Esquio

Release 0.0.1

Aug 02, 2021

About Esquio

Esquio is a Feature Toggles (aka Feature Flags) and A/B testing library for ASP.NET Core 3.0. Feature toggling is a powerful technique that allows developers to deliver new functionality to users without changing code. Feature toggles provide an alternative to mantaining multiple branches (aka feature branches), so any feature can be tested even before it is completed and ready for the release. We can release a version of our product without production-ready features. These non production-ready features are hidden (toggled) for the broader set of users but can be enabled to any subset of testing or internal users we want to try out the features. We can even use feature toggling to enable or disable features during runtime.

Esquio is built with the possibility of use it not only in ASP.NET Core 3.0 in mind, but making it possible to use also in other .NET Core 3.0 projects like workers, webjobs, classlibraries, … almost any kind of .NET Core 3.0 project. For the Esquio team, this is not only about using a library, but using a full Feature Toggles framework for all of our projects, and as a delivery mechanism.

We believe Feature Toggling is, somekind, a way of delivering software, making it a first class citizen in your DevOps processes, therefore we are working hard towards integrating it, via extension and pipelines tasks, with Azure DevOps, so you can use Esquio Toggles directly in your releases and delivery flows. Having a full toggle delivery experience.

Esquio Azure DevOps extensions are built in top of the Esquio API, in the case you need to integrate Esquio with any other tool, you can always use this API to handle the toggles.

Additionally, if you need it, Esquio has a full UI developed, so you can be able to handle all your Toggles in it, making it fairly simple to use and manage.

CHAPTER 1

Terminology

The documentation and object model use a certain terminology that you should be aware of.

1.1 Product

Allows you to manage multiple different software projects, for example, one solution can contains a web application and windows application that need the same set of features. Each product has its own unique set of features.

1.2 Feature

Features are characteristics of your product that describe its appearance, components, and capabilities. A feature is a slice of business functionality that has a corresponding benefit or set of benefits for that product's end user. Each feature has its own set of toggles.

1.3 Toggle

Toggles allows you to control when a feature is enabled or not. Esquio provides many toggles out-of-the-box such us percentage rollouts, target specific users or environments, expiration dates or even hit the 'kill' switch for a feature programmatically.

1.4 Parameter

Parameters are variables that toggles need in their validation process.

1.5 Deployment

Deployments are the different environments or tenants where you deploy a Product.

1.6 Store

A mechanisim to allow you to store persistent the Esquio's object model such us products, features, toggles, parameters. Esquio provides out of the box two stores:

- ASP.NET Core JSON Configuration Provider.
- Http Provider to use with Esquio UI.

CHAPTER 2

Contributing

When contributing to this repository, please first discuss the change you wish to make via issue, email, or any other method with the owners of this repository before making a change.

Please note we have a code of conduct, please follow it in all your interactions with the project.

CHAPTER 3

Pull Request Process

1. Ensure any install or build dependencies are removed before the end of the layer when doing a build.
2. Update the README.md with details of changes to the interface, this includes new environment variables, exposed ports, useful file locations and container parameters.
3. Increase the version numbers in any examples files and the README.md to the new version that this Pull Request would represent. The versioning scheme we use is [SemVer](http://semver.org/).
4. You may merge the Pull Request in once you have the sign-off of two other developers, or if you do not have permission to do that, you may request the second reviewer to merge it for you.

CHAPTER 4

Code of Conduct

4.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

4.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

4.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

4.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at [INSERT EMAIL ADDRESS]. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

4.6 Attribution

This Code of Conduct is adapted from the [Contributor Covenant][homepage], version 1.4, available at version

homepage

version

CHAPTER 5

Getting started with Esquio in .NET Core

In this article, we are going to see how easy it is to use Esquio in your .NET Core application using the NuGet packages provided by Xabaril.

> In samples/GettingStarted.ConsoleApp you'll find a complete Esquio example in ASP.NET Core.

5.1 Create a folder for your new project

Open a command prompt an run:

```
mkdir consoleapp
cd consoleapp
```

5.2 Create the project

To create the project type the following command using the .NET Core CLI:

```
dotnet new console
```

5.3 Installation

To install Esquio type the following command:

```
dotnet package add Microsoft.Extensions.DependencyInjection
dotnet package add Microsoft.Extensions.Logging.Console
dotnet package add Microsoft.Extensions.Configuration.Json
dotnet package add Esquio.Configuration.Store
dotnet restore
```

or using Powershell or Package Manager:

```
Install-Package Microsoft.Extensions.DependencyInjection
Install-Package Microsoft.Extensions.Logging.Console
Install-Package Microsoft.Extensions.Configuration.Json
Install-Package Esquio.Configuration.Store
```

5.4 Setup

In the Program.cs, change the Main method to async:

```
static async Task Main(string[] args)
```

Import the configuration namespace:

```
using Microsoft.Extensions.Configuration;
```

Create the configuration object:

```
var configuration = new ConfigurationBuilder()
  .SetBasePath(Directory.GetCurrentDirectory())
  .AddJsonFile("appsettings.json")
  .Build();
```

Add a `appsettings.json` file to the project and add the content below:

```
{
  "Esquio": {
    "Products": [
      {
        "Name": "Console",
        "Features": [
          {
            "Name": "Colored",
            "Enabled": true,
            "Toggles": []
          }
        ]
      }
    ]
  }
}
```

Register the Esquio services and the configuration store:

```
var services = new ServiceCollection()
  .AddLogging(configure => configure.AddConsole())
  .AddEsquio()
  .AddConfigurationStore(configuration, "Esquio")
  .Services;
```

Build the service provider:

```
var serviceProvider = services.BuildServiceProvider();
```

Obtain a context for the `IFeatureService`:

```
var featureService = serviceProvider.GetService<IFeatureService>();
```

Check if “Colored” feature for the “Console” project is enabled and set the background color of the console to blue:

```
if (await featureService.IsEnabledAsync("Colored"))
{
  Console.BackgroundColor = ConsoleColor.Blue;
}
```

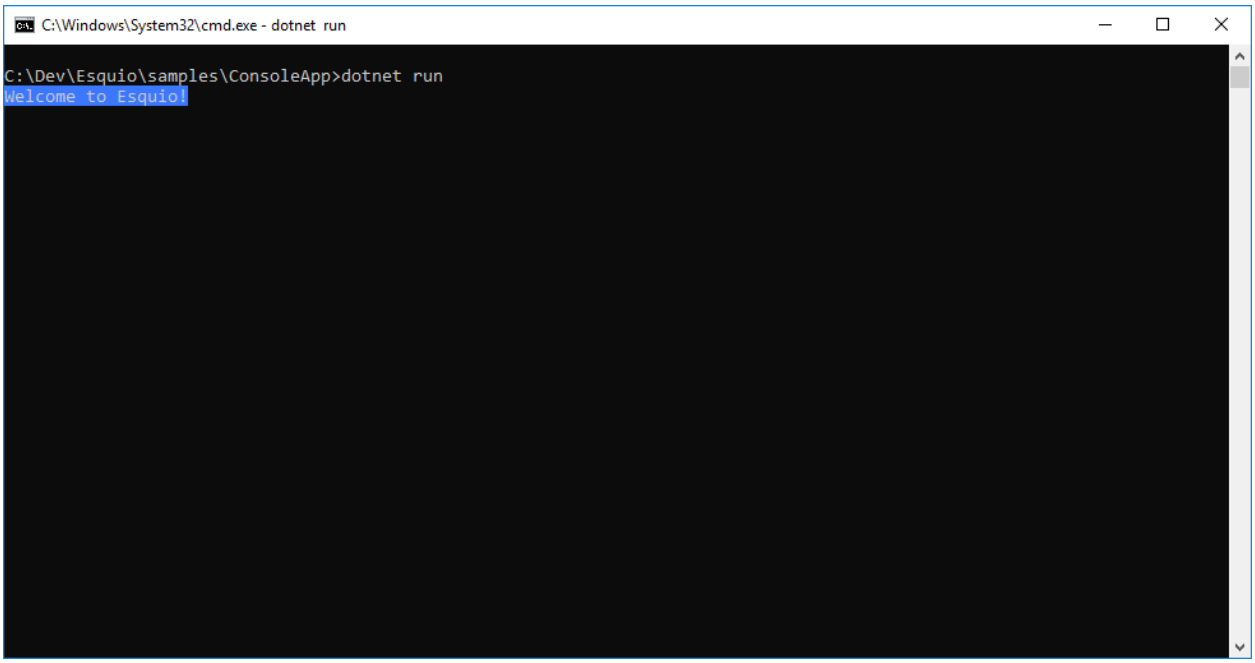
Finally, write a message to check it:

```
Console.WriteLine("Welcome to Esquio!");
Console.Read();
```

5.5 Test the application

Test the app on your computer running:

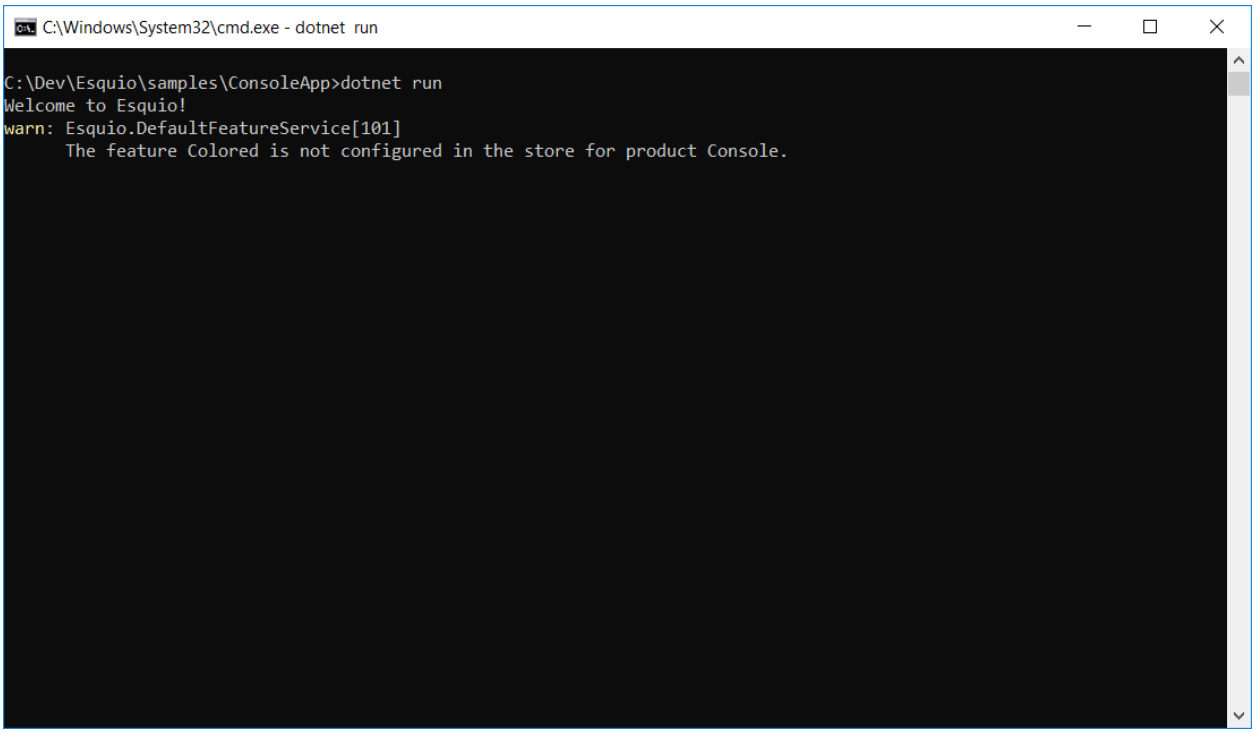
```
dotnet run
```



To disable the feature, change the `appsettings.json`:

```
"Enabled": false
```

And test again the app:


C:\Windows\System32\cmd.exe - dotnet run
C:\Dev\Esquio\samples\ConsoleApp>dotnet run
Welcome to Esquio!
warn: Esquio.DefaultFeatureService[101]
The feature Colored is not configured in the store for product Console.

CHAPTER 6

Getting started with Esquio in ASP.NET Core

In this article, we are going to watch an incremental sample to show how to use to use Esquio in your ASP.NET Core application.

6.1 Intro

> In samples/GettingStarted.AspNetCore.Intro you'll find this example in ASP.NET Core.

First, create a new project empty:

```
dotnet new web -n GettingStarted.AspNetCore.Intro
```

Install `Esquio.AspNetCore` package, typing the following command using the .NET Core CLI:

```
dotnet add package Esquio.AspNetCore
```

or using Powershell or Package Manager:

```
Install-Package Esquio.Configuration.Store
Install-Package Esquio.AspNetCore
```

or install via NuGet.

In the **ConfigureServices** method of Startup.cs, register the Esquio services:

```
services
  .AddEsquio()
  .AddAspNetCoreDefaultServices();
```

`AddEsquio` registers the default services for Esquio `AddAspNetCoreDefaultServices` register the default ASP.NET Core services for Esquio (i.e. claims or environment services)

6.2 Setting ConfigurationStore

Let's start using our `appsettings.json` to configurate Esquio. We don't recommend this store for production, only for small projects or for testing purposes.

Install `Esquio.AspNetCore` package, typing the following command using the .NET Core CLI:

```
dotnet add package Esquio.Configuration.Store
```

or using Powershell or Package Manager:

```
Install-Package Esquio.Configuration.Store
```

or install via NuGet.

And register the specific service for this store:

```
public class Startup
{
    IConfiguration _configuration;

    public Startup(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        services
            .AddEsquio()
            .AddAspNetCoreDefaultServices()
            .AddConfigurationStore(_configuration);
    }
```

`AddConfigurationStore` method registers the configuration store to use, in this case, based on the default configuration system of ASP.NET Core

So, let's open our `appsettings.json` file. To help us in this task, we can use the Esquio schema, selecting it on the `Schema` options:

Add the content below to your `appsettings.json` file:

```
{
  "Esquio": {
    "Products": [
      {
        "Name": "default",
        "Features": [
          {
            "Name": "HiddenGem",
            "Enabled": true,
            "Toggles": []
          }
        ]
```

(continues on next page)

(continued from previous page)

```
        }
      ]
    }
}
```

By default, `Esquio` will be the root element. However, you could change it on adding the configurationStore:

```
.AddConfigurationStore(_configuration, key: "MyNewCustomRoot");
```

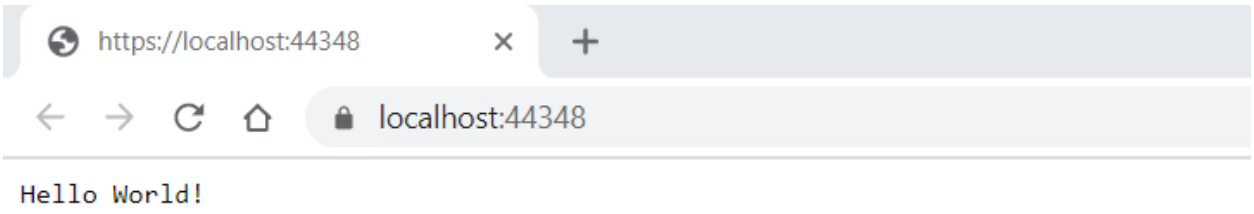
With this configuration, we are defining a new `feature` named `HiddenGem`, initially set to enabled, but with no toggle on it.

In order to test it, let's use the current endpoint already defined on the method `Configure` on the class `Startup`. We can attach feature metadata to an endpoint using the route mappings configuration fluent API `RequireFeature` method:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    }).RequireFeature("HiddenGem");
});
```

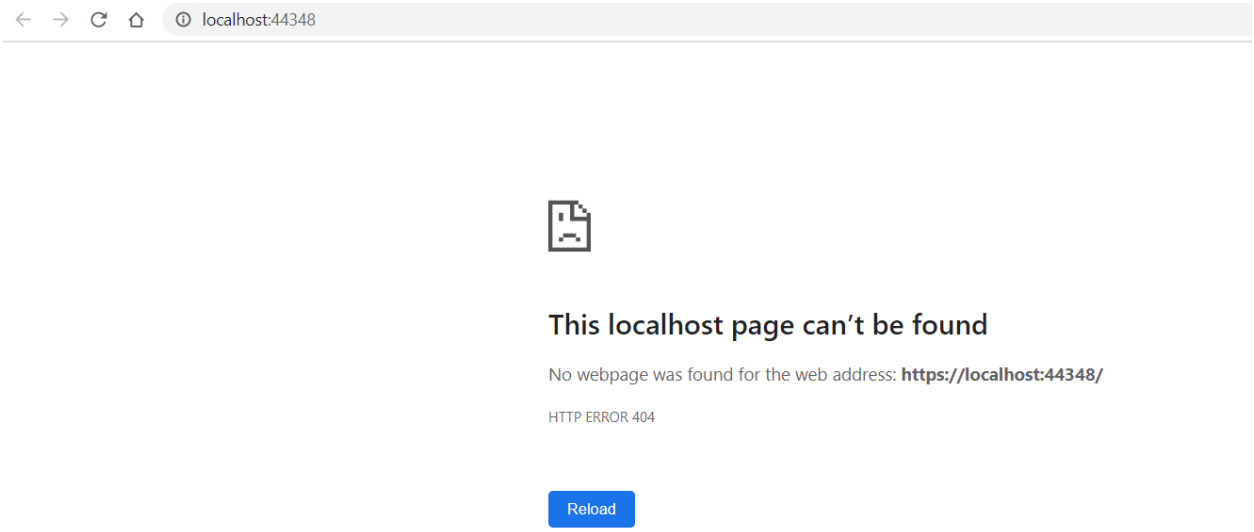
This method will filter if the endpoint can be executed depending on feature(s) state. If the configured feature is enabled this endpoint is executed, if not, by default a NotFound result is obtained.

So, if we run the project with this configuration, we will reach this endpoint:



However, let's modify the configuration file again, setting `Enabled` property to false, and refresh the browser:

```
"Features": [
  {
    "Name": "HiddenGem",
    "Enabled": false,
    "Toggles": []
  }
]
```



6.3 Creating a client endpoint

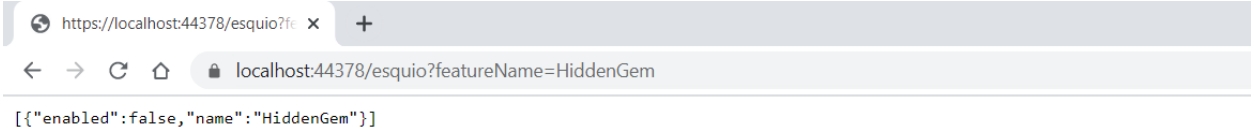
> In samples/GettingStarted.AspNetCore.ClientEndpoint you'll find this example in ASP.NET Core.

You can query Esquio with your own api request. To enable this endpoint, from the previous code, let's come back to the `Configure` method of the `Startup` class:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    }).RequireFeature("HiddenGem");

    endpoints.MapEsquio();
});
```

With `MapEsquio` a new endpoint is mapped that can be used to get the activation state of any configured feature. An Uri pattern can be specify, if not, `esquio` would be the default pattern.

So, after running your project, you can open a browser and go to `/esquio?featureName=HiddenGem`:



6.4 Exploring more options

> In samples/GettingStarted.AspNetCore.IntroOptions you'll find this example in ASP.NET Core.

Let's explore some configuration options that `Esquio` provides us.

We can configurate what would be the result of evaluating a feature that cannot be found or whom evaluation returns an error.

So, let's back again to the `ConfigureServices` and set the behaviour of *NotFound* and *OnError* to *SetDisable*. This is the value by default, so for the moment these configuration doesn't change anything. Add also a new fallback endpoint to verify if the fallback is executed or not:

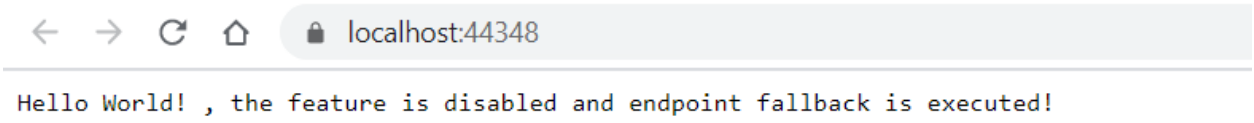
```
services
    .AddEsquio(options =>
    {
        options.ConfigureNotFoundBehavior(NotFoundBehavior.SetDisabled);
        options.ConfigureOnErrorBehavior(OnErrorBehavior.SetDisabled);
    })
    .AddEndpointFallback(new RequestDelegate(async context =>
    {
        await context.Response.WriteAsync("Hello World! , the feature is disabled and␣
↪endpoint fallback is executed!");
    }))
```

On *Configure* method, let's modify the endpoint to call to require a feature that has not been configured (*NotExistingFeature* instead of *HiddenGem*):

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    }).RequireFeature("NonExistingFeature");
});
```

> You can specify many features separated by comma, so you can restrict access to the endpoints if a feature or a group features are enabled or not.

If you launch again the project, you will get the fallback message:



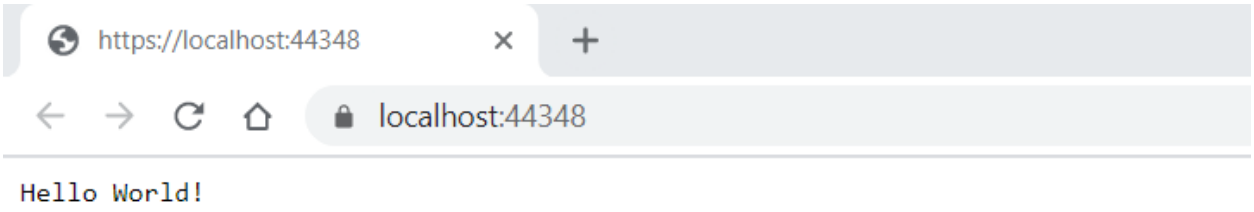
We could also use any already defined fallback actions, instead of creating the request delegate directly. In this case, on fallback we want to be redirected to Google page:

```
.AddEsquio(options =>
{
    options.ConfigureNotFoundBehavior(NotFoundBehavior.SetDisabled);
    options.ConfigureOnErrorBehavior(OnErrorBehavior.SetDisabled);
})
.AddEndpointFallback(EndpointFallbackAction.RedirectTo("https://www.google.com"))
```

In case we would like `Esquio` to evaluate as enabled a feature that doesn't exist, we can change *NotFound* behaviour:

```
services
    .AddEsquio(options =>
    {
        options.ConfigureNotFoundBehavior(NotFoundBehavior.SetEnabled);
        options.ConfigureOnErrorBehavior(OnErrorBehavior.SetDisabled);
    })
```

If we run the project, we get again our normal endpoint:



6.5 ASP.NET Core Web Apps

When working with Esquio you can attach feature metadata to an endpoint. We do this using the route mappings configuration fluent API `RequireFeature` method:

```
app.UseEndpoints(routes =>
{
    routes.MapControllerRoute(
      name: "default",
      pattern: "{controller=Match}/{action=Index}/{id?}").RequireFeature("HiddenGem");
});
```

You can specify many features separated by comma, so you can restrict access to the endpoints if a feature or a group features are enabled or not.

If you want more fine-grained control over your Controllers, Esquio provides a `FeatureFilter` attribute that forces you to supply a comma separated list of features names. You can specifies that access to a controller or action method is restricted to users if theses features are enabled or not:

```
[FeatureFilter(Names = Flags.MinutesRealTime)]
public IActionResult DetailLive()
{
    return View();
}
```

Also, you can use `FeatureFilter` to act as an Action constraint. You can create two Actions with the same `ActionName` and decorate one with `FeatureFilter` attribute to match the action only when the predefined feature name values are enabled or not.:

```
[ActionName("Detail")]
public IActionResult DetailWhenFlagsIsNotActive()
{
```

(continues on next page)

(continued from previous page)

```
    return View();
}

[FeatureFilter(Names = Flags.MinutesRealTime)]
[ActionName("Detail")]
public IActionResult DetailWhenFlagsIsActive()
{
    return View();
}
```

Sometimes you will need to configure a fallback action. Esquio provides an `AddEndpointFallback` method that accepts a `RequestDelegate` in order to configure your custom fallback:

```
services
  .AddEsquio()
  .AddAspNetCoreDefaultServices()
  .AddConfigurationStore(Configuration, "Esquio")
  .AddEndpointFallback((context) =>
  {
      context.Response.StatusCode = StatusCodes.Status404NotFound;

      return Task.CompletedTask;
  })
```

Out-of-the-box Esquio provides `EndpointFallbackAction` class that defines common fallback actions to be used when no matching endpoints found:

- Redirect result to MVC action:

```
public static RequestDelegate RedirectToAction(string controllerName, string␣
↪actionName)
```

- Redirect result:

```
public static RequestDelegate RedirectTo(string uri)
```

- NotFound status response:

```
public static RequestDelegate NotFound()
```

6.6 ASP.NET Core MVC

With **ASP.NET MVC Core** we can use the `FeatureTagHelper` inside our Razor views to show or hide Razor fragments depending on feature is enabled or not.

```
<feature names="@Flags.MatchScore">
    <span class="badge badge-secondary badge-pill">@match.ScoreLocal - @match.
↪ScoreVisitor</span>
</feature>
```

In this example, if the feature **MatchScore** is enabled, you can show a new design of the match score. Names property is comma-separated list of feature names to be evaluated. If any feature is not active, the tag helper will suppress the content.

The `FeatureTagHelper` supports `Include` and `Exclude` attributes:

- Include: *A comma-separated list of feature names to be evaluated. If any feature is not active, this tag helper suppresses the content.*
- Exclude: *A comma-separated list of feature names to be evaluated. If any feature is active, this tag helper suppresses the content.*

Make sure you add the taghelper on your `_ViewImports.cshtml` file with:

```
...
@using Esquio
@using Esquio.Abstractions
@using Esquio.AspNetCore
...
@addTagHelper *, Esquio.AspNetCore
```

6.7 SPA and Native Apps

Single-Page-Applications and native apps are becoming the new wave for modern applications. The challenge with feature flags in these kinds of applications is handling the state transformations. In case of SPAs the changes in a webpage's DOM and the platform specific controls in native apps. We will need an endpoint to query if a feature or a set of features are enabled or not in order make real time personalization in the UX for example.

To enable this endpoint, in the `Configure` method, insert the middleware to expose the Esquio endpoint:

```
app.UseEndpoints(routes =>
{
    routes.MapEsquio(pattern: "esquio");
});
```

Now you can start your application and check out your features at http(s)://server:port/esquio?featureName=Colored:

```
[
  {
    "enabled": true,
    "name": "Colored"
  }
]
```

To disable the feature, change the `appsettings.json`:

```
"Enabled": false,
```

Test again the app:

```
[
  {
    "enabled": false,
    "name": "Colored"
  }
]
```

CHAPTER 7

Getting started with Esquio in .NET Core Worker

In this article, we are going to see how easy it is to use Esquio in your .NET Core worket using the NuGet packages provided by Xabaril.

> In samples/GettingStarted.Worker you'll find a complete Esquio example in a .Net core worker.

In this sample, a different message will be shown according with the state of the feature named `ComputeMatch`. This feature will be store in the configuration.

7.1 Create the project

To create the project, type the following command using the .NET Core CLI:

```
dotnet new worker -n GettingStarted.Worker
cd GettingStarted.Worker
```

7.2 Installation

To install Esquio, type the following command:

```
dotnet add package Esquio
dotnet add package Esquio.Configuration.Store
dotnet restore
```

or using Powershell or Package Manager:

```
Install-Package Esquio
Install-Package Esquio.Configuration.Store
```

> In addition to Esquio package, it will install the specific package for using `configuration` as the Esquio store.

7.3 Setup

In the `Program.cs`, modify the `CreateHostBuilder` to register Esquio services and the specific configuration store:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureServices((context, services) =>
        {
            services
                .AddEsquio()
                .AddConfigurationStore(context.Configuration, key: "Esquio");

            services.AddHostedService<Worker>();
        });
```

Modify the `appsettings.json` file to the project and add the content below:

```
"Esquio": {
  "Products": [
    {
      "Name": "default",
      "Features": [
        {
          "Name": "ComputeMatch",
          "Enabled": false,
          "Toggles": []
        }
      ]
    }
  ]
}
```

Let's consume this feature. In the `Worker.cs`, create a new property `scopeFactory` and change the `Worker` method to receive it as a parameter:

```
private readonly ILogger<Worker> _logger;
private readonly IServiceScopeFactory _scopeFactory;

public Worker(IServiceScopeFactory scopeFactory, ILogger<Worker> logger)
{
    _scopeFactory = scopeFactory;
    _logger = logger;
}
```

Import the `DependencyInjection` extension namespace:

```
using Microsoft.Extensions.DependencyInjection;
```

Change the method `ExecuteAsync` for creating a new context and require a `IFeatureService`:

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        using (var scope = _scopeFactory.CreateScope())
        {
```

(continues on next page)

(continued from previous page)

```
            var featureService = scope.ServiceProvider
                .GetRequiredService<IFeatureService>();
        }
```

Import the `Esquio.Abstractions` namespace for resolving `IFeatureService`:

```
using Esquio.Abstractions;
```

Using this service, you can implement the logic to show a different message on the console according with the feature in several ways.

You can use it querying the feature and receiving a boolean result depending on its state:

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        using (var scope = _scopeFactory.CreateScope())
        {
            var featureService = scope.ServiceProvider.GetRequiredService
↪<IFeatureService>();

            if (await featureService.IsEnabledAsync("ComputeMatch"))
            {
                _logger.LogInformation("Worker running with ComputeMatch Feature␣
↪enabled at: {time}", DateTimeOffset.Now);
            }
            _logger.LogInformation("Worker running at: {time}", DateTimeOffset.Now);

            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

Or using the method *Do* where you can explictly define both states in different actions:

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        using (var scope = _scopeFactory.CreateScope())
        {
            var featureService = scope.ServiceProvider
                .GetRequiredService<IFeatureService>();

            await featureService.Do("ComputeMatch",
                enabled: () =>
                {
                    _logger.LogInformation("Worker running with ComputeMatch Feature␣
↪enabled at: {time}", DateTimeOffset.Now);
                },
                disabled: () =>
                {
                    _logger.LogInformation("Worker running at: {time}",␣
↪DateTimeOffset.Now);
                });
```

(continues on next page)

(continued from previous page)

```
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

7.4 Test the application

Test the app on your computer running:

```
dotnet run
```

```
λ dotnet run
warn: Esquio[102]
      The feature ComputeMatch is disabled in the store for product default.
```

To enable the feature, change the `appsettings.json`:

```
"Enabled": true
```

And test again the app:

```
λ dotnet run
info: GettingStarted.Worker.Worker[0]
      Worker running with ComputeMatch Feature enabled at: 04/27/2020 08:51:29 +02:00
```

CHAPTER 8

Getting started with Esquio Azure DevOps tasks

In this article, we are going to see how to configure Esquio Azure DevOps tasks for your pipelines.

> In samples/WebApp you'll find a complete Esquio example in ASP.NET Core.

8.1 Setup

The first step is install the Esquio Azure DevOps Task from Visual Studio Marketplace . There isn't anything special needed, just install it as a normal Azure DevOps extension.

Once installed, you will have new elements in your Azure DevOps:

- Esquio Service Connection
- Esquio rollout task
- Esquio rollback task
- Esquio set parameter value task

8.2 Prerequisites

To be able to use Esquio tasks, we will need to setup a Esquio Service Connection and to configure it you need to create an Esquio API key, to use it with the Esquio Service Connection.

8.3 Create Esquio API key

First of all you need to create an Esquio API key to create the Service Connection it is easily created with the Esquio UI.

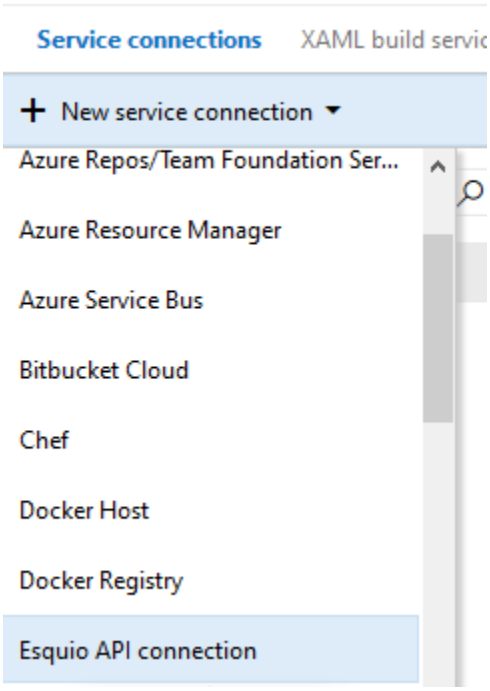
Open your current deployment of Esquio UI url in a browser, and once logged-in, click on your user name in the upper right corner.

Click on *Get Private token* and you will receive a confirmation window, and your token will be copied to the clipboard, so note it to paste it when configuring the Esquio Service Connection

Now you can continue to configure the Service Connection.

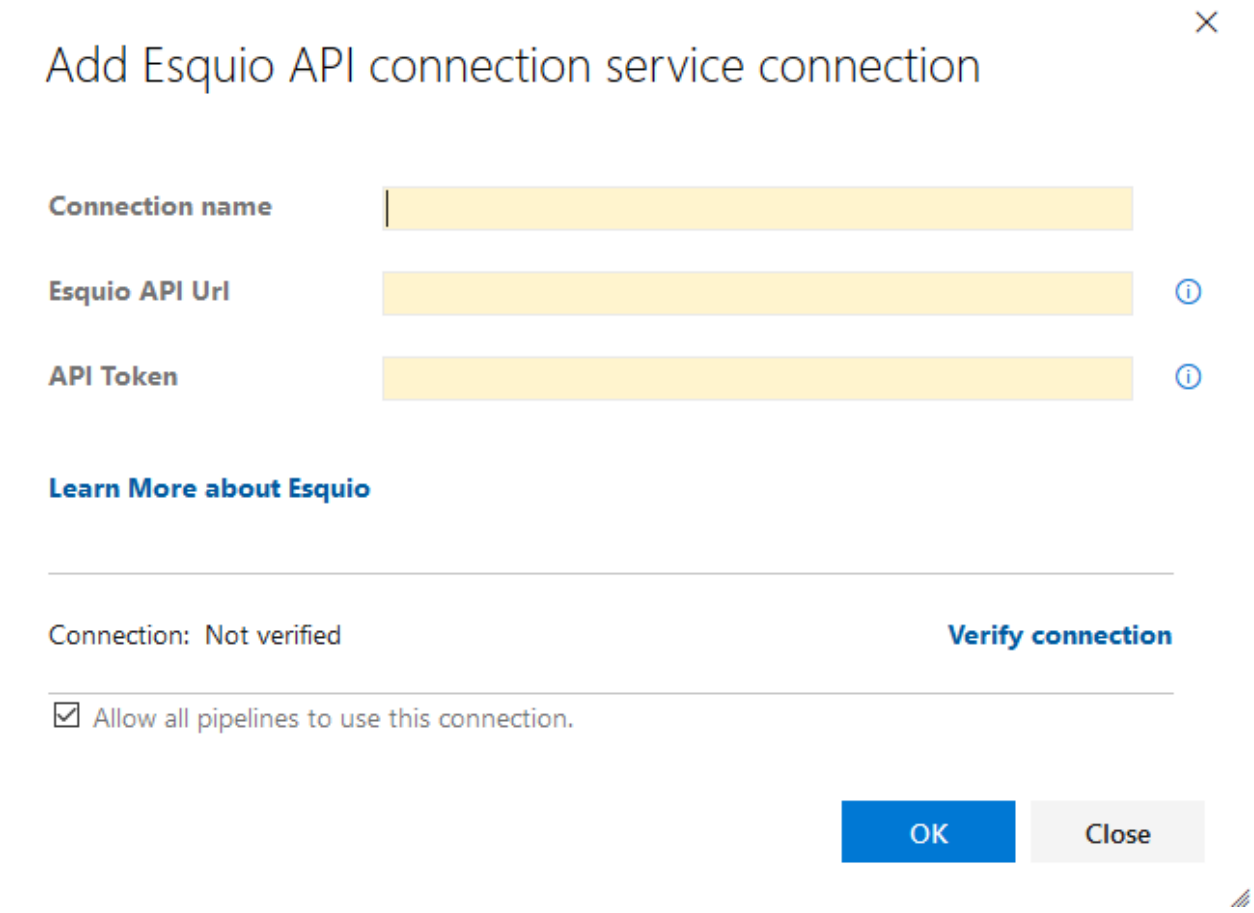
8.4 Setup Esquio Service Connection

Service Connection are setup per project, so open your Azure DevOps settings page, and go to Service Connections, click on *New service Connection* and select *Esquio API Connection*

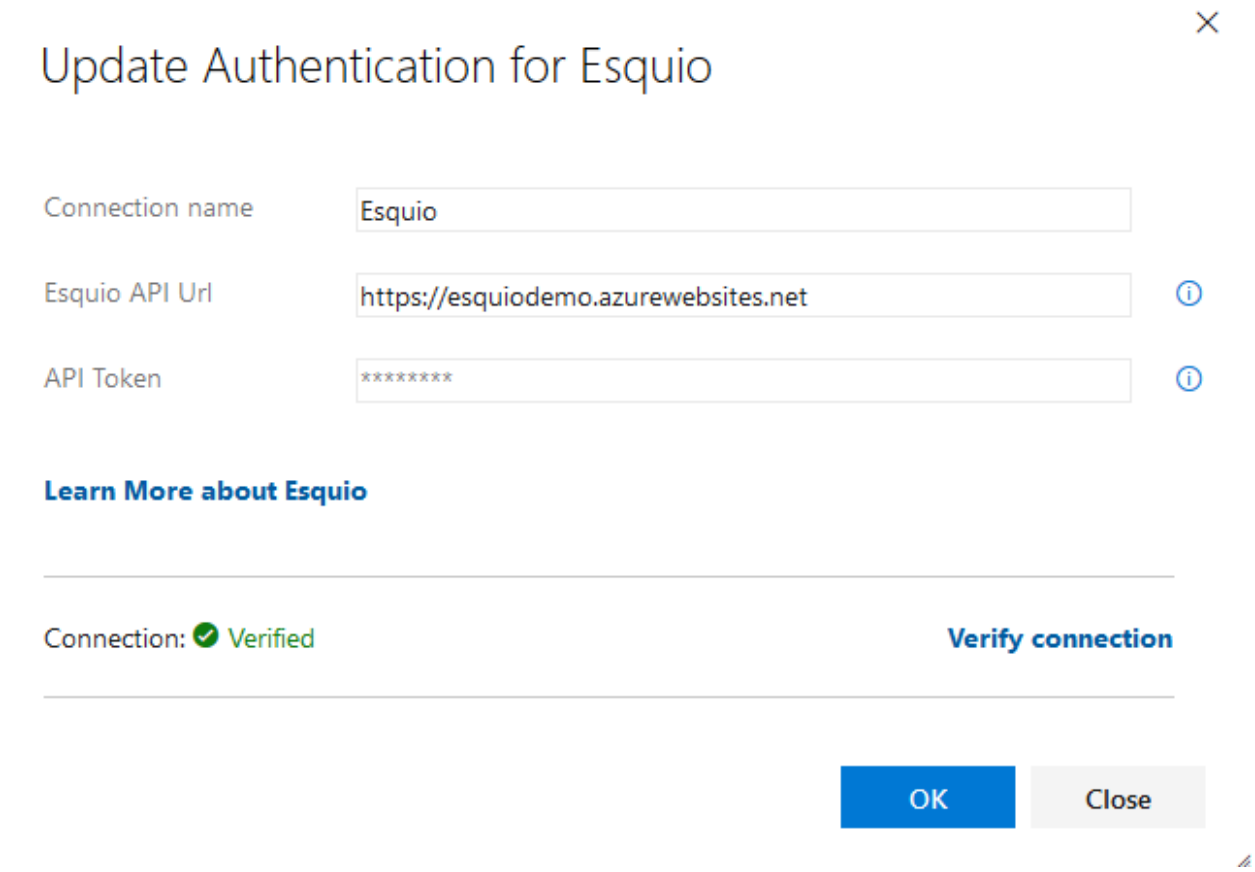


This will bring the *Esquio API Connection* configuration screen, here you need to setup three parameters:

- **Connection name:** To use it in the Azure DevOps tasks.
- **Esquio API Url:** The complete url in which yoy have your Esquio API.
- **API token:** The Esquio API key you have setup.



Once filled all the information, make sure it is correct, clicking on *Verify connection* and make sure it says *Connection: Verified*

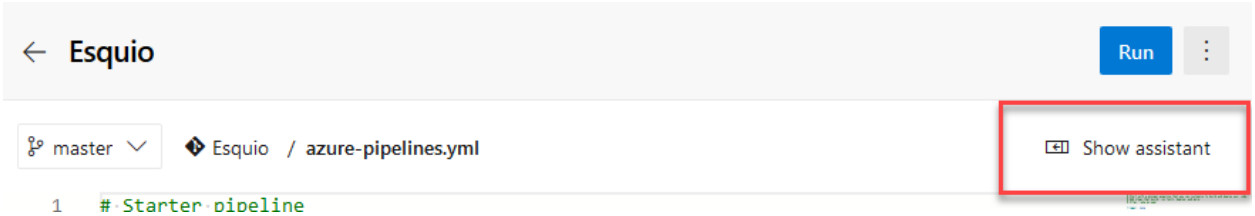


Now you have setup the Esquio Connection we will need to use for the tasks.

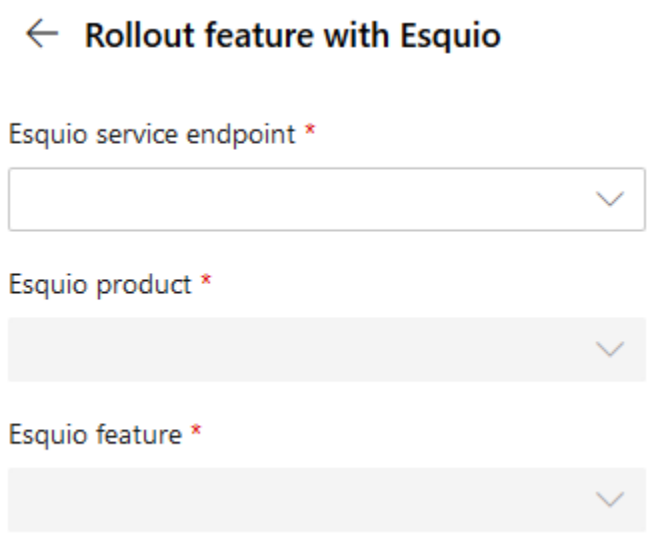
8.5 Esquio rollout task

This task allow us to enable a feature to everyone cleaning all the toggles.

If you are creating your Azure Pipelines with *YAML* it is better to use the *YAML assistant* as it will allow you to use the datasources for the picklists.



To setup the rollout task, look for **Rollout feature with Esquio** task:



We will configure three parameters:

- **Esquio Service Endpoint:** Select the previously created *Esquio Service Connection*.
- **Esquio Product:** From the list of products configured in Esquio.
- **Esquio feature:** Select, from the list of features.

The final YAML should be (with different ids) like this:

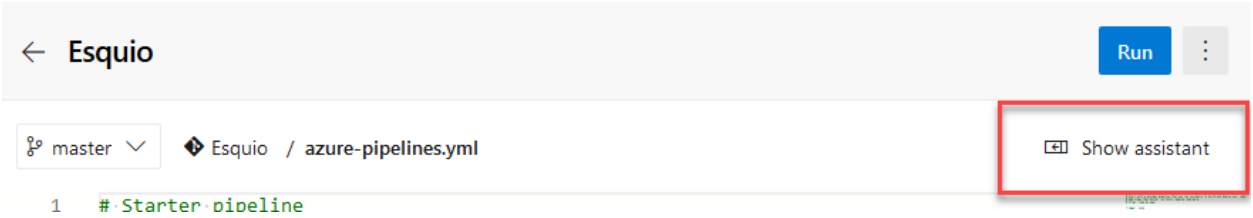
```
- task: esquio-rollout-feature@1
  inputs:
    EsquioService: 'Esquio'
    productId: '1'
    flagId: '1'
```

If you are using the classic pipelines (the visual ones), the setup is exactly the same.

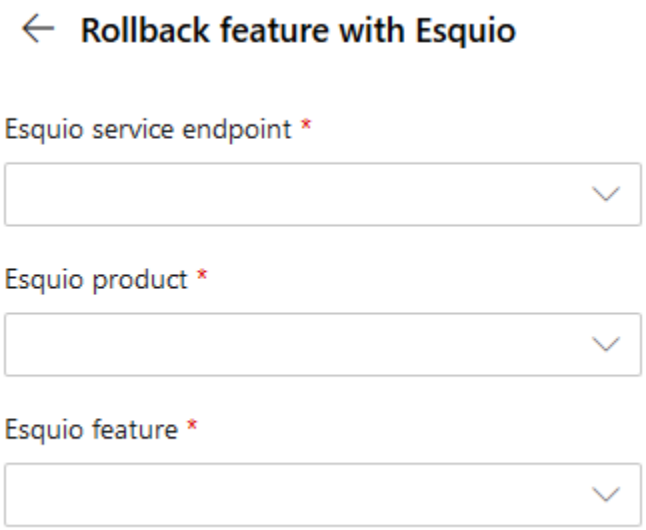
8.6 Esquio rollback task

This task allow us to set a *OffToggle* for a feature, thus disabling it.

If you are creating your Azure Pipelines with *YAML* it is better to use the *YAML assistant* as it will allow you to use the datasources for the picklists.



To setup the rollout task, look for **Rollback feature with Esquio** task:



We will configure three parameters:

- **Esquio Service Endpoint:** Select the previously created *Esquio Service Connection*.
- **Esquio Product:** From the list of products configured in Esquio, select the one with the feature you want to setup the *OffToggle*.
- **Esquio feature:** Select, from the list of features, the one to setup the *OffToggle*.

The final YAML should be (with different ids) like this:

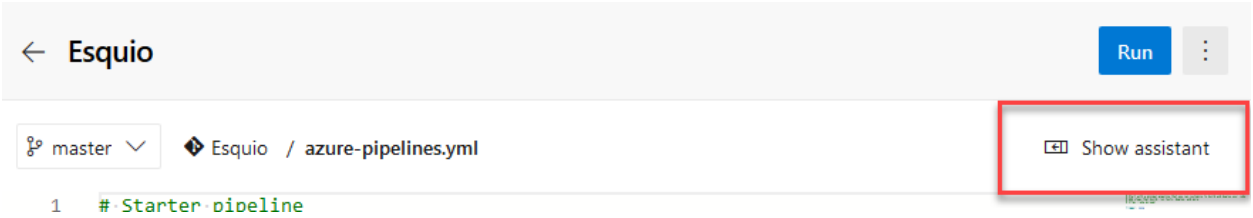
```
- task: esquio-rollback-feature@1
  inputs:
    EsquioService: 'Esquio'
    productId: '1'
    flagId: '1'
```

If you are using the classic pipelines (the visual ones), the setup is exactly the same.

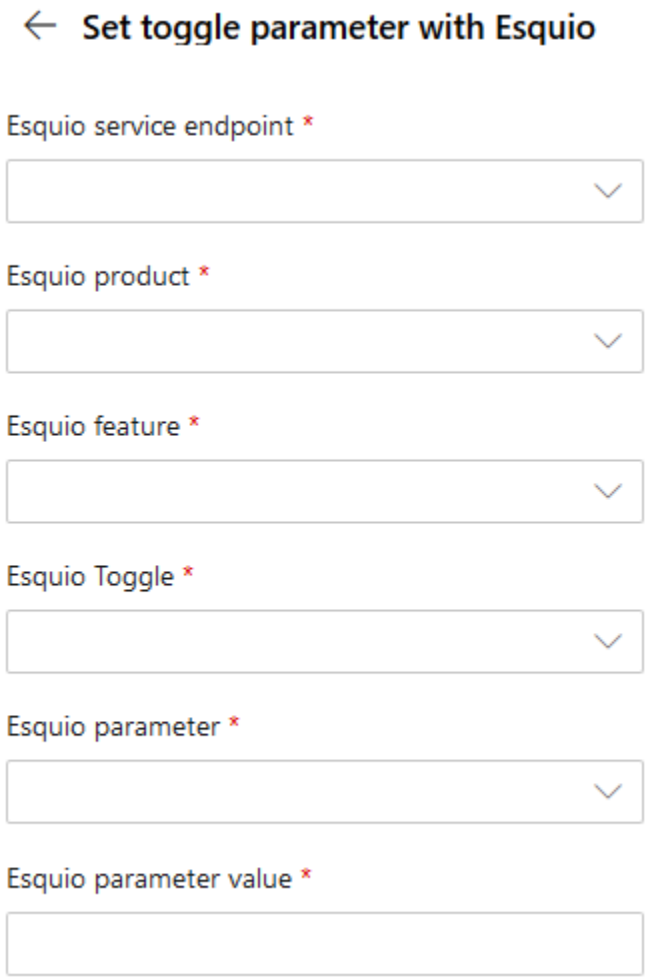
8.7 Esquio set toggle parameter task

This task allow us to set a value for a particular parameter in a toggle, with this task you can setup any other type of *toggle* acepting parameters.

If you are creating your Azure Pipelines with *YAML* it is better to use the *YAML assistant* as it will allow you to use the datasources for the picklists.



To setup the rollout task, look for **Set toggle parameter with Esquio** task and select, using the picklists, the parameter for the feature toggle you want to set, and then fill-in the value you want to set for the parameter:



We will configure six parameters:

- **Esquio Service Endpoint:** Select the previously created *Esquio Service Connection*.
- **Esquio Product:** From the list of products configured in Esquio.
- **Esquio feature:** Select, from the list of features, the one with the parameter you want to set the value.
- **Esquio toggle:** Select, from the list of toggles, the one with the parameter you want to set the value.
- **Esquio parameter:** Select, from the list of parameters for the previously selected toggle, the one you want to set the value.
- **Esquio parameter value:** Introduce manually the value you want to setup for the parameter.

The final YAML should be (with different ids) like this:

```
- task: set-toggle-parameter@1
  inputs:
    EsquioService: 'esquio'
    productId: '1'
    flagId: '2'
    toggleId: '14'
    parameterId: 'Percentage'
    parameterValue: '59'
```

If you are using the classic pipelines (the visual ones), the setup is exactly the same.

CHAPTER 9

Getting started with Esquio monitoring & diagnostics

In this article, we are going to watch an incremental sample to show how to use to use Esquio in your ASP.NET Core application.

9.1 Intro

9.2 Application Insights

> In samples/GettingStarted.AspNetCore.Mvc.ApplicationInsights you'll find this example in ASP.NET Core.

An easy way to discover what is happening in our application is using *ApplicationInsights*.

First, install `Microsoft.ApplicationInsights.AspNetCore` package, typing the following command using the .NET Core CLI:

```
dotnet add package Microsoft.ApplicationInsights.AspNetCore
```

or using Powershell or Package Manager:

```
Install-Package Microsoft.ApplicationInsights.AspNetCore
```

or install via NuGet.

On the *Startup'class, modify your 'ConfigureServices* method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews()
        .Services
        .AddEsquio(setup =>
        {
            setup.UseScopedEvaluation(useScopedEvaluation: true);
        })
```

(continues on next page)

(continued from previous page)

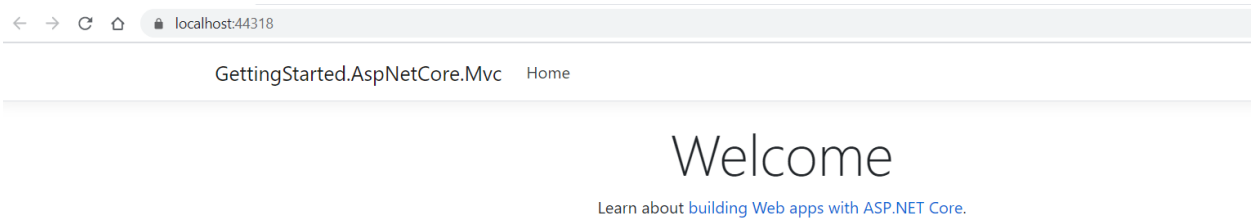
```
        .AddAspNetCoreDefaultServices()
        .AddApplicationInsightProcessor()
        .AddConfigurationStore(Configuration);
}
```

> We set on the configuration *ScopedEvaluation* to use evaluation session, that is, to store feature evaluation results on the same execution scope. This ensures result consistency on the same scope. By default, a no evaluation holder is used and scoped evaluation results are never stored and reused.

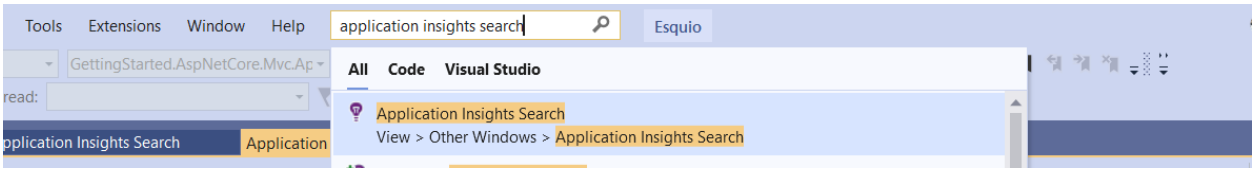
To register *ApplicationInsights* telemetry and *Esquio* customized processor, we had just added the following extension:

```
.AddApplicationInsightProcessor()
```

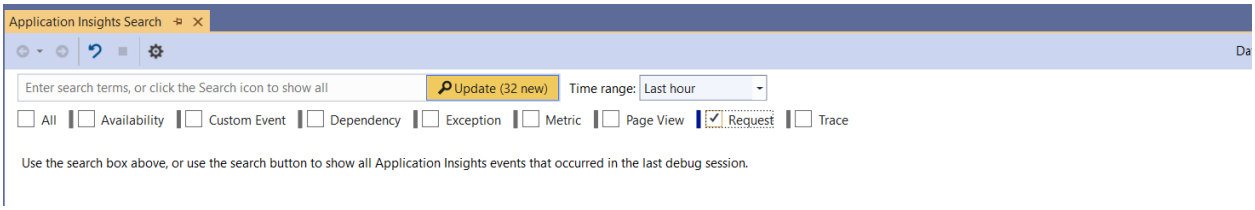
Open this sample and launch it. It will open a very simple Mvc application.



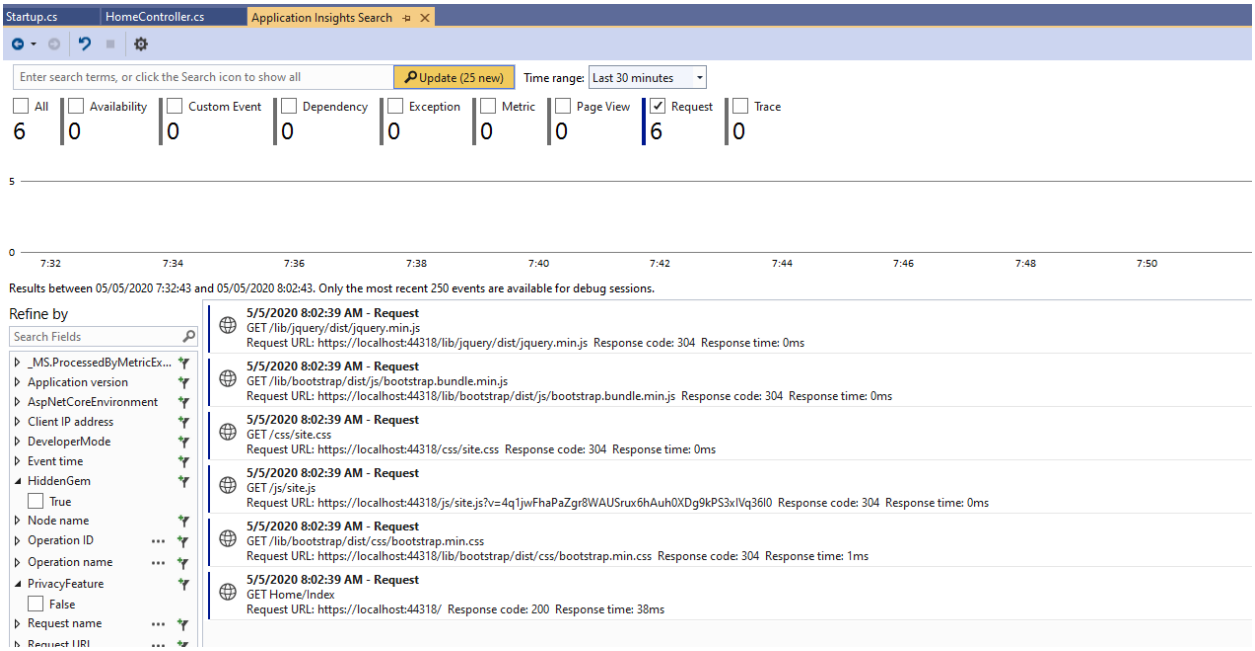
Let's open *ApplicationInsights* watcher typing *Application Insights Search* on the *Search* field of your Visual Studio or clicking on *View*> *Other Windows*> *Application Insights Search*:



In this sample, we are interested on the *Request*, so select only this option and click on *Update* button:



On the results, same requests should be shown. On the left, among the fields to refine by, you can see *HiddenGem* and *PrivacyFeature*:



Let's go to the *appsettings.json*, enable the *PrivacyFeature*:

```
"Esquio": {
  "Products": [
    {
      "Name": "default",
      "Features": [
        {
          "Name": "PrivacyFeature",
          "Enabled": true,
          "Toggles": []
        },
        {
          "Name": "HiddenGem",
          "Enabled": true,
          "Toggles": []
        }
      ]
    }
  ]
}
```

Launch the application and open *Application Insights Search* again. A new value for *PrivacyFeature* filter appears.

In this way, you can filter among all your requests showing only the request when *PrivacyFeature* is enable. This can be very useful on canary release to be able to detect undesirable behaviours.

9.3 Dotnet Counter

> Counters

9.4 VisualStudio Diagnostics

9.5 PerfWatcher

CHAPTER 10

Behaviors

Esquio allows you to configure their behaviors when a feature does not exists or fails during evaluation. You can configure these behaviors by modifying the Esquio setup.

10.1 OnErrorBehavior

There are three options to configure when a feature fails during evaluation:

- **OnErrorBehavior.Throw**: Re-throw the exception.
- **OnErrorBehavior.SetDisabled**: Returns disabled as a result of the evaluation.
- **OnErrorBehavior.SetEnabled**: Returns enabled as a result of the evaluation.

The `AddEsquio` method provides you a way to configure the behavior when a feature fails during evalution:

```
services
  .AddEsquio(setup => setup.ConfigureOnErrorBehavior(OnErrorBehavior.Throw))
  .AddAspNetCoreDefaultServices()
  .AddConfigurationStore(Configuration, "Esquio");
```

In the above example, the exception will be thrown if some fail happens during the evalution process.

10.2 NotFoundBehavior

There are three options to configure when a feature does not exists in the store:

- **NotFoundBehavior.SetDisabled**: Returns disabled as a result of the evaluation.
- **NotFoundBehavior.SetEnabled**: Returns enabled as a result of the evaluation.

The `AddEsquio` method provides you a way to configure the behavior when a feature does not exists in the store:

```
services
  .AddEsquio(setup => setup.ConfigureNotFoundBehavior(NotFoundBehavior.SetDisabled))
  .AddAspNetCoreDefaultServices()
  .AddConfigurationStore(Configuration, "Esquio");
```

In the above example, if the feature does not exists in the store, Esquio will returns disabled as a result of the feature evaluation process.

CHAPTER 11

Defaults

Esquio allows to configure the Product and Deployment using specifed methods on EsquioOptions.

11.1 ConfigureDefaultProductName

Allow to configure default Product name to use, the default value is 'default'

```
services.AddEsquio(setup=>
{
     setup.ConfigureDefaultProductName("default");
});
```

11.2 ConfigureDefaultDeploymentName

Allow to configure default Deployment name to use, the default value is 'Tests'

```
services.AddEsquio(setup=>
{
     setup.ConfigureDefaultDeploymentName("Tests");
});
```

CHAPTER 12

Scoped Evaluation

Esquio allows to cache, at scoped level (the request on ASP.NET Core), the evalution result of any feature. This reduce the number of evaluation, better performance, and improve the consistency of evaluation results on the same scope.

12.1 UseScopedEvaluation

Allow to set the use of scoped evaluation results, on ASP.NET Core the scope is the request and the results are stored on the HTTP Context Items property

```
services.AddEsquio(setup=>
{
     setup.UseScopedEvaluation(useScopedEvaluation: true);
});
```

12.2 Customize Scoped Evaluation Holder

You can override the *IScopedEvaluationHolder* and create your own evaluation result holder implementing the interface

```
public interface IScopedEvaluationHolder
{
     Task<bool> TryGetAsync(string featureName, out bool enabled);
     Task SetAsync(string featureName, bool enabled);
}
```

CHAPTER 13

Esquio toggles

A **Toggle** is piece of code that defines when a feature is enabled or not. Each feature can use one or more toggles at the same time, but never more than one toggle of the same type. In Esquio you have many different toggles out of the box, and of course you can write your custom toggles.

13.1 Environment Variable

This toggle enables the feature if the value of the configured environment variable is in the list. This environment information is provided by IEnvironmentNameProviderService. When you add Esquio to your application using `AddEsquio()` method, by default Esquio registers a NoEnvironmentNameProviderService. For ASP.NET Core projects, Esquio provides a method called `AddAspNetCoreDefaultServices` that registers by default an AspNetEnvironmentNameProviderService based on Microsoft.AspNetCore.Hosting.Abstractions.IWebHostEnvironment.

Type

- Esquio.Toggles.EnvironmentToggle

Parameters

- EnvironmentVariable: *The environment variable name.*
- Values: *The values to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesRealTime",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.Toggles.EnvironmentToggle",
            "Parameters":
            {
                "EnvironmentVariable": "ASPNETCORE_ENVIRONMENT",
                "Values": "Staging;Production"
            }
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

13.2 Between dates

This toggle enables the feature when current UTC date falls within the interval.

Type

- Esquio.Toggles.FromToToggle

Parameters

- From: *The interval start (yyyy-MM-dd HH:mm:ss) when this toggle is activated.*
- To: *The interval end (yyyy-MM-dd HH:mm:ss) when this toggle is activated.*

```
{
    "Name": "DarkMode",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.Toggles.FromToToggle",
            "Parameters":
            {
                "From": "2019-06-12 00:00:00",
                "To": "2019-06-14 23:59:59"
            }
        }
    ]
}
```

CHAPTER 14

Esquio ASP.NET Core toggles

In addition to the toggles that Esquio provides out of the box, Esquio.AspNetCore provides more toggles to work with ASP.NET Core applications.

14.1 Identity Claim Value

This toggle enables its feature if the identity claim of the current user exists and its value is in the list.

Type

- Esquio.Toggles.ClaimValueToggle

Parameters

- ClaimType: *The claim type name.*
- ClaimValues: *The claim values to activate this toggle separated by ';' character.*

```
{
    "Name": "AnimationsMatch",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.Toggles.ClaimValueToggle",
            "Parameters":
            {
                "ClaimType": "Company",
                "ClaimValues": "Contoso;ACME"
            }
        }
    ]
}
```

14.2 Client IP Address

This toggle enables its feature if the client IP address is in the list.

Type

- Esquio.Toggles.ClientIpAddressToggle

Parameters

- IpAddresses: *The IP addresses to activate this toggle separated by ';' character.*

```
{
    "Name": "SecretZoneMatch",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.Toggles.ClientIpAddressToggle",
            "Parameters":
            {
                "ClaimType": "IpAddresses",
                "ClaimValues": "11.22.44.88;11.22.33.44"
            }
        }
    ]
}
```

14.3 Partial rollout by Identity Claim value

This toggle enables its feature The claim exists and its value falls within the percentage created by Esquio Partitioner. Stickiness is based on the claim type value. Esquio uses Jenkins hash function that guarantees to the user get the same experience across many devices and also assures that a user which is among the first 30% will also be among the first 50% of the users.

Type

- Esquio.AspNetCore.Toggles.GradualRolloutClaimValueToggle

Parameters

- Percentage: *The percentage of users that activates this toggle. Percentage from 0 to 100.*
- ClaimType: *The identity claim type used whom value is used by Esquio Partitioner.*

```
{
    "Name": "DarkMode",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.GradualRolloutClaimValueToggle",
            "Parameters":
            {
                "Percentage": 50,
                "ClaimType": "role"
            }
        }
    ]
}
```

14.4 Partial rollout by Http Header value

This toggle enables its feature when the request header exists and its value falls within percentage created by Esquio Partitioner. Stickiness is based on the HTTP header value. Esquio uses Jenkins hash function that guarantees to the user get the same experience across many devices and also assures that a user which is among the first 30% will also be among the first 50% of the users.

Type

- Esquio.AspNetCore.Toggles.GradualRolloutHeaderValueToggle

Parameters

- Percentage: *The percentage of users that activates this toggle. Percentage from 0 to 100.*
- HeaderName: *The header name used whom value is used by Esquio Partitioner.*

```
{
    "Name": "DarkMode",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.Toggles.GradualRolloutHeaderValueToggle",
            "Parameters":
            {
                "Percentage": 50,
                "HeaderName": "X-Tenant"
            }
        }
    ]
}
```

14.5 Partial rollout by Http Session Id

This toggle enables its feature if the session identifier falls within percentage created by Esquio Partitioner. Stickiness is based on the ASP.NET Core SessionId value.

Type

- Esquio.AspNetCore.Toggles.GradualRolloutSessionToggle

Parameters

- Percentage: *The percentage of sessions that activates this toggle. Percentage from 0 to 100.*

```
{
    "Name": "DarkMode",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.GradualRolloutSessionToggle",
            "Parameters":
            {
                "Percentage": 50
            }
        }
    ]
}
```

14.6 Partial rollout by UserName

This toggle allows the current user name falls within percentage created by Esquio Partitioner. Stickiness is based on the user name. Esquio uses Jenkins hash function which guarantees the user gets the same experience across many devices and also ensures that a user who is among the first 30% will also be among the first 50% of users.

Type

- Esquio.AspNetCore.Toggles.GradualRolloutUserNameToggle

Parameters

- Percentage: *The percentage of users that activates this toggle. Percentage from 0 to 100.*

```
{
    "Name": "DarkMode",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.GradualRolloutUserNameToggle",
            "Parameters":
            {
                "Percentage": 50
            }
        }
    ]
}
```

14.7 Http Header value

This toggle enables its feature if the request header exists and its value its in the list.

Type

- Esquio.AspNetCore.Toggles.HeaderValueToggle

Parameters

- HeaderName: *The header name.*
- HeaderValues: *The header values to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.HeaderValueToggle",
            "Parameters":
            {
                "HeaderName": "Accept-Language",
                "HeaderValues": "en-US;es-ES"
            }
        }
    ]
}
```

14.8 Environment

This toggle enables its feature if the host execution environment and its value is in the list.

Type

- Esquio.AspNetCore.Toggles.HostEnvironmentToggle

Parameters

- Environments: *The ASP.NET Core host environments to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.HostEnvironmentToggle",
            "Parameters":
            {
                "Environments": "Staging;Production"
            }
        }
    ]
}
```

14.9 Host name

This toggle enables its feature if the hostname of the client instance is in the list.

Type

- Esquio.AspNetCore.Toggles.HostNameToggle

Parameters

- HostNames: *The request connection hostnames values to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.HostNameToggle",
            "Parameters":
            {
                "HostNames": "mycompany.org;en.domain.com"
            }
        }
    ]
}
```

14.10 Country

This toggle enables its feature if the request country is in the list (Ip geolocation through https://ip2c.org service).

Type

- Esquio.AspNetCore.Toggles.Ip2CountryToggle

Parameters

- Countries: *The request country values (two letters, ISO 3166) to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.Ip2CountryToggle",
            "Parameters":
            {
                "Countries": "ES;IT"
            }
        }
    ]
}
```

14.11 Identity Role

This toggle enables its feature if the identity role is in the list.

Type

- Esquio.AspNetCore.Toggles.RoleNameToggle

Parameters

- Roles: *The identity role values to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.RoleNameToggle",
            "Parameters":
            {
                "Roles": "betauser;beta"
            }
        }
    ]
}
```

14.12 Server IP

This toggle enables its feature if the host IP address is in the list.

Type

- Esquio.AspNetCore.Toggles.ServerIpAddressToggle

Parameters

- IpAddresses: *The host IP adddresses to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.ServerIpAddressToggle",
            "Parameters":
            {
                "IpAddresses": "11.22.44.88;11.22.33.44"
            }
        }
    ]
}
```

14.13 User Agent

This toggle enables its feature if the request user agent browser is in the list.

Type

- Esquio.AspNetCore.Toggles.UserAgentToggle

Parameters

- Browsers: *The user agents to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.UserAgentToggle",
            "Parameters":
            {
                "Browsers": "Mozilla/5.0;Chrome/81.0.4"
            }
        }
    ]
}
```

14.14 Identity name

This toggle enables its feature if the identity name is in the list.

Type

- Esquio.AspNetCore.Toggles.UserNameToggle

Parameters

- Users: *The identity names to activate this toggle separated by ';' character.*

```
{
    "Name": "MinutesProgressBar",
    "Enabled": true,
    "Toggles": [
        {
            "Type": "Esquio.AspNetCore.Toggles.UserNameToggle",
            "Parameters":
            {
                "Users": "betauser;beta"
            }
        }
    ]
}
```

CHAPTER 15

Extensibility

Esquio provides lots of toggles out-of-the-box, but sometimes that is not enough. Here, extensibility becomes a key quality attribute. If Esquio does not solve your problem, it can provide you an extensible part that would enable you to adapt it to your needs.

All main components have interfaces which are extensible. Using the inversion of control design, they can be substituted on **Microsoft.Extensions.DependencyInjection** configuration.

15.1 Creating your custom toggle

Supose that you need to create a toggle to enable features based on the user's browser (User-Agent header).

First, create a new toggle and implement the interface `IToggle`:

```
public class UserAgentBrowserToggle : IToggle
{
    public async Task<bool> IsActiveAsync(string featureName, string productName =␣
↪null, CancellationToken cancellationToken = default)
    {

    }
}
```

`IsActiveAsync` returns a boolean if the feature X for the product Y is enabled or not based on this toggle. In our case, depending on the user's browser.

To be able to specify for which browsers will be this feature enabled, you need to add a property:

```
public class UserAgentBrowserToggle : IToggle
{
    public string Browsers { get; set; }

    //code omited for brevity
}
```

If you want an application (Esquio UI) to be able to understand these fields (and for example built an user interface to provide users a more friendly way to configure the toggles of a feature), you need to decorate the toggle with some attributes:

```
[DesignType(Description = "Toggle that is active depending on request browser␣
↪information.")]
[DesignTypeParameter(ParameterName = Browsers, ParameterType = "System.String",␣
↪ParameterDescription = "Collection of browser names delimited by ';' character.")]
public class UserAgentBrowserToggle : IToggle
{
    public string Browsers { get; set; }

    //code omited for brevity
}
```

Attributes

- **DesignType**: *Allow to add a friendly description for the toggle.*
- **DesignTypeParameter**: *Allow to add a friendly description for the toggle's parameters.*

Once we have defined our attributes, it's time to use the services that our toggle needs to determinate if it is active or not. We can specify these services in the constructor

```
public UserAgentBrowserToggle(
    IRuntimeFeatureStore featureStore,
    IHttpContextAccessor httpContextAccessor,
    ILogger<UserAgentBrowserToggle> logger)
{
    _featureStore = featureStore ?? throw new␣
↪ArgumentNullException(nameof(featureStore));
    _contextAccessor = httpContextAccessor ?? throw new␣
↪ArgumentNullException(nameof(httpContextAccessor));
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

Services

- **IRuntimeFeatureStore**: We use this service to retrieve the feature. Once we have the feature, we can retrieve the toggle and its data (The parameters and their values).
- **IHttpContextAccessor**: To access the HttpContext.
- **ILogger<T>**: To log whatever you want.

It's time to finish our feature. We need to complete the `IsActiveAsync` method with the code below:

```
public async Task<bool> IsActiveAsync(
    string featureName,
    string productName = null,
    CancellationToken cancellationToken = default)
{
    var feature = await _featureStore.FindFeatureAsync(featureName, productName,␣
↪cancellationToken);
    var toggle = feature.GetToggle(typeof(UserAgentBrowserToggle).FullName);
    var data = toggle.GetData();

    var allowedBrowsers = data.Browsers.ToString();
    var currentBrowser = GetCurrentBrowser();
```

(continues on next page)

(continued from previous page)

```
    if (allowedBrowsers != null && !String.IsNullOrEmpty(currentBrowser))
    {
        _logger.LogDebug("{nameof(UserAgentBrowserToggle)} is trying to verify if
↪{currentBrowser} is satisfying allowed browser configuration.");

        var tokenizer = new StringTokenizer(allowedBrowsers, split_characters);

        foreach (var segment in tokenizer)
        {
            if (segment.Value?.IndexOf(currentBrowser, StringComparison.
↪InvariantCultureIgnoreCase) >= 0)
            {
                _logger.LogInformation("The browser {currentBrowser} is satisfied␣
↪using {allowedBrowsers} configuration.");

                return true;
            }
        }
    }

    _logger.LogInformation("The browser {currentBrowser} is not allowed using current␣
↪toggle configuration.");

    return false;
}

private string GetCurrentBrowser()
{
    return _contextAccessor.HttpContext
        .Request
        .Headers[UserAgent]
        .FirstOrDefault() ?? string.Empty;
}
```

Finally, we can register our custom toggle using the method `RegisterTogglesFromAssemblyContaining` in our Startup class:

```
services.AddEsquio(setup => setup.RegisterTogglesFromAssemblyContaining<Startup>())
```

As you can see, Esquio provides a flexible way to customize as you need.

You can see this full sample and much more in this repository and of course, any PR is welcomed ;)

CHAPTER 16

Configuration Store

It is the simplest way to store your Esquio configuration. However, it is not recommended for production, only for small projects or for testing purposes.

16.1 Installation

Install `Esquio.AspNetCore` package, typing the following command using the .NET Core CLI:

```
dotnet add package Esquio.Configuration.Store
```

or using Powershell or Package Manager:

```
Install-Package Esquio.Configuration.Store
```

or install via NuGet.

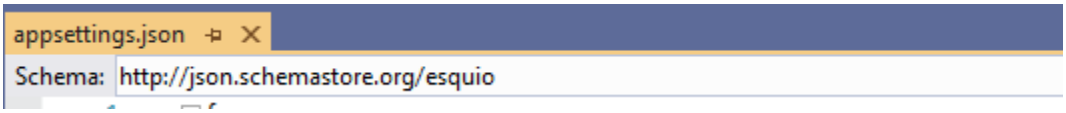
And register the specific service for this store:

```
public class Startup
{
    IConfiguration _configuration;

    public Startup(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        services
            .AddEsquio()
            .AddConfigurationStore(_configuration);
    }
```

`AddConfigurationStore` method registers the configuration store to use, in this case, based on the default configuration system of ASP.NET Core

16.2 Setting your values

So, let's open our `appsettings.json` file. To help us in this task, we can use the Esquio schema, selecting it on the `Schema` options:



Add the content below to your `appsettings.json` file:

```
{
  "Esquio": {
    "Products": [
      {
        "Name": "default",
        "Features": [
          {
            "Name": "HiddenGem",
            "Enabled": true,
            "Toggles": []
          }
        ]
      }
    ]
  }
}
```

By default, `Esquio` will be the root element. However, you could change it on adding the configurationStore:

```
.AddConfigurationStore(_configuration, key: "MyNewCustomRoot");
```

CHAPTER 17

Http Store

Esquio Http store connects against a Http Api that has been previously deployed on a Esquio UI.

> In samples/GettingStarted.AspNetCore.Mvc.HttpStore you'll find a full example of this store.

17.1 Installation

Install `Esquio.Http.Store` package, typing the following command using the .NET Core CLI:

```
dotnet add package Esquio.Http.Store
```

or using Powershell or Package Manager:

```
Install-Package Esquio.Http.Store
```

or install via NuGet.

In the `ConfigureServices` method of the `Startup` class, register the specific service for this store:

```
AddEsquio()
.AddHttpStore(options =>
{
    options
        .UseBaseAddress("http://localhost:1368/") //this is Esquio UI base address
        .UseApiKey("b6+KYpSY8VPMBmHLNJ00z80aPOe+Li4EGe4idoKKI1A="); // this is a Api␣
↪Key on Esquio UI (only Reader permission is Required)
});
```

`AddHttpStore` method registers the http store to use. in this case, based on the default configuration system of ASP.NET Core

And that's all. Log in Esquio UI and configure your toggles as you need.

Optionally, you can enable *cache* on http store configuration, this reduce the number of request to *Esquio UI* and improve the client performance. By default *cache* is disabled but this can be enabled and configured with UseCache extension method:

```
AddEsquio()
.AddHttpStore(options =>
{
    options
        .UseBaseAddress("http://localhost:1368/") //this is Esquio UI base address
        .UseApiKey("b6+KYpSY8VPMBmHLNJ00z80aPOe+Li4EGe4idoKKI1A=") // this is a Api␣
↪Key on Esquio UI (only Reader permission is Required)
        .UseCache(true, slidingExpiration: TimeSpan.FromSeconds(10));
});
```

CHAPTER 18

UI Deployment

Esquio UI is a regular ASP.NET Core project as Blazor Wasm, you can get the code and deploy as you want, by default you need to configure the Database to be used (Sql Server, Postress or MySql) and the OpenId provider.

To simplify a local deployment we provide different *Docker compose* files located on **build** directory:

```
docker-compose -f ./build/docker-compose-demo-with-ui-sqlserver up --build
docker-compose -f ./build/docker-compose-demo-with-ui-npgsql up --build
docker-compose -f ./build/docker-compose-demo-with-ui-mysql up --build
```

By default, a OpenId provider with `https://demo.identityserver.io` is used but you can change to use any other standard OpenId (Auth0 etc) or Azure AD.

18.1 Docker

On `https://hub.docker.com/r/xabarilcoding/esquioui` you can find also Esquio UI docker images.

To run locally, pull it typing:

```
docker pull xabarilcoding/esquioui:4.0.0
```

To run this image, you have to set several environment variables, indicating some database and OpenId server properties:

```
docker run xabarilcoding/esquioui:4.0.0 -p 9000:80 \
 -e Data__ConnectionString=<your-connection-string> \
 -e Data__Store=[SqlServer|MySql|NpgSql]
 -e Security__DefaultUsers__0__ApplicationRole=[Contributor|Reader|Management]
 -e Security__DefaultUsers__0__SubjectId=<your-subject-id>
 -e Security__OpenId__ClientId=<openid-clientid> \
 -e Security__OpenId__Audience=<openid-audience> \
 -e Security__OpenId__Scope=<openid-scope> \
 -e Security__OpenId__Authority=<openid-authority> \
```

(continues on next page)

(continued from previous page)

```
 -e Security__OpenId__ResponseType=<openid-response-type> \
 -e Security__OpenId__IsAzure=<IsAzure>
```

- Security__OpenId__ClientId: your client id `interactive.public on https://demo.identityserver.io`
- Security__OpenId__Audience: your security audience, this value is used on AddJwtBeaer to validate Beaer Tokens `api on https://demo.identityserver.io`
- Security__OpenId__Scope: your security audience `api on https://demo.identityserver.io`
- Security__OpenId__Authority: your authority `https://demo.identityserver.io`
- Security__OpenId__ResponseType: your openid flow response type to use `code`
- Security__OpenId__IsAzure: If you are using azure Ad as your provider this needs to be `true`

To configure Azure AD please follow existing docs on this like this. An example of AAD configuration is:

- Security__DefaultSubjectClaimType": `"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress"`,
- Security__OpenId__ClientId: `"762ed2b1-8107-44d5-9c9b-72c75749fb35"`
- Security__OpenId__Audience: `"c1cc1b6a-7580-49e5-8ab8-0f3cc9f97127"` *remove the url prefix that AAD use api://*
- Security__OpenId__Scope: `c1cc1b6a-7580-49e5-8ab8-0f3cc9f97127/ui`
- Security__OpenId__Authority: your authority `https://login.microsoftonline.com/bf3ae910-a895-44e6-aed1-0ed9601d9035`
- Security__OpenId__ResponseType: `code`
- Security__OpenId__IsAzure: `true`

18.2 Kubernetes

Alternatively, you can deploy it on your kubernetes cluster in a simple way.

Save your connection string into a file:

```
echo '<your-connection-string>' > ./connection-string.txt
```

And create a secret using this file:

```
kubectl create secret generic esquio-ui-secret --from-file=connection-string=./
↪connection-string.txt
```

Save the following yaml in a file named *esquio-ui.yaml*:

```
apiVersion: v1
kind: Service
metadata:
  name: esquio-ui-release
  labels:
    app.kubernetes.io/name: esquio-ui
    app.kubernetes.io/instance: esquio-ui-release
spec:
```

(continues on next page)

(continued from previous page)

```
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: esquio-ui
    app.kubernetes.io/instance: esquio-ui-release
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: esquio-ui-release
  labels:
    app.kubernetes.io/name: esquio-ui
    app.kubernetes.io/instance: esquio-ui-release
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: esquio-ui
      app.kubernetes.io/instance: esquio-ui-release
  template:
    metadata:
      labels:
        app.kubernetes.io/name: esquio-ui
        app.kubernetes.io/instance: esquio-ui-release
    spec:
      containers:
        - name: esquio-ui
          image: "xabarilcoding/esquioui:3.0.0"
          imagePullPolicy: IfNotPresent
          env:
            - name: ASPNETCORE_ENVIRONMENT
              value: "Development"
            - name: DATA__CONNECTIONSTRING
              valueFrom:
                secretKeyRef:
                  name: esquio-ui-secret
                  key: connection-string
            - name: DATA__STORE
              value: "[SqlServer|NpgSql|MySql]"
            - name: SECURITY__DEFAULTUSERS__0__APPLICATIONROLE
              value: "[Contributor|Reader]"
            - name: SECURITY__DEFAULTUSERS__0__SUBJECTID
              value: "<your-subject-id>"
            - name: DATA__STORE
              value: "[SqlServer|NpgSql|MySql]"
            - name: SECURITY__OPENID__CLIENTID
              value: "<your-openid-clientid>"
            - name: SECURITY__OPENID__AUDIENCE
              value: "<openid-audience>"
            - name: SECURITY__OPENID__SCOPE
              value: "<openid-scope>"
            - name: SECURITY__OPENID__AUTHORITY
              value: "<openid-authority>"
```

(continues on next page)

(continued from previous page)

```
            - name: SECURITY__OPENID__RESPONSETYPE
              value: "<openid-response-type>"
            - name: Security__OpenId__IsAzure
              value: "<true/false>"
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
```

And apply it with the command:

```
kubectl apply -f esquio-ui.yaml
```

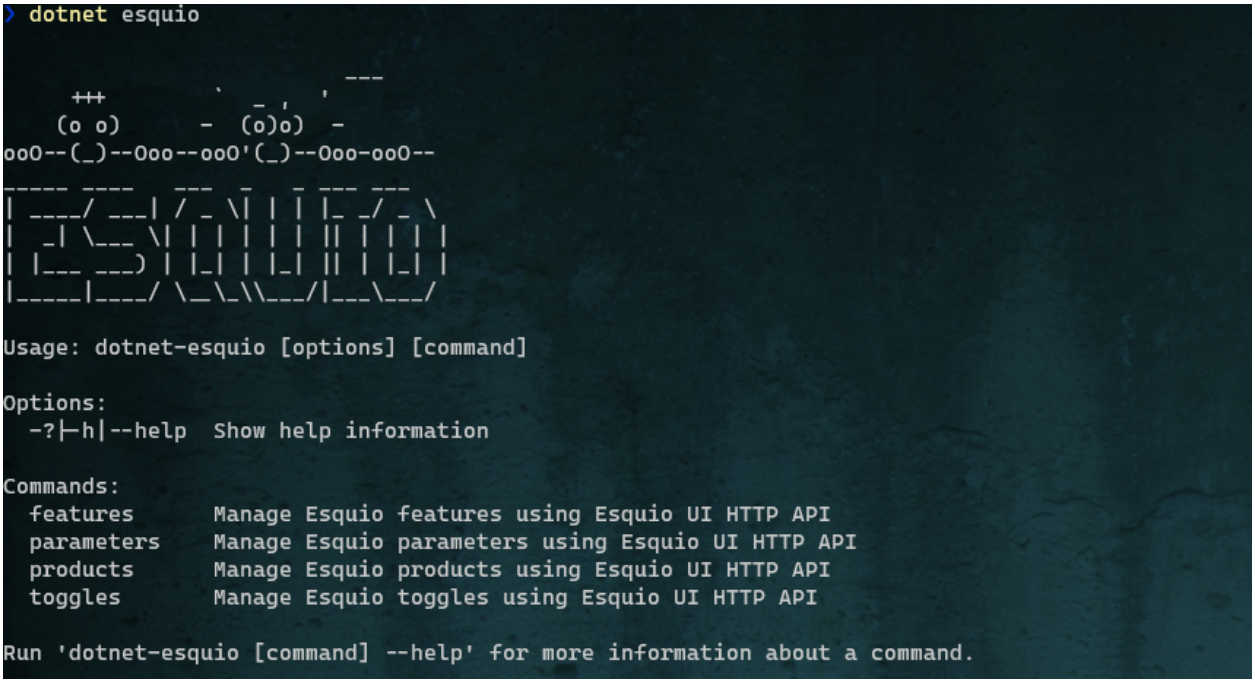
CHAPTER 19

Esquio CLI

Esquio.CLI is a simple crossplattform *dotnet tool* that allows you to interact with *Esquio UI* from the command line. This tool can be used from build or release pipelines on Azure DevOps, Github Actions etc.

To install **Esquio.CLI** open a console window and type the following command using the *.NET Core CLI*:

```
dotnet tool install -g Esquio.Cli
```




```
Usage: dotnet-esquio [options] [command]

Options:
  -?|-h|--help  Show help information
```

Commands:

features *Manage Esquio features using Esquio UI HTTP API*

parameters *Manage Esquio parameters using Esquio UI HTTP API*

products *Manage Esquio products using Esquio UI HTTP API*

toggles *Manage Esquio toggles using Esquio UI HTTP API*

Run 'dotnet-esquio [command] –help' for more information about a command.

All the commands allow to specify the *uri* and the *api key* to connect with Esquio UI, if don't want to specify this parameters in all calls, *Esquio CLI* allow read this values form the ESQUIO_URI and ESQUI_API_KEY environment variables.:

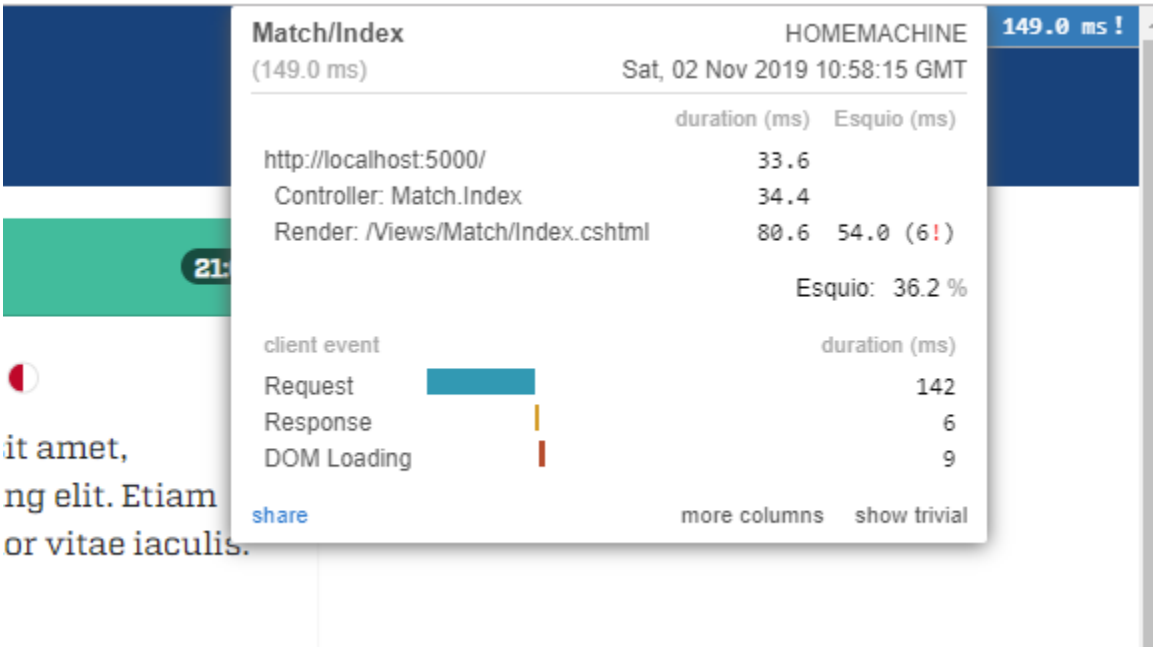
```
$env:ESQUIO_URI="https://[esquio_uri]"
$env:ESQUIO_API_KEY="[my-api-key]"
```

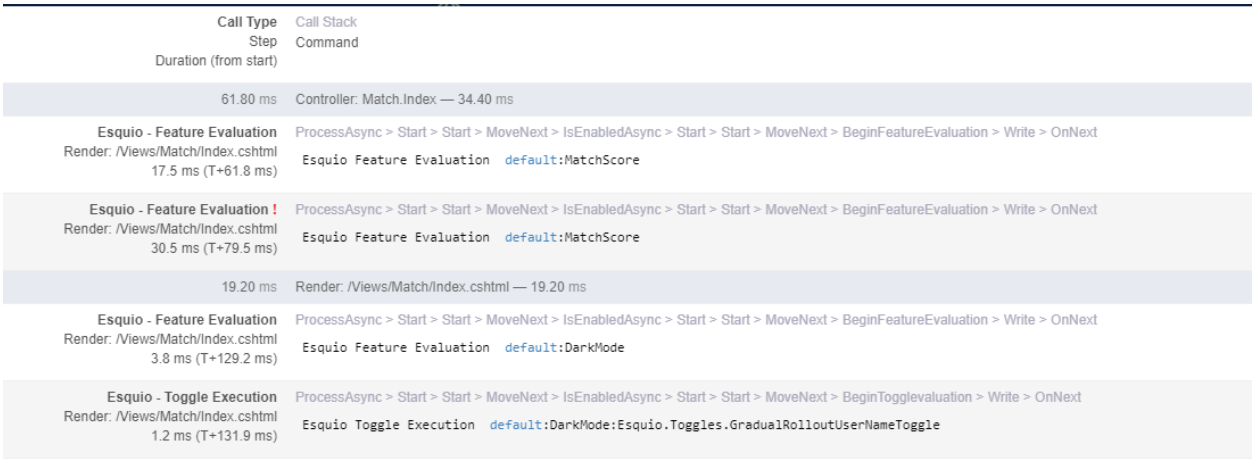
CHAPTER 20

Esquio & MiniProfiler

MiniProfiler is a *simple but effective mini-profioder for .NET*. It provides a *Step instrumentation* that you can add to the code you want to explicitly profile. Out of the box, *MiniProfiler* contains plugins for profiling ASP.NET Core, EF Core. But is extensible!.

We use this extensiblity to perform profiling over *Esquio* feature evaluations and show the elapsed evaluation time for each feature, each toggle and also show common errors for duplicate feature evaluation or exceptions inside toggle executions.





20.1 Setup

To install **MiniProfiler.Esquio** open a console window and type the following command using the *.NET Core CLI*:

```
dotnet package add MiniProfiler.Esquio
```

or using *Powershell* or *Package Manager*:

```
Install-Package Esquio.AspNetCore
```

To configure MiniProfiler.Esquio on ASP.NET Core, add MiniProfiler service configuration and new *Esquio* plugin with **AddEsquio**:

```
services
    .AddMiniProfiler(options =>
    {
        options.RouteBasePath = "/profiler";
        options.EnableServerTimingHeader = true;

        options.ResultsAuthorize = (_) => true;
        options.ShouldProfile = _ => true;

        options.IgnoredPaths.Add("/lib");
        options.IgnoredPaths.Add("/css");
        options.IgnoredPaths.Add("/js");
        options.IgnoredPaths.Add("/assets");

    }).AddEsquio();
```