Voxel.MiddyNet

Release 1.0.2

May 01, 2021

About MiddyNet

MiddyNet is a lightweight middleware library for .NET Core and AWS Lambda that let you focus on what's really important for you: your business logic.

MiddyNet is a port to .NET Core of the fantastic Middy library for Javascript.

MiddyNet will *force* you to organise your lambda's code in a certain way, but we think it's a reasonable and practical way to do it.

MiddyNet also includes a custom logger, that will help you logging structured logs and that let you enrich your logs with custom properties.

MiddyNet also includes a way to read and propagate the TraceContext so you can have distributed tracing without any effort.

CHAPTER 1

How it works

MiddyNet is a lightweight middleware engine. A middleware is a piece of code that can be run before or after your function is run. In the *before* section, middlewares are run in the same order that are added. In the *after* section, middlewares are run in the inverse order that are added. Usually, a middleware will only make sense to do anything in one of the sections.

Exceptions can be thrown by the middlewares and they are captured by the library. If the exception is thrown in the *before* section, the exception is made available to the following middlewares and eventually to your function via the context. If the exception is thrown in the *handler* or in the *after* section, it's eventually thrown by the library as an *AggregateException*. Non cleaned `MiddlewareBeforeExceptions` will also be included in the resulting AggregateException.

MiddyNet also adds a Logger to log structured logging and a way to capture and propagate the TraceContext as described here.

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.
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. Update the docs if necessary.
5. You may merge the Pull Request in once you have the sign-off of one other developer, or if you do not have permission to do that, you may request the 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, version 1.4.

CHAPTER 5

Getting started

5.1 Installation

MiddyNet is splited in different NuGet packages so that you only import what you really use. To start using it, add the main package to your AWS Lambda project:

```
dotnet package add Voxel.MiddyNet
```

5.2 Conventions

The best way to organise your code when working with **MiddyNet** is to have a single file for each lambda function. Once you have that, you need to derive from `MiddyNet<TReq,TRes>`, where TReq is the type of the input event (SNS, SQS, etc) and TRes is the type of the result. If your function doesn't have to return anything, we recommend you to use an int and return 0 from your function. At this moment we don't offer a class with no return type.

MiddyNet does its work in a function called Handler. This is the function you will need to specify when configuring your lambda as your entry point. So, at this point, all your lambdas will have its own source file and all of them will expose the same method called *Handler*. Nice and easy.

MiddyNet will make you implement a function called *Handle* where you will need to put your code.

So, a minimum skeleton of your lambda function would be something like this:

```
public class MySQSLambdaFunction : MiddyNet<SQSEvent, int>
{
    public MySQSLambdaFunction()
    {
        // Your own initializations
        // MiddyNet middleware definitions. More on that later.
    }

    protected override Task<int> Handle(SQSEvent lambdaEvent, MiddyNetContext context)
```

(continues on next page)

(continued from previous page)

```
    {
        // Your business logic

        return Task.FromResult(0);
    }
}
```

CHAPTER 6

Using a Middleware

6.1 The basics

To use a middleware in your lambda function, you need to add it using the function use in the constructor. For example, to add the *SQSTracingMiddleware* you need to follow the following steps:

1. Add the corresponding package:

```
dotnet package add Voxel.MiddyNet.Tracing.SQSMiddleware
```

2. Add the corresponding using in your lambda function:

```
using Voxel.MiddyNet.Tracing.SQSTracingMiddleware;
```

3. Add it to the engine in your constructor:

```
public MySQSLambdaFunction()
{
    // Your own initializations
    Use(new SQSTracingMiddleware());
}
```

And that's all. **MiddyNet** will execute this middleware before and after your function runs. In this particular case, the middleware only will do things in the *before* section.

6.2 Middleware configurations

Some middlewares can have configurations you might want to take advantage of. The configuration will usually be a parameter of the constructor, like in the SSM middleware:

```
public class MySQSLambdaFunction: MiddyNet<SQSEvent,int>
{
    private const string ClientIdName = "clientIdName";
    private const string ClientSecretName = "clientSecretName";

    public MySQSLambdaFunction()
    {
        var clientIdPath = Environment.GetEnvironmentVariable("clientIdPath");
        var clientSecretPath = Environment.GetEnvironmentVariable("clientSecretPath");

        Use(new SSMMiddleware<SNSEvent, int>(new SSMOptions
        {
            CacheExpiryInMillis = 60000,
            ParametersToGet = new List<SSMParameterToGet>
            {
                new SSMParameterToGet(ClientIdName, clientIdPath),
                new SSMParameterToGet(ClientSecretName, clientSecretPath)
            }
        }));

        Use(new SQSTracingMiddleware<SNSEvent, int>());
    }

    protected override async Task<int> Handle(SNSEvent lambdaEvent, MiddyNetContext␣
↪context)
    {
        // Your business logic

        return await Task.FromResult(0);
    }
}
```

CHAPTER 7

Middy Context

MiddyNet offers you a context with additional information you can use in your lambda function. This context has the following properties.

7.1 LambdaContext

An object implementing `ILambdaContext` with additional information provided by AWS. You can read about it here.

7.2 AdditionalContext

A `Dictionary<string, object>` filled with the information that the different middlewares want to add to it. For example, the *SSMMiddleware* will add there the secrets gotten from SSM.

7.3 MiddlewareExceptions

There are 4 properties and a method related to exceptions:

- `List<Exception> MiddlewareBeforeExceptions { get; }`, *Middleare with the exceptions thrown by the middlewares' *Before* method.*
- `Exception HandlerException { get; set; }`, with the exception thrown by the custom handler's *Handle* method.
- `List<Exception> MiddlewareAfterExceptions { get; }`, with the exceptions thrown by the middlewares' *After* method.
- `bool HasExceptions { get; }`, which tells if there are any exceptions in the aforementioned properties
- `List<Exception> GetAllExceptions()`, which returns the exceptions in the three properties as a single list, in the same order as they happened.

You may want to check whether any exceptions occured before proceeding with your custom logic in the handler like so:

```
        if (context.HasExceptions)
{  // do stuff...  context.MiddlewareBeforeExceptions.Clear();
}
```

This will prevent those exceptions to be returned at the end of the pipeline processing.

7.4 Logger

An object of type MiddyLogger to allow you to log structured logs. It uses the `ILambdaLogger` provided by AWS internally. See more information about the /logger.

CHAPTER 8

Middy Logger

As we explained in the Middy Context section, **MiddyNet** gives you a logger to be able to log structured logs easily. You will still have access to the original logger provided by AWS, by accessing the *Logger* property of the `ILambdaContext` object.

8.1 Basic logging

If you just need to log a message, you can just do:

```
middyContext.Logger.Log(LogLevel.Debug, "hello world");
```

This will generate a log like this:

```
{
"Message": "hello world",
"Level": "Debug"
}
```

8.2 Logging extra properties

If you need to add extra properties to your main message, you can add them this way:

```
middyContext.Logger.Log(LogLevel.Info, "hello world", new LogProperty("key1", "value1
↪"), , new LogProperty("key2", "value2"));
```

This will generate a log like this:

```
{
"Message": "hello world",
"Level": "Info",
```

(continues on next page)

(continued from previous page)

```
"key1": "value1",
"key2": "value2"
}
```

8.3 Enriching with properties

If you don't want to add the same properties again and again, you can enrich the logger with them like this:

```
middyContext.Logger.EnrichWith(new LogProperty("key", "value"));
```

Once done that, every time that you log a single message:

```
middyContext.Logger.Log(LogLevel.Debug, "hello world");
```

The property will be added to the final log message:

```
{
"Message": "hello world",
"Level": "Info",
"key": "value"
}
```

The same will happen if you log a message with one or more properties. The enriching properties will be added to those.

8.4 Logging a complex property

The property you add can be a complex object, and it will be serialised as JSON. So, for example, if you log this:

```
var classToLog = new ClassToLog
{
    Property1 = "The value of property1",
    Property2 = "The value of property2"
};

middyContext.Logger.Log(LogLevel.Info, "hello world", new LogProperty("key",␣
↪classToLog));
```

you will have a log like this:

```
{
"Message": "hello world",
"Level": "Info",
"key"": {
    "Property1": "The value of property1",
    "Property2": "The value of property2"
    }
}
```

CHAPTER 9

Writing a Middleware

9.1 The basics

Writing a new middleware is quite easy. You just need to implement the `ILambdaMiddleware` interface and implement both methods. The minimum implementation of a middleware that doesn't do anything would be this one:

```
public class DummyMiddleware<TReq, TRes> : ILambdaMiddleware<TReq, TRes>
{
    public Task Before(TReq lambdaEvent, MiddyNetContext context)
    {
        return Task.CompletedTask;
    }

    public Task<TRes> After(TRes lambdaResponse, MiddyNetContext context)
    {
        return Task.FromResult(lambdaResponse);
    }
}
```

9.2 Storing data

You can add data to the context so that the next middlewares or the lambda function can access it. To do so, you just need to add a new entry into the `AdditionalContext` property:

```
public Task Before(TReq lambdaEvent, MiddyNetContext context)
{
    context.AdditionalContext.Add("key", "value");
    return Task.CompletedTask;
}
```

The value of the property can be whatever you want, from a single type to a complex object.

9.3 Exceptions

You don't need to catch exceptions in the middleware. The library will catch them for you and add them to the context's corresponding exceptions collection (`MiddlewareBeforeExceptions`, `HandlerException` or `MiddlewareAfterExceptions` depending on where was the exception thrown).

CHAPTER 10

Existing NuGet packages

There are a bunch of NuGet packages available for you to use.

10.1 Voxel.Middy.Tracing.Core

This is a quite independent package that you can use on its own. It has an implementation of the `TraceContext` as described here.

The actual implementation has the following logic: * If the `traceparent` header is not valid, it recreates the entire header and resets the `tracestate` one too. We define valid as:

- It has 4 parts separated by hyphens
- The version part is two characters long
- The trace-id part is 32 characters long
- The parent-id part is 16 characters long
- The flags part is 2 characters long
- If the `traceparent` header is valid, it modifies the parent-id part with a new value.
- If the `traceparent` header is valid, it propagate whatever there is in the `tracestate` header.

10.2 Voxel.MiddyNet.Tracing.SNS

This is another package that is quite independent. It contains extension methods to add the information of the `TraceContext` object from `Voxel.MiddyNet.Tracing.Core` to an SNS message via `MessageAttributes`.

You can use this package inside a Lambda function or from any place, to send the `TraceContext` information in a way that MiddyNet will be able to read using a middleware.

10.3 Voxel.MiddyNet.Tracing.Http

This is another package that is quite independent. It contains extension methods to add the information of the `TraceContext` object from `Voxel.MiddyNet.Tracing.Core` to an HttpRequestMessage via `Headers`.

You can use this package inside a Lambda function or from any place, to send the `TraceContext` information in a way that MiddyNet will be able to read using a middleware.

10.4 Voxel.MiddyNet

This is the main package. You need to add it to your project if you want to use MiddyNet.

10.5 Voxel.MiddyNet.Tracing.SNSMiddleware

This package contains a middleware that reads the `TraceContext` information from the `MessageAttributes` of an SNS event and enriches the `MiddyLogger` with them, so that your logs will have it and it will be easier to correlate them.

The logs will have a property for `traceparent`, another one for `tracestate`, and another one for `trace-id`.

10.5.1 Sample code

A typical use of the middelware will look like this:

```
public class MySample : MiddyNet<SNSEvent, int>
{
    public MySample()
    {
        Use(new SNSTracingMiddleware<int>());
    }

    protected override async Task<int> Handle(SNSEvent snsEvent, MiddyNetContext␣
↪context)
    {
        context.Logger.Log(LogLevel.Info, "hello world");

        // Do stuff

        return Task.FromResult(0);
    }
}
```

10.6 Voxel.MiddyNet.Tracing.SQSMiddleware

This package contains a middleware that reads the `TraceContext` information from the `MessageAttributes` of an SQS event and enriches the `MiddyLogger` with them, so that your logs will have it and it will be easier to correlate them.

The logs will have a property for `traceparent`, another one for `tracestate`, and another one for `trace-id`.

10.6.1 Sample code

A typical use of the middelware will look like this:

```
public class MySample : MiddyNet<SQSEvent, int>
{
    public MySample()
    {
        Use(new SQSTracingMiddleware<int>());
    }

    protected override async Task<int> Handle(SQSEvent sqsEvent, MiddyNetContext␣
↪context)
    {
        context.Logger.Log(LogLevel.Info, "hello world");

        // Do stuff

        return Task.FromResult(0);
    }
}
```

10.7 Voxel.MiddyNet.Tracing.ApiGatewayMiddleware

This package contains a middleware that reads the `TraceContext` information from the `traceparent` and `tracestate` headers of an `APIGatewayProxyRequest` or an `APIGatewayHttpApiV2ProxyRequest` and enriches the `MiddyLogger` with them, so that your logs will have it and it will be easier to correlate them.

The logs will have a property for `traceparent`, another one for `tracestate`, and another one for `trace-id`.

In addition, the MiddyNetContext is enriched with a new entry in the AdditionalContext collection that contains a TraceContext object.

This TraceContext object provides a MutateParentId method that can be used to obtain a traceparent with the same Version, TraceId, and TraceFlags but with a new ParentId that can be used to call other systems, as recommended by W3C.

10.7.1 Sample code

A typical use of the middleware for APIGateway will look like this:

```
public class MySample : MiddyNet<APIGatewayProxyRequest, APIGatewayProxyResponse>
{
    public MySample()
    {
        Use(new ApiGatewayTracingMiddleware<APIGatewayProxyResponse>());
    }

    protected override async Task<APIGatewayProxyResponse>␣
↪Handle(APIGatewayProxyRequest apiEvent, MiddyNetContext context)
    {
        //This log is enriched with the tracing information received in the headers␣
↪of the request
        context.Logger.Log(LogLevel.Info, "Function called.");
```

(continues on next page)

(continued from previous page)

```
        //If you need to call another system, you need to obtain a traceparent based␣
↪on the original traceparent
        //received but with the ParentId changed
        var currentTraceContext = (TraceContext)context.
↪AdditionalContext[ApiGatewayTracingMiddleware.TraceContextKey];
        var newTraceContext = TraceContext.MutateParentId(currentTraceContext);

        //Now you can use this newTraceContext in your calls
        var traceparentForCallingAnotherSystem = newTraceContext.TraceParent;
        var tracestateForCallingAnotherSystem = newTraceContext.TraceState;

        return Task.FromResult(new APIGatewayProxyResponse
        {
            StatusCode = 200,
            Body = "Ok"
        });
    }
}
```

and for APIGatewayHttpV2Api will look like this:

```
public class ApiGatewayHttpApiV2Tracing : MiddyNet<APIGatewayHttpApiV2ProxyRequest,␣
↪APIGatewayHttpApiV2ProxyResponse>
{
    public ApiGatewayHttpApiV2Tracing()
    {
        Use(new ApiGatewayHttpApiV2TracingMiddleware());
    }

    protected override Task<APIGatewayHttpApiV2ProxyResponse>␣
↪Handle(APIGatewayHttpApiV2ProxyRequest proxyRequest, MiddyNetContext context)
    {
        //This log is enriched with the tracing information received in the headers␣
↪of the request
        context.Logger.Log(LogLevel.Info, "Function called.");

        //If you need to call another system, you need to obtain a traceparent based␣
↪on the original traceparent
        //received but with the ParentId changed
        var currentTraceContext = (TraceContext)context.
↪AdditionalContext[ApiGatewayHttpApiV2TracingMiddleware.TraceContextKey];
        var newTraceContext = TraceContext.MutateParentId(currentTraceContext);

        //Now you can use this newTraceContext in your calls
        var traceparentForCallingAnotherSystem = newTraceContext.TraceParent;
        var tracestateForCallingAnotherSystem = newTraceContext.TraceState;

        return Task.FromResult(new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = 200,
            Body = "Ok"
        });
    }
}
```

10.8 Voxel.MiddyNet.SSM

This package contains a middleware that allows you to retrieve secrets from `Parameter Store`. It also allows you to cache them to minimise the calls to `Parameter Store`.

10.8.1 Configuration

You need to pass a `SSMOptions` object in the constructor with the following properties: * CacheExpiryInMillis: number of milliseconds that the middleware will cache the parameter. During this time, it won't go again to `ParameterStore` to read the parameter. * ParametersToGet: a list of `SSMParameterToGet`. Each `SSMParameterToGet` has two properties:

- Name: Name of the parameter in the lambda function. You will use this name later to access the value of the parameter inside your lambda function.
- Path: Path of the parameter in `ParameterStore`

The middleware will store the values of the parameters in the `AdditionalContext` of the `MiddyContext`. It will add a property there for each parameter. The key of the property will be the name of the parameter.

10.8.2 Sample code

A typical configuration and use of the middelware will look like this:

```
public class MySSMSample : MiddyNet<SNSEvent, int>
{
    private const string Param1Name = "Param1Name";
    private const string Param2Name = "Param2Name";

    public MySSMSample()
    {
        var param1Path = System.Environment.GetEnvironmentVariable("param1Path");
        var param2Path = System.Environment.GetEnvironmentVariable("param2Path");

        var options = new SSMOptions
        {
            ParametersToGet = new List<SSMParameterToGet>
            {
                new SSMParameterToGet(Param1Name, param1Path),
                new SSMParameterToGet(Param2Name, param2Path)
            }
        };

        Use(new SSMMiddleware<SNSEvent, int>(options));
    }


    protected override async Task<int> Handle(SNSEvent snsEvent, MiddyNetContext␣
↪context)
    {
        var param1Value = context.AdditionalContext[Param1Name].ToString();
        var param2Value = context.AdditionalContext[Param2Name].ToString();

        // Do stuff
```

(continues on next page)

(continued from previous page)

```
        return Task.FromResult(0);
    }
}
```

10.9 Voxel.MiddyNet.HttpCors

This package contains a middleware that allows you to set the CORS headers in the response. There are two versions avalaible:

- One for REST Api (APIGatewayProxyRequest and APIGatewayProxyResponse).
- And another for Http Api (APIGatewayHttpApiV2ProxyRequest and APIGatewayHttpApiV2ProxyResponse).

10.9.1 Configuration

You can pass a `CorsOptions` object in the constructor with the following properties (all of them optional):

- Origin: origin to put in the header (default: "*")
- Origins: an array of allowed origins. The incoming origin is matched against the list and is returned if present.
- Headers: value to put in `Access-Control-Allow-Headers` (default: null)
- Credentials: if true, sets the `Access-Control-Allow-Origin` as request header Origin, if present (default `false`)
- MaxAge: value to put in `Access-Control-Max-Age` header (default: null)
- CacheControl: value to put in `Cache-Control` header on pre-flight (OPTIONS) requests (default: null)

10.9.2 Sample code

A typical use of the middelware will look like this for Rest API:

```
public class MySample : MiddyNet<APIGatewayProxyRequest, APIGatewayProxyResponse>
{
    public MySample()
    {
        Use(new HttpCorsMiddleware(new CorsOptions{Origin = "http://example.com"}));
    }

    protected override async Task<APIGatewayProxyResponse>␣
↪Handle(APIGatewayProxyRequest apiEvent, MiddyNetContext context)
    {
        // Do stuff

        var result = new APIGatewayProxyResponse
        {
            StatusCode = 200,
            Body = "hello from test"
        };

        return Task.FromResult(result);
    }
}
```

And like this for Http API:

```
public class MySample : MiddyNet<APIGatewayHttpApiV2ProxyRequest,␣
↪APIGatewayHttpApiV2ProxyResponse>
{
    public MySample()
    {
        Use(new HttpV2CorsMiddleware(new CorsOptions{Origin = "http://example.com"}));
    }

    protected override async Task<APIGatewayHttpApiV2ProxyResponse>␣
↪Handle(APIGatewayHttpApiV2ProxyResponse apiEvent, MiddyNetContext context)
    {
        // Do stuff

        var result = new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = 200,
            Body = "hello from test"
        };

        return Task.FromResult(result);
    }
}
```

10.10 Voxel.MiddyNet.HttpJsonBodyParser

This package contains a middleware that parses a JSON to an object of an explicit type. There are two versions avalaible:

- One for REST APIs (APIGatewayProxyRequest and APIGatewayProxyResponse).
- One for HTTP APIs (APIGatewayHttpApiV2ProxyRequest and APIGatewayHttpApiV2ProxyResponse).

10.10.1 Configuration

When you use the middleware, You need to specify the type you want to convert the JSON object into. The middleware will put the object in the AdditionalContext[Constants.BodyContextKey] property. To access the object, you will need to perform an explicit cast of that property.

10.10.2 Sample code

A typical use of the middelware will look like this for Rest API:

```
public class MySample : MiddyNet<APIGatewayProxyRequest, APIGatewayProxyResponse>
{
    public MySample()
    {
        Use(new HttpJsonBodyParserMiddleware<Foo>());
    }

    protected override async Task<APIGatewayProxyResponse>␣
↪Handle(APIGatewayProxyRequest apiEvent, MiddyNetContext context)
```

(continues on next page)

(continued from previous page)

```
    {
        var foo = ((Foo)context.AdditionalContext[Constants.BodyContextKey]);

        // Do stuff with foo

        var result = new APIGatewayProxyResponse
        {
            StatusCode = 200,
            Body = "hello from test"
        };

        return Task.FromResult(result);
    }
}
```

And like this for Http API:

```
public class MySample : MiddyNet<APIGatewayHttpApiV2ProxyRequest,␣
↪APIGatewayHttpApiV2ProxyResponse>
{
    public MySample()
    {
        Use(new HttpV2JsonBodyParserMiddleware<Foo>());
    }

    protected override async Task<APIGatewayHttpApiV2ProxyResponse>␣
↪Handle(APIGatewayHttpApiV2ProxyResponse apiEvent, MiddyNetContext context)
    {
         var foo = ((Foo)context.AdditionalContext[Constants.BodyContextKey]);

        // Do stuff with typed foo

        var result = new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = 200,
            Body = "hello from test"
        };

        return Task.FromResult(result);
    }
}
```

10.11 Voxel.MiddyNet.ProblemDetailsMiddleware

The middleware contained in this package formats Api exceptions as ProblemDetails following [RFC7807](https://tools.ietf.org/html/rfc7807).

There are two versions avalaible:

- One for REST Api (APIGatewayProxyRequest and APIGatewayProxyResponse).
- And another for Http Api (APIGatewayHttpApiV2ProxyRequest and APIGatewayHttpApiV2ProxyResponse).

10.11.1 Configuration

It can receive a `ProblemDetailsMiddlewareOptions` to specify mappings from a particular exception type to an Http status code. E.g:

var options = new ProblemDetailsMiddlewareOptions(); options.Map<NotImplementedException>(501)));

When a `NotImplementedException` is thrown, ProblemDetailsMiddleware will return the exception message with a 501 Http status code.

10.11.2 Sample code

A typical usage of the ProblemDetailsMiddleware for REST Api whould look something like:

```
public class ApiGatewayProblemDetails : MiddyNet<APIGatewayProxyRequest, APIGatewayProxyResponse> {

    public ApiGatewayProblemDetails() {

        Use(new ProblemDetailsMiddleware.ProblemDetailsMiddleware(new ProblemDetailsMiddlewareOptions().Map<NotImplementedException>(501)));

    }

    protected override Task<APIGatewayProxyResponse> Handle(APIGatewayProxyRequest lambdaEvent, MiddyNetContext context) {

        throw new NotImplementedException("this will be used in the problem details description");

    }

}
```

And for Http Api:

```
public class ApiGatewayProblemDetails : MiddyNet<APIGatewayHttpApiV2ProxyRequest, APIGatewayHttpApiV2ProxyResponse> {

    public ApiGatewayProblemDetails() {

        Use(new ProblemDetailsMiddleware.ProblemDetailsMiddleware(new ProblemDetailsMiddlewareOptions().Map<NotImplementedException>(501)));

    }

    protected override Task<APIGatewayHttpApiV2ProxyResponse> Handle(APIGatewayHttpApiV2ProxyRequest lambdaEvent, MiddyNetContext context) {

        throw new NotImplementedException("this will be used in the problem details description");

    }

}
```