Ocelot Gateway

Release 25.0 ".NET 10"

Tom Gardham-Pallister, Raman Maksimchuk

Jun 22, 2026

WELCOME

1 Welcome **3**
1.1 Release Notes . . . 3
1.2 Contributing . . . 3

2 Big Picture **5**
2.1 Basic Implementation . . . 5
2.2 Multiple Instances . . . 6
2.3 With Consul . . . 6
2.4 With Service Fabric . . . 7

3 Getting Started **9**
3.1 Install . . . 9
3.2 Configuration . . . 9
3.3 Program . . . 10
3.4 Samples . . . 11

4 Not Supported **13**
4.1 Chunked Encoding . . . 13
4.2 Forwarding `Host` header . . . 13
4.3 Swagger . . . 13

5 Hosting Gotchas **15**
5.1 IIS . . . 15
5.2 Kestrel . . . 15

6 Administration **17**
6.1 Your Own IdentityServer . . . 17
6.2 Internal IdentityServer . . . 18
6.3 Administration API . . . 19

7 Aggregation **21**
7.1 Configuration . . . 21
7.2 Complex Aggregation . . . 22
7.3 Custom Aggregators . . . 24
7.4 Gotchas . . . 26

8 Authentication **27**
8.1 `AuthenticationOptions` Schema . . . 27
8.2 Single Authentication Scheme . . . 28
8.3 Multiple Authentication Schemes . . . 29
8.4 Configuration and `AllowAnonymous` . . . 29

8.5 Global Configuration . . . 30
8.6 Allowed Scopes . . . 31
8.7 JWT Tokens . . . 32
8.8 Identity Server Bearer Tokens . . . 32
8.9 Auth0 by Okta . . . 33
8.10 Warnings . . . 34
8.11 Links . . . 34
8.12 Roadmap . . . 34

9 Authorization **35**
9.1 Authorization Middleware . . . 35

10 Caching **37**
10.1 Install . . . 37
10.2 `CacheOptions` Schema . . . 37
10.3 Configuration . . . 38
10.4 `EnableContentHashing` option . . . 39
10.5 Global Configuration . . . 39
10.6 Custom Caching . . . 41
10.7 Roadmap . . . 41

11 Claims Transformation **43**
11.1 Claims to Claims . . . 43
11.2 Claims to Headers . . . 44
11.3 Claims to Query String Parameters . . . 44
11.4 Claims to Downstream Path . . . 44

12 Configuration **45**
12.1 Route Schema . . . 46
12.2 Dynamic Route Schema . . . 47
12.3 Aggregate Route Schema . . . 47
12.4 Global Configuration Schema . . . 48
12.5 Configuration Overview . . . 48
12.6 Multiple Environments . . . 49
12.7 Merging Files . . . 50
12.8 Reload On Change . . . 51
12.9 React to Changes . . . 52
12.10 Store in Consul . . . 53
12.11 Build From Scratch . . . 54
12.12 `HttpHandlerOptions` . . . 55
12.13 SSL Errors . . . 56
12.14 `DownstreamHttpVersion` . . . 57
12.15 Dependency Injection . . . 58
12.16 Extend with `Metadata` . . . 58
12.17 `Timeout` . . . 59

13 Delegating Handlers **63**
13.1 Configuration . . . 63
13.2 Execution Order . . . 64

14 Dependency Injection **65**
14.1 Services Overview . . . 65
14.2 `IServiceCollection` extensions . . . 65
14.3 `OcelotBuilder` class . . . 66
14.4 Custom Builder . . . 67

14.5 Configuration Overview . . . 68
14.6 IConfigurationBuilder extensions . . . 69

15 Error Handling 71
15.1 Middleware . . . 71
15.2 Client Error Responses . . . 71
15.3 Server Error Responses . . . 72
15.4 Error Mapper . . . 72

16 GraphQL 75
16.1 Sample . . . 75
16.2 Future . . . 75

17 Headers Transformation 77
17.1 Schema . . . 77
17.2 Configuration . . . 77
17.3 Find and Replace . . . 78
17.4 Add to Request . . . 78
17.5 Add to Response . . . 79
17.6 Placeholders . . . 79
17.7 Samples . . . 79
17.8 Roadmap . . . 80

18 Kubernetes (K8s) 1 83
18.1 Install . . . 83
18.2 AddKubernetes(bool) method . . . 83
18.3 AddKubernetes(Action<KubeClientOptions>) method . . . 85
18.4 Configuration . . . 86
18.5 Kube provider . . . 86
18.6 PollKube provider . . . 87
18.7 WatchKube provider . . . 87
18.8 Comparing providers . . . 88
18.9 Downstream Scheme vs Port Names . . . 88

19 Load Balancer 91
19.1 LoadBalancerOptions Schema . . . 91
19.2 Configuration . . . 91
19.3 Balancers . . . 94
19.4 CookieStickySessions Type . . . 94
19.5 Custom Balancers . . . 95

20 Logging 99
20.1 Warning . . . 99
20.2 Best Practices . . . 99
20.3 Request ID . . . 100
20.4 Performance . . . 104
20.5 Benchmarks . . . 104

21 Metadata 105
21.1 Schema . . . 105
21.2 Configuration . . . 106
21.3 GetMetadata<T> Method . . . 108
21.4 Sample . . . 109

22 Method Transformation 111

23 Middleware Injection **113**
23.1 `OcelotPipelineConfiguration` Class . . . 114
23.2 Ocelot Pipeline Builder . . . 116
23.3 Roadmap . . . 117

24 Quality of Service **119**
24.1 Implementations Overview . . . 119
24.2 Built-in QoS . . . 120
24.3 Installation (Polly) . . . 123
24.4 `QoSOptions` Schema . . . 123
24.5 Global Configuration . . . 125
24.6 Circuit Breaker strategy (Polly) . . . 127
24.7 Timeout strategy (Polly) . . . 128
24.8 Notes . . . 128
24.9 Extensibility (Polly) . . . 130

25 Rate Limiting **131**
25.1 `RateLimitOptions` Schema . . . 131
25.2 Configuration . . . 132
25.3 Notes . . . 134
25.4 Algorithms . . . 135
25.5 Rules (Partitions) . . . 135
25.6 Roadmap . . . 136

26 Routing **139**
26.1 Placeholders . . . 140
26.2 Catch All . . . 141
26.3 Priority . . . 142
26.4 Query Placeholders . . . 142
26.5 Upstream Host . . . 144
26.6 Upstream Headers . . . 145
26.7 Security Options . . . 146
26.8 Dynamic Routing . . . 146
26.9 Errors and Gotchas . . . 146

27 Service Discovery **149**
27.1 Consul . . . 149
27.2 Eureka . . . 155
27.3 Service Fabric . . . 156
27.4 Dynamic Routing . . . 156
27.5 Custom Providers . . . 160
27.6 Sample . . . 161

28 Service Fabric **163**
28.1 Configuration . . . 163
28.2 Placeholders . . . 164
28.3 Sample . . . 165

29 Tracing **167**
29.1 OpenTracing . . . 167
29.2 Butterfly . . . 168

30 Websockets **169**
30.1 Configuration . . . 169
30.2 Handy Links . . . 170

30.3 SignalR . . . 170
30.4 WebSocket Secure . . . 171
30.5 Supported . . . 171
30.6 Not Supported . . . 171
30.7 Sample . . . 172
30.8 Roadmap . . . 173

31 Building **175**
31.1 In IDE . . . 175
31.2 In terminal . . . 175
31.3 With Docker . . . 176
31.4 With CI/CD . . . 176
31.5 Documentation . . . 177
31.6 Testing . . . 177
31.7 SSL certificate . . . 178

32 Development Process **179**
32.1 Stages . . . 179
32.2 Notes . . . 180
32.3 Best Practices . . . 180
32.4 Dev Fun . . . 181

33 Release Process **183**
33.1 Stages . . . 183
33.2 Notes . . . 184
33.3 Quality Gates . . . 184

Thanks for taking a look at the Ocelot documentation! Please use the left hand **Navigation** sidebar to get around, or see the **Table of Contents** above.

The team recommends that newcomers to Ocelot's world start with the "*Introduction*" chapters. For seasoned fans of Ocelot with a Production environment, it is advised to always consult the *Release Notes* in the *Welcome* section before upgrading the app to the latest 25.0 version.

All **Features** are listed in alphabetical order. The primary features include *Configuration* and *Routing*.

Additional tips for building Ocelot can be found in the "*Building Ocelot*" section. We adhere to a *Development Process* which is a part of *Release Process*.

CHAPTER

ONE

WELCOME

Welcome to the Ocelot 25.0 documentation!

It is recommended to read all *Release Notes* if you have deployed the Ocelot app in a production environment and are planning to upgrade to major, minor or patched versions.

1.1 Release Notes

Release Tag: 25.0.0
Release Codename: .NET 10

1.2 Contributing

Pull requests, issues, and commentary are welcome at the Ocelot GitHub repository. For ideas and questions, please post them in the Ocelot Discussions space.

Our *Development Process* is a part of successful *Release Process*. If you are a new contributor, it is crucial to read *Development Process* attentively to grasp our methods for efficient and swift feature delivery. We, as a team, advocate adhering to *Best Practices* throughout the development phase.

We extend our best wishes for your successful contributions to the Ocelot product!

CHAPTER

TWO

BIG PICTURE

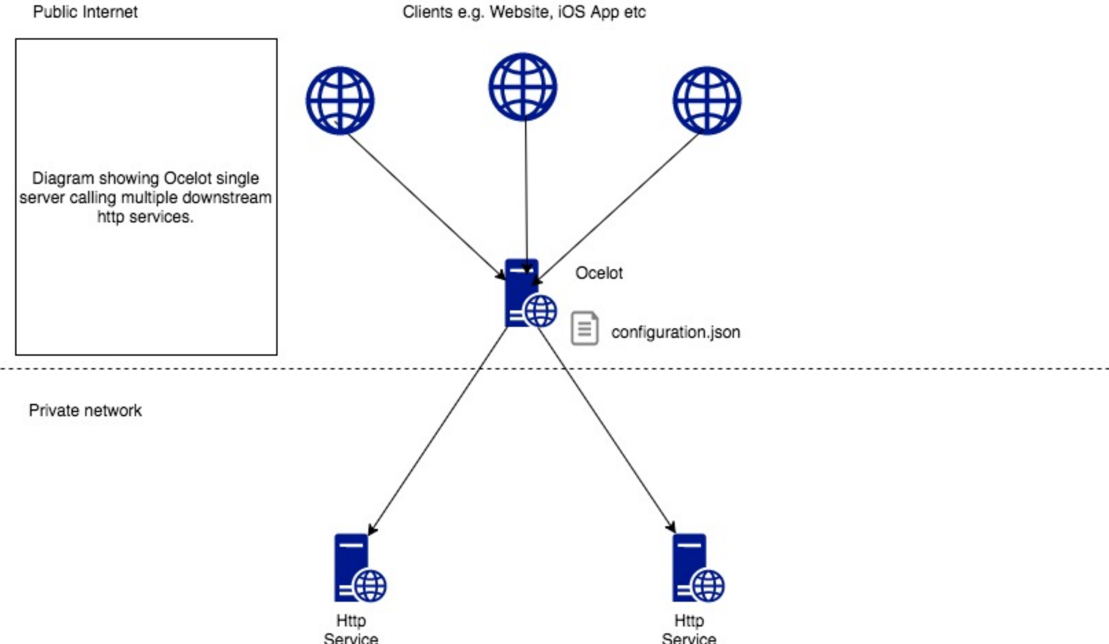
Ocelot is aimed at people using .NET running a microservices (service-oriented) architecture (aka SOA) that need a unified point of entry into their system. However it will work with anything that speaks HTTP(S) and run on any platform that ASP.NET Core supports.

Ocelot consists of a series of ASP.NET Core middlewares arranged in a specific order.

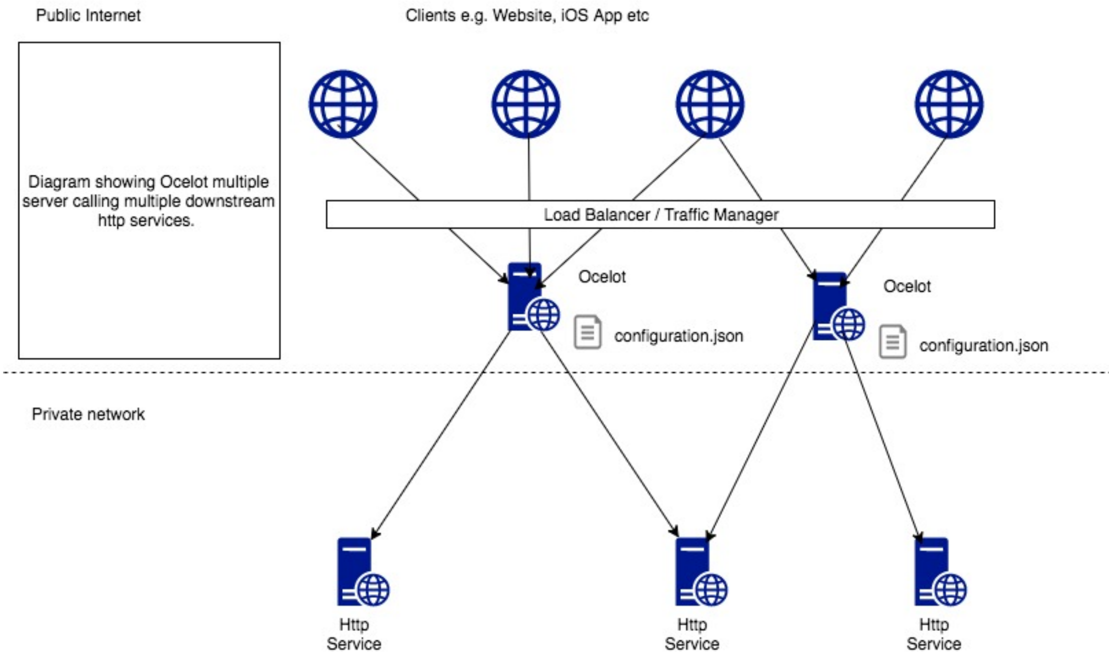
Ocelot manipulates the `HttpRequest` object into a state specified by its configuration until it reaches a request builder middleware, where it creates a `HttpRequestMessage` object which is used to make a request to a downstream service. The middleware that makes the request is the last thing in the Ocelot pipeline. It does not call the next middleware. The response from the downstream service is retrieved as the request goes back up the Ocelot pipeline. There is a piece of middleware that maps the `HttpResponseMessage` onto the `HttpResponse` object, and that is returned to the client. That is basically it with a bunch of other features!

The following are configurations that you use when deploying Ocelot.

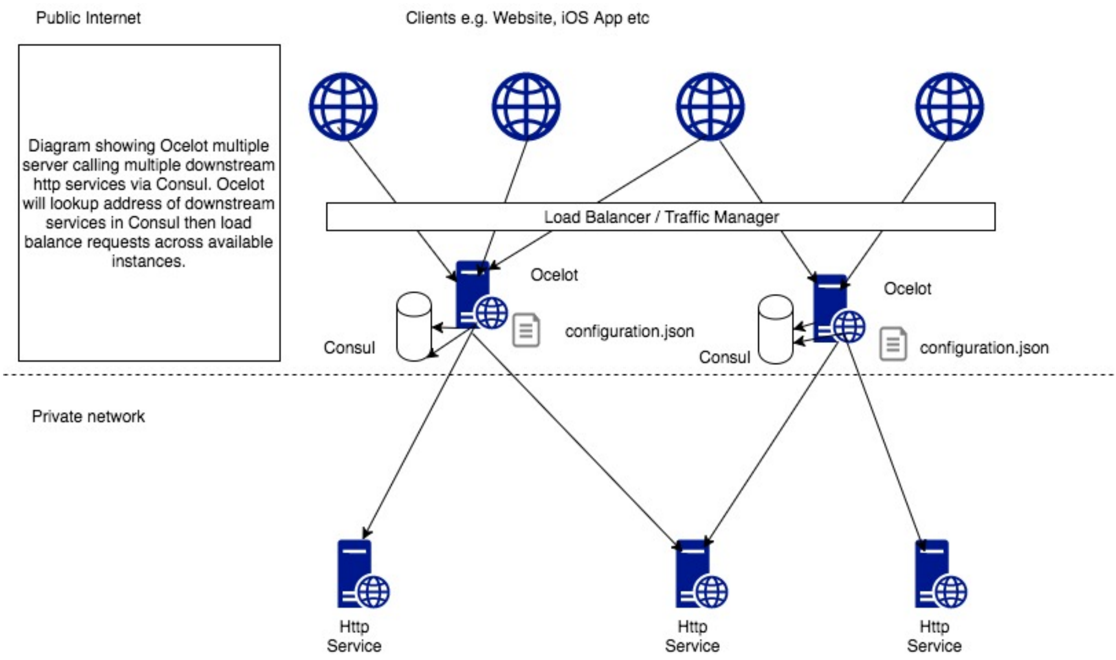
2.1 Basic Implementation



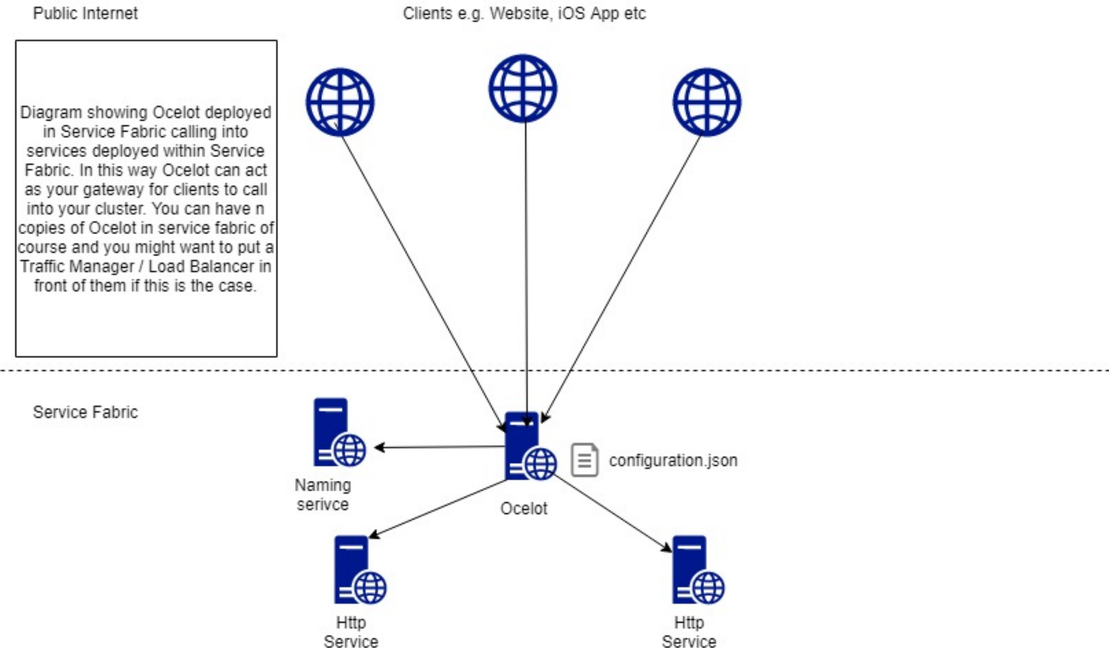
2.2 Multiple Instances



2.3 With Consul



2.4 With Service Fabric



CHAPTER

THREE

GETTING STARTED

Ocelot is designed to work with ASP.NET Core and is currently on .NET 8 LTS and .NET 9 STS frameworks.

3.1 Install

Install Ocelot and it's dependencies using NuGet. You will need to create a ASP.NET Core minimal API project with "ASP.NET Core Empty" template but without `app.Map*` methods, and bring the package into it. Then follow the startup below and *Configuration* sections to get up and running.

```
Install-Package Ocelot
```

All versions can be found in the NuGet Gallery | Ocelot.

3.2 Configuration

The following is a very basic ocelot.json. It won't do anything but should get Ocelot starting.

```
{
  "Aggregates": [], // optional
  "Routes": [], // required section
  "DynamicRoutes": [], // optional section
  "GlobalConfiguration": { // required
    "BaseUrl": "https://api.mybusiness.com"
  }
}
```

If you want some example that actually does something use the following:

```
{
  "Routes": [
    {
      "UpstreamHttpMethod": [ "Get" ],
      "UpstreamPathTemplate": "/ocelot/posts/{id}",
      "DownstreamPathTemplate": "/todos/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        { "Host": "jsonplaceholder.typicode.com", "Port": 443 }
      ]
    }
  ],
```

(continues on next page)

(continued from previous page)

```
  "GlobalConfiguration": {
    "BaseUrl": "https://api.mybusiness.com"
  }
}
```

The most important thing to note here is `BaseUrl` property. Ocelot needs to know the URL it is running under in order to do Header *Find and Replace 2* and for certain *Administration* configurations. When setting this URL it should be the external URL that clients will see Ocelot running on e.g. If you are running containers Ocelot might run on the URL `http://123.12.1.2:6543` but has something like nginx in front of it responding on `https://api.mybusiness.com`. In this case the Ocelot `BaseUrl` should be `https://api.mybusiness.com`.

If you are using containers and require Ocelot to respond to clients on `http://123.12.1.2:6543` then you can do this, however if you are deploying multiple Ocelot's you will probably want to pass this on the command line in some kind of script. Hopefully whatever scheduler you are using can pass the IP.

3.3 Program

Then in your Program.cs (with top-level statements) you will want to have the following.

```
using Ocelot.DependencyInjection;
using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

// Ocelot Basic setup
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddOcelot(); // single ocelot.json file in read-only mode
builder.Services
    .AddOcelot(builder.Configuration);

// Add your features
if (builder.Environment.IsDevelopment())
{
    builder.Logging.AddConsole();
}

// Add middlewares aka app.Use*()
var app = builder.Build();
await app.UseOcelot();
await app.RunAsync();
```

The main things to note are

- `builder.Configuration.AddOcelot()` adds single ocelot.json configuration file in read-only mode.
- `builder.Services.AddOcelot(builder.Configuration)` adds Ocelot required and default services[1]

[1] The *AddOcelot method* adds default ASP.NET services to the DI container. You can call another extended *AddOcelotUsingBuilder method* while configuring services to develop your own *Custom Builder*. See more instructions in the "*AddOcelotUsingBuilder method*" section of the *Dependency Injection* feature.

- `app.UseOcelot()` sets up all the Ocelot middlewares. Note, we have to await the threading result before calling `app.RunAsync()`
- Do not add endpoint mappings (minimal API methods) such as `app.MapGet()` because the Ocelot pipeline is not compatible with them!

3.4 Samples

Solution: Ocelot.Samples.sln

For beginners, we have prepared basic samples to help Ocelot newbies clone, compile, and get it running.

- Basic sample: It has a single configuration file, ocelot.json.
- Basic Configuration sample: It has multiple configuration files (`ocelot.*.json`) to be merged into `ocelot.json` and written back to disk.

After running in Visual Studio[2], you may use `API.http` files to send testing requests to the `localhost` Ocelot application instance.

[2] All *Samples* projects are organized as the Ocelot.Samples.sln file for Visual Studio 2022 IDE.

CHAPTER

FOUR

NOT SUPPORTED

Ocelot does not support…

4.1 Chunked Encoding

Ocelot will always get the body size and return Content-Length header. Sorry, if this doesn't work for your use case!

4.2 Forwarding `Host` header

The Host header that you send to Ocelot will not be forwarded to the downstream service. Obviously this would break everything.

4.3 Swagger

Contributors have looked multiple times at building `swagger.json` out of the Ocelot `ocelot.json` but it doesnt fit into the vision the team has for Ocelot. If you would like to have Swagger in Ocelot then you must roll your own `swagger.json` and do the following in your Program.cs. The code sample below registers a piece of middleware that loads your hand rolled `swagger.json` and returns it on `/swagger/v1/swagger.json`. It then registers the SwaggerUI middleware from Swashbuckle.AspNetCore package:

```
var builder = WebApplication.CreateBuilder(args);
// ...
var app = builder.Build();
app.Map("/swagger/v1/swagger.json", builder =>
    builder.Run(async context => {
        var json = await File.ReadAllTextAsync("swagger.json");
        await context.Response.WriteAsync(json);
    }));
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Ocelot");
});

await app.UseOcelot();
await app.RunAsync();
```

The main reasons why we don't think Swagger makes sense is we already hand roll our definition in `ocelot.json`. If we want people developing against Ocelot to be able to see what routes are available

then either share the `ocelot.json` with them (This should be as easy as granting access to a repo etc) or use the Ocelot *Administration* API so that they can query Ocelot for the configuration.

In addition to this, many people will configure Ocelot to proxy all traffic like `/products/{everything}` to their product service and you would not be describing what is actually available if you parsed this and turned it into a Swagger path. Also Ocelot has no concept of the models that the downstream services can return and linking to the above problem the same endpoint can return multiple models. Ocelot does not know what models might be used in POST, PUT etc, so it all gets a bit messy, and finally, the Swashbuckle package **does not** reload `swagger.json` if it changes during runtime. Ocelot's configuration can change during runtime so the Swagger and Ocelot information would not match. Unless we rolled our own Swagger implementation.

If the developer wants something to easily test against the Ocelot API then we suggest using Postman as a simple way to do this. It might even be possible to write something that maps `ocelot.json` to the Postman JSON spec. However we do not intend to do this.

CHAPTER FIVE

HOSTING GOTCHAS

Microsoft Learn: Web server implementations in ASP.NET Core

Many errors and incidents (gotchas) are related to web server hosting scenarios. Please review deployment and web hosting common user scenarios below depending on your web server.

5.1 IIS

Repository Label: IIS
Microsoft Learn: Host ASP.NET Core on Windows with IIS

We **do not** recommend to deploy Ocelot app to IIS environments, but if you do, keep in mind the gotchas below.

- When using ASP.NET Core 2.2+ and you want to use In-Process hosting, replace `UseIISIntegration()` with `UseIIS()`, otherwise you will get startup errors.
- Make sure you use Out-of-process hosting model instead of In-process one (see Out-of-process hosting with IIS and ASP.NET Core), otherwise you will get very slow responses (see 1657).
- Ensure all DNS servers of all downstream hosts are online and they function perfectly, otherwise you will get slow responses (see 1630).

The community constanly reports issues related to IIS. If you have some troubles in IIS environment to host Ocelot app, first of all, read open/closed issues, and after that, search for IIS-related objects in the repository. Probably you will find a ready solution by Ocelot community members.

Finally, there is the special IIS label for all IIS-related objects. Feel free to put this label onto issues, pull requests, discussions, etc.

5.2 Kestrel

Repository Label: Kestrel
Microsoft Learn: Kestrel web server in ASP.NET Core

We **do** recommend to deploy Ocelot app to self-hosting environments, aka Kestrel vs Docker. We try to optimize Ocelot web app for Kestrel & Docker hosting scenarios, but keep in mind the following gotchas.

1. Upload and download large files[1]

This is proxying the large content through the gateway: when you pump large (static) files using the gateway. We believe that your client apps should have direct integration to (static) files persistent storages and services: remote & destributed file systems, CDNs, static files & blob storages, etc. We **do not** recommend

[1] Large files pumping is stabilized and available as complete solution starting in 23.0 release. We believe our PRs 1724, 1769 helped to resolve the issues and stabilize large content proxying problems of 22.0.1 version and lower.

to pump large files (100Mb+ or even larger 1GB+) using gateway because of performance reasons: consuming memory and CPU, long delay times, producing network errors for downstream streaming, impact on other routes.

The community constanly reports issues related to large files, `application/octet-stream` content type, *Chunked Encoding*, etc., see issues 749, 1472.

If you still want to pump large files through an Ocelot gateway instance, use 23.0 version and higher.

In case of some errors, see the next point.

2. Maximum request body size

Docs: Maximum request body size | Configure options for the ASP.NET Core Kestrel web server.

ASP.NET `HttpRequest` behaves erroneously for application instances that do not have their Kestrel MaxRequestBodySize option configured correctly and having pumped large files of unpredictable size which exceeds the limit.

As a quick fix, use this configuration recipe:

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    int myVideoFileMaxSize = 1_073_741_824; // assume your file storage has␣
↪max file size as 1 GB (1_073_741_824)
    int totalSize = myVideoFileMaxSize + 26_258_176; // and add some extra size
    serverOptions.Limits.MaxRequestBodySize = totalSize; // 1_100_000_000 thus␣
↪1 GB file should not exceed the limit
});
```

Finally, there is the special Kestrel label for all Kestrel-related objects. Feel free to put this label onto issues, pull requests, discussions, etc.

CHAPTER SIX

ADMINISTRATION

Ocelot extension package: Ocelot.Administration.IdentityServer4 with integrated IdentityServer4 package by IdentityServer org (archived on March 6, 2025)

Ocelot supports changing configuration during runtime via an authenticated HTTP API. This can be authenticated in two ways either using Ocelot's internal IdentityServer (for authenticating requests to the *Administration API* only) or hooking the *Administration API* authentication into your own IdentityServer.

The first thing you need to do if you want to use the *Administration API* is bring in the relevant Ocelot.Administration.IdentityServer4 package:

```
NuGet\Install-Package Ocelot.Administration.IdentityServer4
dotnet add package Ocelot.Administration.IdentityServer4
```

This will bring down everything needed by the *Administration API*.

Warning! Currently, the *Administration* feature relies solely on the IdentityServer4 package, whose repository was archived by its owner on July 31, 2024 (for the first time) and again on March 6, 2025. In release 24.0, the Ocelot team deprecated the Ocelot.Administration.IdentityServer4 extension package. However, the repository remains available, allowing for potential patches.

6.1 Your Own IdentityServer[1]

All you need to do to hook into your own IdentityServer is add the following configuration options with authentication to your Program. After that, we must pass these options to the `AddAdministration()` extension of the `OcelotBuilder` being returned by `AddOcelot()`[2], as shown below:

```
Action<JwtBearerOptions> options = o =>
{
    o.Authority = "https://identity-server-host:3333";
    o.RequireHttpsMetadata = true; // false in development environment
    o.TokenValidationParameters = new()
    {
        ValidateAudience = false,
    };
    //...
};
builder.Services
```

(continues on next page)

[1] The "*Your Own IdentityServer*" feature was implemented for issue 228.

[2] The *AddOcelot method* adds default ASP.NET services to the DI container. You can call another extended *AddOcelotUsingBuilder method* while configuring services to develop your own *Custom Builder*. See more instructions in the "*AddOcelotUsingBuilder method*" section of the *Dependency Injection* feature.

(continued from previous page)

```
    .AddOcelot(builder.Configuration)
    .AddAdministration("/administration", options);
```

You now need to get a token from your IdentityServer and use in subsequent requests to Ocelot's *Administration API*.

> **Note**: This feature is useful because the IdentityServer authentication middleware needs the URL of the server. If you are using the *Internal IdentityServer*, it might not always be possible to have the Ocelot URL.

6.2 Internal IdentityServer

The API is authenticated using Bearer tokens that you request from Ocelot itself. This is provided by the amazing IdentityServer project that the .NET community has been using for several years. Check it out.

In order to enable the administration section, you need to do a few things. First of all, add this to your initial Program.

The path can be anything you want and it is obviously recommended don't use a URL you would like to route through with Ocelot as this will not work. The administration uses the `MapWhen` functionality of ASP.NET Core and all requests to `{root}/administration` will be sent there not to the Ocelot middleware.

The secret is the client secret that Ocelot's internal IdentityServer will use to authenticate requests to the *Administration API*. This can be whatever you want it to be! In order to pass this secret string as parameter, we must call the `AddAdministration()` extension of the `OcelotBuilder` being returned by `AddOcelot()`[Page 17, 2], as shown below:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddAdministration("/administration", "secret");
```

In order for the *Administration API* to work, Ocelot and IdentityServer must be able to call themselves for validation. This means that you need to add the base URL of Ocelot to the global configuration if it is not the default `http://localhost:5000`.

> **Note**: If you are using something like Docker to host Ocelot, it might not be able to call back to `localhost`, etc., and you need to know what you are doing with Docker networking in this scenario.

Configuration can be done as follows:

- If you want to run on a different host and port locally:

```
"GlobalConfiguration": {
  "BaseUrl": "http://localhost:5580"
}
```

- or if Ocelot is exposed via DNS:

```
"GlobalConfiguration": {
  "BaseUrl": "http://mydns.net"
}
```

Now, if you went with the configuration options above and want to access the API, you can use the Postman scripts called Ocelot.postman_collection.json in the solution to change the Ocelot configura-

tion. Obviously these will need to be changed if you are running Ocelot on a different URL to `http://localhost:5000`.

The scripts show you how to request a Bearer token from Ocelot and then use it to GET the existing configuration and POST a configuration.

If you are running multiple Ocelot instances in a cluster then you need to use a certificate to sign the Bearer tokens used to access the *Administration API*.

In order to do this, you need to add two more environmental variables for each Ocelot in the cluster:

1. `OCELOT_CERTIFICATE`: The path to a certificate that can be used to sign the tokens. The certificate needs to be of the type X509 and obviously Ocelot needs to be able to access it.
2. `OCELOT_CERTIFICATE_PASSWORD`: The password for the certificate.

Normally Ocelot just uses temporary signing credentials but if you set these environmental variables then it will use the certificate. If all the other Ocelot instances in the cluster have the same certificate then you are good!

6.3 Administration API

- **POST** `{adminPath}/connect/token`

  This gets a token for use with the admin area using the client credentials we talk about setting above. Under the hood this calls into an IdentityServer hosted within Ocelot.

  The body of the request is form-data as follows:

  - `client_id` set as admin
  - `client_secret` set as whatever you used when setting up the administration services.
  - `scope` set as admin
  - `grant_type` set as client_credentials

- **GET** `{adminPath}/configuration`

  This gets the current Ocelot configuration. It is exactly the same JSON we use to set Ocelot up with in the first place.

- **POST** `{adminPath}/configuration`

  This overwrites the existing configuration (should probably be a PUT!). We recommend getting your config from the GET endpoint, making any changes and posting it back... simples.

  The body of the request is JSON and it is the same format as the FileConfiguration that we use to set up Ocelot on a file system.

  Please note, if you want to use this API then the process running Ocelot must have permission to write to the disk where your `ocelot.json` or `ocelot.{environment}.json` is located. This is because Ocelot will overwrite them on save.

- **DELETE** `{adminPath}/outputcache/{region}`

  This clears a region of the cache. If you are using a backplane, it will clear all instances of the cache! Giving your the ability to run a cluster of Ocelots and cache over all of them in memory and clear them all at the same time, so just use a distributed cache.

  The region is whatever you set against the `Region` field in the FileCacheOptions section of the Ocelot configuration.

CHAPTER

SEVEN

AGGREGATION

Aggregation, also known as HTTP response data aggregation, is a well-known Backend for Frontend pattern of Microservices architecture.

- Backend for Frontend (BFF) Pattern: Microservices for UX | Teleport Academy
- Gateway Aggregation pattern | Azure Architecture Center | Microsoft Learn
- Backends for Frontends pattern | Azure Architecture Center | Microsoft Learn
- Implement API Gateways with Ocelot | .NET microservices - Architecture e-book | Microsoft Learn

Ocelot allows you to specify *Aggregate Routes*[1] that combine multiple normal routes and map their responses into a single object. This is particularly useful when a client is making multiple requests to a server that could be consolidated into one. This feature supports the implementation of a Backend for Frontend (BFF) architecture using Ocelot.

7.1 Configuration

In order to set this up, you need to configure the ocelot.json file as follows. In this example, two normal routes are specified, each having a Key property. An *aggregation* is then defined, which combines the two routes using their keys listed in RouteKeys, and the UpstreamPathTemplate is set up to function like a normal route.

> Note that duplicate UpstreamPathTemplates are not allowed between Routes and Aggregates. You can use all of Ocelot's normal route options, except for RequestIdKey, as explained in the *Gotchas* section.

```
{
  "Routes": [
    {
      "UpstreamHttpMethod": [ "Get" ],
      "UpstreamPathTemplate": "/laura",
      "DownstreamPathTemplate": "/",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 51881 }
      ],
      "Key": "Laura"
    },
    {
```

(continues on next page)

[1] This feature was requested as part of issue 79, and further improvements were made as part of issue 298. A significant refactoring and revision of the Multiplexer design was carried out on March 4, 2024, in version 23.1. See pull requests 1462 and 1826 for more details.

(continued from previous page)

```
      "UpstreamHttpMethod": [ "Get" ],
      "UpstreamPathTemplate": "/tom",
      "DownstreamPathTemplate": "/",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 51882 }
      ],
      "Key": "Tom"
    }
  ],
  "Aggregates": [
    {
      "UpstreamPathTemplate": "/",
      "RouteKeys": [ "Tom", "Laura" ]
    }
  ]
}
```

You can also set `UpstreamHost` and `RouteIsCaseSensitive` in the *aggregation* configuration. These settings behave the same as in other routes.

If the route `/tom` returned a body of `{"Age": 19}` and `/laura` returned `{"Age": 25}`, the response after *aggregation* would be as follows:

```
{"Tom":{"Age": 19},"Laura":{"Age": 25}}
```

At the moment, the *aggregation* is quite simple. Ocelot retrieves the response from your downstream service and inserts it into a JSON dictionary, as shown above. The route `Key` becomes the key of the dictionary, and the response body from your downstream service serves as the value. The resulting object is plain JSON without any formatting or additional spaces.

> **Note 1**: All headers will be lost from the downstream service's response.
>
> **Note 2**: Ocelot will always return the content type `application/json` for an aggregate request.
>
> **Note 3**: If your downstream services return a `404` Not Found, the aggregate will simply return nothing for that downstream service. It will not change the aggregate response to a `404`, even if all the downstream services return a `404`.

7.2 Complex Aggregation[2]

Imagine you would like to use aggregated queries but don't have all the parameters for your queries. First, you need to call an endpoint to obtain the necessary data, such as a user's ID, and then return the user's details.

Let's say we have an endpoint that returns a series of comments referencing various users or threads. The author of the comments is identified by their ID, but you want to return all the details about the author.

Here, you could use aggregation to: 1) retrieve all the comments, and 2) attach the author details. In fact, two endpoints are called, but for the second, you dynamically replace the user's ID in the route to obtain the details.

In concrete terms:

[2] The "*Complex Aggregation*" feature is still in its early stages, but it enables searching for data based on an initial request. This feature was requested as part of issue 661, introduced in pull request 704, and released in version 13.4.

1) `/Comments` contains the `authorId` property.
2) `/users/{userId}`, with `{userId}` replaced by `authorId`, is used to obtain the user's details.

To perform the mapping, you need to use the `RouteKeysConfig` list of configuration options for aggreagte route, typed as `AggregateRouteConfig` class:

```
"RouteKeysConfig": [
  {
    "RouteKey": "UserDetails",
    "JsonPath": "$[*].authorId",
    "Parameter": "userId"
  }
]
```

`RouteKey` is used as a reference for the route, `JsonPath` indicates where the parameter of interest is located in the first request's response body, and `Parameter` specifies that the value for `authorId` should be used as the request parameter `userId`.

The final configuration is as follows:

```
{
  "Routes": [
    {
      "UpstreamPathTemplate": "/Comments",
      "DownstreamPathTemplate": "/",
      // ...
      "Key": "Comments"
    },
    {
      "UpstreamPathTemplate": "/UserDetails/{userId}",
      "DownstreamPathTemplate": "/users/{userId}",
      // ...
      "Key": "UserDetails"
    },
    {
      "UpstreamPathTemplate": "/PostDetails/{postId}",
      "DownstreamPathTemplate": "/posts/{postId}",
      // ...
      "Key": "PostDetails"
    }
  ],
  "Aggregates": [
    {
      "UpstreamPathTemplate": "/",
      "UpstreamHost": "localhost",
      "RouteKeys": [ "Comments", "UserDetails", "PostDetails" ],
      "RouteKeysConfig": [
        { "RouteKey": "UserDetails", "JsonPath": "$[*].writerId", "Parameter":
↪"userId" },
        { "RouteKey": "PostDetails", "JsonPath": "$[*].postId", "Parameter":
↪"postId" }
      ]
    }
  ]
```

(continues on next page)

(continued from previous page)

```
}
```

7.3 Custom Aggregators

Ocelot started with basic request *aggregation*, and since then, a more advanced method has been added. This method allows the user to take the responses from downstream services and aggregate them into a response object. The ocelot.json setup is almost identical to the basic *aggregation* approach, except that you need to add an `Aggregator` property, as shown below:

```
{
  "Routes": [
    {
      "UpstreamPathTemplate": "/laura",
      "DownstreamPathTemplate": "/",
      // ...
      "Key": "Laura"
    },
    {
      "UpstreamPathTemplate": "/tom",
      "DownstreamPathTemplate": "/",
      // ...
      "Key": "Tom"
    }
  ],
  "Aggregates": [
    {
      "UpstreamPathTemplate": "/",
      "RouteKeys": [ "Tom", "Laura" ],
      "Aggregator": "MyAggregator"
    }
  ]
}
```

Here, we have added an aggregator called `MyAggregator`. Ocelot will look for this aggregator when it tries to aggregate this route.

In order to make the aggregator available in Ocelot Core, we must add the `MyAggregator` to the `OcelotBuilder` returned by `AddOcelot()`[3], as shown below:

```
using Ocelot.Multiplexer;

builder.Services
    .AddOcelot(builder.Configuration)
    .AddSingletonDefinedAggregator<MyAggregator>();
```

Now, when Ocelot tries to aggregate the route above, it will find the `MyAggregator` in the DI-container and use it to aggregate the route. Since the `MyAggregator` is registered in the DI-container, you can add any dependencies it needs to the container, as shown below:

[3] The *AddOcelot method* adds default ASP.NET services to the DI container. You can call another extended *AddOcelotUsingBuilder method* while configuring services to develop your own *Custom Builder*. See more instructions in the "*AddOcelotUsingBuilder method*" section of the *Dependency Injection* feature.

```
builder.Services
    .AddSingleton<MyDependency>();
// ...
builder.Services
    .AddOcelot(builder.Configuration)
    .AddSingletonDefinedAggregator<MyAggregator>();
```

In this example, `MyAggregator` depends on `MyDependency`, and it will be resolved by the DI container. In addition to this, Ocelot lets you add transient aggregators, as shown below:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddTransientDefinedAggregator<MyAggregator>();
```

In order to create an *aggregator*, you must implement the following interface:

```
public interface IDefinedAggregator
{
    Task<DownstreamResponse> Aggregate(List<HttpContext> responses);
}
```

With this feature, you can essentially do whatever you want, as the `HttpContext` objects contain the results of all the aggregate requests.

> Please note that if the `HttpClient` throws an exception when making a request to a route in the aggregate, you will not receive a `HttpContext` for it. However, you will receive one for any that succeed. If an exception is thrown, it will be logged.

Below is an example of an *aggregator* that can be implemented for your solution:

```
public class MyAggregator : IDefinedAggregator
{
    public async Task<DownstreamResponse> Aggregate(List<HttpContext>␣
↪responseHttpContexts)
    {
        // The aggregator gets a list of downstream responses as parameter.
        // You can now implement your own logic to aggregate the responses␣
↪(including bodies and headers) from the downstream services
        var responses = responseHttpContexts.Select(x => x.Items.
↪DownstreamResponse()).ToArray();

        // In this example we are concatenating the results,
        // but you could create a more complex construct, up to you.
        var contentList = new List<string>();
        foreach (var response in responses)
        {
            var content = await response.Content.ReadAsStringAsync();
            contentList.Add(content);
        }

        // The only constraint here: You must return a DownstreamResponse␣
↪object.
        return new DownstreamResponse(
            new StringContent(JsonConvert.SerializeObject(contentList)),
```

(continues on next page)

(continued from previous page)

```
            HttpStatusCode.OK,
            responses.SelectMany(x => x.Headers).ToList(),
            "reason");
    }
}
```

7.4 Gotchas

- You cannot use routes with specific `RequestIdKeys`, as this would be overly complicated to track.
- *Aggregation* supports only the `GET` HTTP verb.
- *Aggregation* allows the forwarding of `HttpRequest.Body` to downstream services by duplicating the body data. Form data and attached files should also be forwarded. It is essential to specify the `Content-Length` header in requests to the upstream; otherwise, Ocelot will log warnings such as: *"Aggregation does not support body copy without a Content-Length header!"*

CHAPTER

EIGHT

AUTHENTICATION

In order to authenticate routes and subsequently use any of Ocelot's claims based features such as authorization or modifying the request with values from the token, users must register authentication services in their Program as usual but they provide a scheme (authentication provider key) with each registration e.g.

```
const string AuthenticationProviderKey = "MyKey"; // aka scheme
builder.Services
    .AddAuthentication()
    .AddJwtBearer(AuthenticationProviderKey, options =>
    {
        // authentication setup via options initialization
    });
```

In this example, `MyKey` is the scheme with which this provider has been registered, but for JWT bearer authentication, the scheme is usually `Bearer`. We then map this to a route in the configuration using the following *AuthenticationOptions Schema* options:

- `AuthenticationProviderKey` is a string, the legacy definition of *Single Authentication Scheme*.
- `AuthenticationProviderKeys` is an array of strings, the recommended definition of *Multiple Authentication Schemes* feature.

8.1 AuthenticationOptions Schema

Class: FileAuthenticationOptions

The following is the full *authentication* configuration, used in both the *Route Schema* and the *Dynamic Route Schema*. Not all of these options need to be configured; however, the `AuthenticationProviderKeys` option is mandatory when `AuthenticationProviderKey` is absent.

```
"AuthenticationOptions": {
  "AllowAnonymous": false, // nullable boolean
  "AllowedScopes": [], // array of strings
  "AuthenticationProviderKey": "", // deprecated! -> use␣
↪AuthenticationProviderKeys
  "AuthenticationProviderKeys": [] // array of strings
}
```

Option	*Description*
AllowAnonymous	Excludes a route from global *authentication options* by setting it to true. If the global option disables authentication by forcibly having a true value, then at the route level the option can include a route to be authenticated by setting it to false. For more details, refer to the "*Configuration and AllowAnonymous*" section.
AllowedScopes	If specified, enables authorization based on the scope claim after successful authentication by a configured authentication provider. For more details, refer to the "*Allowed Scopes*" section.
AuthenticationProvi	Maps a configured authentication provider, identified by a key (scheme), to a route that requires authentication. *Note: This option is deprecated—see the warning below.* For more details, refer to the "*Single Authentication Scheme*" section.
AuthenticationProvi	Maps all configured authentication providers, identified by their schemes, to a route that requires authentication. For more details, refer to the "*Multiple Authentication Schemes*" section.

Warning

The AuthenticationProviderKey option is deprecated in version 24.1! Use the AuthenticationProviderKeys array option instead. Note that AuthenticationProviderKey will be removed in version 25.0. For backward compatibility in version 24.1, the AuthenticationProviderKey option takes precedence over the schemes in the AuthenticationProviderKeys array. If the AuthenticationProviderKey scheme provider fails, the remaining schemes in the AuthenticationProviderKeys array will enforce the appropriate authentication providers in the specified order.

8.2 Single Authentication Scheme[1]

Option: AuthenticationProviderKey

We map authentication provider to a Route in the configuration e.g.

```
"AuthenticationOptions": {
  "AuthenticationProviderKey": "MyKey",
  "AllowedScopes": []
}
```

When Ocelot runs it will look at this routes AuthenticationProviderKey and check that there is an authentication provider registered with the given key. If there isn't then Ocelot will not start up. If there is then the route will use that provider when it executes.

If a route is authenticated, Ocelot will invoke whatever scheme is associated with it while executing the authentication middleware. If the request fails authentication, Ocelot returns a HTTP status code 401 Unauthorized.

[1] The "*Single Authentication Scheme*" feature has been an Ocelot artifact for ages. Use the AuthenticationProviderKeys property instead of AuthenticationProviderKey one. We support this [Obsolete] property for backward compatibility and migration reasons. In future releases, the property may be removed as a breaking change.

8.3 Multiple Authentication Schemes[2]

Option: AuthenticationProviderKeys

In the real world of ASP.NET Core, apps may need to support multiple types of authentication by a single Ocelot app instance. To register multiple authentication schemes (authentication provider keys) for each appropriate authentication provider, use and develop this abstract configuration of two or more schemes:

```
var DefaultScheme = JwtBearerDefaults.AuthenticationScheme; // Bearer
builder.Services
    .AddAuthentication()
    .AddJwtBearer(DefaultScheme, options => { /* JWT setup */ })
    // AddJwtBearer, AddCookie, AddIdentityServerAuthentication etc.
    .AddMyProvider("MyKey", options => { /* Custom auth setup */ });
```

In this example, the MyKey and Bearer schemes represent the keys with which these providers were registered. We then map these schemes to a route in the configuration as shown below.

```
"AuthenticationOptions": {
  "AuthenticationProviderKeys": [ "Bearer", "MyKey" ] // The order matters!
  "AllowedScopes": []
}
```

Afterward, Ocelot applies all steps that are specified for AuthenticationProviderKey as *Single Authentication Scheme*. The order of the keys in an array definition does matter! We use a "First One Wins" authentication strategy.

8.4 Configuration and AllowAnonymous[3]

To configure *authentication options* uniformly across all static routes, define them in GlobalConfiguration section using the *AuthenticationOptions Schema*. If *authentication options* are specified in both GlobalConfiguration and a route (i.e., AuthenticationProviderKey or AuthenticationProviderKeys are set), the route-level configuration takes precedence.

Excluding a route from global *authentication options* is possible by setting AllowAnonymous option to true. This prevents the route from requiring authentication, keeping it open and anonymous.

In the following example:

- The first route is authenticated using the MyGlobalKey provider's scheme.
- The second route uses the MyKey provider's scheme.
- The third route is not authenticated.

```
"Routes": [
  {
    // route #1 props...
    "AuthenticationOptions": {}
  },
  {
    // route #2 props...
    "AuthenticationOptions": {
```

(continues on next page)

[2] The "*Multiple Authentication Schemes*" feature was requested in issues 740, 1580 and delivered as a part of 23.0 release.

[3] The global "*Configuration and AllowAnonymous*" feature for static routes was requested in issues 842 and 1414, implemented in pull request 2114, and officially released in version 24.1.

(continued from previous page)

```
      "AuthenticationProviderKeys": [ "MyKey" ],
      "AllowedScopes": [ "Bob" ]
    }
  },
  {
    // route #3 props...
    "AuthenticationOptions": {
      "AllowAnonymous": true
    }
  }
],
"GlobalConfiguration": {
  "BaseUrl": "http://ocelot.net",
  "AuthenticationOptions": {
    "RouteKeys": [], // empty -> no grouping, thus opts will apply to all␣
↪routes
    "AuthenticationProviderKeys": [ "MyGlobalKey" ],
    "AllowedScopes": [ "Admin" ]
  }
}
```

Note: Ocelot performs a per-option merging algorithm to combine route and global `AuthenticationOptions`. If global `AuthenticationProviderKeys` are defined together with global `AllowedScopes`, then route options should be specified as a pair of scheme and scopes; otherwise, a scope should not belong to the global authentication provider. Moreover, the route scopes array entirely overrides the global scopes array, so the two collections are not merged but rather interchangeable.

8.5 Global Configuration[4]

Since the global configuration for static routes has already been described above, here are additional details regarding dynamic routes, whose configuration was not supported in versions prior to 24.1. Starting with version 24.1, global and route *authentication options* for *Dynamic Routing* were introduced. These global options may also be overridden in the `DynamicRoutes` configuration section, as defined by the *Dynamic Route Schema*.

```
{
  "DynamicRoutes": [
    {
      "Key": "R1", // optional
      "ServiceName": "my-service",
      "AuthenticationOptions": {
        "AuthenticationProviderKeys": ["MyKey"], // custom authentication␣
↪provider
        "AllowedScopes": ["my-service"] // require authorization with a 'scope'␣
↪claim set to the value 'my-service'
      }
    }
  ],
```

(continues on next page)

[4] The "*Global Configuration*" feature for dynamic routes was requested in issues 585 and 2316, implemented in pull request 2336, and released in version 24.1.

(continued from previous page)

```
  "GlobalConfiguration": {
    "BaseUrl": "https://ocelot.net",
    "DownstreamScheme": "http",
    "ServiceDiscoveryProvider": {
      // required section for dynamic routing
    },
    "AuthenticationOptions": {
      "RouteKeys": [], // or null, no grouping, thus opts apply to all dynamic␣
↪routes
      "AuthenticationProviderKeys": ["Bearer"], // use a global JWT bearer␣
↪auth provider for all discovered services
      "AllowedScopes": ["oc-admin"] // require the global 'scope' claim to gain␣
↪access to all discovered services
    }
  }
}
```

In this configuration, an `oc-admin` scope authorization is applied to all implicit dynamic routes by the global `Bearer` JWT signing service. However, for the "my-service" service, authorization with the `my-service` scope is applied, and authentication is provided by another source of tokens named `MyKey`.

> **Note**
>
> 1. If the `RouteKeys` option is not defined or the array is empty in the global `AuthenticationOptions`, the global options will apply to all routes. If the array contains route keys, it defines a single group of routes to which the global options apply. Routes excluded from this group must specify their own route-level `AuthenticationOptions`.
>
> 2. Prior to version 24.1, global and dynamic route `AuthenticationOptions` were not available. Starting with version 24.1, global configuration is supported for both static and dynamic routes.

8.6 Allowed Scopes

Option: `AllowedScopes`
Middleware: *Authorization Middleware*

To set up authorization by scopes from the `AllowedScopes` collection, after successful authentication by the middleware and after claims have been transformed, the authorization middleware in Ocelot retrieves all user claims (from the token) of the '`scope`' type and ensures that the user has at least one of the scopes in the list. This provides a way to restrict access to a route on a per-scope basis.

> **Note**
>
> [5] Depending on the authentication provider, incoming tokens embed the '`scope`' claim value in the body either as an array or as a single space-separated string of multiple values. For instance, *Identity Server Bearer Tokens* use an array, whereas most *JWT Tokens* providers generate a space-separated list of scopes, in accordance with RFC 8693, as stated in section "4.2. "scope" (Scopes) Claim". Since version 24.1, Ocelot supports RFC 8693 (OAuth 2.0 Token Exchange) for the `scope` claim in the `ScopesAuthorizer` service, also known as the `IScopesAuthorizer` service in the DI container.

[5] The "*Allowed Scopes*" feature fully supports RFC 8693 (OAuth 2.0 Token Exchange) for the `scope` claim in the `ScopesAuthorizer` service,

Note

Starting with version 24.1, specifying global *allowed scopes* is exclusively supported. Be cautious when overriding the global `AllowedScopes` array with a route-level `AllowedScopes` array; a combination of the route scheme (`AuthenticationProviderKeys` array) and its *allowed scopes* might be required, since new *allowed scopes* could belong to another authentication provider's security model. For more details, refer to the "*Configuration and AllowAnonymous*" and "*Global Configuration*" sections.

8.7 JWT Tokens

If you want to authenticate using JWT tokens maybe from a provider like Auth0, you can register your authentication middleware as normal e.g.

```
builder.Services
    .AddAuthentication()
    .AddJwtBearer("Auth0", options =>
    {
        options.Authority = "test";
        options.Audience = "test";
    });
builder.Services
    .AddOcelot(builder.Configuration);
```

Then map the authentication provider key to a route in your configuration e.g.

```
"AuthenticationOptions": {
  "AuthenticationProviderKeys": ["Auth0"],
}
```

JWT Tokens Docs

- Microsoft Learn: Authentication and authorization in minimal APIs
- Andrew Lock | .NET Escapades: A look behind the JWT bearer authentication middleware in ASP.NET Core

8.8 Identity Server Bearer Tokens

In order to use IdentityServer bearer tokens, register your IdentityServer services as usual in Program with a scheme (key). If you don't understand how to do this, please consult the IdentityServer documentation.

```
Action<JwtBearerOptions> options = o =>
{
    o.Authority = "https://whereyouridentityserverlives.com";
    // ...
};
builder.Services
    .AddAuthentication()
    .AddJwtBearer("IS4", options);
```

(continues on next page)

which is part of the *Authorization Middleware*. Refer to section 4.2. "scope" (Scopes) Claim. This enhancement was requested in bug 913, fixed in pull request 1478, and the patch was rolled out as part of the 24.1 release.

(continued from previous page)

```
builder.Services
    .AddOcelot(builder.Configuration);
```

Then map the authentication provider key to a route in your configuration e.g.

```
"AuthenticationOptions": {
  "AuthenticationProviderKeys": ["IS4"],
}
```

8.9 Auth0 by Okta

Yet another identity provider by Okta, see Auth0 Developer Resources.

Add the following, at minimum, to your startup Program:

```
builder.Services
    .AddAuthentication()
    .AddJwtBearer("Okta", o =>
    {
        var conf = builder.Configuration;
        o.Audience = conf["Authentication:Okta:Audience"]; // Okta␣
↪Authorization server Audience
        o.Authority = conf["Authentication:Okta:Server"]; // Okta␣
↪Authorization Issuer URI URL e.g. https://{subdomain}.okta.com/oauth2/
↪{authidentifier}
    });
builder.Services
    .AddOcelot(builder.Configuration);

var app = builder.Build();
await app
    .UseAuthentication()
    .UseOcelot();
await app.RunAsync();
```

In order to get Ocelot to view the scope claim from Okta properly, you have to add the following to map the default Okta `scp` claim to `scope`:

```
// Map Okta "scp" to "scope" claims instead of http://schemas.microsoft.com/
↪identity/claims/scope to allow Ocelot to read/verify them
JsonWebTokenHandler.DefaultInboundClaimTypeMap.Remove("scp");
JsonWebTokenHandler.DefaultInboundClaimTypeMap.Add("scp", "scope");
```

Okta Notes

1. Issue 446 contains some code and examples that might help with Okta integration.
2. Here is documentation for better clarity on claims mapping: Mapping, customizing, and transforming claims in ASP.NET Core.
3. It is highly advisable to read and understand the *Warnings* related to the critical changes in authentication when utilizing .NET 8.

8.10 Warnings

> **Warning**
>
> .NET 8 introduced a breaking change where `JwtSecurityToken` was replaced with `JsonWebToken` to enhance performance and reliability. Consequently, their handlers were changed `JwtSecurityTokenHandler` to `JsonWebTokenHandler`. For a complete understanding of .NET 8 breaking change related to JWT tokens, please refer to the Microsoft Learn documentation: "Security token events return a JsonWebToken".

8.11 Links

- Microsoft Learn: Overview of ASP.NET Core authentication
- Microsoft Learn: Authorize with a specific scheme in ASP.NET Core
- Microsoft Learn: Policy schemes in ASP.NET Core
- Microsoft Learn: Mapping, customizing, and transforming claims in ASP.NET Core
- Microsoft .NET Blog: ASP.NET Core Authentication with IdentityServer4

8.12 Roadmap

Nothing is currently in the stack, but the Ocelot team is rethinking a new version of the "*Administration*" feature, which is closely dependent on authentication.

We invite you to add more examples if you have integrated with other identity providers and the integration solution is working. Please open a "Show and tell" discussion in the repository.

CHAPTER

NINE

AUTHORIZATION

Ocelot supports claims based authorization which is run post authentication. This means if you have a route you want to authorize, you can add the following to your route configuration:

```
"RouteClaimsRequirement": {
  "UserType": "registered"
}
```

In this example, when the *Authorization Middleware* is called, Ocelot will check to see if the user has the claim type `UserType` and if the value of that claim is `"registered"`. If it isn't then the user will not be authorized and the response will be 403 Forbidden.

9.1 Authorization Middleware

The AuthorizationMiddleware is built-in into Ocelot pipeline.

> Previous private: `ClaimsToClaimsMiddleware`
> Previous public: `PreAuthorizationMiddleware`
> **This**: `AuthorizationMiddleware`
> Next private: `ClaimsToHeadersMiddleware`
> Next public: `PreQueryStringBuilderMiddleware`

So, the closest middlewares are in order of calling:

`ClaimsToClaimsMiddleware` `PreAuthorizationMiddleware` **AuthorizationMiddleware** `ClaimsToHeadersMiddleware` `PreQueryStringBuilderMiddleware`

As you may know from the *Middleware Injection* chapter, the Authorization middleware can be overridden like this:

```
var app = builder.Build();
await app.UseOcelot(new OcelotPipelineConfiguration
{
    AuthorizationMiddleware = async (context, next) =>
    {
        await next.Invoke();
    }
});
await app.RunAsync();
```

Note! Do this in very rare cases, because overriding the Authorization middleware means you will lose claims and scopes authorizer through the `RouteClaimsRequirement` property of the route. Another

option is preparing before the actual authorization in `PreAuthorizationMiddleware`, which is public and open to overriding.

```
await app.UseOcelot(new OcelotPipelineConfiguration
{
    PreAuthorizationMiddleware = async (context, next) =>
    {
        // Do whatever you want here
        await next.Invoke(); // next is AuthorizationMiddleware
    }
});
```

CHAPTER TEN

CACHING

[1] Ocelot currently supports caching on the URL of the downstream service and setting a TTL in seconds to expire the cache. Users can also clear the cache for a specific region by using Ocelot's *Administration API*.

Ocelot utilizes some very rudimentary caching at the moment provider by the CacheManager project. This is an amazing project that is solving a lot of caching problems. We would recommend using this package to cache with Ocelot.

The following example shows how to add *CacheManager* to Ocelot so that you can do output caching.

10.1 Install

First of all, add the following Ocelot.Cache.CacheManager package:

```
Install-Package Ocelot.Cache.CacheManager
```

This will give you access to the Ocelot cache manager extension methods. The second step is to add the following to your Program:

```
using Ocelot.Cache.CacheManager;

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

10.2 `CacheOptions` Schema

Class: FileCacheOptions

The following is the full *caching* configuration, used in both the *Route Schema* and the *Dynamic Route Schema*. Not all of these options need to be configured; however, the `TtlSeconds` option is mandatory.

```
"CacheOptions": {
  "TtlSeconds": 1, // nullable integer
  "Region": "", // string
  "Header": "", // string
  "EnableContentHashing": false // nullable boolean
}
```

[1] Historically, *Caching* is one of Ocelot's earliest features, first introduced in version 1.1 on February 2, 2017, the initial release of Ocelot. The "Clear cache region via *Administration API*" feature was first delivered in pull request 109 and released in version 1.4.8.

Option	Description
TtlSeconds	Time-To-Live (TTL) in seconds for the cached downstream response, i.e., the absolute expiration timeout starting from when the item is added to the cache. This option is required. If undefined, it defaults to 0 (zero), which disables caching.
Region	Specifies the cache region to be cleared via Ocelot's *Administration API*. See: `DELETE {adminPath}/outputcache/{region}`
Header	Specifies the header name used for native Ocelot caching control, defaulting to the special `OC-Cache-Control` header. If the header is present, its value is included in the cache key constructed by the `ICacheKeyGenerator` service. Varying header values result in different cache keys, effectively invalidating the cache.
EnableContentHashin	Toggles inclusion of request body hashing in the cache key. Disabled by default (`false`) due to potential performance impact. Recommended for POST/PUT routes where request body affects response. Refer to the *EnableContentHashing option* section.

The actual `CacheOptions` schema with all the properties can be found in the C# FileCacheOptions class.

10.3 Configuration

Finally, in order to use caching on a route in your route configuration add these sections:

```
"CacheOptions": {
  "TtlSeconds": 15,
  "Region": "europe-central",
  "Header": "OC-Cache-Control",
  "EnableContentHashing": false // my route has GET verb only, assigning 'true'␣
↪for requests with body: POST, PUT etc.
},
// Warning! FileCacheOptions section is deprecated! -> use CacheOptions
"FileCacheOptions": {
  "TtlSeconds": 15,
  "Region": "europe-central",
  "Header": "OC-Cache-Control",
  "EnableContentHashing": false // my route has GET verb only, assigning 'true'␣
↪for requests with body: POST, PUT etc.
}
```

- In this example, `TtlSeconds` is set to 15, which means the cache will expire 15 seconds after the response is stored.
- The `Region` property specifies a cache region. Cache entries within a region can be cleared by calling Ocelot's *Administration API*.
- If a header name is defined in the `Header` property, its value is retrieved from the `HttpRequest` headers. If the header is present, its value is included in the cache key constructed by the `ICacheKeyGenerator` service. Varying header values result in different cache keys, effectively invalidating the cache.
- Finally, `EnableContentHashing` is disabled due to the current route using the `GET` verb, which does not include a request body.

Warning

According to the static *Route Schema*, the `FileCacheOptions` section has been deprecated!

The old schema `FileCacheOptions` section is deprecated in version 24.1! Use `CacheOptions` instead of `FileCacheOptions`! Note that `FileCacheOptions` will be removed in version 25.0! For backward compatibility in version 24.1, the `FileCacheOptions` section takes precedence over the `CacheOptions` section.

10.4 `EnableContentHashing` option[2]

Previously, in versions prior to 23.0, the request body was used to compute the cache key. However, due to potential performance issues arising from request body hashing, it has been disabled by default. Clearly, this constitutes a breaking change and presents challenges for users who require cache key calculations that consider the request body (e.g., for the POST method). To address this issue, it is recommended to enable the option either at the route level or globally in the "*Global Configuration*" section:

```
"CacheOptions": {
  // ...
  "EnableContentHashing": true
}
```

Ocelot Team Recommendation

Although the community raised concerns about backward compatibility in issue 2234, Ocelot team maintains that *caching* performance takes precedence over backward compatibility when migrating from versions prior to 23.0. The proposed option clarifies that `POST` requests should **not** be cached; only `GET` requests are eligible for caching. Therefore, `POST` and `GET` verbs must be separated into distinct routes:

- POST routes with *caching* disabled
- GET routes with *caching* enabled

10.5 Global Configuration[3]

Copying route-level properties for each static route is no longer necessary, as version 23.3 allows these values to be set in the `GlobalConfiguration` section. This convenience applies to `Header` and `Region` as well. However, if no global `TtlSeconds` value is defined, this option must still be explicitly set per route to enable caching. As a result, the final configuration for static routes might look like:

```
{
  "Routes": [
    {
      "Key": "R0", // optional
      "CacheOptions": {
        "TtlSeconds": 60 // 1-minute short-term caching
      },
      // ...
    },
```

(continues on next page)

[2] The "*EnableContentHashing option*" feature was requested in issue 2059 and released in version 23.3.

[3] *Global Configuration* for static routes was first introduced in pull request 2058 and released in version 23.3. Support for dynamic routes was added in pull request 2331 and delivered in version 24.1.

(continued from previous page)

```
    {
      "Key": "R1", // this route is part of a group
      "CacheOptions": {}, // optional due to grouping
      // ...
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://ocelot.net",
    "CacheOptions": {
      "RouteKeys": ["R1",], // if undefined or empty array, opts will apply to␣
↪all routes
      "TtlSeconds": 300 // enable global caching for a duration of 5 minutes
    },
    // ...
  }
}
```

Dynamic routes were not supported in versions prior to 24.1. Starting with version 24.1, global *cache options* for *Dynamic Routing* were introduced. These global options may also be overridden in the DynamicRoutes configuration section, as defined by the *Dynamic Route Schema*.

```
{
  "DynamicRoutes": [
    {
      "Key": "", // optional
      "ServiceName": "my-service",
      "CacheOptions": {
        "TtlSeconds": 60 // 1-minute short-term caching
      }
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://ocelot.net",
    "DownstreamScheme": "http",
    "ServiceDiscoveryProvider": {
      // required section for dynamic routing
    },
    "CacheOptions": {
      "RouteKeys": [], // or null, no grouping, thus opts apply to all dynamic␣
↪routes
      "TtlSeconds": 300 // enable global caching for a duration of 5 minutes
    }
  }
}
```

In this configuration, a 5-minute *caching* duration is applied to all implicit dynamic routes. However, for the "my-service" service, the *caching* TTL has been explicitly reduced from 5 minutes to 1 minute.

Note

1. If the RouteKeys option is not defined or the array is empty in the global CacheOptions, the global options will apply to all routes. If the array contains route keys, it defines a single group of routes to

which the global options apply. Routes excluded from this group must specify their own route-level `CacheOptions`.

2. Prior to version 23.3, global `CacheOptions` were not available. Starting with version 24.1, global configuration is supported for both static and dynamic routes.

10.6 Custom Caching

If you want to add your own caching method, implement the following interfaces and register them in DI e.g.

```
builder.Services
    .AddSingleton<IOcelotCache<CachedResponse>, MyCache>();
```

- `IOcelotCache<CachedResponse>` this is for output caching.
- `IOcelotCache<FileConfiguration>` this is for caching the file configuration if you are calling something remote to get your config such as Consul.

10.7 Roadmap

Please dig into the Ocelot source code to find more. We would really appreciate it if anyone wants to implement Redis, Memcached etc. Please, open a new Show and tell thread in Discussions space of the repository.

CHAPTER

ELEVEN

CLAIMS TRANSFORMATION

Ocelot allows the user to access claims and transform them into headers, query string parameters, other claims and change downstream paths. This is only available once a user has been authenticated.

After the user is authenticated, we run the claims to claims transformation middleware (see the ClaimsToClaimsMiddleware class). This allows the user to transform claims before the authorization middleware is called. After the user is authorized, we call the claims to headers middleware (see the ClaimsToHeadersMiddleware class), then the claims to query string parameters middleware (see the ClaimsToQueryStringMiddleware class), and finally the claims to downstream path middleware (see the ClaimsToDownstreamPathMiddleware class).

The syntax for performing the transforms is the same for each process. In the route configuration, a JSON dictionary is added with a specific name either `AddClaimsToRequest`, `AddHeadersToRequest`, `AddQueriesToRequest`, or `ChangeDownstreamPathTemplate`.

> **Note**: This syntax is not ideal. So any suggestions are welcome…

Within this dictionary the entries specify how Ocelot should transform things! The key to the dictionary is going to become the key of either a claim, header or query parameter. In the case of `ChangeDownstreamPathTemplate`, the key must be also specified in the `DownstreamPathTemplate`, in order to do the transformation.

The value of the entry is parsed to logic that will perform the transform. First of all, a dictionary accessor is specified e.g. `Claims[CustomerId]`. This means we want to access the claims and get the `CustomerId` claim type. Next is a "greater than" > symbol which is just used to split the string. The next entry is either value or value with an indexer. If value is specified, Ocelot will just take the value and add it to the transform. If the value has an indexer, Ocelot will look for a delimiter which is provided after another "greater than" > symbol. Ocelot will then split the value on the delimiter and add whatever was at the index requested to the transform.

11.1 Claims to Claims

Below is an example configuration that will transform claims to claims

```
"AddClaimsToRequest": {
  "UserType": "Claims[sub] > value[0] > |",
  "UserId": "Claims[sub] > value[1] > |"
}
```

This shows a transforms where Ocelot looks at the users `sub` claim and transforms it into `UserType` and `UserId` claims. Assuming the `sub` looks like this `usertypevalue|useridvalue`.

11.2 Claims to Headers

Below is an example configuration that will transform claims to headers

```
"AddHeadersToRequest": {
  "CustomerId": "Claims[sub] > value[1] > |"
}
```

This shows a transform where Ocelot looks at the users `sub` claim and transforms it into a `CustomerId` header. Assuming the `sub` looks like this `usertypevalue|useridvalue`.

11.3 Claims to Query String Parameters

Below is an example configuration that will transform claims to query string parameters

```
"AddQueriesToRequest": {
  "LocationId": "Claims[LocationId] > value",
}
```

This shows a transform where Ocelot looks at the users `LocationId` claim and add it as a query string parameter to be forwarded onto the downstream service.

11.4 Claims to Downstream Path

Below is an example configuration that will transform claims to downstream path custom placeholders:

```
"UpstreamPathTemplate": "/api/users/me/{everything}",
"DownstreamPathTemplate": "/api/users/{userId}/{everything}",
"ChangeDownstreamPathTemplate": {
  "userId": "Claims[sub] > value[1] > |",
}
```

This shows a transform where Ocelot looks at the users `userId` claim and substitutes the value to the `{userId}` placeholder specified in the `DownstreamPathTemplate`. Take into account that the key specified in the `ChangeDownstreamPathTemplate` must be the same than the placeholder specified in the `DownstreamPathTemplate`.

> **Note**: If a key specified in the `ChangeDownstreamPathTemplate` does not exist as a placeholder in `DownstreamPathTemplate`, it will fail at runtime returning an error in the response.

CHAPTER

TWELVE

CONFIGURATION

An example configuration can be found here in ocelot.json. There are two major sections to the configuration: an array of `Routes` and a `GlobalConfiguration` sections:

```
{
  "Routes": [],
  "GlobalConfiguration": {}
}
```

From the *Getting Started* chapter and its *Configuration* section, you may already know that there are four total configuration sections:

```
{
  "Routes": [], // static routes
  "DynamicRoutes": [],
  "Aggregates": [], // BFF
  "GlobalConfiguration": {}
}
```

Section	*Description*
`Routes` with *Route Schema*	The static objects that tell Ocelot how to treat an upstream request. Once static routes have been loaded during gateway startup, in general, they cannot be changed during the lifetime of the app instance, with a few exceptional use cases.
`DynamicRoutes` with *Dynamic Route Schema*	This section enables dynamic routing when using a *Service Discovery* provider. Please refer to the *Dynamic Routing 8* docs for more details.
`Aggregates` with *Aggregate Route Schema*	This section allows specifying aggregated routes that compose multiple normal routes and map their responses into one JSON object. It allows you to start implementing a *Back-end For a Front-end* (BFF) type architecture with Ocelot. Please refer to the *Aggregation* chapter for more details.
`GlobalConfiguration` with *Global Configuration Schema*	This section is a bit hacky and allows overrides of static route-specific settings. It is useful if you do not want to manage lots of route-specific settings.

To fully understand all configuration capabilities, we recommend reading all sections below.

12.1 Route Schema

Class: FileRoute

Here is the complete route configuration, also known as the *"route schema,"* of top-level properties. You do not need to set all of these things, but this is everything that is available at the moment.

```
{
  "AddClaimsToRequest": {}, // dictionary
  "AddHeadersToRequest": {}, // dictionary
  "AddQueriesToRequest": {}, // dictionary
  "AuthenticationOptions": {}, // object
  "ChangeDownstreamPathTemplate": {}, // dictionary
  "DangerousAcceptAnyServerCertificateValidator": false,
  "DelegatingHandlers": [], // array of strings
  "DownstreamHeaderTransform": {}, // dictionary
  "DownstreamHostAndPorts": [], // array of FileHostAndPort
  "DownstreamHttpMethod": "",
  "DownstreamHttpVersion": "",
  "DownstreamHttpVersionPolicy": "",
  "DownstreamPathTemplate": "",
  "DownstreamScheme": "",
  "CacheOptions": {}, // object
  "FileCacheOptions": {}, // deprecated! -> use CacheOptions
  "HttpHandlerOptions": {}, // object
  "Key": "",
  "LoadBalancerOptions": {}, // object
  "Metadata": {}, // dictionary
  "Priority": 1, // integer
  "QoSOptions": {}, // object
  "RateLimitOptions": {}, // object
  "RequestIdKey": "",
  "RouteClaimsRequirement": {}, // dictionary
  "RouteIsCaseSensitive": false,
  "SecurityOptions": {}, // object
  "ServiceName": "",
  "ServiceNamespace": "",
  "Timeout": 0, // nullable integer
  "UpstreamHeaderTemplates": {}, // dictionary
  "UpstreamHeaderTransform": {}, // dictionary
  "UpstreamHost": "",
  "UpstreamHttpMethod": [], // array of strings
  "UpstreamPathTemplate": ""
},
```

The actual route schema with all the properties can be found in the C# FileRoute class.

Note: The old schema `FileCacheOptions` section is deprecated in version 24.1! Use `CacheOptions` instead of `FileCacheOptions`! Note that `FileCacheOptions` will be removed in version 25.0! For backward compatibility in version 24.1, the `FileCacheOptions` section takes precedence over the `CacheOptions` section.

12.2 Dynamic Route Schema

Class: FileDynamicRoute

Here is the complete dynamic route configuration, also known as the *"dynamic route schema,"* of top-level properties.

```
{
  "AuthenticationOptions": {},
  "CacheOptions": {},
  "DownstreamHttpVersion": "",
  "DownstreamHttpVersionPolicy": "",
  "HttpHandlerOptions": {},
  "LoadBalancerOptions": {},
  "Metadata": {}, // dictionary
  "QoSOptions": {},
  "RateLimitRule": {}, // deprecated! -> use RateLimitOptions
  "RateLimitOptions": {},
  "ServiceName": "",
  "ServiceNamespace": "",
  "Timeout": 0 // nullable integer
}
```

The actual dynamic route schema with all the properties can be found in the C# FileDynamicRoute class.

Note 1: The old schema `RateLimitRule` section is deprecated in version 24.1! Use `RateLimitOptions` instead of `RateLimitRule`! Note that `RateLimitRule` will be removed in version 25.0! For backward compatibility in version 24.1, the `RateLimitRule` section takes precedence over the `RateLimitOptions` section.

Note 2: The following options were not supported in versions prior to 24.1 for overriding globally configured options: `AuthenticationOptions`, `CacheOptions`, `HttpHandlerOptions`, `LoadBalancerOptions`, `QoSOptions`, `RateLimitOptions`, `ServiceNamespace`, and `Timeout`. Starting with version 24.1, both global and route-level options for *Dynamic Routing* were introduced. For a clearer understanding of the changes, refer to the previous schema (version 24.0).

12.3 Aggregate Route Schema

Class: FileAggregateRoute

Here is the complete aggregated route configuration, also known as the *"aggregate route schema,"* of top-level properties.

```
{
  "Aggregator": "",
  "Priority": 1, // integer
  "RouteIsCaseSensitive": false,
  "RouteKeys": [], // array of strings
  "RouteKeysConfig": [], // array of AggregateRouteConfig
  "UpstreamHeaderTemplates": {}, // dictionary
  "UpstreamHost": "",
  "UpstreamHttpMethod": [], // array of strings
  "UpstreamPathTemplate": ""
}
```

The actual aggregated route schema with all the properties can be found in the C# FileAggregateRoute class.

12.4 Global Configuration Schema

Class: FileGlobalConfiguration

Here is the complete global configuration, also known as the *"global configuration schema,"* of top-level properties.

```
{
  "AuthenticationOptions": {},
  "BaseUrl": "",
  "CacheOptions": {},
  "DownstreamHeaderTransform": {}, // dictionary
  "DownstreamHttpVersion": "",
  "DownstreamHttpVersionPolicy": "",
  "DownstreamScheme": "",
  "HttpHandlerOptions": {},
  "LoadBalancerOptions": {},
  "Metadata": {}, // dictionary
  "MetadataOptions": {},
  "QoSOptions": {},
  "RateLimitOptions": {},
  "RequestIdKey": "",
  "SecurityOptions": {},
  "ServiceDiscoveryProvider": {},
  "Timeout": 0, // nullable integer
  "UpstreamHeaderTransform": {} // dictionary
}
```

The actual global configuration schema with all the properties can be found in the C# FileGlobalConfiguration class.

Note 1: The following global options were not supported in versions prior to 24.1 for overriding in the *Dynamic Route Schema*: `AuthenticationOptions`, `CacheOptions`, `HttpHandlerOptions`, `LoadBalancerOptions`, `QoSOptions`, `RateLimitOptions`, and `Timeout`. Moreover, these global options were not available in versions prior to 24.1 for static routes, as stated in issue 585. Starting with version 24.1, both static and dynamic route *global* options are fully supported. For a clearer understanding of the changes, refer to the *Dynamic Route Schema* and related notes.

Note 2: The `DownstreamHeaderTransform` and `UpstreamHeaderTransform` global options were introduced in version 24.1, but they are available only for static routes.

12.5 Configuration Overview

Dependency Injection of the *Configuration* feature in Ocelot allows you to extend, manage, and build Ocelot Core *configuration* **before** the stage of building ASP.NET Core services.

To configure the Ocelot Core and services, use the following abstract program-structure, which must be presented in your Program:

1. **Create application builder**: The `Microsoft.AspNetCore.Builder.WebApplication` has three overloaded versions of the `CreateBuilder()` methods. Our recommendation is to utilize arguments

possibly coming from terminal sessions into an app host; thus, use the `CreateBuilder(args)` method.

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

2. **Set up the configuration builder**: Utilize the `WebApplicationBuilder.Configuration` property, which returns a `ConfigurationManager` object implementing the target `IConfigurationBuilder` interface.

```
builder.Configuration.AddOcelot(...);
```

3. **Forward configuration to the Ocelot builder**: The `Ocelot.DependencyInjection.ServiceCollectionExtensions` class has three overloaded versions of the `AddOcelot(IServiceCollection)` methods, which return an `IOcelotBuilder` object.

```
builder.Services.AddOcelot(builder.Configuration);
```

4. **Finish the app setup**, add middlewares, and finally run the application: Let's write the final algorithm.

```
var builder = WebApplication.CreateBuilder(args); // step 1
builder.Configuration.AddOcelot(...); // step 2
builder.Services.AddOcelot(builder.Configuration); // step 3

// Step 4
var app = builder.Build();
await app.UseOcelot();
await app.RunAsync();
```

For comprehensive documentation of configuration DI-extensions, please refer to the *Configuration Overview* section in the *Dependency Injection* chapter.

12.6 Multiple Environments

Like any other ASP.NET Core project Ocelot supports configuration file names such as `appsettings.dev.json`, `appsettings.test.json` etc. In order to implement this add the following to you:

```
var builder = WebApplication.CreateBuilder(args);
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddJsonFile("ocelot.json") // primary config file
    .AddJsonFile($"ocelot.{builder.Environment.EnvironmentName}.json");
builder.Services
    .AddOcelot(builder.Configuration);
```

Ocelot will now use the environment specific configuration and fall back to ocelot.json if there isn't one. Another version of the configuration above, which is based on configuration providers, is the following:

```
var builder = WebApplication.CreateBuilder(args);
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddOcelot() // single ocelot.json file without environment one
    // or
    .AddOcelot(builder.Environment)
```

(continues on next page)

(continued from previous page)

```
    .AddJsonFile($"ocelot.{builder.Environment.EnvironmentName}.json");
builder.Services
    .AddOcelot(builder.Configuration);
```

You also need to set the corresponding ASPNETCORE_ENVIRONMENT variable.

Note 1: More info on configuration can be found in the ASP.NET Core documentation:

- Use multiple environments in ASP.NET Core
- Configuration in ASP.NET Core

Note 2: Calling the following configuration methods is rudimentary in ASP.NET Core because of internal encapsulation in the default builder, aka CreateBuilder(args) method.

```
var builder = WebApplication.CreateBuilder(args);
builder.Configuration
    .AddJsonFile("appsettings.json", true, true) // not required
    .AddJsonFile($"appsettings.{builder.Environment.EnvironmentName}.
↪json", true, true) // not required
    .AddEnvironmentVariables() // not required
    // ...
```

This is explained in the Default application configuration sources docs; thus, remove these optional methods.

12.7 Merging Files[1]

Sample: Ocelot.Samples.Configuration

This feature allows users to have multiple configuration files to make managing large configurations easier.

Rather than directly adding the configuration e.g., using AddJsonFile("ocelot.json"), you can achieve the same result by invoking AddOcelot() as shown below:

```
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddOcelot(builder.Environment); // will skip environment file
```

In this scenario, Ocelot will look for any files that match the pattern `^ocelot\.(.*?)\.json$` as the regular expression and then merge these together. The environment file will be skipped aka ocelot.{builder.Environment.EnvironmentName}.json. If you want to set the GlobalConfiguration property, you must have a file called ocelot.global.json.

The way Ocelot merges the files is basically load them, loop over them, skip environment file, add any Routes, add any AggregateRoutes and if the file is called ocelot.global.json add the GlobalConfiguration aswell as any Routes or AggregateRoutes. Ocelot will then save the merged configuration to a file called ocelot.json and this will be used as the source of truth while Ocelot is running.

Note 1: Currently, validation occurs only during the final merging of configurations in Ocelot. It's essential to be aware of this when troubleshooting issues. We recommend thoroughly inspecting the contents of the ocelot.json file if you encounter any problems.

Note 2: The Merging feature is operational only during the application's startup. Consequently, the merged configuration in ocelot.json remains static post-merging and startup. Once the

[1] The "*Merging Files*" feature was requested in issue 296, since then we extended it in issue 1216 (PR 1227) as "*Merging files to memory*" subfeature which was released as a part of version 23.2.

Ocelot application has started, you cannot call the `AddOcelot` method, nor can you employ the merging feature within `AddOcelot`. If you still require on-the-fly updating of the primary configuration file, `ocelot.json`, please refer to the *React to Changes* section. Additionally, note that merging partial configuration files (such as `ocelot.*.json`) on the fly using *Administration* API is not currently implemented.

Note 3: An alternative to static merged configurations could be the construction of the `FileConfiguration` object before passing it as an argument to the *AddOcelot methods* method. Refer to the *Build From Scratch* subsection for details.

12.7.1 Keep files in a folder

You can also give Ocelot a specific path to look in for the configuration files as shown below:

```
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddOcelot("/my/folder", builder.Environment); // happy path
```

Ocelot needs the `builder.Environment` so it knows to exclude any environment-specific files from the merging algorithm, such as `ocelot.{builder.Environment.EnvironmentName}.json`.

12.7.2 Merging files to memory[2]

By default, Ocelot writes the merged configuration to disk as ocelot.json (the primary configuration file) by adding the file to the ASP.NET configuration provider.

If your web server lacks write permissions for the configuration folder, you can instruct Ocelot to use the merged configuration directly from memory. Here's how:

```
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    // It implicitly calls ASP.NET AddJsonStream extension method for IConfigurationBuilder
    // .AddJsonStream(new MemoryStream(Encoding.UTF8.GetBytes(json)));
    .AddOcelot(builder.Environment, MergeOcelotJson.ToMemory);
```

This feature proves exceptionally valuable in cloud environments like Azure, AWS, and GCP, especially when the app lacks sufficient write permissions to save files. Furthermore, within Docker container environments, permissions can be scarce, necessitating substantial DevOps efforts to enable file write operations. Therefore, save time by leveraging this feature!

12.8 Reload On Change

Ocelot supports reloading the JSON configuration file on change. For instance, the following will recreate Ocelot internal configuration when the ocelot.json file is updated manually:

```
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddJsonFile("ocelot.json", optional: false, reloadOnChange: true) // ASP.NET framework version
```

[2] The "*Merging files to memory*" feature is based on the MergeOcelotJson enumeration type with values: `ToFile` and `ToMemory`. The 1st one is implicit by default, and the second one is exactly what you need when merging to memory. See more details on implementations in the ConfigurationBuilderExtensions class.

> **Note**: Starting from version 23.2, most *AddOcelot methods* include optional `bool?` arguments, specifically `optional` and `reloadOnChange`. Therefore, you have the flexibility to provide these arguments when invoking the native AddJsonFile method during the final configuration step (see AddOcelotJsonFile implementation).

We recommend using the *AddOcelot methods* to control reloading, rather than relying on the framework's `AddJsonFile` method. For example:

```
// Old solution based on native framework functionality
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddJsonFile(ConfigurationBuilderExtensions.PrimaryConfigFile, optional:␣
↪false, reloadOnChange: true);

var config = builder.Configuration;
var env = builder.Environment;
var mergeTo = MergeOcelotJson.ToFile; // ToMemory
var folder = "/My/folder";
var configuration = new FileConfiguration(); // read from anywhere and␣
↪initialize

// Advanced solutions based on Ocelot functionality
config.AddOcelot(env, mergeTo, optional: false, reloadOnChange: true); //␣
↪with environment and merging type
config.AddOcelot(folder, env, mergeTo, optional: false, reloadOnChange:␣
↪true); // with folder, environment and merging type
config.AddOcelot(configuration, optional: false, reloadOnChange: true); //␣
↪with configuration object created by your own
config.AddOcelot(configuration, env, mergeTo, optional: false,␣
↪reloadOnChange: true); // with configuration object, environment and merging␣
↪type
```

Examining the code within the `ConfigurationBuilderExtensions` class would be helpful for gaining a better understanding of the signatures of the overloaded *AddOcelot methods*.

12.9 React to Changes

Resolve `IOcelotConfigurationChangeTokenSource` interface from the DI container if you wish to react to changes to the Ocelot configuration via the *Administration API* or ocelot.json being reloaded from the disk.

You may either poll the change token's `IChangeToken.HasChanged` property, or register a callback with the `RegisterChangeCallback` method.

How to poll is explained here:

```
public class ConfigurationNotifyingService : BackgroundService
{
    private readonly IOcelotConfigurationChangeTokenSource _
↪tokenSource;
    private readonly ILogger _logger;

    public␣
↪ConfigurationNotifyingService(IOcelotConfigurationChangeTokenSource␣
```

(continues on next page)

(continued from previous page)

```
↪tokenSource, ILogger logger)
    {
        _tokenSource = tokenSource;
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken␣
↪stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            if (_tokenSource.ChangeToken.HasChanged)
            {
                _logger.LogInformation("Configuration has changed");
            }
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

How to register a callback is explained here:

```
public sealed class MyConfigurationNotifying : IDisposable
{
    private readonly IOcelotConfigurationChangeTokenSource _
↪tokenSource;
    private readonly IDisposable _callbackHolder;

    public␣
↪MyConfigurationNotifying(IOcelotConfigurationChangeTokenSource␣
↪tokenSource)
    {
        _tokenSource = tokenSource;
        _callbackHolder = tokenSource.ChangeToken
            .RegisterChangeCallback(_ => Console.WriteLine(
↪"Configuration has changed"), null);
    }

    public void Dispose() => _callbackHolder.Dispose();
}
```

12.10 Store in Consul

As a developer, if you have enabled *Service Discovery* with Consul support in Ocelot, you may choose to manage your configuration saving to the *Consul* KV store.

Beyond the traditional methods of storing configuration in a file vs folder (*Merging Files 1*), or in-memory (*Merging files to memory 2*), you also have the alternative to utilize the Consul server's storage capabilities.

For further details on managing Ocelot configurations via a Consul instance, please consult the "*Configuration in KV Store*" section.

12.11 Build From Scratch

Class: FileConfiguration

Storing, reading, and writing static configurations may have limitations. Therefore, for more flexible and advanced scenarios the `FileConfiguration` object can be built from scratch in C# code of Ocelot application startup. Additionally after reading static configuration from various sources such as, remote file systems, remote storages or cloudages, you can rewrite options to the configuration.

Ocelot does not provide a fluent syntax to build configuration on fly as other products do. However, it is possible to inject a `FileConfiguration` object during Ocelot startup using the *AddOcelot methods* with a special parameter:

```
public static IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder, FileConfiguration fileConfiguration, /* optional */);
```

The method above will deserialize the object to disk. If you prefer to keep the configuration in memory, the following method includes the `MergeOcelotJson` parameter:

```
public static IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder, FileConfiguration fileConfiguration, IWebHostEnvironment env, MergeOcelotJson mergeTo, /* optional */);
```

In summary, the final .NET 8+ solution should be written in Program using top-level statements:

```
using Ocelot.Configuration.File;
using Ocelot.DependencyInjection;
using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

// Build Ocelot's configuration object on the fly:
var config = new FileConfiguration(); // create new or read static state from anywhere
// ... initialize or rewrite props: add routes, global config, etc.

builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddOcelot(config) // MergeOcelotJson.ToFile : writing config JSON back to disk
    .AddOcelot(config, builder.Environment, MergeOcelotJson.ToMemory); // merging to memory
builder.Services
    .AddOcelot(builder.Configuration);

var app = builder.Build();
await app.UseOcelot();
await app.RunAsync();
```

As a final step, you could add shutdown logic to save the complete configuration back to the storage, deserializing it to JSON format.

12.12 HttpHandlerOptions

Class: FileHttpHandlerOptions
MS Learn: SocketsHttpHandler Class

This route configuration section allows for following HTTP redirects, for instance, via the boolean AllowAutoRedirect option. These options can be set at the route or global level for both static and *dynamic routing*.

Use HttpHandlerOptions in a route configuration to set up HttpMessageHandler behavior based on a SocketsHttpHandler instance:

```
"HttpHandlerOptions": {
  "AllowAutoRedirect": false,
  "MaxConnectionsPerServer": 2147483647, // max integer
  "PooledConnectionLifetimeSeconds": 120,
  "UseCookieContainer": false,
  "UseProxy": false,
  "UseTracing": false
}
```

Option	*Description*
AllowAutoRedirect default: false	This value indicates whether the request should follow Redirection messages (HTTP 3xx status codes). Set it true if the request should automatically follow redirection responses from the downstream resource; otherwise false.
MaxConnectionsPerSe default: 2147483647, maximum integer	This controls how many connections the internal HttpMessageInvoker will open to a single *IIS*/*Kestrel* server.
PooledConnectionLif default: 120 seconds	This controls how long a connection can be in the pool to be considered reusable. Also refer to the **1st note** below!
UseCookieContainer default: false	This indicates whether the handler uses the CookieContainer property to store server cookies and uses these cookies when sending requests. Also refer to the **2nd note** below!
UseProxy default: false	Refer to MS Learn: UseProxy Property
UseTracing default: false	This enables *Tracing* feature in Ocelot. Also refer to the **3rd note** below!

Note

1. If the PooledConnectionLifetimeSeconds option is not defined, the default value is 120 seconds, which is hardcoded in the HttpHandlerOptions class as the DefaultPooledConnectionLifetimeSeconds constant.

2. If you use the `CookieContainer`, Ocelot caches the `HttpMessageInvoker` for each downstream service. This means that all requests to that downstream service will share the same cookies. Issue 274 was created because a user noticed that the cookies were being shared. The Ocelot team tried to think of a nice way to handle this but we think it is impossible. If you don't cache the clients, that means each request gets a new client and therefore a new cookie container. If you clear the cookies from the cached client container, you get race conditions due to inflight requests. This would also mean that subsequent requests don't use the cookies from the previous response! All in all not a great situation. We would avoid setting `UseCookieContainer` to `true` unless you have a really really good reason. Just look at your response headers and forward the cookies back with your next request!

3. `UseTracing` option adds a tracing `DelegatingHandler` (aka `Ocelot.Requester.ITracingHandler`) after obtaining it from `ITracingHandlerFactory`, encapsulating the `Ocelot.Logging.IOcelotTracer` service of DI-container.

4. Prior to version 24.1, global `HttpHandlerOptions` were not accessible, as they were only available at the route level for static routes. Since version 24.1, global configuration is supported for both static and dynamic routes.

12.13 SSL Errors

If you want to ignore SSL warnings (errors), set the following in your route configuration:

```
"DangerousAcceptAnyServerCertificateValidator": true
```

We don't recommend doing this! The team suggests creating your own certificate and then getting it trusted by your local (or remote) machine, if you can. For `https` scheme, this fake validator was requested by issue 309. For `wss` scheme, this fake validator was added by PR 1377.

> **Note**: As a team, we do not consider it an ideal solution. On one hand, the community wants to have an option to work with self-signed certificates. But on the other hand, currently, source code scanners detect two serious security vulnerabilities because of this fake validator in version 20.0 and higher. The Ocelot team will rethink this unfortunate situation, and it is highly likely that this feature will at least be redesigned or removed completely.

For now, the SSL fake validator makes sense in local development environments when a route has `https` or `wss` schemes with self-signed certificates for those routes. There are no other reasons to use the `DangerousAcceptAnyServerCertificateValidator` property at all!

As a team, we highly recommend following these instructions when developing your gateway app with Ocelot:

- **Local development environments**: Use this feature to avoid SSL errors for self-signed certificates in the case of `https` or `wss` schemes. We understand that some routes should have the downstream scheme exactly with SSL, because they are also in development and/or deployed using SSL protocols. However, we believe that, especially for local development, you can switch from `https` to `http` without any objection since the services are in development and there is no risk of data leakage.
- **Remote development environments**: Everything is the same as for local development. However, this case is less strict; you have more options to use real certificates to switch off the feature. For instance, you can deploy downstream services to cloud and hosting providers that have their own signed certificates for SSL. At least your team can deploy one remote web server to host downstream services. Install your own certificate or use the cloud provider's one.
- **Staging or testing environments**: We do not recommend using self-signed certificates because web servers should have valid certificates installed. Ask your system administrator or DevOps engineers to create valid certificates.

- **Production environments**: **Do not use self-signed certificates at all!** System administrators or DevOps engineers must create real valid certificates signed by hosting or cloud providers. **Switch off the feature for all routes!** Remove the `DangerousAcceptAnyServerCertificateValidator` property for all routes in the production version of the ocelot.json file!

12.14 DownstreamHttpVersion

MS Learn: HttpVersion Class

Ocelot allows you to choose the HTTP version it will use to make the proxy request. It can be set as `1.0`, `1.1`, or `2.0`.

12.14.1 DownstreamHttpVersionPolicy[3]

Enum: HttpVersionPolicy

This routing property enables the configuration of the `VersionPolicy` property within `HttpRequestMessage` objects for downstream HTTP requests. For additional details, refer to the following documentation:

- HttpRequestMessage.VersionPolicy Property
- HttpVersionPolicy Enum
- HttpVersion Class

The `DownstreamHttpVersionPolicy` option is intricately linked with the *DownstreamHttpVersion* setting. Therefore, merely specifying `DownstreamHttpVersion` may sometimes be inadequate, particularly if your downstream services or Ocelot logs report HTTP connection errors such as `PROTOCOL_ERROR`. In these routes, selecting the precise `DownstreamHttpVersionPolicy` value is crucial for the `HttpVersion` policy to prevent such protocol errors.

12.14.2 HTTP2 version policy

Given you aim to ensure a smooth HTTP/2 connection setup for the Ocelot app and downstream services with SSL enabled:

```
{
  "DownstreamScheme": "https",
  "DownstreamHttpVersion": "2.0",
  "DownstreamHttpVersionPolicy": "", // empty or not defined
  "DangerousAcceptAnyServerCertificateValidator": true
}
```

And you configure global settings to use *Kestrel* with this snippet:

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
    });
});
```

[3] The "*DownstreamHttpVersionPolicy*" feature was requested in issue 1672 as a part of version 23.3.

When all components are set to communicate exclusively via HTTP/2 without TLS (plain HTTP).

Then the downstream services may display error messages such as:

```
HTTP/2 connection error (PROTOCOL_ERROR): Invalid HTTP/2 connection preface
```

To resolve the issue, ensure that `HttpRequestMessage` has its `VersionPolicy` set to `RequestVersionOrHigher`. Therefore, the `DownstreamHttpVersionPolicy` should be defined as follows:

```
{
  "DownstreamHttpVersion": "2.0",
  "DownstreamHttpVersionPolicy": "RequestVersionOrHigher" // !
}
```

12.15 Dependency Injection

Class: ConfigurationBuilderExtensions

Dependency Injection for this *Configuration* feature in Ocelot is designed to extend and/or control the configuration of the Ocelot Core before the stage of building ASP.NET Core pipeline services. The primary methods are *AddOcelot methods* within the `ConfigurationBuilderExtensions` class, which offers several overloaded versions with corresponding signatures. You can utilize these methods in the Program.cs file of your gateway app to configure the Ocelot pipeline and services.

Find additional details in the dedicated *Configuration Overview* section and in subsequent sections related to the *Dependency Injection* chapter.

12.16 Extend with `Metadata`

Feature: *Metadata*[4]

The `Metadata` options can store any arbitrary data that users can access in middlewares, delegating handlers, etc. By using the *metadata*, users can implement their own logic and extend the functionality of Ocelot.

The *Metadata* feature is designed to extend both the static *Route Schema* and *Dynamic Route Schema*. Global *metadata* must be defined in the `Metadata` section, while parsing options should be placed in the `MetadataOptions` section.

The following example demonstrates practical usage of this feature:

```
{
  "Routes": [
    {
      // other opts...
      "Metadata": {
        "api-id": "FindPost",
        "my-extension/param1": "overwritten-value",
        "other-extension/param1": "value1",
        "other-extension/param2": "value2",
        "tags": "tag1, tag2, area1, area2, func1",
        "json": "[1, 2, 3, 4, 5]"
```

(continues on next page)

[4] The "*Extend with Metadata*" feature was requested in issues 738 and 1990, and it was released as part of version 23.3.

(continued from previous page)

```
      }
    }
  ],
  "GlobalConfiguration": {
    // other opts...
    "Metadata": {
      "instance_name": "dc-1-54abcz",
      "my-extension/param1": "default-value"
    },
    "MetadataOptions": {
      // parsing metadata opts...
    }
  }
}
```

Note: Route *metadata* prevails over global *metadata* from the `GlobalConfiguration` section. Therefore, if the same key data are defined both at the route and global levels, the route *metadata* overrides the global ones.

Now, the route *metadata* can be accessed through the DownstreamRoute object:

```
using Ocelot.Metadata;

public static class OcelotMiddlewares
{
    public static Task PreAuthenticationMiddleware(HttpContext context, Func
↪<Task> next)
    {
        var route = context.Items.DownstreamRoute();
        var param1 = route.GetMetadata<string>("my-extension/param1") ?? throw␣
↪new ArgumentNullException("my-extension/param1");
        var param2 = route.GetMetadata<string>("other-extension/param2",
↪"default-value");
        // Working with metadata...
        return next();
    }
}
```

For comprehensive documentation, please refer to the *Metadata* chapter.

12.17 Timeout

This feature[5] is designed as part of the `MessageInvokerPool`, which contains cached `HttpMessageInvoker` objects per route. Each created `HttpMessageInvoker` encapsulates an `HttpMessageHandler`, specifically a `SocketsHttpHandler` instance, which serves as the base handler for the request pipeline. This pipeline also includes all user-defined *Delegating Handlers*. Finally, both the *Delegating Handlers* and the base `SocketsHttpHandler` are wrapped by Ocelot's custom `TimeoutDelegatingHandler`, which provides the internal timeout functionality.

[5] The initial draft design of the *Timeout* feature was implemented in pull request 1824 as `TimeoutDelegatingHandler` (released in version 23.0), but this version supported only the built-in default timeout of 90 seconds. The full *Timeout* feature was requested in issue 1314, implemented in pull request 2073, and officially released as part of version 24.1.

Note: This design is subject to future review because `TimeoutDelegatingHandler` overrides/mimics the default timeout properties of `SocketsHttpHandler`, as well as the behavior of `HttpMessageInvoker` as a controller for `HttpMessageHandler` objects.

To configure timeouts (in seconds) at different levels, choose the appropriate level and provide the corresponding JSON configuration.

12.17.1 Route timeout

A *route timeout* (also known as Requester middleware timeout based on `TimeoutDelegatingHandler`) can be easily defined using the following JSON, according to the *Route Schema*:

```
{
  // upstream props
  // downstream props
  "Timeout": 3 // seconds
}
```

Please note that the route-level timeout takes precedence over the global timeout. The same configuration applies to *dynamic routes*, according to the *Dynamic Route Schema*.

12.17.2 Global timeout

A *global configuration timeout* can be defined using the following JSON, according to the *Global Configuration Schema*:

```
{
  // routes...
  "GlobalConfiguration": {
    // other props
    "Timeout": 60 // seconds, 1 minute
  }
}
```

Please note that the global timeout is substituted into a route if the route-level timeout is not defined, and it takes precedence over the absolute *Default timeout*. Additionally, the global timeout may be omitted in the JSON configuration in favor of the absolute *Default timeout*, which is also configurable via a property of the C# static class.

12.17.3 QoS timeout

A *Quality of Service* (QoS) *timeout* can be defined using the *QoSOptions Schema* and the QoS *Timeout strategy (Polly)*:

```
"QoSOptions": {
  "Timeout": 5000 // milliseconds
}
```

Please note, the *Quality of Service* timeout takes precedence over both route-level and global timeouts, which are ignored when QoS is enabled. Additionally, avoid defining both *timeouts* in the same route, as the QoS timeout has higher priority than the route-level timeout. Therefore, the following route configuration **is not** recommended:

```
{
  // route props...
```

(continues on next page)

(continued from previous page)

```
  "Timeout": 3, // seconds
  "QoSOptions": {
    "Timeout": 5000 // milliseconds
  }
}
```

So, route `Timeout` will be ignored in favor of QoS `Timeout`. Moreover, because the 3-second duration is shorter than 5000 milliseconds, you may observe warning messages in the logs that begin with the following sentence:

```
Route '/xxx' has Quality of Service settings (QoSOptions) enabled, but either
↪the route Timeout or the QoS Timeout is misconfigured: ...
```

For more details about this warning, refer to the *QoS and route (global) timeouts* note in the *Quality of Service* chapter. Your next recommended action is to completely remove the 3-second `Timeout` property or comment it out:

```
{
  // "Timeout": 3, // seconds
  "QoSOptions": {
    "Timeout": 5000 // milliseconds
  }
}
```

Note

1. Both route `Timeout` and QoS `Timeout` are nullable positive integers, with a minimum valid value of 1. Values in the range `(∞, 0]` are treated as "no value" and will be automatically converted to the absolute *Default timeout*, effectively ignoring the property.

2. The unit of measurement for route `Timeout` is seconds, whereas QoS `Timeout` is measured in milliseconds.

12.17.4 Default timeout

Timeout values defined at different levels in the JSON configuration can serve as fallback defaults for other levels.

- The absolute timeout (also known as `DownstreamRoute` DefaultTimeoutSeconds) defaults to 90 seconds (as defined by the `DownstreamRoute` DefTimeout constant). It acts as the default timeout when neither route-level nor global timeouts are defined.
- The global configuration timeout, if not defined, also defaults to `DownstreamRoute.DefaultTimeoutSeconds`. If defined, it serves as the default timeout for all routes.
- The Quality of Service (QoS) global timeout acts as the default timeout for all routes where QoS is enabled.

To configure the absolute timeout (currently 90 seconds, as defined by the `DownstreamRoute` DefTimeout constant), assign the desired number of seconds to the `DownstreamRoute` DefaultTimeoutSeconds static property in your Program class:

```
using Ocelot.Configuration;

DownstreamRoute.DefaultTimeoutSeconds = 3; // seconds, value must be >= 3
```

However, keep in mind that the absolute timeout has the lowest priority—therefore, route-level and global timeouts will override this C# property if they are defined.

CHAPTER

THIRTEEN

DELEGATING HANDLERS

MS Learn Documentation:

- DelegatingHandler Class
- HTTP Message Handlers in ASP.NET Web API
- HttpClient Message Handlers in ASP.NET Web API

Ocelot allows the user to add delegating handlers to the `HttpClient` transport.[1]

13.1 Configuration

In order to utilize the *Delegating Handlers* feature, you need to do the following three steps of configuration.

1. Create a class that can be used as a *delegating handler*: it must inherit from the `DelegatingHandler` class. We are going to register these handlers in the ASP.NET Core DI container, so you can inject any other services you have registered into the constructor of your handler.

```
public class MyHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage>␣
↪SendAsync(HttpRequestMessage request, CancellationToken token)
    {
        // Do stuff before sending request, and optionally call the base␣
↪handler...
        var response = await base.SendAsync(request, token);
        // Do post-processing of the response...
        return response;
    }
}
```

2. You must add the handlers to the DI container in your Program, as shown below:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddDelegatingHandler<MyHandler>()
    .AddDelegatingHandler<MyHandlerTwo>();
```

Both of these `AddDelegatingHandler{T}` methods have an optional parameter called `global`, which is set to `false`. If it is `false`, then the intent of the *delegating handler* is to be applied to

[1] This feature was requested in issue 208, and the team decided that it would be useful in various ways, releasing it in version 3.0.3. Since then, we extended it in issue 264 and released it in version 5.0.0.

specific routes via ocelot.json (see step 3). If it is set to `true`, then it becomes a global handler and will be applied to all routes, as shown below:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddDelegatingHandler<MyGlobalHandler>(true);  // it's global!
```

Note 1: The generic `AddDelegatingHandler<T>(bool)` method has another overloaded non-generic one with the `Type` parameter: `AddDelegatingHandler(Type, bool)`. Thus, here is an alternative to set it up:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddDelegatingHandler(typeof(MyHandler)) // for selected routes␣
↪only
    .AddDelegatingHandler(typeof(MyGlobalHandler), true); // it's␣
↪global!
```

Note 2: Both versions of the methods add transient services to the DI container. It is recommended to utilize the generic version.

3. If you want route-specific *delegating handlers* or to order your specific and/or global *delegating handlers* (more on this in the *Execution Order* section), then you must add the following to the specific route in ocelot.json. The names in the array must match the class names of your *delegating handlers* for Ocelot to match them together:

```
"DelegatingHandlers": [ "MyHandlerTwo", "MyHandler" ]
```

13.2 Execution Order

You can have as many *delegating handlers* as you want, and they are run in the following order:

1. Any globals that are left in the order they were added to services and are not in the `DelegatingHandlers` option array from ocelot.json.
2. Any non-global *delegating handlers* plus any globals that were in the `DelegatingHandlers` option array from ocelot.json, ordered as they are in the `DelegatingHandlers` array.
3. Tracing *delegating handler*, if enabled (refer to the *Tracing* chapter).
4. Quality of Service *delegating handler*, if enabled (refer to the *Quality of Service* chapter).
5. The `HttpClient` sends the `HttpRequestMessage`.

Hopefully, other people will find this feature useful!

CHAPTER

FOURTEEN

DEPENDENCY INJECTION

Namespace: `Ocelot.DependencyInjection`
Source code: DependencyInjection

14.1 Services Overview

Dependency Injection feature in Ocelot is designed to extend and/or control the building of Ocelot Core as ASP.NET Core pipeline services. The main methods of the ServiceCollectionExtensions class are:

- The *AddOcelot method* adds the required Ocelot services to the DI container and adds default services using the *AddDefaultAspNetServices method*.
- The *AddOcelotUsingBuilder method* adds the required Ocelot services to the DI container and adds custom ASP.NET services with configuration injected implicitly or explicitly.

Use *IServiceCollection extensions* in your Program (ASP.NET Core app) to add and build Ocelot Core services. The fact is, the *OcelotBuilder class* is Ocelot's cornerstone logic.

14.2 `IServiceCollection` extensions

Class: `Ocelot.DependencyInjection.` ServiceCollectionExtensions

Based on the current implementations for the *OcelotBuilder class*, the *AddOcelot method* adds the required ASP.NET services to the DI container. You could call the more extended *AddOcelotUsingBuilder method* while configuring services to build and use a custom builder via an `IMvcCoreBuilder` object.

14.2.1 `AddOcelot` method

Signatures:

```
IOcelotBuilder AddOcelot(this IServiceCollection services);
IOcelotBuilder AddOcelot(this IServiceCollection services, IConfiguration configuration);
```

These `IServiceCollection` extension methods add default ASP.NET services and Ocelot application services with configuration injected implicitly or explicitly.

Note: Both methods add the required and *default* ASP.NET Core services for Ocelot Core in the *AddDefaultAspNetServices method*, which is the default builder.

In this scenario, you do nothing other than call the `AddOcelot` method, which is often mentioned in feature chapters if additional startup settings are required. With this method, you simply reuse the default settings to build the Ocelot Core. The alternative is the `AddOcelotUsingBuilder` method; see the next subsection.

14.2.2 AddOcelotUsingBuilder method

Signatures:

```
using CustomBuilderFunc = System.Func<IMvcCoreBuilder, Assembly, IMvcCoreBuilder>;

IOcelotBuilder AddOcelotUsingBuilder(this IServiceCollection services, CustomBuilderFunc customBuilder);
IOcelotBuilder AddOcelotUsingBuilder(this IServiceCollection services, IConfiguration configuration, CustomBuilderFunc customBuilder);
```

These `IServiceCollection` extension methods add Ocelot application services and **custom** ASP.NET Core services with configuration injected implicitly or explicitly.

> **Note**: The method adds **custom** ASP.NET Core services required for Ocelot Core using a custom builder (aka `customBuilder` parameter). It is highly recommended to read the documentation of the *AddDefaultAspNetServices method*, or even review the implementation to understand the default ASP.NET Core services which are the minimal part of the gateway pipeline.

In this custom scenario, you control everything during the ASP.NET Core build process, and you provide custom settings to build Ocelot Core.

14.3 OcelotBuilder class

The OcelotBuilder class is the core of Ocelot which does the following:

- Contructs itself by single public constructor:

```
public OcelotBuilder(IServiceCollection services, IConfiguration configurationRoot, Func<IMvcCoreBuilder, Assembly, IMvcCoreBuilder> customBuilder = null);
```

- Initializes and stores public properties: `Services` (of `IServiceCollection` type), `Configuration` (of `IConfiguration` type), and `MvcCoreBuilder` (of `IMvcCoreBuilder` type).
- Adds *all application services* during the construction phase via the `Services` property.
- Adds ASP.NET Core services by builder using `Func<IMvcCoreBuilder, Assembly, IMvcCoreBuilder>` object in these 2 development scenarios:
- Adds ASP.NET Core services by builder using a `Func<IMvcCoreBuilder, Assembly, IMvcCoreBuilder>` object in these two development scenarios:
  1. By default builder (*AddDefaultAspNetServices method*) if there is no `customBuilder` parameter provided.
  2. By *Custom Builder* with the provided delegate object as the `customBuilder` parameter.
- Adds (switches on/off) Ocelot features through the following methods:
  - `AddSingletonDefinedAggregator` and `AddTransientDefinedAggregator` methods
  - `AddCustomLoadBalancer` method
  - `AddDelegatingHandler` method
  - `AddConfigPlaceholders` method

14.3.1 `AddDefaultAspNetServices` method

Part of the *OcelotBuilder class*

Currently, the method is protected, and overriding is forbidden. The role of the method is to inject the required services via both the `IServiceCollection` and `IMvcCoreBuilder` interface objects for the minimal part of the gateway pipeline.

Current implementation is the folowing:

```
protected IMvcCoreBuilder AddDefaultAspNetServices(IMvcCoreBuilder builder,
↪Assembly assembly)
{
    Services
        .AddLogging()
        .AddMiddlewareAnalysis()
        .AddWebEncoders();
    return builder
        .AddApplicationPart(assembly)
        .AddControllersAsServices()
        .AddAuthorization()
        .AddNewtonsoftJson();
}
```

The method cannot be overridden. It is not virtual, and there is no way to override the current behavior by inheritance. The method is the default builder of Ocelot Core when calling the *AddOcelot method*. As an alternative, to "override" this default builder, you can design and reuse a custom builder as a `Func<IMvcCoreBuilder, Assembly, IMvcCoreBuilder>` delegate object and pass it as a parameter to the *AddOcelotUsingBuilder method*. It gives you full control over the design and building of Ocelot Core, but be careful when designing your custom Ocelot pipeline as a customizable ASP.NET Core pipeline.

Warning: Most of the services from the minimal part of the pipeline should be reused, but only a few services can be removed.

Warning: The method above is called after adding the required services of the ASP.NET Core pipeline by the AddMvcCore method via the `Services` property in the upper calling context. These services are the absolute minimum core services for the ASP.NET MVC pipeline. They must always be added to the DI container and are added implicitly before calling the method by the caller in the upper context. So, `AddMvcCore` creates an `IMvcCoreBuilder` object and assigns it to the `MvcCoreBuilder` property. Finally, as a default builder, the method above receives the `IMvcCoreBuilder` object, making it ready for further extensions.

The next section shows you an example of designing a custom Ocelot Core using a custom builder.

14.4 Custom Builder

Goal: Replace `Newtonsoft.Json` services with `System.Text.Json` services.

14.4.1 Problem

The main *AddOcelot method* adds Newtonsoft JSON services using the `AddNewtonsoftJson` extension method in the default builder (*AddDefaultAspNetServices method*). The `AddNewtonsoftJson` method was introduced in earlier .NET and Ocelot releases, which was necessary before Microsoft launched the `System.Text.Json` library. However, it now affects normal use, so we intend to solve the problem.

Modern JSON services out of the box will help configure JSON settings using the JsonSerializerOptions property for JSON formatters during (de)serialization.

14.4.2 Solution

We have the following methods in ServiceCollectionExtensions class:

```
IOcelotBuilder AddOcelotUsingBuilder(this IServiceCollection services, Func
↪<IMvcCoreBuilder, Assembly, IMvcCoreBuilder> customBuilder);
IOcelotBuilder AddOcelotUsingBuilder(this IServiceCollection services,␣
↪IConfiguration configuration, Func<IMvcCoreBuilder, Assembly,␣
↪IMvcCoreBuilder> customBuilder);
```

These methods with a custom builder allow you to use any desired JSON library for (de)serialization. However, we are going to create a custom MvcCoreBuilder with support for JSON services, such as System.Text.Json. To do that, we need to call the AddJsonOptions extension of the MvcCoreMvcCoreBuilderExtensions class (NuGet Microsoft.AspNetCore.Mvc.Core package) in Program:

```
builder.Services
    .AddLogging()
    .AddMiddlewareAnalysis()
    .AddWebEncoders()
    // Add your custom builder
    .AddOcelotUsingBuilder(builder.Configuration, MyCustomBuilder);

static IMvcCoreBuilder MyCustomBuilder(IMvcCoreBuilder builder, Assembly␣
↪assembly) => builder
    .AddApplicationPart(assembly)
    .AddControllersAsServices()
    .AddAuthorization()
    // Replace AddNewtonsoftJson() by AddJsonOptions()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.WriteIndented = true; // use System.
↪Text.Json
    });
```

The sample code provides settings to render JSON as indented text rather than as compressed plain JSON text without spaces. This is just one common use case, and you can add additional services to the builder.

14.5 Configuration Overview

Dependency Injection for the *Configuration* feature in Ocelot is designed to extend and set up the configuration of the Ocelot Core **before** the stage of building ASP.NET Core services (see *Services Overview*). To configure the Ocelot Core services, use the *IConfigurationBuilder extensions* in your Program of your gateway app.

14.6 IConfigurationBuilder extensions

Class: Ocelot.DependencyInjection. ConfigurationBuilderExtensions

The main methods are the *AddOcelot methods* within the ConfigurationBuilderExtensions class. These methods have a list of overloaded versions with corresponding signatures.

The purpose of the AddOcelot method is to prepare everything before actually configuring with native extensions. It involves the following steps:

1. **Merging Partial JSON Files**: The GetMergedOcelotJson method merges partial JSON files.
2. **Selecting Merge Type**: It allows you to choose a merge type to save the merged JSON configuration data either ToFile or ToMemory.
3. **Framework Extensions**: Finally, the method calls the following native IConfigurationBuilder framework extensions:

- The AddJsonFile method adds the primary configuration file (commonly known as ocelot.json) after the merge stage. It writes the file back *to the file system* using the ToFile merge type option, which is implicitly the default.
- The AddJsonStream method adds the JSON data of the primary configuration file as a UTF-8 stream *into memory* after the merge stage. It uses the ToMemory merge type option.

14.6.1 AddOcelot methods

Signatures of the most common versions:

```
IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder, IWebHostEnvironment env);
IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder, string folder, IWebHostEnvironment env);
```

Note: These versions use the implicit ToFile merge type to write ocelot.json back to disk. Finally, they call the AddJsonFile extension.

Signatures of the versions to specify a MergeOcelotJson option:

```
IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder, IWebHostEnvironment env, MergeOcelotJson mergeTo,
    string primaryConfigFile = null, string globalConfigFile = null, string environmentConfigFile = null, bool? optional = null, bool? reloadOnChange = null);
IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder, string folder, IWebHostEnvironment env, MergeOcelotJson mergeTo,
    string primaryConfigFile = null, string globalConfigFile = null, string environmentConfigFile = null, bool? optional = null, bool? reloadOnChange = null);
```

Note: These versions include optional arguments to specify the location of the three main files involved in the merge operation. In theory, these files can be located anywhere, but in practice, it is better to keep them in one folder.

Signatures of the versions to indicate the FileConfiguration object of a self-created out-of-the-box configuration:[1]

[1] The *Build From Scratch* feature was requested in issues 1228 and 1235. It was delivered by PR 1569 as part of version 20.0. Since then, we have extended it in PR 1227 and released it as part of version 23.2.

```
IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder,␣
↪FileConfiguration fileConfiguration,
    string primaryConfigFile = null, bool? optional = null, bool?␣
↪reloadOnChange = null);
IConfigurationBuilder AddOcelot(this IConfigurationBuilder builder,␣
↪FileConfiguration fileConfiguration, IWebHostEnvironment env,␣
↪MergeOcelotJson mergeTo,
    string primaryConfigFile = null, string globalConfigFile = null, string␣
↪environmentConfigFile = null, bool? optional = null, bool? reloadOnChange =␣
↪null);
```

Note 1: These versions include optional arguments to specify the location of the three main files involved in the merge operation.

Note 2: Your `FileConfiguration` object can be serialized/deserialized from anywhere: local or remote storage, Consul KV storage, and even a database. For more information about this super useful feature, please read PR 1569.

CHAPTER FIFTEEN

ERROR HANDLING

MS Learn: Handle errors in ASP.NET Core

Ocelot has custom error handling for `Exception` objects. Thus, we override the standard error handling provided by ASP.NET Core, which is based on manipulating `Exception` objects.

15.1 Middleware

Class: ExceptionHandlerMiddleware

The `ExceptionHandlerMiddleware` produces the following status codes, in fallback order, after setting the *Request ID*:

1. Native response status: Returned when no exception is present, or when a mapped error status is available (excluding `499` and `500`).
2. 499 Client Closed Request: A custom Ocelot status returned when an `OperationCanceledException` occurs due to an aborted request. A warning is logged.
3. 500 Internal Server Error: The standard status returned when a generic `Exception` occurs and Ocelot does not process or map the error. An error record is logged.

Ocelot returns HTTP status codes based on internal logic in specific cases of *Client Error Responses* and *Server Error Responses*.

15.2 Client Error Responses

- 400 Bad Request: If something is wrong with the incoming request, Ocelot generates an `HttpRequestException`, for example:[1]
  - An upstream request header contains non-ASCII characters. RFC 7230 specifies that when a server encounters invalid or unparsable request data, it should respond with a 400 Bad Request status code.
- 401 Unauthorized: If the authentication middleware runs and the user is not authenticated.
- 403 Forbidden: If the authorization middleware runs and the user is unauthorized, if the claim value is not authorized, if the scope is not authorized, if the user does not have the required claim, or if the claim cannot be found.

[1] RFC 7230 ("Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing"), Section 3.2 ("Header Fields"), and especially Section 3.2.4 ("Field Parsing"), describe cases that may arise with request data (where a 400 Bad Request status is preferred) and response data (where a 502 Bad Gateway status is preferred). Ocelot does not perform any special validation of header data for upstream requests or downstream responses; it proxies headers as-is, with the exception of the "*Headers Transformation*" feature. This enhancement was requested in bug 2374, fixed in pull request 2379, and the patch was rolled out as part of the 25.0 release.

- 404 Not Found: If a downstream route cannot be found, or if Ocelot is unable to map an internal error code to an HTTP status code.
- 499 Client Closed Request: If the request is canceled by the client.

  Ocelot Error: RequestCanceledError
  Ocelot Code: OcelotErrorCode.RequestCanceled

  According to Ocelot Core's design, HTTP status code `499` is returned in the following `OperationCanceledException` scenarios:

  1. By `ExceptionHandlerMiddleware`, if an `OperationCanceledException` is thrown and the context's cancellation token is in the "cancellation requested" state. Ocelot logs a warning with the exception body. If the response has not started, the status code will be set to `499`.
  2. By `ResponderMiddleware`, if the default `IErrorsToHttpStatusCodeMapper` service maps the detected OcelotErrorCode.RequestCanceled to status `499`. This error code is produced by the `IExceptionToErrorMapper` service when an `OperationCanceledException` is thrown by other middlewares.

15.3 Server Error Responses

- 500 Internal Server Error: If unable to complete the HTTP request to the downstream service, and the exception is not `OperationCanceledException` or `HttpRequestException`.
- 502 Bad Gateway: If unable to connect to the downstream service.
- 503 Service Unavailable: Returned when the downstream request times out.

  Ocelot Error: RequestTimedOutError
  Ocelot Code: OcelotErrorCode.RequestTimedOutError

  According to Ocelot Core's design, status code `503` is produced in the following `TimeoutException` scenarios:

  1. By `TimeoutDelegatingHandler` from the `IMessageInvokerPool` service, when an `OperationCanceledException` is thrown and the context's cancellation token is not in the "cancellation requested" state. Ocelot does not log an error with the exception body, but the `IExceptionToErrorMapper` service generates the internal OcelotErrorCode.RequestTimedOutError.
  2. By `ResponderMiddleware`, if the default `IErrorsToHttpStatusCodeMapper` service maps the detected OcelotErrorCode.RequestTimedOutError to status `503`. This error code is produced by the `IExceptionToErrorMapper` service when a `TimeoutException` is thrown by other middlewares—especially by `TimeoutDelegatingHandler`.

15.4 Error Mapper

Class: HttpExceptionToErrorMapper

Historically, Ocelot errors are implemented by the Exception-to-Error mapper. The `Map` method converts an `Exception` object to a native `Ocelot.Errors.Error` object.

We override HTTP status codes because of `Exception`-to-`Error` mapping. This can be confusing for the developer since the actual status code of the downstream service may be different and get lost. Please research and review all response headers of the upstream service. If you do not find status codes and/or required headers, then the *Headers Transformation* feature should help.

We expect you to share your use case with us in the Discussions space of the repository.

CHAPTER SIXTEEN

GRAPHQL

Ocelot does not directly support GraphQL, but many people have asked about it. We wanted to show how easy it is to integrate the GraphQL for .NET library.

16.1 Sample

Sample: Ocelot.Samples.GraphQL

Please see the sample project Ocelot.Samples.GraphQL. Using a combination of the graphql-dotnet project and Ocelot *Delegating Handlers* feature, this is pretty easy to do. However, we do not intend to integrate more closely with GraphQL at the moment. Check out the sample's README.md for detailed instructions on how to do this.

16.2 Future

If you have sufficient experience with GraphQL and the mentioned .NET graphql-dotnet package, we would welcome your contribution to the sample.

Who knows, maybe you will get inspired by the sample development and come up with a design solution in the form of a rough draft of a *GraphQL* feature to implement in Ocelot. Good luck! And welcome to the Discussions space of the repository!

CHAPTER

SEVENTEEN

HEADERS TRANSFORMATION

Ocelot allows the user to transform HTTP headers both before and after the downstream request.

> **Note**: *Headers Transformation* is generally available for static routes with a global configuration. For dynamic and aggregate routes, this feature is not implemented. This limitation is noted in the current *Roadmap*.

17.1 Schema

As you may already know from the *Configuration* chapter and the *Route Schema* section, the route's *Headers Transformation* schema is quite simple, a JSON dictionary:

```
"DownstreamHeaderTransform": {
  // "header_name": "transformation_expression",
},
"UpstreamHeaderTransform": {
  // "header_name": "transformation_expression",
},
```

Typically, a `transformation_expression` is a constant header value, a single placeholder from the *Placeholders* list, or a "*Find and Replace*" expression. Additionally, the *Global Configuration Schema* allows configuring global *Headers Transformations* (refer to the *Configuration* section).

17.2 Configuration[1]

A complete *configuration* consists of both route-level and global *Headers Transformations*.

```
{
  "Routes": [
    {
      "DownstreamHeaderTransform": {
        // ...
      },
      "UpstreamHeaderTransform": {
        // ...
      }
    }
  ],
  "GlobalConfiguration": {
```

(continues on next page)

[1] The global *Configuration* feature was requested in issue 1658 and released in version 24.1.

(continued from previous page)

```
    "DownstreamHeaderTransform": {
      // ...
    },
    "UpstreamHeaderTransform": {
      // ...
    }
  }
}
```

Note: Route-level transformations take precedence over global transformations. In addition, when route-level transformations are defined, they do not entirely override the full set of header names from the global configuration. Ocelot's Core internal Merge algorithm identifies global header names not specified at the route level and appends them to the route's header set.

17.3 Find and Replace[2]

In order to transform a header first we specify the header key and then the type of transform we want e.g.

```
"Test": "http://www.bbc.co.uk/, http://ocelot.net/"
```

The key is `Test` and the value is `http://www.bbc.co.uk/, http://ocelot.net/`. The value is saying: replace `http://www.bbc.co.uk/` with `http://ocelot.net/`. The syntax is `{find}, {replace}`. Hopefully pretty simple. There are examples below that explain more.

Pre Downstream Request

Add the following to a Route in ocelot.json in order to replace `http://www.bbc.co.uk/` with `http://ocelot.net/`. This header will be changed before the request downstream and will be sent to the downstream server.

```
"UpstreamHeaderTransform": {
  "Test": "http://www.bbc.co.uk/, http://ocelot.net/"
}
```

Post Downstream Request

Add the following to a Route in ocelot.json in order to replace `http://www.bbc.co.uk/` with `http://ocelot.net/`. This transformation will take place after Ocelot has received the response from the downstream service.

```
"DownstreamHeaderTransform": {
  "Test": "http://www.bbc.co.uk/, http://ocelot.net/"
}
```

17.4 Add to Request[3]

If you want to add a header to your upstream request please add the following to a route in your ocelot.json:

[2] The "*Find and Replace*" feature was requested in issue 190, initially released in version 2.0.11, and the team decided that it would be useful in various ways.

[3] The "*Add to Request*" feature was requested in issue 313 and released in version 5.5.3.

```
"UpstreamHeaderTransform": {
  "Uncle": "Bob"
}
```

In the example above a header with the key `Uncle` and value `Bob` would be send to to the upstream service.

> *Placeholders* are supported too (see below).

17.5 Add to Response[4]

If you want to add a header to your downstream response, please add the following to a route in ocelot.json:

```
"DownstreamHeaderTransform": {
  "Uncle": "Bob"
}
```

In the example above a header with the key `Uncle` and value `Bob` would be returned by Ocelot when requesting the specific route.

If you want to return the *Butterfly* Trace ID, do something like the following:

```
"DownstreamHeaderTransform": {
  "AnyKey": "{TraceId}"
}
```

17.6 Placeholders

Ocelot allows placeholders that can be used in header transformation.

Placeholder	*Description*
`{BaseUrl}`	This will use Ocelot base URL e.g. `http://localhost:5000` as its value.
`{DownstreamBaseUrl}`	This will use the downstream services base URL e.g. `http://localhost:5000` as its value. This only works for `DownstreamHeaderTransform` route option at the moment.
`{RemoteIpAddress}`	This will find the clients IP address using `HttpContext.Connection.RemoteIpAddress`, so you will get back some IP. See more in the GetRemoteIpAddress method.
`{TraceId}`	This will use the *Butterfly* Trace ID. This only works for `DownstreamHeaderTransform` route option at the moment.
`{UpstreamHost}`	This will look for the incoming `Host` header.

For now, we believe these placeholders are sufficient for basic user scenarios. However, if you need additional placeholders, refer to the *Roadmap*.

17.7 Samples

17.7.1 Handling 302 redirects

Ocelot will by default automatically follow redirects, however if you want to return the location header to the client, you might want to change the location to be Ocelot not the downstream service. Ocelot allows

[4] The "*Add to Response*" feature was requested in issue 280 and released in version 5.1.0.

this with the following configuration:

```
"DownstreamHeaderTransform": {
  "Location": "http://www.bbc.co.uk/, http://ocelot.net/"
},
"HttpHandlerOptions": {
  "AllowAutoRedirect": false,
}
```

Or, you could use the `{BaseUrl}` placeholder.

```
"DownstreamHeaderTransform": {
  "Location": "http://localhost:6773, {BaseUrl}"
},
"HttpHandlerOptions": {
  "AllowAutoRedirect": false,
}
```

Finally, if you are using a load balancer with Ocelot, you will get multiple downstream base URLs so the above would not work. In this case you can do the following:

```
"DownstreamHeaderTransform": {
  "Location": "{DownstreamBaseUrl}, {BaseUrl}"
},
"HttpHandlerOptions": {
  "AllowAutoRedirect": false,
}
```

17.7.2 `X-Forwarded-For` header

An example of using `{RemoteIpAddress}` placeholder:

```
"UpstreamHeaderTransform": {
  "X-Forwarded-For": "{RemoteIpAddress}"
}
```

17.8 Roadmap

1. Ideally the "*Find and Replace*" feature would be able to support the fact that a header can have multiple values. At the moment it just assumes one. It would also be nice if it could multi find and replace e.g.

   ```
   "DownstreamHeaderTransform": {
     "Location": "[{one,one},{two,two}]"
   },
   "HttpHandlerOptions": {
     "AllowAutoRedirect": false,
   }
   ```

2. The *Headers Transformation* feature is not implemented for *Dynamic Routes* and *Aggregate Routes*. For *Dynamic Routing*, potential development would require moderate effort. However, the Ocelot team expects that designing and implementing *Headers Transformation* for *Aggregation* will demand significant effort, as aggregated routes typically lose their headers.

Ideas and proposals are welcome in the repository's Discussions space.

CHAPTER

EIGHTEEN

KUBERNETES (K8S)[1]

Feature of: *Service Discovery*
Quick Links: K8s Website | K8s Documentation | K8s GitHub

Ocelot will call the K8s endpoints API in a given namespace to get all of the endpoints for a pod and then load balance across them. Ocelot used to use the services API to send requests to the K8s service but this was changed in pull request 1134 because the service did not load balance as expected.

18.1 Install

The first thing you need to do is install the package that provides support in Ocelot:

```
dotnet add package Ocelot.Discovery.KubeClient
```

> **Note**
>
> Our NuGet Ocelot.Discovery.KubeClient extension package is based on the KubeClient package. For a comprehensive understanding, it is essential refer to the KubeClient documentation.

> **Warning**
>
> Prior to version 25.0, the package was named Ocelot.Provider.Kubernetes. If you are using version 24.1 or earlier, install the Ocelot.Provider.Kubernetes package. For version 25.0 and later, the package ID is Ocelot.Discovery.KubeClient.

18.2 `AddKubernetes(bool)` method

```
public static class OcelotBuilderExtensions
{
    public static IOcelotBuilder AddKubernetes(this IOcelotBuilder builder,
```

(continues on next page)

[1] The " *Kubernetes (K8s)* " feature was requested as part of issue 345 to add support for Kubernetes *Service Discovery* provider, and released in version 13.5.0

(continued from previous page)

```
↪bool usePodServiceAccount = true);
}
```

This extension-method adds K8s services **with** or **without** using a pod service account. Then add the following to your Program:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddKubernetes(); // usePodServiceAccount is true
```

If you have services deployed in Kubernetes, you will normally use the naming service to access them.

1. By default the `useServiceAccount` argument is true, which means that Service Account using Pod to access the service of the K8s cluster needs to be Service Account based on RBAC authorization:

   You can replicate a Permissive using RBAC role bindings (see Permissive RBAC Permissions), K8s API server and token will read from pod.

   ```
   kubectl create clusterrolebinding permissive-binding --clusterrole=cluster-
   ↪admin --user=admin --user=kubelet --group=system:serviceaccounts
   ```

   Finally, it creates the KubeClient from pod service account.

2. When the `useServiceAccount` argument is false, you need to provide KubeClientOptions to create KubeClient using them. You have to bind the options configuration section for the DI `IOptions<KubeClientOptions>` interface or register a custom action to initialize the options:

   ```
   Action<KubeClientOptions> configureKubeClient = opts =>
   {
       opts.ApiEndPoint = new UriBuilder("https", "my-host", 443).Uri;
       opts.AccessToken = "my-token";
       opts.AuthStrategy = KubeAuthStrategy.BearerToken;
       opts.AllowInsecure = true;
   };
   builder.Services
       .AddOptions<KubeClientOptions>()
       .Configure(configureKubeClient); // manual binding options via␣
   ↪IOptions<KubeClientOptions>
   builder.Services
       .AddOcelot(builder.Configuration)
       .AddKubernetes(false); // don't use pod service account, and IOptions
   ↪<KubeClientOptions> is reused
   ```

   Note, this could also be written like this (shortened version):

   ```
   builder.Services
       .AddKubeClientOptions(opts =>
       {
           opts.ApiEndPoint = new UriBuilder("https", "my-host", 443).
   ↪Uri;
           opts.AuthStrategy = KubeAuthStrategy.BearerToken;
           opts.AccessToken = "my-token";
           opts.AllowInsecure = true;
       })
   ```

(continues on next page)

(continued from previous page)

```
    .AddOcelot(builder.Configuration)
    .AddKubernetes(false); // don't use pod service account, and
↪client options provided via AddKubeClientOptions
```

Finally, it creates the KubeClient from your options.

Note 1: For understanding the `IOptions<TOptions>` interface, please refer to the Microsoft Learn documentation: Options pattern in .NET.

Note 2: Please consider this Case 2 as an example of manual setup when you **do not** use a pod service account. We recommend using our official extension method, which receives an `Action<KubeClientOptions>` argument with your options: refer to the *AddKubernetes(Action<KubeClientOptions>) method 2* below.

18.3 AddKubernetes(Action<KubeClientOptions>) method[2]

```
public static class OcelotBuilderExtensions
{
    public static IOcelotBuilder AddKubernetes(this IOcelotBuilder builder,
↪Action<KubeClientOptions> configureOptions, /*optional params*/);
}
```

This extension method adds K8s services **without** using a pod service account, explicitly calling an action to initialize configuration options for KubeClient. It operates in two modes:

1. If `configureOptions` is provided (action is not null), it calls the action, ignoring all optional arguments.

   ```
   Action<KubeClientOptions> configureKubeClient = opts =>
   {
       opts.ApiEndPoint = new UriBuilder("https", "my-host", 443).Uri;
       // ...
   };
   builder.Services
       .AddOcelot(builder.Configuration)
       .AddKubernetes(configureKubeClient); // without optional arguments
   ```

   Note: Optional arguments do not make sense; all settings are defined inside the `configureKubeClient` action.

2. If `configureOptions` is not provided (action is null), it reads the global `ServiceDiscoveryProvider` *Configuration* options and reuses them to initialize the following properties: `ApiEndPoint`, `AccessToken`, and `KubeNamespace`, finally initializing the rest of the properties with optional arguments.

   ```
   builder.Services
       .AddOcelot(builder.Configuration)
       .AddKubernetes(null, allowInsecure: true, /*optional args*/) //
   ↪shortened version
       // or
       .AddKubernetes(configureOptions: null, allowInsecure: true, /*optional
   ↪args*/); // long version
   ```

[2] The "*AddKubernetes(Action{KubeClientOptions}) method*" was requested as part of issue 2255 (pull request 2257), and released in version 24.0

Note: Optional arguments must be used here in addition to the options coming from the global `ServiceDiscoveryProvider` *Configuration*. Find the comprehensive documentation in the C# code of the AddKubernetes methods.

18.4 Configuration

The following examples show how to set up a route that will work in Kubernetes. The most important thing is the `ServiceName` which is made up of the Kubernetes service name. We also need to set up the `ServiceDiscoveryProvider` in `GlobalConfiguration`.

Regarding global and route configurations, if your downstream service resides in a different namespace, you can override the global setting at the route level by specifying a `ServiceNamespace`.

```
"Routes": [
  {
    "ServiceName": "my-service",
    "ServiceNamespace": "my-namespace"
  }
]
```

18.5 Kube provider

The example here shows a typical configuration:

```
"Routes": [
  {
    "ServiceName": "my-service",
    // ...
  }
],
"GlobalConfiguration": {
  "ServiceDiscoveryProvider": {
    "Scheme": "https",
    "Host": "my-host",
    "Port": 443,
    "Token": "my-token",
    "Namespace": "Dev",
    "Type": "Kube"
  }
}
```

Service deployment in `Dev` namespace, and discovery provider type is `Kube`, you also can set *PollKube* or *WatchKube* provider type.

Note 1: `Scheme`, `Host`, `Port`, and `Token` are not used if `usePodServiceAccount` is true when KubeClient is created from a pod service account. Please refer to the *Install* section for technical details.

Note 2: The `Kube` provider searches for the service entry using `ServiceName` and then retrieves the first available port from the `EndpointSubsetV1.Ports` collection. Therefore, if the port name is not specified, the default downstream scheme will be `http`; Please refer to the “*Downstream Scheme vs Port Names*” section for technical details.

18.6 PollKube provider[3]

You use Ocelot to poll Kubernetes for latest service information rather than per request. If you want to poll Kubernetes for the latest services rather than per request (default behaviour of the *Kube provider*) then you need to set the following configuration:

```
"ServiceDiscoveryProvider": {
  "Namespace": "dev",
  "Type": "PollKube",
  "PollingInterval": 100 // ms
}
```

The polling interval is in milliseconds and tells Ocelot how often to call Kubernetes for changes in service configuration.

> **Note**, there are tradeoffs here. If you poll Kubernetes, it is possible Ocelot will not know if a service is down depending on your polling interval and you might get more errors than if you get the latest services per request. This really depends on how volatile your services are. We doubt it will matter for most people and polling may give a tiny performance improvement over calling Kubernetes per request. There is no way for Ocelot to work these out for you, except perhaps through a discussion.

18.7 WatchKube provider[4]

With this configuration, Kubernetes API "watch requests" are used to fetch service configuration. Essentially, it establishes one streamed HTTP connection with the Kubernetes API per downstream service. Changes streamed through this connection will be used to update the list of available endpoints.

```
"ServiceDiscoveryProvider": {
  "Namespace": "dev",
  "Type": "WatchKube"
}
```

> **Note**
>
> The WatchKube provider is specifically designed for high-load Ocelot vs. Kubernetes environments with high RPS ratios. To better understand which type is suitable for your needs, we have added a table *Comparing providers*.

The provider has an implicit configuration for fine-tuned watching, which are available and can only be initialized in C# code.

- `WatchKube.FirstResultsFetchingTimeoutSeconds`: This is the default number of seconds to wait after Ocelot starts, following the provider's creation, to fetch the first result from the Kubernetes endpoint. [1]
- `WatchKube.FailedSubscriptionRetrySeconds`: This is the default number of seconds to wait before scheduling the next retry for the subscription operation. [1]

  [1] For both `static int` properties, the default value is 1 (one) second. The constraint ensures that the assigned value is greater than or equal to 1 (one). Therefore, the minimum value is 1

[3] The evolution of the "*PollKube provider*" began with pull request 772 (version 13.2.0). Since then, the provider's design was reviewed due to reported bug 2304, and patch 2335 was applied and rolled out in version 24.1.

[4] The "*WatchKube provider*" was first discussed in thread 2168, later implemented in pull request 2174, and released in version 24.1.

(one) second.

18.8 Comparing providers

This table explains the most important indicators that may influence Ocelot vs. Kubernetes deployment or DevOps strategy. The evolution path of all providers follows: `Kube` -> `PollKube` -> `WatchKube`, with `WatchKube` being the most advanced provider.

Indicators \ Providers	*Kube*	*PollKube*	*WatchKube*
Extra latency	One hop per route	-	-
Speed of response to endpoints changes	High	Low [1]	High
Pressure on Kubernetes API	High	Low [1]	Low
Ocelot load (estimated) [2]	< 1000 RPS	> 1000 RPS	> 5000 RPS
Ocelot deployment [3]	Single instance	Multiple instances	Cluster of instances

[1] Depends on the `PollingInterval` option.

[2] Please consider this a rough load estimation, as our team has not provided any tests or benchmarks.

[3] The term "instance" refers to an Ocelot instance, not a Kubernetes one.

18.9 Downstream Scheme vs Port Names[5]

Kubernetes configuration permits the definition of multiple ports with names for each address of an endpoint subset. When binding multiple ports, you assign a name to each subset port. To allow the `Kube` provider to recognize the desired port by its name, you need to specify the `DownstreamScheme` with the port's name; if not, the collection's first port entry will be chosen by default.

For instance, consider a service on Kubernetes that exposes two ports: `https` for 443 and `http` for 80, as follows:

```
Name:         my-service
Namespace:    default
Subsets:
  Addresses:  10.1.161.59
  Ports:
    Name   Port  Protocol
    ----   ----  --------
    https  443   TCP
    http   80    TCP
```

When you need to use the `http` port while intentionally bypassing the default `https` port (first one), you must define `DownstreamScheme` to enable the provider to recognize the desired `http` port by comparing `DownstreamScheme` with the port name as follows:

```
"Routes": [
  {
    "ServiceName": "my-service",
    "DownstreamScheme": "http", // port name -> http -> port is 80
  }
]
```

[5] The "*Downstream Scheme vs Port Names*" feature was requested as part of issue 1967 and released in version 23.3

Note

In the absence of a specified `DownstreamScheme` (the default behavior), the *Kube provider*—as well as other providers—will select *the first available port* from the `EndpointSubsetV1.Ports` collection. Consequently, if the port name is not designated, the default downstream scheme utilized will be `http`.

CHAPTER

NINETEEN

LOAD BALANCER

Ocelot can load balance across available downstream services for each route. This means you can scale your downstream services, and Ocelot can use them effectively.

19.1 LoadBalancerOptions Schema

Class: FileLoadBalancerOptions

The following is the full *load balancer* configuration, used in both the *Route Schema* and the *Dynamic Route Schema*. Not all of these options need to be configured; however, the `Type` option is mandatory.

```
"LoadBalancerOptions": {
  "Type": "",
  "Key": "", // CookieStickySessions balancer
  "Expiry": 1 // ms, CookieStickySessions balancer
}
```

Option	*Description*
`Type`	An in-built *load balancer* type selected from the list of available *Balancers*, or a user-defined type (refer to the "*Custom Balancers*" section).
`Key`	The name of the cookie you wish to use for sticky sessions. This option is applicable only to the *CookieStickySessions type*.
`Expiry`	Expiration period specifies how long, in milliseconds, the session should remain sticky. This value refreshes with each request to mimic typical session behavior. Note: This option applies only to the *CookieStickySessions type*.

The actual `LoadBalancerOptions` schema with all the properties can be found in the C# FileLoadBalancerOptions class.

19.2 Configuration

The following shows how to set up multiple downstream services for a static route using ocelot.json and then select the `LeastConnection` *load balancer*. This is the simplest way to configure load balancing without using service discovery.

```
{
  "UpstreamPathTemplate": "/posts/{postId}",
  "UpstreamHttpMethod": [ "Put", "Delete" ],
  "DownstreamPathTemplate": "/api/posts/{postId}",
```

(continues on next page)

(continued from previous page)

```
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    { "Host": "10.0.1.10", "Port": 5000 },
    { "Host": "10.0.1.11", "Port": 5000 }
  ],
  "LoadBalancerOptions": {
    "Type": "LeastConnection"
  }
}
```

The following shows how to set up a route using *Service Discovery* and then select the `RoundRobin` *load balancer*.

```
{
  // ...
  "ServiceName": "product",
  "LoadBalancerOptions": {
    "Type": "RoundRobin"
  }
}
```

When this is set up, Ocelot will look up the downstream host and port from the *Service Discovery* provider and load balance requests across any available services. If you add and remove services from the *Service Discovery* provider[1], Ocelot should respect this and stop calling services that have been removed and start calling services that have been added.

19.2.1 Global Configuration[2]

A complete configuration consists of both route-level and global *load balancing*. You can configure the following options in the `GlobalConfiguration` section of ocelot.json:

```
"Routes": [
  {
    "Key": "R0", // optional
    "LoadBalancerOptions": {
      "Type": "CookieStickySessions",
      "Key": ".AspNetCore.Session",
      "Expiry": 1200000 // milliseconds, 20 minutes
    }
  },
  {
    "Key": "R1", // this route is part of a group
    "LoadBalancerOptions": {} // optional due to grouping
  }
],
"GlobalConfiguration": {
  "BaseUrl": "https://ocelot.net",
  "LoadBalancerOptions": {
    "RouteKeys": ["R1"], // if undefined or empty array, opts will apply to␣
↪all routes
```

(continues on next page)

[1] Currently supported *Service Discovery* providers are *Consul*, *Kubernetes*, *Eureka*, *Service Fabric*, and manually developed *Custom Providers*.
[2] The "*Global Configuration*" feature, as part of issue 585, was introduced in pull request 2324 and released in version 24.1.

(continued from previous page)

```
    "Type": "LeastConnection"
  }
}
```

Service Discovery dynamic routes intentionally override the global *dynamic routing* configuration:

```
"DynamicRoutes": [
  {
    "Key": "", // optional
    "ServiceName": "my-service",
    "LoadBalancerOptions": {
      "Type": "LeastConnection" // switch from RoundRobin to LeastConnection
    }
  }
],
"GlobalConfiguration": {
  "BaseUrl": "https://ocelot.net",
  "DownstreamScheme": "http",
  "ServiceDiscoveryProvider": {
    // required section for dynamic routing
  },
  "LoadBalancerOptions": {
    "RouteKeys": [], // no grouping, thus opts apply to all dynamic routes
    "Type": "RoundRobin"
  }
}
```

In this configuration, the `RoundRobin` balancer is used for all implicit dynamic routes. However, for the "my-service" service, the load balancer type has been explicitly switched from `RoundRobin` to `LeastConnection`.

Note

1. If the RouteKeys option is not defined or the array is empty in the global LoadBalancerOptions, the global options will apply to all routes. If the array contains route keys, it defines a single group of routes to which the global options apply. Routes excluded from this group must specify their own route-level LoadBalancerOptions.

2. Prior to version 24.1, global LoadBalancerOptions were only accessible in the special *Dynamic Routing* mode. Since version 24.1, global configuration has been available for both static and dynamic routes. As a team, we would consider the idea of implementing such a global configuration for aggregated routes. However, an aggregated route is essentially a combination of static routes.

19.3 Balancers

The available types of built-in *load balancers* are:

Type	*Description*
CookieStickySession	This uses a cookie to stick all requests to a specific server. More information can be found in the "*CookieStickySessions Type*" section.
LeastConnection	This tracks which services are dealing with requests and sends new requests to the service with the fewest ("least") existing requests. The algorithm state is not distributed across a cluster of Ocelots.
RoundRobin	This loops through available services and sends requests. The algorithm state is not distributed across a cluster of Ocelots.
NoLoadBalancer	This takes the first available service from *configuration* or *Service Discovery* provider.

You must choose which *load balancer* to use in your *configuration*.

19.4 CookieStickySessions Type[3]

We have implemented a basic sticky session type of *load balancer*. The scenario it is meant to support involves having a number of downstream servers that do not share session state. If you receive more than one request for one of these servers, it should go to the same server each time; otherwise, the session state might be incorrect for the given user.

In order to set up the CookieStickySessions *load balancer*, you need to do something like the following:

```
{
  "UpstreamPathTemplate": "/posts/{postId}",
  "UpstreamHttpMethod": [ "Put", "Delete" ],
  "DownstreamPathTemplate": "/api/posts/{postId}",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    { "Host": "10.0.1.10", "Port": 5000 },
    { "Host": "10.0.1.11", "Port": 5000 }
  ],
  "LoadBalancerOptions": {
```

(continues on next page)

[3] The "*CookieStickySessions Type*" feature was requested in issue 322, though what the user wants is more complicated than just sticky sessions. Anyway, we thought this would be a nice feature to have! Initially, the feature was released in version 6.0.0.

(continued from previous page)

```
    "Type": "CookieStickySessions",
    "Key": ".AspNetCore.Session",
    "Expiry": 1200000 // milliseconds, 20 minutes
  }
}
```

These LoadBalancerOptions configure the CookieStickySessions load balancer using the standard session cookie Key for ASP.NET Core apps with sessions enabled. The default expiration time is 20 minutes, matching the default session timeout in ASP.NET Core.

> **Note 1**: If you have multiple routes with the same LoadBalancerOptions, then all of those routes will use the same *load balancer* for their subsequent requests. This means the sessions will be stuck across routes.
>
> **Note 2**: If you define more than one DownstreamHostAndPort, or if you are using a *Service Discovery* provider such as *Consul* and it returns more than one service, then CookieStickySessions uses RoundRobin to select the next server. This is hard-coded at the moment but could be changed.

19.5 Custom Balancers[4]

In order to create and use a custom *load balancer*, you can do the following. Below, we set up a basic load balancing configuration, and note that the Type is MyLoadBalancer, which is the name of a class we will set up to perform load balancing.

```
{
  // ...
  "DownstreamHostAndPorts": [
    { "Host": "10.0.1.10", "Port": 5000 },
    { "Host": "10.0.1.11", "Port": 5000 }
  ],
  "LoadBalancerOptions": {
    "Type": "MyLoadBalancer"
  }
}
```

Then, you need to create a class that implements the ILoadBalancer interface. Below is a simple round-robin example:

```
using Ocelot.LoadBalancer.LoadBalancers;
using Ocelot.Responses;
using Ocelot.Values;

public class MyLoadBalancer : ILoadBalancer
{
    private readonly Func<Task<List<Service>>> _services;
    private static object Locker = new();
    private int _last;

    public MyLoadBalancer() { }
```

(continues on next page)

[4] The "*Custom Balancers*" feature by David Lievrouw implemented a way to provide Ocelot with a custom *load balancer* in pull request 1155 (issue 961, released in version 15.0.3).

(continued from previous page)

```
    public MyLoadBalancer(Func<Task<List<Service>>> services)
        => _services = services;

    public string Type => nameof(MyLoadBalancer);
    public void Release(ServiceHostAndPort hostAndPort) { }

    public async Task<Response<ServiceHostAndPort>> LeaseAsync(HttpContext␣
↪context)
    {
        var services = await _services.Invoke();
        lock (Locker)
        {
            _last = (_last >= services.Count) ? 0 : _last;
            var next = services[_last++];
            return new OkResponse<ServiceHostAndPort>(next.HostAndPort);
        }
    }
}
```

Finally, you need to register this class with Ocelot. We have used the most complex example below to show all of the data and types that can be passed into the factory that creates *load balancers*.

```
using Ocelot.Configuration;
using Ocelot.DependencyInjection;
using Ocelot.ServiceDiscovery.Providers;

Func<IServiceProvider, DownstreamRoute, IServiceDiscoveryProvider,␣
↪MyLoadBalancer> lbFactory
    = (serviceProvider, Route, discoveryProvider) => new␣
↪MyLoadBalancer(discoveryProvider.GetAsync);
builder.Services
    .AddOcelot(builder.Configuration)
    .AddCustomLoadBalancer(lbFactory);
```

However, there is a much simpler example that will work the same way:

```
using Ocelot.DependencyInjection;

builder.Services
    .AddOcelot(builder.Configuration)
    .AddCustomLoadBalancer<MyLoadBalancer>();
```

Note

1. There are numerous `IOcelotBuilder` methods to add a custom *load balancer*. The interface is as follows:

```
IOcelotBuilder AddCustomLoadBalancer<T>()
    where T : ILoadBalancer, new();
IOcelotBuilder AddCustomLoadBalancer<T>(Func<T> loadBalancerFactoryFunc)
    where T : ILoadBalancer;
IOcelotBuilder AddCustomLoadBalancer<T>(Func<IServiceProvider, T>␣
↪loadBalancerFactoryFunc)
```

```
    where T : ILoadBalancer;
IOcelotBuilder AddCustomLoadBalancer<T>(Func<DownstreamRoute,␣
↪IServiceDiscoveryProvider, T> loadBalancerFactoryFunc)
    where T : ILoadBalancer;
IOcelotBuilder AddCustomLoadBalancer<T>(Func<IServiceProvider,␣
↪DownstreamRoute, IServiceDiscoveryProvider, T> loadBalancerFactoryFunc)
    where T : ILoadBalancer;
```

2. When you enable custom *load balancers*, Ocelot looks up your *load balancer* by its class name when it decides whether to perform load balancing.

- If it finds a match, it will use your load balancer to load balance.
- If Ocelot cannot match the *load balancer* type in your configuration with the name of the registered *load balancer* class, then you will receive an HTTP 500 Internal Server Error.
- If your *load balancer* factory throws an exception when Ocelot calls it, you will receive an HTTP 500 Internal Server Error.

Warning

Remember, if you specify no *load balancer* in your *Configuration*, Ocelot will not attempt to load balance.

CHAPTER

TWENTY

LOGGING

MS Learn: Logging in .NET Core and ASP.NET Core

Interfaces: ILoggerFactory and ILogger<T>

Ocelot uses the standard ASP.NET Core logging interfaces `ILoggerFactory` and `ILogger<T>` at the moment. This is encapsulated in IOcelotLogger and IOcelotLoggerFactory with the implementation for the standard ASP.NET Core logging stuff at the moment. This is because Ocelot adds some extra info to the logs such as *Request ID* if it is configured.

There is a global *Error Handling Middleware* that catches any exceptions thrown and logs them as errors. Finally, if logging is set to the `Trace` level, Ocelot will log the start, finish, and any middlewares that throw an exception, which can be quite useful.

20.1 Warning

If you are logging to the MS Console, you will experience terrible performance. The community has encountered many performance issues with Ocelot, and it is always related to the `Debug` logging level when logging to the console/terminal.

- **Warning**! Make sure you are logging to an appropriate destination/storage in the production environment!
- Use `Error` and `Critical` levels in the production environment!
- Use the `Warning` level in testing & staging environments!

These and other recommendations can be found below in the *Best Practices* section.

20.2 Best Practices

Microsoft Learn complete reference: Logging in .NET Core and ASP.NET Core

Our recommendations for achieving the best logging with Ocelot are as follows.

1. Ensure the minimum log level while Configure logging. The minimum log level is set in the application's `appsettings.json` file. This level is defined in the `Logging` section, for example:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Whether you are using Serilog or the standard Microsoft providers, the logging configuration will be retrieved from this section.

```
builder.Configuration
    .SetBasePath(builder.Environment.ContentRootPath)
    .AddJsonFile($"appsettings.{builder.Environment.EnvironmentName}.json",
↪ false, false) // read logging settings of the environment
    .AddOcelot(builder.Environment);
```

However, there is one thing to be aware of. It is possible to use the `SetMinimumLevel()` method to define the minimum logging level. Be careful and make sure you set the log level in only one place, like this:

```
builder.Logging
    .ClearProviders()
    .SetMinimumLevel(LogLevel.Warning);
// MS Console for Development and/or Testing environments only
if (!builder.Environment.IsProduction())
{
    builder.Logging.AddConsole();
}
```

Please also use the `ClearProviders()` method so that only the providers you wish to use are taken into account, such as the console in the example above.

2. Ensure the proper usage of the minimum logging level for each environment: development, testing, production, etc. So, once again, read the important notes in the *Warning* section!

3. Ocelot's logging has been improved in version 22.0: it is now possible to use a factory method for message strings that will only be executed if the minimum log level allows it.

   For example, let's take a message containing information about several variables that should only be generated if the minimum log level is `Debug`. If the minimum log level is `Warning`, then the string is never generated.

   Therefore, when the string contains dynamic information (e.g., `string.Format`), or the string value is generated by a string interpolation expression, it is recommended to call the `LogX` method using an anonymous delegate via an => expression function:

```
Logger.LogDebug(
    () => $"Downstream template is {httpContext.Items.DownstreamRoute().
↪DownstreamPathTemplate.Value}");
```

   otherwise a constant string is sufficient

```
Logger.LogDebug("My const string");
```

20.3 Request ID

Also known as "Correlation ID" or HttpContext.TraceIdentifier

Ocelot allows a client to send a *Request ID* through an HTTP header. If provided, Ocelot uses the *Request ID* for logging as soon as it becomes available in the middleware pipeline. Additionally, Ocelot forwards the *Request ID* via the specified header to the downstream service.

- You can still obtain the ASP.NET Core *Request ID* in the logs if you set `IncludeScopes` to `true` in your logging configuration.
- The reason for not just using the bog standard framework logging is that we could not work out how to override the `RequestId` that gets logged when setting `IncludeScopes` to `true` in the logging settings. Nicely onto the next feature.

Every log record has these 2 properties:

Property	*Description*
`RequestId`	This represents ID of the current request as plain string, for example `0HMVD33IIJRFR:00000001`
`PreviousRequestId`	This represents ID of the previous request

> As an `IOcelotLogger` interface object is injected into the constructors of service classes, the current default Ocelot logger (`OcelotLogger` class) reads these two properties from the `IRequestScopedDataRepository` interface object.

Find out more about these properties and other details on the *Request ID* logging feature below.

20.3.1 Configuration

In order to use the *Request ID* feature, you have two options: specifying it globally or for the route.

In your ocelot.json, set the following configuration in the `GlobalConfiguration` section. This setting will apply to all requests processed by Ocelot.

```
"GlobalConfiguration": {
  "RequestIdKey": "Oc-RequestId"
}
```

> We recommend using the `GlobalConfiguration` unless it is absolutely necessary to make it route-specific.

If you want to override this for a specific route, add the following to ocelot.json:

```
"RequestIdKey": "Oc-RequestId"
```

Once Ocelot identifies incoming requests that match a route, it will set the *Request ID* based on the route configuration.

20.3.2 Problem

This can lead to a small issue. If you set a `GlobalConfiguration`, it is possible to use one *Request ID* until the route is identified and then another afterward, as the *Request ID* key can change. This behavior is intentional and represents the best solution we have devised for now. In this case, the `OcelotLogger` will display both the current *Request ID* and the previous *Request ID* in the logs.

Below is an example of the logging when the `Debug` level is set for a normal request:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:7778/ocelot2/
↪posts/3 - - -
```

(continues on next page)

(continued from previous page)

```
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      Ocelot pipeline started
dbug: Ocelot.DownstreamRouteFinder.Middleware.
↪DownstreamRouteFinderMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      Upstream URL path: /ocelot2/posts/3
dbug: Ocelot.DownstreamRouteFinder.Middleware.
↪DownstreamRouteFinderMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      Downstream templates: /ocelot/posts/{id}
info: Ocelot.RateLimiting.Middleware.RateLimitingMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      EnableEndpointEndpointRateLimiting is not enabled for downstream␣
↪path: /ocelot/posts/{id}
info: Ocelot.Authentication.Middleware.AuthenticationMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      No authentication needed for path: /ocelot2/posts/3
info: Ocelot.Authorization.Middleware.AuthorizationMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      No authorization needed for upstream path: /ocelot2/posts/{id}
dbug: Ocelot.DownstreamUrlCreator.Middleware.
↪DownstreamUrlCreatorMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      Downstream URL: http://localhost:5555/ocelot/posts/3
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:7778/ocelot2/
↪posts/5 - - -
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      Ocelot pipeline started
dbug: Ocelot.DownstreamRouteFinder.Middleware.
↪DownstreamRouteFinderMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      Upstream URL path: /ocelot2/posts/5
dbug: Ocelot.DownstreamRouteFinder.Middleware.
↪DownstreamRouteFinderMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      Downstream templates: /ocelot/posts/{id}
info: Ocelot.RateLimiting.Middleware.RateLimitingMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      EnableEndpointEndpointRateLimiting is not enabled for downstream␣
↪path: /ocelot/posts/{id}
info: Ocelot.Authentication.Middleware.AuthenticationMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      No authentication needed for path: /ocelot2/posts/5
info: Ocelot.Authorization.Middleware.AuthorizationMiddleware[0]
```

(continues on next page)

(continued from previous page)

```
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      No authorization needed for upstream path: /ocelot2/posts/{id}
dbug: Ocelot.DownstreamUrlCreator.Middleware.
↪DownstreamUrlCreatorMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      Downstream URL: http://localhost:5555/ocelot/posts/5
info: Ocelot.Requester.Middleware.HttpRequesterMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      200 OK status code of request URI: http://localhost:5555/ocelot/
↪posts/3
dbug: Ocelot.Requester.Middleware.HttpRequesterMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      Setting HTTP response message...
dbug: Ocelot.Responder.Middleware.ResponderMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      No pipeline errors: setting and returning completed response...
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      RequestId: 0HNBA3NEIQUNJ:11111111, PreviousRequestId: -
      Ocelot pipeline finished
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET https://localhost:7778/ocelot2/
↪posts/3 - 200 84 application/json;+charset=utf-8 404.7256ms
info: Microsoft.AspNetCore.Hosting.Diagnostics[16]
      Request reached the end of the middleware pipeline without being␣
↪handled by application code. Request path: GET https://
↪localhost:7778/ocelot2/posts/3, Response status code: 200
info: Ocelot.Requester.Middleware.HttpRequesterMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      200 OK status code of request URI: http://localhost:5555/ocelot/
↪posts/5
dbug: Ocelot.Requester.Middleware.HttpRequesterMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      Setting HTTP response message...
dbug: Ocelot.Responder.Middleware.ResponderMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      No pipeline errors: setting and returning completed response...
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      RequestId: 0HNBA3NEIQUNK:AAAAAAAA, PreviousRequestId:␣
↪0HNBA3NEIQUNJ:11111111
      Ocelot pipeline finished
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET https://localhost:7778/ocelot2/
↪posts/5 - 200 128 application/json;+charset=utf-8 347.2607ms
info: Microsoft.AspNetCore.Hosting.Diagnostics[16]
      Request reached the end of the middleware pipeline without being␣
↪handled by application code. Request path: GET https://
↪localhost:7778/ocelot2/posts/5, Response status code: 200
```

20.3.3 Technical Facts

- Every log record has these 2 properties:
  - `RequestId` represents ID of the current request as plain string, for example `0HNBA3NEIQUNJ:00000001`.
  - `PreviousRequestId` represents ID of the previous request.
- As an `IOcelotLogger` interface object is injected into the constructors of service classes, the current default Ocelot logger (the `OcelotLogger` class) retrieves these two properties from the `IRequestScopedDataRepository` service.

20.4 Performance[1]

Here is a quick recipe for your production environment to achieve top *performance*. You need to ensure the minimum log level is `Critical` or `None`. Nothing more! Having top logging *performance* means having fewer log records written by the logging provider. So, the logs should be pretty empty.

Anyway, during the initial period after a version release to production, we recommend monitoring the system and the current version's app behavior by specifying the minimum log level as `Error`. If the release engineer ensures the stability of the version in production, then the minimum log level can be increased to `Critical` or `None` to achieve top *performance*. Technically, this will disable the logging feature entirely.

20.5 Benchmarks

We currently have two types of benchmarks:

- `SerilogBenchmarks` with Serilog logging to a file. See the `ConfigureLogging` method with `logging.AddSerilog(_logger)`.
- `MsLoggerBenchmarks` with MS default logging to the MS Console. See the `ConfigureLogging` method with `logging.AddConsole()`.

Benchmark results largely depend on the environment and hardware on which they run. We are pleased to invite you to run logging benchmarks on your machine by following the instructions below.

1. Open PowerShell or Command Prompt console
2. Build the Ocelot solution in Release mode: `dotnet build --configuration Release`
3. Go to the `test\Ocelot.Benchmarks\bin\Release\` folder
4. Choose the .NET version by changing the folder, for example, to `net9.0`
5. Run benchmarks: `.\Ocelot.Benchmarks.exe`
6. Run `SerilogBenchmarks` or `MsLoggerBenchmarks` by pressing the appropriate number of a benchmark (5 or 6), then press Enter.
7. Wait for 3+ minutes to complete the benchmark and get the final results.
8. Read and analyze your benchmark session results.

[1] Logging *performance* was improved in pull request 1745 and released in version 22.0. These changes were requested as part of issue 1744 following the team's discussion in thread 1736.

CHAPTER

TWENTYONE

METADATA

[1] Feature of: *Configuration*

Ocelot provides various features such as routing, authentication, caching, load balancing, and more. However, some users may encounter situations where Ocelot does not meet their specific needs or they want to customize its behavior. In such cases, Ocelot allows users to add *metadata* to the route configuration. This property can store any arbitrary data that users can access in middlewares or delegating handlers.

21.1 Schema

As you may already know from the *Configuration* chapter and the *Extend with Metadata* section, the route *metadata* schema is quite simple which is JSON dictionary:

```
"Metadata": {
  // "key": "value",
}
```

However, **global** metadata configuration consists of both the `Metadata` and `MetadataOptions` sections. You do not need to set all of these things, but this is everything that is available at the moment.

```
"GlobalConfiguration": {
  "Metadata": {
    // "key": "value",
  },
  "MetadataOptions": {
    "CurrentCulture": "en-GB",
    "NumberStyle": "Any",
    "Separators": [","],
    "StringSplitOption": "None",
    "TrimChars": [" "],
  }
}
```

The actual global *metadata* schema with all the properties can be found in the C# FileMetadataOptions class. This configuration type is parsed to a MetadataOptions type object.

[1] The *Metadata* feature was requested in issues 738 and 1990, and it was released as part of version 23.3.

Option	*Description*
`CurrentCulture`	Parsed as the `System.Globalization.CultureInfo` object (refer to CultureInfo class) Default value is current culture aka `CultureInfo.CurrentCulture.Name`
`NumberStyle`	Parsed as the `System.Globalization.NumberStyles` object (refer to NumberStyles enum) Default value is `NumberStyles.Any`
`Separators`	Array of `string`. Default value is `[","]` aka comma.
`StringSplitOpti`	Parsed as the `System.StringSplitOptions` object (refer to StringSplitOptions enum) Default value is `StringSplitOptions.None`
`TrimChars`	Array of `char`. Default value is `[" "]` aka whitespace.
`Metadata`	Parsed as the `Dictionary<string, string>` object containing all global *metadata* which `string` values are parsed to a target type value by the *GetMetadata<T> Method*.

21.2 Configuration

By using the *metadata*, users can implement their own logic and extend the functionality of Ocelot e.g.

```
{
  "Routes": [
    {
      "UpstreamHttpMethod": [ "GET" ],
      "UpstreamPathTemplate": "/posts/{postId}",
      "DownstreamPathTemplate": "/api/posts/{postId}",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 80 }
      ],
      "Metadata": {
        "id": "FindPost",
        "tags": "tag1, tag2, area1, area2, func1",
        "plugin1.enabled": "true",
        "plugin1.values": "[1, 2, 3, 4, 5]",
        "plugin1.param": "value2",
        "plugin1.param2": "123",
        "plugin2/param1": "overwritten-value",
        "plugin2/data": "{\"name\":\"John Doe\",\"age\":30,\"city\":\"New York\
↪",\"is_student\":false,\"hobbies\":[\"reading\",\"hiking\",\"cooking\"]}"
      }
    }
  ],
  "GlobalConfiguration": {
    "Metadata": {
      "instance_name": "machine-1",
      "plugin2/param1": "default-value"
```

(continues on next page)

(continued from previous page)

```
    },
    "MetadataOptions": {
    }
  }
}
```

Now, the route *metadata* can be accessed through the `DownstreamRoute` object:

```
using Ocelot.Middleware;
using Ocelot.Metadata;
using Ocelot.Logging;

public class MyMiddleware : OcelotMiddleware
{
    private readonly RequestDelegate _next;
    private readonly IMyService _myService;

    public MyMiddleware(RequestDelegate next, IOcelotLoggerFactory␣
↪loggerFactory, IMyService myService)
        : base(loggerFactory.CreateLogger<MyMiddleware>())
    {
        _next = next;
        _myService = myService;
    }

    public Task Invoke(HttpContext context)
    {
        Logger.LogDebug("My middleware started");
        var route = context.Items.DownstreamRoute();
        var id = route.GetMetadata<string>("id");
        var tags = route.GetMetadata<string[]>("tags");

        // Plugin 1 data
        var p1Enabled = route.GetMetadata<bool>("plugin1.enabled");
        var p1Values = route.GetMetadata<string[]>("plugin1.values");
        var p1Param = route.GetMetadata<string>("plugin1.param", "system-
↪default-value");
        var p1Param2 = route.GetMetadata<int>("plugin1.param2");

        // Plugin 2 data
        var p2Param1 = route.GetMetadata<string>("plugin2/param1", "default-
↪value");
        var json = route.GetMetadata<string>("plugin2/data");
        var plugin2 = System.Text.Json.JsonSerializer.Deserialize<Plugin2Data>
↪(json);

        // Reading global metadata
        var globalInstanceName = route.GetMetadata<string>("instance_name");
        var globalPlugin2Param1 = route.GetMetadata<string>("plugin2/param1");

        // Working with plugin's metadata
        // ...
```

(continues on next page)

(continued from previous page)

```
        return _next.Invoke(context);
    }
    public class Plugin2Data
    {
        public string name { get; set; }
        public int age { get; set; }
        public string city { get; set; }
        public bool is_student { get; set; }
        public string[] hobbies { get; set; }
    }
}
```

21.3 GetMetadata<T> Method

Ocelot provides one DowstreamRoute extension method to help you retrieve your *metadata* values effortlessly. With the exception of the types string, bool, bool?, string[] and numeric, all strings passed as parameters are treated as json strings and an attempt is made to convert them into objects of generic type T. If the value is null, then, if not explicitely specified, the default for the chosen target type is returned.

Method	*Description*
GetMetadata<str	The *metadata* value is returned as string without further parsing
GetMetadata<str	The *metadata* value is splitted by a given separator (default ,) and returned as a string array. **Note**: Several parameters can be set in the global configuration, such as Separators (default = [","]), StringSplitOptions (default None) and TrimChars, the characters that should be trimmed (default = [' ']).
GetMetadata<TIn	The *metadata* value is parsed to a number. The TInt is any known numeric type, such as byte, sbyte, short, ushort, int, uint, long, ulong, float, double, decimal. **Note**: Some parameters can be set in the global configuration, such as NumberStyle (default Any) and CurrentCulture (default CultureInfo.CurrentCulture)
GetMetadata<T>	The *metadata* value is converted to the given generic type. The value is treated as a json string and the json serializer tries to deserialize the string to the target type. **Note**: A JsonSerializerOptions object can be passed as method parameter, Web is used as default.
GetMetadata<boo	Check if the *metadata* value is a truthy value, otherwise return false. **Note**: The truthy values are: true, yes, ok, on, enable, enabled
GetMetadata<boo >	Check if the *metadata* value is a truthy value (return true), or falsy value (return false), otherwise return null. **Note**: The known truthy values are: true, yes, ok, on, enable, enabled, 1, the known falsy values are: false, no, off, disable, disabled, 0

21.4 Sample

The *Metadata* feature is a relatively new *Configuration* feature (anchored in the "*Extend with Metadata*" section).

To introduce a standardized approach to middleware development, we have prepared a comprehensive sample project:

Project: samples / Metadata
Solution: Ocelot.Samples.sln

The solution for the `Ocelot.Samples.Metadata.csproj` project includes the following capabilities:

- It has two custom Ocelot middlewares attached: `PreErrorResponderMiddleware` and `ResponderMiddleware`. The `PreErrorResponderMiddleware` reads the route *metadata* based on the route ID and parses it. This is an example of how to parse or read the *metadata* of a specific route.
- The custom `ResponderMiddleware` simply calls the base Ocelot middleware (default implementation). Ocelot's `ResponderMiddleware` is responsible for writing the final body data into the `HttpResponse` of the current `HttpContext`.
- The main Program replaces Ocelot's default `IHttpResponder` service with a custom `MetadataResponder` service. It attaches both `PreErrorResponderMiddleware` and `ResponderMiddleware` using the `OcelotPipelineConfiguration` argument in the `UseOcelot` method.
- The `MetadataResponder` service processes all JSON data when the `Content-Type` header has the value `application/json`. This custom responder service writes the original data into the `Response` section and writes the route *metadata* back to the `Metadata` section using the following JSON schema:

```
{
  "Response": {
    // Original data of the downstream response
  },
  "Metadata": {
    // current route metadata
  }
}
```

- The `MetadataResponder` service always generates the custom `OC-Route-Metadata` header, containing the route *metadata* as a plain JSON string for all routes, regardless of the media type of the content. This allows you to parse it on the client side for specific purposes.
- The `MetadataResponder` service attempts to decompress the content body if it is compressed using one of the following algorithms from downstream endpoints: Brotli (`br`), GZip (`gzip`), or Zstandard (`zstd`). However, data compressed with the `deflate` algorithm is ignored and transferred to the client as-is because decompressing a third-party algorithm with a custom implementation is not feasible. Finally, the responder service returns uncompressed data and indicates this in the `Content-Encoding` header, where the value is always set to `identity`.
- Processing JSON data can be disabled for specific routes using the `disableMetadataJson` option in the *metadata*. In this case, all JSON data is returned to the client as-is, preserving the original body streams (see the `/ocelot/docs/` route).

Conclusion: The purpose of this sample is to detect JSON data, process it, and embed a custom `Metadata` section while returning the original JSON data in the `Response` section. This sample and its

`MetadataResponder` service significantly increase response time due to on-the-fly JSON data processing, leading to degraded overall performance. Please consider this as an example of processing *metadata*. For production environments, such processing should be disabled. Instead, returning *metadata* in a custom header is likely the best solution if your client needs to know the currently executed route on Ocelot's side.

CHAPTER

TWENTYTWO

METHOD TRANSFORMATION[1]

Ocelot allows users to modify the HTTP request method used when making requests to a downstream service. This is achieved by setting the following route configuration:

```
{
  "UpstreamPathTemplate": "/{everything}",
  "DownstreamPathTemplate": "/{everything}",
  // other props and opts...
  "UpstreamHttpMethod": [ "Get" ], // we transform HTTP verb...
  "DownstreamHttpMethod": "Post" // ...from GET to POST
}
```

The key property here is `DownstreamHttpMethod`, which is set to `POST`, and the route will only match `GET`, as specified by `UpstreamHttpMethod`.

This feature is useful when interacting with downstream APIs that only support `POST` while presenting a RESTful interface.

---

[1] The *"Method Transformation"* feature was released in version 14.0.8.

CHAPTER TWENTYTHREE

MIDDLEWARE INJECTION

When setting up Ocelot in your Program, you can provide additional middleware and override it with your custom middlewares. This is done as follows:

```
// Set it up: configuration, services, etc.
// Middleware setup is only possible during the final stage of app␣
↪configuration and execution
var app = builder.Build();
var pipeline = new OcelotPipelineConfiguration
{
    PreErrorResponderMiddleware = async (context, next) =>
    {
        await next.Invoke();
    }
};
await app.UseOcelot(pipeline);
await app.RunAsync();
```

In the example above, the provided function will run before the first piece of Ocelot middleware. This allows users to supply any behavior they want before and after the Ocelot pipeline has run.

> **Warning**
>
> Be cautious, as this means you can break everything — use at your own risk or pleasure! If you notice any exceptions or strange behavior in your middleware pipeline and are using any of the following, remove your custom middlewares and try again.

23.1 `OcelotPipelineConfiguration` Class

Class: OcelotPipelineConfiguration

The user can set middleware-functions aka custom user's middleware against the following:

Middleware	*Description*
`PreErrorResponderMiddleware` Prev: `ExceptionHandlerMiddleware` Next: `ResponderMiddleware`	This is called after the global error-handling middleware, so any code before calling `next.Invoke` is the next action executed in the Ocelot pipeline. Any code after `next.Invoke` is the final action executed in the Ocelot pipeline before reaching the global error handler.
`ResponderMiddleware` Prev: `PreErrorResponderMiddleware` Next: `DownstreamRouteFinderMiddleware`	This allows the user to completely override Ocelot's ResponderMiddleware. [1]
`PreAuthenticationMiddleware` Prev: `RequestIdMiddleware` Next: `AuthenticationMiddleware`	This allows the user to run any extra authentication before the Ocelot authentication kicks in.
`AuthenticationMiddleware` Prev: `PreAuthenticationMiddleware` Next: `ClaimsToClaimsMiddleware`	This allows the user to completely override Ocelot's AuthenticationMiddleware. [1]
`PreAuthorizationMiddleware` Prev: `ClaimsToClaimsMiddleware` Next: `AuthorizationMiddleware`	This allows the user to run any extra authorization before the Ocelot authorization kicks in.
`AuthorizationMiddleware` Prev: `PreAuthorizationMiddleware` Next: `ClaimsToHeadersMiddleware`	This allows the user to completely override Ocelot's AuthorizationMiddleware. [1]
`ClaimsToHeadersMiddleware` Prev: `AuthorizationMiddleware` Next: `PreQueryStringBuilderMiddleware`	This allows the user to completely override Ocelot's ClaimsToHeadersMiddleware. [1]
`PreQueryStringBuilderMiddleware` Prev: `ClaimsToHeadersMiddleware` Next: `ClaimsToQueryStringMiddleware`	This allows the user to implement own query string manipulation logic.
`WebSocketsMiddleware` [2]	This allows the user to completely override Ocelot's WebSocketsProxyMiddleware for *WebSockets* requests. [1] This delegate is ignored when `WebSocketsMiddlewareType` is also set.
`WebSocketsMiddlewareType` [2]	This allows the user to specify a `Type` derived from WebSocketsProxyMiddleware to use as the custom *WebSockets* proxy middleware. [1] This option takes **priority** over `WebSocketsMiddleware` when both properties are set. Useful for customizing behaviors such as the buffer size; see the *Sample* section in the *Websockets* chapter.

Obviously, you can add the mentioned Ocelot middleware overrides as normal before the call to `app.UseOcelot`. They cannot be added afterward because Ocelot does not invoke subsequent middleware

overrides based on the specified middleware configuration. As a result, the next-called middleware **will not** affect the Ocelot configuration.

Warning

[1] Use the mentioned middleware overrides with caution! Overridden middleware removes the default implementation. If you encounter any exceptions or strange behavior in your middleware pipeline, remove the overridden middleware and try again.

Note

[2] Overriding the WebSocketsProxyMiddleware is available starting from Ocelot version 25.0.

23.2 Ocelot Pipeline Builder

Class: `Ocelot.Middleware.OcelotPipelineExtensions`
Method: `BuildOcelotPipeline(IApplicationBuilder, OcelotPipelineConfiguration)`

The Ocelot pipeline is part of the entire ASP.NET Core Middleware conveyor, also known as the app pipeline. The BuildOcelotPipeline method encapsulates the Ocelot pipeline. The last middleware in the `BuildOcelotPipeline` method is `HttpRequesterMiddleware`, which calls the next middleware if it is added to the pipeline.

The internal HttpRequesterMiddleware is part of the pipeline, but it is private and cannot be overridden since this middleware is not included in the list of user-accessible public middlewares that can be overridden. Therefore, it is the final middleware in both the Ocelot and ASP.NET pipelines, and it handles non-user operations. The last user (public) middleware that can be overridden is PreQueryStringBuilderMiddleware, which is read from the pipeline configuration object. For more details, see the previous *OcelotPipelineConfiguration Class* section.

To understand the actual order of middleware execution, here is a quick list of them, with an asterisk (*) marking the ones that can be overridden:

1. `ConfigurationMiddleware`
2. `ExceptionHandlerMiddleware`
3. `PreErrorResponderMiddleware`*
4. `ResponderMiddleware`*
5. `DownstreamRouteFinderMiddleware`
6. `MultiplexingMiddleware`
7. `SecurityMiddleware`
8. `HttpHeadersTransformationMiddleware`
9. `DownstreamRequestInitialiserMiddleware`
10. `RateLimitingMiddleware`
11. `RequestIdMiddleware`
12. `PreAuthenticationMiddleware`*

13. `AuthenticationMiddleware*`
14. `ClaimsToClaimsMiddleware`
15. `PreAuthorizationMiddleware*`
16. `AuthorizationMiddleware*`
17. `ClaimsToHeadersMiddleware*`
18. `PreQueryStringBuilderMiddleware*`
19. `ClaimsToQueryStringMiddleware`
20. `ClaimsToDownstreamPathMiddleware`
21. `LoadBalancingMiddleware`
22. `DownstreamUrlCreatorMiddleware`
23. `OutputCacheMiddleware`
24. `HttpRequesterMiddleware`

Considering that `PreQueryStringBuilderMiddleware` and `HttpRequesterMiddleware` are the final user and system middleware, there are no other middleware components in the pipeline. However, you can still extend the ASP.NET pipeline, as demonstrated in the following code:

```
await app.UseOcelot();
app.UseMiddleware<MyCustomMiddleware>();
```

However, we do not recommend adding custom middleware before or after calling `UseOcelot()` because it affects the stability of the entire pipeline and has not been tested. This type of custom pipeline building falls outside the Ocelot pipeline model, and the quality of the solution is your responsibility.

Finally, do not confuse the distinction between system (private, non-overridden) and user (public, overridden) middleware. Private middleware is hidden and cannot be overridden, but the entire ASP.NET pipeline can still be extended. The public middleware of the *OcelotPipelineConfiguration Class* is fully customizable and can be overridden.

23.3 Roadmap

The community has shown interest in adding more overridden middleware. One such request is pull request 1497, which may possibly be included in an upcoming release.

In any case, if the current overridden middleware does not provide enough pipeline flexibility, you can open a new topic in the Discussions of the repository.

CHAPTER

TWENTYFOUR

QUALITY OF SERVICE

Label: QoS
Repository: Ocelot.QualityOfService.Polly

Ocelot supports *Quality of Service* (QoS) features that allow you to protect downstream services from overload and control request flow on a per-route basis. Two implementations are available and are **mutually exclusive** — exactly one may be active at a time:

- *Built-in QoS* — included in the Ocelot core package; no additional dependencies required.
- *Installation (Polly)* via Polly — a full-featured resilience pipeline powered by the well-regarded Polly .NET library (repository).

The last registration wins: calling `AddQualityOfService()` after `AddPolly()` replaces the Polly handler, and vice versa.

Note

Polly v7 syntax is no longer supported as of version 23.2, when the Ocelot team upgraded Polly from v7 to v8.

24.1 Implementations Overview

The table below summarises the key differences between the two QoS implementations.

Capability	*Built-in*	*Polly*
Extra NuGet package	None (Ocelot core)	`Ocelot.QualityOfService.Polly`
Activation	`.AddQualityOfService()`	`.AddPolly()`
Circuit Breaker	✓ Custom (count mode & ratio mode)	✓ Polly `CircuitBreakerResilienceStrategy`
Per-request Timeout	✓ `CancellationToken`-based	✓ Polly `TimeoutResilienceStrategy`
Full Polly resilience pipeline		✓
Extensibility API		✓ (custom providers, handlers, error maps)
Invalid-value handling	Silent default substitution	Logged warning + default substitution
`MinimumThroughput = 0` *	Disables circuit breaking, but not timing out	Disables circuit breaking, but not timeout strategy
`Timeout = 0` *	Disables timing out, but not circuit breaking	Disables timing out, but not circuit breaker strategy

Note

* Use with caution, since other `QoSOptions` values will be substituted at runtime if at least one other strategy option is defined while the others are null.

24.2 Built-in QoS

Class: CircuitBreakerDelegatingHandler

Ocelot's built-in *Quality of Service* implementation is part of the core `Ocelot` package. It wraps every outgoing downstream request in a CircuitBreakerDelegatingHandler, providing circuit-breaker protection and an optional per-request timeout with no external dependencies.

To activate it, call `AddQualityOfService()` on the `OcelotBuilder`[1]:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddQualityOfService();
```

The circuit breaker state is maintained **per route** — each route has its own independent circuit breaker instance. There are two operating modes, selected automatically based on the options you configure.

24.2.1 Count mode (default)

Activated when `MinimumThroughput` is set **without** `FailureRatio` and `SamplingDuration`. The circuit opens after `MinimumThroughput` **consecutive failures**.

```
"QoSOptions": {
  "MinimumThroughput": 3,
```

(continues on next page)

[1] The *AddOcelot method* adds default ASP.NET services to the DI container. You can call another extended *AddOcelotUsingBuilder method* while configuring services to develop your own *Custom Builder*. See more instructions in the "*AddOcelotUsingBuilder method*" section of the *Dependency Injection* feature.

(continued from previous page)

```
  "BreakDuration": 1000
}
```

With this configuration, the circuit opens after 3 consecutive failures and remains open for 1 second.

24.2.2 Ratio mode

Activated when `FailureRatio` **and** `SamplingDuration` are set alongside `MinimumThroughput`. The circuit opens when the ratio of failed requests within a rolling `SamplingDuration` window equals or exceeds `FailureRatio`, **provided** at least `MinimumThroughput` requests have been made in that window.

```
"QoSOptions": {
  "MinimumThroughput": 10,
  "FailureRatio": 0.5,
  "SamplingDuration": 10000,
  "BreakDuration": 5000
}
```

With this configuration, once 10 or more requests have been recorded in a 10-second rolling window, the circuit opens if 50 % or more of them are failures. The circuit then stays open for 5 seconds before transitioning to `HalfOpen`.

24.2.3 Circuit Breaker state machine

The built-in circuit breaker implements the standard three-state machine:

State	*Behaviour*
`Closed`	Normal operation: requests pass through and failures are counted.
`Open`	Circuit is open: requests are **immediately** rejected with `503 Service Unavailable` — no downstream call is made. After `BreakDuration` has elapsed, the circuit transitions to `HalfOpen`.
`HalfOpen`	Exactly **one** probe request is allowed through. If it succeeds, the circuit closes. If it fails, the circuit reopens and the `BreakDuration` timer restarts. All other concurrent requests while the probe is in flight are rejected with `503`.

24.2.4 Timeout

An optional per-request timeout can be configured independently of or alongside the circuit breaker:

```
"QoSOptions": {
  "Timeout": 5000
}
```

When a request exceeds `Timeout` milliseconds, it is cancelled. A `503 Service Unavailable` response is returned, and the event is recorded as a circuit-breaker failure.

Setting `Timeout` to `0` or a negative value disables the timeout. To disable the per-request timeout entirely, omit the `Timeout` option or set it to `0`.

Note

When `Timeout` is the only option configured, the built-in circuit breaker is still active with its default values: `MinimumThroughput = 100` and `BreakDuration = 5000 ms`. The circuit opens after 100 consecutive timeout failures and stays open for 5 seconds. To control these defaults, configure `MinimumThroughput` and `BreakDuration` explicitly.

24.2.5 Server Error Codes

The following HTTP response status codes are treated as failures by the built-in handler:

Code	*Status*
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected

Any other status code (including `4xx` client errors) is recorded as a success and does not contribute to the failure count. Unhandled exceptions (excluding `OperationCanceledException`) are also counted as failures.

Overriding server error codes

The set of failure codes is exposed as the `protected virtual` property `ServerErrorCodes` on `CircuitBreakerDelegatingHandler`. You can extend or replace this set by creating a subclass and overriding the property:

```
public class MyCircuitBreakerHandler : CircuitBreakerDelegatingHandler
{
    public MyCircuitBreakerHandler(DownstreamRoute route, IOcelotLoggerFactory loggerFactory)
        : base(route, loggerFactory) { }

    // Treat all 5xx codes AND 429 Too Many Requests as failures
    protected override HashSet<HttpStatusCode> ServerErrorCodes { get; } =
        new(DefaultServerErrorCodes) { HttpStatusCode.TooManyRequests };
}
```

Then register it with the `AddQualityOfService<THandler>()` overload on `OcelotBuilder`:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddQualityOfService<MyCircuitBreakerHandler>();
```

24.2.6 Built-in Value Constraints

The built-in handler silently substitutes a default when an option is unset or outside its valid range — no warning is logged.

Option	*Valid range*	*Default*	*Notes*
`BreakDuration`	> 500 ms	5000 ms	Duration the circuit stays open before transitioning to `HalfOpen`.
`MinimumThroughput`	2	100	Set to `0` or negative to **disable** circuit-breaking entirely.
`FailureRatio`	(0.0, 1.0]	0.5	Ratio mode only.
`SamplingDuration`	> 500 ms	10 000 ms	Ratio mode only.
`Timeout`	> 10 ms, < 86 400 000 ms	30 000 ms	Set to `0` or negative to **disable** timing out. Invalid positive values outside the range use the default (30 s).

24.3 Installation (Polly)

To utilise *Quality of Service* via the Polly library, begin by importing the appropriate Ocelot.QualityOfService.Polly extension package:

```
Install-Package Ocelot.QualityOfService.Polly
```

Next, in your Program, incorporate Polly services by invoking the `AddPolly()` extension on the `OcelotBuilder`, as shown below[Page 120, 1]:

```
using Ocelot.QualityOfService.Polly;

builder.Services
    .AddOcelot(builder.Configuration)
    .AddPolly();
```

Note

Prior to version 25.0, the package was named Ocelot.Provider.Polly. If you are using version 24.1 or earlier, install the Ocelot.Provider.Polly package. For version 25.0 and later, the package ID is Ocelot.QualityOfService.Polly.

24.4 `QoSOptions` Schema

Class: FileQoSOptions

Here is the complete *Quality of Service* configuration, also known as the "QoS options schema". This schema is shared by both the *Built-in QoS* and the *Installation (Polly)* implementations. Depending on your needs and chosen strategies, definition of all properties is not required. If you skip a property, a default value will be substituted — see *Built-in Value Constraints* for the built-in implementation and *Value constraints (Polly)* for Polly.

```
"QoSOptions": {
  // Circuit Breaker strategy
  "BreakDuration": 0, // integer
  "MinimumThroughput": 0, // integer
  "FailureRatio": 0.0, // floating number
  "SamplingDuration": 0, // integer
  // Timeout strategy
  "Timeout": 0, // integer
  // Deprecated options
  "DurationOfBreak": 0, // deprecated! -> use BreakDuration
  "ExceptionsAllowedBeforeBreaking": 0, // deprecated! -> use MinimumThroughput
  "TimeoutValue": 0, // deprecated! -> use Timeout
}
```

Ocelot Option and Polly equivalent	*Description*
`BreakDuration` (formerly `DurationOfBreak`) as BreakDuration	This is duration of break the circuit will stay open before resetting. The unit is milliseconds.
`MinimumThroughput` (formerly `ExceptionsAllowedBefore` as MinimumThroughput, a primary option	This number of actions or more must pass through the circuit within the time slice for the statistics to be considered significant and for the circuit breaker to engage
`FailureRatio` is FailureRatio	This is the failure-to-success ratio at which the circuit will break
`SamplingDuration` is SamplingDuration	This is the duration of the sampling over which failure ratios are assessed. The unit is milliseconds.
`Timeout` (formerly `TimeoutValue`) as Timeout, a primary option	This is the default timeout. The unit is milliseconds.

Warning

The following options are deprecated in version 24.1: `DurationOfBreak`, `ExceptionsAllowedBeforeBreaking`, and `TimeoutValue`! Use the appropriate new options as shown in the table above. These deprecated options will be removed in version 25.0. For backward compatibility in version 24.1, a deprecated option takes precedence over its replacement.

Note[2]: Ocelot checks that the values of options are valid during execution. If not, it logs errors or warnings (refer to the *Value constraints (Polly)* section in *Notes* for Polly, or *Built-in Value Constraints* for the built-in implementation). For a complete explanation about strategies and mechanisms, consult Polly's Resilience strategies documentation.

24.5 Global Configuration[3]

According to the *Global Configuration Schema*, global *Quality of Service* options for static routes were introduced in version 24.1. These global options can also be overridden in the `Routes` configuration section, a capability that has been supported for a long time.

```
{
  "Routes": [
    {
      "Key": "R0", // optional
      "QoSOptions": {
        "Timeout": 15000 // 15s
      },
      // ...
    },
    {
      "Key": "R1", // this route is part of a group
      "QoSOptions": {}, // optional due to grouping
      // ...
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://ocelot.net",
    "QoSOptions": {
      "RouteKeys": ["R1",], // if undefined or empty array, opts will apply to all routes
      "BreakDuration": 1000, // 1s
      "MinimumThroughput": 3
    },
    // ...
  }
}
```

Dynamic routes were not supported in versions prior to 24.1. However, global *Quality of Service* options have been available in *Dynamic Routing* mode for a long time. Starting with version 24.1, global *QoS* options can also be overridden in the `DynamicRoutes` configuration section, as defined by the *Dynamic Route Schema*.

[2] If something doesn't work or you're stuck, consider reviewing the current QoS issues filtered by the label.

[3] The *Global Configuration* for dynamic routes was first introduced in pull request 351 and released in version 7.0.1. Since then, global configuration for static routes was added in pull requests 2081 and 2339, and delivered in version 24.1. Support for dynamic routes was also added in pull request 2339 and delivered in version 24.1.

```
{
  "DynamicRoutes": [
    {
      "Key": "", // optional
      "ServiceName": "my-service",
      "QoSOptions": {
        "Timeout": 15000 // 15s
      },
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://ocelot.net",
    "DownstreamScheme": "http",
    "ServiceDiscoveryProvider": {
      // required section for dynamic routing
    },
    "QoSOptions": {
      "RouteKeys": [], // or null, no grouping, thus opts apply to all dynamic␣
↪routes
      "BreakDuration": 1000, // 1s
      "MinimumThroughput": 3,
      "FailureRatio": 0.1, // 10%
      "SamplingDuration": 30000 // 30s
    }
  }
}
```

In this dynamic routing configuration, the *Timeout strategy (Polly)* is applied to the `my-service` service in addition to the *Circuit Breaker strategy (Polly)*, resulting in Polly timing out after 15 seconds. However, for all implicit dynamic routes, the *Timeout strategy (Polly)* is not globally configured, in favor of the standard *Timeout* option managed by the Ocelot Core requester middleware. Lastly, the *Circuit Breaker strategy (Polly)* has been globally configured for all routes due to the absence of route grouping, with the following options: allow 3 errors before breaking the circuit for 1 second, and allow up to 10% errors during the default 30-second sampling period.

Note

1. Please note that route-level options take precedence over global options.

2. If the `RouteKeys` option is not defined or the array is empty in the global `QoSOptions`, the global options will apply to all routes. If the array contains route keys, it defines a single group of routes to which the global options apply. Routes excluded from this group must specify their own route-level `QoSOptions`.

3. When using the **Polly** implementation: Ocelot's Polly provider utilizes the Resilience pipeline registry, so each route has a dedicated pipeline cached in Polly's registry using the route's load-balancing key. For a static route, the load-balancing key uniquely identifies the route by its upstream options, whereas for dynamic routes the load-balancing key is typically the service name from the discovery provider. Thus, Polly's registry maintains dedicated pipelines for each discovered service, and those pipelines behave independently. Finally, it is important to understand that global *QoS* options do not create a single shared resilience pipeline in the registry. When using the **built-in** implementation: each route also gets its own independent `CircuitBreakerDelegatingHandler` instance, so circuit state is always per-route.

4. Dynamic routes were not supported in versions prior to 24.1. Beginning with version 24.1, global *QoS* options for *Dynamic Routing* may be overridden in the `DynamicRoutes` configuration section, as defined by the *Dynamic Route Schema*. Additionally, global configuration for static routes (also known as `Routes`) has been supported since version 24.1.

24.6 Circuit Breaker strategy (Polly)

Implementation: Polly
Documentation: Circuit breaker resilience strategy
Primary option: `MinimumThroughput`, formerly `ExceptionsAllowedBeforeBreaking`

Note

This section describes the *Circuit Breaker* behaviour when using the **Polly** implementation. For the built-in implementation, see *Count mode (default)* and *Ratio mode*.

The options `MinimumThroughput` and `BreakDuration` can be configured independently from `Timeout`:

```
"QoSOptions": {
  "MinimumThroughput": 3,
  "BreakDuration": 1000 // ms
}
```

Alternatively, you can omit `BreakDuration`, which will default to the implicit 5-second setting as specified in Polly's BreakDuration documentation:

```
"QoSOptions": {
  "MinimumThroughput": 3
}
```

This setup activates only the Circuit breaker resilience strategy.

Additionally, there is a failure handling strategy based on `FailureRatio`, which serves as a counterpart to, or supplement for, the number of failures, also known as `MinimumThroughput`.

```
"QoSOptions": {
  "MinimumThroughput": 10,
  "FailureRatio": 0.5, // 50%
  "SamplingDuration": 10000, // ms, 10 seconds
}
```

Thus, a failure ratio of `0.5` indicates that the circuit will break if 50% or more of actions result in handled failures, after reaching the minimum threshold of 10 failures, also known as the `MinimumThroughput` option. Additionally, the 10-second sampling duration defines the time window over which the 50% failure ratio is evaluated.

Note: The `MinimumThroughput` option (also known as Polly's MinimumThroughput) is the primary option that enables the *Circuit Breaker strategy*. Its value must be valid (set to 2 or greater, refer to the *Value constraints (Polly)* section in *Notes*) and may be supplemented with additional Circuit Breaker options.

24.7 Timeout strategy (Polly)

Implementation: Polly
Documentation: Timeout resilience strategy
Primary option: `Timeout`, formerly `TimeoutValue`

> **Note**
>
> This section describes the *Timeout* behaviour when using the **Polly** implementation. For the built-in implementation, see *Timeout*.

The `Timeout` can be configured independently from the options of the *Circuit Breaker strategy (Polly)*:

```
"QoSOptions": {
  "Timeout": 5000 // ms
}
```

This setup activates only the Timeout resilience strategy.

To configure a global QoS timeout using the *Timeout strategy* for all routes (both static and dynamic) set the `Timeout` option as defined in the *Global Configuration Schema*:

```
"GlobalConfiguration": {
  // other global props
  "QoSOptions": {
    "Timeout": 10000 // ms, 10 seconds
  }
}
```

Please note that the route-level timeout takes precedence over the global timeout. For example, a route timeout may be shorter, while the global timeout can be longer and apply to all routes.

Note: There are *Value constraints (Polly)* for `Timeout`: it must be a positive number starting from *1 millisecond* to enable the *Timeout strategy*. If `Timeout` is undefined, zero or a negative number, the *Timeout strategy* will not be added to the resilience pipeline. Also, keep in mind Polly's Timeout constraint, thus Ocelot validates the `Timeout`. If the value violates Polly's requirements, it will be rolled back to the default of *30 seconds*.

24.8 Notes

24.8.1 Absolute timeout[4]

If a *QoS* section is not included, *QoS* will not be applied, and Ocelot will enforce an absolute timeout of 90 seconds (defined by the `DownstreamRoute` DefTimeout constant) for all downstream requests. This absolute timeout is configurable via the `DownstreamRoute` DefaultTimeoutSeconds static C# property. For more information, refer to the *Default timeout* section of the *Configuration* chapter.

[4] The *Absolute timeout* configuration, used as the *Default timeout*, and the *Timeout* feature were requested in issue 1314, implemented in pull request 2073, and officially released in version 24.1.

24.8.2 Value constraints (Polly)

Note

The constraints below apply to the **Polly** implementation. For the **built-in** implementation's constraints, see *Built-in Value Constraints*.

Starting with Polly v8, the Resilience strategies documentation outlines the following constraints on values:

- The `BreakDuration` value must exceed **500** milliseconds and be less than **24** hours (1 day = `86 400 000` milliseconds). If unspecified or invalid, it defaults to **5000** milliseconds (5 seconds); refer to the BreakDuration documentation.
- The `MinimumThroughput` value must be **2** or greater. If unspecified or invalid, it defaults to **100** failures; refer to the MinimumThroughput documentation.
- The `FailureRatio` must be greater than **0.0** and no more than **1.0**. If unspecified or invalid, it defaults to **0.1** (10%); refer to the FailureRatio documentation.
- The `SamplingDuration` value must exceed **500** milliseconds and be less than **24** hours (1 day = `86 400 000` milliseconds). If unspecified or invalid, it defaults to **30000** milliseconds (30 seconds); refer to the SamplingDuration documentation.
- The `Timeout` must be greater than **10** milliseconds and less than **24** hours (1 day = `86 400 000` milliseconds). If unspecified or invalid, it defaults to **30000** milliseconds (30 seconds); refer to the Timeout documentation. And please note, when both route-level and global *QoS* timeouts have positive values but are invalid, a default value will be automatically substituted from the `TimeoutStrategy` class DefaultTimeout static C# property, which can also be configured in your Program.

Ocelot logs warnings containing failed validation messages for all options, but it does not block Ocelot startup, even when *QoS* options are invalid. Inspect your logs for these messages and adjust your configuration if necessary.

24.8.3 QoS and route (global) timeouts

The `Timeout` option in *QoS* always takes precedence over the route *Timeout* property, so *Timeout* will be ignored in favor of QoS `Timeout`. In Ocelot Core, `Timeout` and configuration *Timeout* are not intended to be used together. Moreover, there is an Ocelot Core design constraint: if the route or global `Timeout` duration is shorter than the *QoS* `Timeout`, you may encounter warning messages in the logs that begin with the following sentence:

```
Route '/xxx' has Quality of Service settings (QoSOptions) enabled, but either␣
↪the route Timeout or the QoS Timeout is misconfigured: ...
```

This warning means that the route or global timeout will occur before the *QoS Timeout strategy (Polly)* has a chance to handle its own timeout event, which is configured with a longer duration. Technically, this situation results in the functional disabling of the Polly's Timeout resilience strategy. Ocelot handles this misconfiguration by logging a warning and automatically applying a longer timeout to the `TimeoutDelegatingHandler` in order to effectively unblock the *QoS Timeout strategy (Polly)*. To avoid this warning, ensure that your *QoS* timeouts are shorter than the route or global timeouts, or remove the *Timeout* property from routes where *QoS* is enabled with the `Timeout` option.

24.8.4 Global and default QoS timeouts

If a route-level *QoS* timeout is undefined, the global `Timeout` takes precedence over the default timeout (30 seconds, see the Timeout docs). This means the global *QoS* timeout can override Polly's default of 30 seconds via the *Global Configuration Schema*.

24.9 Extensibility (Polly)[5]

To use your `ResiliencePipeline<T>` provider, you can apply the following syntax:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddPolly<MyProvider>();
// MyProvider should implement IPollyQoSResiliencePipelineProvider
↪<HttpResponseMessage>
// Note: you can use standard provider PollyQoSResiliencePipelineProvider
```

Additionally, if you want to utilize your own `DelegatingHandler`, the following syntax can be applied:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddPolly<MyProvider>(MyQosDelegatingHandlerDelegate);
// MyQosDelegatingHandlerDelegate is a delegate use to get a DelegatingHandler.
↪ Refer to Ocelot's PollyResiliencePipelineDelegatingHandler
```

Finally, to define your own set of exceptions for mapping, you can apply the following syntax:

```
static Error CreateError(Exception e) => new RequestTimedOutError(e);
Dictionary<Type, Func<Exception, Error>> MyErrorMapping = new()
{
    {typeof(TaskCanceledException), CreateError},
    {typeof(TimeoutRejectedException), CreateError},
    {typeof(BrokenCircuitException), CreateError},
    {typeof(BrokenCircuitException<HttpResponseMessage>), CreateError},
};
builder.Services
    .AddOcelot(builder.Configuration)
    .AddPolly<MyProvider>(MyErrorMapping);
// Note: Default error mapping is defined in the DefaultErrorMapping field of␣
↪the Ocelot.QualityOfService.Polly.OcelotBuilderExtensions class
```

[5] The *Extensibility* feature was requested in issue 1875 and implemented through pull request 1914, as part of version 23.2.

CHAPTER

TWENTYFIVE

RATE LIMITING

Feature label: Rate Limiting

Handy articles:

- What is rate limiting? | Microsoft Cloud | Microsoft Learn
- Rate Limiting pattern | Azure Architecture Center | Microsoft Learn
- Rate limit an HTTP handler in .NET | .NET | Microsoft Learn
- Rate limiting middleware in ASP.NET Core | Microsoft Learn

Ocelot implements *rate limiting*[1] for upstream requests to prevent downstream services from being overwhelmed.

25.1 `RateLimitOptions` Schema

Class: FileRateLimitByHeaderRule

As you may already know from the *Configuration* chapter and the *Route Schema* and *Dynamic Route Schema* sections, there is a special `RateLimitOptions` object schema for routes:

```
"RateLimitOptions": {
  // rule, partition by
  "ClientIdHeader": "",
  "ClientWhitelist": [""],
  // management opts
  "EnableRateLimiting": true,
  "EnableHeaders": true,
  // algorithm
  "Limit": 1,
  "Period": "",
  "Wait": "",
  // extended opts
  "StatusCode": 1,
  "QuotaMessage": "",
  "KeyPrefix": ""
}
```

Additionally, the *Global Configuration Schema* allows configuring global *Rate Limiting* options.

[1] Historically, the "*Rate Limiting*" feature is one of Ocelot's oldest and first features. This feature was introduced in pull request 37 and it was initially released in version 1.3.2.

Note 1: The complete route-level `RateLimitOptions` schema, including all available properties, is defined in the C# FileRateLimitByHeaderRule class. The global `RateLimitOptions` schema includes an additional `RouteKeys` array option, which allows grouping routes to which the global options will apply (refer to the C# FileGlobalRateLimitByHeaderRule class for details). If the `RouteKeys` option is not defined in the global `RateLimitOptions`, the global settings will apply to all routes.

Note 2: You do not need to set all of these options due to default values, but the following rule options are required: `Limit` and `Period`. If these required options are undefined and no global configuration is present, Ocelot will fail to start due to an internally generated validation error, which will be visible in the logs.

Note 3: Several *deprecated options* originating from version 24.0 and earlier (see old schema) are retained for one release cycle. Both introduced and *deprecated options* are detailed in the *Configuration 2* table below.

25.2 Configuration[2]

A complete configuration consists of both route-level and global *Rate Limiting*. You can configure the following options in the `GlobalConfiguration` section of ocelot.json:

```
"Routes": [
  {
    "Key": "R1",
    "RateLimitOptions": {
      "ClientWhitelist": ["ocelot-client1-preshared-key"],
      "Limit": 1000,
      "Period": "20s", // (milli)seconds, minutes, hours, days
      "Wait": "1.5m" // (milli)seconds, minutes, hours, days
      "StatusCode": 418, // I'm a teapot -> this is special status
      "QuotaMessage": "Out of coffee! Our bar can only serve up to {0} cups of␣
↪coffee every {1}. In the meantime, why not grab some tea and relax for Retry-
↪After seconds until we're ready to serve again?"
    }
  }
],
"GlobalConfiguration": {
  "BaseUrl": "https://api.ocelot.net",
  "RateLimitOptions": {
    "RouteKeys": ["R1"], // if undefined or empty array, opts will apply to␣
↪all routes
    "ClientIdHeader": "Oc-Client", // std (default) header name
    "Limit": 100,
    "Period": "30s", // ms, s, m, h, d
    "Wait": "1m", // ms, s, m, h, d
    "StatusCode": 429, // Too Many Requests -> standard status
    "QuotaMessage": "Ocelot API calls quota exceeded! Maximum admitted {0} per
↪{1}.", // standard template with 2 parameters
    "KeyPrefix": "ocelot-rate-limiting" // for caching key
  }
}
```

[2] Global *Configuration* feature was introduced in pull request 2294 and delivered in version 24.1.

Schema Option	Description
`ClientIdHeader`	Specifies the header used to identify clients, with “Oc-Client” set as the default.
`ClientWhitelist`	An array that contains the clients exempt from *rate limiting*.
`EnableRateLimiting`	Enables or disables rate limiting. Defaults to `true` (enabled).
`EnableHeaders`	Specifies whether the `X-RateLimit-*` and `Retry-After` headers are enabled. If undefined, defaults to `true` (enabled).
`Limit`	The maximum number of requests a client can make within a given time `Period`.
`Period`	Rate limiting period (fixed window) can be expressed as milliseconds (1ms), as seconds (1s), minutes (1m), hours (1h), or days (1d). If the exact `Limit` of requests is reached (quota exceeded*), the request is immediately blocked, and if `Wait` is defined, a waiting period begins.
`Wait`	Rate limiting wait window (no servicing period) can be expressed as milliseconds (1ms), as seconds (1s), minutes (1m), hours (1h), or days (1d). This option can have shorter or longer durations compared to the fixed window duration specified as `Period`. The waiting interval either extends or shortens the Quota Exceeded period*, which typically ends after the fixed window elapses.
`StatusCode`	The rejection status code returned during the Quota Exceeded period*. Default value: 429 (Too Many Requests).
`QuotaMessage`	Specifies the message displayed when the quota is exceeded. The value to be used as the formatter for the Quota Exceeded* response message. If none specified the default will be informative.
`KeyPrefix`	The counter prefix, used to compose the rate limiting counter caching key to be used by the `IRateLimitStorage` service. Default value: “Ocelot.RateLimiting”

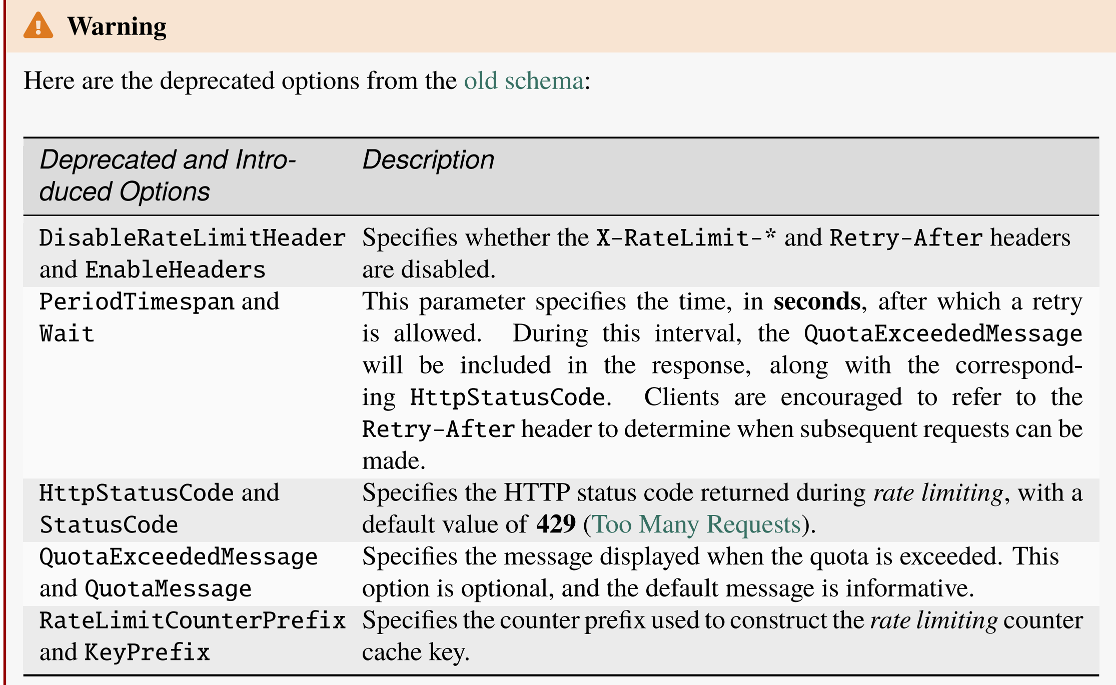
“Quota Exceeded period” term

The **Quota Exceeded period** refers to the `Wait` window, if defined, or the remaining duration of the fixed `Period` following the moment the request `Limit` is exceeded. During this time, the configured rejection `StatusCode` is returned, and the formatted `QuotaMessage` is written to the response body. To determine when this period ends, clients should inspect the `Retry-After` header, which provides a floating-point value indicating the number of seconds until the next allocated fixed window begins. The `X-RateLimit-*` headers are included in the response during the *Quota Exceeded period*, provided that headers are enabled via the `EnableHeaders` option.

Note 1: If the `RouteKeys` option is not defined or the array is empty in the global `RateLimitOptions`, the global settings will apply to all routes. If the array contains route keys, it defines a single group of routes to which the global options apply. Routes excluded from this group must specify their own route-level `RateLimitOptions`.

Note 2: The string values for the `Period` and `Wait` options must contain a floating-point number followed by one of the supported time units: ‘ms’, ‘s’, ‘m’, ‘h’, or ‘d’. If no unit is specified, the value defaults to milliseconds. For example, “333.5” is interpreted as 333 milliseconds and 500 microseconds (equivalent to “333.5ms”). The floating-point component may be omitted; for example, “10.0s” is equivalent to “10s”. These values are parsed dynamically at runtime, so the required `Period` option in ocelot.json is validated early through fluent validation when the Ocelot app starts. If an invalid value is provided, the *Rate Limiting* middleware will throw a `FormatException`, which is logged accordingly.

25.2.1 Deprecated options[3]



Warning

Here are the deprecated options from the old schema:

Deprecated and Introduced Options	*Description*
`DisableRateLimitHeader` and `EnableHeaders`	Specifies whether the `X-RateLimit-*` and `Retry-After` headers are disabled.
`PeriodTimespan` and `Wait`	This parameter specifies the time, in **seconds**, after which a retry is allowed. During this interval, the `QuotaExceededMessage` will be included in the response, along with the corresponding `HttpStatusCode`. Clients are encouraged to refer to the `Retry-After` header to determine when subsequent requests can be made.
`HttpStatusCode` and `StatusCode`	Specifies the HTTP status code returned during *rate limiting*, with a default value of **429** (Too Many Requests).
`QuotaExceededMessage` and `QuotaMessage`	Specifies the message displayed when the quota is exceeded. This option is optional, and the default message is informative.
`RateLimitCounterPrefix` and `KeyPrefix`	Specifies the counter prefix used to construct the *rate limiting* counter cache key.

25.3 Notes

Note

1. Prior to version 24.1, global options were only accessible in the special *Dynamic Routing* mode. Since version 24.1, global configuration has been available for both static and dynamic routes. As a team, we would consider the idea of implementing such a global configuration for aggregated routes. However, an aggregated route is essentially a combination of static routes.

2. Global *rate limiting* options may not be practical as they apply limits to all routes. In a microservices architecture, it is unusual to enforce the same limitations across all routes. Configuring per-route *rate limiting* could offer a more tailored solution. However, global *rate limiting* can be logical if all routes share the same downstream hosts, thereby restricting the usage of a single service or a single product.

3. The `DisableRateLimitHeaders` option is deprecated as of version 24.1. Use `EnableHeaders` instead, applying boolean value negation as needed. If `DisableRateLimitHeaders` is defined, it takes precedence; otherwise, `EnableHeaders` will be used. Do not define both options. This setting is retained for backward compatibility but is subject to change. Therefore, the `DisableRateLimitHeaders` option will be removed in the upcoming major release, version 25.0. The same applies to other *deprecated options*.

4. Ocelot's own *rate limiting* does not utilize built-in ASP.NET Core features, so it is not based on the "Rate limiting middleware in ASP.NET Core" described in the *Roadmap* below. The Ocelot team

[3] Several *deprecated options* originating from version 24.0 and earlier (see old schema) are retained for one release cycle. They are likely to be removed in the upcoming major release, version 25.0, which will include a significant upgrade to the *Rate Limiting* feature (refer to the *Roadmap*). The Ocelot team plans to implement an automatic configuration upgrade mechanism to support backward compatibility. However, we recommend reviewing the updated schema and beginning to adopt the new options.

believes that the ASP.NET Core rate limiting middleware enables global limitations through its rate-limiting policies.

25.4 Algorithms

The currently implemented rate limiter algorithms in Ocelot are:

- **Fixed window**: Based on the `Period` option, without the `Wait` option (previously known as the deprecated `PeriodTimespan`).
- **Hybrid fixed window**: The combination of `Period` and `Wait` enables fixed-window-like behavior with additional control over the duration and handling of the *"Quota Exceeded period"*.

Historically, Ocelot's rate limiting algorithm was a hybrid, combining the classic "fixed window" approach with a waiting no-service period. Since version 24.1, the hybrid algorithm has been split into two distinct algorithms, allowing the classic "fixed window" to be used independently.

To understand the terminology, please refer to the Handy Articles listed at the beginning of this chapter. For beginners, here is a quick link: Announcing ASP.NET Core rate limiting algorithms. For professionals, we recommend reading the official Microsoft Learn article "Rate limiting middleware in ASP.NET Core", especially the Rate Limiter Algorithms section, and/or searching the internet for additional resources.

> **Note 1**: Ocelot's own rate limiter does not implement other classic algorithms such as "Sliding Window", "Token Bucket", or "Concurrency". However, these algorithms are outlined in the *Roadmap*.
>
> **Note 2**: Ocelot's own rate limiter does not manage concurrent HTTP requests via a queue. Therefore, all concurrency handling and decision-making should be implemented on the client side using classic retry patterns to ensure quality of service. The management strategy is deliberately simple: *First-In means First Wins*. If the first request acquires a virtual lease from the limiting quota and the quota is immediately exceeded, the second request will be rejected with a 429 Too Many Requests response.

25.5 Rules (Partitions)

Ocelot's rate limiting *rule* is a superset of the configuration options used to set up rate-limited access to a route. It enables partitioned rate limiting by processing the following artifacts through distinct stages: the client's identifier, a dedicated partition counter (quota), rate limiter algorithms, and the quota-exceeded response behavior.

25.5.1 By Client's Header

> Class: FileRateLimitByHeaderRule
> JSON: *RateLimitOptions Schema*

Currently, Ocelot's own rate limiting middleware supports and processes only the *"By Client's Header"* rule (partition), commonly referred to as the "API Key partition" in ASP.NET Core terminology. Ocelot's rate limiting architecture provides dedicated subpartitions for each route, each with an independent counter for the rate limiter algorithm. Therefore, when client traffic enters the Ocelot pipeline, the current request is processed as follows:

1. Ocelot identifies the route by matching the URL path to the upstream route path, and allows the rate limiting middleware to process the client as part of the route partition.
2. Ocelot's rate limiting middleware creates the client's identity based on the configured *"By Client's Header"* rule and assigns a dedicated rate limiter counter to that client.

3. The rate limiting middleware executes the configured rate limiter algorithm, specifically the (hybrid) fixed window. Refer to the currently implemented *Algorithms* for details.
4. If the quota is exceeded, the rate limiting middleware returns appropriate "Quota Exceeded period" artifacts in the response, such as the status code, body message, and headers including `Retry-After`.

> **Note**
>
> If the client is not successfully identified, the rate limiting middleware blocks the request with a 503 Service Unavailable status and writes an appropriate error message to the response body. Possible reasons for an empty identity include a missing header or an invalid `ClientIdHeader` value, as explained in the warning below. Whitelisted clients (defined via the `ClientWhitelist` option) are processed without limitation.

> **Warning**
>
> Ocelot's rate limiting middleware is not responsible for validating API keys, also known as client header values, to be read from the configured header (`ClientIdHeader` option). Users and developers must register these header values as pre-shared API keys on Ocelot's side and ensure they are validated before handing control over to the `RateLimitingMiddleware`.
>
> We recommend implementing a custom middleware to validate API keys (client header values) and injecting it into the Ocelot pipeline using the *Middleware Injection* feature. Specifically, the `PreErrorResponderMiddleware` (position 3) should be overridden, as it is invoked before the `RateLimitingMiddleware` at position 10. A more advanced solution may involve using the `SecurityMiddleware` at position 7, but in this case, users must implement their own `ISecurityPolicy` service and replace it in the *Dependency Injection* (DI) container. To understand the Ocelot pipeline and its middleware positions, refer to the "*Ocelot Pipeline Builder*" documentation.

25.6 Roadmap

Feature label: **`Rate Limiting`_**
Development history: Rate Limiting[4]

- **Rules**: The Ocelot team is considering a redesign of the *Rate Limiting* feature in light of the "Announcing Rate Limiting for .NET" article by Brennan Conroy, published on July 13th, 2022.

> **Note**
>
> Discover the new rate limiting functionality in ASP.NET Core:
>
> – The RateLimiter Class, available since ASP.NET Core 7.0
>
> – The System.Threading.RateLimiting NuGet package
>
> – The Rate limiting middleware in ASP.NET Core article by Arvin Kahbazi, Maarten Balliauw, and Rick Anderson

As of now, the decision has been made to retain Ocelot's own RateLimitingMiddleware and extend it with an additional rule that will reference the attached ASP.NET Core rate limiting policy. This

[4] Since pull request 37 and version 1.3.2, the Ocelot team has reviewed and redesigned the *Rate Limiting* feature. A fix for bug 1590 (pull request 1592) was released as part of version 23.3 to ensure stable behavior. Global *configuration* support was introduced in pull request 2294 and delivered in version 24.1.

new rule is highly likely to be delivered in version 25.0, following the opening of pull request 2188.

- **Algorithms**: In addition to the currently implemented hybrid “Fixed window” algorithm, which is built into Ocelot, the team plans to introduce other industry-standard algorithms, such as “Sliding window”, “Token bucket”, and “Concurrency, with priority given to “Sliding window” as the first. These lightweight algorithms should be easily configurable via JSON by end users who are not .NET developers, in order to avoid writing additional C# source code. Other interesting algorithms are welcome for discussion.

We encourage you to share your thoughts with us in the Discussions of the repository. Filter the current discussions by the Rate Limiting label.

CHAPTER

TWENTYSIX

ROUTING

Ocelot's primary function is to handle incoming HTTP requests and forward them to a downstream service. Currently, Ocelot supports this only through HTTP requests. In the future, it might support other transport mechanisms.

Ocelot defines the process of routing one request to another as a "Route". To make Ocelot functional, you must set up a *route* in its configuration.

```
{
  "Routes": []
}
```

To configure a *route*, you need to add one to the `Routes` JSON array.

```
{
  "UpstreamHttpMethod": [ "Get", "Post" ],
  "UpstreamPathTemplate": "/posts/{postId}",
  "DownstreamPathTemplate": "/api/posts/{postId}",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 80 }
  ]
}
```

The `DownstreamPathTemplate`, `DownstreamScheme`, and `DownstreamHostAndPorts` properties define the URL to which a request will be forwarded.

The `DownstreamHostAndPorts` property is a collection that specifies the host and port of downstream services to which you intend to forward requests. Typically, it contains a single entry; however, in cases where *load balancing* is required, Ocelot allows you to add multiple entries and select an appropriate *Load Balancer*.

The `UpstreamPathTemplate` property specifies the URL that Ocelot uses to determine the appropriate `DownstreamPathTemplate` for a given request. The `UpstreamHttpMethod` property enables Ocelot to differentiate between requests with different HTTP verbs directed to the same URL. You can either specify a particular list of HTTP methods or leave the list empty to allow all methods.

> **Note**: The complete schema on a single *route* object is described in the *Route Schema* section of the *Configuration* feature.

26.1 Placeholders

In Ocelot, you can add placeholders for variables to your templates using the format of `{something}`. The placeholder variable must be included in both the `DownstreamPathTemplate` and `UpstreamPathTemplate` properties. When present, Ocelot attempts to substitute the value of the placeholder from the `UpstreamPathTemplate` into the `DownstreamPathTemplate` for each request it processes.

You can also do a *Catch All* type of *route* e.g.

```
{
  "UpstreamHttpMethod": [ "Get", "Post" ],
  "UpstreamPathTemplate": "/{everything}",
  "DownstreamPathTemplate": "/api/{everything}",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 80 }
  ]
}
```

This will forward all path and query string combinations to the downstream service, appending them after the `/api` path.

Note: The default routing configuration is **case-insensitive**. To change this, you can specify the following setting on a per-route basis:

```
"RouteIsCaseSensitive": true
```

This means that when Ocelot attempts to match an incoming upstream URL with an upstream template, the evaluation will be *case-sensitive*.

26.1.1 Embedded Placeholders[1]

Before version 23.4, Ocelot could not evaluate multiple placeholders embedded between two forward slashes (`/`). Additionally, it faced difficulties distinguishing placeholders from other elements within the slashes. For example, when the pattern `/{url}-2/` was applied to `/y-2/`, it would produce `{url}` with `y-2` value.

We have introduced an improved method for placeholder evaluation, making it easier to identify placeholders in complex URLs.

Example:

- **Path Pattern**: `/api/invoices_{url0}/{url1}-{url2}_abcd/{url3}?urlId={url4}`
- **Upstream URL Path**: `/api/invoices_super/123-456_abcd/789?urlId=987`
- **Resulting Placeholders**:
  - `{url0}` = `super`
  - `{url1}` = `123`
  - `{url2}` = `456`
  - `{url3}` = `789`
  - `{url4}` = `987`

[1] The "*Embedded Placeholders*" feature was requested as part of issue 2199 , and released in version 23.4

Note, we believe this feature should be compatible with any URL query strings, although it has not been thoroughly tested.

26.1.2 Empty Placeholders[2]

This represents a special edge case of *Placeholders*, in which the value of the placeholder is simply an empty string ("").

For example, given the following *route* configuration:

```
{
  "UpstreamPathTemplate": "/invoices/{url}",
  "DownstreamPathTemplate": "/api/invoices/{url}",
}
```

This route works correctly when `{url}` is specified. For instance:

- `/invoices/123` `/api/invoices/123`

Edge Cases with Empty Placeholder Values:

1. **Empty Placeholder**: When `{url}` is empty, the upstream path `/invoices/` routes to the downstream path `/api/invoices/`.
2. **Omitting the Last Slash**: When the trailing slash is omitted, the upstream path `/invoices` should still route to the downstream path `/api/invoices`. This behavior aligns intuitively with user expectations.

26.2 Catch All

Ocelot's *routing* supports a *"Catch All"* style, allowing users to specify routes that match all incoming traffic.

If you configure your settings as shown below, all requests will be proxied directly. The placeholder `{catchAll}` is not significant, and any name can be used.

```
{
  "UpstreamPathTemplate": "/{catchAll}",
  "DownstreamPathTemplate": "/{catchAll}",
  // ...
}
```

The *"Catch All"* route has a lower *priority* than other routes. If the following route is included in your configuration, Ocelot will match it before the *"Catch All"* route.

```
{
  "UpstreamPathTemplate": "/",
  "DownstreamPathTemplate": "/",
  // ...
}
```

[2] The "*Empty Placeholders*" feature is available starting in version 23.0, see issue 748 and the 23.0 release notes for details.

26.3 Priority[3]

You can define the order in which your *routes* match the upstream URL by including a `Priority` property in the ocelot.json file.

```
{
  "Priority": 0
}
```

Priority **0** is the lowest *priority*. Ocelot always assigns `0` to *Catch All* routes, and this value is still hard-coded. Beyond that, you are free to assign any *priority* you wish.

e.g. you could have

```
{
  "UpstreamPathTemplate": "/goods/{catchAll}",
  "Priority": 0
}
```

and

```
{
  "UpstreamPathTemplate": "/goods/delete",
  "Priority": 1
}
```

In the example above, if a request is made to Ocelot on `/goods/delete`, it will match the `/goods/delete` route. Previously, it would have matched `/goods/{catchAll}`, as this was the first *route* in the list.

26.4 Query Placeholders

In addition to URL path *Placeholders*, Ocelot can forward query string parameters, processing them in the form of `{something}`. The query parameter placeholder must be included in both the `DownstreamPathTemplate` and `UpstreamPathTemplate` properties. Placeholder replacement works bi-directionally between paths and query strings, although it is subject to certain restrictions (see *Merging of Query Parameters*).

26.4.1 Path to Query String direction

Ocelot allows you to include a query string as part of the `DownstreamPathTemplate`, as demonstrated in the following example:

```
{
  "UpstreamPathTemplate": "/api/units/{subscription}/{unit}/updates",
  "DownstreamPathTemplate": "/api/subscriptions/{subscription}/updates?unitId=
↪{unit}",
}
```

In this example, Ocelot uses the value of the `{unit}` placeholder from the upstream path template and includes it in the downstream request as a query string parameter named `unitId`.

> **Note**: Ensure that the placeholder is named differently to account for the *Merging of Query Parameters*.

[3] The "*Priority*" feature was requested as part of issue 270, and released in version 5.0.1

26.4.2 Query String to Path direction

Ocelot also allows you to include query string parameters in the `UpstreamPathTemplate`, enabling you to match specific queries to corresponding services:

```
{
  "UpstreamPathTemplate": "/api/subscriptions/{subscriptionId}/updates?unitId=
↪{uid}",
  "DownstreamPathTemplate": "/api/units/{subscriptionId}/{uid}/updates",
}
```

In this example, Ocelot matches only requests with a corresponding URL path where the query string begins with `unitId=something`. Additional queries are permitted but must follow the matching parameter. Additionally, Ocelot replaces the `{uid}` parameter in the query string and incorporates it into the downstream request path.

> **Note**: The best practice is to use a placeholder name that differs from the name of the query parameter to accommodate the *Merging of Query Parameters*.

26.4.3 Catch All Query String

Ocelot's *routing* also supports a "*Catch All*" style, allowing all query string parameters to be forwarded. The placeholder `{query}` is not significant, and any name can be used.

```
{
  "UpstreamPathTemplate": "/contracts?{query}",
  "DownstreamPathTemplate": "/apipath/contracts?{query}",
}
```

This query string routing feature is particularly useful in scenarios where the query string needs to be routed without any transformations—for example, OData filters (see issue 1174).

> **Note**: The `{query}` placeholder can remain empty while catching all query strings, as this functionality is part of the "*Empty Placeholders*" feature[Page 141, 2]. Consequently, upstream paths `/contracts?` and `/contracts` are routed to the downstream path `/apipath/contracts`, with no query string attached.

26.4.4 Merging of Query Parameters

Query string parameters are unsorted and merged to form the final downstream URL. This process is crucial because the `DownstreamUrlCreatorMiddleware` requires control over placeholder replacement and the merging of duplicate parameters. A parameter that appears first in the `UpstreamPathTemplate` may occupy a different position in the final downstream URL. Moreover, if the `DownstreamPathTemplate` includes query parameters at the beginning, their positions in the `UpstreamPathTemplate` will remain undefined unless explicitly specified.

In a typical scenario, the merging algorithm constructs the final downstream URL query string as follows:

1. It begins by taking the initially defined query parameters in the `DownstreamPathTemplate` and placing them at the start, along with any necessary placeholder replacements.
2. Next, it adds all parameters from the *Catch All Query String*, represented by the placeholder `{query}`, in the second position—following the explicitly defined parameters from *step 1*.
3. Finally, it appends any remaining replaced placeholder values as parameter values to the end of the query string, if present.

Array parameters in ASP.NET API's model binding

Due to the merging of parameters, ASP.NET API's special model binding for arrays does not support the array item representation format `selectedCourses=1050&selectedCourses=2000`. This query string will be merged into `selectedCourses=1050` in the downstream URL, leading to the loss of array data. It is essential for upstream clients to generate the correct query string for array models, such as `selectedCourses[0]=1050&selectedCourses[1]=2000`. For a detailed explanation of array model binding, refer to the documentation: "Bind arrays and string values from headers and query strings".

Control over parameter existence

Be aware that query string placeholders are subject to naming restrictions due to the implementation of the `DownstreamUrlCreatorMiddleware` merging algorithm. Nevertheless, this restriction also offers flexibility in managing the presence of parameters in the final downstream URL based on their names.

Consider the following two development scenarios

1. A developer wishes **to preserve a parameter** after substituting a placeholder (refer to issue 473). This requires the use of the following template definition:

```
{
  "UpstreamPathTemplate": "/path/{serverId}/{action}",
  "DownstreamPathTemplate": "/path2/{action}?server={serverId}"
}
```

   In this case, the `{serverId}` placeholder and the server parameter **names differ**. As a result, the `server` parameter is retained.
   It is important to note that, due to the case-sensitive comparison of names, the `server` parameter will not be preserved with the `{server}` placeholder. However, using the `{Server}` placeholder is acceptable for retaining the parameter.

2. The developer intends **to remove an outdated parameter** after substituting a placeholder (refer to issue 952). To achieve this, identical names must be used, adhering to case-sensitive comparison rules.

```
{
  "UpstreamPathTemplate": "/users?userId={userId}",
  "DownstreamPathTemplate": "/persons?personId={userId}"
}
```

   Thus, the `{userId}` placeholder and the `userId` parameter **have identical names**. As a result, the `userId` parameter is eliminated.
   Be aware that, due to the case-sensitive nature of the comparison, the `userId` parameter will not be removed if the `{userid}` placeholder is used.

26.5 Upstream Host[4]

This feature allows you to define routes based on the *upstream host*. It works by examining the `Host` header used by the client and incorporating it into the information used to identify a *route*.

[4] The "*Upstream Host*" feature was requested as part of issue 209 (pull request 216), and released in version 3.0.1

In order to use this feature, add the following to your configuration:

```
{
  "UpstreamHost": "mydomain.com"
}
```

The *route* above will only match requests where the `Host` header value is `mydomain.com`.

If you do not set the `UpstreamHost` on a *route*, any `Host` header will match it. As a result, if you have two routes that are identical except for the `UpstreamHost`, where one is null and the other is set, Ocelot will prioritize the one that is set.

26.6 Upstream Headers[5]

In addition to routing by `UpstreamPathTemplate`, you can also define `UpstreamHeaderTemplates`. For a *route* to match, all headers specified in this dictionary object must be included in the request headers.

```
{
  "UpstreamPathTemplate": "/",
  "UpstreamHeaderTemplates": { // dictionary
    "country": "uk", // 1st header
    "version": "v1"  // 2nd header
  }
}
```

In this scenario, the *route* matches only if a request contains both headers with the specified values.

26.6.1 Header placeholders

Let's explore a more interesting scenario where placeholders can be effectively utilized within your `UpstreamHeaderTemplates`.

Consider the following approach using the special placeholder format `{header:placeholdername}`:

```
{
  // downstream opts...
  "DownstreamPathTemplate": "/{versionnumber}/api", // with placeholder
  // upstream opts...
  "UpstreamHeaderTemplates": {
    "version": "{header:versionnumber}" // 'header:' prefix vs placeholder
  }
}
```

In this scenario, the entire value of the request header `version` is inserted into the `DownstreamPathTemplate`. If needed, a more complex upstream header template can be specified using placeholders such as `version-{header:version}_country-{header:country}`.

> **Note 1**: Placeholders are not required in the `DownstreamPathTemplate`. This scenario can be used to enforce a specific header, regardless of its value.
>
> **Note 2**: Additionally, the `UpstreamHeaderTemplates` dictionary options are applicable for *Aggregation* as well.

[5] The "*Upstream Headers*" feature was proposed in issue 360 (pull request 1312), and released in version 23.3.

26.7 Security Options[6]

Ocelot facilitates the management of multiple patterns for allowed and blocked IPs using the IPAddress-Range package, which is licensed under the MPL-2.0 license.

This feature is designed to enhance IP management, allowing for the inclusion or exclusion of a broad IP range using CIDR notation or specific IP ranges. The current managed patterns are as follows:

IP Rule	*Example*
Single IP	`192.168.1.1`
IP Range	`192.168.1.1-192.168.1.250`
IP Short Range	`192.168.1.1-250`
IP Subnet	`192.168.1.0/255.255.255.0`
CIDR IPv4	`192.168.1.0/24`
CIDR IPv6	`fe80::/10`

Here is a simple example:

```
{
  "SecurityOptions": {
    "IPBlockedList": [ "192.168.0.0/23" ],
    "IPAllowedList": ["192.168.0.15", "192.168.1.15"],
    "ExcludeAllowedFromBlocked": true
  }
}
```

Please **note**:

- The allowed/blocked lists are evaluated during configuration loading.
- The `ExcludeAllowedFromBlocked` property enables specifying a wide range of blocked IP addresses while allowing a subrange of IP addresses. Default value: `false`.
- The absence of a property in *Security Options* is permitted, as it takes the default value.
- *Security Options* can be configured *globally* in the `GlobalConfiguration` JSON[7]. However, they are ignored if overriding options are specified at the route level.

26.8 Dynamic Routing[8]

The concept of dynamic *routing* allows you to use a *Service Discovery* provider, eliminating the need to manually configure *route* settings. For more details, refer to the *Dynamic Routing* complete reference in the "*Service Discovery*" chapter.

26.9 Errors and Gotchas

In this section, Ocelot team has gathered user scenarios where routing behavior was unclear or errors appeared in the logs. Please note that the failed routing cases listed below do not represent all application configurations. If your case is not included, feel free to open a "Show and tell" discussion.

[6] The "*Security Options*" feature was requested as part of issue 628 (version 12.0.1), then redesigned and improved by issue 1400 (version 23.4.1), and published in version 20.0 docs.

[7] Global "*Security Options*" feature was requested as part of issue 2165 , and released in version 23.4.1.

[8] The "*Dynamic Routing*" feature was requested as part of issue 340, and released in version 7.0.1. Refer to complete reference in the "*Service Discovery*" chapter: *Dynamic Routing*.

- **Magic 499 status**. According to Ocelot Core's design, HTTP status code 499 (Client Closed Request) is returned in cases involving an `OperationCanceledException`. Please note that due to extensive warning-level logging, you may encounter spikes in `499` responses—as discussed in thread 2072. This status is typically caused by:

  A) Forced cancellation of the request by the client

  B) Browser events such as page refreshes or closures while the downstream request is still in progress

  As a quick recipe, the Ocelot team recommends ensuring client stability and, if necessary, adjusting the *Timeout* strategy: either increasing or decreasing the route *Timeout* depending on your usage scenario and the behavior of the downstream service.

- **Timeout errors aka 503 status**. According to Ocelot Core's design, HTTP status code 503 (Service Unavailable) is returned in cases involving a `TimeoutException`. This status is typically caused by:

  A) Slow downstream services that may fail to respond

  B) Large requests forwarded to slow downstream services

  As a quick recipe, the Ocelot team recommends increasing the route *Timeout* in your configuration. This adjustment can help resolve timeout-related issues with sluggish downstream services, ultimately reducing occurrences of 503 (Service Unavailable).

  Note: For comprehensive documentation regarding errors and status codes in Ocelot, please refer to the *Error Handling* chapter.

CHAPTER

TWENTYSEVEN

SERVICE DISCOVERY

Ocelot allows you to specify a *service discovery* provider, which it uses to determine the host and port for the downstream service to which it forwards requests. Currently, this feature is only supported in the `GlobalConfiguration` section. This means the same *service discovery* provider is applied to all routes where a `ServiceName` is specified at the route level.

27.1 Consul

Package: Ocelot.Discovery.Consul
Namespace: `Ocelot.Discovery.Consul`
Repository: ThreeMammals/Ocelot.Discovery.Consul

The first step is to install the package, which adds Consul support to Ocelot:

```
dotnet add package Ocelot.Discovery.Consul
```

To register *Consul* services, invoke the `AddConsul()` extension method using the `OcelotBuilder` returned by `AddOcelot()`[1]. Include the following code in your Program:

```
using Ocelot.Discovery.Consul;

builder.Services
    .AddOcelot(builder.Configuration)
    .AddConsul(); // or .AddConsul<T>()
```

Currently, there are two types of *Consul* service discovery providers: `Consul` and `PollConsul`. The default provider is `Consul`. If the `ConsulProviderFactory` cannot read, understand, or parse the `Type` property of the `ServiceProviderConfiguration` object, a *Consul Provider* instance is created by the factory. Explore these types of *service discovery* providers and learn about their differences in the subsections: *Consul Provider* and *PollConsul Provider*.

Note

We have made the *Consul Provider* the default *service discovery* provider in Ocelot.

[1] The *AddOcelot method* adds default ASP.NET services to the DI container. You can call another extended *AddOcelotUsingBuilder method* while configuring services to develop your own *Custom Builder*. See more instructions in the "*AddOcelotUsingBuilder method*" section of the *Dependency Injection* feature.

> **Warning**
>
> Prior to version 25.0, the package was named Ocelot.Provider.Consul. If you are using version 24.1 or earlier, install the Ocelot.Provider.Consul package. For version 25.0 and later, the package ID is Ocelot.Discovery.Consul.

27.1.1 Configuration in KV Store

Add the following when registering your services. Ocelot will attempt to store and retrieve its *Configuration* in the *Consul* KV Store:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddConsul()
    .AddConfigStoredInConsul();
```

You also need to add the following to your ocelot.json file. This allows Ocelot to locate your *Consul* agent and handle configuration loading and storage from *Consul*.

```
"GlobalConfiguration": {
  "ServiceDiscoveryProvider": {
    "Host": "localhost",
    "Port": 9500
  }
}
```

The team decided to create this feature after working on the Raft consensus algorithm and realizing how challenging it was. Why not take advantage of the fact that Consul already provides this functionality? We believe this means that, to use Ocelot to its fullest potential, you currently need to adopt *Consul* as a dependency.

> **Note**: This feature has a 3-second TTL cache before it makes a new request to your local *Consul* agent.

27.1.2 Configuration Key[2]

If you are using *Consul* for *Configuration* (or other providers in the future), you may want to assign keys to your configurations. This allows you to manage multiple configurations.

In order to specify the key, you need to set the `ConfigurationKey` property in the `ServiceDiscoveryProvider` options of the configuration JSON file. For example:

```
"GlobalConfiguration": {
  "ServiceDiscoveryProvider": {
    "Host": "localhost",
    "Port": 9500,
    "ConfigurationKey": "Ocelot_A"
  }
}
```

In this example, Ocelot will use `Ocelot_A` as the key for your configuration when looking it up in *Consul*. If you do not set the `ConfigurationKey`, Ocelot will default to using the string `InternalConfiguration` as the key.

[2] The "*Configuration Key*" feature was requested in issue 346 and introduced in version 7.0.0.

27.1.3 `Consul` Provider

Class: Ocelot.Discovery.Consul.Consul

The following is required in the `GlobalConfiguration` section. The `ServiceDiscoveryProvider` property is mandatory. If you do not specify a host and port, the default Consul values will be used.

```
"ServiceDiscoveryProvider": {
  "Scheme": "https",
  "Host": "localhost",
  "Port": 8500,
  "Type": "Consul"
}
```

In the future, we may add a feature that allows route-specific configuration.

Note: The `Scheme` option defaults to HTTP. This was introduced in pull request 1154 and defaults to `http` to avoid introducing a breaking change.

To instruct Ocelot that a route should use the *service discovery* provider for its host and port, you need to specify the `ServiceName` and the load balancer you wish to use for downstream requests. Currently, Ocelot supports the RoundRobin and LeastConnection algorithms. If no load balancer is specified, Ocelot will not perform load balancing for requests.

```
{
  "ServiceName": "product",
  "LoadBalancerOptions": {
    "Type": "LeastConnection"
  }
}
```

When set up, Ocelot will look up the downstream host and port from the *service discovery* provider and balance requests across available services.

Implementation tips: The `GetAsync()` method fetches health service entries and catalog nodes simultaneously using `Task.WhenAll`, minimizing round-trip latency when *Consul* is accessed over the network. If no healthy service entries are found for the configured service name, a warning is logged and an empty list is returned. On the happy path, the provider hands control over to the *Consul Service Builder*, which constructs the services for consumption by Ocelot's middleware components (e.g. load balancing).

27.1.4 `PollConsul` Provider

Class: Ocelot.Discovery.Consul.PollConsul

A lot of users have requested a feature where Ocelot *polls Consul* for the latest service information instead of doing so per request. If you want Ocelot to *poll Consul* for the latest services, rather than relying on the default behavior (per request), you need to configure the following options:

```
"ServiceDiscoveryProvider": {
  "Host": "localhost",
  "Port": 8500,
  "Type": "PollConsul",
  "PollingInterval": 100 // ms
}
```

The polling interval, measured in milliseconds, specifies how frequently Ocelot calls Consul for service configuration updates.

There are **trade-offs** to consider. If you *poll Consul*, Ocelot may not detect if a service is down, depending on your polling interval. This could result in more errors compared to retrieving the latest services per request. The impact largely depends on the volatility of your services. For most users, this is unlikely to be a significant concern, and polling may offer a slight performance improvement over querying Consul per request (as a sidecar agent). However, if you are communicating with a remote Consul agent, polling provides a more noticeable performance improvement.

Note

Implementation tips

1. *First-call behavior:* On the very first request, `PollConsul` always fetches fresh service data from *Consul* regardless of the polling interval, because the internal timer is initialized to `DateTime.MinValue`. Subsequent requests respect the configured `PollingInterval` and return the cached list until the interval elapses.
2. *Thread safety:* The `PollConsul` provider uses a lock object to serialize access to the shared services list, making it safe for concurrent requests.
3. `ServiceName` *property*: The provider exposes a `ServiceName` property that identifies which downstream service this instance is polling. The `ConsulProviderFactory` uses this property to ensure a single `PollConsul` instance is created per service name across all concurrent requests.

27.1.5 Service Definition

Your services need to be added to Consul in a manner similar to the example below (C# style, but hopefully it makes sense). The key point to note is to avoid including `http` or `https` in the `Address` field. We have received feedback regarding issues with the scheme being included in the `Address`. After reviewing the "Agent Basics" and "Define services" documentation, we believe the **scheme** should not be included.

In C#

```
new AgentService()
{
    ID = "some-id",
    Service = "some-service-name",
    Address = "localhost",
    Port = 8080,
}
```

Or, in JSON

```
"Service": {
  "ID": "some-id",
  "Service": "some-service-name",
  "Address": "localhost",
  "Port": 8080
}
```

27.1.6 ACL Token

If you are using ACL with *Consul*, Ocelot supports adding the `X-Consul-Token` header. To enable this functionality, you must add the following option:

```
"ServiceDiscoveryProvider": {
  "Host": "localhost",
  "Port": 8500,
  "Type": "Consul",
  "Token": "my-token"
}
```

Ocelot will add this token to the *Consul* client it uses for making requests, and this token will be applied to all subsequent requests.

27.1.7 Consul Service Builder[3]

Interface: `IConsulServiceBuilder`
Implementation: `DefaultConsulServiceBuilder`

The IConsulServiceBuilder interface defines three public methods that control the entire service-building pipeline:

Method	Description
`bool IsValid(ServiceEntry entry)`	Validates a *Consul* service entry before it is converted into an Ocelot `Service` object. Entries with an empty `Address`, a scheme prefix in the address (`http://` or `https://`), or a zero/negative `Port` are rejected with a warning log message.
`IEnumerable<Service> BuildServices(ServiceEntry[] entries, Node[] nodes)`	Iterates over all health entries returned by *Consul*, filters them through `IsValid`, resolves the matching catalog node, and delegates individual construction to `CreateService`.
`Service CreateService(ServiceEntry entry, Node node)`	Assembles a single Ocelot `Service` object from a *Consul* health entry and its optional catalog node.

The Ocelot community has consistently reported issues with *Consul* services, both in the past and present, such as connectivity problems due to varying *Consul* agent definitions. Some DevOps engineers prefer grouping services as *Consul* catalog nodes by customizing the assignment of hostnames to node names, while others prioritize defining agent services using pure IP addresses as hosts, which is linked to the 954-bug dilemma.

Since version 13.5.2, the process for constructing the downstream host and port in pull request 909 has been changed to prioritize the node name as the host over the agent service address IP. This may raise some criticism from the community.

Version 23.3 introduced a customization feature that enables control over the service-building process through the `DefaultConsulServiceBuilder` class. This class includes virtual methods that developers and DevOps teams can override to suit their specific requirements.

The DefaultConsulServiceBuilder class exposes the following protected and public virtual methods as override points:

[3] The customization of "*Consul Service Builder*" was implemented as part of bug fix 954, and the feature was delivered in version 23.3.

Public virtual method	Description
`bool IsValid(ServiceEntry entry)`	Returns `true` when the entry's `Address` is non-empty, does not start with `http://` or `https://`, and `Port` is greater than zero. Override to implement custom validation logic.
`IEnumerable<Service> BuildServices(ServiceEntry[] entries, Node[] nodes)`	Filters, resolves, and builds the full list of services. Override for full control over how the collection is assembled.
`Service CreateService(ServiceEntry entry, Node node)`	Constructs a single `Service` by delegating to the property-level helpers below. Override to change the assembled type or add extra data.
Protected virtual method	**Description**
`Node GetNode(ServiceEntry entry, Node[] nodes)`	Resolves the catalog node for an entry by first checking `entry.Node`, then searching the `nodes` array for a node whose `Address` matches `entry.Service.Address`. Override to apply a different node-selection strategy.
`string GetDownstreamHost(ServiceEntry entry, Node node)`	Returns `node.Name` when a catalog node is available, otherwise falls back to `entry.Service.Address`. Override this method when you want to use the service IP address instead of the node name.
`ServiceHostAndPort GetServiceHostAndPort(ServiceEntry entry, Node node)`	Combines the result of `GetDownstreamHost` with `entry.Service.Port`. Override to provide a fully custom host-and-port resolution strategy.
`string GetServiceName(ServiceEntry entry, Node node)`	Returns `entry.Service.Service`.
`string GetServiceId(ServiceEntry entry, Node node)`	Returns `entry.Service.ID`.
`string GetServiceVersion(ServiceEntry entry, Node node)`	Extracts the version from the service tags by locating the first tag prefixed with `version-` and stripping that prefix (e.g. the tag `version-v2` yields `v2`). Returns an empty string when no version tag is present.
`IEnumerable<string> GetServiceTags(ServiceEntry entry, Node node)`	Returns `entry.Service.Tags`, or an empty enumerable when the tags collection is `null`.

The most frequently customized method is `GetDownstreamHost`. Its default logic is:

```
protected virtual string GetDownstreamHost(ServiceEntry entry, Node node)
    => node != null ? node.Name : entry.Service.Address;
```

Some DevOps engineers choose to disregard node names, opting for abstract identifiers instead of actual hostnames. However, our team strongly recommends assigning real hostnames or IP addresses to node names, considering this a best practice. If this approach does not align with your needs, or if you prefer not to invest time in detailing nodes for downstream services, you could define agent services without node names. In such cases, within a *Consul* setup, you would need to override the behavior of the `DefaultConsulServiceBuilder` class.

For further information, refer to the "*AddConsul<T> method*" section below.

AddConsul<T> method

Signature: IOcelotBuilder AddConsul<TServiceBuilder>(this IOcelotBuilder builder)

Overriding the DefaultConsulServiceBuilder behavior involves two steps: creating a new class that inherits from the IConsulServiceBuilder interface, and injecting this new behavior into the DI container using the AddConsul<TServiceBuilder>() helper. However, the fastest and most streamlined approach is to inherit directly from the DefaultConsulServiceBuilder class, as it provides greater flexibility.

First, define a new service-building class:

```
using Ocelot.Logging;
using Ocelot.Discovery.Consul;

public class MyConsulServiceBuilder : DefaultConsulServiceBuilder
{
    public MyConsulServiceBuilder(IHttpContextAccessor contextAccessor, IConsulClientFactory clientFactory, IOcelotLoggerFactory loggerFactory)
        : base(contextAccessor, clientFactory, loggerFactory) { }

    // Use the agent service IP address as the downstream hostname
    protected override string GetDownstreamHost(ServiceEntry entry, Node node)
        => entry.Service.Address;
}
```

Next, inject the new behavior into the DI container, as shown in the Ocelot-Consul setup:

```
builder.Services
    .AddOcelot(builder.Configuration)
    .AddConsul<MyConsulServiceBuilder>();
```

Refer to the repository's acceptance test for further examples.

27.2 Eureka[4]

Package: Ocelot.Discovery.Eureka

Namespace: Ocelot.Discovery.Eureka

This feature supports the Netflix Eureka *service discovery* provider. The primary reason for this is that it is a key product of Steeltoe, which is associated with Pivotal. Now, enough of the background!

The first step is to install the package that provides Eureka support for Ocelot:

```
Install-Package Ocelot.Discovery.Eureka
```

Next, add the following to your Program:

```
using Ocelot.Discovery.Eureka;

builder.Services
    .AddOcelot(builder.Configuration)
    .AddEureka();
```

[4] The *Eureka* feature, requested in issue 262 to add support for the Netflix Eureka *service discovery* provider, was released in version 5.5.4.

Finally, to enable this setup, include the following in your ocelot.json file:

```
"ServiceDiscoveryProvider": {
  "Type": "Eureka"
}
```

Following the guide here, you may also need to add some configurations to appsettings.json. For example, the JSON below informs the Steeltoe / Pivotal services where to locate the service discovery server and whether the service should register with it:

```
"eureka": {
  "client": {
    "serviceUrl": "http://localhost:8761/eureka/",
    "shouldRegisterWithEureka": false,
    "shouldFetchRegistry": true
  }
}
```

If `shouldRegisterWithEureka` is set to `false`, `shouldFetchRegistry` will default to `true`, so you do not need to set it explicitly; however, it has been included here for clarity.

Ocelot will now register all necessary services during startup and, if the JSON above is provided, it will register itself with *Eureka*. One of the services polls *Eureka* every 30 seconds (default) to retrieve the latest service state and persists this information in memory. When Ocelot requests a given service, it retrieves the data from memory, minimizing performance issues.

If not explicitly specified in ocelot.json, Ocelot will use the scheme (`http`, `https`) set in *Eureka*.

Note

Prior to version 25.0, the package was named Ocelot.Provider.Eureka. If you are using version 24.1 or earlier, install the Ocelot.Provider.Eureka package. For version 25.0 and later, the package ID is Ocelot.Discovery.Eureka.

27.3 Service Fabric

If you have services deployed in Azure Service Fabric, you typically use the naming service to access them.

Please refer to the *Service Fabric* chapter for the complete *essential* documentation.

> **Note**: Currently, the `ServiceFabric` *service discovery* provider is tightly coupled with Ocelot core interfaces, making it a part of Ocelot Core and implemented as the `ServiceFabricServiceDiscoveryProvider` class. At present, there is no Ocelot extension package that integrates with the Microsoft.ServiceFabric package or any other relevant package. However, the Ocelot team plans to address this in future development, as we believe Service Fabric is an essential and popular product in the .NET and Azure development world. If anyone in the Ocelot community is a professional Azure developer with extensive Service Fabric experience, please contact our development team directly via GitHub or email.

27.4 Dynamic Routing[5]

[5] The "*Dynamic Routing*" feature was requested in issue 340 (pull request 351) and released in version 7.0.1. Later, the new `DynamicRoutes` *Configuration* section was introduced in pull request 508 and released in version 8.0.4.

The idea is to enable *dynamic routing* mode when using a *service discovery* provider. In this mode, Ocelot uses the first segment of the upstream path to look up the downstream service via the *service discovery* provider.

An example of this would be calling Ocelot with a URL like

- `https://api.ocelot.net/product/products`

Ocelot will take the first segment of the path, which is `product`, and use it as a key to look up the service in *Consul*. If *Consul Provider* returns a service, Ocelot will request it using the host and port provided by Consul, appending the remaining path segments—in this case, `products`—to construct final downstream URL:

- `http://hostfromconsul:portfromconsul/products`

Ocelot will append any query string to the downstream URL as usual.

Warning

To enable *dynamic routing*, the ocelot.json configuration must contain no static routes in the `Routes` collection! Currently, dynamic routes and static routes cannot be mixed. Additionally, you need to specify the details of the *service discovery* provider as outlined above, along with the downstream `http(s)` scheme under `DownstreamScheme`.

Note

In addition to the global `ServiceDiscoveryProvider` section, the *Global Configuration Schema* includes configurable options such as `DownstreamScheme`, `CacheOptions`, `HttpHandlerOptions`, `LoadBalancerOptions`, `QoSOptions`, `RateLimitOptions`, and `Timeout`. These options are applicable to all dynamic routes, globally. Moreover, starting with version 24.1, the *Dynamic Route Schema* also supports these options for overriding global settings.

For instance, when exposing Ocelot publicly over HTTPS while routing to internal services over HTTP, your configuration may resemble the following:

```
{
  "Routes": [], // must be empty to enable dynamic routing!
  "DynamicRoutes": [
    // overriding goes here
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://api.ocelot.net",
    "DownstreamScheme": "http", // default scheme for all internal␣
↪services, no SSL
    "ServiceDiscoveryProvider": {
      "Host": "localhost", // if Consul is hosted on the same machine␣
↪as Ocelot
      "Port": 8500,
      "Type": "Consul",
      "Namespace": "" // not supported for Consul, but supported for␣
↪Kubernetes
    },
    "CacheOptions": {
```

(continues on next page)

(continued from previous page)

```
      "TtlSeconds": 300 // 5 minutes
    },
    "HttpHandlerOptions": {
      "AllowAutoRedirect": false,
      "UseCookieContainer": false,
      "UseTracing": false
    },
    "LoadBalancerOptions": {
      "Type": "LeastConnection"
    },
    "QoSOptions": {
      "MinimumThroughput": 2,
      "BreakDuration": 333,
      "Timeout": 3000 // ms
    },
    "RateLimitOptions": {
      "ClientIdHeader": "Oc-DynamicRouting-Client",
      "QuotaMessage": "No Quota!",
      "StatusCode": 499 // special shared status
    }
  }
}
```

27.4.1 Configuration[6]

Ocelot also allows configuration of a `DynamicRoutes` collection consisting of *Dynamic Route Schema* objects. This enables overriding `RateLimitOptions` for each downstream service, along with other schema-level overrides. Dynamic route options are particularly useful when there are multiple services—such as a '`product`' service and a '`search`' service—and stricter rate limits need to be applied to one over the other. The final configuration looks like:

```
{
  "DynamicRoutes": [
    {
      "ServiceName": "product",
      "ServiceNamespace": "", // not supported for Consul, but␣
↪supported for Kubernetes
      "RateLimitOptions": {
        "Limit": 5,
        "Period": "1s",
        "Wait": "1.5s" // hybrid fixed window
      }
    },
    {
      "ServiceName": "notification",
      "CacheOptions": {
        "TtlSeconds": 0 // disable cache for notifying
      },
      "HttpHandlerOptions": {
        "UseTracing": false // disable tracing
```

(continues on next page)

[6] The *Configuration* feature of *Dynamic Routing* was requested in issue 585, then significantly redeveloped and released in version 24.1.

(continued from previous page)

```
      },
      "LoadBalancerOptions": {
        "Type": "LeastConnection" // switch from RoundRobin to␣
↪LeastConnection
      },
      "RateLimitOptions": {
        "EnableRateLimiting": false // notification service is␣
↪unlimited!
      }
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://api.ocelot.net",
    "DownstreamScheme": "http",
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 8500,
      "Type": "Consul",
      "Namespace": "" // not supported for Consul, but supported for␣
↪Kubernetes
    },
    "CacheOptions": {
      "TtlSeconds": 300 // 5 minutes
    },
    "HttpHandlerOptions": {
      "PooledConnectionLifetimeSeconds": 600, // change the default␣
↪value from 2 minutes to 10 minutes
      "UseTracing": true // enable tracing globally
    },
    "LoadBalancerOptions": {
      "Type": "RoundRobin"
    },
    "RateLimitOptions": {
      "ClientIdHeader": "Oc-DynamicRouting-Client",
      "ClientWhitelist": ["ocelot-client1-preshared-key"],
      "Limit": 5,
      "Period": "10s", // fixed window
      "QuotaMessage": "No Quota!",
      "StatusCode": 499 // special shared status
    }
  }
}
```

This configuration means that when a request is sent to Ocelot at `/product/*`, *dynamic routing* is activated, and Ocelot applies the rate limiting rules defined for the '`product`' service in the `DynamicRoutes` section, as described in the *Rate Limiting* documentation. The '`notification`' service is unlimited because both caching, tracing, and rate limiting are disabled. All other services use the global `RateLimitOptions` along with the other specified options.

Warning

Dynamic route `RateLimitRule` option is deprecated!

The old schema `RateLimitRule` section is deprecated in version 24.1! Use `RateLimitOptions` instead of `RateLimitRule`! Note that `RateLimitRule` will be removed in version 25.0! For backward compatibility in version 24.1, the `RateLimitRule` section takes precedence over the `RateLimitOptions` section.

Note

The `ServiceNamespace` option was introduced in version 24.1 to enable precise overrides for the *Kubernetes (K8s)* providers. If `ServiceNamespace` is left empty or undefined, only **one** dynamic route with the same `ServiceName` may be defined in the `DynamicRoutes` collection.

27.5 Custom Providers

Ocelot also enables you to create a custom *Service Discovery* implementation by implementing the `IServiceDiscoveryProvider` interface, as demonstrated in the following example:

```
public class MyServiceDiscoveryProvider : IServiceDiscoveryProvider
{
    private readonly IServiceProvider _serviceProvider;
    private readonly ServiceProviderConfiguration _config;
    private readonly DownstreamRoute _downstreamRoute;

    public MyServiceDiscoveryProvider(IServiceProvider serviceProvider,␣
↪ServiceProviderConfiguration config, DownstreamRoute downstreamRoute)
    {
        _serviceProvider = serviceProvider;
        _config = config;
        _downstreamRoute = downstreamRoute;
    }

    public Task<List<Service>> GetAsync()
    {
        var services = new List<Service>();
        // ...
        // Add service(s) to the list matching the _downstreamRoute
        return services;
    }
}
```

And set its class name as the provider type in ocelot.json:

```
"GlobalConfiguration": {
  "ServiceDiscoveryProvider": {
    "Type": "MyServiceDiscoveryProvider"
  }
}
```

Finally, in the Program, register a `ServiceDiscoveryFinderDelegate` to initialize and return the provider:

```
ServiceDiscoveryFinderDelegate serviceDiscoveryFinder = (provider, config,␣
↪route)
    => new MyServiceDiscoveryProvider(provider, config, route);
builder.Services
    .AddSingleton(serviceDiscoveryFinder)
    .AddOcelot(builder.Configuration);
```

27.6 Sample

To offer a basic template for a *Custom Providers*, we have created a sample:

Project: samples / ServiceDiscovery
Solution: Ocelot.Samples.ServiceDiscovery.sln

This solution includes the following projects:

- *ApiGateway*
- *DownstreamService*

The solution is ready for deployment. All services are fully configured, with ports and hosts prepared for immediate use (when running in Visual Studio). Complete instructions for running this solution can be found in the README.md file.

27.6.1 DownstreamService

This project provides a single downstream service that can be reused across *ApiGateway* routes. It includes multiple `launchSettings.json` profiles to support your preferred launch and hosting scenarios, such as Visual Studio sessions, Kestrel console hosting, and Docker deployments.

27.6.2 ApiGateway

This project includes a custom *Service Discovery* provider and contains only route(s) to *DownstreamService* services in the ocelot.json file. You are free to add more routes!

The main source code for the custom provider is located in the ServiceDiscovery folder, specifically in the `MyServiceDiscoveryProvider` and `MyServiceDiscoveryProviderFactory` classes. Feel free to design and develop these classes to suit your needs!

Additionally, the cornerstone of this custom provider is the Program code, where you can select from simple or more complex design and implementation options:

```
// Perform initialization from application configuration or hardcode/
↪choose the best option.
bool easyWay = true;
if (easyWay)
{
    // Design #1: Define a custom finder delegate to instantiate a␣
↪custom provider
    // under the default factory (ServiceDiscoveryProviderFactory).
    builder.Services
        .AddSingleton<ServiceDiscoveryFinderDelegate>((serviceProvider,
↪ config, downstreamRoute)
```

(continues on next page)

(continued from previous page)

```
            => new MyServiceDiscoveryProvider(serviceProvider, config,␣
↪downstreamRoute));
}
else
{
    // Design #2: Abstract from the default factory␣
↪(ServiceDiscoveryProviderFactory) and FinderDelegate,
    // and create your own factory by implementing the␣
↪IServiceDiscoveryProviderFactory interface.
    builder.Services
        .RemoveAll<IServiceDiscoveryProviderFactory>()
        .AddSingleton<IServiceDiscoveryProviderFactory,␣
↪MyServiceDiscoveryProviderFactory>();

    // This will not be called but is required for internal validators.
↪ It's also a handy workaround.
    builder.Services
        .AddSingleton<ServiceDiscoveryFinderDelegate>((serviceProvider,
↪ config, downstreamRoute) => null);
}
builder.Services
    .AddOcelot(builder.Configuration);
```

The "easy way" (lite design #1) involves designing only the provider class and specifying the `ServiceDiscoveryFinderDelegate` object for the default `ServiceDiscoveryProviderFactory` in the Ocelot core.

A more complex design #2 involves developing both the provider and provider factory classes. Once this is done, you need to add the `IServiceDiscoveryProviderFactory` interface to the DI container and remove the default `ServiceDiscoveryProviderFactory` class. Note that in this case, the Ocelot core will not use the `ServiceDiscoveryProviderFactory` by default. Additionally, you do not need to specify `"Type": "MyServiceDiscoveryProvider"` in the `ServiceDiscoveryProvider` global options. However, you can retain this `Type` option to maintain compatibility between both designs.

CHAPTER TWENTYEIGHT

SERVICE FABRIC

[1] Feature of: *Service Discovery*

If you have services deployed in Azure Service Fabric you will normally use the naming service to access them.

This feature allows to set up a route that will work in Service Fabric.

28.1 Configuration

The most important thing is the `ServiceName`, which is composed of the Service Fabric application name followed by the specific service name. Additionally, the `ServiceDiscoveryProvider` needs to be configured in `GlobalConfiguration`. The example below demonstrates a typical configuration. It assumes that *Service Fabric* is running on `localhost` and that the naming service is using port `19081`.

The example below is taken from the *Sample*, so please check it if this doesn't make sense!

```
{
  "Routes": [
    {
      "DownstreamScheme": "http",
      "DownstreamPathTemplate": "/api/values",
      "UpstreamPathTemplate": "/EquipmentInterfaces",
      "UpstreamHttpMethod": [ "Get" ],
      "ServiceName": "OcelotServiceApplication/OcelotApplicationService"
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://ocelot.net",
    "RequestIdKey": "Oc-RequestId",
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 19081,
      "Type": "ServiceFabric"
    }
  }
}
```

If you are using stateless or guest exe services, Ocelot can proxy through the naming service without requiring additional configuration. However, if you are using stateful or actor services, you must include the `PartitionKind` and `PartitionKey` query string values in the client request, e.g.,

[1] Historically, the "*Service Fabric*" feature is one of Ocelot's earliest and foundational features, first requested in issue 238. It was initially released in version 3.1.9.

GET `http://ocelot.com/EquipmentInterfaces?PartitionKind=xxx&PartitionKey=xxx`

There is no way for Ocelot to determine these values automatically.

28.2 Placeholders[2]

In Ocelot, *placeholders* for variables can be inserted into the `UpstreamPathTemplate` and `ServiceName` using the format `{something}`.

> **Note**: The *placeholder* variable must exist in both the `DownstreamPathTemplate` (or `ServiceName`) and the `UpstreamPathTemplate`. Specifically, the `UpstreamPathTemplate` must include all *placeholders* found in the `DownstreamPathTemplate` and `ServiceName`. Failure to meet this requirement will prevent Ocelot from starting due to validation errors, which are logged.

Once the validation stage is completed, Ocelot replaces the placeholder values in the `UpstreamPathTemplate` with those from the `DownstreamPathTemplate` and/or `ServiceName` for each processed request. Thus, the *Service Fabric Placeholders 2* feature operates similarly to the original routing *Placeholders* feature but includes the `ServiceName` property in its processing.

Here is an example of the `version` variable in the *Service Fabric* service name.

Given the following ocelot.json:

```
{
  "Routes": [
    {
      "UpstreamPathTemplate": "/api/{version}/{endpoint}",
      "DownstreamPathTemplate": "/{endpoint}",
      "ServiceName": "Service_{version}/Api",
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://ocelot.com",
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 19081,
      "Type": "ServiceFabric"
    }
  }
}
```

When you make Ocelot request:

- GET `https://ocelot.com/api/1.0/products`

The *Service Fabric* request will be:

- GET `http://localhost:19081/Service_1.0/Api/products`

[2] The "*Placeholders*" feature was requested in issue 721 and implemented by pull request 722 as part of version 13.0.0.

28.3 Sample

In order to introduce the *Service Fabric* feature, we have prepared a sample:

Project: samples / ServiceFabric
Solution: Ocelot.Samples.ServiceFabric.sln

This solution includes the following projects:

- `Ocelot.Samples.ServiceFabric.ApiGateway.csproj`
- `Ocelot.Samples.ServiceFabric.DownstreamService.csproj`

Complete instructions for running this solution can be found in the README.md file.

> **Note**
>
> Please consider this solution as a demonstration of integration; it is outdated as of 2025. Therefore, this solution is a draft and requires further development for practical usage and deployment in the Azure cloud. Additionally, refer to the team's notes in the *Service Fabric* section!

CHAPTER TWENTYNINE

TRACING

Feature of: *Logging*

- .NET logging and tracing | .NET | Microsoft Learn
- .NET distributed tracing | .NET | Microsoft Learn

This chapter explains how to perform distributed tracing using Ocelot.

29.1 OpenTracing

Package: Ocelot.Tracing.OpenTracing
Namespace: `Ocelot.Tracing.OpenTracing`

Ocelot provides tracing functionality through the excellent project from opentracing-csharp repository. The code for Ocelot integration can be found in this Ocelot project.

The example below uses the C# Client for Jaeger to provide the tracer used in Ocelot. To add OpenTracing services, you must call the `AddOpenTracing()` extension method on the `OcelotBuilder` returned by `AddOcelot()`[1], as shown below:

```
builder.Services
    .AddSingleton(serviceProvider =>
    {
        var loggerFactory = serviceProvider.GetService<ILoggerFactory>();
        var config = new Jaeger.Configuration(builder.Environment.
↪ApplicationName, loggerFactory);
        var tracer = config.GetTracer();
        GlobalTracer.Register(tracer);
        return tracer;
    })
    .AddOcelot(builder.Configuration)
    .AddOpenTracing();
```

Then, in your ocelot.json, add the following to the route you want to trace:

```
"HttpHandlerOptions": {
  "UseTracing": true
}
```

[1] The *AddOcelot method* adds default ASP.NET services to the DI container. You can call another extended *AddOcelotUsingBuilder method* while configuring services to develop your own *Custom Builder*. See more instructions in the "*AddOcelotUsingBuilder method*" section of the *Dependency Injection* feature.

Ocelot will now send tracing information to Jaeger whenever this route is called.

> **Note 1**: A clean yet functional sample can be found here: Ocelot.Samples.OpenTracing.
>
> **Note 2**: The OpenTracing project was archived on January 31, 2022 (see the article). The Ocelot team is planning to decide on a migration to OpenTelemetry, which is highly desirable.

29.2 Butterfly

Package: Ocelot.Tracing.Butterfly
Namespace: `Ocelot.Tracing.Butterfly`

Ocelot provides tracing functionality through the excellent Butterfly project. The code for the Ocelot integration can be found in this Ocelot project. To use the tracing functionality, please refer to the Butterfly documentation.

In Ocelot, you need to add the NuGet package if you wish to trace a route:

```
Install-Package Ocelot.Tracing.Butterfly
```

In your Program, to add Butterfly services, you must call the `AddButterfly()` extension method on the `OcelotBuilder` returned by `AddOcelot()`, as shown below:

```
using Ocelot.Tracing.Butterfly;

builder.Services
    .AddOcelot(builder.Configuration)
    .AddButterfly(options =>
    {
        // This is the URL that the Butterfly collector server is running on...
        options.CollectorUrl = "http://localhost:9618";
        options.Service = "Ocelot";
    });
```

Then, in your ocelot.json, add the following to the route you want to trace:

```
"HttpHandlerOptions": {
  "UseTracing": true
}
```

Ocelot will now send tracing information to Butterfly whenever this route is called.

> **Note**: The Butterfly project has not been supported for more than seven years, as of 2025. The latest release of the Butterfly.Client package (version 0.0.8) was made on February 22, 2018. The Ocelot team is planning to discontinue the Ocelot.Tracing.Butterfly package, which is scheduled to happen after the release of Ocelot version 24.1.

CHAPTER THIRTY

WEBSOCKETS

- Ocelot Middleware: WebSocketsProxyMiddleware
- RFC 6455 Specification: The WebSocket Protocol by Internet Engineering Task Force (IETF) organization
- JavaScript Living Standard: WebSockets Standard by WHATWG organization
- Mozilla Developer Network: The WebSocket API (WebSockets)

Ocelot supports proxying WebSockets[1] with some extra bits.

30.1 Configuration

To enable *WebSockets* proxying with Ocelot, you need to do the following in your Program:

```
var app = builder.Build();
app.UseWebSockets(); // required for Ocelot 24.x and earlier; called␣
↪automatically since version 25.0
await app.UseOcelot();
await app.RunAsync();
```

Then, in your ocelot.json, add the following to proxy a route using *WebSockets*:

```
{
  "UpstreamPathTemplate": "/",
  "DownstreamPathTemplate": "/ws",
  "DownstreamScheme": "ws",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 5001 }
  ]
}
```

With this configuration, Ocelot will match any *WebSockets* traffic that comes in on `/` and proxy it to `localhost:5001/ws`. For clarity, Ocelot will receive messages from the upstream client, proxy them to the downstream service, receive messages from the downstream service, and then proxy them back to the upstream client.

[1] The *Websockets* functionality was requested in issue 212 and introduced in version 5.3.0.

30.2 Handy Links

- WHATWG: WebSockets Standard
- Mozilla Developer Network: The WebSocket API (WebSockets)
- Microsoft Learn: WebSockets support in ASP.NET Core
- Microsoft Learn: WebSockets support in .NET

30.3 SignalR[2]

Welcome to Real-time ASP.NET with SignalR

Ocelot supports proxying *SignalR*. To enable this with Ocelot, you need to do the following:

First, install the SignalR Client NuGet package:

```
Install-Package Microsoft.AspNetCore.SignalR.Client
```

Note: SignalR is part of the ASP.NET Core and can be referenced as follows:

```
<ItemGroup>
  <FrameworkReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

More information on framework compatibility can be found in the instructions: Use ASP.NET Core APIs in a class library.

Second, you need to configure your application to use *SignalR*. A complete reference can be found here: ASP.NET Core SignalR configuration.

```
builder.Services.AddOcelot(builder.Configuration);
builder.Services.AddSignalR();
```

Note: Make sure to pay attention to the transport-level configuration for *WebSockets*. Ensure that allowed transports are properly configured to enable *WebSockets* connections: ASP.NET Core SignalR configuration.

Next, include the following in your ocelot.json file to proxy a route using *SignalR*. Note that standard Ocelot routing rules apply; the key aspect is that the scheme is set to `ws` (*WebSockets*).

```
{
  "UpstreamPathTemplate": "/gateway/{catchAll}",
  "DownstreamPathTemplate": "/{catchAll}",
  "DownstreamScheme": "ws",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 5001 }
  ]
}
```

[2] The *SignalR* functionality was requested in issue 344 and published in version 8.0.7.

30.4 WebSocket Secure

If you define a route with the *secured WebSockets* protocol, use the `wss` scheme:

```
"DownstreamScheme": "wss",
```

Keep in mind that you can use WebSocket SSL for both *SignalR* and *Websockets*.

Note: To understand `wss` scheme, browse to this documentation:

- IETF | The WebSocket Protocol: WebSocket URIs
- Microsoft Learn: Secure your connection with TLS/SSL
- Microsoft Learn: Search for "secure websocket"

If you want to ignore SSL warnings (errors)[3], configure your route as follows:

```
"DownstreamScheme": "wss",
"DangerousAcceptAnyServerCertificateValidator": true,
```

However, we strongly advise against this! Refer to the official notes regarding *SSL Errors* in the *Configuration* documentation. There, you can also explore best practices tailored for your environments.

30.5 Supported

1. *Routing*
2. *Load Balancer*
3. *Service Discovery*

This means you can configure your downstream services to run *WebSockets* and either:

- Include multiple `DownstreamHostAndPorts` in your route configuration.
- Connect your route to a *Service Discovery* provider. This allows you to load balance requests, which we think is pretty cool!

30.6 Not Supported

Unfortunately, many Ocelot features are not specific to *WebSockets*, such as header handling and HTTP client functionalities. Below is a list of features that will not work:

1. *Tracing*
2. *Logging Request ID*
3. *Aggregation*
4. *Rate Limiting*
5. *Quality of Service*
6. *Middleware Injection* (except the *OcelotPipelineConfiguration Class* `WebSocketsMiddlewareType` and `WebSocketsMiddleware` properties, see *Sample*)
7. *Headers Transformation*

[3] The "*WebSocket Secure*" feature includes a `wss` scheme fake validator, which was introduced in pull request 1377 as part of issues 1375, 1237, and others. This "life hack" for self-signed SSL certificates is available starting from version 20.0. However, it will be either removed or reworked in future releases. For further details, refer to the *SSL Errors* section.

8. *Delegating Handlers*
9. *Claims Transformation*
10. *Caching*
11. *Authentication*[4]
12. *Authorization*

We cannot be entirely sure how this feature will behave once it is widely used. Therefore, thorough testing is strongly recommended!

30.7 Sample[5]

> **Project**: samples / WebSocket
> **Solution**: Ocelot.Samples.slnx

The `Ocelot.Samples.WebSocket.csproj` sample project demonstrates how to proxy *WebSocket* connections with a customized buffer size by subclassing WebSocketsProxyMiddleware and registering it via `OcelotPipelineConfiguration`:

```
public class MyWebSocketsProxyMiddleware : WebSocketsProxyMiddleware
{
    protected override int BufferSize => 65536; // 64 KB for high-throughput␣
↪streams (e.g. HTTP.sys video streaming)

    public MyWebSocketsProxyMiddleware(RequestDelegate next,␣
↪IOcelotLoggerFactory logging, IWebSocketsFactory factory)
        : base(next, logging, factory) { }
}
```

The custom middleware type is then registered through `WebSocketsMiddlewareType` option of the *OcelotPipelineConfiguration Class*:

```
var wsPipeline = new OcelotPipelineConfiguration
{
    WebSocketsMiddlewareType = typeof(MyWebSocketsProxyMiddleware),
};
await app.UseOcelot(wsPipeline);
```

Alternatively, the same can be achieved with a delegate via `WebSocketsMiddleware`:

```
var wsPipeline = new OcelotPipelineConfiguration
{
    WebSocketsMiddleware = (context, next) =>
    {
        Task Next(HttpContext ctx) => next();
        var loggerFactory = context.RequestServices.GetRequiredService
↪<IOcelotLoggerFactory>();
        var wsFactory = context.RequestServices.GetRequiredService
↪<IWebSocketsFactory>();
        var middleware = new MyWebSocketsProxyMiddleware(Next, loggerFactory,␣
↪wsFactory);
```

(continues on next page)

[4] If requested, we might explore options for implementing basic authentication.
[5] The *Sample* was introduced for issue 2386 and implemented in pull request 2387, as part of version 25.0.

(continued from previous page)

```
        return middleware.Invoke(context);
    },
};
await app.UseOcelot(wsPipeline);
```

When `WebSocketsMiddlewareType` is set, it takes **priority** over `WebSocketsMiddleware` and the delegate is ignored. For the full reference, see the *OcelotPipelineConfiguration Class* section in *Middleware Injection* chapter.

> **Note**
>
> Starting from Ocelot version 25.0, `app.UseWebSockets()` is called internally during Ocelot pipeline setup. You no longer need to call it explicitly before `await app.UseOcelot()`.

30.8 Roadmap

WebSockets and *SignalR* are being actively developed by the .NET community. It is important to stay updated with trends and regularly check for new releases in the official documentation:

- WebSockets docs
- SignalR docs

As a team, we are unable to provide direct development advice. However, feel free to ask questions or explore coding recipes in Discussions of the repository. Additionally, we welcome any bug reports, enhancement suggestions, or proposals related to this feature.

> **Note**
>
> The Ocelot team considers the current implementation of the *WebSockets* feature to be obsolete, as it is based on the WebSocketsProxyMiddleware class. *WebSockets* are a part of the ASP.NET Core framework, which includes the native WebSocketMiddleware class. We have a strong intention to either migrate or redesign this feature. For more details, see issue 1707.

CHAPTER

THIRTYONE

BUILDING

This document summarises the build and release process for the Ocelot project. The build scripts are written using Cake (C# Make), with relevant build tasks defined in the 'build.cake' file located in the root of the Ocelot project. The scripts are designed to be run by developers locally in a Bash terminal (on any OS), in Command Prompt (CMD) or PowerShell consoles (on Windows OS), or by a CI/CD server (currently GitHub Actions), with minimal logic defined in the build server itself.

The final goal of the build process is to create `Ocelot.*` NuGet packages (.nupkg files) for redistribution via the NuGet repository or manually. The build process consists of several steps: (1) compilation, (2) testing, (3) creating and publishing NuGet packages, and (4) making an official GitHub release. The build process requires pre-installed .NET SDKs on the build machine (host) for all target framework monikers: TFMs are `net8.0` and `net9.0` currently. In general, the build process is the same across all environments and tools, with a few differences described below.

31.1 In IDE

In an IDE, a DevOps engineer can build the project in Visual Studio IDE or another IDE in Release configuration mode, but the latest .NET 8/9 SDKs must be pre-installed on the local machine. However, this approach is not practical because the generated '.nupkg' files must be uploaded to NuGet manually, and the GitHub release must also be created manually. A better approach is to utilize the 'build.cake' script *In terminal*, which covers all building scenarios.

31.2 In terminal

Folder: ./

These are local machine or remote server building scenarios using build scripts, aka 'build.cake'. In these scenarios, the following two commands should be run in a terminal from the project's root folder:

```
dotnet tool restore && dotnet cake  # In Bash terminal
dotnet tool restore; dotnet cake  # In PowerShell terminal
```

Note: The default target task ("Default") is "Build", and output files will be stored in the `./artifacts` directory.

To run a desired target task, you need to specify its *name*:

```
dotnet tool restore && dotnet cake --target=name  # In Bash terminal
dotnet tool restore; dotnet cake --target=name  # In PowerShell terminal
```

For example,

- ```
  dotnet cake --target=Build
  ```

  It runs a local build, performing compilation and testing only.

- ```
  dotnet cake --target=Version
  ```

  It checks the next version to be tagged in the Git repository during the next release, without performing compilation or testing tasks.

- ```
  dotnet cake --target=CreateReleaseNotes
  ```

  It generates Release Notes artifacts in the `/artifacts/Packages` folder using the `ReleaseNotes.md` template file.

- ```
  dotnet cake --target=Release
  ```

  It creates a release, consisting of the following steps: compilation, testing, generating release notes, creating .nupkg files, publishing NuGet packages, and finally, making a GitHub release.

  Note 1: The building tools for the `dotnet tool restore` command are configured in the dotnet-tools.json file.

  Note 2: Some targets (build tasks) require appropriate environment variables to be defined directly in the terminal session (aka secret tokens).

31.3 With Docker

Folder: ./docker

The best way to replicate the CI/CD process and build Ocelot locally is by using the Dockerfile.build file, which can be found in the 'docker' folder in the Ocelot root directory. For example, use the following command:

```
docker build --platform linux/amd64 -f ./docker/Dockerfile.build .
```

You may need to adjust the platform flag depending on your system.

Note: This approach is somewhat excessive, but it will work if you are a masterful Docker user. The Ocelot team has not followed this approach since version 24.0, favoring *With CI/CD*-based builds and occasionally building *In terminal* instead.

31.4 With CI/CD

Folder: ./.github/workflows

Provider: GitHub Actions

Workflows: PR, Develop, Release

Dashboard: Workflow runs (Actions tab)

The Ocelot project utilizes GitHub Actions as a CI/CD provider, offering seamless integrations with the GitHub ecosystem and APIs. Starting from version 24.0, all pull requests, development commits, and releases are built using GitHub Actions workflows. There are three workflows: one for pull requests (PR), one for the `develop` branch (Develop), and one for the `main` branch (Release).

Note: Each workflow has a dedicated status badge in the Ocelot README: the Release button and the Develop button, with the PR status being published directly in a pull request under the "Checks" tab.

The PR workflow will track code coverage using Coveralls. After opening a pull request or submitting a new commit to a pull request, Coveralls will publish a short message with the current code coverage once the top commit is built. Considering that Coveralls retains the entire history but does not fail the build if coverage falls below the threshold, all workflows have a built-in 80% threshold, applied internally within the `build-cake` job, particularly during the "Cake Build" step-action. If the code coverage of a newly opened pull request drops below the 80% threshold, the 'build-cake' job will fail, logging an appropriate message in the "Cake Build" step.

> **Note 1**: There are special code coverage badges in Ocelot README: the Develop button and the Release button.
>
> **Note 2**: The current code coverage of the Ocelot project is around 85-86%. The coverage threshold is subject to change in upcoming releases. All Coveralls builds can be viewed by navigating to the ThreeMammals/Ocelot project on Coveralls.io.

31.5 Documentation

Folder: ./docs

Dashboard: Ocelot app project

Documentation building is configured using the '.readthedocs.yaml' integration file, which allows builds to run separately via the Read the Docs publisher. All build artifacts and document sources are located in the 'docs' folder. More details on the documentation build process can be found in the README.

> **Note 1**: Documentation builds have a dedicated status badges in Ocelot README: the Develop button and the Release button.
>
> **Note**: Documentation can be easily built locally in a terminal from the 'docs' folder by running the `make.sh` or `make.bat` scripts. The resulting documentation build files will be located in the `./docs/_build` folder, with the HTML documentation specifically written to the `./docs/_build/html` folder.

31.6 Testing

The tests should run and function correctly as part of the *building* process using the `dotnet test` command. You can also run them in Visual Studio IDE within the Test Explorer window. Depending on your build scenario, Ocelot *testing* can be performed as follows.

In IDE: Simply run tests via the Test Explorer window of Visual Studio IDE.

In terminal: There are two main approaches:

1. Run the `dotnet test` command to perform all tests (unit, integration, and acceptance):

```
dotnet test -f net10.0 ./Ocelot.slnx
```

   Or run tests separately per project:

```
dotnet test -f net10.0 ./unit/Ocelot.UnitTests.csproj  # Unit tests only
dotnet test -f net10.0 ./acceptance/Ocelot.Acceptance.csproj  # Acceptance tests only
```

2. Run `dotnet cake` command: `dotnet cake --target=Tests` to perform all tests (unit, integration and acceptance). Or run tests separately per *testing* project:

```
dotnet cake --target=UnitTests # unit tests only
dotnet cake --target=AcceptanceTests # acceptance tests only
```

With Docker: This approach is not recommended. Instead, perform automated testing *With CI/CD* or opt for *In terminal*-based testing, which is a more advanced method.

With CI/CD: In GitHub Actions workflows, the *testing* process consists of separate testing steps, organized per job:

- In the 'build' job: There are 'Unit Tests', 'Integration Tests', and 'Acceptance Tests' steps.
- In the 'build-cake' job: There is a 'Cake Build' step responsible for performing tests internally.

31.7 SSL certificate

To create a certificate for *Testing*, you can use OpenSSL:

- Install the openssl package (if you are using Windows, download the binaries here).
- Generate a private key:

```
openssl genrsa 2048 > private.pem
```

- Generate a self-signed certificate:

```
openssl req -x509 -days 1000 -new -key private.pem -out public.pem
```

- If needed, create a PFX file:

```
openssl pkcs12 -export -in public.pem -inkey private.pem -out mycert.pfx
```

CHAPTER

THIRTYTWO

DEVELOPMENT PROCESS

- The *development process* is optimized when using Gitflow branching, as detailed here: Gitflow Workflow. It's important to note that the Ocelot team does not utilize GitHub Flow, which, despite being quicker, does not align with the efficiency required for Ocelot's delivery.
- Contributors are free to manage their pull requests and feature branches as they see fit to contribute to the 'develop' branch.
- Maintainers have the autonomy to handle pull requests and merges. Any merges to the 'main' branch will trigger the release of packages to GitHub and NuGet.
- In conclusion, while users should adhere to the guidelines in *Development Process*, maintainers should follow the procedures outlined in *Release Process*.

32.1 Stages

Ocelot project follows this *development process* to integrate work into a merged commit in the 'develop' branch:

1. Users either create a new issue or select an existing issue(s) on GitHub. Issues can also be generated from discussion topics when necessary and agreed upon.
2. Users should create a fork and branch off of it (unless they are a core team member, in which case they can branch directly from the main/head/upstream repository), e.g., `feature/xxx`, `bug/xxx`, etc. The "xxx" can be the issue number or a brief description.
3. Once contributors are satisfied with their work, they can submit a pull request against the develop branch on GitHub with their changes.
4. The Ocelot team will review the pull request and, if satisfactory, merge it; otherwise, they will provide feedback for the contributor to address. To expedite pull request approval, contributors should consider:
    - Ensuring all changes are covered by unit and acceptance tests.
    - Ensuring that the code coverage percentage from unit tests does not decrease; thus, the Coveralls check reports a green status.
    - Updating any documentation affected by the changes, with a required review of the appropriate feature document.
    - Verifying that the feature is necessary and does not duplicate existing Ocelot features.
5. A pull request must meet the following criteria before merging:
    - All new code must be covered by unit tests.
    - There must be at least one acceptance test for the happy path of the new code.

   - Tests must pass locally, in Visual Studio Test Explorer or in terminal after performing `dotnet test` command.
   - The build must have a green status on repository Actions as passed *checks* of the pull request (aka Checks tab).
   - The build's performance must not be significantly degraded on repository Actions page for PR workflow.
   - The main Ocelot package must not introduce any non-Microsoft dependencies.

6. Once the pull request is merged with *"Squash and Merge"* option into the develop branch, the `Ocelot.*` NuGet packages will not be updated until a release is crafted. The concluding step involves returning to GitHub to close any resolved issue(s).

32.2 Notes

Note 1: The issue(s) linked to the pull request within the *Development* settings (on the right sidebar of the pull request settings) will automatically close upon merging. It is crucial for developers to utilize the *"Link an issue from this repository"* feature in the *Development* settings. An alternative way to link issue(s) is by specifying them in the pull request description, where the developer lists the linked issue(s) that need to be closed. For example:

```
## Fixes #1222
- #1222

## Closes #1333
- #1333

## Proposed Changes
- change 1
- change 2
```

This Markdown should automatically link the desired bug/issue in the open status. For bugs, the developer needs to write "Fixes #xxxx", and for features, "Closes #xxxx".

Note 2: All pull request builds are conducted using GitHub Actions, but developers have the freedom to build Ocelot as needed. Details can be found in the *Building* chapter. Additionally, for a deeper understanding of the current Ocelot CI/CD environment and a clearer view of the CI/CD build process, refer to the "Building *With CI/CD*" section.

Note 3: Should you encounter any confusion or obstacles, do not hesitate to reach out to the members of the 'Ocelot Team' or the repository maintainers.

32.3 Best Practices

- Refer to the Ocelot Actions dashboard on GitHub to verify the latest build statuses for the three current workflows. It is recommended to monitor the build status of each workflow on the Actions dashboard or directly in the Checks tab of a pull request. If a build fails, initiate a new build by pushing a new commit, or consult with online maintainers or code reviewers to ensure the current pull request build is successful.
- Request a code review after reaching the "Development Complete" stage, and address all feedback issues. Code is deemed complete when robust code, relevant unit and acceptance tests, and documentation updates are in place.

- Set up your development environment on Windows OS using Visual Studio IDE. While development in Linux OS with alternative IDEs is possible, it is not recommended. For more details, refer to the *Dev Fun* section.
- Remain online after submitting a pull request/issue to ensure maintainers can reach you promptly. Note that if you are offline for extended periods, such as days, weeks, or months, maintainers may deprioritize your work. A strong contribution ethic implies constant online presence and proactivity.

32.4 Dev Fun

This section is part of the *Best Practices* and is written to be more amusing D)

32.4.1 EOL Gotchas

> *Also known as, "Line-Endings problem"*
>
> Since the project's inception in 2016, this issue has been persistent. Indeed, some lines end with the `LF` character, typical of the Linux OS. Many of our contributors work on Linux and use IDEs like Visual Studio Code, JetBrains .NET Rider, which defaults to the `LF` as the newline character. As a result, we have numerous files with inconsistent or mixed EOL characters.

This problem stems from the well-known dilemma of End-of-Line (EOL) characters in cross-OS development. For the Windows OS, the EOL character is `CRLF`, while for Linux, it is `LF`. Modern IDEs and Git repositories have their own strategies for detecting inconsistencies of mixed EOLs in source files. However, the GitHub "Files Changed" tool unfortunately registers a line change in two scenarios: `CRLF` to `LF` and `LF` to `CRLF`, even when there's no actual code change! Reviewing such pull requests with fictitious ("fake") changes is always challenging because the reviewer's focus should be on actual code changes.

> Please note, if a pull request is filled with "fake" changes in *"Files Changed"*, the code reviewer has the right to not provide a code review, mark the PR as a draft, or even close it.

Our standard practice is to maintain end-of-line characters as they are. Moreover, we utilize Visual Studio's unique `.editorconfig` IDE analyzer settings for EOL to avoid issues with line endings. These settings are specific to Visual Studio, hence we recommend rebasing a feature branch onto develop using Visual Studio exclusively.

> Special EOL settings can be specified in the `.gitattributes` file of the git repository, although we do not currently manage this.

Our current recommendations for addressing the end-of-line (EOL) issue are as follows:

- Ideally, resolve merge conflicts by prioritizing the changes in the develop branch, then manually incorporate your changes in the merge tool dialog. It appears that changes from the feature branch are being included, even if they are minor. Conflicts should be addressed by manually applying your changes to the develop branch with a merge tool.
- If changes from the feature branch are given priority (despite being minor), the merge tool will document them and apply `CRLF` end-of-line characters according to the rules specified in `.editorconfig`. This is the source of the issue.
- Renaming a method in an IDE, such as Visual Studio, or using another auto-refactoring command, causes Visual Studio to apply the command using the default styling rules in `.editorconfig`, which includes CRLF settings. Thus, applying auto-refactoring commands inadvertently alters the EOL characters, leading to "fake" changes in pull requests. Note that Visual Studio analyzers (IDE, StyleCop, etc.) may also recommend auto-refactoring, which could be applied implicitly. To preserve the original EOL characters, manual code editing is necessary. Therefore, "fake" changes result from auto-refactoring commands in IDEs like Visual Studio, Visual Code, Rider, etc.

- **Our final recommendation** is to boot into Windows, use Visual Studio Community (which is free), refrain from using auto-refactoring commands, and ensure that EOLs remain unchanged. If your OS differs, you **must** ensure that the appropriate settings are provided in the `.gitattributes` file to always commit files with `CRLF` EOL characters.

CHAPTER

THIRTYTHREE

RELEASE PROCESS

- The *release process* is optimized when using Gitflow branching, as detailed here: Gitflow Workflow. It's important to note that the Ocelot team does not utilize GitHub Flow, which, despite being quicker, does not align with the efficiency required for Ocelot's delivery.
- Contributors are free to manage their pull requests and feature branches as they see fit to contribute to the 'develop' branch.
- Maintainers have the autonomy to handle pull requests and merges. Any merges to the 'main' branch will trigger the release of packages to GitHub and NuGet.
- In conclusion, while users should adhere to the guidelines in *Development Process*, maintainers should follow the procedures outlined in *Release Process*.

33.1 Stages

Ocelot project follows this *release process* to incorporate work into NuGet packages:

1. As a code reviewers, maintainers review pull requests and, if satisfactory, merge them; otherwise, they provide feedback for the contributor to address. Contributors are supported through continuous Pair Programming sessions, which include multiple code reviews, resolving code review issues, and problem-solving.
2. As a release engineers, maintainers must adhere to Semantic Versioning (SemVer) supported by GitVersion. For breaking changes, maintainers should use the correct commit message (containing *"+semver: breaking|major|minor|patch"*) to ensure GitVersion applies the appropriate SemVer tags. Manual tagging of the Ocelot repository should be avoided to prevent disruptions.
3. Once a pull request is merged into the 'develop' branch, the Ocelot NuGet packages remain unchanged until a release is initiated. When sufficient work warrants a new release, the 'develop' branch is merged into 'main' as a `release/X.Y` branch, triggering the Release workflow that builds the code, assigns versions, and pushes artifacts to GitHub and packages to NuGet.
4. The release engineer, who holds the integration tokens in GitHub Environments, automates each release build using the primary build script, 'build.cake'. Automated or manual *Building* can be performed *In terminal* or *With CI/CD*. The release engineer is also responsible for DevOps within the ThreeMammals organization, across all (sub)repositories, supporting the primary build script, and scripting for other repositories.
5. The release engineer drafts the `ReleaseNotes.md` template file, informing the community about key aspects of the release, including new or updated features, bug fixes, documentation updates, breaking changes, contributor acknowledgments, version upgrade guidelines, and more.
6. The final stage of the *release process* involves returning to GitHub to close the current milestone, ensuring that:

- All issues within the milestone are closed; any remaining work from open issues should be transferred to the next milestone.
- All pull requests associated with the milestone are either closed or reassigned to the upcoming release milestone.
- Release Notes are published on GitHub releases, with an additional review of the text.
- The published release is designated as the latest, provided the corresponding Ocelot NuGet packages have been successfully uploaded to the ThreeMammals account.

7. Optional support for the major version `X.Y.0` should be available in cases such as Microsoft official patches and critical Ocelot defects of that major version. Maintainers should release patched versions `X.Y.1-z` as hot-fix patch versions.

33.2 Notes

Note 1: All NuGet package builds and releases are conducted through the GitHub Actions CI/CD provider. For details, refer to the dedicated Actions dashboard, which should be used to monitor the current status of three workflows.

Note 2: Currently, only Tom Pallister, Raman Maksimchuk, the owners—along with the Ocelot Team maintainers—have the authority to merge releases into the 'main' branch of the Ocelot repository. This policy ensures that final *Quality Gates* are in place. The maintainers' primary focus during the final merge is to identify any security issues, as outlined in Stage 7 of the process.

33.3 Quality Gates

Gate 1: Static code analysis. The Ocelot repository includes the following integrated style analyzers:

- In-built IDE (.NET SDK): The code analysis rule set is defined in the 'codeanalysis.ruleset' file, with configuration instructions available here. For comprehensive documentation, refer to the following article:
  - Microsoft Learn: Overview of .NET source code analysis
- StyleCop.Analyzers: The package is somewhat outdated with slow support, but Ocelot projects still reference it because it has remained functional since 2015/16 as an older style analyzer. The Ocelot team plans to replace this library with a more advanced tool in upcoming releases.