Introduction To API Gateway – Ocelot

Ocelot is a .NET API Gateway. This project is aimed at people using .NET running a microservices / service-oriented architecture that need a unified point of entry into their system. However, it will work with anything that speaks HTTP(S) and run on any platform that ASP.NET Core supports.

Ocelot is a strong instrument that simplifies API management by reducing complexity in microservice architectures. While organizing communication between services, this API gateway solution provides the advantage of managing all requests from a single point.

Microservice Architectures and API Gateway Concept

Ocelot is a reliable solution which has been designed to manage incoming requests, implement security measures and realize efficient traffic management by creating a bridge between different microservices, thereby contributing to the efficient operation of microservice architectures and making inter-system communication crystalline.

Ocelot at Work as API Gateway

Ocelot, while working as an API gateway, also facilitates communication between different services by offering a flexible and modular structure. Its working principle is based on processing incoming requests according to predefined routing and forwarding them to the relevant services in line with predefined routing. The library consists of a number of features, and it receives requests from clients, directs them, filters them and provides communication between services.

Besides, it draws attention thanks to its additional functionality such as monitoring and logging communication with clients and services. By means of the flexible configuration capabilities of Ocelot, a simple and straightforward structure can be set in order to route incoming requests, and thus, this gives developers a wide range of customization possibilities.

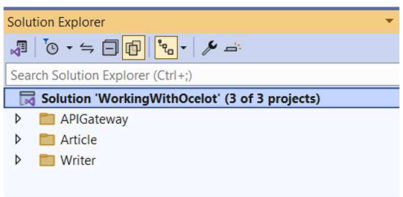
Sample Application Using Ocelot

Just to understand basic functionality of Ocelot with Microservices, we are going to create three APIs to represent our microservices. When we finish with implementation, every microservice is going to be inside the same .NET Solution.

By adding 2 microservices behind the API Gateway, it will redirect all requests from outside to a single Ocelot API project. Requests will then be directed to the relevant services.

Create an empty solution that will be the container for all of our APIs. I will keep it in a separate folder named OcelotDemos and name of solution will be WorkingWithOcelot.

Once the solution is ready, let's create three folders:



- API gateway
- Article
- Writer

Creating Microservices

Now, let's create our first microservice inside the Article folder.

We are going to create an ASP.NET Core Web API project and name it - Article.Api.

Create Models folder and a model class named Article in it:

```
namespace Article.Api.Models
{
    public class Article
    {
        public int Id { get; set; }
        public string? Title { get; set; }
        public DateTime LastUpdate { get; set; }
        public int WriterId { get; set; }
    }
}
```

Add a repository folder to the project and to follow repository design pattern create an interface named IArticleRepository inside Repository folder.

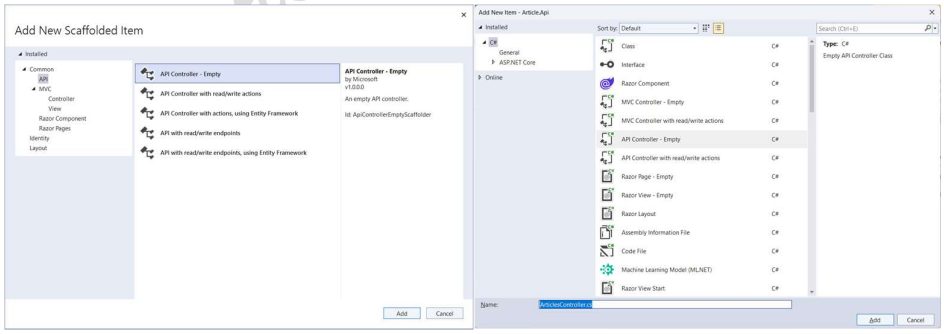
```
namespace Article.Api.Repository
{
    public interface IArticleRepository
    {
        int Delete(int id);
        Models.Article? Get(int id);
        List<Models.Article> GetAll();
    }
}
```

Also create a repository class named ArticleRepository that represents our database and the operations.

```
namespace Article.Api.Repository
{
    public class ArticleRepository : List<Models.Article>,
IArticleRepository
    {
        private readonly static List<Models.Article> _articles =
Populate();

        private static List<Models.Article> Populate()
        {
            var result = new List<Models.Article>()
        {
            new Models.Article
            {
                Id = 1,
                Title = "First Article",
                WriterId = 1,
                LastUpdate = DateTime.Now
            },
            new Models.Article
            {
                Id = 2,
                Title = "Second title",
```

```
                WriterId = 2,
                LastUpdate = DateTime.Now
            },
            new Models.Article
            {
                Id = 3,
                Title = "Third title",
                WriterId = 3,
                LastUpdate = DateTime.Now
            }
        };

            return result;
        }

        public List<Models.Article> GetAll()
        {
            return _articles;
        }

        public Models.Article? Get(int id)
        {
            return _articles.FirstOrDefault(x => x.Id == id);
        }

        public int Delete(int id)
        {
            var removed = _articles.SingleOrDefault(x => x.Id ==
id);
            if (removed != null)
                _articles.Remove(removed);

            return removed?.Id ?? 0;
        }
    }
}
```

Add an API Controller named ArticlesController to the Controllers folder:



```
using Article.Api.Repository;
using Microsoft.AspNetCore.Mvc;

namespace Article.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ArticlesController : ControllerBase
    {
        private readonly IArticleRepository _articleRepository;

        public ArticlesController(IArticleRepository
articleRepository)
        {
            _articleRepository = articleRepository;
        }

        [HttpGet]
        public IActionResult Get()
        {
            return Ok(_articleRepository.GetAll());
        }

        [HttpGet("{id}")]
        public IActionResult Get(int id)
        {
            var article = _articleRepository.Get(id);
            if (article is null)
                return NotFound();

            return Ok(article);
        }

        [HttpDelete("{id}")]
        public IActionResult Delete(int id)
        {
            var deletedId = _articleRepository.Delete(id);
            if (deletedId == 0)
                return NotFound();

            return NoContent();
        }
    }
}
```

Notice that the first endpoint is responsible for returning every article from the database.

The second is responsible for returning a single article based on the id we send as a parameter. If it doesn't exist in the database, this endpoint is going to return a NotFoundResult (HTTP status code 404).

The third endpoint receives an id as a parameter and deletes this specific record from the database. In case this id doesn't exist, this endpoint returns a NotFound (HTTP status 404).

As we are interested in running it as isolated project and using a standard PORT id so, it can be configured in Ocelot Gateway API, modify the launchSettings.json. Here I am removing all profiles and creating a new profile named Article.Api.

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:25027",
      "sslPort": 44342
    }
  },
  "profiles": {
    "Article.Api": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "https://localhost:5001",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

To finish this initial project, above code set up our application to run at localhost port 5001 and don't start a swagger page.

Create a Writer Microservice

Similar to the previous API, let's create our second microservice inside the Writer folder. Create an API inside the writer folder. Create an ASP.NET Core Web API project and name it - Writer.Api. First create Models folder and a Writer model class in it:

```
namespace Writer.Api.Models
{
    public class Writer
    {
        public int Id { get; set; }
        public string? Name { get; set; }
    }
}
```

Add a repository folder to the project and create an interface named IWriterRepository inside Repository folder.

```
namespace Writer.Api.Repository
{
    public interface IWriterRepository
    {
        List<Models.Writer> GetAll();
        Models.Writer? Get(int id);
        Models.Writer Insert(Models.Writer writer);
    }
}
```

Now, create a list of writers and the methods responsible for filtering and inserting writers. Add implementation of IWriterRepository in WriterRepository class:

```
namespace Writer.Api.Repository
{
    public class WriterRepository : List<Models.Writer>,
IWriterRepository
    {
        private readonly static List<Models.Writer> _writers =
Populate();

        private static List<Models.Writer> Populate()
        {
            return new List<Models.Writer>
            {
                new Models.Writer
                {
                    Id = 1,
                    Name = "Leanne Graham"
                },
                new Models.Writer
                {
                    Id = 2,
                    Name = "Ervin Howell"
                },
                new Models.Writer
                {
                    Id = 3,
                    Name = "Glenna Reichert"
                }
```

```
            };
        }

        public List<Models.Writer> GetAll()
        {
            return _writers;
        }

        public Models.Writer Insert(Models.Writer writer)
        {
            var maxId = _writers.Max(x => x.Id);

            writer.Id = ++maxId;
            _writers.Add(writer);

            return writer;
        }

        public Models.Writer? Get(int id)
        {
            return _writers.FirstOrDefault(x => x.Id == id);
        }
    }
}
```

Now, we are going to add a WritersController class and three endpoints. The first and the second endpoint return every writer in the database and a single writer filtered by an id (very similar to the Article.Api). The third endpoint is responsible for inserting a new writer into the database:

```
using Microsoft.AspNetCore.Mvc;
using Writer.Api.Repository;

namespace Writer.Api.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class WritersController : ControllerBase
    {
        private readonly IWriterRepository _writerRepository;

        public WritersController(IWriterRepository writerRepository)
        {
            _writerRepository = writerRepository;
```

```
        }

        [HttpGet]
        public IActionResult Get()
        {
            return Ok(_writerRepository.GetAll());
        }

        [HttpGet("{id}")]
        public IActionResult Get(int id)
        {
            var writer = _writerRepository.Get(id);

            if (writer is null)
                return NotFound();

            return Ok(writer);
        }

        [HttpPost]
        public IActionResult Post([FromBody] Models.Writer writer)
        {
            var result = _writerRepository.Insert(writer);

            return Created($"/get/{result.Id}", result);
        }
    }
}
```

At the last of Writer Microservice, we should modify the launchSettings.json file to set up this microservice to start at port 5002 and to don't use the swagger page (similar to the article API).

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:34025",
      "sslPort": 44346
    }
  },
  "profiles": {
```

```
    "Writer.Api": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "https://localhost:5002",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Creating the API Gateway

Finally, we are going to implement our API Gateway using Ocelot. First, let's create a new ASP.NET Core Web API project inside the ApiGateway folder and name it OcelotApiGateway.

At first, we will install Ocelot and the libraries which will make our configuration easier. Depending on the .net version you will use, the version of the packages may vary, so we will continue without specifying the version. Now, let's install the Ocelot package into our project. Go to Package Manager Console and run following command.

PM>Install-Package Ocelot

Then, let's change the LaunchSettings.json file to run at port 5003.

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:48198",
      "sslPort": 44356
    }
  },
  "profiles": {
    "OcelotApiGateway": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5003",
      "dotnetRunMessages": true
```

```
    }
  }
}
```

Once it is done, we are ready to create our configuration file.

Create the ocelot.json File

We need to create, at the root of our project, a new JSON file, and name it ocelot.json.

To start, let's modify the file and define two keys, GlobalConfiguration and Routes (an array of objects)

Inside the GlobalConfiguration key, we are going to set up our API Gateway host and port:

```
{
  "GlobalConfiguration": {
    "BaseUrl": "https://localhost:5003"
  },
  "Routes": [
    {}
  ]
}
```

As a continuation, let's configure our project to use the Ocelot.

Inside the Program class, in the builder section, let's add the ocelot.json file and the Ocelot service to our application:

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

Next, inside the Routes key, let's configure the endpoints of our application with some nested keys:

```
{
  "GlobalConfiguration": {
    "BaseUrl": "https://localhost:5003"
  },
  "Routes": [
    {
      "UpstreamPathTemplate": "/gateway/writers",
      "UpstreamHttpMethod": [
        "Get"
      ],
      "DownstreamPathTemplate": "/api/writers",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
```

```
        "Port": 5002
      }
    ],
    "FileCacheOptions": {
      "TtlSeconds": 10
    }
  },
  {
    "UpstreamPathTemplate": "/gateway/writers/{id}",
    "UpstreamHttpMethod": [
      "Get"
    ],
    "DownstreamPathTemplate": "/api/writers/{id}",
    "DownstreamScheme": "https",
    "DownstreamHostAndPorts": [
      {
        "Host": "localhost",
        "Port": 5002
      }
    ]
  },
  {
    "UpstreamPathTemplate": "/gateway/writers",
    "UpstreamHttpMethod": [
      "Post"
    ],
    "DownstreamPathTemplate": "/api/writers",
    "DownstreamScheme": "https",
    "DownstreamHostAndPorts": [
      {
        "Host": "localhost",
        "Port": 5002
      }
    ]
  },
  {
    "UpstreamPathTemplate": "/gateway/articles",
    "UpstreamHttpMethod": [
      "Get"
    ],
    "DownstreamPathTemplate": "/api/articles",
```

```
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "RateLimitOptions": {
        "EnableRateLimiting": true,
        "Period": "10s",
        "PeriodTimespan": 10,
        "Limit": 3
      }
    },
    {
      "UpstreamPathTemplate": "/gateway/articles/{id}",
      "UpstreamHttpMethod": [
        "Get",
        "Delete"
      ],
      "DownstreamPathTemplate": "/api/articles/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": 5001
        }
      ],
      "RateLimitOptions": {
        "EnableRateLimiting": true,
        "Period": "10s",
        "PeriodTimespan": 10,
        "Limit": 1
      }
    }
  ]
}
```

The **UpstreamPathTemplate** defines the URL of the API Gateway that receives the requests and then redirects to the microservice API (DownstreamPathTemplate).

The **UpstreamHttpMethod** defines the HTTP Methods (Get, Put, Post, Patch …) that the API Gateway uses to distinguish between the requests.

The **DownstreamPathTemplate** represents the endpoint at the microservice that is going to receive the request. In other words, it takes the request of the UpstreamPathTemplate and redirects to the DownstreamPathTemplate.

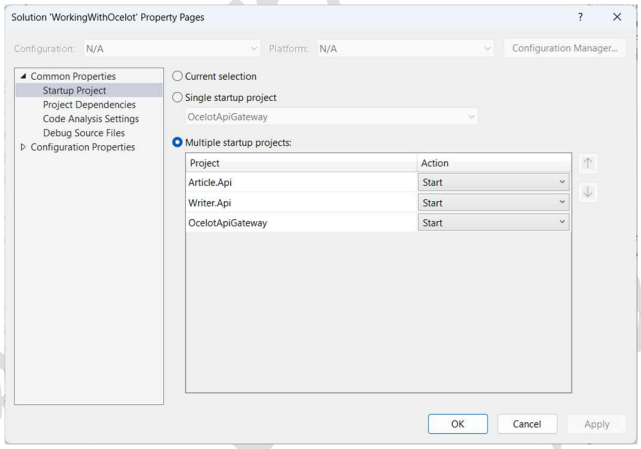
The **DownstreamScheme** represents the protocol to communicate with the microservice.

The **DownstreamHostAndPorts** defines the URL and the port from the microservices that are going to receive the requests.

the Application

Once all our APIs are inside the same .NET Solution, we need to configure the Visual Studio to run every API simultaneously.

To do that, let's right-click on the solution and then, click Properties:



With the property window open, in the left menu, we choose the option Startup Project. Then, in the right section, we choose the Multiple startup projects option and change all microservices Action to Start.

Notice that, when we run the application, it is going to start all microservices we have in our solution.

Requesting Endpoints

Now, instead of requesting our microservices endpoints directly, we are going to gather all requests in the Ocelot API Gateway UpstreamPathTemplate.

To request every article from our database, let's make a request to:

https://localhost:5003/gateway/articles

To get a specific writer from the database, let's make a request to:

https://localhost:5003/gateway/writers/1

The localhost:5003 means that we are making requests to our Ocelot API Gateway. The /gateway/article and the /gateway/writers/1, represents the UpstreamPathTemplate we previously configured in our ocelot.json file from our API Gateway.

Using Rate Limiting

Rate limiting is the process of restricting the number of requests for a resource within a specific time window.

In this scenario, we are going to limit the number of requests for the /api/articles endpoint:

```
{
  "UpstreamPathTemplate": "/gateway/articles",
  "UpstreamHttpMethod": [
    "Get"
  ],
  "DownstreamPathTemplate": "/api/articles",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    {
      "Host": "localhost",
      "Port": 5001
    }
  ],
  "RateLimitOptions": {
    "EnableRateLimiting": true,
    "Period": "10s",
    "PeriodTimespan": 10,
    "Limit": 3
  }
}
```

In our current configuration, we added a new RateLimitOptions object with some properties.

First, we need to set the EnableRateLimiting property value to true.

The second property we need to set up is the Period. This property defines the specific time window that this rate limit is acting on.

The PeriodTimespan defines the number of seconds we need to wait to request this endpoint after we got the maximum number of requests within the Period. In this case, let's set its value to 10 seconds.

The last property is the Limit. It defines the maximum number of requests within 10 seconds (Period property).

Once we request this UpstreamPathTemplate (/gateway/articles) more than 3 times within 10 seconds, the API Gateway is going to return a Too Many Request (HTTP status 429).

Using Cache With Ocelot

Caching is another easy-to-implement Ocelot feature.

First, we need to install the ocelot caching package using PMC:

>Install-Package Ocelot.Cache.CacheManager

The second step is to add the cache manager to our services inside the Program class:

```
builder.Services.AddOcelot(builder.Configuration)
      .AddCacheManager(x =>
      {
         x.WithDictionaryHandle();
      });
```

Then, we need to add the FileCacheOptions object to the endpoint we want to cache and set the TtlSeconds:

```
"Routes": [
  {
    "UpstreamPathTemplate": "/gateway/writers",
    "UpstreamHttpMethod": [
      "Get"
    ],
    "DownstreamPathTemplate": "/api/writers",
    "DownstreamScheme": "https",
    "DownstreamHostAndPorts": [
      {
        "Host": "localhost",
        "Port": 5002
      }
    ],
    "FileCacheOptions": {
      "TtlSeconds": 10
    }
  },
```

The TtlSeconds (Time-to-live in seconds) means the time that the Ocelot is going to cache the data. After that time, Ocelot is going to discard the cache.

While the data is in the cache, the API Gateway doesn't make an HTTP request to our microservice. That means we are saving resources from our microservice. Once the cache expires, the API Gateway requests the microservice once more and saves the data in the cache again.

Ocelot has a lot more features, and to learn more about it you can refer:

https://ocelot.readthedocs.io/en/latest/

☸✠End of Chapter✠☸