
Ocelot

Release 23.2

Tom Pallister, Raman Maksimchuk and Ocelot Core team at Three

Apr 05, 2024

INTRODUCTION

1	Release Notes	3
1.1	What's new?	3
1.2	Focus On	3

Thanks for taking a look at the Ocelot documentation! Please use the left hand navigation to get around. The team would suggest taking a look at the **Introduction** chapter first.

All **Features** are arranged in alphabetical order. The main features are *Configuration* and *Routing*.

We **do** follow development process which is described in *Release Process*.

RELEASE NOTES

Release Tag: 23.2.0

Release Codename: Lunar Eclipse

1.1 What's new?

- *Configuration*: A brand new *Merging files to memory 2* by @ebjornset as a part of the *Merging Configuration Files* feature.

The AddOcelot method merges the **ocelot.*.json** files into a single **ocelot.json** file as the primary configuration file, which is written back to disk and then added to the IConfigurationBuilder for the well-known IConfiguration. You can now call another AddOcelot method that adds the merged JSON directly from memory to the IConfigurationBuilder, using AddJsonStream instead.

See more details in *Configuration Overview of Dependency Injection*.

- *Service Fabric*: Published old undocumented *Placeholders in Service Name 1* feature of *Service Fabric* service discovery provider.

This feature by @FelixBoers is available starting from version 13.0.0.

- *Quality of Service*: A brand new Polly v8 pipelines *Extensibility 3* feature by @RaynaldM

1.2 Focus On

1.2.1 Updates of the features

- *Configuration*: New *Merging files to memory 2* feature by @ebjornset
- *Dependency Injection*: Added new overloaded *AddOcelot methods* by @ebjornset
- *Quality of Service*: Support of new Polly v8 syntax and new *Extensibility 3* feature by @RaynaldM

1.2.2 Ocelot extra packages

- `Ocelot.Provider.Polly`: Support of new Polly v8 syntax.

Polly 8.0+ versions introduced the concept of [resilience pipelines](#).

All `AddPolly` extensions have been automatically migrated from **v7** to **v8**.

Please note that older **v7** extensions are marked with the `[Obsolete]` attribute and renamed using the **V7** suffix. And the old **v7** implementation has been moved to the **v7 namespace**.

See more details in *Polly v7 vs v8* section of *Quality of Service* chapter.

1.2.3 Stabilization aka bug fixing

- 683 by PR 1927. Thanks to [@AlyHKafoury](#)!

[New rules](#) have been added to Ocelot's configuration validation logic to find duplicate placeholders in path templates.

See more in the `FileConfigurationFluentValidator` class.

- 1518 hotfix by PR 1986. Thanks to [@ArwynFr](#)!

Using the default `IServiceCollection` [DI extensions](#) to register Ocelot services resulted in the `ServiceCollection` provider being forced to be created by calling `BuildServiceProvider()`.

This resulted in problems with dependency injection libraries, or worse, causing the Ocelot app to crash!

See more in the `ServiceCollectionExtensions` class.

- See [all bugs](#) of the February'24 milestone

1.2.4 Updated Documentation

- *Configuration*
- *Dependency Injection*
- *Quality of Service*
- *Service Fabric*

Big Picture

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

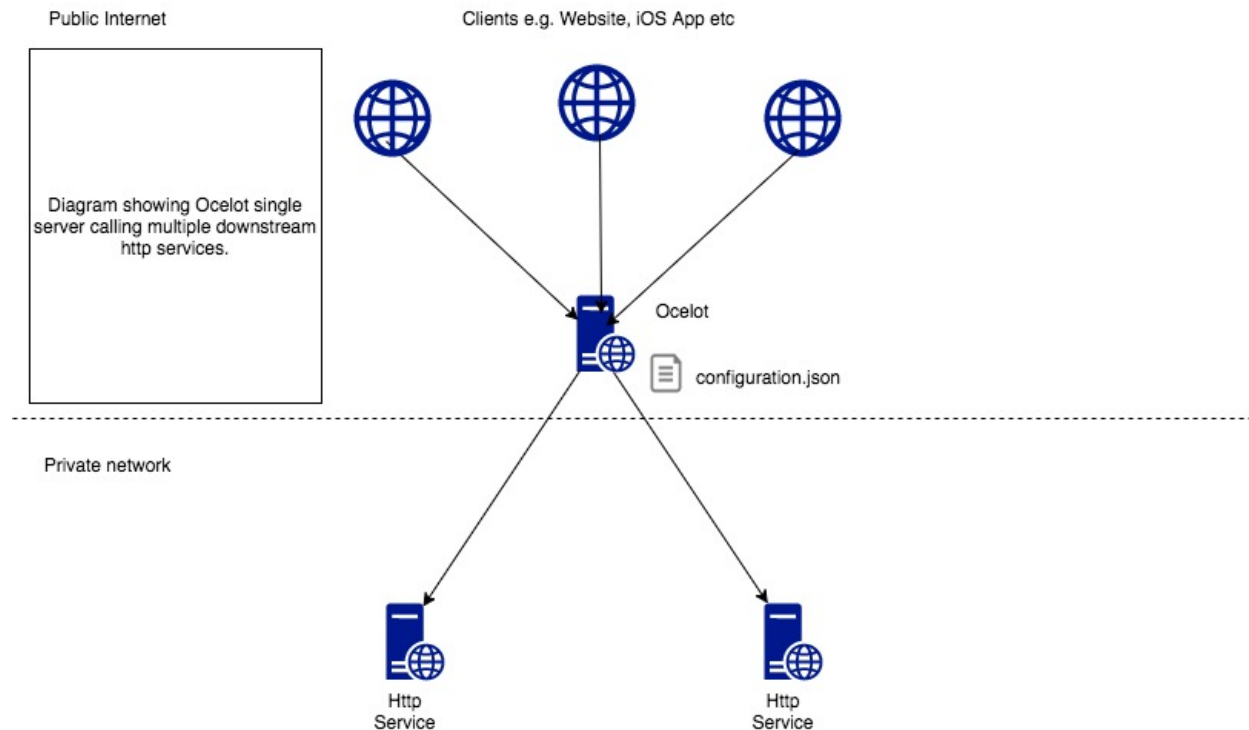
In particular we want easy integration with [IdentityServer](#) reference and [Bearer](#) tokens. We have been unable to find this in our current workplace without having to write our own Javascript middlewares to handle the IdentityServer reference tokens. We would rather use the IdentityServer code that already exists to do this.

Ocelot is a bunch of middlewares in a specific order.

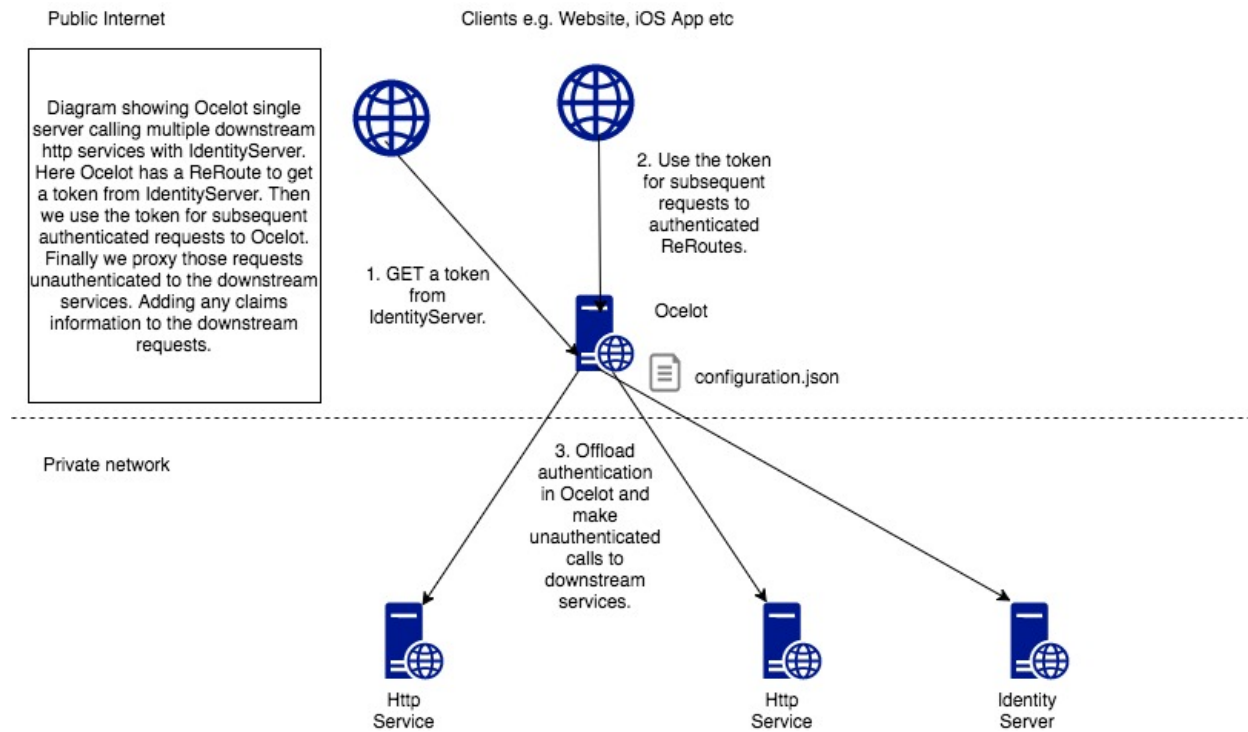
Ocelot manipulates the `HttpRequest` object into a state specified by its configuration until it reaches a request builder middleware, where it creates a `HttpRequestMessage` object which is used to make a request to a downstream service. The middleware that makes the request is the last thing in the Ocelot pipeline. It does not call the next middleware. The response from the downstream service is retrieved as the requests 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.

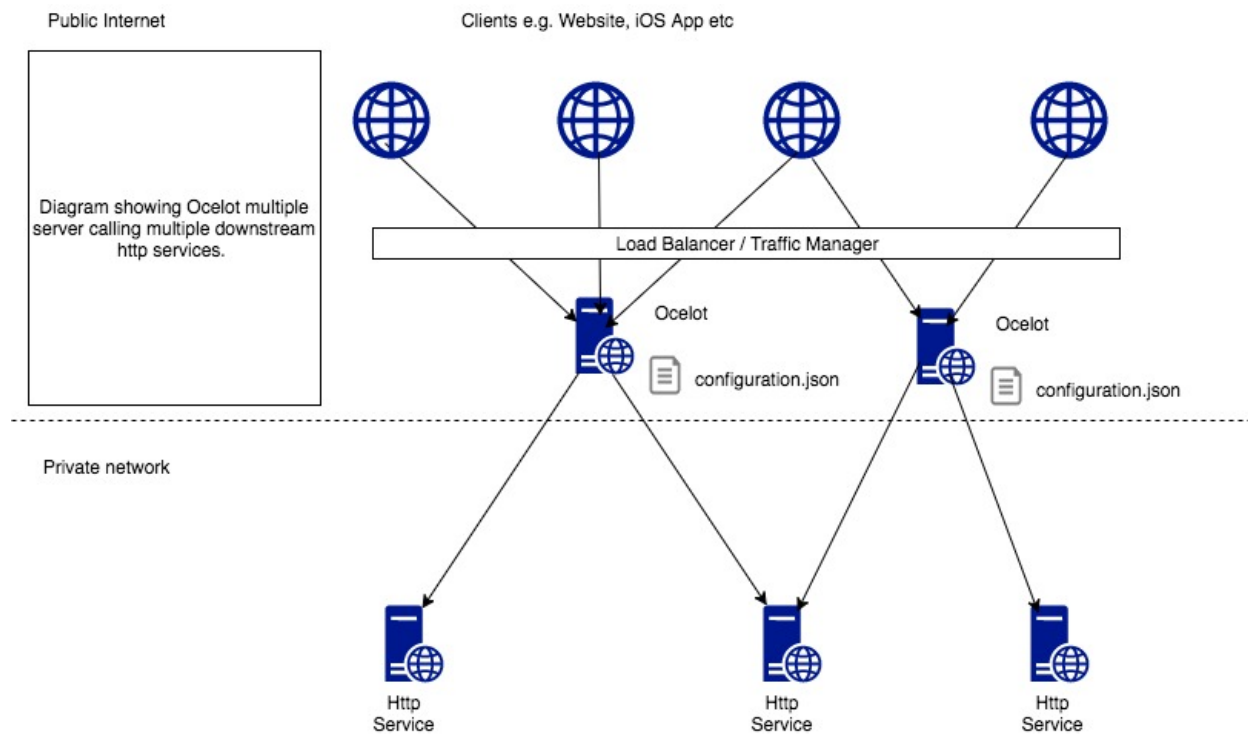
Basic Implementation



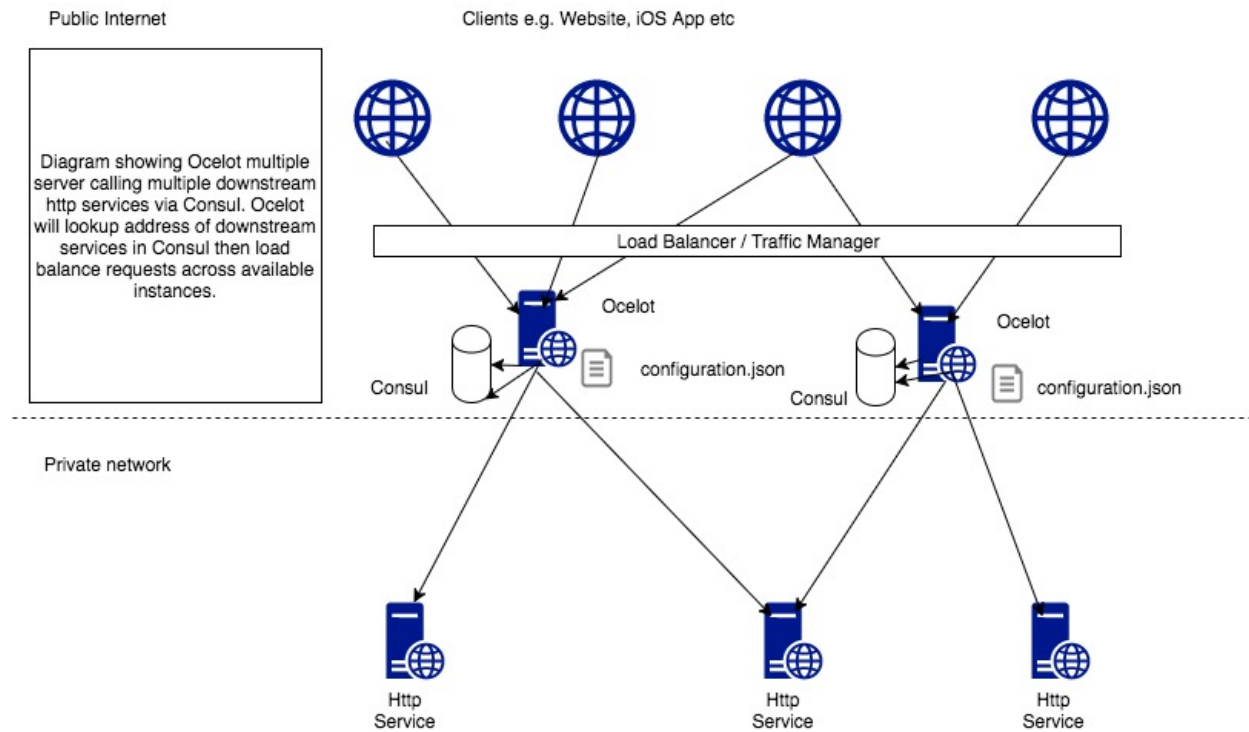
With IdentityServer



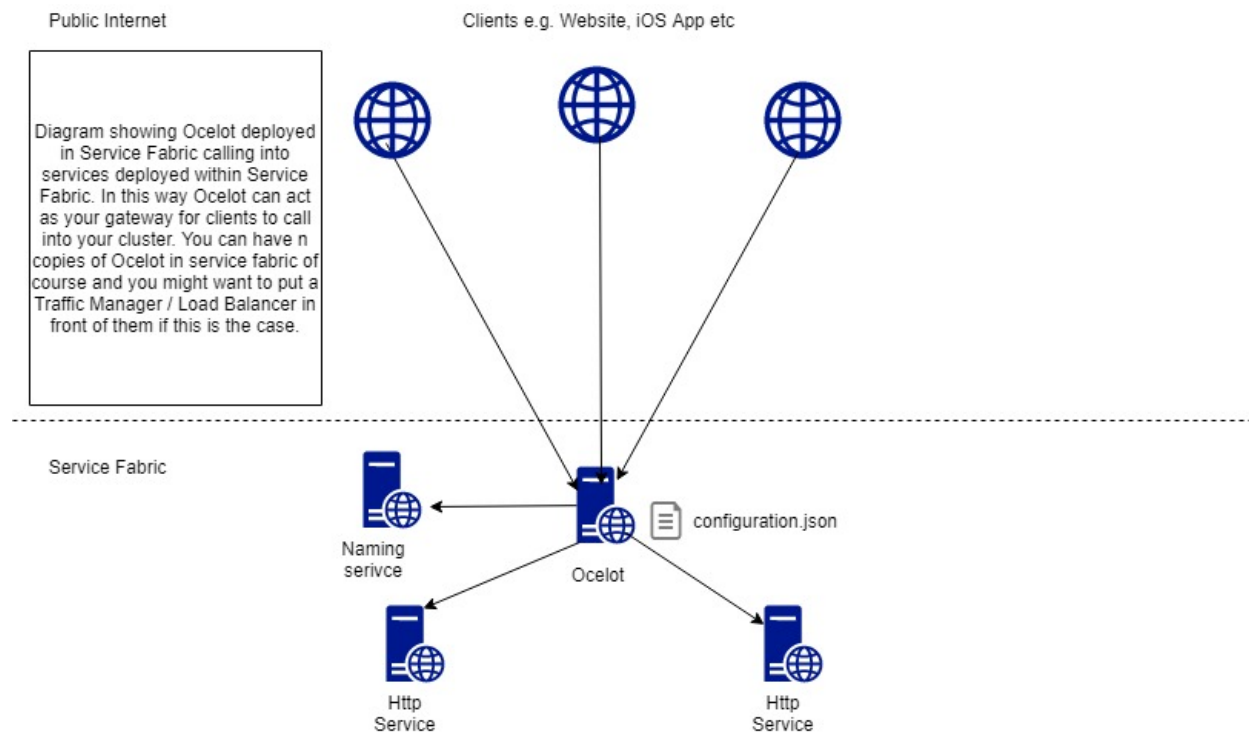
Multiple Instances



With Consul



With Service Fabric



Getting Started

Ocelot is designed to work with ASP.NET and is currently on `net6.0`, `net7.0` and `net8.0` frameworks.

.NET 8.0

Install NuGet package

Install Ocelot and its dependencies using [NuGet](#). You will need to create a [ASP.NET Core minimal API project](#) 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](#).

Configuration

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

```
{
  "Routes": [],
  "GlobalConfiguration": {
    "BaseUrl": "https://api.mybusiness.com"
  }
}
```

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

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/todos/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "jsonplaceholder.typicode.com",
          "Port": 443
        }
      ],
      "UpstreamPathTemplate": "/todos/{id}",
      "UpstreamHttpMethod": [ "Get" ]
    }
  ],
  "GlobalConfiguration": {
    "BaseUrl": "https://localhost:5000"
  }
}
```

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 & replace 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://123.12.1.1:6543` 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://123.12.1.1:6543` 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.

Program

Then in your **Program.cs** you will want to have the following.

The main things to note are

- `AddOcelot()` adds Ocelot required and default services¹
- `UseOcelot().Wait()` sets up all the Ocelot middlewares.

```
using System.IO;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Ocelot.DependencyInjection;
using Ocelot.Middleware;

namespace OcelotBasic
{
    public class Program
    {
        public static void Main(string[] args)
        {
            new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .ConfigureAppConfiguration((hostingContext, config) =>
                {
                    config
                        .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
                        .AddJsonFile("appsettings.json", true, true)
                        .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.
↵EnvironmentName}.json", true, true)
                        .AddJsonFile("ocelot.json")
                        .AddEnvironmentVariables();
                })
                .ConfigureServices(s => {
                    s.AddOcelot();
                })
                .ConfigureLogging((hostingContext, logging) =>
                {
                    //add your logging
                })
        }
    }
}
```

(continues on next page)

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

(continued from previous page)

```
.UseIISIntegration()
.Configure(app =>
{
    app.UseOcelot().Wait();
})
.Build()
.Run();
}
}
```

Contributing

Pull requests, issues and commentary welcome!

Ideas, questions could be posted to Ocelot [Discussions](#) space.

We do follow development process which is described in [Release Process](#).

Not Supported

Ocelot does not support...

Chunked Encoding

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

Forwarding a Host header

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

Swagger

Contributors have looked multiple times at building **swagger.json** out of the Ocelot **ocelot.json** but it doesn't 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 **Startup.cs** or **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:

```
app.Map("/swagger/v1/swagger.json", b =>
{
    b.Run(async x => {
        var json = File.ReadAllText("swagger.json");
        await x.Response.WriteAsync(json);
    });
});
```

(continues on next page)

(continued from previous page)

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Ocelot");
});

app.UseOcelot().Wait();
```

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 doesn't 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 don't intend to do this.

Gotchas

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.

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 constantly 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](#) in the repository. Probably you will find a ready solution by Ocelot community members.

Finally, we have special label for all IIS related objects. Feel free to put this label onto issues, PRs, discussions, etc.

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.

- **Upload and download large files**¹, proxying the content through the gateway. It is strange 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 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 higher¹.

In case of some errors, see the next point.

- **Maximum request body size.** 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.

Please review these docs: [Maximum request body size | Configure options for the ASP.NET Core Kestrel web server](#).

As a quick fix, use this configuration recipe:

```
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
});
```

Hope it helps.

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.

Administration

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 [NuGet package](#):

```
Install-Package Ocelot.Administration
```

This will bring down everything needed by the administration API.

Providing your own IdentityServer

All you need to do to hook into your own IdentityServer is add the following configuration options with authentication to your `ConfigureServices` method. After that we must pass these options to `AddAdministration()` extension of the `OcelotBuilder` being returned by `AddOcelot()`¹ like below:

```
public virtual void ConfigureServices(IServiceCollection services)
{
    Action<JwtBearerOptions> options = o =>
    {
        o.Authority = identityServerRootUrl;
        o.RequireHttpsMetadata = false;
        o.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateAudience = false,
        };
        // etc...
    };

    services
        .AddOcelot()
        .AddAdministration("/administration", options);
}
```

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

This feature was implemented for [Issue 228](#). It is useful because the IdentityServer authentication middleware needs the URL of the IdentityServer. If you are using the internal IdentityServer, it might not always be possible to have the Ocelot URL.

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

Internal IdentityServer

The API is authenticated using Bearer tokens that you request from Ocelot itself. This is provided by the amazing [Identity Server](#) project that the .NET community has been using for several years. Check them out.

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

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 13, 1} like below:

```
public virtual void ConfigureServices(IServiceCollection services)
{
    services
        .AddOcelot()
        .AddAdministration("/administration", "secret");
}
```

In order for the administration API to work, Ocelot / IdentityServer must be able to call itself for validation. This means that you need to add the base URL of Ocelot to global configuration if it is not default `http://localhost:5000`. Please 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. Anyway, this can be done as follows.

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

```
"GlobalConfiguration": {
  "BaseUrl": "http://localhost:55580"
}
```

or if Ocelot is exposed via DNS:

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

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 configuration. 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!

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.

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 **Startup.cs** as usual but they provide a [scheme](#) (authentication provider key) with each registration e.g.

```
public void ConfigureServices(IServiceCollection services)
{
    const string AuthenticationProviderKey = "MyKey";
    services
        .AddAuthentication()
```

(continues on next page)

(continued from previous page)

```

        .AddJwtBearer(AuthenticationProviderKey, options =>
        {
            // Custom Authentication setup via options initialization
        });
    }

```

In this example MyKey is the [scheme](#) that this provider has been registered with. We then map this to a Route in the configuration using the following [AuthenticationOptions](#) properties:

- `AuthenticationProviderKey` is a string object, obsolete¹. This is legacy definition when you define *Single Key aka Authentication Scheme 1*.
- `AuthenticationProviderKeys` is an array of strings, the recommended definition of *Multiple Authentication Schemes 2* feature.

Single Key aka Authentication Scheme^{Page 16, 1}

Property: `AuthenticationOptions.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](#).

Multiple Authentication Schemes²

Property: `AuthenticationOptions.AuthenticationProviderKeys`

In real world of ASP.NET, apps may need to support multiple types of authentication by 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:

```

public void ConfigureServices(IServiceCollection services)
{
    const string DefaultScheme = JwtBearerDefaults.AuthenticationScheme; // Bearer
    services.AddAuthentication()
        .AddJwtBearer(DefaultScheme, options => { /* JWT setup */ })
        // AddJwtBearer, AddCookie, AddIdentityServerAuthentication etc.
        .AddMyProvider("MyKey", options => { /* Custom auth setup */ });
}

```

¹ 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.

² "Multiple authentication schemes" feature was requested in issues 740, 1580 and delivered as a part of 23.0 release.

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 Key aka Authentication Scheme 1*.

Note that the order of the keys in an array definition does matter! We use a “First One Wins” authentication strategy.

Finally, we would say that registering providers, initializing options, forwarding authentication artifacts can be a “real” coding challenge. If you’re stuck or don’t know what to do, just find inspiration in our [acceptance tests](#) (currently for Identity Server 4 only)³.

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.

```
public void ConfigureServices(IServiceCollection services)
{
    var authenticationProviderKey = "MyKey";
    services
        .AddAuthentication()
        .AddJwtBearer(authenticationProviderKey, options =>
        {
            options.Authority = "test";
            options.Audience = "test";
        });
    services.AddOcelot();
}
```

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

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

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](#)

³ We would appreciate any new PRs to add extra acceptance tests for your custom scenarios with multiple authentication schemes.

Identity Server Bearer Tokens

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

```
public void ConfigureServices(IServiceCollection services)
{
    var authenticationProviderKey = "MyKey";
    Action<JwtBearerOptions> options = (opt) =>
    {
        opt.Authority = "https://whereyouridentityserverlives.com";
        // ...
    };
    services
        .AddAuthentication()
        .AddJwtBearer(authenticationProviderKey, options);
    services.AddOcelot();
}
```

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

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

Auth0 by Okta

Yet another identity provider by [Okta](#), see [Auth0 Developer Resources](#).

Add the following to your startup `Configure` method:

```
app.UseAuthentication()
    .UseOcelot().Wait();
```

Add the following, at minimum, to your startup `ConfigureServices` method:

```
services
    .AddAuthentication()
    .AddJwtBearer(oktaProviderKey, options =>
    {
        options.Audience = configuration["Authentication:Okta:Audience"]; // Okta_
        ↪Authorization server Audience
        options.Authority = configuration["Authentication:Okta:Server"]; // Okta_
        ↪Authorization Issuer URI URL e.g. https://{subdomain}.okta.com/oauth2/{authidentifier}
    });
services.AddOcelot(configuration);
```

Note 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
```

(continues on next page)

(continued from previous page)

```
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Remove("scp");  
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Add("scp", "scope");
```

[Issue 446](#) contains some code and examples that might help with Okta integration.

Allowed Scopes

If you add scopes to **AllowedScopes**, Ocelot will get all the user claims (from the token) of the type scope and make sure that the user has at least one of the scopes in the list.

This is a way to restrict access to a Route on a per scope basis.

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 .NET Blog: [ASP.NET Core Authentication with IdentityServer4](#)

Future

We invite you to add more examples, if you have integrated with other identity providers and the integration solution is working. Please, open [Show and tell](#) discussion in the repository.

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 **AuthorizationMiddleware** 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](#).

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](#) section, the Authorization middleware can be overridden like this:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    var configuration = new OcelotPipelineConfiguration
    {
        AuthorizationMiddleware = async (context, next) =>
        {
            await next.Invoke();
        }
    };
    app.UseOcelot(configuration);
}
```

Do this in very rare cases, because overriding Authorization middleware means you will lose claims & 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.

Caching

Ocelot supports some very rudimentary caching at the moment provided 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.

Install

First of all, add the following [NuGet package](#):

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

This will give you access to the Ocelot cache manager extension methods.

The second thing you need to do something like the following to your ConfigureServices method:

```
using Ocelot.Cache.CacheManager;

ConfigureServices(services =>
```

(continues on next page)

(continued from previous page)

```
{
    services.AddOcelot()
        .AddCacheManager(x => x.WithDictionaryHandle());
};
```

Configuration

Finally, in order to use caching on a route in your Route configuration add this setting:

```
"FileCacheOptions": {
    "TtlSeconds": 15,
    "Region": "europe-central",
    "Header": "Authorization"
}
```

In this example **TtlSeconds** is set to 15 which means the cache will expire after 15 seconds. The **Region** represents a region of caching.

Additionally, if a header name is defined in the **Header** property, that header value is looked up by the key (header name) in the `HttpRequest` headers, and if the header is found, its value will be included in caching key. This causes the cache to become invalid due to the header value changing.

If you look at the example [here](#) you can see how the cache manager is setup and then passed into the Ocelot `AddCacheManager` configuration method. You can use any settings supported by the **CacheManager** package and just pass them in.

Anyway, Ocelot currently supports caching on the URL of the downstream service and setting a TTL in seconds to expire the cache. You can also clear the cache for a region by calling Ocelot's administration API.

Your Own Caching

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

```
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.

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.

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.

Claims to Claims Transformation

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`.

Claims to Headers Transformation

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`.

Claims to Query String Parameters Transformation

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.

Claims to Downstream Path Transformation

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.

Configuration

An example configuration can be found here in [ocelot.json](#). There are two sections to the configuration: an array of **Routes** and a **GlobalConfiguration**:

- The **Routes** are the objects that tell Ocelot how to treat an upstream request.
- The **GlobalConfiguration** is a bit hacky and allows overrides of Route specific settings. It’s useful if you do not want to manage lots of Route specific settings.

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

Here is an example Route configuration. You don’t need to set all of these things but this is everything that is available at the moment:

```
{
  "DownstreamPathTemplate": "/",
  "UpstreamPathTemplate": "/",
  "UpstreamHttpMethod": [ "Get" ],
  "DownstreamHttpMethod": "",
  "DownstreamHttpVersion": "",
  "AddHeadersToRequest": {},
}
```

(continues on next page)

(continued from previous page)

```
"AddClaimsToRequest": {},
"RouteClaimsRequirement": {},
"AddQueriesToRequest": {},
"RequestIdKey": "",
"FileCacheOptions": {
  "TtlSeconds": 0,
  "Region": "europe-central"
},
"RouteIsCaseSensitive": false,
"ServiceName": "",
"DownstreamScheme": "http",
"DownstreamHostAndPorts": [
  { "Host": "localhost", "Port": 51876 }
],
"QoSOptions": {
  "ExceptionsAllowedBeforeBreaking": 0,
  "DurationOfBreak": 0,
  "TimeoutValue": 0
},
"LoadBalancer": "",
"RateLimitOptions": {
  "ClientWhitelist": [],
  "EnableRateLimiting": false,
  "Period": "",
  "PeriodTimespan": 0,
  "Limit": 0
},
"AuthenticationOptions": {
  "AuthenticationProviderKey": "",
  "AllowedScopes": []
},
"HttpHandlerOptions": {
  "AllowAutoRedirect": true,
  "UseCookieContainer": true,
  "UseTracing": true,
  "MaxConnectionsPerServer": 100
},
"DangerousAcceptAnyServerCertificateValidator": false,
"SecurityOptions": {
  "IPAllowedList": [],
  "IPBlockedList": [],
  "ExcludeAllowedFromBlocked": false
}
}
```

More information on how to use these options is below.

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:

```
ConfigureAppConfiguration((context, config) =>
{
    var env = context.HostingEnvironment;
    config.SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", true, true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true)
        .AddJsonFile("ocelot.json") // primary config file
        .AddJsonFile($"ocelot.{env.EnvironmentName}.json") // environment file
        .AddEnvironmentVariables();
})
```

Ocelot will now use the environment specific configuration and fall back to `ocelot.json` if there isn't one.

You also need to set the corresponding environment variable which is `ASPNETCORE_ENVIRONMENT`. More info on this can be found in the ASP.NET Core docs: [Use multiple environments in ASP.NET Core](#).

Merging Configuration Files

This feature allows users to have multiple configuration files to make managing large configurations easier.¹

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

```
ConfigureAppConfiguration((context, config) =>
{
    var env = context.HostingEnvironment;
    config.SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", true, true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true)
        .AddOcelot(env) // happy path
        .AddEnvironmentVariables();
})
```

In this scenario Ocelot will look for any files that match the pattern `^ocelot\.(.*?)\.json$` and then merge these together. 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, add any **Routes**, add any **AggregateRoutes** and if the file is called `ocelot.global.json` add the **GlobalConfiguration** as well 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.

At the moment there is no validation at this stage it only happens when Ocelot validates the final merged configuration. This is something to be aware of when you are investigating problems. We would advise always checking what is in `ocelot.json` file if you have any problems.

¹ “Merging Configuration Files” feature was requested in [issue 296](#), since then we extended it in [issue 1216](#) (PR 1227) as “Merging files to memory 2” subfeature which was released as a part of version 23.2.

Keep files in a folder

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

```
ConfigureAppConfiguration((context, config) =>
{
    var env = context.HostingEnvironment;
    config.SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", true, true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true)
        .AddOcelot("/my/folder", env) // happy path
        .AddEnvironmentVariables();
})
```

Ocelot needs the `HostingEnvironment` so it knows to exclude anything environment specific from the merging algorithm.

Merging files to memory^{Page 26, 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:

```
// It implicitly calls ASP.NET AddJsonStream extension method for IConfigurationBuilder
// config.AddJsonStream(new MemoryStream(Encoding.UTF8.GetBytes(json)));
config.AddOcelot(context.HostingEnvironment, 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!²

Reload JSON Config 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:

```
config.AddJsonFile("ocelot.json", optional: false, reloadOnChange: true); // ASP.NET
↪ framework version
```

Important 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 internal `AddJsonFile` method during the final configuration step (see `AddOcelotJsonFile` implementation):

```
config.AddJsonFile(ConfigurationBuilderExtensions.PrimaryConfigFile, optional ?? false,
↪ reloadOnChange ?? false);
```

² “Merging files to memory 2” subfeature 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.

As you can see, in versions prior to 23.2, the `AddOcelot` extension methods did not apply the `reloadOnChange` argument because it was set to `false`. We recommend using the `AddOcelot` extension methods to control reloading, rather than relying on the framework's `AddJsonFile` method. For example:

```
ConfigureAppConfiguration((context, config) =>
{
    config.AddJsonFile(ConfigurationBuilderExtensions.PrimaryConfigFile, optional: false,
    ↪ reloadOnChange: true); // old approach
    var env = context.HostingEnvironment;
    var mergeTo = MergeOcelotJson.ToFile; // ToMemory
    var folder = "/My/folder";
    FileConfiguration configuration = new(); // read from anywhere and initialize
    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 methods².

Store Configuration in Consul

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

```
Install-Package Ocelot.Provider.Consul
```

Then you add the following when you register your services Ocelot will attempt to store and retrieve its configuration in Consul KV store. In order to register Consul services we must call the `AddConsul()` and `AddConfigStoredInConsul()` extensions using the `OcelotBuilder` being returned by `AddOcelot()`³ like below:

```
services.AddOcelot()
    .AddConsul()
    .AddConfigStoredInConsul();
```

You also need to add the following to your `ocelot.json`. This is how Ocelot finds your Consul agent and interacts to load and store the configuration from Consul.

```
"GlobalConfiguration": {
  "ServiceDiscoveryProvider": {
    "Host": "localhost",
    "Port": 9500
  }
}
```

The team decided to create this feature after working on the Raft consensus algorithm and finding out its super hard. Why not take advantage of the fact Consul already gives you this! We guess it means if you want to use Ocelot to its

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

fullest, you take on Consul as a dependency for now.

This feature has a 3 seconds TTL cache before making a new request to your local Consul agent.

Consul Configuration Key^{Page 28, 4}

If you are using Consul for configuration (or other providers in the future), you might want to key your configurations: so you can have multiple configurations.

In order to specify the key you need to set the **ConfigurationKey** property in the **ServiceDiscoveryProvider** options of the configuration JSON file e.g.

```
"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 use the string InternalConfiguration as the key.

Follow Redirects aka **HttpHandlerOptions**

Class: `FileHttpHandlerOptions`

Use `HttpHandlerOptions` in a Route configuration to set up `HttpHandler` behavior:

```
"HttpHandlerOptions": {  
  "AllowAutoRedirect": false,  
  "UseCookieContainer": false,  
  "UseTracing": true,  
  "MaxConnectionsPerServer": 100  
},
```

- **AllowAutoRedirect** is a value that indicates whether the request should follow redirection responses. Set it `true` if the request should automatically follow redirection responses from the downstream resource; otherwise `false`. The default value is `false`.
- **UseCookieContainer** is a value that indicates whether the handler uses the `CookieContainer` property to store server cookies and uses these cookies when sending requests. The default value is `false`. Please note, if you use the `CookieContainer`, Ocelot caches the `HttpClient` 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!
- **MaxConnectionsPerServer** This controls how many connections the internal `HttpClient` will open. This can be set at Route or global level.

⁴ “*Consul Configuration Key 4*” feature was requested in [issue 346](#) as a part of version 7.0.0.

SSL Errors

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

```
"DangerousAcceptAnyServerCertificateValidator": true
```

We don't recommend doing this! The team suggests creating your own certificate and then getting it trusted by your local (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](#).

As a team, we do not consider it as an ideal solution. From one side, the community wants to have an option to work with self-signed certificates. But from other side, currently source code scanners detect 2 serious security vulnerabilities because of this fake validator in `20.0` release. 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 having self-signed certificate 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 the feature to avoid SSL errors for self-signed certificates in case of `https` or `wss` schemes. We understand that some routes should have downstream scheme exactly with SSL, because they are also in development, and/or deployed using SSL protocols. But 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. But 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 & hosting providers which have own signed certificates for SSL. At least your team can deploy one remote web server to host downstream services. Install own certificate or use cloud provider's one.
- **Staging or testing environments.** We do not recommend to use self-signed certificates because web servers should have valid certificates installed. Ask your system administrator or DevOps engineers of your team to create valid certificates.
- **Production environments. Do not use self-signed certificates at all!** System administrators or DevOps engineers must create real valid certificates being signed by hosting or cloud providers. **Switch off the feature for all routes!** Remove the `DangerousAcceptAnyServerCertificateValidator` property for all routes in production version of `ocelot.json` file!

React to Configuration 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 `ICancellationToken.HasChanged` property, or register a callback with the `RegisterChangeCallback` method.

Polling the HasChanged property

```
public class ConfigurationNotifyingService : BackgroundService
{
    private readonly IOcelotConfigurationChangeTokenSource _tokenSource;
    private readonly ILogger _logger;

    public ConfigurationNotifyingService(IOcelotConfigurationChangeTokenSource _
↪tokenSource, ILogger logger)
    {
        _tokenSource = tokenSource;
        _logger = logger;
    }

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

Registering a callback

```
public class MyDependencyInjectedClass : IDisposable
{
    private readonly IOcelotConfigurationChangeTokenSource _tokenSource;
    private readonly IDisposable _callbackHolder;

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

    public void Dispose()
    {
        _callbackHolder.Dispose();
    }
}
```

DownstreamHttpVersion

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.

Dependency Injection

Dependency Injection for this **Configuration** feature in Ocelot is designed to extend and/or control **the configuration** of the Ocelot kernel before the stage of building ASP.NET MVC 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 *ConfigureAppConfiguration* method (located in both **Program.cs** and **Startup.cs**) of your ASP.NET MVC gateway app (minimal web app) to configure the Ocelot pipeline and services.

```
namespace Microsoft.AspNetCore.Hosting;

public interface IWebHostBuilder
{
    IWebHostBuilder ConfigureAppConfiguration(Action<WebHostBuilderContext,
    ↳ IConfigurationBuilder> configureDelegate);
}
```

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

Delegating Handlers

Ocelot allows the user to add delegating handlers to the *HttpClient* transport. This feature was requested by [issue 208](#) and the team decided that it was going to be useful in various ways. Since then we extended it in [issue 264](#).

How to Use

In order to add delegating handlers to the *HttpClient* transport you need to do two main things.

First, in order to create a class that can be used a delegating handler it must look as follows. 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 FakeHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage
    ↳ request, CancellationToken token)
    {
        // Do stuff and optionally call the base handler...
        return await base.SendAsync(request, token);
    }
}
```

Second, you must add the handlers to DI container like below:

```
ConfigureServices(s => s
    .AddOcelot()
    .AddDelegatingHandler<FakeHandler>()
    .AddDelegatingHandler<FakeHandlerTwo>()
)
```

Both of these `AddDelegatingHandler` methods have a default parameter called `global` which is set to `false`. If it is `false` then the intent of the *Delegating Handler* is to be applied to specific Routes via **ocelot.json** (more on that later). If it is set to `true` then it becomes a global handler and will be applied to all Routes, as below:

```
services.AddOcelot()
    .AddDelegatingHandler<FakeHandler>(true)
```

Finally, if you want Route specific *Delegating Handlers* or to order your specific and (or) global (more on this later) *Delegating Handlers* 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": [
  "FakeHandlerTwo",
  "FakeHandler"
]
```

Order of Execution

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** array from **ocelot.json**.
2. Any non global *Delegating Handlers* plus any globals that were in the **DelegatingHandlers** array from **ocelot.json** ordered as they are in the **DelegatingHandlers** array.
3. Tracing *Delegating Handler*, if enabled (see [Tracing](#) docs).
4. Quality of Service *Delegating Handler*, if enabled (see [Quality of Service](#) docs).
5. The `HttpClient` sends the `HttpRequestMessage`.

Hopefully other people will find this feature useful!

Dependency Injection

Namespace: `Ocelot.DependencyInjection`

Source code: [DependencyInjection](#)

Overview

Dependency Injection feature in Ocelot is designed to extend and/or control building of Ocelot core as ASP.NET MVC pipeline services.

The main methods of the [ServiceCollectionExtensions](#) class are:

- [AddOcelot](#) adds required Ocelot services to DI and it adds default services using [AddDefaultAspNetServices](#) method.
- [AddOcelotUsingBuilder](#) adds required Ocelot services to DI, and **it adds custom ASP.NET services** with configuration injected implicitly or explicitly.

Use [IServiceCollection extensions](#) in the following `ConfigureServices` method (**Program.cs** and **Startup.cs**) of your ASP.NET MVC gateway app (minimal web app) to add/build Ocelot pipeline services:

```
namespace Microsoft.AspNetCore.Hosting;
public interface IWebHostBuilder
{
    IWebHostBuilder ConfigureServices(Action<IServiceCollection> configureServices);
}
```

The fact is, the [OcelotBuilder](#) class is Ocelot's cornerstone logic.

IServiceCollection extensions

Namespace: `Ocelot.DependencyInjection`

Class: [ServiceCollectionExtensions](#)

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

The 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 required and **default** ASP.NET services for Ocelot pipeline in the [AddDefaultAspNetServices](#) method which is 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 pipeline. The alternative is `AddOcelotUsingBuilder` method, see the next subsection.

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 they add **custom ASP.NET services** with configuration injected implicitly or explicitly.

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

In this custom scenario, you control everything during ASP.NET MVC pipeline building, and you provide custom settings to build Ocelot pipeline.

OcelotBuilder class

Source code: [Ocelot.DependencyInjection.OcelotBuilder](#)

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

- Contracts itself by single public constructor:

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

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

AddDefaultAspNetServices method

Class: *OcelotBuilder*

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

Current [implementation](#) is the following:

```
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 current behavior by inheritance. And, the method is default builder of Ocelot pipeline while calling the [AddOcelot](#) method. As alternative, to “override” this default builder, you can design and reuse custom builder as a `Func<IMvcCoreBuilder, Assembly, IMvcCoreBuilder>` delegate object and pass it as parameter to the [AddOcelotUsingBuilder](#) extension method. It gives you full control on design and building of Ocelot pipeline, but be careful while designing your custom Ocelot pipeline as customizable ASP.NET MVC pipeline.

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

Warning!! The method above is called after adding required services of ASP.NET MVC pipeline building by [AddMvcCore](#) method over the `Services` property in upper calling context. These services are absolute minimum core services for ASP.NET MVC pipeline. They must be added to DI container always, and they are added implicitly before calling of the method by caller in upper context. So, `AddMvcCore` creates an *IMvcCoreBuilder* object with its assignment to the `MvcCoreBuilder` property. Finally, as a default builder, the method above receives *IMvcCoreBuilder* object being ready for further extensions.

The next section shows you an example of designing custom Ocelot pipeline by custom builder.

Custom Builder

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

Problem

The main `AddOcelot` method adds `Newtonsoft JSON` services by the `AddNewtonsoftJson` extension method in default builder (`AddDefaultAspNetServices` method). The `AddNewtonsoftJson` method calling was introduced in old .NET and Ocelot releases which was necessary when Microsoft did not launch the `System.Text.Json` library, but now it affects normal use, so we have an intention to solve the problem.

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

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

```
using Microsoft.Extensions.DependencyInjection;
using Ocelot.DependencyInjection;
using System.Reflection;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services
            .AddLogging()
            .AddMiddlewareAnalysis()
            .AddWebEncoders()
            // Add your custom builder
            .AddOcelotUsingBuilder(MyCustomBuilder);
    }

    private static IMvcCoreBuilder MyCustomBuilder(IMvcCoreBuilder builder, Assembly assembly)
    {
        return builder
            .AddApplicationPart(assembly)
            .AddControllersAsServices()
            .AddAuthorization();
    }
}
```

(continues on next page)

(continued from previous page)

```
// 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 compressed plain JSON text without spaces. This is just one common use case, and you can add additional services to the builder.

Configuration Overview

Dependency Injection for the *Configuration* feature in Ocelot is designed to extend and/or control **the configuration** of Ocelot kernel before the stage of building ASP.NET MVC pipeline services.

To configure the Ocelot pipeline and services, use the *IConfigurationBuilder extensions* in the following *ConfigureAppConfiguration* method (located in *Program.cs* and *Startup.cs*) of your minimal web app:

```
namespace Microsoft.AspNetCore.Hosting;
public interface IWebHostBuilder
{
    IWebHostBuilder ConfigureAppConfiguration(Action<WebHostBuilderContext, ↪
↪IConfigurationBuilder> configureDelegate);
}
```

IConfigurationBuilder extensions

Namespace: Ocelot.DependencyInjection

Class: ConfigurationBuilderExtensions

The main methods are the *AddOcelot methods* within the *ConfigurationBuilderExtensions* class. This method has a list of overloaded versions with corresponding signatures.

The purpose of this 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.

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

```
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](#)^{Page 38, 1}.

¹ The Dynamic *Configuration* 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.

Error Status Codes

Ocelot will return HTTP status error codes based on internal logic in certain situations:

Client error responses

- **401** - if the authentication middleware runs and the user is not authenticated.
- **403** - if the authorization middleware runs and the user is unauthenticated, claim value not authorized, scope not authorized, user doesn't have required claim, or cannot find claim.
- **404** - if unable to find a downstream route, or Ocelot is unable to map an internal error code to a HTTP status code.
- **499** - if the request is cancelled by the client.


Server error responses

- **500** - if unable to complete the HTTP request to downstream service, and the exception is not `OperationCanceledException` or `HttpRequestException`.
- **502** - if unable to connect to downstream service.
- **503** - if the downstream request times out.

Design

Historically Ocelot errors are implemented by the [HttpExceptionToErrorMessageMapper](#) class. The `Map` method converts a `System.Exception` object to native `Ocelot.Errors.ErrorMessage` object.

We do HTTP status code overriding 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 upstream service. If you did not find statuses and (or) required headers then [Headers Transformation](#) feature should help.

We expect you to share your user case with us in the [Discussions](#) space of the repository. 



Ocelot doesn't directly support [GraphQL](#), but so many people have asked about it. We wanted to show how easy it is to integrate the [GraphQL for .NET](#) library.

Please see the sample project [OcelotGraphQL](#). Using a combination of the [graphql-dotnet](#) project and Ocelot [Delegating Handlers](#) features, this is pretty easy to do. However we do not intend to integrate more closely with **GraphQL** at the moment. Check out the samples [README.md](#) and that should give you enough instruction on how to do this!

Future

If you have sufficient experience with GraphQL and mentioned .NET [package](#), we would welcome your contribution to the sample. 🐙

Who knows, maybe you'll get inspired by the sample development and come up with some design solution in the form of a rough draft of GraphQL feature to implement in Ocelot. Good luck!

And, welcome to [Discussions](#) space of the repository!

Headers Transformation

Ocelot allows the user to transform headers pre and post downstream request. At the moment Ocelot only supports find and replace. This feature was requested in [issue 190](#) and the team decided that it was going to be useful in various ways.

Add to Request

This feature was requested in [issue 313](#).

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

```
"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).

Add to Response

This feature was requested in [issue 280](#).

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 APM](#) trace id then do something like the following:

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

Find and Replace

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.com/"
```

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

Post Downstream Request

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

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

Placeholders

Ocelot allows placeholders that can be used in header transformation.

- `{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** at the moment.
- `{RemoteIpAddress}` - This will find the clients IP address using `IHttpContextAccessor.HttpContext.Connection.RemoteIpAddress.ToString()`, so you will get back some IP. See more in the [GetRemoteIpAddress](#) method.
- `{TraceId}` - This will use the [Butterfly APM](#) Trace Id. This only works for **DownstreamHeaderTransform** at the moment.
- `{UpstreamHost}` - This will look for the incoming Host header.

For now, we believe these placeholders are sufficient for basic user scenarios. But if you need more placeholders, you can head to the *future*.

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 this with the following configuration:

```
"DownstreamHeaderTransform": {
  "Location": "http://www.bbc.co.uk/, http://ocelot.com/"
},
"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,
}
```

X-Forwarded-For

An example of using `{RemoteIpAddress}` placeholder:

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

Future


Ideally this 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,
}
```

If anyone wants to have a go at this please, help yourself!

Global Headers Transformation

We have pending open [PR 1659](#) for the [1658](#) issue. Current 20.0 version provides Route-level *Headers Transformation* feature, but we hope **global** transformations will be included in the next upcoming [release](#).

Any ideas and proposals can be shared in the [Discussions](#) space of the repository! 



Kubernetes^{Page 43, 1} aka K8s

A part of feature: *Service Discovery*²

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 [PR 1134](#) because the service did not load balance as expected.

Install

The first thing you need to do is install the [NuGet package](#) that provides **Kubernetes**¹ support in Ocelot:

```
Install-Package Ocelot.Provider.Kubernetes
```

Then add the following to your `ConfigureServices` method:

```
services.AddOcelot().AddKubernetes();
```

If you have services deployed in Kubernetes, you will normally use the naming service to access them. Default `usePodServiceAccount = 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:

```
public static class OcelotBuilderExtensions
{
    public static IOcelotBuilder AddKubernetes(this IOcelotBuilder builder, bool_
↪ usePodServiceAccount = true);
}
```

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
```

¹ [Wikipedia](#) | [K8s Website](#) | [K8s Documentation](#) | [K8s GitHub](#)

² This feature was requested as part of [issue 345](#) to add support for [Kubernetes Service Discovery](#) provider.

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**.

Kube default provider

The example here shows a typical configuration:

```
"Routes": [  
  {  
    "ServiceName": "downstreamservice",  
    // ...  
  }  
],  
"GlobalConfiguration": {  
  "ServiceDiscoveryProvider": {  
    "Host": "192.168.0.13",  
    "Port": 443,  
    "Token": "txpc696iUhbVoudg164r93CxDTKrKRVWG",  
    "Namespace": "Dev",  
    "Type": "Kube"  
  }  
}
```

Service deployment in **Namespace** Dev, **ServiceDiscoveryProvider** type is Kube, you also can set *PollKube provider* type. Note: **Host**, **Port** and **Token** are no longer in use.

PollKube provider

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) 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.

Please 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.

Global vs Route levels

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": "downstreamservice",
    "ServiceNamespace": "downstream-namespace"
  }
]
```

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.

The types of load balancer available are:

- **LeastConnection** tracks which services are dealing with requests and sends new requests to service with least existing requests. The algorithm state is not distributed across a cluster of Ocelot's.
- **RoundRobin** loops through available services and sends requests. The algorithm state is not distributed across a cluster of Ocelot's.
- **NoLoadBalancer** takes the first available service from config or service discovery.
- **CookieStickySessions** uses a cookie to stick all requests to a specific server. More info below.

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

Configuration

The following shows how to set up multiple downstream services for a Route using **ocelot.json** and then select the LeastConnection load balancer. This is the simplest way to get load balancing set up.

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

Service Discovery

The following shows how to set up a Route using service discovery then select the `LeastConnection` load balancer.

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

When this is set up Ocelot will lookup the downstream host and port from the service discover provider and load balance requests across any available services. If you add and remove services from the service discovery provider (Consul) then Ocelot should respect this and stop calling services that have been removed and start calling services that have been added.

CookieStickySessions Type

We have implemented a really basic sticky session type of load balancer. The scenario it is meant to support is you have a bunch of downstream servers that don't share session state, so if you get more than one request for one of these servers then it should go to the same box each time or the session state might be incorrect for the given user. This 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!

In order to set up `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": {
    "Type": "CookieStickySessions",
    "Key": "ASP.NET_SessionId",
    "Expiry": 1800000
  }
}
```

The `LoadBalancerOptions` are

- **Type** this needs to be `CookieStickySessions`
- **Key** this is the key of the cookie you wish to use for the sticky sessions
- **Expiry** this is how long in milliseconds you want to the session to be stuck for. Remember this refreshes on every request which is meant to mimick how sessions work usually.

If you have multiple Routes with the same `LoadBalancerOptions` then all of those Routes will use the same load balancer for there subsequent requests. This means the sessions will be stuck across Routes.

Please note that if you give more than one **DownstreamHostAndPort** or you are using a Service Discovery provider such as Consul and this returns more than one service then **CookieStickySessions** uses round robin to select the next server. This is hard coded at the moment but could be changed.

Custom Load Balancers

David Lievrouw implemented a way to provide Ocelot with custom load balancer in [PR 1155](#) (his [issue 961](#)).

In order to create and use a custom load balancer you can do the following. Below we setup a basic load balancing config and not the **Type** is CustomLoadBalancer which is the name of a class we will setup to do load balancing.

```
{
  "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": {
    "Type": "CustomLoadBalancer"
  }
}
```

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

```
public class CustomLoadBalancer : ILoadBalancer
{
    private readonly Func<Task<List<Service>>> _services;
    private readonly object _lock = new object();
    private int _last;

    public CustomLoadBalancer(Func<Task<List<Service>>> services)
    {
        _services = services;
    }

    public async Task<Response<ServiceHostAndPort>> Lease(HttpContext httpContext)
    {
        var services = await _services?.Invoke();
        lock (_lock)
        {
            if (_last >= services.Count)
                _last = 0;

            var next = services[_last++];
            return new OkResponse<ServiceHostAndPort>(next.HostAndPort);
        }
    }

    public void Release(ServiceHostAndPort hostAndPort) { }
}
```

Finally, you need to register this class with Ocelot.

We have used the most complex example below to show all of the data / types that can be passed into the factory that creates load balancers.

```
Func<IServiceProvider, DownstreamRoute, IServiceDiscoveryProvider, CustomLoadBalancer>␣  
↪loadBalancerFactoryFunc =  
    (serviceProvider, Route, serviceDiscoveryProvider) => new␣  
↪CustomLoadBalancer(serviceDiscoveryProvider.Get);  
  
services.AddOcelot()  
    .AddCustomLoadBalancer(loadBalancerFactoryFunc);
```

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

```
services.AddOcelot()  
    .AddCustomLoadBalancer<CustomLoadBalancer>();
```

There are numerous extension methods to add a custom load balancer and 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;
```

When you enable custom load balancers Ocelot looks up your load balancer by its class name when it decides if it should do 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 registered load balancer class then you will receive a [HTTP 500 Internal Server Error](#). If your load balancer factory throw an exception when Ocelot calls it, you will receive a [HTTP 500 Internal Server Error](#).

Remember, if you specify no load balancer in your config, Ocelot will not try and load balance.

Logging

Ocelot uses the standard 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 **RequestId** if it is configured.

There is a global [error handler middleware](#) that should catch any exceptions thrown and log them as errors.

Finally, if logging is set to Trace level, Ocelot will log starting, finishing and any middlewares that throw an exception which can be quite useful.

Request ID

The reason for not just using [bog standard](#) framework logging is that we could not work out how to override the **RequestId** that get's logged when setting **IncludeScopes** to `true` for logging settings. Nicely onto the next feature.

Every log record has these 2 properties:

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

As an `IOcelotLogger` interface object being injected to constructors of service classes, current default Ocelot logger (`OcelotLogger` class) reads these 2 properties from the `IRequestScopedDataRepository` interface object. Find out more about these properties and other details on the *Request ID* logging feature in the [Request ID](#) chapter.

Warning

If you are logging to MS [Console](#), you will get terrible performance. The team has had so many issues about performance issues with Ocelot and it is always logging level Debug, logging to [Console](#).

- **Warning!** Make sure you are logging to something proper in production environment!
- Use Error and Critical levels in production environment!
- Use Warning level in testing & staging environments!

These and other recommendations are below in the [Best Practices](#) section.

Best Practices

Microsoft Learn omplete reference: [Logging in .NET Core and ASP.NET Core](#)

Our recommendations to gain Ocelot best logging are the following.

First

Ensure minimum 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 using [Serilog](#) or the standard Microsoft providers, the logging configuration will be retrieved from this section.

```

.ConfigureAppConfiguration(_, config) =>
{
    config.AddJsonFile($"appsettings.{env.EnvironmentName}.json", false, false);
    // ...
})

```

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 one place only, like:

```

ConfigureLogging(logging =>
{
    logging.ClearProviders();
    logging.SetMinimumLevel(minLogLevel);
    logging.AddConsole(); // MS Console for Development and/or Testing environments only
})

```

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

Second

Ensure proper usage of minimum logging level for each environment: development, testing, production, etc. So, once again, read important notes of the [Warning](#) section!

Third

Ocelot's logging has been improved in 22.0 version: 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 aka `string.Format`, or string value is generated by [string interpolation](#) expression, it is recommended to call the log method using anonymous delegate via an `=>` expression function:

```

Logger.LogDebug(() => $"downstream templates are {string.Join(", ", response.Data.Route.
    ↳ DownstreamRoute.Select(r => r.DownstreamPathTemplate.Value))}");

```

otherwise a constant string is sufficient

```

Logger.LogDebug("My const string");

```

Performance Review

Ocelot's logging performance has been improved in version 22.0 (see PR 1745). These changes were requested as part of issue 1744 after team's [discussion](#).

Top Logging Performance?

Here is a quick recipe for your Production environment! You need to ensure the minimal level is `Critical` or `None`. Nothing more! For sure, having top logging performance means having less log records written by logging provider. So, logs should be pretty empty.

Anyway, during the first time after a version release to production, we recommend to watch the system and current version app behavior by specifying `Error` minimum level. If release engineer will ensure stability of the version in production then minimum level can be increased to `Critical` or `None` to gain top performance. Technically this will switch off the logging feature at all.

Run Benchmarks

We have 2 types of benchmarks currently

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

1. Open PowerShell or Command Prompt console
2. Build Ocelot solution in Release mode: `dotnet build --configuration Release`
3. Go to `test\Ocelot.Benchmarks\bin\Release\` folder.
4. Choose .NET version changing the folder, for example to `net8.0`
5. Run **Ocelot.Benchmarks.exe**: `.\Ocelot.Benchmarks.exe`
6. Run `SerilogBenchmarks` or `MsLoggerBenchmarks` by pressing appropriate number of a benchmark: 5 or 6, + Enter
7. Wait for 3+ minutes to complete benchmark, and get final results.
8. Read and analyze your benchmark session results.

Indicators

To be developed...

Method Transformation

Ocelot allows the user to change the HTTP request method that will be used when making a request to a downstream service.

This achieved by setting the following Route configuration:

```
{
  "UpstreamPathTemplate": "{url}",
  "DownstreamPathTemplate": "{url}",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 54321 }
  ],
  "UpstreamHttpMethod": [ "Get" ],
  "DownstreamHttpMethod": "POST" // !
}
```

The key property here is **DownstreamHttpMethod** which is set as POST and the Route will only match on GET as set by **UpstreamHttpMethod**.

This feature can be useful when interacting with downstream APIs that only support POST and you want to present some kind of RESTful interface.

Middleware Injection

Warning, use with caution! If you are seeing any exceptions or strange behavior in your middleware pipeline and you are using any of the following. Remove them and try again!

When setting up Ocelot in your **Startup.cs** you can provide some additional middleware and override middleware. This is done as follows:

```
var configuration = new OcelotPipelineConfiguration
{
    PreErrorResponderMiddleware = async (context, next) =>
    {
        await next.Invoke();
    }
};
app.UseOcelot(configuration);
```

In the example above the provided function will run before the first piece of Ocelot middleware. This allows a user to supply any behaviors they want before and after the Ocelot pipeline has run. This means you can break everything, so use at your own pleasure!

The user can set functions against the following (see more in the [OcelotPipelineConfiguration](#) class):

- **PreErrorResponderMiddleware** injection is already explained above.
- **PreAuthenticationMiddleware** injection allows the user to run pre authentication logic and then call Ocelot authentication middleware.
- **AuthenticationMiddleware** overrides Ocelot authentication middleware.¹

¹ **Warning, use mentioned middlewares overrides with caution!** Overridden middleware removes the default implementation! If you are seeing any exceptions or strange behavior in your middleware pipeline, remove overridden middlewares and try again!

- `PreAuthorizationMiddleware` injection allows the user to run pre authorization logic and then call Ocelot authorization middleware.
- `AuthorizationMiddleware` overrides Ocelot's authorization middleware. [Page 52, 1](#)
- `PreQueryStringBuilderMiddleware` injection allows the user to manipulate the query string on the http request before it is passed to Ocelot request creator.

Obviously you can just add mentioned Ocelot middleware overrides as normal before the call to `app.UseOcelot()`. It cannot be added after as Ocelot does not call the next Ocelot middleware overrides based on specified middleware configuration. So, the next called middlewares **will not** affect Ocelot configuration.

ASP.NET Core Middlewares and Ocelot Pipeline Builder

Ocelot pipeline is a part of entire [ASP.NET Core Middlewares](#) conveyor aka app pipeline. The [BuildOcelotPipeline](#) method encapsulates Ocelot pipeline. The last middleware in the `BuildOcelotPipeline` method is `HttpRequesterMiddleware` that calls the next middleware, if added to the pipeline.

The internal `HttpRequesterMiddleware` is part of the pipeline but it is private and cannot be overridden, since this middleware does not belong to the list of [user's public ones](#) that can be overridden! So, this is the [last middleware](#) of the entire Ocelot and ASP.NET pipeline, and it handles non-user operation. The last user (public) middleware that can be overridden is `PreQueryStringBuilderMiddleware` being read from the [pipeline configuration object](#), see [previous section](#).

Considering that `PreQueryStringBuilderMiddleware` and `HttpRequesterMiddleware` are the last user and system middlewares, there are no other middlewares in the pipeline at all. But you can still extend the ASP.NET pipeline, as shown in the following code:


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

But we do not recommend adding this custom middleware before or after calling of `UseOcelot()` as it affects the stability of the entire pipeline and has not been tested. Such kind of custom pipeline building is out of the Ocelot pipeline model and the quality of the solution is at your own risk.

Finally, do not get confused about the distinction between system (private, non-overridden) and user (public, overridden) middlewares. Private middlewares are hidden and cannot be overridden, but the entire ASP.NET pipeline can still be extended. The [public middlewares](#) are fully customizable and can be overridden.

Future

The community shows an interest in adding more overridden middlewares. One of such request is [PR 1497](#), and possibly it will be included in a next upcoming [release](#).

Anyway, in your opinion, if current overridden middlewares do not provide enough pipeline flexibility, you can open new topic in [Discussions](#) space of the repository. 

Quality of Service

Label: [QoS](#)

Ocelot supports one QoS capability at the current time. You can set on a per Route basis if you want to use a circuit breaker when making requests to a downstream service. This uses an awesome .NET library called [Polly](#), check them out in [official repository](#).

The first thing you need to do if you want to use the *Administration* API is bring in the relevant NuGet [package](#):

```
Install-Package Ocelot.Provider.Polly
```

Then in your `ConfigureServices` method to add [Polly](#) services we must call the `AddPolly()` extension of the `OcelotBuilder` being returned by `AddOcelot()`¹ like below:

```
services.AddOcelot()  
    .AddPolly();
```

Then add the following section to a Route configuration:

```
"QoSOptions": {  
  "ExceptionsAllowedBeforeBreaking": 3,  
  "DurationOfBreak": 1000,  
  "TimeoutValue": 5000  
}
```

- You must set a number equal or greater than 2 against **ExceptionsAllowedBeforeBreaking** for this rule to be implemented.²
- **DurationOfBreak** means the circuit breaker will stay open for 1 second after it is tripped.
- **TimeoutValue** means if a request takes more than 5 seconds, it will automatically be timed out.

You can set the **TimeoutValue** in isolation of the **ExceptionsAllowedBeforeBreaking** and **DurationOfBreak** options:

```
"QoSOptions": {  
  "TimeoutValue": 5000  
}
```

There is no point setting the other two in isolation as they affect each other!

Defaults

If you do not add a QoS section, QoS will not be used, however Ocelot will default to a **90** seconds timeout on all downstream requests. If someone needs this to be configurable, open an issue.^{Page 54, 2}

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

² If something doesn't work or you get stuck, please review current [QoS issues](#) filtering by label.

Polly v7 vs v8

Important changes in version 23.2:³

- With **Polly** version 8+, the `ExceptionsAllowedBeforeBreaking` value must be equal to or greater than **2**!
- The `AddPolly` method has been migrated from v7 policy wrappers to v8 resilience pipelines. Consequently, it now exhibits different behavior based on v8 pipelines.

If you prefer not to modify your settings, you can continue using **Polly** v7 as follows:

```
services.AddOcelot()
    .AddPollyV7();
```

Note: Support for **Polly** v7 will be removed in a future version. We recommend avoiding this method (which is tagged as `Obsolete`) unless absolutely necessary.

Extensibility^{Page 55, 3}

If you want to use your `ResiliencePipeline<T>` provider, you can use the following syntax:

```
services.AddOcelot()
    .AddPolly<MyProvider>();
// MyProvider should implement IPollyQoSResiliencePipelineProvider<HttpResponseMessage>
// Note: you can use standard provider PollyQoSResiliencePipelineProvider
```

If, in addition, you want to use your own `DelegatingHandler`, you can use the following syntax:

```
services.AddOcelot()
    .AddPolly<MyProvider>(MyQoSDelegatingHandlerDelegate);
// MyProvider should implement IPollyQoSResiliencePipelineProvider<HttpResponseMessage>
// Note: you can use standard provider PollyQoSResiliencePipelineProvider
// MyQoSDelegatingHandlerDelegate is a delegate use to get a DelegatingHandler
```

And finally, if you want to define your own set of exceptions to map, you can use the following syntax:

```
services.AddOcelot()
    .AddPolly<MyProvider>(MyErrorMapping);
// MyProvider should implement IPollyQoSResiliencePipelineProvider<HttpResponseMessage>
// Note: you can use standard provider PollyQoSResiliencePipelineProvider

// MyErrorMapping is a Dictionary<Type, Func<Exception, Error>>, eg:
private static readonly Dictionary<Type, Func<Exception, Error>> MyErrorMapping = new()
{
    {typeof(TaskCanceledException), CreateError},
    {typeof(TimeoutRejectedException), CreateError},
    {typeof(BrokenCircuitException), CreateError},
    {typeof(BrokenCircuitException<HttpResponseMessage>), CreateError},
};
private static Error CreateError(Exception e) => new RequestTimedOutError(e);
```

³ We upgraded **Polly** version from v7.x to v8.x! The *Extensibility 3* feature was requested in issue 1875 and delivered by PR 1914 as a part of version 23.2.

Rate Limiting

Ocelot Own Implementation

Ocelot supports rate limiting of upstream requests so that your downstream services do not become overloaded.

The authors of this feature were inspired by [@catcherwong](#) article to finally write this documentation. This feature was added by [@geffzhang](#) on GitHub! Thanks very much!

To get rate limiting working for a Route you need to add the following JSON to it:

```
"RateLimitOptions": {
  "ClientWhitelist": [],
  "EnableRateLimiting": true,
  "Period": "1s",
  "PeriodTimespan": 1,
  "Limit": 1
}
```

- **ClientWhitelist** - This is an array that contains the whitelist of the client. It means that the client in this array will not be affected by the rate limiting.
- **EnableRateLimiting** - This value specifies enable endpoint rate limiting.
- **Period** - This value specifies the period that the limit applies to, such as 1s, 5m, 1h, 1d and so on. If you make more requests in the period than the limit allows then you need to wait for **PeriodTimespan** to elapse before you make another request.
- **PeriodTimespan** - This value specifies that we can retry after a certain number of seconds.
- **Limit** - This value specifies the maximum number of requests that a client can make in a defined period.

You can also set the following in the **GlobalConfiguration** part of **ocelot.json**:

```
"GlobalConfiguration": {
  "BaseUrl": "https://api.mybusiness.com",
  "RateLimitOptions": {
    "DisableRateLimitHeaders": false,
    "QuotaExceededMessage": "Customize Tips!",
    "HttpStatusCode": 123,
    "ClientIdHeader": "Test"
  }
}
```

- **DisableRateLimitHeaders** - This value specifies whether X-Rate-Limit and Retry-After headers are disabled.
- **QuotaExceededMessage** - This value specifies the exceeded message.
- **HttpStatusCode** - This value specifies the returned HTTP status code when rate limiting occurs.
- **ClientIdHeader** - Allows you to specify the header that should be used to identify clients. By default it is ClientId

Future and ASP.NET Core Implementation

The Ocelot team considers to redesign *Rate Limiting* feature, because of [Announcing Rate Limiting for .NET](#) by Brennan Conroy on July 13th, 2022. There is no decision at the moment, and the old version of the feature is included as a part of release 20.0 for .NET 7.

See more about new feature being added into ASP.NET Core 7.0 release:

- [RateLimiter Class](#), since ASP.NET Core 7.0
- [System.Threading.RateLimiting](#) NuGet package
- [Rate limiting middleware in ASP.NET Core](#) article by Arvin Kahbazi, Maarten Balliauw, and Rick Anderson

However, it makes sense to keep the old implementation as a Ocelot built-in native feature, but we are going to migrate to the new Rate Limiter from `Microsoft.AspNetCore.RateLimiting` namespace.

Please, share your opinion to us in the [Discussions](#) space of the repository. 

Request Aggregation^{Page 57, 1}

Ocelot allows you to specify Aggregate Routes that compose multiple normal Routes and map their responses into one object. This is usually where you have a client that is making multiple requests to a server where it could just be one. This feature allows you to start implementing back-end for a front-end (BFF) type architecture with Ocelot.¹

In order to set this up you must do something like the following in your `ocelot.json`. Here we have specified two normal Routes and each one has a **Key** property. We then specify an Aggregate that composes the two Routes using their keys in the **RouteKeys** list and says then we have the **UpstreamPathTemplate** which works like a normal Route. Obviously you cannot have duplicate **UpstreamPathTemplates** between **Routes** and **Aggregates**. You can use all of Ocelot's normal Route options apart from **RequestIdKey** (explained in [Gotchas](#) below).

Basic Expecting JSON from Downstream Services

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

(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 PRs 1826 and 1462.

(continued from previous page)

```

    { "Host": "localhost", "Port": 51882 }
  ],
  "Key": "Tom"
},
"Aggregates": [
  {
    "UpstreamPathTemplate": "/",
    "RouteKeys": [ "Tom", "Laura" ]
  }
]
}

```

You can also set **UpstreamHost** and **RouteIsCaseSensitive** in the Aggregate configuration. These behave the same as any other Routes.

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

```

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

```

At the moment the aggregation is very simple. Ocelot just gets the response from your downstream service and sticks it into a JSON dictionary as above. With the Route key being the key of the dictionary and the value the response body from your downstream service. You can see that the object is just JSON without any pretty spaces etc.

Note, all headers will be lost from the downstream services response.

Ocelot will always return content type `application/json` with an aggregate request.

If you downstream services return a `404 Not Found`, the aggregate will just return nothing for that downstream service. It will not change the aggregate response into a `404` even if all the downstreams return a `404`.

Use Complex Aggregation

Imagine you'd like to use aggregated queries, but you don't know all the parameters of your queries. You first need to call an endpoint to obtain the necessary data, for example a user's id, and then return the user's details.

Let's say we have an endpoint that returns a series of comments with references to various users or threads. The author of the comments is referenced by his Id, but you'd like to return all the details about the author.

Here, you could use aggregation to get 1) all the comments, 2) attach the author details. In fact there are 2 endpoints that are called, but for the 2nd, you dynamically replace the user's Id in the route to obtain the details.

In concrete terms:

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

This functionality is still in its early stages, but it does allow you to search for data based on an initial request.

To perform the mapping, you need to use **AggregateRouteConfig**:

```

new AggregateRouteConfig
{
    RouteKey = "UserDetails",
    JsonPath = "$[*].authorId",

```

(continues on next page)

(continued from previous page)

```
Parameter = "userId"
};
```

RouteKey is used as a reference for the route, **JsonPath** indicates where the parameter you are interested in is located in the first request response body and **Parameter** tells us that the value for authorId should be used for the request parameter userId.

Register Your Own Aggregators

Ocelot started with just the basic request aggregation and since then we have added a more advanced method that let's the user take in the responses from the downstream services and then aggregate them into a response object. The **ocelot.json** setup is pretty much the same as the basic aggregation approach apart from you need to add an **Aggregator** property like below:

```
{
  "Routes": [
    {
      "UpstreamHttpMethod": [ "Get" ],
      "UpstreamPathTemplate": "/laura",
      "DownstreamPathTemplate": "/",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 51881 }
      ],
      "Key": "Laura" // <--
    },
    {
      "UpstreamHttpMethod": [ "Get" ],
      "UpstreamPathTemplate": "/tom",
      "DownstreamPathTemplate": "/",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 51882 }
      ],
      "Key": "Tom" // <--
    }
  ],
  "Aggregates": [
    {
      "UpstreamPathTemplate": "/",
      "RouteKeys": [
        "Tom",
        "Laura"
      ],
      "Aggregator": "FakeDefinedAggregator"
    }
  ]
}
```

Here we have added an aggregator called **FakeDefinedAggregator**. Ocelot is going to look for this aggregator when it tries to aggregate this Route.

In order to make the aggregator available we must add the `FakeDefinedAggregator` to the `OcelotBuilder` being returned by `AddOcelot()`² like below:

```
services
    .AddOcelot()
    .AddSingletonDefinedAggregator<FakeDefinedAggregator>();
```

Now when Ocelot tries to aggregate the Route above it will find the `FakeDefinedAggregator` in the container and use it to aggregate the Route. Because the `FakeDefinedAggregator` is registered in the container you can add any dependencies it needs into the container like below:

```
services.AddSingleton<FooDependency>();
// ...
services.AddOcelot()
    .AddSingletonDefinedAggregator<FooAggregator>();
```

In this example `FooAggregator` takes a dependency on `FooDependency` and it will be resolved by the container.

In addition to this Ocelot lets you add transient aggregators like below:

```
services
    .AddOcelot()
    .AddTransientDefinedAggregator<FakeDefinedAggregator>();
```

In order to make an Aggregator you must implement this interface:

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

With this feature you can pretty much do whatever you want because the `HttpContext` objects contain the results of all the aggregate requests.

Please note, if the `HttpClient` throws an exception when making a request to a Route in the aggregate then you will not get a `HttpContext` for it, but you would for any that succeed. If it does throw an exception, this will be logged.

Below is an example of an aggregator that you could implement for your solution:

```
public class FakeDefinedAggregator : 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>();
```

(continues on next page)

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

(continued from previous page)

```

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

```

Gotchas

You cannot use Routes with specific **RequestIdKeys** as this would be crazy complicated to track.

Aggregation only supports the GET HTTP verb.

Request ID

aka **Correlation ID** or `HttpContext.TraceIdentifier`

Ocelot supports a client sending a *request ID* in the form of a header. If set, Ocelot will use the **RequestId** for logging as soon as it becomes available in the middleware pipeline. Ocelot will also forward the *RequestId* with the specified header to the downstream service.

You can still get the ASP.NET Core *Request ID* in the logs if you set **IncludeScopes** true in your logging config.

In order to use the *Request ID* feature you have two options.

Global

In your **ocelot.json** set the following in the **GlobalConfiguration** section. This will be used for all requests into Ocelot.

```

"GlobalConfiguration": {
  "RequestIdKey": "OcRequestId"
}

```

We recommend using the **GlobalConfiguration** unless you really need it to be Route specific.

Route

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

```
"RequestIdKey": "OcRequestId"
```

Once Ocelot has identified the incoming requests matching Route object it will set the *request ID* based on the Route configuration.

Gotcha

This can lead to a small gotcha. If you set a **GlobalConfiguration**, it is possible to get one *request ID* until the Route is identified and then another after that because the *request ID* key can change. This is by design and is the best solution we can think of at the moment. In this case the OcelotLogger will show the *request ID* and previous *request ID* in the logs.

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

```
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message: ocelot_
↳ pipeline started,
dbug: Ocelot.DownstreamRouteFinder.Middleware.DownstreamRouteFinderMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message: upstream url_
↳ path is {upstreamUrlPath},
dbug: Ocelot.DownstreamRouteFinder.Middleware.DownstreamRouteFinderMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message: downstream_
↳ template is {downstreamRoute.Data.Route.DownstreamPath},
dbug: Ocelot.RateLimit.Middleware.ClientRateLimitMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message:_
↳ EndpointRateLimiting is not enabled for Ocelot.Values.PathTemplate,
dbug: Ocelot.Authorization.Middleware.AuthorizationMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: /posts/{postId} route does not_
↳ require user to be authorized,
dbug: Ocelot.DownstreamUrlCreator.Middleware.DownstreamUrlCreatorMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: downstream url is
↳ {downstreamUrl.Data.Value},
dbug: Ocelot.Request.Middleware.HttpRequestBuilderMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: setting upstream request,
dbug: Ocelot.Requester.Middleware.HttpRequesterMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: setting http response message,
dbug: Ocelot.Responder.Middleware.ResponderMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: no pipeline errors, setting and_
↳ returning completed response,
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: ocelot pipeline finished,
```

And more practical example from secret production environment in Switzerland:

```
warn: Ocelot.DownstreamRouteFinder.Middleware.DownstreamRouteFinderMiddleware[0]
      requestId: 0HMVD33IIJRFR:00000001, previousRequestId: no previous request id,_
↳ message: DownstreamRouteFinderMiddleware setting pipeline errors._
↳ IDownstreamRouteFinder returned Error Code: UnableToFindDownstreamRouteError Message:_
↳ Failed to match Route configuration for upstream path: /, verb: GET.
```

(continues on next page)

(continued from previous page)

```
warn: Ocelot.Responder.Middleware.ResponderMiddleware[0]
      requestId: 0HMVD33IIJRFR:000000001, previousRequestId: no previous request id,
  ↳ message: Error Code: UnableToFindDownstreamRouteError Message: Failed to match Route,
  ↳ configuration for upstream path: /, verb: GET. errors found in ResponderMiddleware.
  ↳ Setting error response for request path:/, request method: GET
```

Curious?

Request ID is a part of big *Logging* feature.

Every log record has these 2 properties:

- **RequestId** represents ID of the current request as plain string, for example 0HMVD33IIJRFR:000000001
- **PreviousRequestId** represents ID of the previous request

As an *IOcelotLogger* interface object being injected to constructors of service classes, current default Ocelot logger (the *OcelotLogger* class) reads these 2 properties from the *IRequestScopedDataRepository* interface object.

Routing

Ocelot's primary functionality is to take incoming HTTP requests and forward them on to a downstream service. Ocelot currently only supports this in the form of another HTTP request (in the future this could be any transport mechanism).

Ocelot describes the routing of one request to another as a *Route*. In order to get anything working in Ocelot you need to set up a *Route* in the configuration.

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

To configure a *Route* you need to add one to the *Routes* JSON array.

```
{
  "UpstreamHttpMethod": [ "Put", "Delete" ],
  "UpstreamPathTemplate": "/posts/{postId}",
  "DownstreamPathTemplate": "/api/posts/{postId}",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 80 }
  ]
}
```

The **DownstreamPathTemplate**, **DownstreamScheme** and **DownstreamHostAndPorts** define the URL that a request will be forwarded to.

The **DownstreamHostAndPorts** property is a collection that defines the host and port of any downstream services that you wish to forward requests to. Usually this will just contain a single entry, but sometimes you might want to load balance requests to your downstream services and Ocelot allows you add more than one entry and then select a load balancer.

The **UpstreamPathTemplate** property is the URL that Ocelot will use to identify which **DownstreamPathTemplate** to use for a given request. The **UpstreamHttpMethod** is used so Ocelot can distinguish between requests with different HTTP verbs to the same URL. You can set a specific list of HTTP methods or set an empty list to allow any of them.

Placeholders

In Ocelot you can add placeholders for variables to your Templates in the form of `{something}`. The placeholder variable needs to be present in both the **DownstreamPathTemplate** and **UpstreamPathTemplate** properties. When it is Ocelot will attempt to substitute the value in the **UpstreamPathTemplate** placeholder into the **DownstreamPathTemplate** for each request Ocelot 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 any path + query string combinations to the downstream service after the path `/api`.

Note, the default Routing configuration is case insensitive!

In order to change this you can specify on a per Route basis the following setting:

```
"RouteIsCaseSensitive": true
```

This means that when Ocelot tries to match the incoming upstream URL with an upstream template the evaluation will be case sensitive.

Empty Placeholders^{Page 64, 1}

This is a special edge case of *Placeholders*, where the value of the placeholder is simply an empty string `""`.

For example, **Given a route**:

```
{
  "UpstreamPathTemplate": "/invoices/{url}",
  "DownstreamPathTemplate": "/api/invoices/{url}",
}
```

Then, it works correctly when `{url}` is specified: `/invoices/123 /api/invoices/123`.

And then, there are two edge cases with empty placeholder value:

- Also, it works when `{url}` is empty. We would expect upstream path `/invoices/` to route to downstream path `/api/invoices/`
- Moreover, it should work when omitting last slash. We also expect upstream `/invoices` to be routed to downstream `/api/invoices`, which is intuitive to humans

¹ “*Empty Placeholders I*” feature is available starting in version 23.0, see issue 748 and the 23.0 release notes for details.

Catch All

Ocelot's routing also supports a *Catch All* style routing where the user can specify that they want to match all traffic.

If you set up your config like below, all requests will be proxied straight through. The placeholder `{url}` name is not significant, any name will work.

```
{
  "UpstreamHttpMethod": [ "Get" ],
  "UpstreamPathTemplate":("/{url}",
  "DownstreamPathTemplate":("/{url}",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 80 }
  ]
}
```

The *Catch All* has a lower priority than any other Route. If you also have the Route below in your config then Ocelot would match it before the *Catch All*.

```
{
  "UpstreamHttpMethod": [ "Get" ],
  "UpstreamPathTemplate": "/",
  "DownstreamPathTemplate": "/",
  "DownstreamScheme": "https",
  "DownstreamHostAndPorts": [
    { "Host": "10.0.10.1", "Port": 80 }
  ]
}
```

Upstream Host^{Page 65, 2}

This feature allows you to have Routes based on the *upstream host*. This works by looking at the Host header the client has used and then using this as part of the information we use to identify a Route.

In order to use this feature please add the following to your config:

```
{
  "UpstreamHost": "somedomain.com"
}
```

The Route above will only be matched when the Host header value is `somedomain.com`.

If you do not set **UpstreamHost** on a Route then any Host header will match it. This means that if you have two Routes that are the same, apart from the **UpstreamHost**, where one is null and the other set Ocelot will favour the one that has been set.

² “*Upstream Host 2*” feature was requested as part of [issue 216](#).

Priority

You can define the order you want your Routes to match the Upstream `HttpRequest` by including a **Priority** property in `ocelot.json`. See [issue 270](#) for reference.

```
{
  "Priority": 0
}
```

0 is the lowest priority, Ocelot will always use 0 for `/catchAll` Routes and this is still hardcoded. After that you are free to set 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 you make a request into Ocelot on `/goods/delete`, Ocelot will match `/goods/delete` Route. Previously it would have matched `/goods/{catchAll}`, because this is the first Route in the list!

Query String Placeholders

In addition to URL path *Placeholders* Ocelot is able to forward query string parameters with their processing in the form of `{something}`. Also, the query parameter placeholder needs to be present in both the **DownstreamPathTemplate** and **UpstreamPathTemplate** properties. Placeholder replacement works bi-directionally between path and query strings, with some *restrictions* on usage.

Path to Query String direction

Ocelot allows you to specify a query string as part of the **DownstreamPathTemplate** like the example below:

```
{
  "UpstreamPathTemplate": "/api/units/{subscription}/{unit}/updates",
  "DownstreamPathTemplate": "/api/subscriptions/{subscription}/updates?unitId={unit}",
}
```

In this example Ocelot will use the value from the `{unit}` placeholder in the upstream path template and add it to the downstream request as a query string parameter called `unitId`! Make sure you name the placeholder differently due to *restrictions* on usage.

Query String to Path direction

Ocelot will also allow you to put query string parameters in the **UpstreamPathTemplate** so you can match certain queries to certain services:

```
{
  "UpstreamPathTemplate": "/api/subscriptions/{subscriptionId}/updates?unitId={uid}",
  "DownstreamPathTemplate": "/api/units/{subscriptionId}/{uid}/updates",
}
```

In this example Ocelot will only match requests that have a matching URL path and the query string starts with `unitId=something`. You can have other queries after this but you must start with the matching parameter. Also Ocelot will swap the `{uid}` parameter from the query string and use it in the downstream request path. Note, the best practice is giving different placeholder name than the name of query parameter due to *restrictions* on usage.

Catch All Query String

Ocelot's routing also supports a *Catch All* style routing to forward all query string parameters. The placeholder `{everything}` name does not matter, any name will work.

```
{
  "UpstreamPathTemplate": "/contracts?{everything}",
  "DownstreamPathTemplate": "/apipath/contracts?{everything}",
}
```

This entire query string routing feature is very useful in cases where the query string should not be transformed but rather routed without any changes, such as OData filters and etc (see issue 1174).

Note, the `{everything}` placeholder can be empty while catching all query strings, because this is a part of the *Empty Placeholders 1* feature!^{Page 64, 1} Thus, upstream paths `/contracts?` and `/contracts` are routed to downstream path `/apipath/contracts`, which has no query string at all.

Restrictions on use

The query string parameters are ordered and merged to produce the final downstream URL. This is necessary because the `DownstreamUrlCreatorMiddleware` needs to have some control when replacing placeholders and merging duplicate parameters. So, even if your parameter is presented as the first parameter in the upstream, then in the final downstream URL the said query parameter will have a different position. But this doesn't seem to break anything in the downstream API.

Because of parameters merging, special ASP.NET API *model binding* for arrays is not supported if you use array items representation like `selectedCourses=1050&selectedCourses=2000`. This query string will be merged as `selectedCourses=1050` in downstream URL. So, array data will be lost! Make sure upstream clients generate correct query string for array models like `selectedCourses[0]=1050&selectedCourses[1]=2000`. To understand array model bindings, see *Bind arrays and string values from headers and query strings* docs.

Warning! Query string placeholders have naming restrictions due to `DownstreamUrlCreatorMiddleware` implementations. On the other hand, it gives you the flexibility to control whether the parameter is present in the final downstream URL. Here are two user scenarios.

- User wants to save the parameter after replacing the placeholder (see issue 473). To do this you need to use the following template definition:

```
{
  "UpstreamPathTemplate": "/path/{serverId}/{action}",
  "DownstreamPathTemplate": "/path2/{action}?server={serverId}"
}
```

So, {serverId} placeholder and server parameter **names are different!** Finally, the server parameter is kept.

- User wants to remove old parameter after replacing placeholder (see issue 952). To do this you need to use the same names:

```
{
  "UpstreamPathTemplate": "/users?userId={userId}",
  "DownstreamPathTemplate": "/persons?personId={userId}"
}
```

So, both {userId} placeholder and userId parameter **names are the same!** Finally, the userId parameter is removed.

Security Options^{Page 68, 3}

Ocelot allows you to manage multiple patterns for allowed/blocked IPs using the [IPAddressRange](#) package with [MPL-2.0 License](#).

This feature is designed to allow greater IP management in order to include or exclude a wide IP range via CIDR notation or IP range. The current patterns managed are the following:

- 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 Range with subnet: 192.168.1.0/255.255.255.0
- CIDR: 192.168.1.0/24
- CIDR for IPv6: fe80::/10
- The allowed/blocked lists are evaluated during configuration loading
- The **ExcludeAllowedFromBlocked** property is intended to provide the ability to specify a wide range of blocked IP addresses and allow a subrange of IP addresses. Default value: false
- The absence of a property in **SecurityOptions** is allowed, it takes the default value.

```
{
  "SecurityOptions": {
    "IPBlockedList": [ "192.168.0.0/23" ],
    "IPAllowedList": [ "192.168.0.15", "192.168.1.15" ],
    "ExcludeAllowedFromBlocked": true
  }
}
```

³ “Security Options 3” feature was requested as part of issue 628 (of 12.0.1 version), then redesigned and improved by issue 1400, and published in version 20.0 docs.

Dynamic Routing^{Page 69, 4}

The idea is to enable dynamic routing when using a *Service Discovery* provider so you don't have to provide the Route config. See the *Dynamic Routing* docs if this sounds interesting to you.

Service Discovery

Ocelot allows you to specify a *service discovery* provider and will use this to find the host and port for the downstream service to which Ocelot forwards the request. At the moment this is only supported in the **GlobalConfiguration** section, which means the same *service discovery* provider will be used for all Routes for which you specify a *ServiceName* at Route level.

Consul

Namespace: `Ocelot.Provider.Consul`

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

```
Install-Package Ocelot.Provider.Consul
```

Then add the following to your `ConfigureServices` method:

```
services.AddOcelot()
    .AddConsul();
```

Currently there are 2 types of Consul *service discovery* providers: `Consul` and `PollConsul`. The default provider is `Consul`, which means that if `ConsulProviderFactory` cannot read, understand, or parse the **Type** property of the `ServiceProviderConfiguration` object, then a `Consul` provider instance is created by the factory.

Explore these types of providers and understand the differences in the subsections below.

Consul Provider Type

Class: `Ocelot.Provider.Consul.Consul`

The following is required in the `GlobalConfiguration`. The **ServiceDiscoveryProvider** property is required, and if you do not specify a host and port, the Consul default ones will be used.

Please note the **Scheme** option defaults to HTTP. It was added in [PR 1154](#). It defaults to HTTP to not introduce a breaking change.

```
"ServiceDiscoveryProvider": {
  "Scheme": "https",
  "Host": "localhost",
  "Port": 8500,
  "Type": "Consul"
}
```

⁴ “Dynamic Routing 4” feature was requested as part of [issue 340](#). Complete reference: *Dynamic Routing*.

In the future we can add a feature that allows Route specific configuration.

In order to tell Ocelot a Route is to use the *service discovery* provider for its host and port you must add the **ServiceName** and load balancer you wish to use when making requests downstream. At the moment Ocelot has a [RoundRobin](#) and [LeastConnection](#) algorithms you can use. If no load balancer is specified, Ocelot will not load balance requests.

```
{
  "ServiceName": "product",
  "LoadBalancerOptions": {
    "Type": "LeastConnection"
  }
}
```

When this is set up Ocelot will lookup the downstream host and port from the *service discovery* provider and load balance requests across any available services.

PollConsul Provider Type

Class: `Ocelot.Provider.Consul.PollConsul`

A lot of people have asked the team to implement a feature where Ocelot *polls Consul* for latest service information rather than per request. If you want to *poll Consul* for the latest services rather than per request (default behaviour) then you need to set the following configuration:

```
"ServiceDiscoveryProvider": {
  "Host": "localhost",
  "Port": 8500,
  "Type": "PollConsul",
  "PollingInterval": 100
}
```

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

Please note, there are tradeoffs here. If you *poll Consul* 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 Consul per request (as sidecar agent). If you are calling a remote Consul agent then polling will be a good performance improvement.

Service Definition

Your services need to be added to Consul something like below (C# style but hopefully this make sense)... The only important thing to note is not to add `http` or `https` to the `Address` field. We have been contacted before about not accepting scheme in `Address`. After reading [this](#) we do not think the scheme should be in there.

In C#

```
new AgentService()
{
    Service = "some-service-name",
    Address = "localhost",
    Port = 8080,
    ID = "some-id",
}
```

Or, in JSON

```
"Service": {
  "ID": "some-id",
  "Service": "some-service-name",
  "Address": "localhost",
  "Port": 8080
}
```

ACL Token

If you are using [ACL](#) with Consul, Ocelot supports adding the `X-Consul-Token` header. In order so this to work you must add the additional property below:

```
"ServiceDiscoveryProvider": {
  "Host": "localhost",
  "Port": 8500,
  "Type": "Consul",
  "Token": "footoken"
}
```

Ocelot will add this token to the Consul client that it uses to make requests and that is then used for every request.

Eureka

This feature was requested as part of [issue 262](#) to add support for [Netflix Eureka](#) service discovery provider. The main reason for this is it is a key part of [Steeltoe](#) which is something to do with [Pivotal](#)! Anyway enough of the background.

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

```
Install-Package Ocelot.Provider.Eureka
```

Then add the following to your `ConfigureServices` method.

```
s.AddOcelot().AddEureka();
```

Then in order to get this working add the following to `ocelot.json`:

```
"ServiceDiscoveryProvider": {
  "Type": "Eureka"
}
```

And following the guide [here](#) you may also need to add some stuff to `appsettings.json`. For example the JSON below tells the Steeltoe / Pivotal services where to look for the service discovery server and if the service should register with it:

```
"eureka": {
  "client": {
    "serviceUrl": "http://localhost:8761/eureka/",
    "shouldRegisterWithEureka": false,
    "shouldFetchRegistry": true
  }
}
```

If **shouldRegisterWithEureka** is false then **shouldFetchRegistry** will default to **true**, so you need not it explicitly but left it in there.

Ocelot will now register all the necessary services when it starts up and if you have the JSON above will register itself with Eureka. One of the services polls Eureka every 30 seconds (default) and gets the latest service state and persists this in memory. When Ocelot asks for a given service it is retrieved from memory so performance is not a big problem.

Ocelot will use the scheme (**http**, **https**) set in Eureka if these values are not provided in **ocelot.json**

Dynamic Routing

This feature was requested in [issue 340](#). The idea is to enable dynamic routing when using a service discovery provider (see that section of the docs for more info). In this mode Ocelot will use the first segment of the upstream path to lookup the downstream service with the service discovery provider.

An example of this would be calling Ocelot with a URL like **https://api.mywebsite.com/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 returns a service, Ocelot will request it on whatever host and port comes back from Consul plus the remaining path segments in this case **products** thus making the downstream call **http://hostfromconsul:portfromconsul/products**. Ocelot will append any query string to the downstream URL as normal.

Note, in order to enable dynamic routing you need to have **0** Routes in your config. At the moment you cannot mix dynamic and configuration Routes. In addition to this you need to specify the Service Discovery provider details as outlined above and the downstream **http/https** scheme as **DownstreamScheme**.

In addition to that you can set **RateLimitOptions**, **QoSOptions**, **LoadBalancerOptions** and **HttpHandlerOptions**, **DownstreamScheme** (You might want to call Ocelot on https but talk to private services over http) that will be applied to all of the dynamic Routes.

The config might look something like:

```
{
  "Routes": [],
  "Aggregates": [],
  "GlobalConfiguration": {
    "RequestIdKey": null,
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 8500,
      "Type": "Consul",
      "Token": null,
      "ConfigurationKey": null
    },
  },
  "RateLimitOptions": {
    "ClientIdHeader": "ClientId",
    "QuotaExceededMessage": null,
    "RateLimitCounterPrefix": "ocelot",
    "DisableRateLimitHeaders": false,
    "HttpStatusCode": 429
  },
  "QoSOptions": {
    "ExceptionsAllowedBeforeBreaking": 0,
    "DurationOfBreak": 0,
    "TimeoutValue": 0
  },
}
```

(continues on next page)

(continued from previous page)

```

    "BaseUrl": null,
    "LoadBalancerOptions": {
      "Type": "LeastConnection",
      "Key": null,
      "Expiry": 0
    },
    "DownstreamScheme": "http",
    "HttpHandlerOptions": {
      "AllowAutoRedirect": false,
      "UseCookieContainer": false,
      "UseTracing": false
    }
  }
}

```

Ocelot also allows you to set **DynamicRoutes** collection which lets you set rate limiting rules per downstream service. This is useful if you have for example a product and search service and you want to rate limit one more than the other. An example of this would be as follows:

```

{
  "DynamicRoutes": [
    {
      "ServiceName": "product",
      "RateLimitRule": {
        "ClientWhitelist": [],
        "EnableRateLimiting": true,
        "Period": "1s",
        "PeriodTimespan": 1000.0,
        "Limit": 3
      }
    }
  ],
  "GlobalConfiguration": {
    "RequestIdKey": null,
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 8523,
      "Type": "Consul"
    },
    "RateLimitOptions": {
      "ClientIdHeader": "ClientId",
      "QuotaExceededMessage": "",
      "RateLimitCounterPrefix": "",
      "DisableRateLimitHeaders": false,
      "HttpStatusCode": 428
    },
    "DownstreamScheme": "http"
  }
}

```

This configuration means that if you have a request come into Ocelot on `/product/*` then dynamic routing will kick in and Ocelot will use the rate limiting set against the product service in the **DynamicRoutes** section.

Please take a look through all of the docs to understand these options.

Custom Providers

Ocelot also allows you to create your own *Service Discovery* implementation. This is done by implementing the `IServiceDiscoveryProvider` interface, as shown in the following example:

```
public class MyServiceDiscoveryProvider : IServiceDiscoveryProvider
{
    private readonly DownstreamRoute _downstreamRoute;

    public MyServiceDiscoveryProvider(DownstreamRoute downstreamRoute)
    {
        _downstreamRoute = downstreamRoute;
    }

    public async Task<List<Service>> Get()
    {
        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 application's **ConfigureServices** method, register a `ServiceDiscoveryFinderDelegate` to initialize and return the provider:

```
ServiceDiscoveryFinderDelegate serviceDiscoveryFinder = (provider, config, route) =>
{
    return new MyServiceDiscoveryProvider(route);
};
services.AddSingleton(serviceDiscoveryFinder);
services.AddOcelot();
```

Custom Provider Sample

In order to introduce a basic template for a custom Service Discovery provider, we've prepared a good sample:

Link: [samples / OcelotServiceDiscovery](#)

Solution: [Ocelot.Samples.ServiceDiscovery.sln](#)

This solution contains the following projects:

- *ApiGateway*
- *DownstreamService*

This solution is ready for any deployment. All services are bound, meaning all ports and hosts are prepared for immediate use (running in Visual Studio).

All instructions for running this solution are in [README.md](#).

DownstreamService

This project provides a single downstream service that can be reused across *ApiGateway* routes. It has multiple **launch-Settings.json** profiles for your favorite launch and hosting scenarios: Visual Studio running sessions, Kestrel console hosting, and Docker deployments.

ApiGateway

This project includes a custom *Service Discovery* provider and it only has route(s) to *DownstreamService* services in the **ocelot.json** file. You can add more routes!

The main source code for the custom provider is in the [ServiceDiscovery](#) folder: the *MyServiceDiscoveryProvider* and *MyServiceDiscoveryProviderFactory* classes. You are welcome to design and develop them!

Additionally, the cornerstone of this custom provider is the *ConfigureServices* method, where you can choose design and implementation options: simple or more complex:

```
builder.ConfigureServices(s =>
{
    // 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, which is ServiceDiscoveryProviderFactory
        s.AddSingleton<ServiceDiscoveryFinderDelegate>((serviceProvider, config,
        ↪downstreamRoute)
            => new MyServiceDiscoveryProvider(serviceProvider, config, downstreamRoute));
    }
    else
    {
        // Design #2. Abstract from the default factory
        ↪(ServiceDiscoveryProviderFactory) and from FinderDelegate,
        // and create your own factory by implementing the
        ↪IServiceDiscoveryProviderFactory interface.
```

(continues on next page)

(continued from previous page)

```

        s.RemoveAll<IServiceDiscoveryProviderFactory>();
        s.AddSingleton<IServiceDiscoveryProviderFactory,
↪MyServiceDiscoveryProviderFactory>();

        // It will not be called, but it is necessary for internal validators, it is
↪also a lifehack
        s.AddSingleton<ServiceDiscoveryFinderDelegate>((serviceProvider, config,
↪downstreamRoute) => null);
    }

    s.AddOcelot();
});

```

The easy way, lite design means that you only design the provider class, and specify `ServiceDiscoveryFinderDelegate` object for default `ServiceDiscoveryProviderFactory` in Ocelot core.

A more complex design means that you design both provider and provider factory classes. After this, you need to add the `IServiceDiscoveryProviderFactory` interface to the DI container, removing the default registered `ServiceDiscoveryProviderFactory` class. Note that in this case the Ocelot pipeline will not use `ServiceDiscoveryProviderFactory` by default. Additionally, you do not need to specify "Type": "MyServiceDiscoveryProvider" in the **ServiceDiscoveryProvider** properties of the **GlobalConfiguration** settings. But you can leave this Type option for compatibility between both designs.

Service Fabric

If you have services deployed in [Azure Service Fabric](#) you will normally use the naming service to access them.

The following example shows how to set up a Route that will work in *Service Fabric*. The most important thing is the **ServiceName** which is made up of the *Service Fabric* application name then the specific service name. We also need to set up the **ServiceDiscoveryProvider** in **GlobalConfiguration**. The example here shows a typical configuration. It assumes *Service Fabric* is running on localhost and that the naming service is on port 19081.

The example below is taken from the [OcelotServiceFabric](#) 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.com"
    "RequestIdKey": "OcRequestId",
    "ServiceDiscoveryProvider": {
      "Host": "localhost",
      "Port": 19081,
      "Type": "ServiceFabric"
    }
  }
}

```

(continues on next page)

(continued from previous page)

```
}
}
```

If you are using stateless / guest exe services, Ocelot will be able to proxy through the naming service without anything else. However, if you are using statefull / actor services, you must send the **PartitionKind** and **PartitionKey** query string values with the client request e.g.

GET <http://ocelot.com/EquipmentInterfaces?PartitionKind=xxx&PartitionKey=xxx>

There is no way for Ocelot to work these out for you.

Placeholders in Service Name^{Page 77, 1}

In Ocelot, you can insert placeholders for variables into your `UpstreamPathTemplate` and `ServiceName` using the format `{something}`.

Important Note: The placeholder variable must exist in both the `DownstreamPathTemplate` (or `ServiceName`) and the `UpstreamPathTemplate`. Specifically, the `UpstreamPathTemplate` should include all placeholders from the `DownstreamPathTemplate` and `ServiceName`. Failure to do so will result in Ocelot not starting due to validation errors, which are logged.

Once the validation stage is cleared, Ocelot will replace the placeholder values in the `UpstreamPathTemplate` with those from the `DownstreamPathTemplate` and/or `ServiceName` for each processed request. Thus, the *Placeholders in Service Name 1* behave similarly to the *Placeholders* feature, but with the `ServiceName` property considered during processing.

Placeholders example

Here is the example of the version variable in *Service Fabric* service name.

Given you have 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 a request: GET <https://ocelot.com/api/1.0/products>

¹ “Placeholders in Service Name 1” feature was requested in issue 721 and delivered by PR 722 as a part of version 13.0.0.

Then the *Service Fabric* request: GET `http://localhost:19081/Service_1.0/Api/products`

Tracing

This page details how to perform distributed tracing with Ocelot.



Ocelot provides tracing functionality from the excellent [OpenTracing API for .NET](#) project. The code for the Ocelot integration can be found in `Ocelot.Tracing.OpenTracing` project.

The example below uses [C# Client for Jaeger](#) client to provide the tracer used in Ocelot. In order to add [OpenTracing](#) services we must call the `AddOpenTracing()` extension of the `OcelotBuilder` being returned by `AddOcelot()`¹ like below:

```
services.AddSingleton<ITracer>(sp =>
{
    var loggerFactory = sp.GetService<ILoggerFactory>();
    Configuration config = new Configuration(context.HostingEnvironment.ApplicationName,
    loggerFactory);

    var tracer = config.GetTracer();
    GlobalTracer.Register(tracer);
    return tracer;
});

services
    .AddOcelot()
    .AddOpenTracing();
```

Then in your `ocelot.json` add the following to the Route you want to trace:

```
"HandlerOptions": {
  "UseTracing": true
}
```

Ocelot will now send tracing information to [Jaeger](#) when this Route is called.

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

OpenTracing Status

The [OpenTracing](#) project was archived on January 31, 2022 (see [the article](#)). The Ocelot team will decide on a migration to [OpenTelemetry](#) which is highly desired.

Butterfly

Ocelot provides tracing functionality from the excellent [Butterfly](#) project. The code for the Ocelot integration can be found in [Ocelot.Tracing.Butterfly](#) project.

In order to use the tracing please read 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 `ConfigureServices` method to add Butterfly services: we must call the `AddButterfly()` extension of the `OcelotBuilder` being returned by `AddOcelot()` ^{Page 78, 1} like below:

```
services
    .AddOcelot()
    // This comes from Ocelot.Tracing.Butterfly package
    .AddButterfly(option =>
    {
        // This is the URL that the Butterfly collector server is running on...
        option.CollectorUrl = "http://localhost:9618";
        option.Service = "Ocelot";
    });
```

Then in your `ocelot.json` add the following to the Route you want to trace:

```
"HandlerOptions": {
  "UseTracing": true
}
```

Ocelot will now send tracing information to Butterfly when this Route is called.

Websockets

- [WebSockets Standard](#) by WHATWG organization
- [The WebSocket Protocol](#) by Internet Engineering Task Force (IETF) organization

Ocelot supports proxying [WebSockets](#) with some extra bits. This functionality was requested in [issue 212](#).

In order to get [WebSocket](#) proxying working with Ocelot you need to do the following. In your `Configure` method you need to tell your application to use [WebSockets](#):

```
Configure(app =>
{
    app.UseWebSockets();
    app.UseOcelot().Wait();
})
```

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 set Ocelot will match any *WebSocket* traffic that comes in on / and proxy it to **localhost:5001/ws**. To make this clearer Ocelot will receive messages from the upstream client, proxy these to the downstream service, receive messages from the downstream service and proxy these to the upstream client.

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](#)

SignalR

Welcome to [Real-time ASP.NET with SignalR](#)

Ocelot supports proxying *SignalR*. This functionality was requested in [issue 344](#). In order to get *WebSocket* proxying working with Ocelot you need to do the following.

First, install *SignalR Client* NuGet package:

```
NuGet\Install-Package Microsoft.AspNetCore.SignalR.Client
```

The package is deprecated, but [new versions](#) are still built from the source code. So, *SignalR* is [the part](#) of the ASP.NET Framework which can be referenced like:

```
<ItemGroup>
  <FrameworkReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

More information on framework compatibility can be found in instructions: [Use ASP.NET Core APIs in a class library](#).

Second, you need to tell your application to use *SignalR*. Complete reference is here: [ASP.NET Core SignalR configuration](#)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddOcelot();
    services.AddSignalR();
}
```

Pay attention to configuration of transport level of *WebSockets*, so [configure allowed transports](#) to allow *WebSockets* connections.

Then in your `ocelot.json` add the following to proxy a Route using SignalR. Note normal Ocelot routing rules apply the main thing is the scheme which is set to `ws`.

```
{
  "UpstreamHttpMethod": [ "GET", "POST", "PUT", "DELETE", "OPTIONS" ],
  "UpstreamPathTemplate": "/gateway/{catchAll}",
  "DownstreamPathTemplate": "/{catchAll}",
  "DownstreamScheme": "ws",
  "DownstreamHostAndPorts": [
    { "Host": "localhost", "Port": 5001 }
  ]
}
```

WebSocket Secure

If you define a route with Secured WebSocket protocol, use the `wss` scheme:

```
{
  "DownstreamScheme": "wss",
  // ...
}
```

Keep in mind: you can use WebSocket SSL for both *SignalR* and *WebSockets*.

To understand `wss` scheme, browse to this:

- Microsoft Learn: [Secure your connection with TLS/SSL](#)
- IETF | The WebSocket Protocol: [WebSocket URIs](#)

If you have questions, it may be helpful to search for documentation on MS Learn:

- Search for “secure websocket”

SSL Errors

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

```
{
  "DownstreamScheme": "wss",
  "DangerousAcceptAnyServerCertificateValidator": true,
  // ...
}
```

But we don't recommend doing this! Read the official notes regarding *SSL Errors* in the *Configuration* doc, where you will also find best practices for your environments.

Note, the `wss` scheme fake validator was added by [PR 1377](#), as a part of issues [1375](#), [1237](#) and etc. This life hacking feature for self-signed SSL certificates is available in version `20.0`. It will be removed and/or reworked in future releases. See the *SSL Errors* section for details.

Supported

1. *Load Balancer*
2. *Routing*
3. *Service Discovery*

This means that you can set up your downstream services running *WebSockets* and either have multiple **DownstreamHostAndPorts** in your Route config, or hook your Route into a service discovery provider and then load balance requests... Which we think is pretty cool.

Not Supported

Unfortunately a lot of Ocelot features are non *WebSocket* specific, such as header and http client stuff. We have listed what will not work below:


1. *Tracing*
2. *Request ID*
3. *Request Aggregation*
4. *Rate Limiting*
5. *Quality of Service*
6. *Middleware Injection*
7. *Headers Transformation*
8. *Delegating Handlers*
9. *Claims Transformation*
10. *Caching*
11. *Authentication*¹
12. *Authorization*

We are not 100% sure what will happen with this feature when it gets into the wild, so please make sure you test thoroughly!

Future

Websockets and *SignalR* are being developed intensively by the .NET community, so you need to watch for trends, releases in official docs regularly:

- [WebSockets docs](#)
- [SignalR docs](#)

As a team, we cannot advise you on development, but feel free to ask questions, get coding recipes in the [Discussions](#) space of the repository. 

Also, we welcome any bug reports, enhancements or proposals regarding this feature.

¹ If anyone requests it, we might be able to do something with basic authentication.

The Ocelot team considers the current implementation of WebSockets feature obsolete, based on the [WebSocketsProxyMiddleware](#) class. Websockets are the part of ASP.NET Core framework having native [WebSocketMiddleware](#) class. We have a strong intention to migrate or at least redesign the feature, see [issue 1707](#).

Overview

This document summarises the build and release process for the project. The build scripts are written using [Cake](#), and they are defined in `./build.cake`. The scripts have been designed to be run by either developers locally or by a build server (currently [CircleCi](#)), with minimal logic defined in the build server itself.

Building

- You can also just run `dotnet tool restore && dotnet cake` locally! Output will go to the `./artifacts` directory.
- The best way to replicate the CI process is to build Ocelot locally is using the [Dockerfile.build](#) file which can be found in the `docker` folder in [Ocelot](#) root. Use the following command `docker build --platform linux/amd64 -f ./docker/Dockerfile.build .` for example. You will need to change the platform flag depending on your platform.
- There is a [Makefile](#) to make it easier to call the various targets in `build.cake`. The scripts are called with `.sh` but can be easily changed to `.ps1` if you are using Windows.
- Alternatively you can build the project in VS2022 with the latest [.NET 8.0](#) SDK.

Tests

The tests should all just run and work as part of the build process. You can of course also run them in Visual Studio.

Create SSL Cert for Testing

You can do this via [OpenSSL](#):

- Install [openssl package](#) (if you are using Windows, download binaries [here](#)).
- Generate private key: `openssl genrsa 2048 > private.pem`
- Generate the self-signed certificate: `openssl req -x509 -days 1000 -new -key private.pem -out public.pem`
- If needed, create PFX: `openssl pkcs12 -export -in public.pem -inkey private.pem -out mycert.pfx`

Release Process

- The release process works best with [Gitflow](#) branching.
- Contributors can do whatever they want on PRs and feature branches to deliver a feature to **develop** branch.
- Maintainers can do whatever they want on PRs and merges to **main** will result in packages being released to GitHub and NuGet.

Ocelot uses the following process to accept work into the NuGet packages.

1. User creates an issue or picks up an [existing issue](#) in GitHub. An issue can be created by converting [discussion](#) topics if necessary and agreed upon.
2. User creates a [fork](#) and branches from this (unless a member of core team, they can just create a branch on the head repo) e.g. `feature/xxx`, `bug/xxx` etc. It doesn't really matter what the "xxx" is. It might make sense to use the issue number and maybe a short description.
3. When the contributor is happy with their work they can create a pull request against **develop** in GitHub with their changes.
4. The maintainer must follow the [SemVer](#) support for this is provided by [GitVersion](#). So if the maintainer needs to make breaking changes, be sure to use the correct commit message, so [GitVersion](#) uses the correct **SemVer** tags. Do not manually tag the Ocelot repo: this will break things!
5. The Ocelot team will review the PR and if all is good merge it, else they will suggest feedback that the user will need to act on.

In order to speed up getting a PR the contributor should think about the following:

- Have I covered all my changes with tests at unit and acceptance level?
- Have I updated any documentation that my changes may have affected?
- Does my feature make sense, have I checked all of Ocelot's other features to make sure it doesn't already exist?

In order for a PR to be merged the following must have occurred:

- All new code is covered by unit tests.
 - All new code has at least 1 acceptance test covering the happy path.
 - Tests must have passed locally.
 - Build must have green status.
 - Build must not have slowed down dramatically.
 - The main Ocelot package must not have taken on any non MS dependencies.
6. After the PR is merged to **develop** the Ocelot NuGet packages will not be updated until a release is created.
 7. When enough work has been completed to justify a new release, **develop** branch will be merged into **main** as **release/xxx** branch, the release process will begin which builds the code, versions it, pushes artifacts to GitHub and NuGet packages to NuGet.
 8. The final step is to go back to GitHub and close any issues that are now fixed. **Note:** All linked issues to the PR in **Development** settings (right side PR settings) will be closed automatically while merging the PR. It is imperative that developer uses the "**Link an issue from this repository**" pop-up dialog of the **Development** settings!

Notes

All NuGet package builds and releases are done with CircleCI, see [Pipelines - ThreeMammals/Ocelot](#).

Only Tom Pallister (owner) and Ocelot Core Team members (maintainers) can merge releases into **main** at the moment. This is to ensure there is a final *quality gate* in place. Tom is mainly looking for security issues on the final merge.

We **do** follow this development and release process! If anything is unclear or you get stuck in the process, please contact the [Ocelot Core Team](#) members or repository maintainers.

Quality Gates

To be developed...