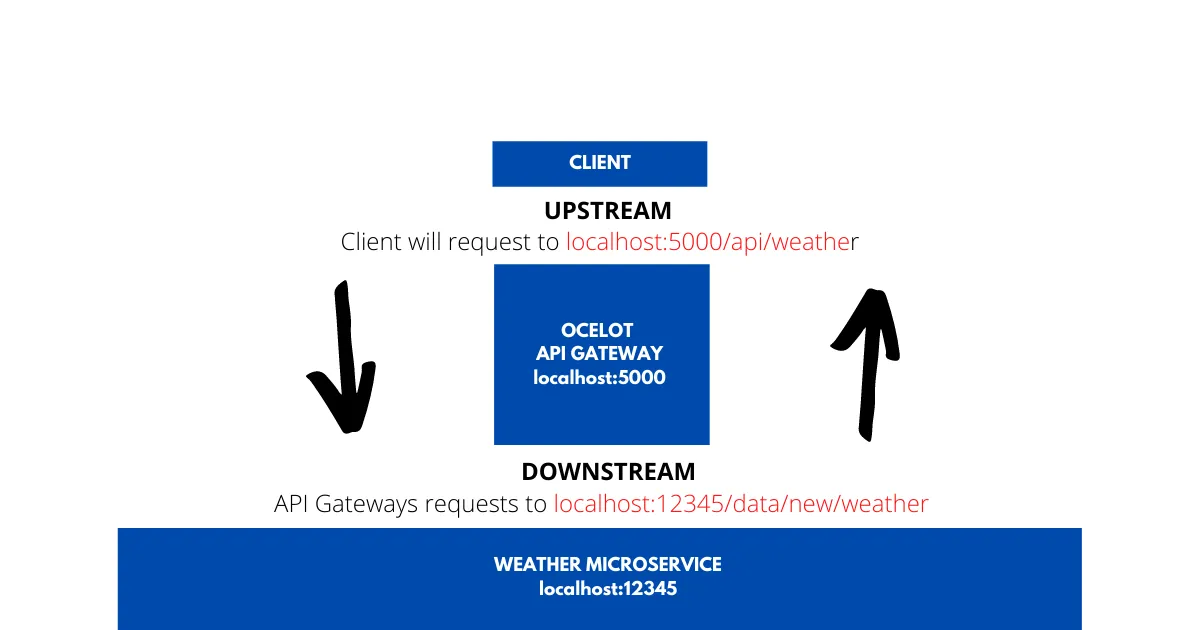
## What Is an API Gateway?

* When building microservices-based applications, an API Gateway is needed to have a central place where authentication, routing, orchestration, etc., is implemented.
* Without an API Gateway in place, you might typically implement each of these in each service, and hence maintaining them for each service would be a daunting task at hand.
* An API Gateway decouples the service producer from its consumer, providing a security layer since you need not expose your microservices directly.

**Introduction To Ocelot API Gateway**

* Ocelot is an Open Source API Gateway for the .NET/Core Platform.
* It unifies multiple microservices so that the client does not have to worry about the location of each and every Microservice.
* Ocelot API Gateway transforms the Incoming HTTP Request from the client and forward it to an appropriate Microservice.
* Ocelot is widely used by Microsoft and other tech-giants as well for Microservice Management.
* It will be as easy and installing the Ocelot package to your API Gateway project and setting up a JSON Configuration file that states the upstream and downstream routes.
* Upstream and Downstream are 2 terms that you have to be clear with.
* Upstream Request is the Request sent by the Client to the API Gateway. Downstream request is the request sent to the Microservice by the API Gateway. All these are from the perspective of the API Gateway.



The API gateway is located at port **5000**, whereas the Microservice Port is at **12345**. Now, the client will not have access to port 12345, but only to 5000. Thus, client sends a request to **localhost:5000/api/weather** to receive the latest weather. Now what the Ocelot API Gateway does is. It takes in the incoming request from the client and sends another HTTP Request to the Microsevrice, which in turn returns the required response. Once that is done, the Gateway send the response to the client. Here is localhost:5000 is the upstream path that the client knows of. localhost:123456 is the downstream path that the API Gateways knows about.

**In this way, Ocelot API Gateway will be able to re-route various requests from client to all the involved Microservices.** We will have to configure all these routes within the API Gateway so that Ocelot knows how and where to route the incoming requests.

Here are few Features of Ocelot

1. Routing the Incoming Request to the required Microservice
2. Authentication
3. Authorization
4. Load Balance for Enterprise Applications.

**Building An Ocelot API Gateway**

Install-Package Ocelot

Let’s configure Ocelot to work with our ASP.NET Core 5.0 Application. Go to the Program.cs of the Gateway.WebApi Project and change the CreateHostBuilder method as follows.

public static IHostBuilder CreateHostBuilder**(**string**[]** args**)** =**>**

Host.CreateDefaultBuilder**(**args**)**

.ConfigureWebHostDefaults**(**webBuilder =**>**

**{**

webBuilder.UseStartup**<**Startup**>()**;

**})**

.ConfigureAppConfiguration**((**hostingContext, config**)** =**>**

**{**

config

.SetBasePath**(**hostingContext.HostingEnvironment.ContentRootPath**)**

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

**})**;

Since Ocelot reads it’s route configuration from a JSON config file, adds the new json file so that the ASP.NET Core Application is able to access these settings. Note that we have not yet created the ocelot.json file. We will be doing it once we have configured the Ocelot Middleware.

Next, Navigate to the Startup.cs of the same Gateway.WebApi Project and add Ocelot to the ConfigureServices method.

public void ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddOcelot**()**;

**}**

Finally, go to the Configure method and make the following changes. This adds Ocelot Middleware to the ASP.NET Core 5.0 Application’s Pipeline.

public async void Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

if **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapControllers**()**;

**})**;

await app.UseOcelot**()**;

**}**

**Configuring Ocelot Routes**

This is the most important part . Here is where you would configure the Upstream / Downstream routes for the API Gateways, which helps Ocelot to know the routes.

Create a new JSON file in the root of the Gateways.WebApi Project. This file would contain the configurations needed for Ocelot. We will name this file as ocelot.json , as we have already registered this name back in Program.cs file, remember?

**{**

"Routes": **[**

**{**

"DownstreamPathTemplate": "/api/product",

"DownstreamScheme": "https",

"DownstreamHostAndPorts": **[**

**{**

"Host": "localhost",

"Port": 44337

**}**

**]**,

"UpstreamPathTemplate": "/gateway/product",

"UpstreamHttpMethod": **[** "POST", "PUT", "GET",”Delete” **]**

**}**

**]**

**}**

Ocelot takes in an Array of Route Objects.  
As the first element in the array, let’s configure the Product Microservice’s Get All, Update and Insert Endpoints.

**DownstreamPathTemplate** denotes the route of the actual endpoint in the Microservice.  
**DownstreamScheme**is the scheme of the Microservice, here it is HTTPS  
**DownstreamHostAndPorts**defines the location of the Microservice. We will add the **host**and **port**number here.

**UpstreamPathTemplate** is the path at which the client will request the Ocelot API Gateway.  
**UpstreamHttpMethod**are the supported HTTP Methods to the API Gateway. Based on the Incoming Method, Ocelot sends a similar HTTP method request to the microservice as well.

Build the solution to ensure that there are no errors. Now, there is one thing to change. We have 3 APIs now. Let’s configure the Solution so that all the 3 APIs get fired when you run the application. This is because we will need all the APIs online.

Great, we have successfully implemented API Gateways and made a simple Microservice Architecture in ASP.NET Core for ourselves.

With POSTMAN, you can test the POST and UPDATE methods as well.

But we are still missing**GetById and Delete HTTP Methods**. We will have to add another route for this, because we are also passing an ID parameter to these endpoints. Understand?

This is how the next route will look like. This is more of a parameter based route with other settings similar to the previous one.

**{**

"Routes": **[**

**{**

"DownstreamPathTemplate": "/api/product",

"DownstreamScheme": "https",

"DownstreamHostAndPorts": **[**

**{**

"Host": "localhost",

"Port": 44337

**}**

**]**,

"UpstreamPathTemplate": "/gateway/product",

"UpstreamHttpMethod": **[** "POST", "PUT", "GET" **]**

**}**,

**{**

"DownstreamPathTemplate": "/api/product/{catchall}",

"DownstreamScheme": "https",

"DownstreamHostAndPorts": **[**

**{**

"Host": "localhost",

"Port": 44337

**}**

**]**,

"UpstreamPathTemplate": "/gateway/product/{catchall}",

"UpstreamHttpMethod": **[** "GET", "DELETE" **]**

**}**

**]**

**}**

We are accepting any Parameter in the route.  
We restrict the methods to just GET and DELETE