API For Getting Indicative Exchange Rates From TCMB

April 15, 2018 – ISTANBUL

Written By Murat YA$AR

http://www.yasarmurat.com/ | **https://www.linkedin.com/in/muratyasar/**

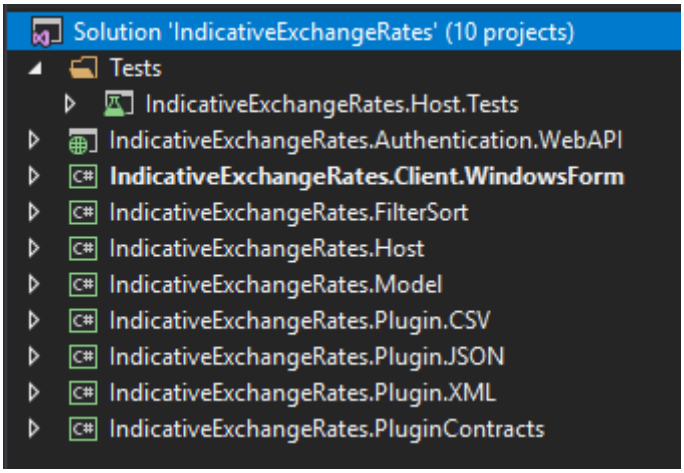
This guide has been prepared for **IndicativeExchangeRates** solution.

Throughout this guide, I will name the IndicativeExchangeRates solution as **API**.

Development Environment

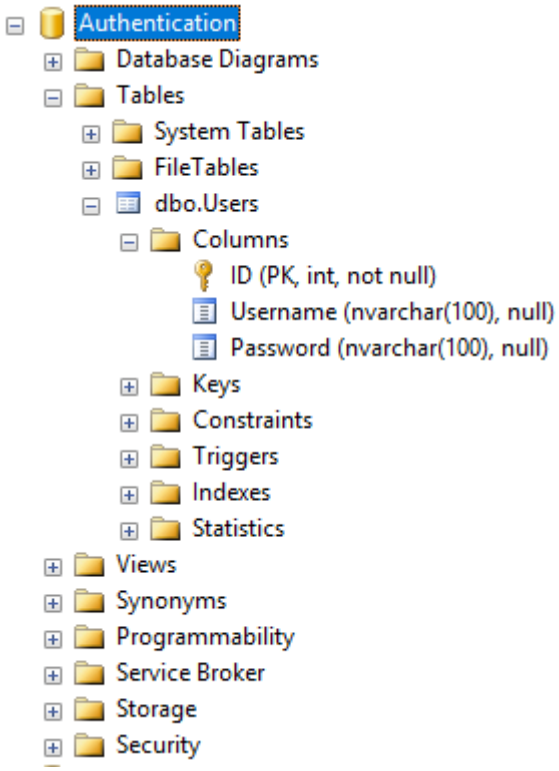
Operating System	: Windows 10 Pro Version 1709 OS Build 16299.371
System Type	: 64-bit operating system, x64-based processor
Visual Studio	: Microsoft Visual Studio 2017 Version 15.6.6
Target Framework	: .NET Framework 4.6
Solution Name	: IndicativeExchangeRates
Solution Configurations	: Debug
Solution Platforms	: Any CPU
Programming Language	: C#
Solution Content Project Types	: Class Library, Web API, Windows Forms

Solution Content Project Details



►Tests: This folder contains unit test projects which covers core part of my implementation.

►IndicativeExchangeRates.Authentication.WebAPI: This is a Web API project. Purpose of this project is authenticate the client who is going to use this API. I created a database called **Authentication** which is used for storing user information in a **User** table.



I manually inserted two sample rows in order to test this authentication mechanism.

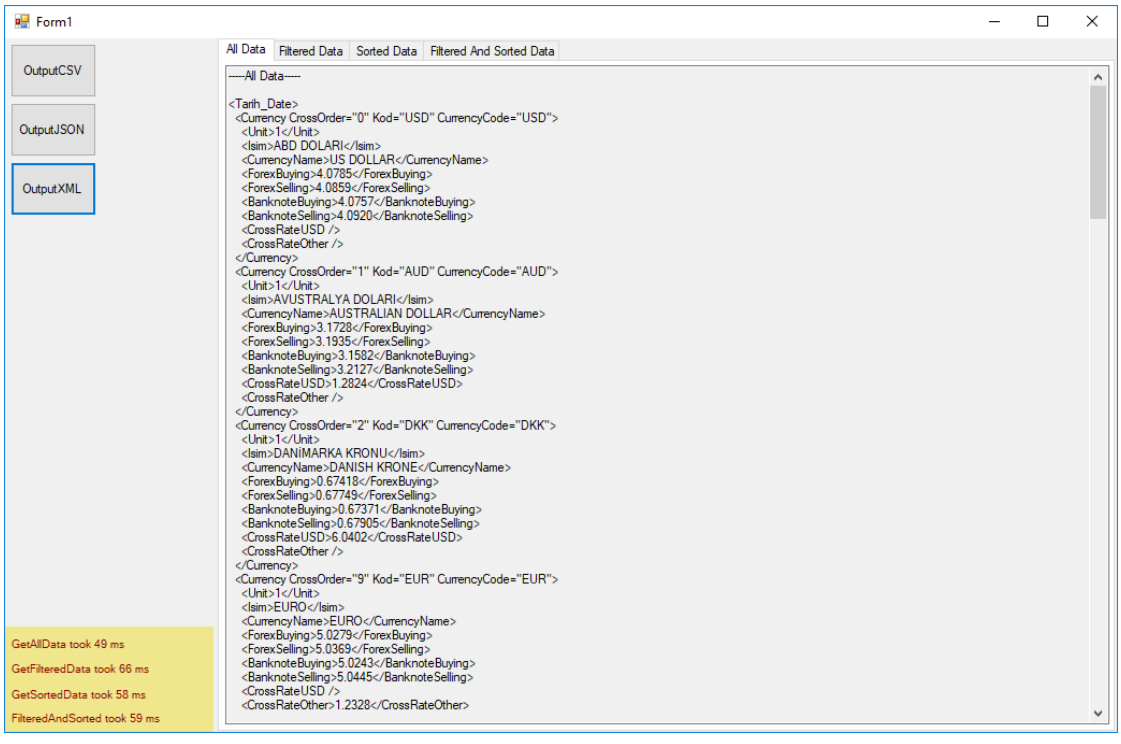
MYPC.Authentication - dbo.Users

	ID	Username	Password
▶	1	muratyasar	denemetry!4
	2	anotherperson	denemetry!5
*	NULL	NULL	NULL

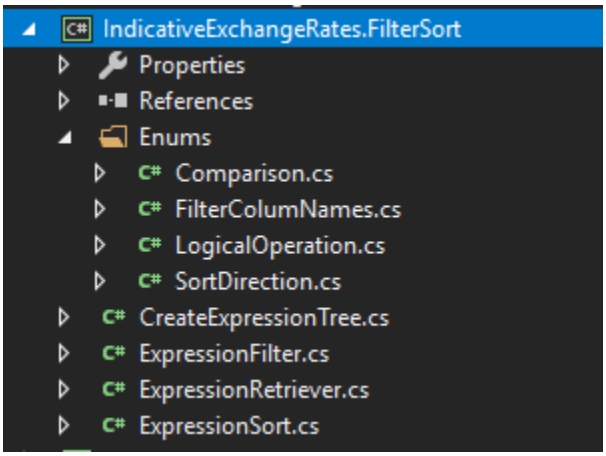
You can use following scripts to create table.

```
USE [Authentication]
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Users](
	[ID] [int] IDENTITY(1,1) NOT NULL,
	[Username] [nvarchar](100) NULL,
	[Password] [nvarchar](100) NULL,
PRIMARY KEY CLUSTERED
(
	[ID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]

GO
```

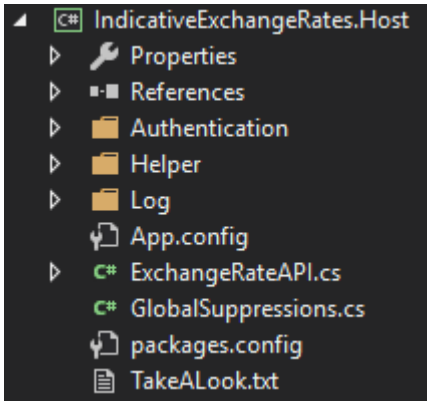
►IndicativeExchangeRates.Client.WindowsForm: This is a Windows Forms project. This project has been created to test for the whole API logic. It provides graphical user interface in order to make interaction with the API.



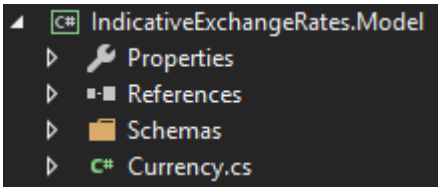
►IndicativeExchangeRates.FilterSort: This is a class library project which has been created for filtering and sorting mechanism the API uses.



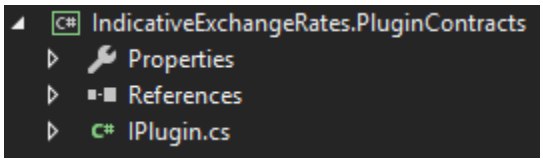
►IndicativeExchangeRates.Host: This is a class library project. It can be said that is the core of the API. It verifies a client, searches for the appropriate plugin/s to be loaded, creates detailed log files on disk and do other things that I will explain in detail later in this guide.



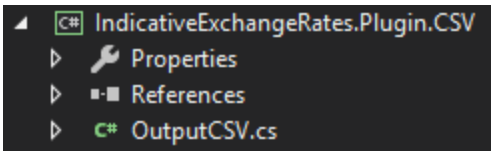
►IndicativeExchangeRates.Model: This is a class library project. It contains schema files and also the main model class which is being used throughout this API solution.



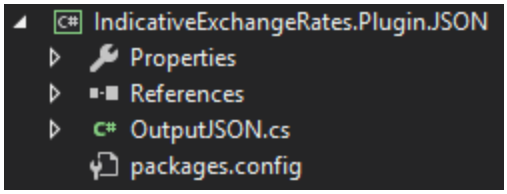
►IndicativeExchangeRates.PluginContracts: This is a class library project. It contains only an interface for the plugin mechanism that is used in this API solution.



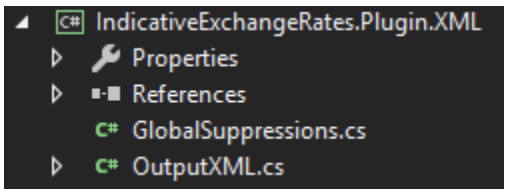
►IndicativeExchangeRates.Plugin.CSV: This is a class library project. It consists of only one class which implements the interface from **IndicativeExchangeRates.PluginContracts** project. By implementing this interface we are telling **IndicativeExchangeRates.Host** project to know this is a plugin and can be used in API. The output of this project, as the name suggests, is CSV formatted data.



►IndicativeExchangeRates.Plugin.JSON: This is a class library project. It consists of only one class which implements the interface from **IndicativeExchangeRates.PluginContracts** project. By implementing this interface we are telling **IndicativeExchangeRates.Host** project to know this is a plugin and can be used in API. The output of this project, as the name suggests, is JSON formatted data.



►IndicativeExchangeRates.Plugin.XML: This is a class library project. It consists of only one class which implements the interface from **IndicativeExchangeRates.PluginContracts** project. By implementing this interface we are telling **IndicativeExchangeRates.Host** project to know this is a plugin and can be used in API. The output of this project, as the name suggests, is XML formatted data.



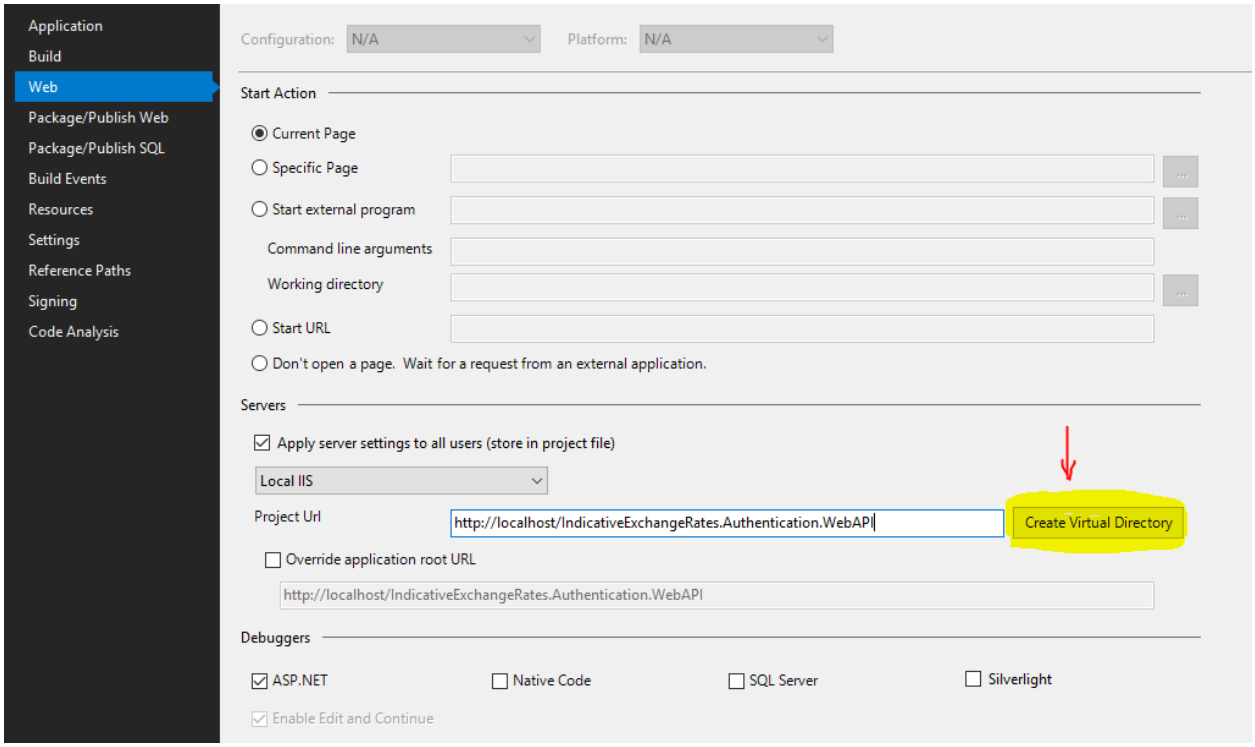
Preparing The Working Environment

To use this API, we first need to setup the Authentication mechanism. The all steps I follow are below.

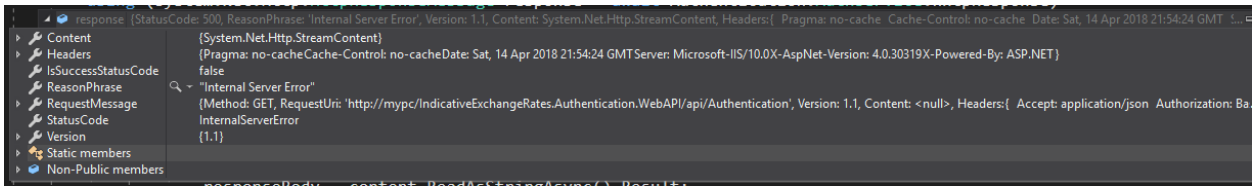
►1: Preparing Authentication System

I chose the authentication method using the basic authentication and the Web API. I wanted to address the control of users who will use this API. I can create or delete users, change their password, etc. There are many authentication mechanisms in the market, but I do not have the experience to use them and I do not want to spend a lot of time applying them. This does not mean that I do not want to learn, but just getting things done quickly since this is a sample application.

Install IIS manager if it is not already installed. You can use another IIS in another machine as well. It will be the host for our Web API project. When a client wishes to use the API, authentication will be performed on this host computer. In my Visual Studio development environment, when I right-clicked the project and asked to create a virtual directory for the project, I hosted Web API project in the local IIS Manager. You should customize your connection string in **web.config** according to your SQL Server and credentials. It can be found in web.config file as the name of "**AuthenticationEntities**".

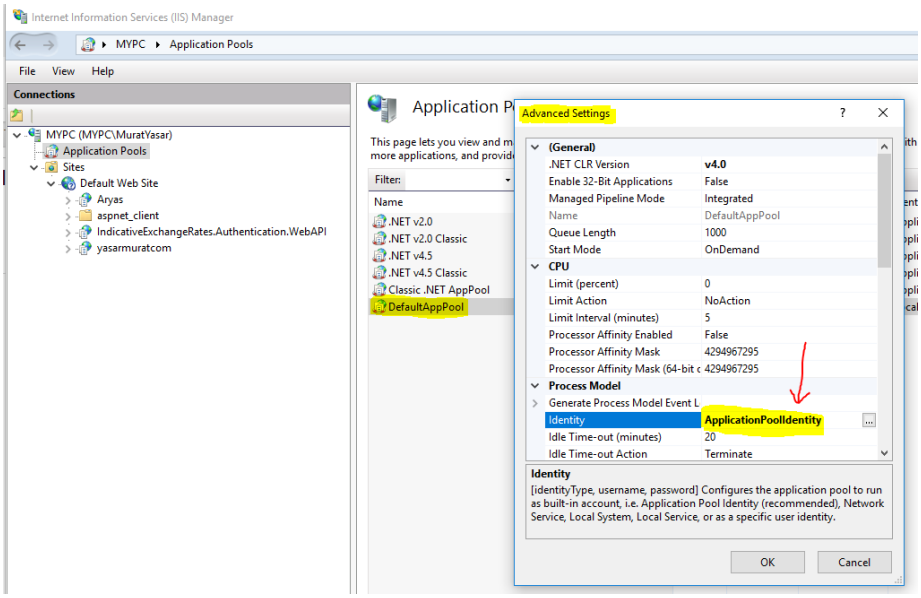


I was getting an error when trying to validate a client from a Windows Forms application. Some googling and found a solution.

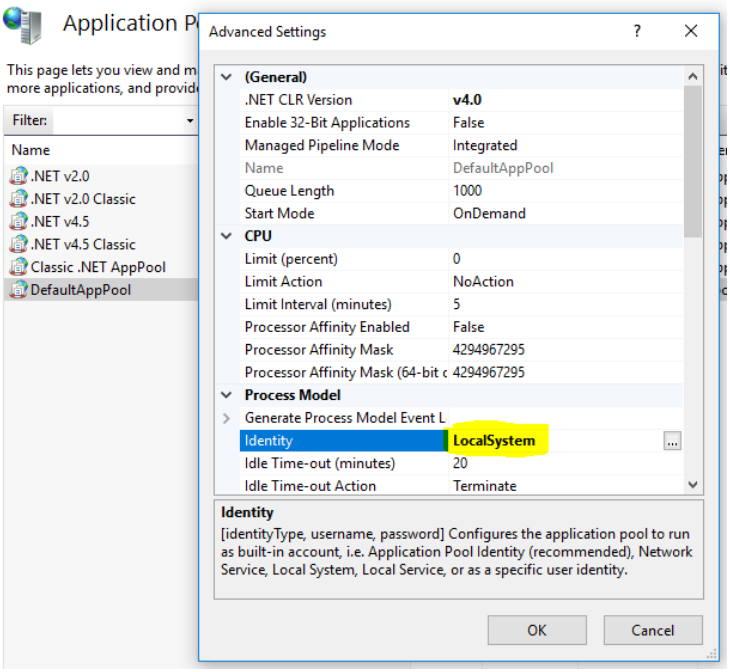


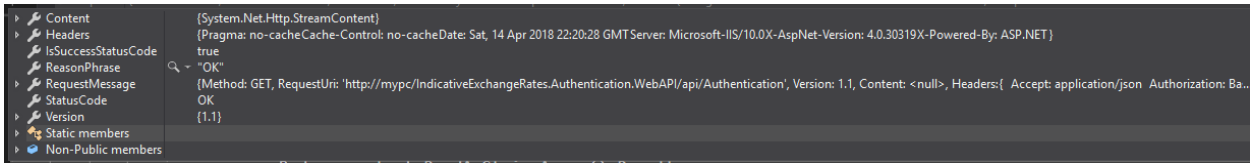
The reason was about ApplicationPool settings.

Before I got this error, the setting was as follows.

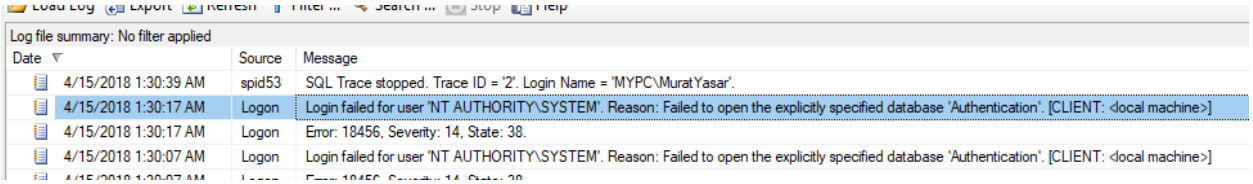


The error went away after changing the **Identity** setting from **ApplicationPoolIdentity** to **LocalSystem**.

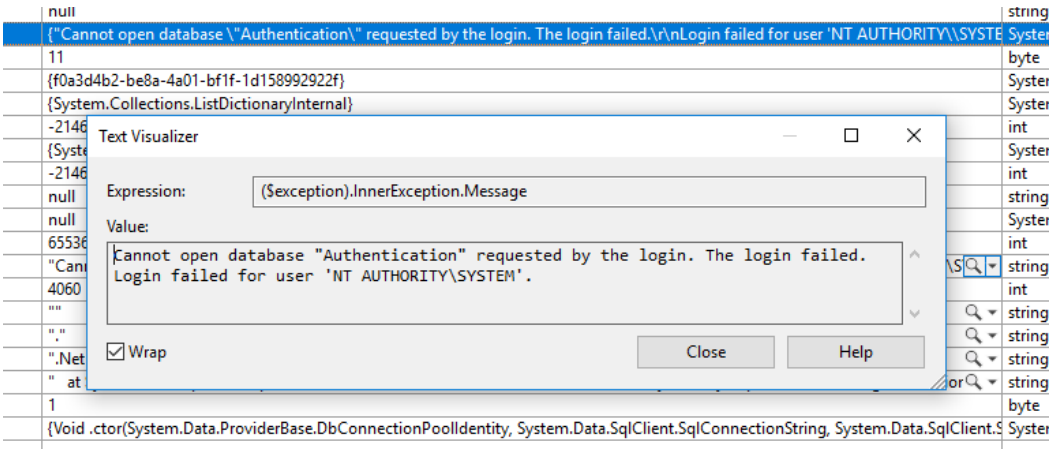




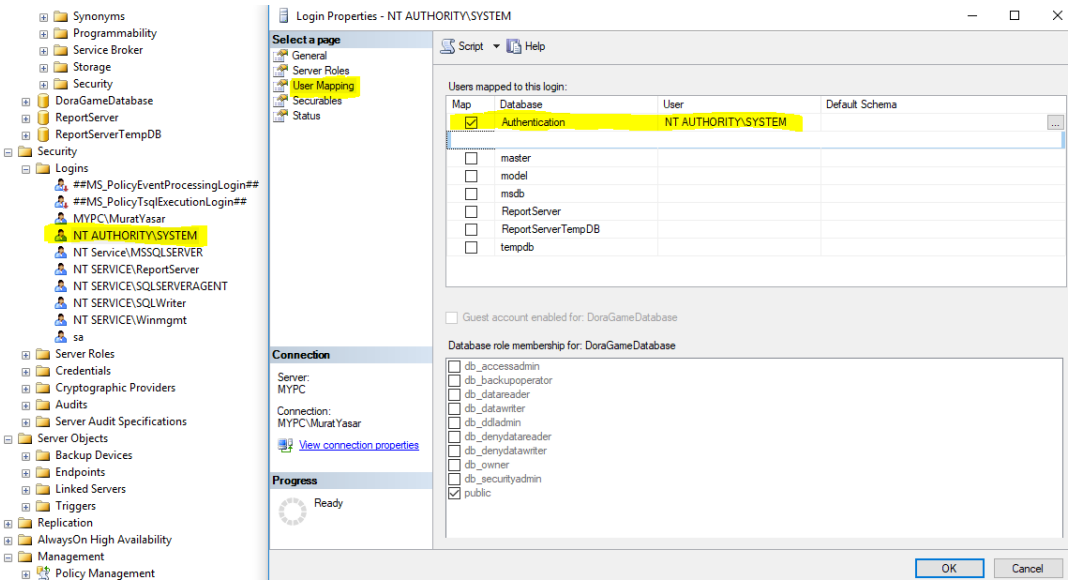
But this time I faced a problem with SQL Server. I have reviewed the SQL Server logs and found the error.



I got the following screen view from Visual Studio.

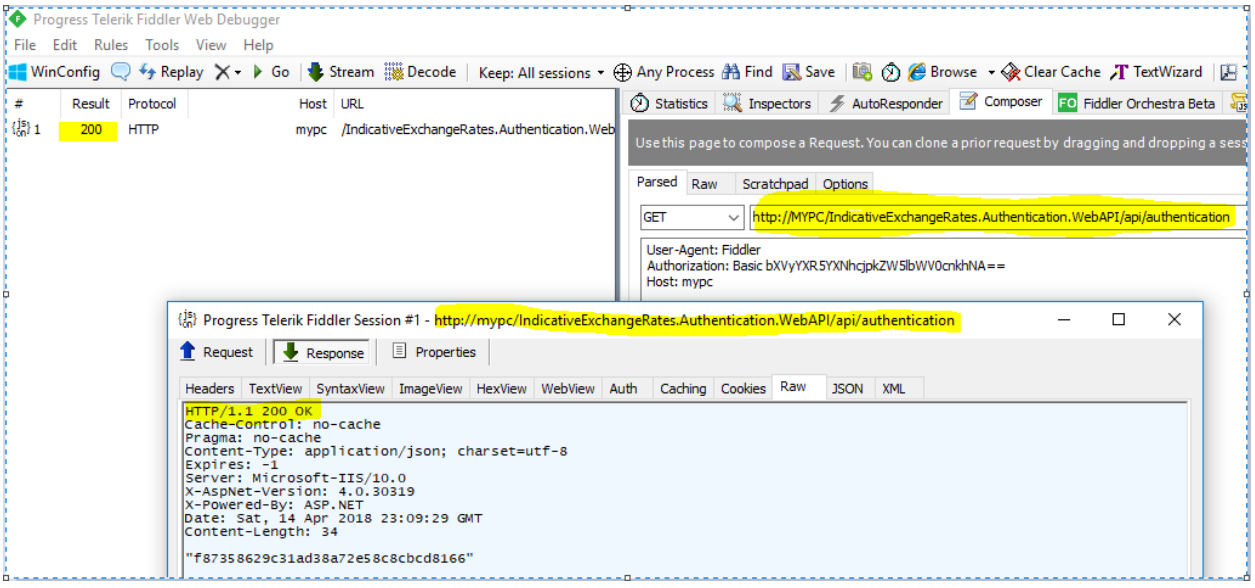


I have mapped the “**NT AUTHORITY\SYSTEM**” user to the **Authentication** database to correct this.



Please select if you have not yet, the **db_datareader** role membership from the above screen. You should have at least read permission so that you can check the user.

After completing all these steps, I tested using **Fiddler** to see if it worked as expected.



The first step, the Authentication mechanism, was established and tested and seen to be working as expected.

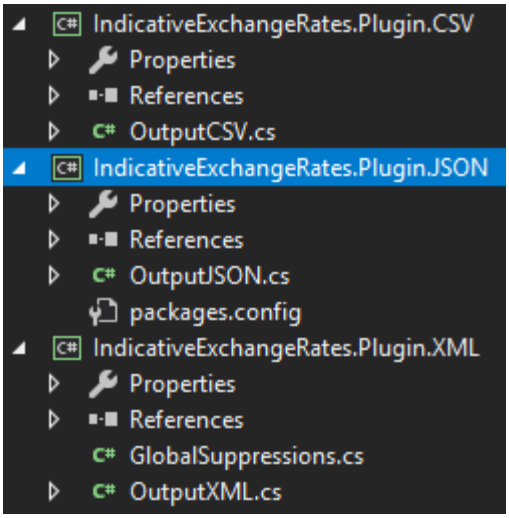
►2: Create New PlugIn

The API already supports XML and CSV output types. I will write another plugin for another output type JSON.

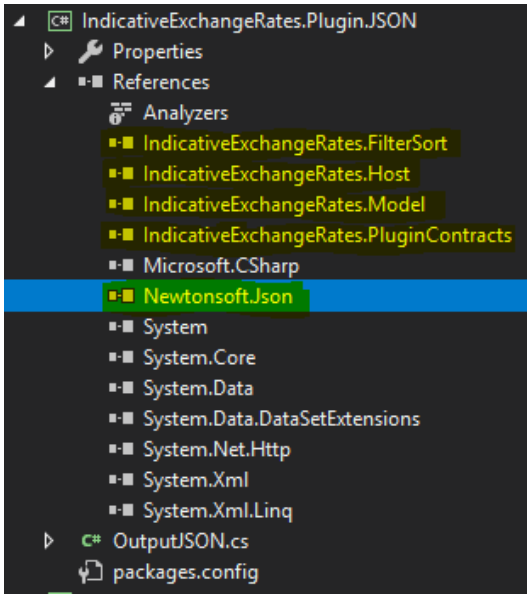
XML and CSV output types had already been added as plugin.

I created a new class library project and named it **IndicativeExchangeRates.Plugin.JSON** since being a follower of naming convention.

I chose **.NET Framework 4.6** as the target framework.



We have to add some references into our newly created class library which is a candidate plugin to our API.



IndicativeExchangeRates.FilterSort is going to be used for filtering and sorting purpose.

IndicativeExchangeRates.Host is going to be used as the core of this API implementation.

IndicativeExchangeRates.Model is going to be used for data modeling. This model will be used throughout the API implementation.

IndicativeExchangeRates.PluginContracts will be used as a contract for the plugin mechanism and also a contract enforcing the capabilities that the newly created plugin must support.

Newtonsoft.Json is going to be used for .NET object to serialize and deserialize. I used nuget package manager to include the project. More details can be found at the followind url.

https://docs.microsoft.com/en-us/nuget/quickstart/install-and-use-a-package-in-visual-studio

The project will consist of just one class file and I will create a name like OutputJSON. I wil use "**Output**" as prefix and the suffix is going to be the name of the output type in our case JSON. At the end, the file name is going to be **OutputJSON.cs**.

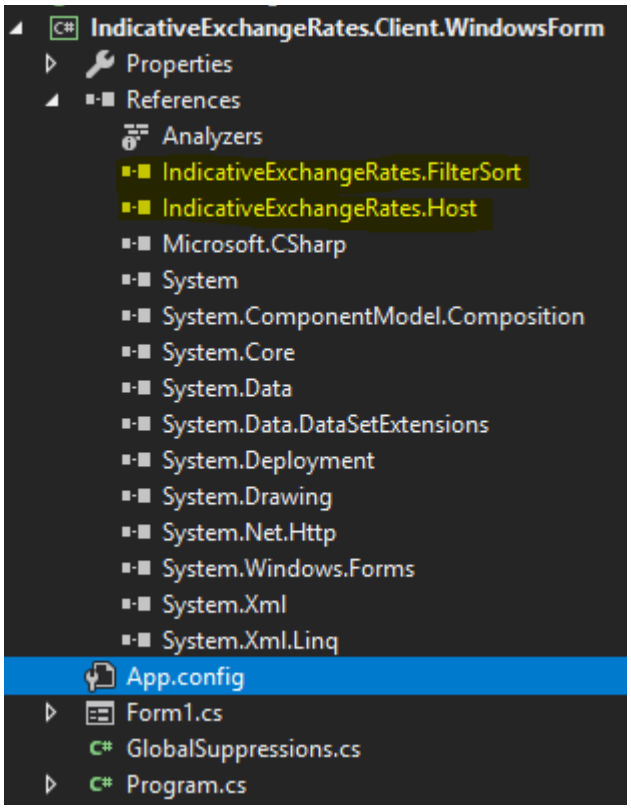
This class will have to implement an interface in order to support certain abilities and also being noticed as a plugin to our API. All the code is as following.

```
namespace IndicativeExchangeRates.Plugin.JSON
{
    0 references
    public class OutputJSON : IPlugin
    {
        6 references | 0 exceptions
        public string Name => this.GetType().Name;

        3 references | 0 exceptions
        public string Description => "Produce JSON Output";

        4 references | 0 exceptions
        public async Task<object> GetAllData()
        {
            var response = await Host.ExchangeRateAPI.GetResult();

            var output = JsonConvert.SerializeObject(response);

            return output;
        }

        4 references | 0 exceptions
        public async Task<object> GetFilteredData(List<ExpressionFilter> filters)
        {
            var response = await Host.ExchangeRateAPI.GetFilteredResult(filters);

            var output = JsonConvert.SerializeObject(response);

            return output;
        }

        4 references | 0 exceptions
        public async Task<object> GetSortedData(List<ExpressionSort> sorts)
        {
            var response = await Host.ExchangeRateAPI.GetSortedResult(sorts);

            var output = JsonConvert.SerializeObject(response);

            return output;
        }

        4 references | 0 exceptions
        public async Task<object> GetFilteredAndSortedData(List<ExpressionFilter> filters, List<ExpressionSort> sorts)
        {
            var response = await Host.ExchangeRateAPI.GetFilteredAndSortedResult(filters, sorts);

            var output = JsonConvert.SerializeObject(response);

            return output;
        }
    }
}
```

After completing the above steps, our plugin is ready for use. I will explain how to make it active later in this guide.

►3: Creating Client Application That Uses the API

I created Windows Forms application and name it **IndicativeExchangeRates.Client.WindowsForm** and added some references in order to use the API.

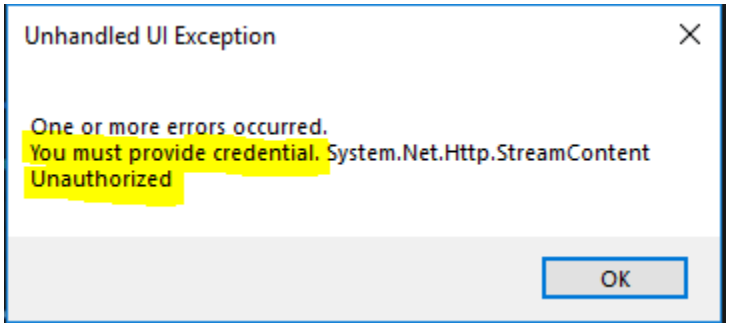


I have already explained what it is used for these references. Let's take a look at the codes.

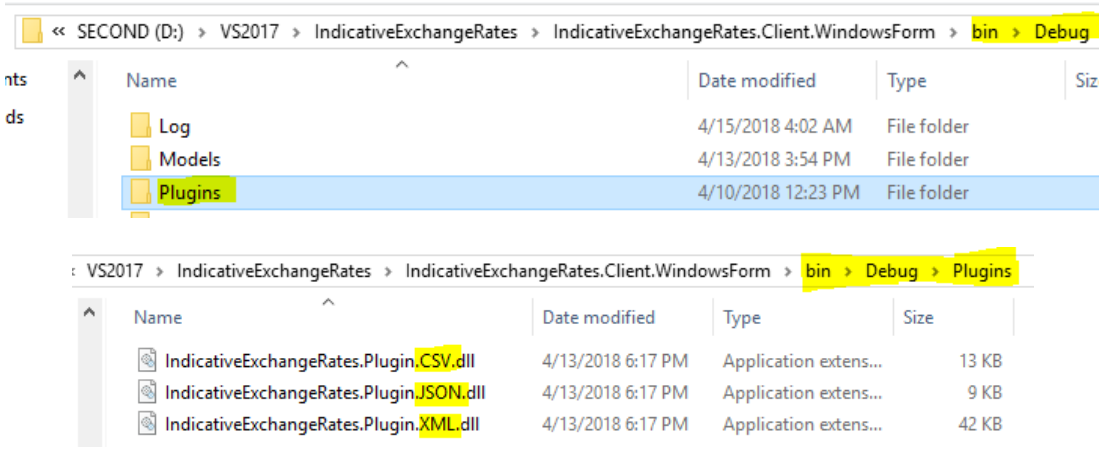
```
namespace IndicativeExchangeRates.Client.WindowsForm
{
    3 references
    public partial class Form1 : Form
    {
        Host.ExchangeRateAPI era = new Host.ExchangeRateAPI("muratyasar", "denemetry!4");

        1 reference | 0 exceptions
        public Form1()
        {
            InitializeComponent();

            Point newLoc = new Point(5, 5);

            foreach (var item in era.GetAvailableOutputTypes)
            {
                Button b = new Button();
                b.Text = item;
                b.Click += B_Click;
                b.Size = new Size(80, 50);
                b.Location = newLoc;
                newLoc.Offset(0, b.Height + 5);
                splitContainer1.Panel1.Controls.Add(b);
            }

        }
```

We create an instance of our core part and supply user name and password. By doing this, the dll (**IndicativeExchangeRates.Host**) will check the credentials and search for available extensions to work with. The username and password information has already been added to the table.

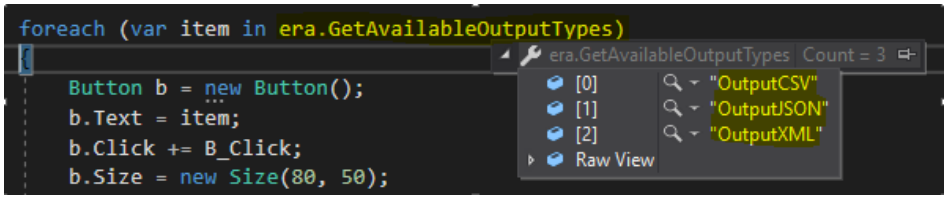
If something goes wrong then a message will popup.



We must specify a **RequestedOutput** value before any data is requested. These values are prepared when you create an instance of the core (**IndicativeExchangeRates.Host**). These values come from the available plugins in the **Plugins** folder. We have to copy the compiled dll's manually from the bin directory of the plugin projects to the **Plugins** folder under the client executable directory.



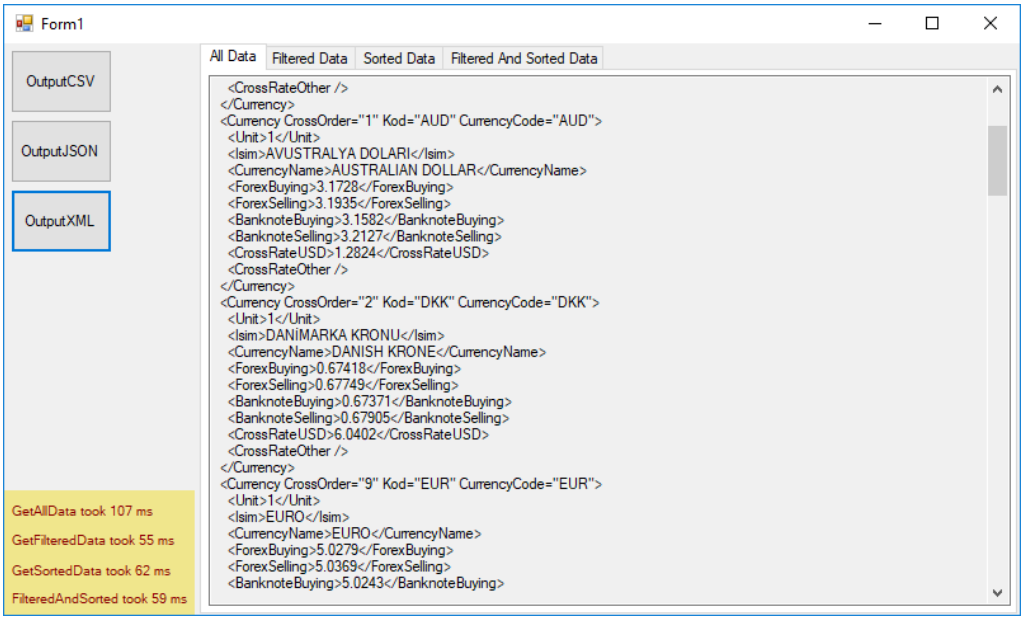
When we debug, we will see **RequestedOutput** values come from Plugins folder.

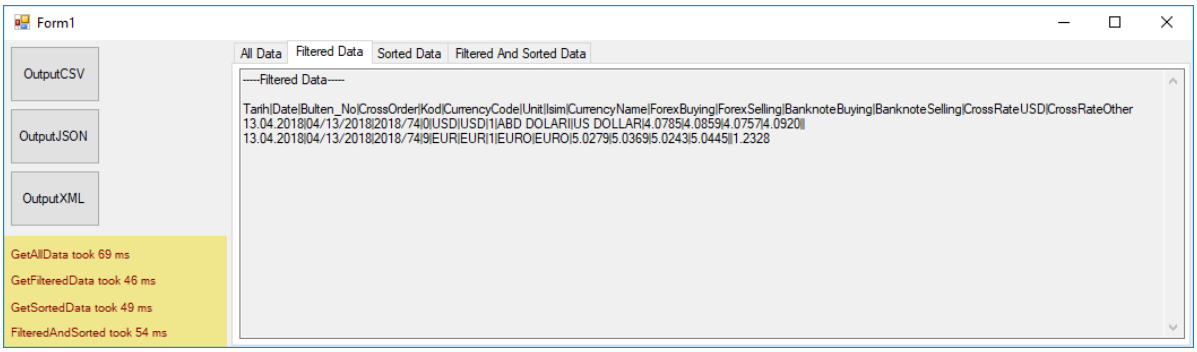


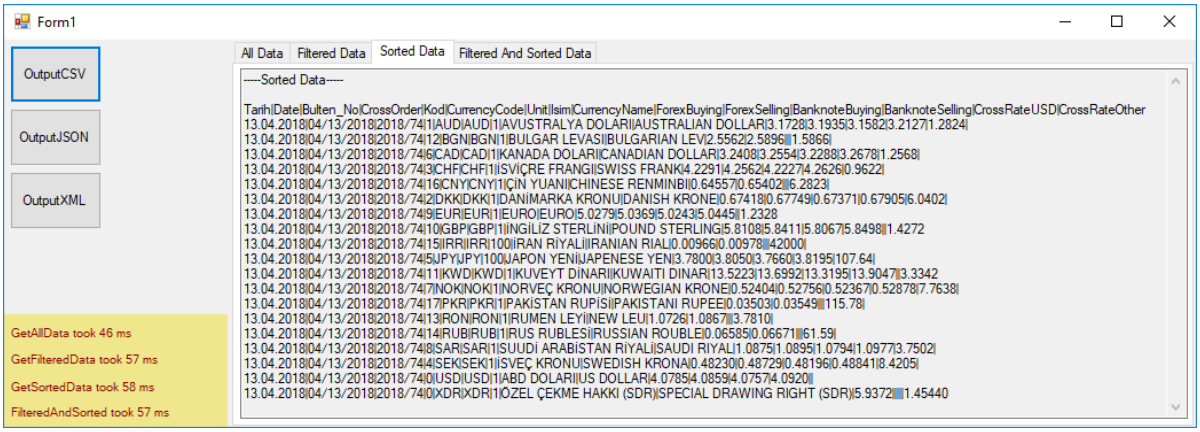
We can also define filter and sorting expression. In a very dynamic way, we can query and/or sort and get the resulting data back by using the API.

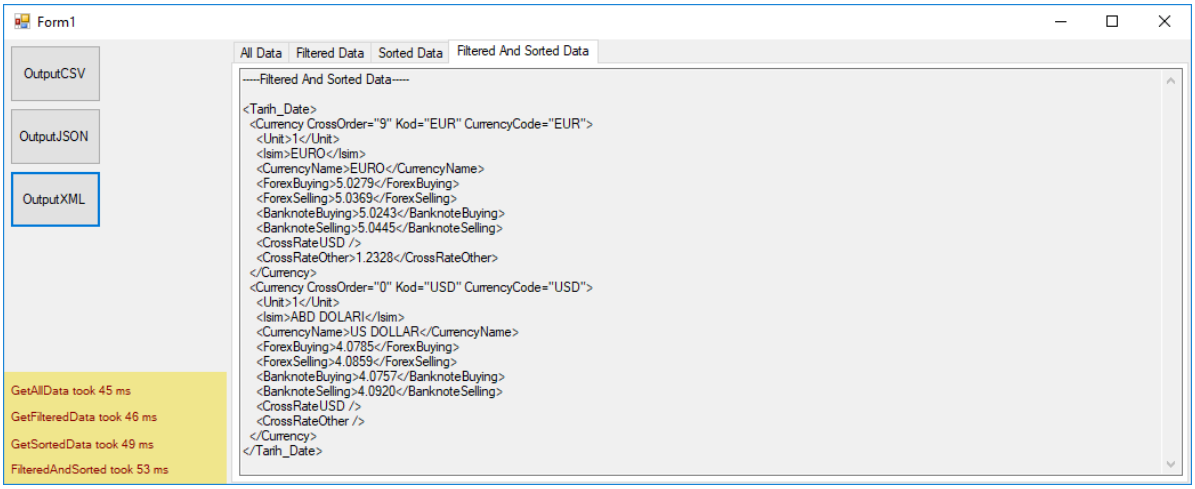
```
switch (outputtype)
{
    case "OutputXML":
        era.RequestedOutput = "OutputXML";
        break;
    case "OutputCSV":
        era.RequestedOutput = "OutputCSV";
        break;
    case "OutputJSON":
        era.RequestedOutput = "OutputJSON";
        break;
}

System.Diagnostics.Stopwatch stopwatch = new System.Diagnostics.Stopwatch();
stopwatch.Start();

List<ExpressionFilter> filters = new List<ExpressionFilter>
{
    new ExpressionFilter
    {
        PropertyName =FilterSort.Enums.FilterColumNames.CurrencyCode,
        Value ="USD",
        Comparison =FilterSort.Enums.Comparison.Equal,
        LogicalOperator=FilterSort.Enums.LogicalOperation.Or
    },
    new ExpressionFilter...
};

List<ExpressionSort> sorts = new List<ExpressionSort>
{
    new ExpressionSort
    {
        PropertyName =FilterSort.Enums.FilterColumNames.CurrencyCode,
        SortDirection = FilterSort.Enums.SortDirection.Ascending
    },
    new ExpressionSort...,
};
```

We can see all the codes by looking at the project codes. Now, I will show some sample screen outputs when you run the Windows Forms project in this solution.









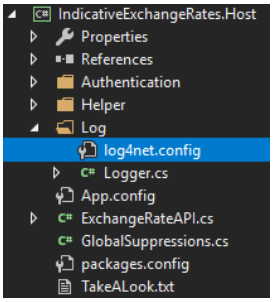
Below I summarize the things to be done in order to write client application that uses the API.

- Create a project and choose .NET Framework 4.6 as the target framework.
- Add the **IndicativeExchangeRates.FilterSort** reference.
- Add the **IndicativeExchangeRates.Host** reference.
- Create a folder in new project's executable directory and name it as **Plugins**.
- Copy the dll file of the plugin you want to use in your application to the newly created **Plugins** folder.

- Create an **app.config** file and add the following lines. If you already have an **app.config** file, update it by adding the following lines.

```
<appSettings>
    <add key="authenticationUrl" value="http://mypc/IndicativeExchangeRates.Authentication.WebAPI/api/Authentication" />
</appSettings>
```

- This setting specifies the Authentication url. Change according to host address.
- Create a folder in new project's executable directory and name it as **Log**.
- Copy the **log4net.config** file to the newly created **Log** folder. This file resides in **IndicativeExchangeRates.Host** project and provides logging ability. We need to provide this file and give it to client while giving other dll files to write software.



- We need to give this configuration file as well when giving other dll files to the people who will use the API. As can be seen in the config file, API creates log files and you can configure this config file however you want. You can disable logging or make only error logging as well. You can write the logs to the database by configuring only this config file.
- Once we have set them up, we can write code that uses the API.
- Below is the sample for getting XML data synchronously.

```
private void buttonXML_Click(object sender, EventArgs e)
{
    textBox1.Text = string.Empty;

    using (IndicativeExchangeRates.Host.ExchangeRateAPI era = new IndicativeExchangeRates.Host.ExchangeRateAPI("muratyasar", "denemetry!4"))
    {
        era.RequestedOutput = "OutputXML";
        var result = Task.Run(() => era.GetAllData());
        textBox1.Text = result.Result.ToString();
    }
}
```

- The following example is for getting the XML data asynchronously.

```
private async void buttonXMLAsync_Click(object sender, EventArgs e)
{
    textBox1.Text = string.Empty;

    using (IndicativeExchangeRates.Host.ExchangeRateAPI era = new IndicativeExchangeRates.Host.ExchangeRateAPI("muratyasar", "denemetry!4"))
    {
        era.RequestedOutput = "OutputXML";
        var result = await era.GetAllData();
        textBox1.Text = result.ToString();
    }
}
```

- Below is the sample for getting CSV data synchronously.

```
private void buttonCSV_Click(object sender, EventArgs e)
{
    textBox1.Text = string.Empty;

    using (IndicativeExchangeRates.Host.ExchangeRateAPI era = new IndicativeExchangeRates.Host.ExchangeRateAPI("muratyasar", "denemetry!4"))
    {
        era.RequestedOutput = "OutputCSV";
        var result = Task.Run(() => era.GetAllData());
        textBox1.Text = result.Result.ToString();
    }
}
```

- The following example is for getting the CSV data asynchronously.

```
private async void buttonCSVAsync_Click(object sender, EventArgs e)
{
    textBox1.Text = string.Empty;

    using (IndicativeExchangeRates.Host.ExchangeRateAPI era = new IndicativeExchangeRates.Host.ExchangeRateAPI("muratyasar", "denemetry!4"))
    {
        era.RequestedOutput = "OutputCSV";
        var result = await era.GetAllData();
        textBox1.Text = result.ToString();
    }
}
```

- Below is the sample for getting JSON data synchronously. Also, as I mentioned above, we need to include the necessary dll files in the project using the nuget package manager to install the necessary libraries.

```
private void buttonJSON_Click(object sender, EventArgs e)
{
    textBox1.Text = string.Empty;

    using (IndicativeExchangeRates.Host.ExchangeRateAPI era = new IndicativeExchangeRates.Host.ExchangeRateAPI("muratyasar", "denemetry!4"))
    {
        era.RequestedOutput = "OutputJSON";
        var result = Task.Run(() => era.GetAllData());
        textBox1.Text = result.Result.ToString();
    }
}
```

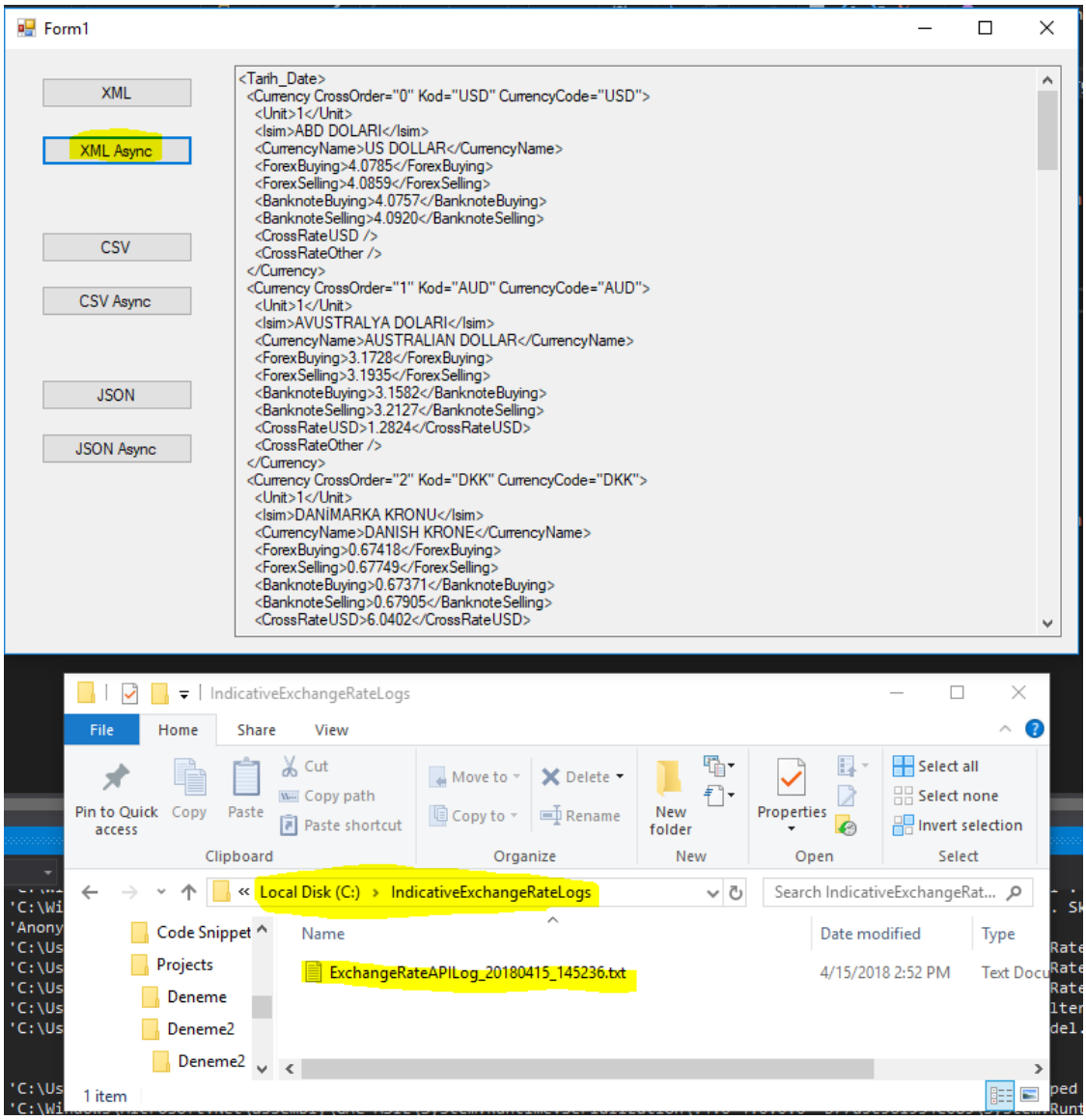
- The following example is for getting the JSON data asynchronously.

```
private async void buttonJSONAsync_Click(object sender, EventArgs e)
{
    textBox1.Text = string.Empty;

    using (IndicativeExchangeRates.Host.ExchangeRateAPI era = new IndicativeExchangeRates.Host.ExchangeRateAPI("muratyasar", "denemetry!4"))
    {
        era.RequestedOutput = "OutputJSON";
        var result = await era.GetAllData();
        textBox1.Text = result.ToString();
    }
}
```

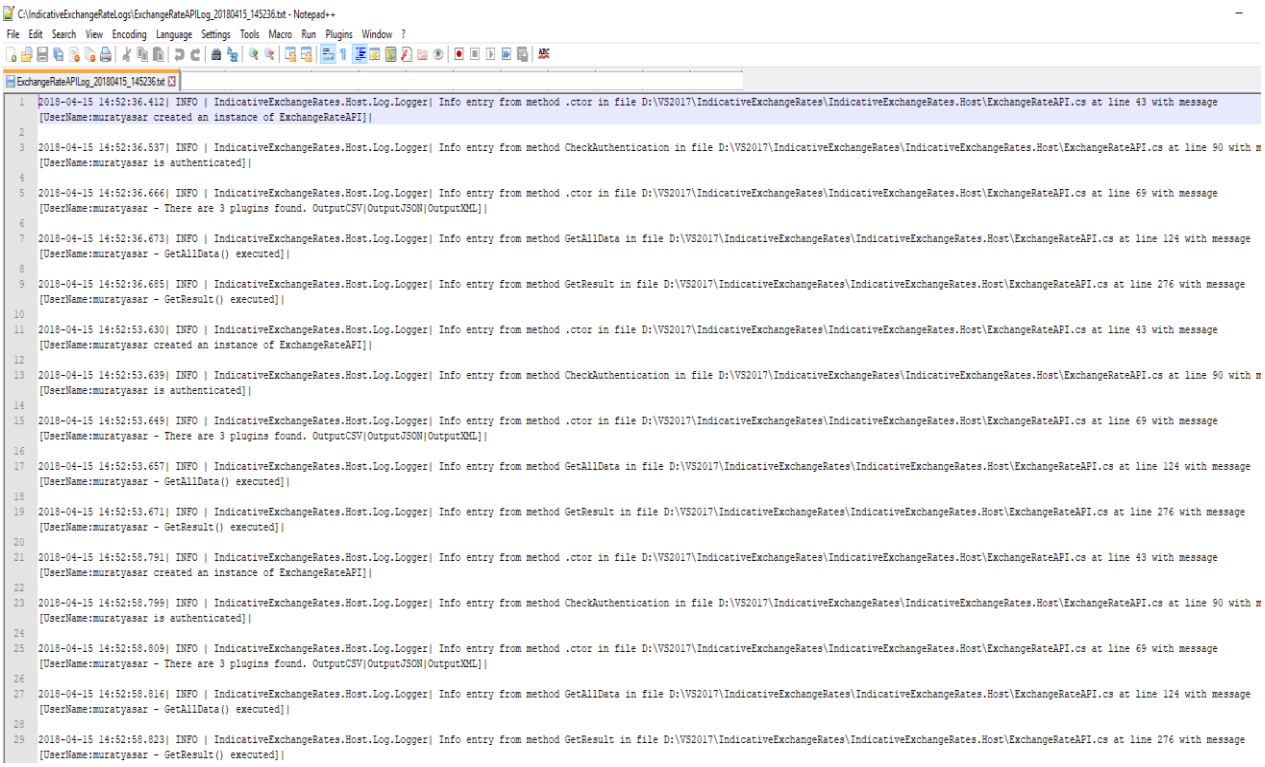
- Below is an example of how to retrieve XML data using filters and sort parameters. The API has only filters and only sort methods, but I wanted to show you how to use all of them with this single code block.

```
private void buttonXML_Click(object sender, EventArgs e)
{
    textBox1.Text = string.Empty;

    using (IndicativeExchangeRates.Host.ExchangeRateAPI era = new IndicativeExchangeRates.Host.ExchangeRateAPI("muratyasar", "denemetry!4"))
    {
        List<ExpressionFilter> filters = new List<ExpressionFilter>
        {
            new ExpressionFilter
            {
                PropertyName =IndicativeExchangeRates.FilterSort.Enums.FilterColumNames.CurrencyCode,
                Value ="USD",
                Comparison =IndicativeExchangeRates.FilterSort.Enums.Comparison.Equal,
                LogicalOperator =IndicativeExchangeRates.FilterSort.Enums.LogicalOperation.Or
            },
            new ExpressionFilter
            {
                PropertyName =IndicativeExchangeRates.FilterSort.Enums.FilterColumNames.CurrencyCode,
                Value ="EUR",
                Comparison =IndicativeExchangeRates.FilterSort.Enums.Comparison.Equal,
                LogicalOperator =IndicativeExchangeRates.FilterSort.Enums.LogicalOperation.Or
            }
        };

        List<ExpressionSort> sorts = new List<ExpressionSort>
        {
            new ExpressionSort
            {
                PropertyName =IndicativeExchangeRates.FilterSort.Enums.FilterColumNames.CurrencyCode,
                SortDirection = IndicativeExchangeRates.FilterSort.Enums.SortDirection.Ascending
            },
            new ExpressionSort
            {
                PropertyName =IndicativeExchangeRates.FilterSort.Enums.FilterColumNames.ForexBuying,
                SortDirection = IndicativeExchangeRates.FilterSort.Enums.SortDirection.Descending
            },
        };

        era.RequestedOutput = "OutputXML";
        var result = Task.Run(() => era.GetFilteredAndSortedData(filters, sorts));
        textBox1.Text = result.Result.ToString();
    }
}
```

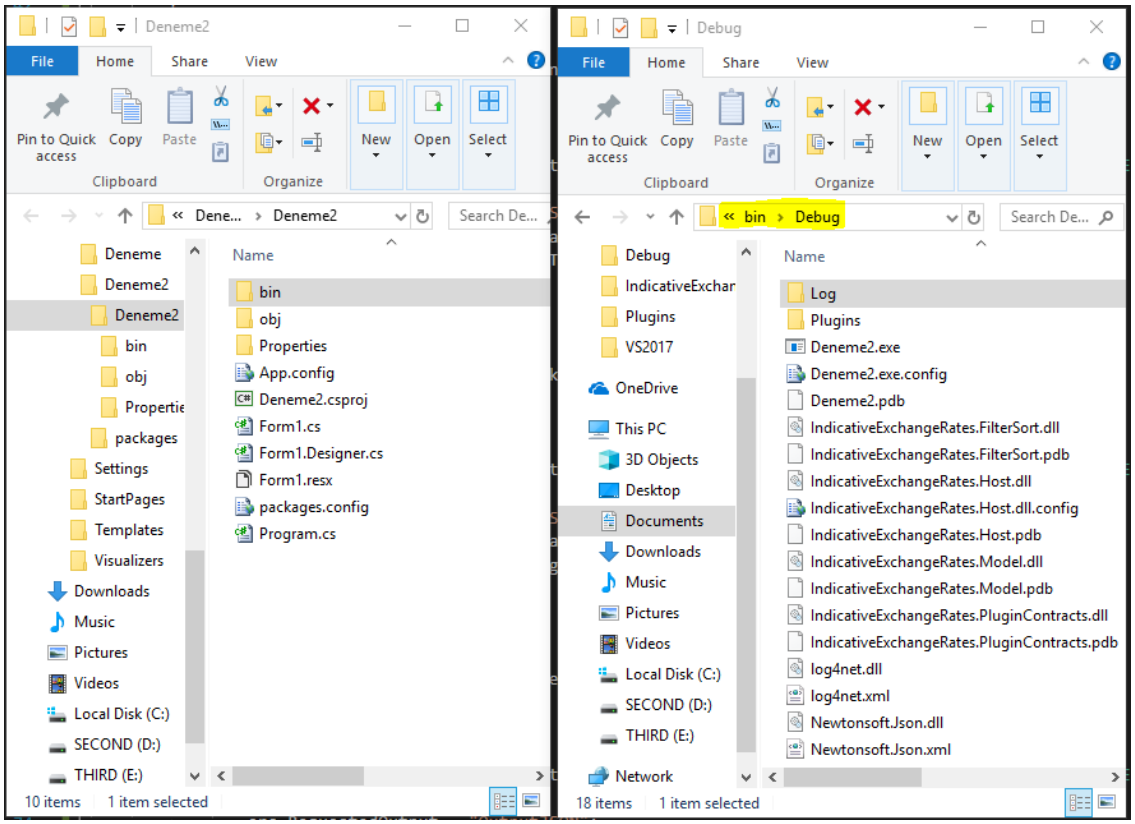
- Once we have the API in hand, we can get results and display them as we want. Each request creates log information to be stored in the folder specified in the **log4net.config** file. The screen images I have taken in my environment are as follows.

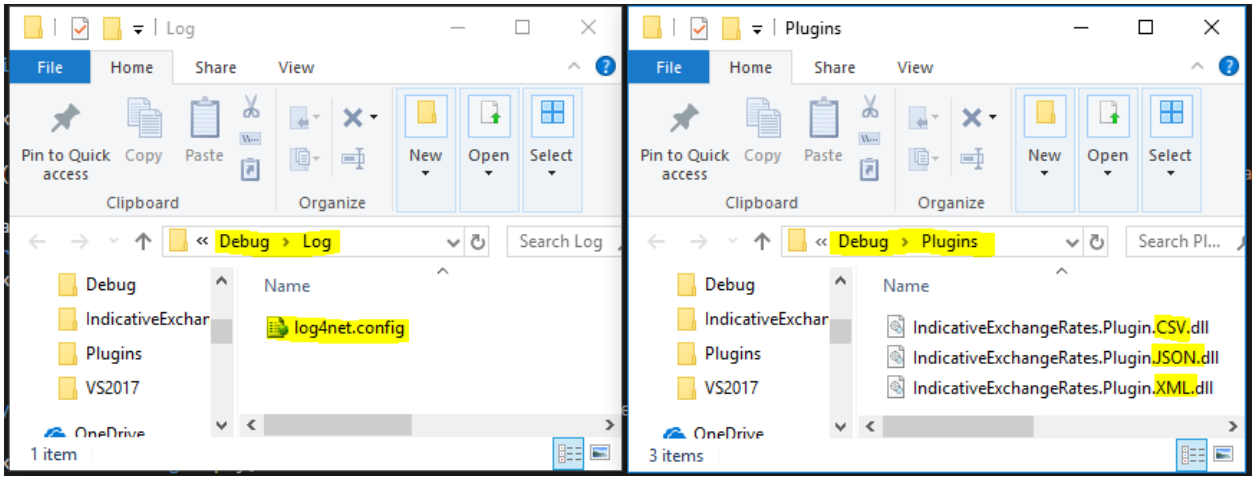


- Below is the content of a log file.



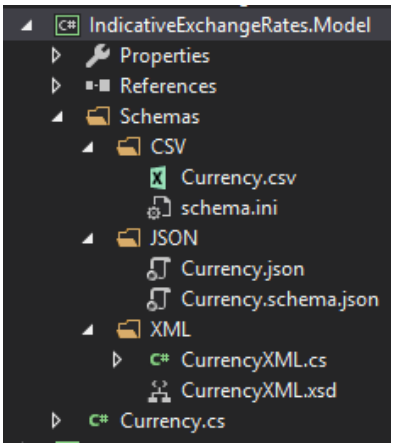
- After you have taken all of these steps, you should have a screenshot similar to the one below in your project bin folder.





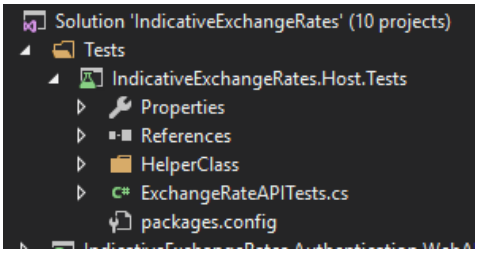
►4: Data Format Specifications

I have added the schema files as well. The files are located in the **IndicativeExchangeRates.Model** project. The following screenshot shows where they are.

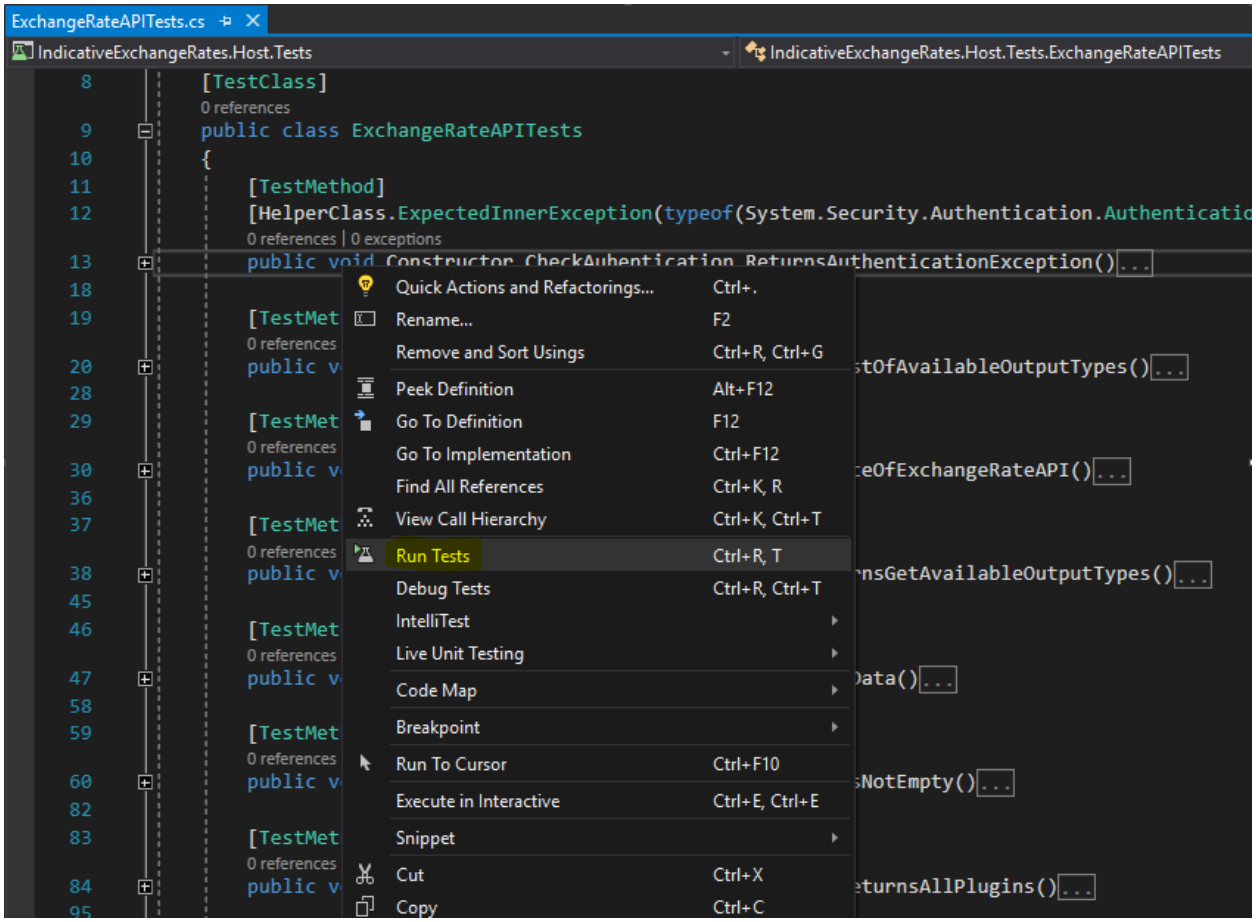


►5: Unit Tests

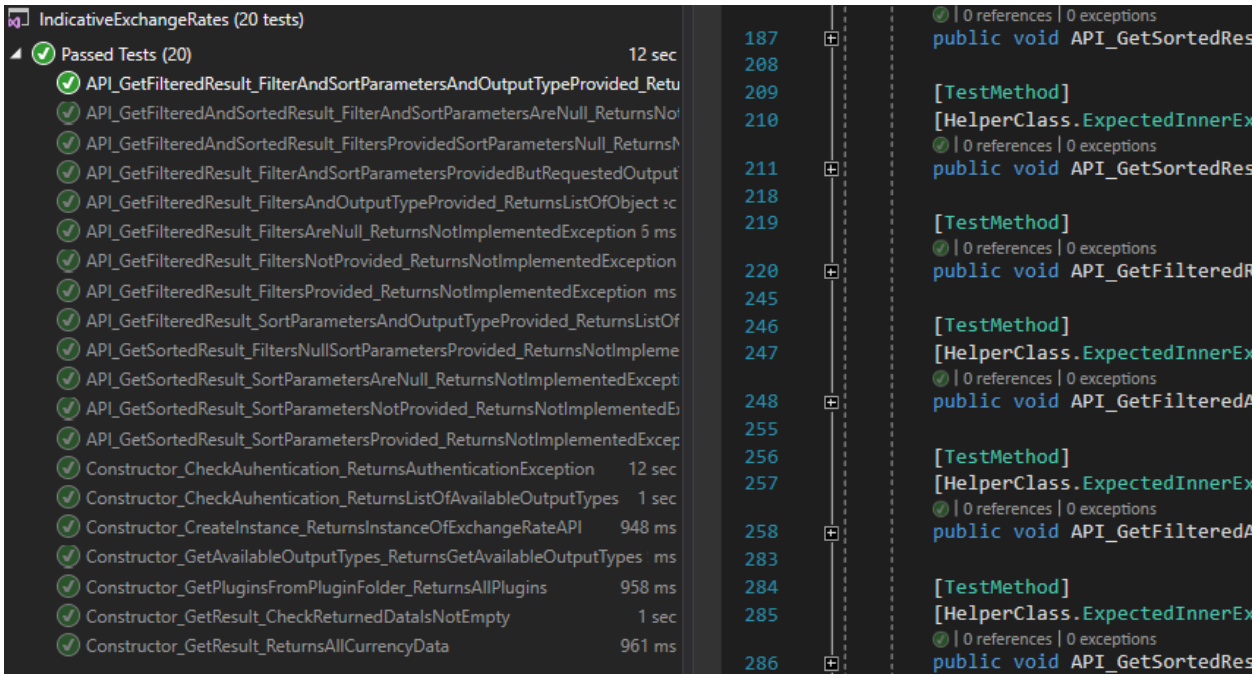
I wrote unit tests. You can see where they are from the screenshot below.



To run the unit tests, I opened the **ExchangeRateAPITests.cs** file with Visual Studio, right-clicked on the methods, and ran the **"Run Tests"** command.



After running test methods I got a screenshot like the one below.



►6: Brief Explanation Of My Solution

The API is written for getting daily indicative exchange rates from Central Bank of the Republic of Turkey. It supports 3 output types which are XML, CSV and JSON.

The API was written as a class library project with the logging and authentication mechanism. As explained in detail above, output dll files should be referenced in a software solution that wants to write code against this API.

As explained above, the client who will use this API has to do some preparation in their executable folder.

If it does not exist **app.config** file should be added. Updating should be done if it is available. The required setting contains the URL information we will use for authentication.

```
<appSettings>
     <add key="authenticationUrl" value="http://mypc/IndicativeExchangeRates.Authentication.WebAPI/api/Authentication" />
</appSettings>
```

They have to create a folders and name them **Plugins** and **Log** respectively.

Plugins folder is the home of the all plugin dll files. Client has to copy plugin dll files into this folder manually.

Log folder is the home of logging configuration file and its name is **log4net.config**. Client also has to copy this configuration file into this folder manually. This does not mean log mechanism has to work and log all processes. You can disable log mechanism, inside log4net.config file according to your needs.

You can also set your configuration to **log into a database** from this config file. If this is the case then you can create a database specified in connection string entry of this file.

Client also has to copy necessary dll files into executable directory if they use some other plugin. We copied, actually not, used nuget for **Newtonsoft.Json** dll files in order to use JSON plugin in our solution.

I use singleton and factory design patterns where I see fit.

I used **log4net** and **Newtonsoft.Json** as third party libraries.

http://logging.apache.org/log4net/

https://www.newtonsoft.com/json

►7: How to test, debug and open zip file

I put the solution and the documentation in zip format as well.

After unzipping the zip file you will see three folders.

The solution is in API folder as Visual Studio Solution.

Documentation is in Documentation folder as PDF format.

SampleClientApplication_WindowsForms is the folder where you can see a sample client application written against the API.

►8: Final Thoughts

I have to admit that the practice can be done smarter. There is nothing wrong, but when I developed this solution, after seeing some problems, I thought the implementation could be done differently. Could have been used more design patterns after researching each of them. I would liked to be able to apply aspect oritented approach especially for logging and authentication mechanism.

I learned things and I found useful writing this implementation. **I used some code that I have taken from forums on the internet.**

Thank you.

Murat YAŞAR