Ocelot

Release 18.0

Tom Pallister, Ocelot Core team at ThreeMammals

Feb 16, 2024

INTRODUCTION

Thanks for taking a look at the Ocelot documentation. Please use the left hand nav to get around. I would suggest taking a look at introduction first.

CHAPTER

ONE

BIG PICTURE

Ocelot is aimed at people using .NET running a micro services / service orientated architecture that need a unified point of entry into their system.

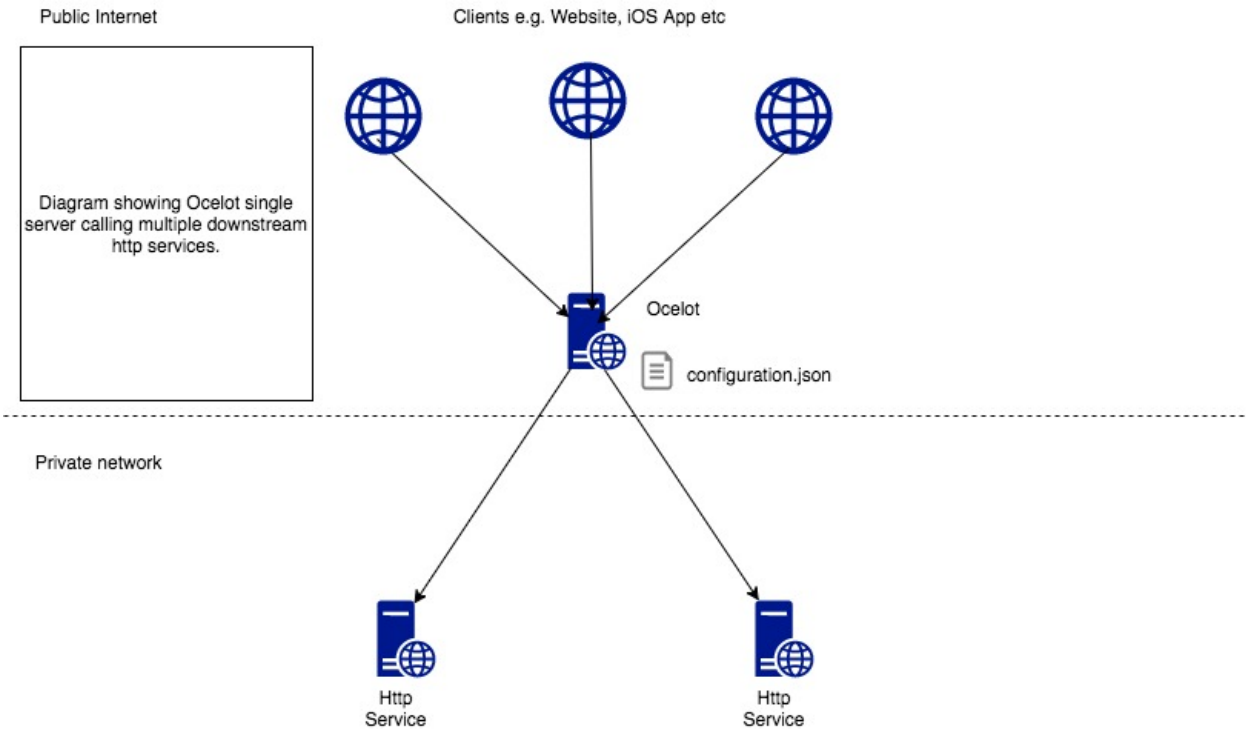
In particular I want easy integration with IdentityServer reference and bearer tokens.

Ocelot is a bunch of middlewares in a specific order.

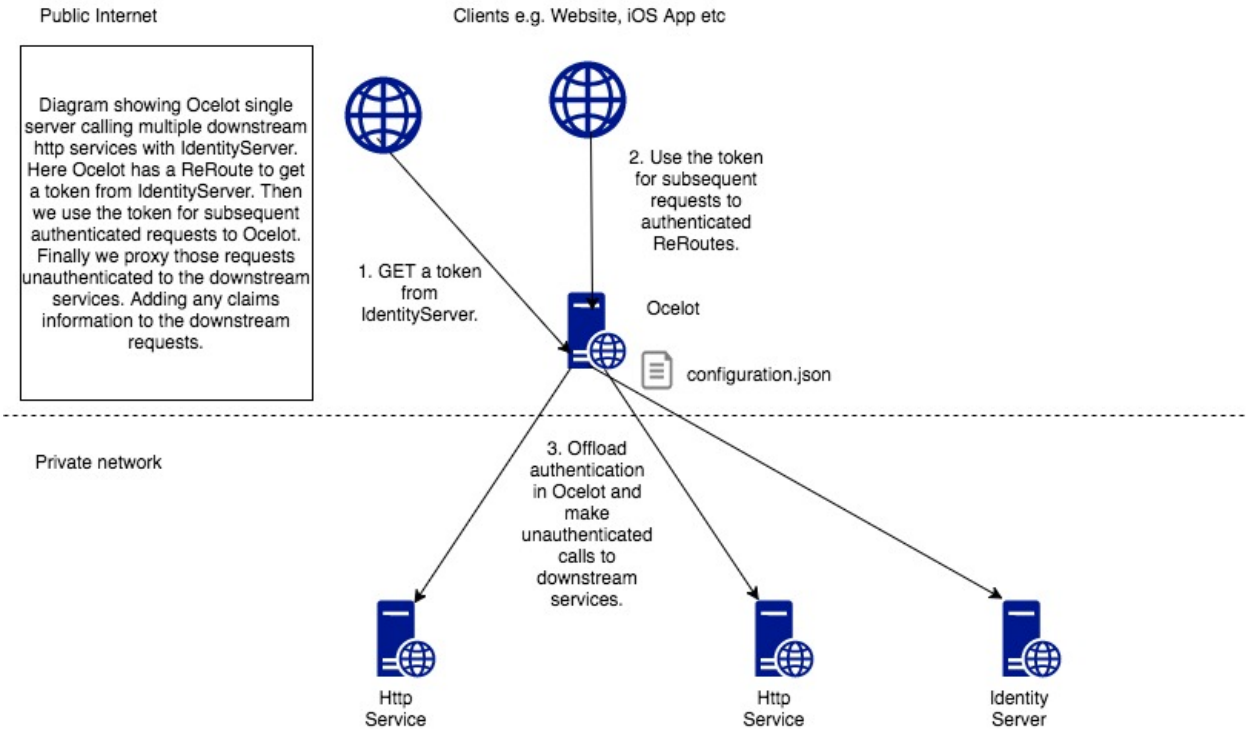
Ocelot manipulates the HttpRequest object into a state specified by its configuration until it reaches a request builder middleware where it creates a HttpRequestMessage object which is used to make a request to a downstream service. The middleware that makes the request is the last thing in the Ocelot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

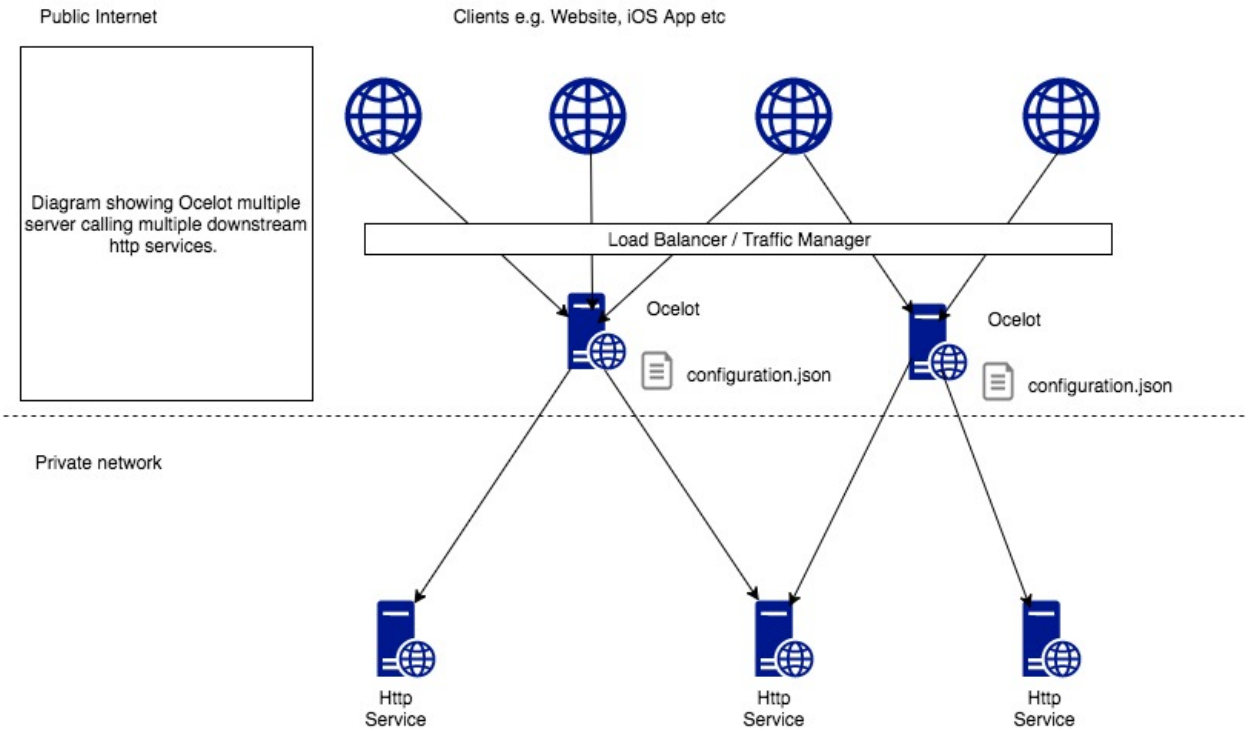
1.1 Basic Implementation



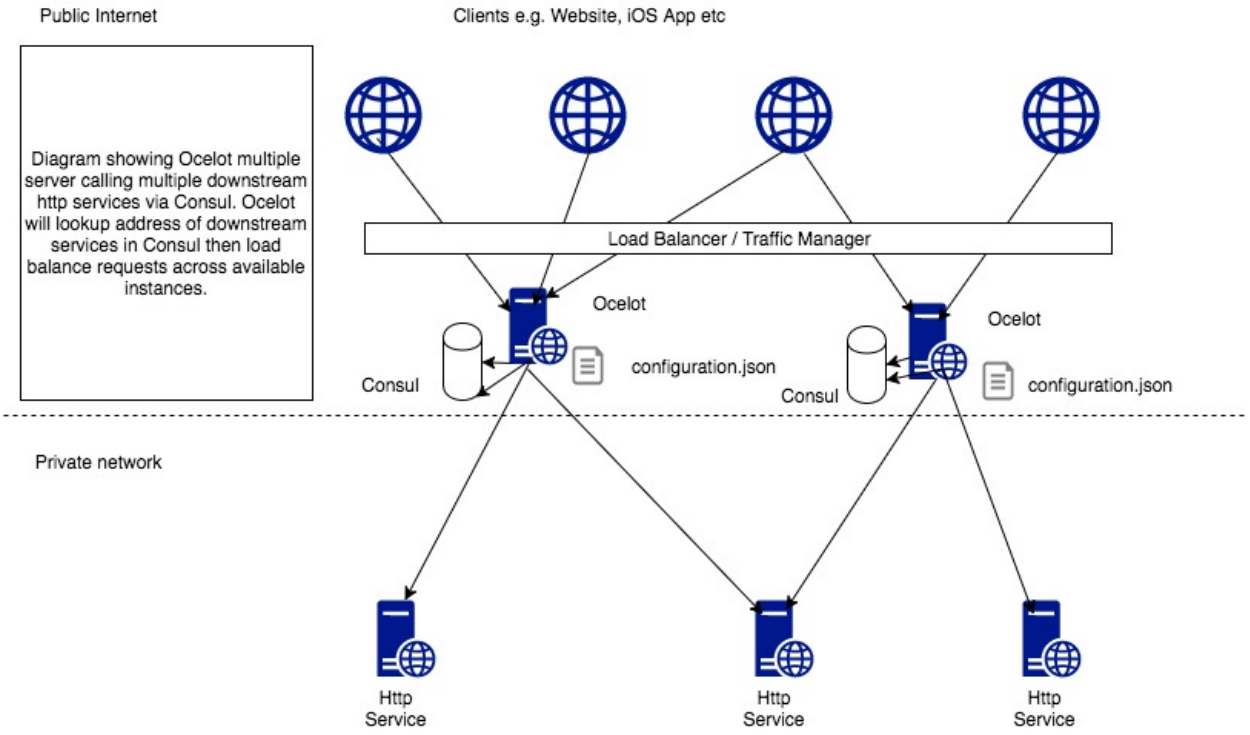
1.2 With IdentityServer



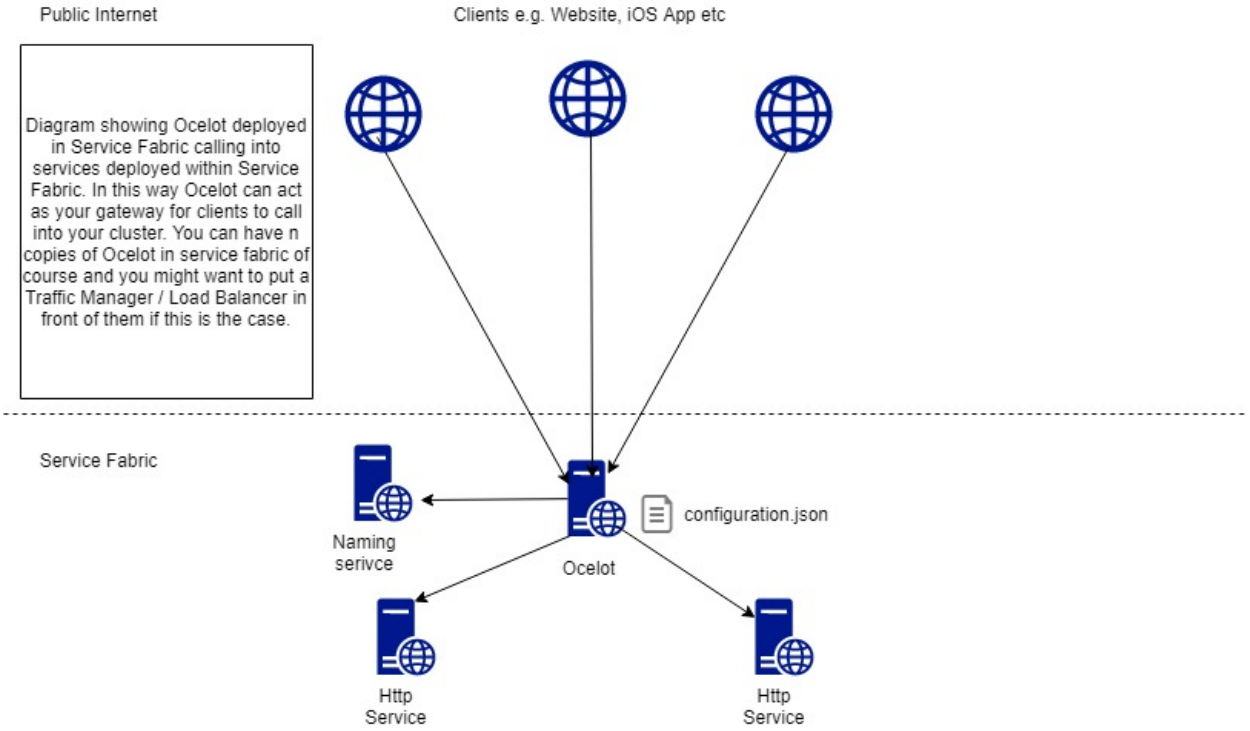
1.3 Multiple Instances



1.4 With Consul



1.5 With Service Fabric



CHAPTER

TWO

GETTING STARTED

Ocelot is designed to work with ASP.NET and is currently on net6.0.

2.1 .NET 6.0

Install NuGet package

Install Ocelot and it's dependencies using nuget. You will need to create a net6.0 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 here.

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" ]
        }
    ],
```

(continues on next page)

(continued from previous page)

```
    "GlobalConfiguration": {
        "BaseUrl": "https://localhost:5000"
    }
}
```

The most important thing to note here is BaseUrl. 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 base url 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 services), UseOcelot().Wait() (sets up all the Ocelot middleware).

```
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
            })
            .UseIISIntegration()
```

(continues on next page)

(continued from previous page)

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

CHAPTER

THREE

CONTRIBUTING

Pull requests, issues and commentary welcome! No special process just create a request and get in touch either via gitter or create an issue.

CHAPTER

FOUR

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 - I have looked multiple times at building swagger.json out of the Ocelot ocelot.json but it doesnt fit into the vision I have 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

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

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

The main reasons why I 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 doesnt 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 I rolled my own Swagger implementation.

If the user wants something to easily test against the Ocelot API then I 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 I don't intend to do this.

CHAPTER

FIVE

GOTCHAS

Note: When using ASP.NET Core 2.2 and you want to use In-Process hosting, replace **.UseIISIntegration()** with **.UseIIS()**, otherwise you'll get startup errors.

CHAPTER

SIX

CONFIGURATION

An example configuration can be found here. 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 Global configuration is a bit hacky and allows overrides of Route specific settings. It's useful if you don't 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": {},
          "AddClaimsToRequest": {},
          "RouteClaimsRequirement": {},
          "AddQueriesToRequest": {},
          "RequestIdKey": "",
          "FileCacheOptions": {
              "TtlSeconds": 0,
              "Region": ""
          },
          "RouteIsCaseSensitive": false,
          "ServiceName": "",
          "DownstreamScheme": "http",
          "DownstreamHostAndPorts": [
              {
                  "Host": "localhost",
                  "Port": 51876,
              }
          ],
          "QoSOptions": {
              "ExceptionsAllowedBeforeBreaking": 0,
```

(continues on next page)

(continued from previous page)

```
            "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
    }
```

More information on how to use these options is below..

6.1 Multiple environments

Like any other asp.net core project Ocelot supports configuration file names such as configuration.dev.json, configuration.test.json etc. In order to implement this add the following to you

```
.ConfigureAppConfiguration((hostingContext, config) =>
        {
            config
                .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
                .AddJsonFile("appsettings.json", true, true)
                .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.
↪EnvironmentName}.json", true, true)
                .AddJsonFile("ocelot.json")
                .AddJsonFile($"configuration.{hostingContext.HostingEnvironment.
↪EnvironmentName}.json")
                .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.

6.2 Merging configuration files

This feature was requested in Issue 296 and allows users to have multiple configuration files to make managing large configurations easier.

Instead of adding the configuration directly e.g. AddJsonFile("ocelot.json") you can call AddOcelot() like below.

```
.ConfigureAppConfiguration((hostingContext, config) =>
    {
        config
            .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
            .AddJsonFile("appsettings.json", true, true)
            .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.
↪EnvironmentName}.json", true, true)
            .AddOcelot(hostingContext.HostingEnvironment)
            .AddEnvironmentVariables();
    })
```

In this scenario Ocelot will look for any files that match the pattern (?i)ocelot.([a-zA-Z0-9]*).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 aswell as any Routes or AggregateRoutes. Ocelot will then save the merged configuration to a file called ocelot.json and this will be used as the source of truth while ocelot is running.

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. I would advise always checking what is in ocelot.json if you have any problems.

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

```
.ConfigureAppConfiguration((hostingContext, config) =>
    {
        config
            .SetBasePath(hostingContext.HostingEnvironment.ContentRootPath)
            .AddJsonFile("appsettings.json", true, true)
            .AddJsonFile($"appsettings.{hostingContext.HostingEnvironment.
↪EnvironmentName}.json", true, true)
            .AddOcelot("/foo/bar", hostingContext.HostingEnvironment)
            .AddEnvironmentVariables();
    })
```

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

6.3 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.

```
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
    }
}
```

I 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! I guess it means if you want to use Ocelot to its fullest you take on Consul as a dependency for now.

This feature has a 3 second ttl cache before making a new request to your local consul agent.

6.4 Reload JSON config on change

Ocelot supports reloading the json configuration file on change. e.g. the following will recreate Ocelots internal configuration when the ocelot.json file is updated manually.

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

6.4.1 Configuration Key

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 :) This feature was requested in issue 346! In order to specify the key you need to set the ConfigurationKey property in the ServiceDiscoveryProvider section of the configuration json file e.g.

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

In this example Ocelot will use Oceolot_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.

6.5 Follow Redirects / Use CookieContainer

Use HttpHandlerOptions in Route configuration to set up HttpHandler behavior:

1. 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.
2. 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 that if you are using the CookieContainer Ocelot caches the HttpClient for each downstream service. This means that all requests to that DownstreamService will share the same cookies. Issue 274 was created because a user noticed that the cookies were being shared. I tried to think of a nice way to handle this but I 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. I 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!

6.6 SSL Errors

If you want to ignore SSL warnings / errors set the following in your Route config.

```
"DangerousAcceptAnyServerCertificateValidator": true
```

I don't recommend doing this, I suggest creating your own certificate and then getting it trusted by your local / remote machine if you can.

6.7 MaxConnectionsPerServer

This controls how many connections the internal HttpClient will open. This can be set at Route or global level.

6.8 React to Configuration Changes

Resolve IOcelotConfigurationChangeTokenSource from the DI container if you wish to react to changes to the Ocelot configuration via the Ocelot.Administration API or ocelot.json being reloaded from the disk. You may either poll the change token's HasChanged property, or register a callback with the RegisterChangeCallback method.

6.8.1 Polling the HasChanged property

6.8.2 Registering a callback

6.8.3 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".

CHAPTER

SEVEN

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's 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.

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

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

DownstreamHostAndPorts 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 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.

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.

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

This will forward any path + query string combinations to the downstream service after the path /api.

The default ReRouting 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.

7.1 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.

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

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.

```
{
    "DownstreamPathTemplate": "/",
    "DownstreamScheme": "https",
    "DownstreamHostAndPorts": [
            {
                "Host": "10.0.10.1",
```

(continues on next page)

(continued from previous page)

```
                "Port": 80,
            }
        ],
    "UpstreamPathTemplate": "/",
    "UpstreamHttpMethod": [ "Get" ]
}
```

7.2 Upstream Host

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.

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

This feature was requested as part of Issue 216 .

7.3 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!).

7.4 Dynamic Routing

This feature was requested in issue 340.

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 docs service-discovery if this sounds interesting to you.

7.5 Query Strings

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

```
{
    "Routes": [
        {
            "DownstreamPathTemplate": "/api/subscriptions/{subscriptionId}/updates?
↪unitId={unitId}",
            "UpstreamPathTemplate": "/api/units/{subscriptionId}/{unitId}/updates",
            "UpstreamHttpMethod": [
                "Get"
            ],
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "localhost",
                    "Port": 50110
                }
            ]
        }
    ],
    "GlobalConfiguration": {
    }
}
```

In this example Ocelot will use the value from the {unitId} in the upstream path template and add it to the downstream request as a query string parameter called unitId!

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

```
{
    "Routes": [
        {
            "DownstreamPathTemplate": "/api/units/{subscriptionId}/{unitId}/updates",
            "UpstreamPathTemplate": "/api/subscriptions/{subscriptionId}/updates?unitId=
↪{unitId}",
            "UpstreamHttpMethod": [
                "Get"
            ],
            "DownstreamScheme": "http",
            "DownstreamHostAndPorts": [
                {
                    "Host": "localhost",
                    "Port": 50110
                }
            ]
        }
    ],
    "GlobalConfiguration": {
    }
}
```

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 {unitId} parameter from the query string and use it in the downstream request path.

CHAPTER

EIGHT

REQUEST AGGREGATION

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 type architecture with Ocelot.

This feature was requested as part of Issue 79 and further improvements were made as part of Issue 298.

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).

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

(continues on next page)

(continued from previous page)

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

8.2 Basic expecting JSON from Downstream Services

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

(continues on next page)

(continued from previous page)

```
            "UpstreamPathTemplate": "/"
        }
    ]
}
```

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.

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 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.

8.2.1 Gotcha's / Further info

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

Aggregation only supports the GET HTTP Verb.

CHAPTER

NINE

GRAPHQL

OK you got me Ocelot doesn't directly support GraphQL but so many people have asked about it I wanted to show how easy it is to integrate the graphql-dotnet library.

Please see the sample project OcelotGraphQL. Using a combination of the graphql-dotnet project and Ocelot's DelegatingHandler features this is pretty easy to do. However I do not intend to integrate more closely with GraphQL at the moment. Check out the samples readme and that should give you enough instruction on how to do this!

Good luck and have fun :>

CHAPTER

TEN

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 Ocelot is forwarding a request to. At the moment this is only supported in the GlobalConfiguration section which means the same service discovery provider will be used for all Routes you specify a ServiceName for at Route level.

10.1 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 add the following to your ConfigureServices method.

```
s.AddOcelot()
    .AddConsul();
```

The following is required in the GlobalConfiguration. The Provider is required and if you do not specify a host and port the Consul default will be used.

Please note the Scheme option defauls to HTTP. It was added in this PR. It defaults to HTTP to not introduce a breaking change.

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

In the future we can add a feature that allows Route specfic 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 algorithm you can use. If no load balancer is specified Ocelot will not load balance requests.

```
{
    "DownstreamPathTemplate": "/api/posts/{postId}",
    "DownstreamScheme": "https",
    "UpstreamPathTemplate": "/posts/{postId}",
    "UpstreamHttpMethod": [ "Put" ],
    "ServiceName": "product",
```

(continues on next page)

(continued from previous page)

```
    "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.

A lot of people have asked me 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. I 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.

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. I have been contacted before about not accepting scheme in Address and accepting scheme in address. After reading this I don't think the scheme should be in there.

Or

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

10.1.1 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,
    "Token": "footoken",
    "Type": "Consul"
}
```

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

10.2 Eureka

This feature was requested as part of Issue 262 . to add support for Netflix's 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 NuGet 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
    }
}
```

I am told that if shouldRegisterWithEureka is false then shouldFetchRegistry will defaut to true so you don't need 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. Please note that this code is provided by the Pivotal.Discovery.Client NuGet package so big thanks to them for all the hard work.

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

10.3 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 apprend any query string to the downstream url as normal.

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
        },
        "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 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": [
```

(continues on next page)

(continued from previous page)

```
        {
        "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.

CHAPTER

ELEVEN

SERVICE FABRIC

If you have services deployed in 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 samples folder so please check it if this doesnt make sense!

```
{
    "Routes": [
        {
        "DownstreamPathTemplate": "/api/values",
        "UpstreamPathTemplate": "/EquipmentInterfaces",
        "UpstreamHttpMethod": [
            "Get"
        ],
        "DownstreamScheme": "http",
        "ServiceName": "OcelotServiceApplication/OcelotApplicationService",
        }
    ],
    "GlobalConfiguration": {
        "RequestIdKey": "OcRequestId",
        "ServiceDiscoveryProvider": {
            "Host": "localhost",
            "Port": 19081,
            "Type": "ServiceFabric"
        }
    }
}
```

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.

CHAPTER

TWELVE

KUBERNETES

This feature was requested as part of Issue 345 . to add support for kubernetes's provider.

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.

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

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

Then add the following to your ConfigureServices method.

```
s.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 ServiceAccount using Pod to access the service of the k8s cluster needs to be ServiceAccount based on RBAC authorization

You can replicate a Permissive. Using RBAC role bindings. 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

The following example shows 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. The example here shows a typical configuration.

```
  {
"Routes": [
  {
    "DownstreamPathTemplate": "/api/values",
    "DownstreamScheme": "http",
    "UpstreamPathTemplate": "/values",
    "ServiceName": "downstreamservice",
    "UpstreamHttpMethod": [ "Get" ]
  }
],
"GlobalConfiguration": {
  "ServiceDiscoveryProvider": {
    "Host": "192.168.0.13",
    "Port": 443,
    "Token": "txpc696iUhbVoudg164r93CxDTrKRVWG",
```

(continues on next page)

(continued from previous page)

```
    "Namespace": "dev",
    "Type": "kube"
  }
}
```

}

Service deployment in Namespace Dev , ServiceDiscoveryProvider type is kube, you also can set pollkube ServiceDiscoveryProvider type.

Note: Host Port and Token are no longer in use

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": {
 "Host": "192.168.0.13",
 "Port": 443,
 "Token": "txpc696iUhbVoudg164r93CxDTrKRVWG",
 "Namespace": "dev",
 "Type": "pollkube",
 "PollingInterval": 100
}
```

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. I 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.

If your downstream service resides in a different namespace you can override the global setting at the Route level by specifying a ServiceNamespace.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/values",
      "DownstreamScheme": "http",
      "UpstreamPathTemplate": "/values",
      "ServiceName": "downstreamservice",
      "ServiceNamespace": "downstream-namespace",
      "UpstreamHttpMethod": [ "Get" ]
    }
  ]
}
```

CHAPTER

THIRTEEN

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)
{
    var authenticationProviderKey = "TestKey";

    services.AddAuthentication()
        .AddJwtBearer(authenticationProviderKey, x =>
        {
        });
}
```

In this example TestKey is the scheme that this provider has been registered with. We then map this to a Route in the configuration e.g.

```
"Routes": [{
        "DownstreamHostAndPorts": [
            {
                "Host": "localhost",
                "Port": 51876,
            }
        ],
        "DownstreamPathTemplate": "/",
        "UpstreamPathTemplate": "/",
        "UpstreamHttpMethod": ["Post"],
        "RouteIsCaseSensitive": false,
        "DownstreamScheme": "http",
        "AuthenticationOptions": {
            "AuthenticationProviderKey": "TestKey",
            "AllowedScopes": []
        }
    }]
```

When Ocelot runs it will look at this Routes AuthenticationOptions.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.

13.1 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 = "TestKey";

    services.AddAuthentication()
        .AddJwtBearer(authenticationProviderKey, x =>
        {
            x.Authority = "test";
            x.Audience = "test";
        });

    services.AddOcelot();
}
```

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

```
"Routes": [{
        "DownstreamHostAndPorts": [
            {
                "Host": "localhost",
                "Port": 51876,
            }
        ],
        "DownstreamPathTemplate": "/",
        "UpstreamPathTemplate": "/",
        "UpstreamHttpMethod": ["Post"],
        "RouteIsCaseSensitive": false,
        "DownstreamScheme": "http",
        "AuthenticationOptions": {
            "AuthenticationProviderKey": "TestKey",
            "AllowedScopes": []
        }
    }]
```

13.2 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 = "TestKey";
    Action<JwtBearerOptions> options = o =>
        {
            o.Authority = "https://whereyouridentityserverlives.com";
            // etc
```

(continues on next page)

(continued from previous page)

```
        };

    services.AddAuthentication()
        .AddJwtBearer(authenticationProviderKey, options);

    services.AddOcelot();
}
```

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

```
"Routes": [{
        "DownstreamHostAndPorts": [
            {
                "Host": "localhost",
                "Port": 51876,
            }
        ],
        "DownstreamPathTemplate": "/",
        "UpstreamPathTemplate": "/",
        "UpstreamHttpMethod": ["Post"],
        "RouteIsCaseSensitive": false,
        "DownstreamScheme": "http",
        "AuthenticationOptions": {
            "AuthenticationProviderKey": "TestKey",
            "AllowedScopes": []
        }
    }]
```

13.3 Okta

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

Issue 446 that contains some code and examples that might help with Okta integration.

13.4 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 all of the scopes in the list.

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

CHAPTER

FOURTEEN

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 you Route configuration.

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

In this example when the authorization middleware is called Ocelot will check to seeif 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.

CHAPTER

FIFTEEN

WEBSOCKETS

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.

```
{
     "DownstreamPathTemplate": "/ws",
     "UpstreamPathTemplate": "/",
     "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.

15.1 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.

Install Microsoft.AspNetCore.SignalR.Client 1.0.2 you can try other packages but this one is tested.

Do not run it in IISExpress or install the websockets feature in the IIS features

In your Configure method you need to tell your application to use SignalR.

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

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".

```
  {
 "Routes": [
   {
     "DownstreamPathTemplate": "/{catchAll}",
     "DownstreamScheme": "ws",
     "DownstreamHostAndPorts": [
       {
         "Host": "localhost",
         "Port": 50000
       }
     ],
     "UpstreamPathTemplate": "/gateway/{catchAll}",
     "UpstreamHttpMethod": [ "GET", "POST", "PUT", "DELETE", "OPTIONS" ]
   }
]
```

}

With this configuration set Ocelot will match any SignalR 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.

15.2 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 I think is pretty cool :)

15.3 Not Supported

Unfortunately a lot of Ocelot's features are non websocket specific such as header and http client stuff. I've listed what won't work below.

1. Tracing
2. RequestId
3. Request Aggregation

4. Rate Limiting
5. Quality of Service
6. Middleware Injection
7. Header Transformation
8. Delegating Handlers
9. Claims Transformation
10. Caching
11. Authentication - If anyone requests it we might be able to do something with basic authentication.
12. Authorization

I'm not 100% sure what will happen with this feature when it get's into the wild so please make sure you test thoroughly!

CHAPTER

SIXTEEN

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 relavent NuGet package..

`Install-Package Ocelot.Administration`

This will bring down everything needed by the admin API.

16.1 Providing your own IdentityServer

All you need to do to hook into your own IdentityServer is add the following to your ConfigureServices method.

```
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 alaways be possible to have the Ocelot URL.

16.2 Internal IdentityServer

The API is authenticated using bearer tokens that you request from Ocelot iteself. This is provided by the amazing Identity Server project that I have been using for a few years now. Check them out.

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

The path can be anything you want and it is obviously reccomended don't usea url you would like to route through with Ocelot as this will not work. The administration uses theMapWhen 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!

```
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 scriptscalled 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.

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.

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 thecluster have the same certificate then you are good!

16.3 Administration API

POST {adminPath}/connect/token

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

The body of the request is form-data as follows

`client_id` set as admin

`client_secret` set as whatever you used when setting up the administration services.

`scope` set as admin

`grant_type` set as client_credentials

GET {adminPath}/configuration

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

POST {adminPath}/configuration

This overrwrites the existing configuration (should probably be a put!). I reccomend 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.cs that we use to set up Ocelot on a file system.

Please note that 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 / just use a distributed cache.

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

CHAPTER

SEVENTEEN

RATE LIMITING

Thanks to @catcherwong article for inspiring me to finally write this documentation.

Ocelot supports rate limiting of upstream requests so that your downstream services do not become overloaded. This feature was added by @geffzhang on GitHub! Thanks very much.

OK so 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

```
"RateLimitOptions": {
  "DisableRateLimitHeaders": false,
  "QuotaExceededMessage": "Customize Tips!",
  "HttpStatusCode": 999,
  "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 specifiy the header that should be used to identify clients. By default it is “ClientId”

CHAPTER

EIGHTEEN

CACHING

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

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

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..

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

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

```
"FileCacheOptions": { "TtlSeconds": 15, "Region": "somename" }
```

In this example ttl seconds is set to 15 which means the cache will expire after 15 seconds.

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.

18.1 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. I would really appreciate it if anyone wants to implement Redis, memcache etc..

CHAPTER

NINETEEN

QUALITY OF SERVICE

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 here.

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

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

Then add the following section to a Route configuration.

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

You must set a number greater than 0 against ExceptionsAllowedBeforeBreaking for this rule to be implemented. Duration of break 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 :)

If you do not add a QoS section QoS will not be used however Ocelot will default to a 90 second timeout on all downstream requests. If someone needs this to be configurable open an issue.

CHAPTER

TWENTY

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 GitHub #190 and I decided that it was going to be useful in various ways.

20.1 Add to Request

This feature was requestes in GitHub #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).

20.2 Add to Response

This feature was requested in GitHub #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}"
},
```

20.3 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.

20.4 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/"
},
```

20.5 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/"
},
```

20.6 Placeholders

Ocelot allows placeholders that can be used in header transformation.

{RemoteIpAddress} - This will find the clients IP address using _httpContextAccessor.HttpContext.Connection.RemoteIpAddress.ToString() so you will get back some IP. {BaseUrl} - This will use Ocelot's 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. {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.

20.7 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,
},
```

20.8 X-Forwarded-For

An example of using {RemoteIpAddress} placeholder...

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

20.9 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": {
```

(continues on next page)

(continued from previous page)

```
    "AllowAutoRedirect": false,
},
```

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

CHAPTER

TWENTYONE

HTTP 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:

```
{
    "DownstreamPathTemplate": "/{url}",
    "UpstreamPathTemplate": "/{url}",
    "UpstreamHttpMethod": [
        "Get"
    ],
    "DownstreamHttpMethod": "POST",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
        {
            "Host": "localhost",
            "Port": 53271
        }
    ],
}
```

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.

CHAPTER

TWENTYTWO

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. This allows the user to transform claims before the authorization middleware is called. After the user is authorized first we call the claims to headers middleware, thenthe claims to query string parameters middleware, and Finally the claims to downstream pathmiddleware.

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: I'm not a hotshot programmer so have no idea if this syntax is good...

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 ofall 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.

22.1 Claims to Claims Transformation

Below is an example configuration that will transforms 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".

22.2 Claims to Headers Tranformation

Below is an example configuration that will transforms 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".

22.3 Claims to Query String Parameters Transformation

Below is an example configuration that will transforms 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.

22.4 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.

CHAPTER

TWENTYTHREE

LOGGING

Ocelot uses the standard logging interfaces ILoggerFactory / ILogger<T> at the moment. This is encapsulated in IOcelotLogger / IOcelotLoggerFactory with an implementation for the standard asp.net core logging stuff at the moment. This is because Ocelot add's some extra info to the logs such as request id if it is configured.

There is a global error handler 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.

The reason for not just using bog standard framework logging is that I could not work out how to override the request id that get's logged when setting IncludeScopes to true for logging settings. Nicely onto the next feature.

23.1 Warning

If you are logging to Console you will get terrible performance. I have had so many issues about performance issues with Ocelot and it is always logging level Debug, logging to Console :) Make sure you are logging to something proper in production :)

CHAPTER

TWENTYFOUR

TRACING

This page details how to perform distributed tracing with Ocelot.

24.1 OpenTracing

Ocelot providers tracing functionality from the excellent OpenTracing C# project. The code for the Ocelot integration can be found here.

The example below uses Jaeger C# client to provide the tracer used in Ocelot.

```
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..

```
"HttpHandlerOptions": {
      "UseTracing": true
  },
```

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

24.2 Butterfly

Ocelot providers tracing functionality from the excellent Butterfly project. The code for the Ocelot integration can be found here.

In order to use the tracing please read the Butterfly documentation.

In ocelot you need to do the following if you wish to trace a Route.

```
Install-Package Ocelot.Tracing.Butterfly
```

In your ConfigureServices method

```
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..

```
"HttpHandlerOptions": {
      "UseTracing": true
  },
```

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

CHAPTER

TWENTYFIVE

REQUEST ID / CORRELATION ID

Ocelot supports a client sending a request id in the form of a header. If set Ocelot willuse the requestid for logging as soon as it becomes available in the middleware pipeline. Ocelot will also forward the request id 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"
}
```

I 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.

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 I 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,
```

(continues on next page)

(continued from previous page)

```
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,
```

CHAPTER

TWENTYSIX

MIDDLEWARE INJECTION AND OVERRIDES

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 (ctx, 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.

- PreErrorResponderMiddleware - Already explained above.
- PreAuthenticationMiddleware - This allows the user to run pre authentication logic and then call Ocelot's authentication middleware.
- AuthenticationMiddleware - This overrides Ocelots authentication middleware.
- PreAuthorizationMiddleware - This allows the user to run pre authorization logic and then call Ocelot's authorization middleware.
- AuthorizationMiddleware - This overrides Ocelots authorization middleware.
- PreQueryStringBuilderMiddleware - This allows the user to manipulate the query string on the http request before it is passed to Ocelots request creator.

Obviously you can just add middleware as normal before the call to app.UseOcelot() It cannot be added after as Ocelot does not call the next middleware.

CHAPTER

TWENTYSEVEN

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 type of load balancer available are:

LeastConnection - tracks which services are dealing with requests and sends new requests to service with least existing requests. The algorythm state is not distributed across a cluster of Ocelot's.

RoundRobin - loops through available services and sends requests. The algorythm 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.

27.1 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.

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

27.2 Service Discovery

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

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

27.3 CookieStickySessions

I've 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 I 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.

```
{
    "DownstreamPathTemplate": "/api/posts/{postId}",
    "DownstreamScheme": "https",
    "DownstreamHostAndPorts": [
            {
                "Host": "10.0.1.10",
                "Port": 5000,
            },
            {
                "Host": "10.0.1.11",
                "Port": 5000,
            }
        ],
    "UpstreamPathTemplate": "/posts/{postId}",
    "LoadBalancerOptions": {
        "Type": "CookieStickySessions",
        "Key": "ASP.NET_SessionId",
        "Expiry": 1800000
    },
    "UpstreamHttpMethod": [ "Put", "Delete" ]
}
```

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.

27.4 Custom Load Balancers

`DavidLievrouw <https://github.com/DavidLievrouw`_ implemented a way to provide Ocelot with custom load balancer in **`PR 1155 <https://github.com/ThreeMammals/Ocelot/pull/1155`_**.

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 this is the name of a class we will setup to do load balancing.

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

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

```
private 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;
    }
```

(continues on next page)

(continued from previous page)

```
    public async Task<Response<ServiceHostAndPort>> Lease(DownstreamContext␣
↪downstreamContext, HttpContext httpContext)
    {
        var services = await _services();
        lock (_lock)
        {
            if (_last >= services.Count)
            {
                _last = 0;
            }

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

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

Finally you need to register this class with Ocelot. I 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);

s.AddOcelot()
    .AddCustomLoadBalancer(loadBalancerFactoryFunc);
```

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

```
s.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;
```

(continues on next page)

(continued from previous page)

```
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 balaner 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.

CHAPTER

TWENTYEIGHT

DELEGATING HANDLERS

Ocelot allows the user to add delegating handlers to the HttpClient transport. This feature was requested GitHub #208 and I decided that it was going to be useful in various ways. Since then we extended it in GitHub #264.

28.1 Usage

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 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 cancellationToken)
    {
        //do stuff and optionally call the base handler..
        return await base.SendAsync(request, cancellationToken);
    }
}
```

Next you must add the handlers to Ocelot's container like below...

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

Both of these Add methods have a default parameter called global which is set to false. If it is false then the intent of the DelegatingHandler 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.

e.g.

As below...

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

Finally if you want Route specific DelegatingHandlers or to order your specific and / or global (more on this later) DelegatingHandlers then you must add the following json to the specific Route in ocelot.json. The names in the array must match the class names of your DelegatingHandlers for Ocelot to match them together.

```
"DelegatingHandlers": [
    "FakeHandlerTwo",
    "FakeHandler"
]
```

You can have as many DelegatingHandlers 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 DelegatingHandlers plus any globals that were in the DelegatingHandlers array from ocelot.json ordered as they are in the DelegatingHandlers array.
3. Tracing DelegatingHandler if enabled (see tracing docs).
4. QoS DelegatingHandler if enabled (see QoS docs).
5. The HttpClient sends the HttpRequestMessage.

Hopefully other people will find this feature useful!

CHAPTER

TWENTYNINE

HTTP ERROR STATUS CODES

Ocelot will return HTTP status error codes based on internal logic in certain siturations: - 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 authroised, scope not authorized, user doesnt have required claim or cannot find claim. - 503 if the downstream request times out. - 499 if the request is cancelled by the client. - 404 if unable to find a downstream route. - 502 if unable to connect to downstream service. - 500 if unable to complete the HTTP request downstream and the exception is not OperationCanceledException or HttpRequestException. - 404 if Ocelot is unable to map an internal error code to a HTTP status code.

CHAPTER

THIRTY

OVERVIEW

This document summarises the build and release process for the project. The build scripts are written using Cake, and 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.

CHAPTER

THIRTYONE

BUILDING

- You can also just run *dotnet tool restore && dotnet cake* locally!. Output will got 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 targers 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 6.0 SDK.

CHAPTER

THIRTYTWO

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.

CHAPTER

THIRTYTHREE

RELEASE PROCESS

- The release process works best with Git Flow branching.
- Contributors 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.
2. User creates a fork and branches from this (unless a member of core team, they can just create a branch on the main repo) e.g. feat/xxx, fix/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. I don't care as long as it has (feat, fix, refactor)/xxx :)
3. When the user is happy with their work they can create a pull request against develop in GitHub with their changes. The user must follow the SemVer support for this is provided by GitVersion. So if you need to make breaking changes please make sure you use the correct commit message so GitVersion uses the correct semver tags. Do not manually tag the Ocelot repo this will break things.
4. **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 user 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 occured.

- 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.
- Build must not have slowed down dramatically.
- The main Ocelot package must not have taken on any non MS dependencies.

5. After the PR is merged to develop the Ocelot NuGet packages will not be updated until a release is created.
6. When enough work has been completed to justify a new release. Develop will be merged into main the release process will begin which builds the code, versions it, pushes artifacts to GitHub and NuGet packages to NuGet.
7. The final step is to go back to GitHub and close any issues that are now fixed. You should see something like this in`GitHub <https://github.com/ThreeMammals/Ocelot/releases/tag/13.0.0>`_ and this in NuGet.

33.1 Notes

All NuGet package builds & releases are done with CircleCI *here <https://circleci.com/gh/ThreeMammals>*_ and all releases are done from *here <https://ci.appveyor.com/project/TomPallister/ocelot-ayj4w>*_.

Only TomPallister 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.