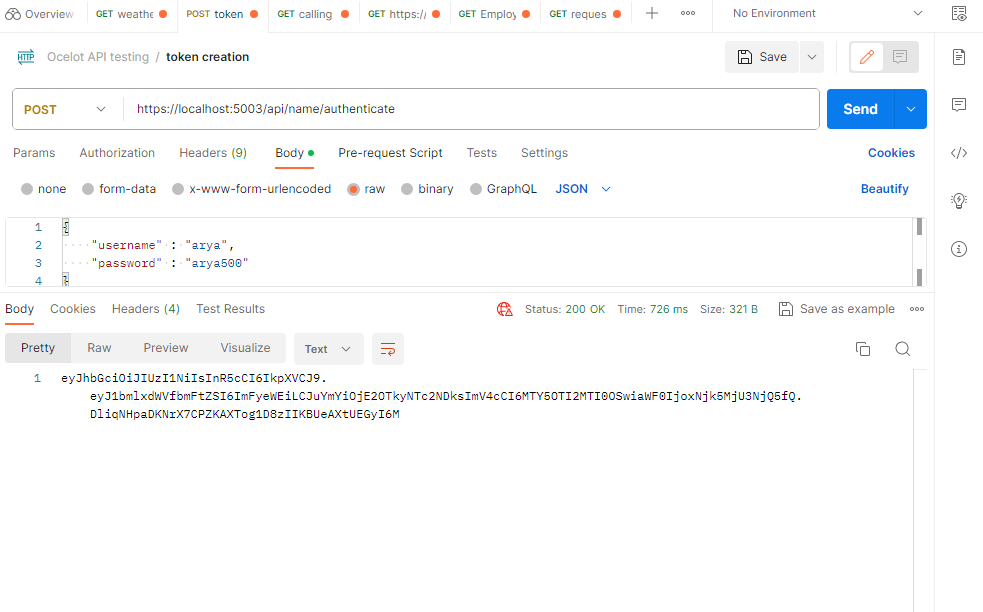
**\*** Microsoft.AspNetCore.Authentication.JwtBearer

**\*** Microsoft.AspNetCore.Authentication

**\*** System.IdentityModel.Tokens.Jwt

* I have created a different project to create the jwt token only and made some configuration in it You can refer the github link attached at the end of this document.



* As you can see , I have created Auth.Demo webapi project . Then I have created one class named JwtAuthenticationManager which is inheriting properties from IjwtAuthenticationManager and it is a interface .
* This inteface has 2 properties named username and password. If the username and password that we gonna pass to create jwt token matches with 2 usernames and password that I have mentioned in users Dictionary , then it will create Jwt Token.
* We need some NuGet package dependency . So I have installed 4 of them.
* After successfully validating the username and password , it will return us a complex JwtToken , which will be used as a security key to get the response from the Gateways.
* Above I have passed the desired username and password in header of the request . After that it is giving a JwtToken .
* For the particular API’s downstream request I am adding some configuration Ocelot.json file.

{

"DownstreamPathTemplate": "/WeatherForecast",

"DownstreamScheme": "https",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 7246

}

],

"UpstreamPathTemplate": "/api/Weather",

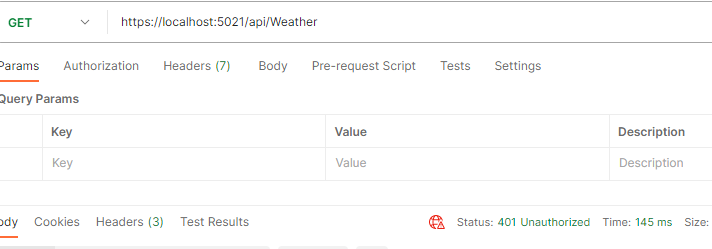
"UpstreamHttpMethod": [ "Get" ],

"AuthenticationOptions": {

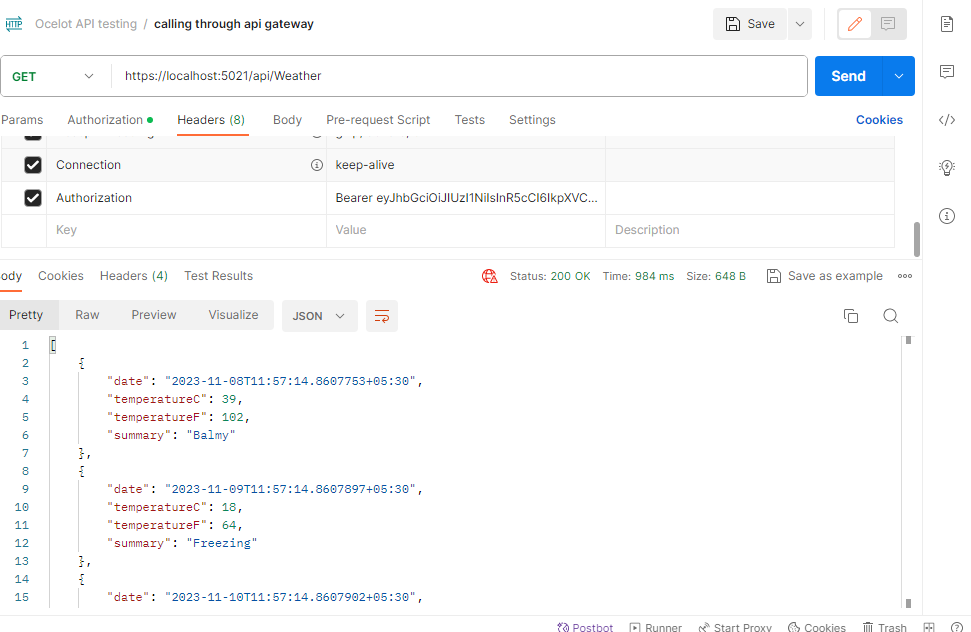
"AuthenticationProviderKey": "Bearer",

"Allowedscopes": []

* Now we want access the web api , we cant access. It will give us unauthrized 401 error in both web and the testing tool.



* Now let us send the request with jwt token to get the result.



**4.Rate Limiting:-**

* Rate Limiting in API Gateways is a method to control howmany requests a client can make to your services within a specific timeframe.
* It ensures fair usage and safeguards your services from overloading.
* We can configure this feature in ocelot.json file .

"RateLimitOptions": {

"ClientWhitelist": [],

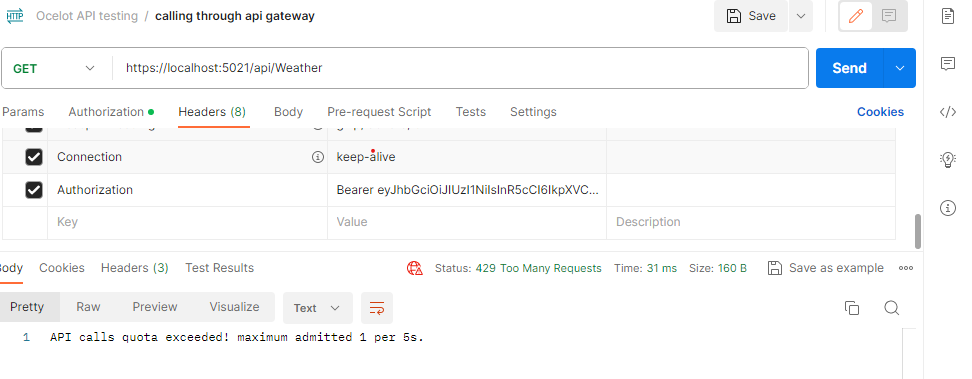
"EnableRateLimiting": true,

"Period": "2s",

"PeriodTimespan": 1,

"Limit": 1

}

* The meaning of above configuration is that we can make only 1 request per 5 seconds . If we try to get more than 1 request in 5 seconds it will give us API Quota exceeded message .

**5.Caching:-**

* Caching in API Gateways involves storing and reusing previously retrieved responses to reduce the load on backend services and improve response times.
* API Gateways store responses to previous requests, like data or content, for a specific period.
* When a client requests the same data, the API Gateway can serve the cached response instead of making the same request to the backend service.
* Caching speeds up responses, reduces load on services, and can enhance the overall performance of your API.
* We can control which data to cache, how long to store it, and when to invalidate or refresh the cache.
* For a specific period of time , previous request’s result will be stored in cache memory and suceeding request’s result will be same as stored in cache memory.
* "FileCacheOptions": { "TtlSeconds": 15 }
* For response caching we need to add a separate NuGet package “Ocelot.Cache.CacheManager”

**6. Header Transformation:-**

* Header transformation in Ocelot API Gateway involves altering or adding HTTP headers in both incoming client requests and outgoing service responses.
* This process allows for customization and adaptation of headers to align with the specific requirements of your services or clients, enhancing compatibility, and enabling tailored interactions within your microservices architecture.
* These transformations are configured in the API Gateway to ensure seamless and efficient communication between clients and services.
* In ocelot.json we can configure this feature.
* "UpstreamHeaderTransform": {

"Uncle": "Bob"

}

Above is the syntax If you want to add a header to your upstream request please add the following to a Route in your **ocelot.json**

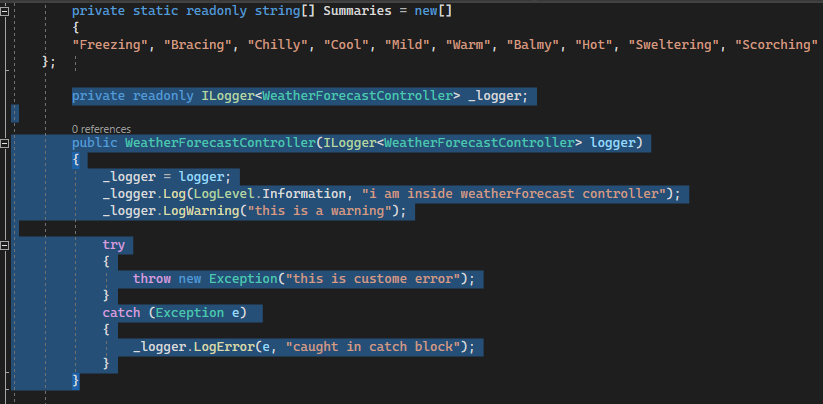
* **"Test":** "http://www.bbc.co.uk/, http://ocelot.com/"

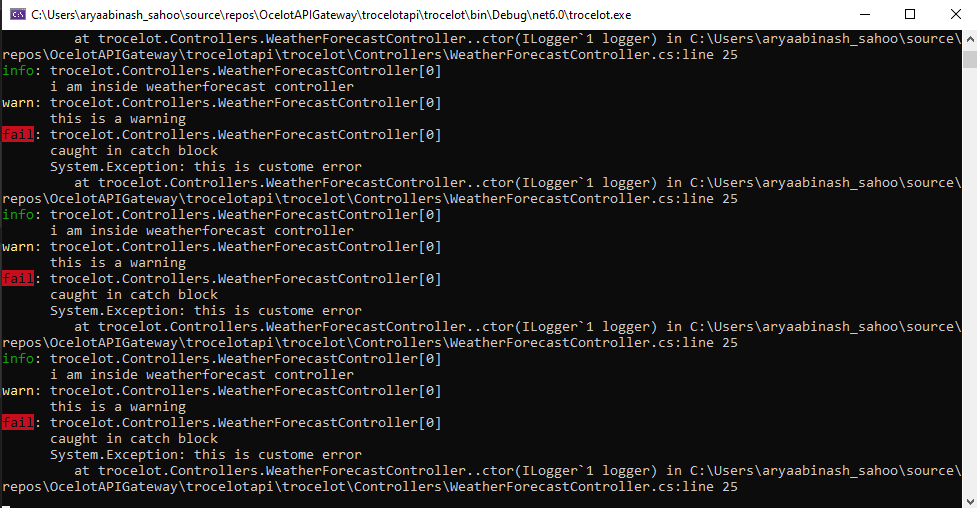
Above is the syntax , The key is Test and the value is http://www.bbc.co.uk/, http://ocelot.com/. The value is saying replace http://www.bbc.co.uk/ with http://ocelot.com/.

**7.Http Error Status Code:-**

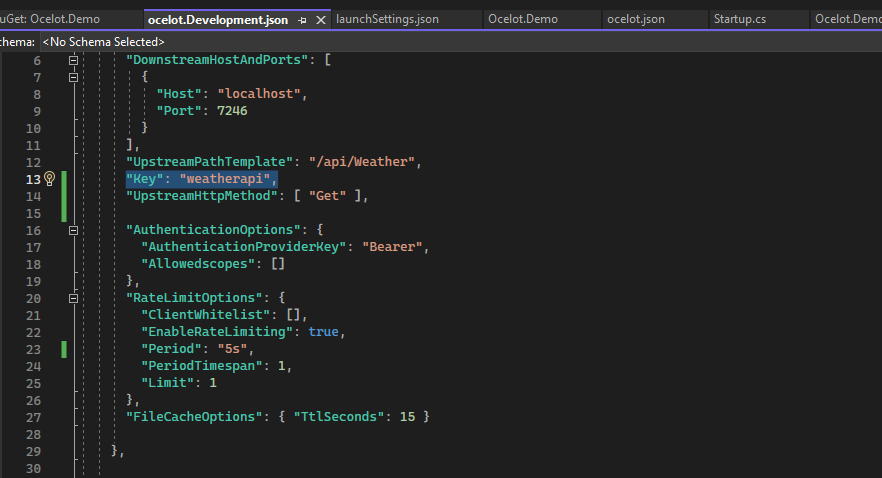
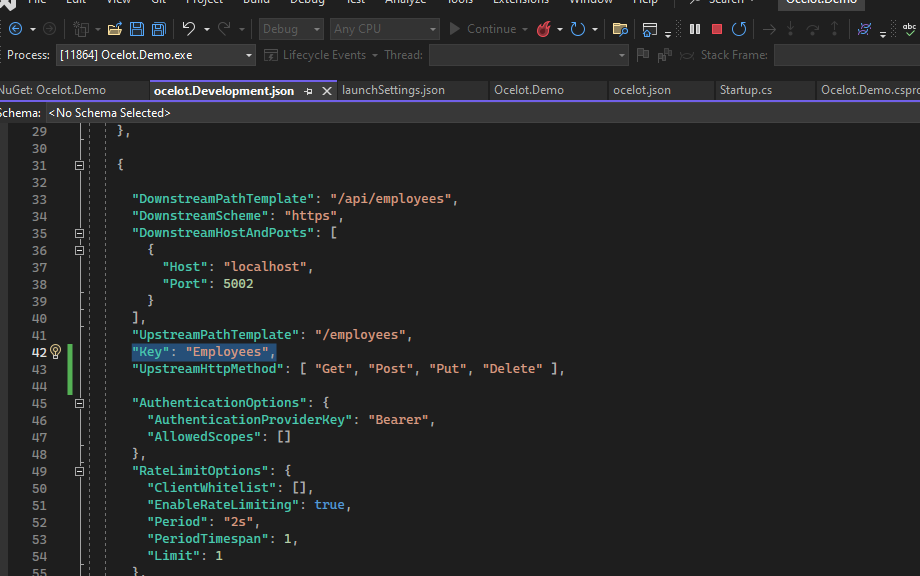
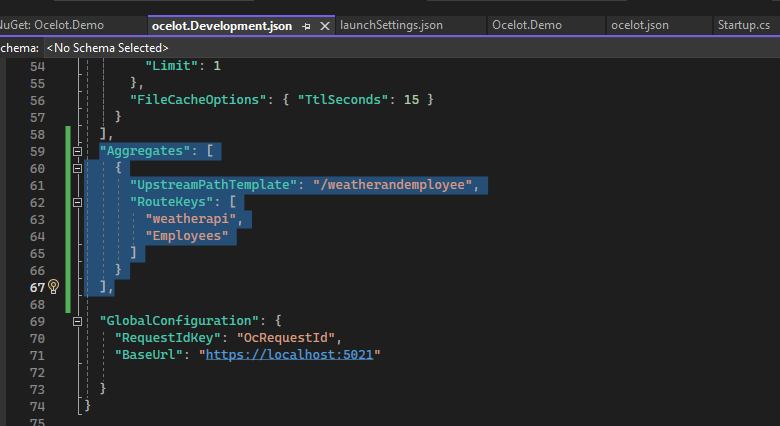
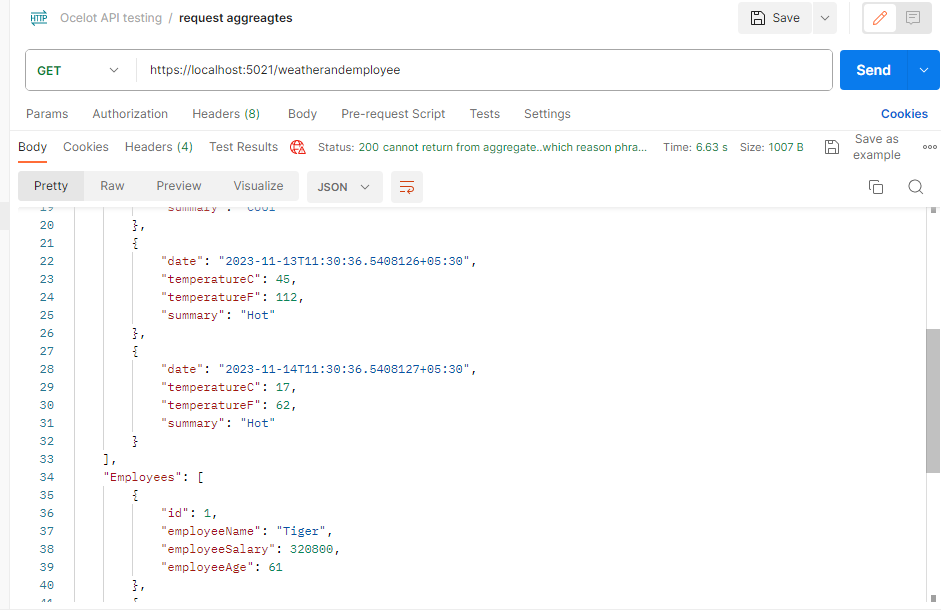
* When Ocelot interacts with downstream services and clients, it may need to communicate various HTTP status codes. Here are some common HTTP error status codes that Ocelot may return to clients or handle when interacting with downstream services .
* Below are error that can occur during executing APIs through APIGateways.
* ****200** :**This status code is returned when a request is successfully processed, and the response is as expected.
* **201 Created:** It's used when a new resource is successfully created as a result of a POST request.
* **204 No Content:** Indicates that the request was successful, but there is no response body to return.
* **400 Bad Request:** Typically returned for malformed or invalid requests.
* **401 Unauthorized:** Indicates that the client needs to provide proper authentication credentials for the request to be processed.
* **403 Forbidden:** The request is understood, but the server refuses to fulfill it. It often relates to permissions and access control.
* **404 Not Found:** The requested resource could not be found on the server.
* **500 Internal Server Error:** Indicates a generic server error, usually for unexpected issues during processing.
* **502 Bad Gateway:** Often returned when the API Gateway is unable to reach the upstream service, indicating a problem in the gateway's configuration or the service's availability.

**8. Logging:-**

* Logging in Ocelot API Gateway is the practice of recording and tracking information related to incoming requests, outgoing responses, and other operational events.
* Logging is crucial for monitoring, debugging, and maintaining the API gateway.
* Ocelot provides built-in logging capabilities to help you understand what's happening within the gateway.
* In simple we can say this is messaging feature and it will send response to client which api is executed.
* We need to use logging interfaces ILoggerFactory and ILogger<T> at the moment. This is encapsulated in IOcelotLogger and IocelotLoggerFactory.
* We can check the response in console windows. Above I have implemented 3 types of logging. One is for simple messaging. 2nd one is for giving warning to Client. 3rd one is for showing error and throw exception.

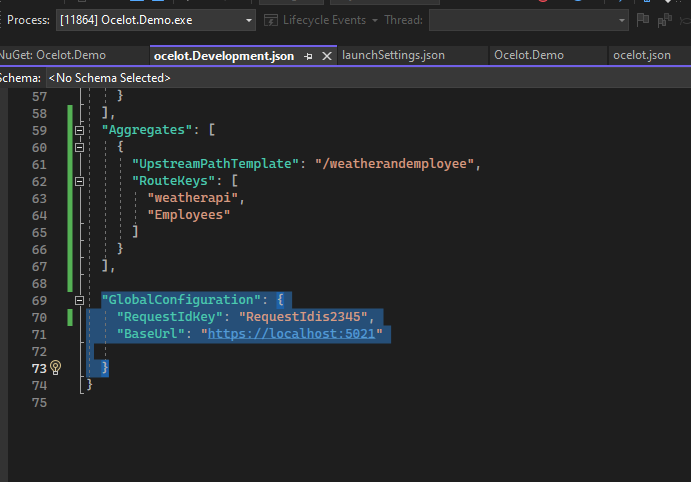


**9.Request Aggregation:-**

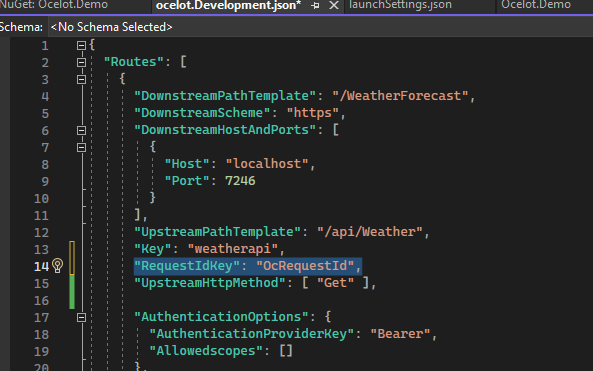
* Request aggregation in Ocelot API Gateway combines multiple incoming requests into a single request sent to a backend service.
* It reduces the number of requests, minimizes latency, and can aggregate responses for clients, improving efficiency.
* If we using multiple APIs , then it need to consume more ports . It is not a practice using multiple ports for multiple APIs.
* In Request Aggregation feature , we can attach keys to each DownstreampathTemplate. We need to add one more term called Aggregates and inside that we can mention the UpstreamPathTemplate and Routekeys.
* Above things need to be in Ocelot.json.
* Above I have two APIs.So I have created one key for each APIs .
* Here the Aggregates parametre is declared and inside that , I have given one new upstreampathtemplate. Below that routekeys is used for concatinating both APIs response.
* So when we want get the result of both the APIs with single request , then we will search “ <https://localhost:5021/weatherandemployee> ” .
* Above is the result , in which we are getting response from both the APIs with single request .

**10.Request ID:-**

* Request IDs are used for tracking and correlating requests as they traverse through a distributed system .
* We can set RequestID in the form of a header . So that ocelot will use the id as logging when the downstreampath is available.
* We can add the feature in 2 ways. Through GlobalConfiguration and Route .
* Through GlobalConfiguration the syntax will be



* If want to assign request id to specific downstreampath , then we can assign a property called RequestIdKey .

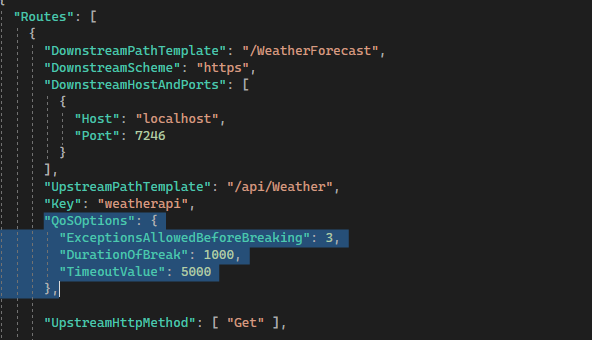


**11.Quality Of Service:-**

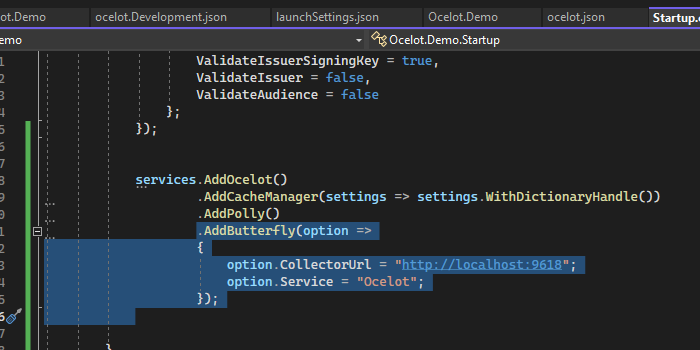
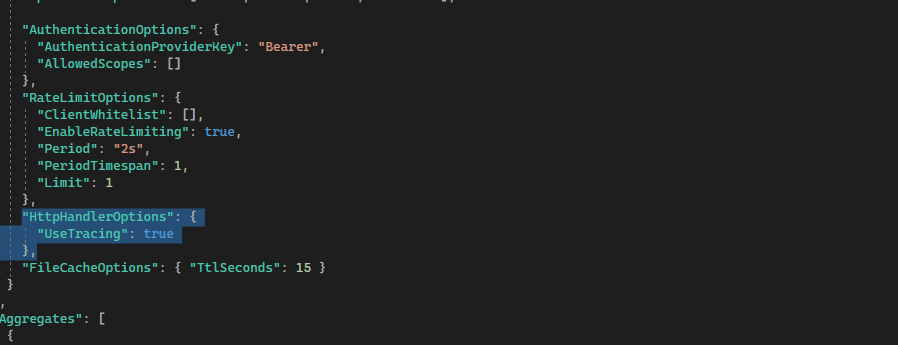
* In ocelot, Quality of Service refers to a set of features that allow you to control and manage quality of service provided by the API Gateway for requests of downstream services.
* It is used to ensure the stability reliability, stability and fault tolerance of your microservice architecture.
* We need to install the Ocelot.Provider.Polly NuGet package and in configuration class we need to add the polly.
* services**.**AddOcelot**()**

**.**AddPolly**();**

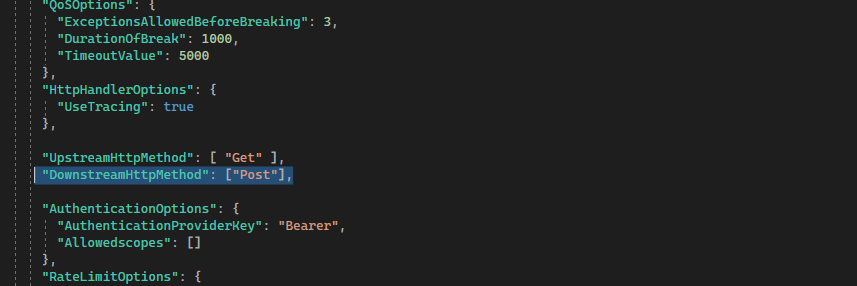
* Ocelot supports one QoS capability at the current time.
* Below we can see ,in the weatherapi , we have added QoSOptions. Inside that we mentioned some features.
* **ExceptionsAllowedBeforeBreaking:-** This specifies the number of exceptions allowed before marking the downstream service as unhealthy and breaking the circuit.
* **DurationOfBreak:-** This is the duration (in ms) for which the circuit remains unavailable after reaching the allowed exception limit.
* **TimeoutValue:-** The maximum time(in ms) allowed for a request to the downstream service. If the request eceeds this timeout, this will be considered as failure.
* These techniques are used to implement circuit breaking and fault tolerance mechanism.When the downstream service exceeds the defined exception or timeout limits , ocelot will temporarily break the circuit and make it unavailable.



**12.Tracing:-**

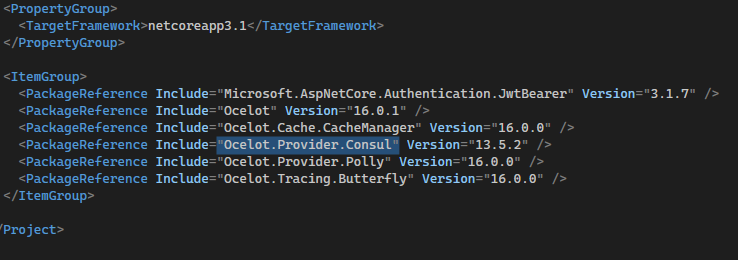
* **O**celot provides tracing functionality from the excellent Butterfly project.
* We need to install a Nuget package called Ocelot.Tracing.Butterfly.
* Above I have added the Butterfly pipeline in startup.cs of Ocelot project.
* After that in Ocelot.devlopment.json ,add the following to the route you want to trace.
* To see the trace , we can browse the [http://localhost:9618](http://localhost:9618/) mentioned in the pipeline.

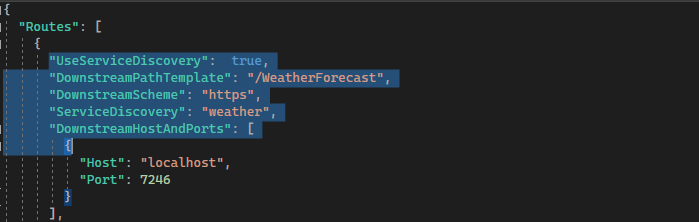
**13. Method Transformation:-**

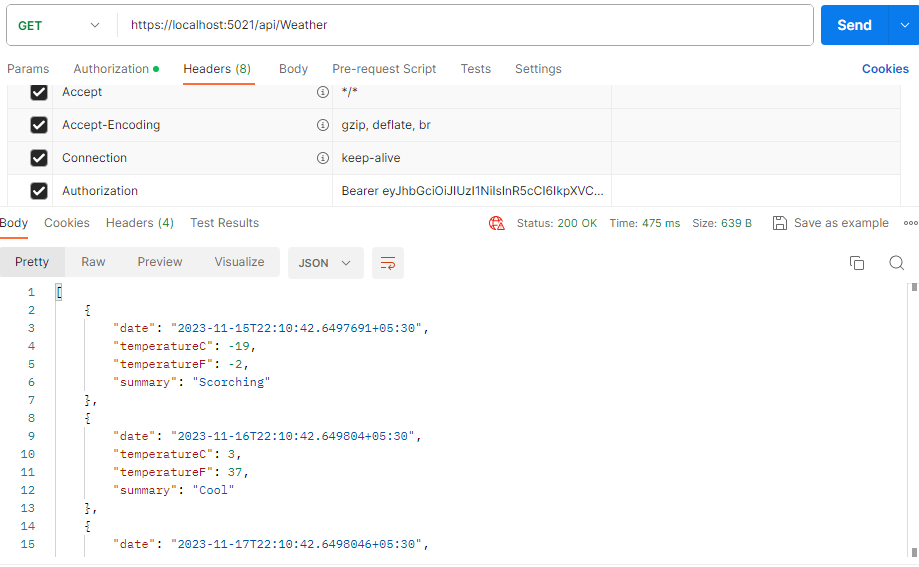
* In Ocelot API Gateway , method transformation involves using custom middleware to modify the requests or responses as they pass through the Ocelot pipeline.
* This transformation allows you to alter the incoming request’s method before it reaches the downstream service.
* For example , I might want to receive a request as a post method at the API Gateway but transform it into different method like PUT ,before forwarding it to actual downstream service.
* In above picture , we can see the UpstreamHttpMethod is GET , but when the request will be passing through Gateway , before that GET request will be converted into Post request.

**14. Service Discovery:-**

* Service discovery-based routing in Ocelot is designed to address the challenges presented by the dynamic and decentralized nature of microservices architectures.
* It enables automatic adaptation to changes, load balancing, fault tolerance, and centralized management of service locations.
* Traditional routing, while suitable for monolithic applications, may lack the flexibility and adaptability needed in a microservices environment which I have xplained alrady in point no.1 & 2 .
* To implement this feature , we need to install the Ocelot.Provider.Consul .
* Then in ocelot.json file we need add some extra paramaters and values to make the whole project service discovery enabled.
* In configurationservice pipeline , we need to add AddConsul() also.
* I am adding pictures below.

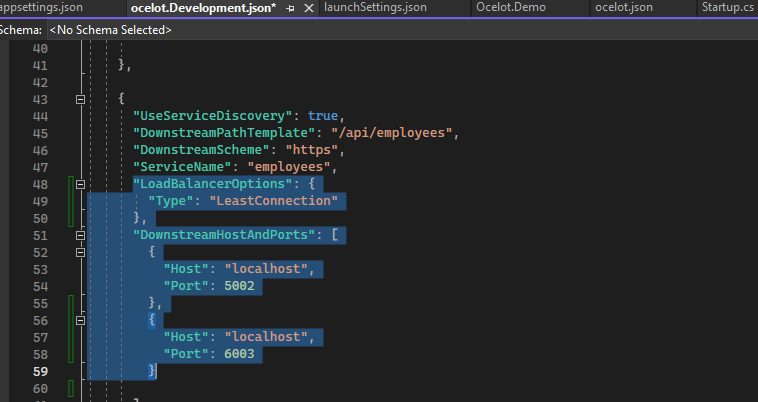




* In above pictures , we have configurred the whole the service discovery and consul implmentation in ocelot.json.
* Same way as in routing , here we are trying to get the response from Gateway using the URL configured in ocelot.json after implementation of Service discovery and we gor the desired Output.

**15. Load Balancer:-**

* Ocelot API Gateway supports various load balancing strategies to distribute incoming requests among multiple instances of a microservice.
* Load balancing is essential in a microservices architecture to ensure efficient resource utilization, fault tolerance, and scalability.
* Ocelot supports several load balancing options that you can configure based on your application's requirements.
* There are multiple type of load balancing such as Round Robin , LeastConnection , StickyCookieSession ,



* In RoundRobin requests are distributed sequentially to each instance in a circular order.
* In LeastConnection requests are sent to the instance with the fewest active connections. This helps distribute the load more evenly, especially in scenarios where instances may have different capacities.
* In StickyCookieSession Requests from the same client are consistently routed to the same instance based on a specific identifier .
* In this project I am going to use LeastConnection. In above picture , we can see the configuration that is implemented for load balancer .

**16. WebSockets:-**

* Ocelot, which is primarily designed as an HTTP API Gateway, doesn't have native or built-in support for handling WebSocket connections.
* For WebSocket support in a microservices architecture, consider using other tools or libraries that specialize in WebSocket communication.
* Some popular options include ASP.NET Core SignalR for .NET-based applications or dedicated WebSocket gateways like Socket.IO for more general use cases.
* In startup.cs we need to add below mentioned configuration as pipeline.
* Then in ocelot.json , add the following to proxy a Route using Websockets
* Configure**(**app =>

**{**

app**.**UseWebSockets**();**

app**.**UseOcelot**().**Wait**();**

**})**

**{**

**"UpstreamPathTemplate":** "/"**,**

**"DownstreamPathTemplate":** "/ws"**,**

**"DownstreamScheme":** "ws"**,**

**"DownstreamHostAndPorts":** **[**

**{** **"Host":** "localhost"**,** **"Port":** 5001 **}**

**]**

**}**

* This Ocelot configuration is tailored to handle WebSocket traffic at the root ("/"). Essentially, Ocelot acts as a mediator, shuttling messages back and forth between the client and a downstream service located at localhost:5001/ws.
* It ensures a smooth bidirectional flow, receiving WebSocket messages from the client, forwarding them to the service, receiving service responses, and sending them back to the client.

**References:-**

* The official documentation of Ocelot APIGateway

[Welcome to Ocelot 20.0 — Ocelot 20.0.0 documentation](https://ocelot.readthedocs.io/en/latest/)

* Source Code attached in below.

[charlie8658/OcelotAPIGatewayRepo (github.com)](https://github.com/charlie8658/OcelotAPIGatewayRepo)

**Conclusion:-**

* Ocelot API Gateway stands out as a powerful and flexible tool in the realm of microservices architectures by providing features such as dynamic routing , load balancing ,authentication , service discovery etc.
* Ocelot simplifies the complexities associated with coordinating communication between miceroservices.
* Its adaptability to changes in service instances , support for various load balancing strategies and integration with popular srvice discovery mechanisms make it wellsuited for building scalable , resilient and dynamic microservices ecosyatem.
* Ocelot’s role as a central entry for managing and routing requests enhances the efficiency and maintainability of microservice based applications
* As a crucial component in modern distributed system , Ocelot empowers developers to built robust and scalable microservices architectures while abstracting away many of the intricate details of service communication and routing.