RESTful Day #6: Request logging and Exception handing/logging in Web APIs using Action Filters, Exception Filters and NLog. – Akhil Mittal, Sachin Verma

Table of Contents

Introduction

We have been learning a lot about WebAPI, its uses, implementations, and security aspects since last five articles of the series. **The article is also written and compiled by my co-author Sachin Verma.** This article of the series will explain how we can handle requests and log them for tracking and for the sake of debugging, how we can handle exceptions and log them. We'll follow centralized way of handling exceptions in WebAPI and write our custom classes to be mapped to the type of exception that we encounter and log the accordingly. I'll use NLog to log requests and exceptions as well. We'll leverage the capabilities of Exception Filters and Action Filters to centralize request logging and exception handling in WebAPI.

Roadmap

Following is the roadmap I have setup to learn WebAPI step by step,

- RESTful Day #1: Enterprise-level application architecture with Web APIs using Entity Framework, Generic Repository pattern and Unit of Work.
- RESTful Day #2: Inversion of control using dependency injection in Web APIs using Unity Container and Bootstrapper.
- RESTful Day #3: Resolve dependency of dependencies using Inversion of Control and dependency injection in ASP.Net Web APIs with Unity Container and Managed Extensibility Framework (MEF).
- RESTful Day #4: Custom URL Re-Writing/Routing using Attribute Routes in MVC 4 Web APIs.
- **RESTful Day #5: Basic Authentication and Token-based custom Authorization in Web APIs using Action Filters.**
- RESTful Day #6: Request logging and Exception handing/logging in Web APIs using Action Filters, Exception Filters and nLog.
- RESTful Day #7: Unit testing ASP.Net Web API controllers using nUnit.
- RESTful Day #8: Extending OData support in ASP.Net Web APIs.

I'll purposely use Visual Studio 2010 and .net Framework 4.0 because there are few implementations that are very hard to find in .Net Framework 4.0, but I'll make it easy by showing how we can do it.

Request Logging

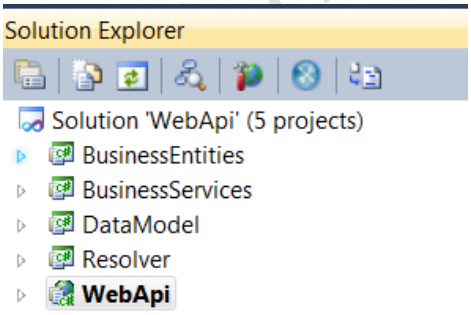
Since we are writing web services, we are exposing our end points. We must know where the requests are coming from and what requests are coming to our server. Logging could be very beneficial and helps us in a lot of ways like, debugging, tracing, monitoring and analytics.

We already have an existing design. If you open the solution, you'll get to see the structure as mentioned below or one can also implement this approach in their existing solution as well,

Setup NLog in WebAPI

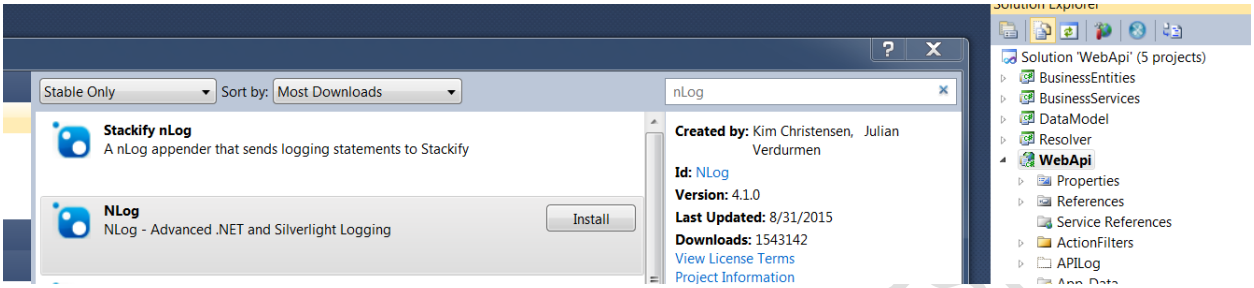
NLog serves various purposes but primarily logging. We'll use NLog for logging into files and windows event as well. You can read more about NLog at http://NLog-project.org/

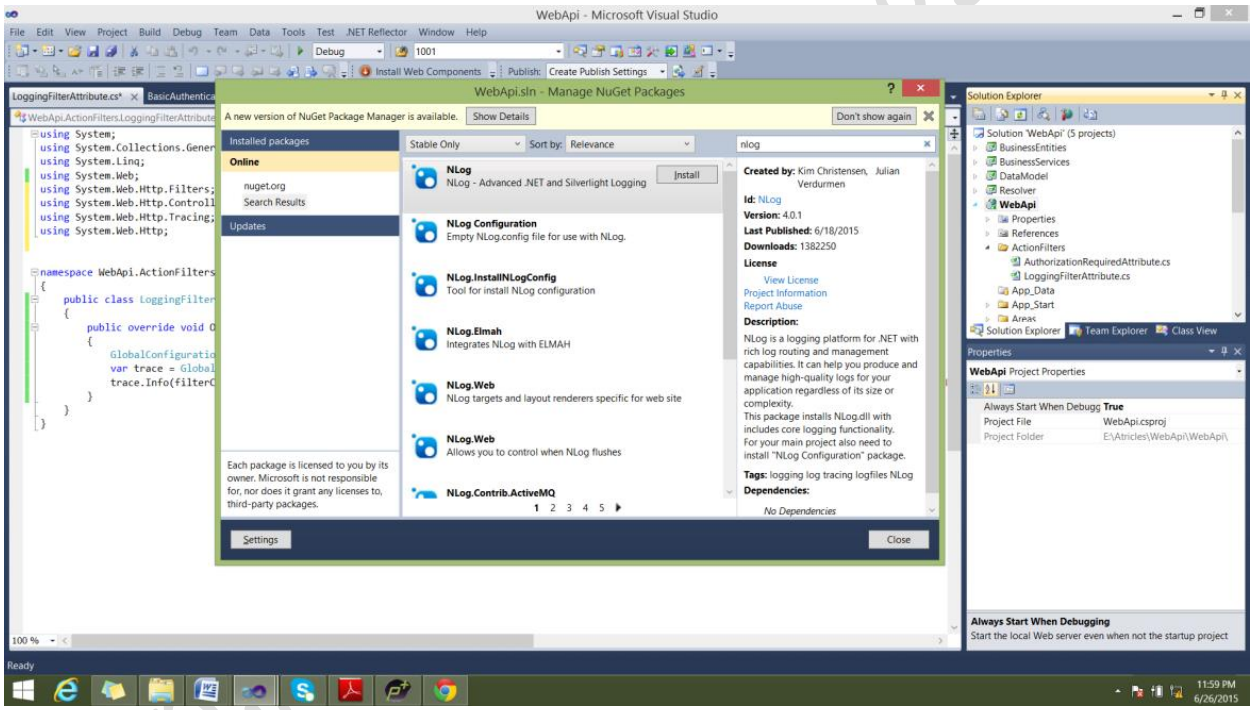
One can use the sample application that we used in Day#5 or can have any other application as well. I am using the existing sample application that we were following throughout all the parts of this series. Our application structure looks something like,



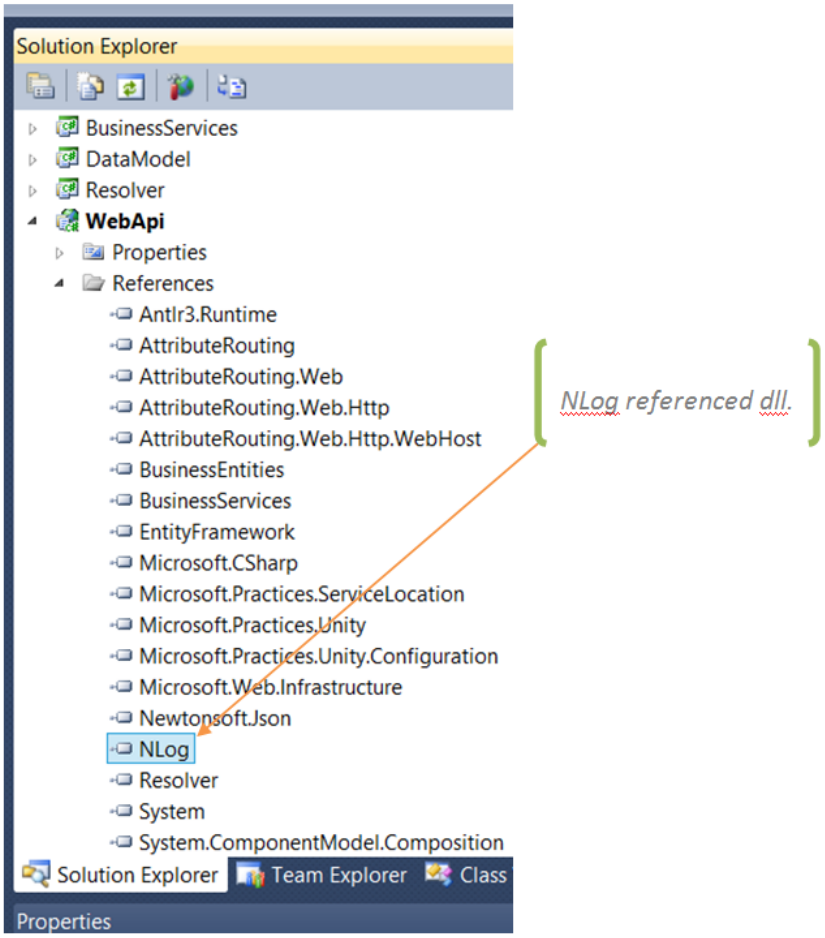
Step 1: Download NLog Package

Right click WebAPI project and select manage Nuget Packages from the list. When the Nuget Package Manager appears, search for NLog. You'll get Nlog like shown in image below, just install it to our project,





After adding this you will find following NLog dll referenced in your application –



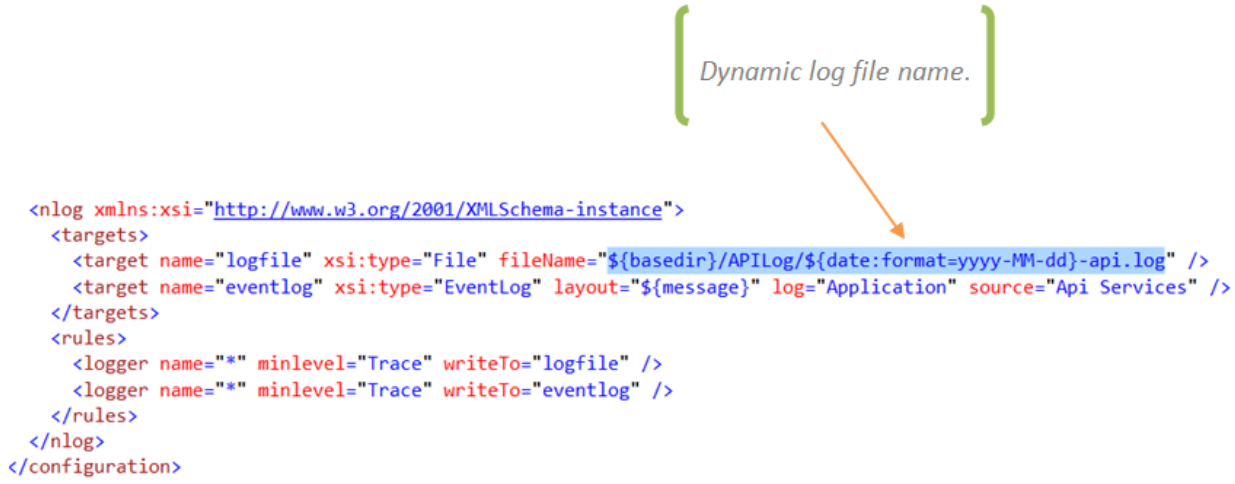
Step 2: Configuring NLog

To configure NLog with application add following settings in our existing WebAPI web.config file,

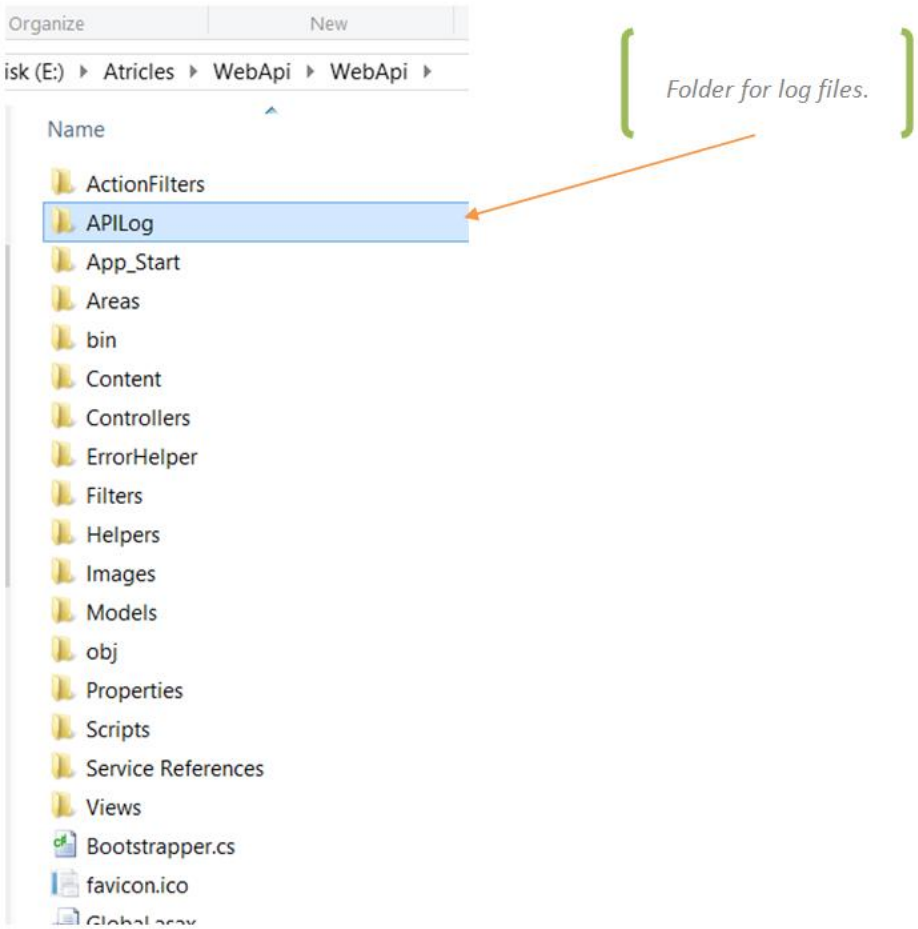
ConfigSection –

```
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 --:
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework
    <section name="nlog" type="NLog.Config.ConfigSectionHandler, NLog" />
  </configSections>
```

Configuration Section – I have added the <NLog> section to configuration and defined the path and format dynamic target log file name, also added the eventlog source to Api Services.



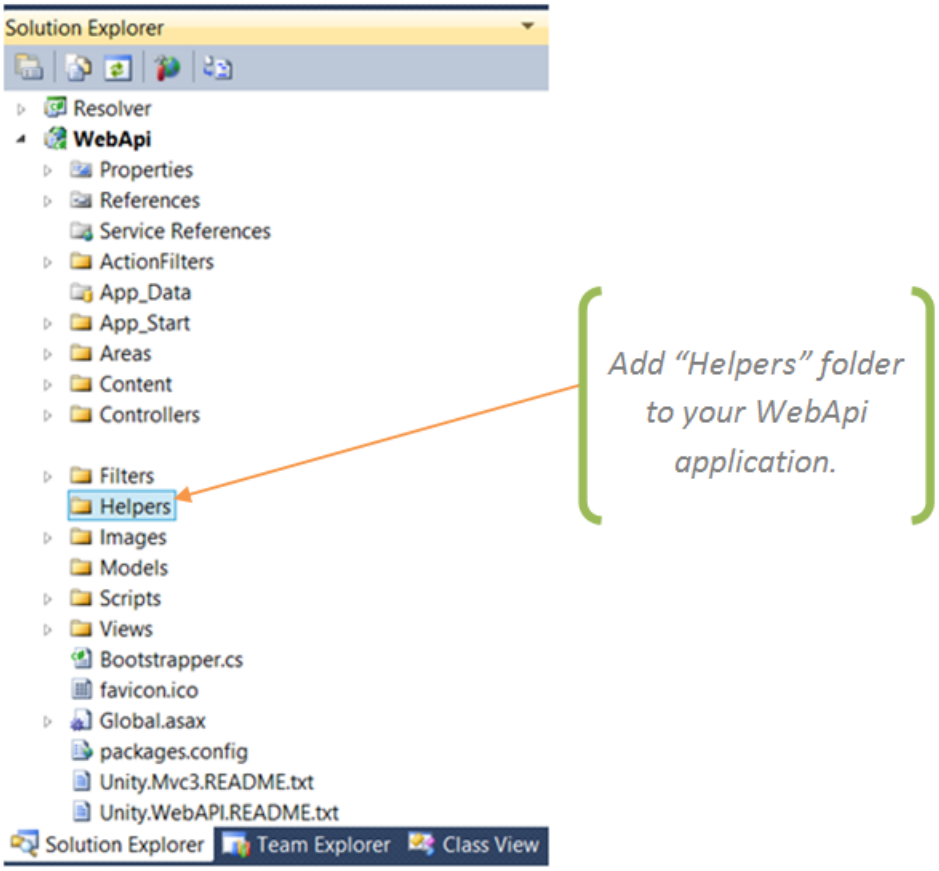
As mentioned in above target path, I have also created to “APILog” folder in the base directory of application –



Now we have configured the NLog in our application, and it is ready to start work for request logging. Note that in the rules section we have defined rules for logging in files as well as in windows events log as well, you can choose both of them or can opt for one too. Let's start with logging request in application, with action filters –

NLogger Class

Add a folder "Helpers" in the API, which will segregate the application code for readability, better understanding and maintainability.



To start add our main class "NLogger", which will responsible for all types of errors and info logging, to same Helper folder. Here NLogger class implements ITraceWriter interface, which provides "Trace" method for the service request –

```
#region Using namespaces.
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http.Tracing;
using NLog;
using System.Net.Http;
using System.Text;
using WebApi.ErrorHelper;
#endregion

namespace WebApi.Helpers
{
    /// <summary>
    /// Public class to log Error/info messages to the access log file
    /// </summary>
    public sealed class NLogger : ITraceWriter
    {
        #region Private member variables.
        private static readonly Logger ClassLogger = LogManager.GetCurrentClassLogger();

        private static readonly Lazy<Dictionary<TraceLevel, Action<string>>> LoggingMap = new
Lazy<Dictionary<TraceLevel, Action<string>>>(() => new Dictionary<TraceLevel, Action<string>>
{ { TraceLevel.Info, ClassLogger.Info }, { TraceLevel.Debug, ClassLogger.Debug }, {
TraceLevel.Error, ClassLogger.Error }, { TraceLevel.Fatal, ClassLogger.Fatal }, { TraceLevel.Warn,
ClassLogger.Warn } });
        #endregion

        #region Private properties.
        /// <summary>
        /// Get property for Logger
        /// </summary>
        private Dictionary<TraceLevel, Action<string>> Logger
        {
            get { return LoggingMap.Value; }
        }
        #endregion

        #region Public member methods.
        /// <summary>
        /// Implementation of TraceWriter to trace the logs.
        /// </summary>
        /// <param name="request"></param>
        /// <param name="category"></param>
        /// <param name="level"></param>
```

```
        /// <param name="traceAction"></param>
        public void Trace(HttpRequestMessage request, string category, TraceLevel level,
Action<TraceRecord> traceAction)
        {
            if (level != TraceLevel.Off)
            {
                if (traceAction != null && traceAction.Target != null)
                {
                    category = category + Environment.NewLine + "Action Parameters : " +
traceAction.Target.ToJSON();
                }
                var record = new TraceRecord(request, category, level);
                if (traceAction != null) traceAction(record);
                Log(record);
            }
        }
        #endregion

        #region Private member methods.
        /// <summary>
        /// Logs info/Error to Log file
        /// </summary>
        /// <param name="record"></param>
        private void Log(TraceRecord record)
        {
            var message = new StringBuilder();

            if (!string.IsNullOrWhiteSpace(record.Message))
                message.Append("").Append(record.Message + Environment.NewLine);

            if (record.Request != null)
            {
                if (record.Request.Method != null)
                    message.Append("Method: " + record.Request.Method + Environment.NewLine);

                if (record.Request.RequestUri != null)
                    message.Append("").Append("URL: " + record.Request.RequestUri +
Environment.NewLine);

                if (record.Request.Headers != null && record.Request.Headers.Contains("Token") &&
record.Request.Headers.GetValues("Token") != null &&
record.Request.Headers.GetValues("Token").FirstOrDefault() != null)
                    message.Append("").Append("Token: " +
record.Request.Headers.GetValues("Token").FirstOrDefault() + Environment.NewLine);
```

```
        }

        if (!string.IsNullOrWhiteSpace(record.Category))
            message.Append("").Append(record.Category);

        if (!string.IsNullOrWhiteSpace(record.Operator))
            message.Append(" ").Append(record.Operator).Append("
").Append(record.Operation);


        Logger[record.Level](Convert.ToString(message) + Environment.NewLine);
    }
    #endregion
  }
}
```

Adding Action Filter

Action filter will be responsible for handling all the incoming requests to our APIs and logging them using NLogger class. We have "OnActionExecuting" method that is implicitly called if we mark our controllers or global application to use that particular filter. So each time any action of any controller will be hit, our "OnActionExecuting" method will execute to log the request.

Step 1: Adding LoggingFilterAttribute class

Create a class `LoggingFilterAttribute` to "ActionFilters" folder and add following code -

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http.Filters;
using System.Web.Http.Controllers;
using System.Web.Http.Tracing;
using System.Web.Http;
using WebApi.Helpers;


namespace WebApi.ActionFilters
{
    public class LoggingFilterAttribute : ActionFilterAttribute
    {
```

```
        public override void OnActionExecuting(HttpActionContext
filterContext)
        {

GlobalConfiguration.Configuration.Services.Replace(typeof(ITraceWriter), new
NLogger());
            var trace =
GlobalConfiguration.Configuration.Services.GetTraceWriter();
            trace.Info(filterContext.Request, "Controller : " +
filterContext.ControllerContext.ControllerDescriptor.ControllerType.FullName
+ Environment.NewLine + "Action : " +
filterContext.ActionDescriptor.ActionName, "JSON",
filterContext.ActionArguments);
        }
    }
}
```

The `LoggingFilterAttribute` class derived from `ActionFilterAttribute`, which is under System.Web.Http.Filters and overriding the OnActionExecuting method.
Here I have replaced the default "ITraceWriter" service with our NLogger class instance in the controller's service container. Now `GetTraceWriter()` method will return our instance (instance NLogger class) and Info() will call trace() method of our NLogger class.

Note that the code below,

```
GlobalConfiguration.Configuration.Services.Replace(typeof(ITraceWriter), new
NLogger());
```

is used to resolve dependency between ITaceWriter and NLogger class. Thereafter we use a variable named trace to get the instance and trace.Info() is used to log the request and whatever text we want to add along with that request.

Step 2: Registering Action Filter (LoggingFilterAttribute)

In order to register the created action filter to application's filters, just add a new instance of your action filter to config.Filters in `WebApiConfig` class.

```
using System.Web.Http;
using WebApi.ActionFilters;

namespace WebApi.App_Start
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.Filters.Add(new LoggingFilterAttribute());
```

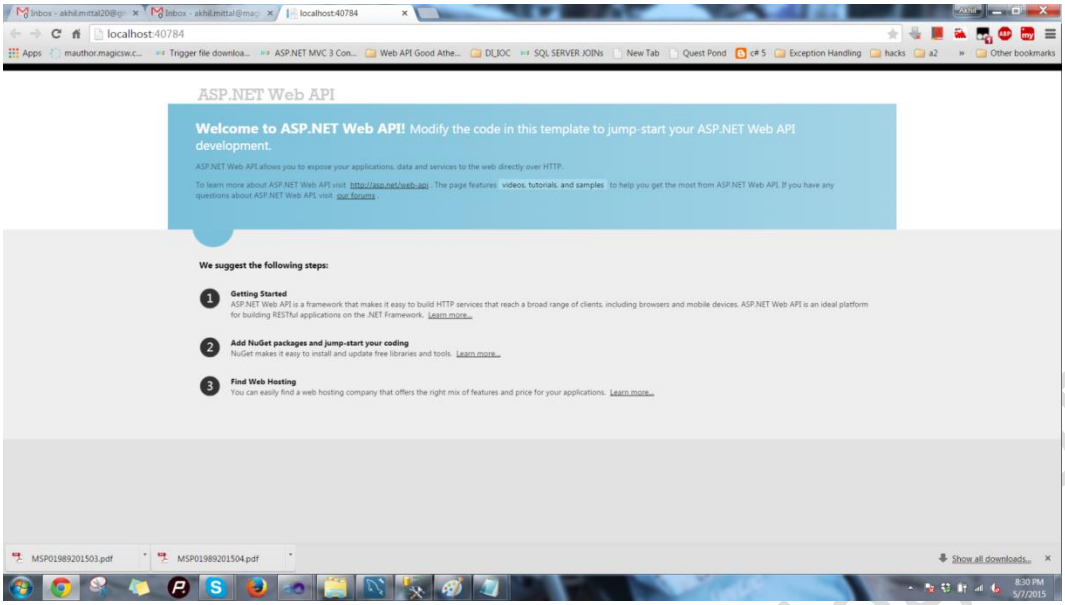
```
        }
    }
}
```

Now this action filter is applicable to all the controllers and actions in our project. You may not believe but request logging is done. It's time to run the application and validate our homework.

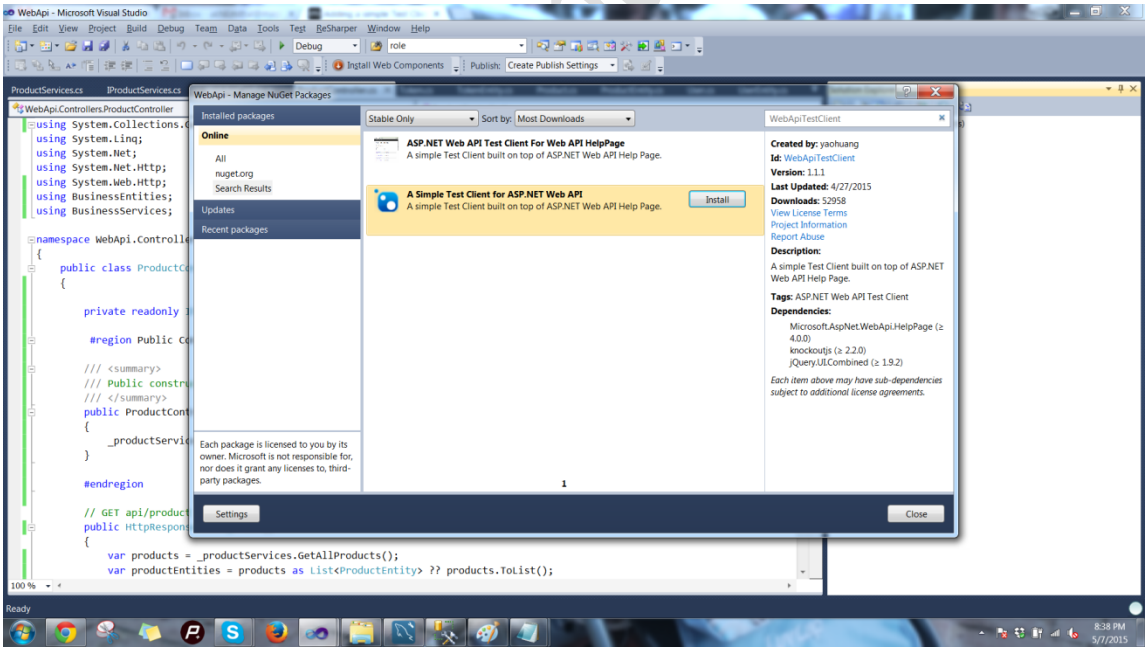
Image credit: https://pixabay.com/en/social-media-network-media-54536/

Running the application

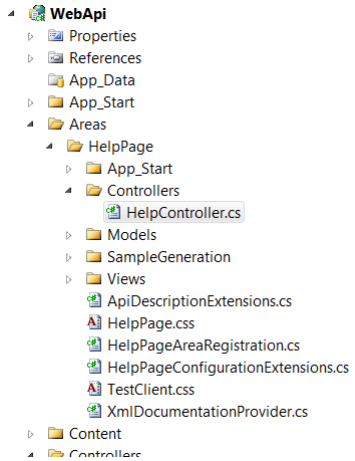
Let's run the application and try to make a call, using token based authorization, we have already covered authorization in day#5. You first need to authenticate your request using login service , and then that service will return a token for making calls to other services. Use that token to make calls to other services, for more details you can read day5 of this series.
Just run the application, we get,



We already have our test client added, but for new readers, just go to Manage Nuget Packages, by right clicking WebAPI project and type WebAPITestClient in searchbox in online packages,



You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,



I have already provided the database scripts and data in my previous article, you can use the same.

Append “/help” in the application url, and you’ll get the test client,

GET:

Product

API	Description
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.
GET v1/Products/Product/myproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/particularproduct	Documentation for 'Get'.
GET v1/Products/Product/particularproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/productid	Documentation for 'Get'.
GET v1/Products/Product/productid/{id}	Documentation for 'Get'.

POST:

POST v1/Products/Product/Create	Documentation for 'Post'.
POST v1/Products/Product/Register	Documentation for 'Post'.

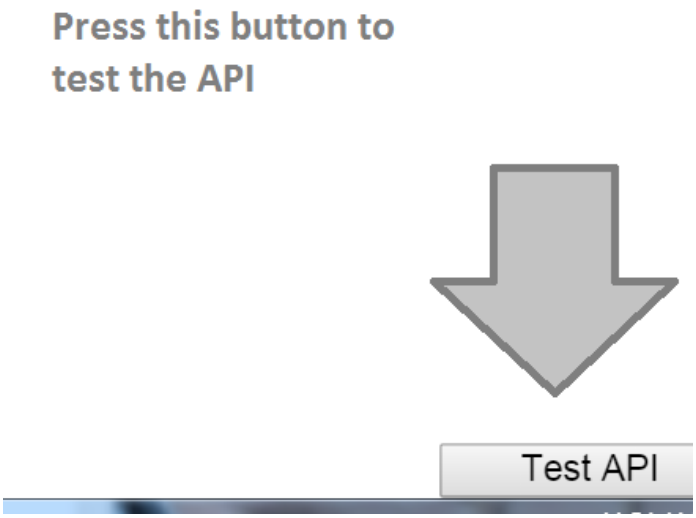
PUT:

PUT v1/Products/Product/Update/productid/{id}	Documentation for 'Put'.
PUT v1/Products/Product/Modify/productid/{id}	Documentation for 'Put'.

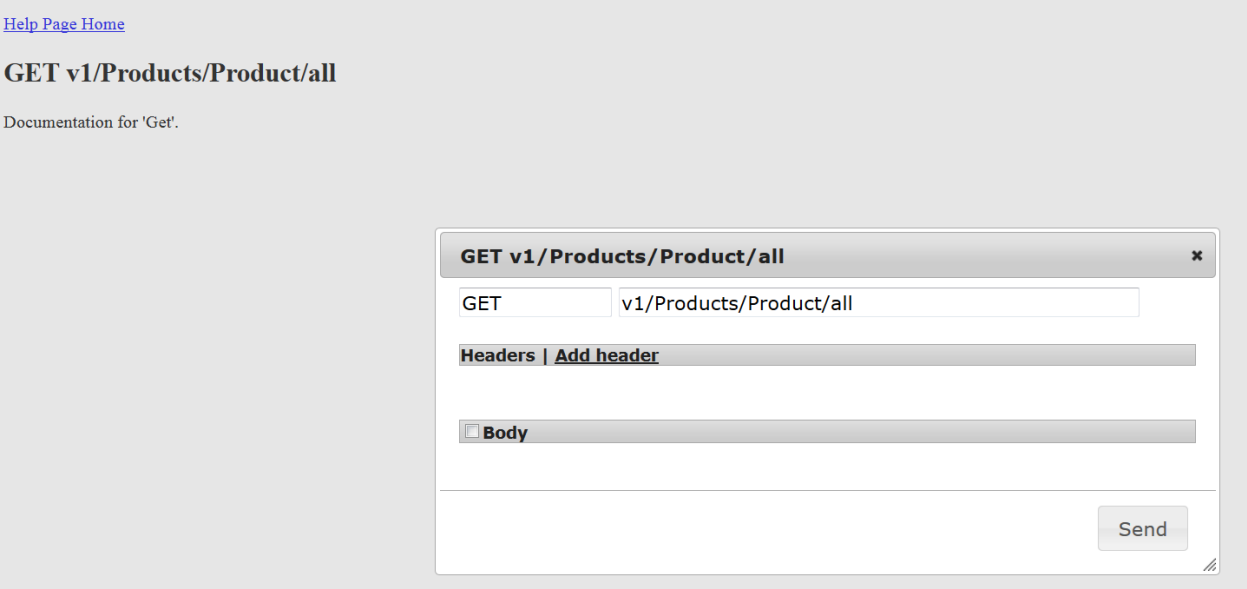
DELETE:

DELETE v1/Products/Product/delete/productid/{id}	Documentation for 'Delete'.
DELETE v1/Products/Product/remove/productid/{id}	Documentation for 'Delete'.
DELETE v1/Products/Product/clear/productid/{id}	Documentation for 'Delete'.

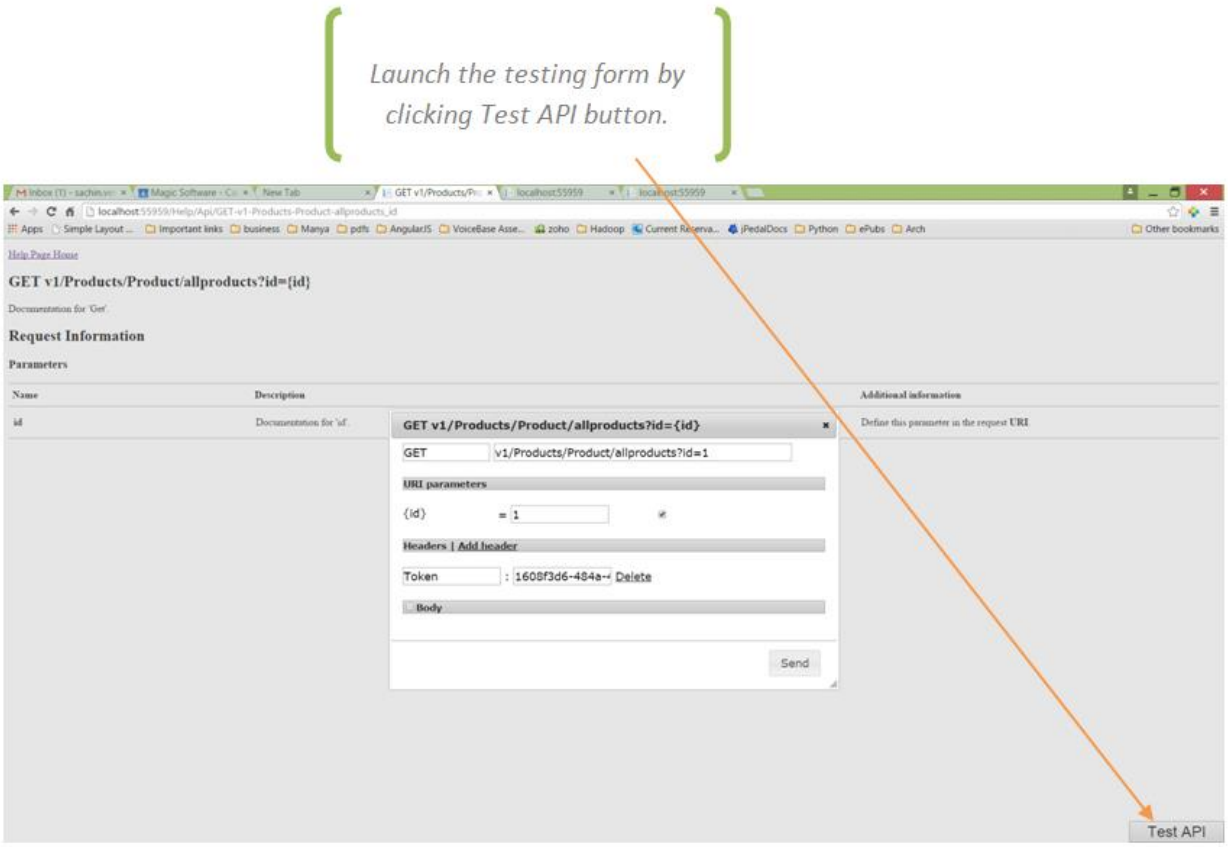
You can test each service by clicking on it. Once you click on the service link, you'll be redirected to test the service page of that particular service.On that page there is a button Test API in the right bottom corner, just press that button to test your service,



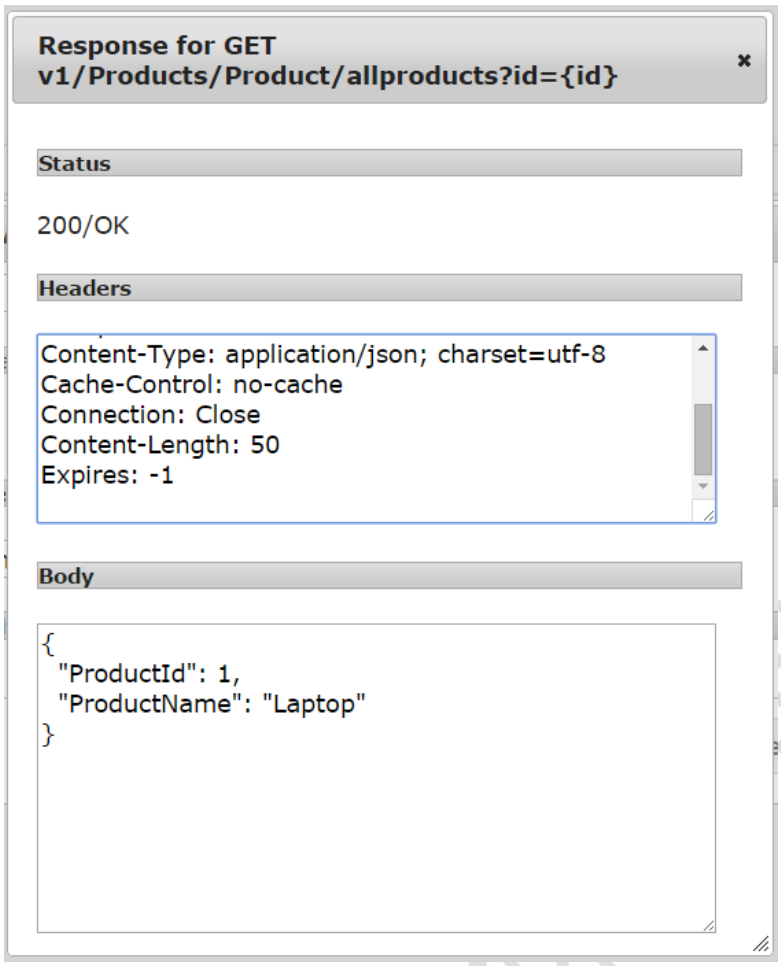
Service for Get All products,



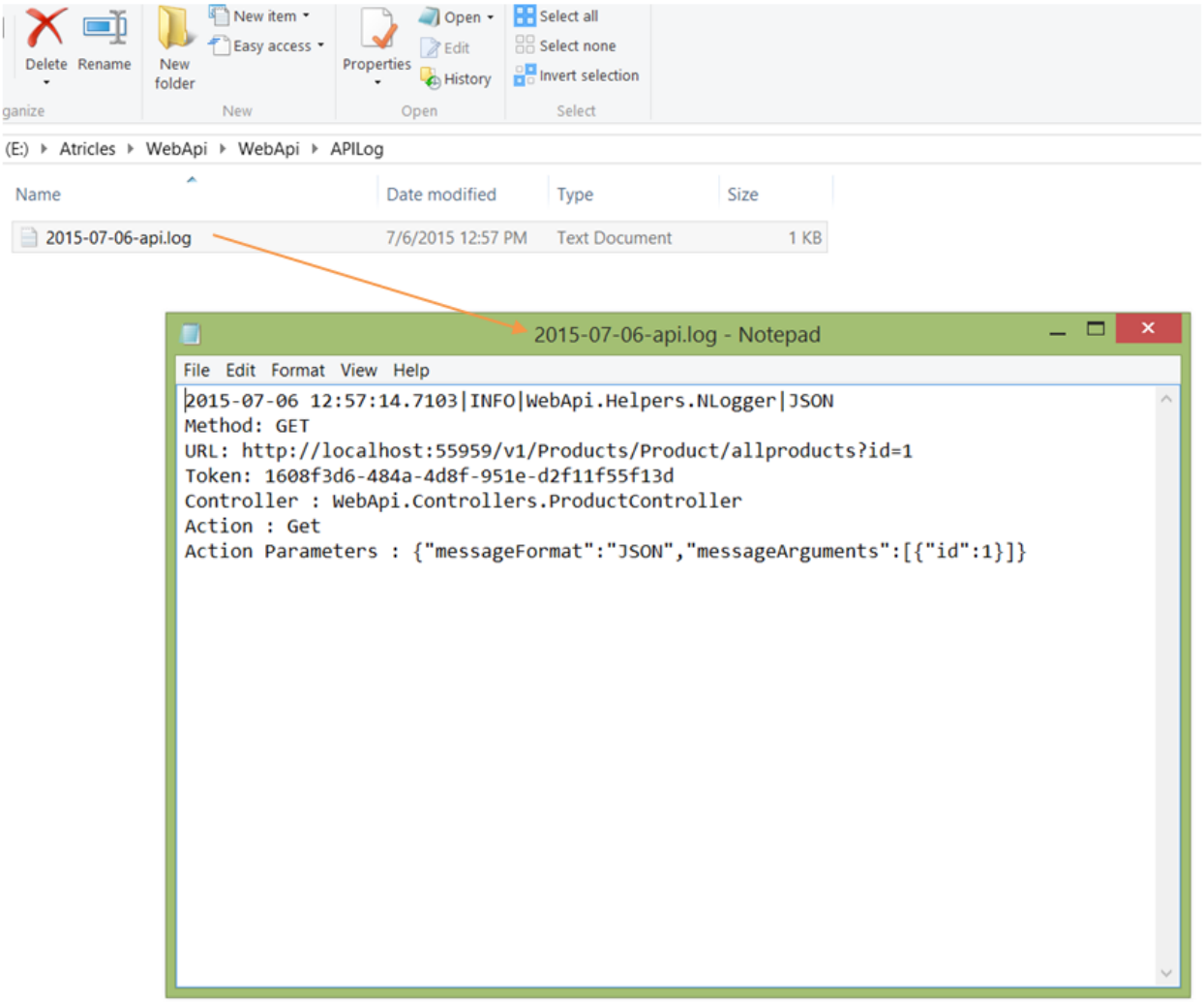
In below case, I have already generated the token and now using it to make call to fetch all the products from products table in database.



Here I have called allproducts API, Add the value for parameter Id and “Token” header with its current value and click to get the result -



Now let's see what happens to our APILog folder in application. Here you find the API log has been created, with the same name we have configured in NLog configuration in web.config file. The log file contains all the supplied details like Timestamp, Method type, URL , Header information (Token), Controller name, action and action parameters. You can also add more details to this log which you deem important for your application.



Logging Done!

Exception Logging

Our logging setup is completed, now we'll focus on centralizing exception logging as well, so that none of the exception escapes without logging itself. Logging exception is of very high importance, it keeps track of all the exceptions. No matter business or application or system exceptions , but all of them have to be logged.

Implementing Exception logging

Step 1: Exception Filter Attribute

Adding an action filter in our application for logging the exceptions, for this create a class GlobalExceptionAttribute to "ActionFilter" folder and add the code below, the class is derived from ExceptionFilterAttribute, which is under System.Web.Http.Filters.

I have override OnException() method, and replaced the default “ITraceWriter” service with our NLogger class instance in the controller’s service container, same as we have done in Action logging in above section. Now `GetTraceWriter()` method will return our instance (instance NLogger class) and Info() will call trace() method of NLogger class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http.Filters;
using System.Web.Http;
using System.Web.Http.Tracing;
using WebApi.Helpers;
using System.ComponentModel.DataAnnotations;
using System.Net.Http;
using System.Net;

namespace WebApi.ActionFilters
{
    /// <summary>
    /// Action filter to handle for Global application errors.
    /// </summary>
    public class GlobalExceptionAttribute : ExceptionFilterAttribute
    {
        public override void OnException(HttpActionExecutedContext
context)
        {

GlobalConfiguration.Configuration.Services.Replace(typeof(ITraceWriter
), new NLogger());
            var trace =
GlobalConfiguration.Configuration.Services.GetTraceWriter();
            trace.Error(context.Request, "Controller : " +
context.ActionContext.ControllerContext.ControllerDescriptor.Controlle
rType.FullName + Environment.NewLine + "Action : " +
context.ActionContext.ActionDescriptor.ActionName, context.Exception);

            var exceptionType = context.Exception.GetType();

            if (exceptionType == typeof(ValidationException))
            {
                var resp = new
HttpResponseMessage(HttpStatusCode.BadRequest) { Content = new
StringContent(context.Exception.Message), ReasonPhrase =
"ValidationException", };
```

```
                throw new HttpResponseException(resp);

            }
            else if (exceptionType ==
typeof(UnauthorizedAccessException))
            {
                throw new
HttpResponseException(context.Request.CreateResponse(HttpStatusCode.Un
authorized));
            }
            else
            {
                throw new
HttpResponseException(context.Request.CreateResponse(HttpStatusCode.In
ternalServerError));
            }
        }
    }
}
```

Step 2: Modify NLogger Class

Our NLogger class is capable to log all info and events, I have done some changes in private method Log() to handle the exceptions –

```
#region Private member methods.
/// <summary>
/// Logs info/Error to Log file
/// </summary>
/// <param name="record"></param>
private void Log(TraceRecord record)
{
     var message = new StringBuilder();

     if (!string.IsNullOrWhiteSpace(record.Message))
                message.Append("").Append(record.Message +
Environment.NewLine);

      if (record.Request != null)
     {
           if (record.Request.Method != null)
                message.Append("Method: " + record.Request.Method +
Environment.NewLine);
```

```
                if (record.Request.RequestUri != null)
                    message.Append("").Append("URL: " +
record.Request.RequestUri + Environment.NewLine);

                if (record.Request.Headers != null &&
record.Request.Headers.Contains("Token") &&
record.Request.Headers.GetValues("Token") != null &&
record.Request.Headers.GetValues("Token").FirstOrDefault() != null)
                    message.Append("").Append("Token: " +
record.Request.Headers.GetValues("Token").FirstOrDefault() +
Environment.NewLine);
            }

            if (!string.IsNullOrWhiteSpace(record.Category))
                message.Append("").Append(record.Category);

            if (!string.IsNullOrWhiteSpace(record.Operator))
                message.Append(" ").Append(record.Operator).Append("
").Append(record.Operation);

            if (record.Exception != null &&
!string.IsNullOrWhiteSpace(record.Exception.GetBaseException().Message
))
            {
                var exceptionType = record.Exception.GetType();
                message.Append(Environment.NewLine);
                message.Append("").Append("Error: " +
record.Exception.GetBaseException().Message + Environment.NewLine);
            }

            Logger[record.Level](Convert.ToString(message) +
Environment.NewLine);
        }
```

Step 3: Modify Controller for Exceptions

Our application is now ready to run, but there is no exception in our code, so I added a throw exception code in ProductController, just the Get(int id) method so that it can throw exception for testing our exception logging mechanism, It will throw an exception if the product is not there in database with the provided id.

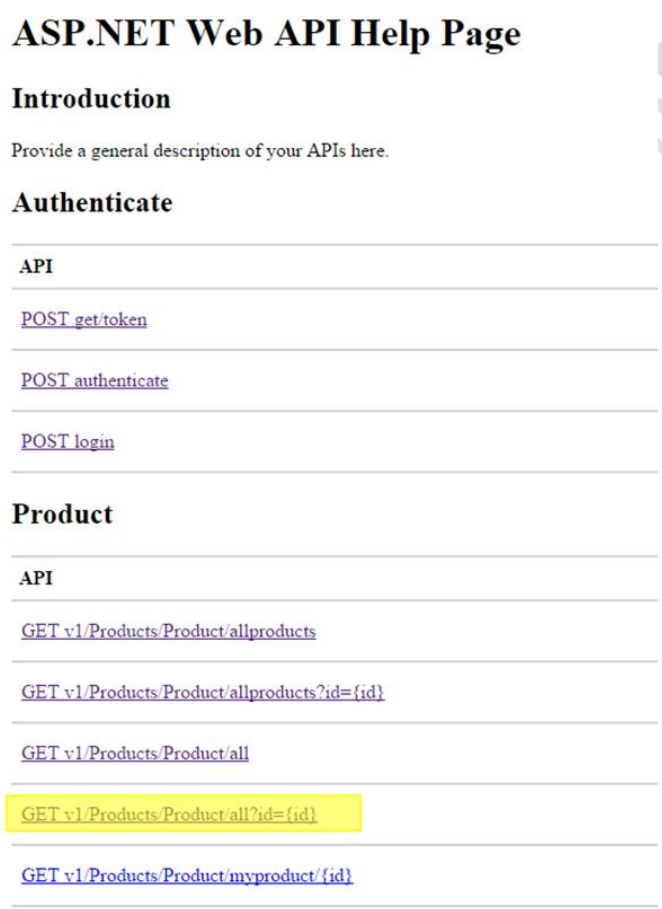
```
  // GET api/product/5
```

```
 [GET("productid/{id?}")]
 [GET("particularproduct/{id?}")]
 [GET("myproduct/{id:range(1, 3)}")]
 public HttpResponseMessage Get(int id)
 {
      var product = _productServices.GetProductById(id);
      if (product != null)
            return Request.CreateResponse(HttpStatusCode.OK, product);

       throw new Exception("No product found for this id");
      //return Request.CreateErrorResponse(HttpStatusCode.NotFound,   "No
product found for this id");
 }
```

Step 4: Run the application

Run the application and click on Product/all API



ASP.NET Web API Help Page

Introduction

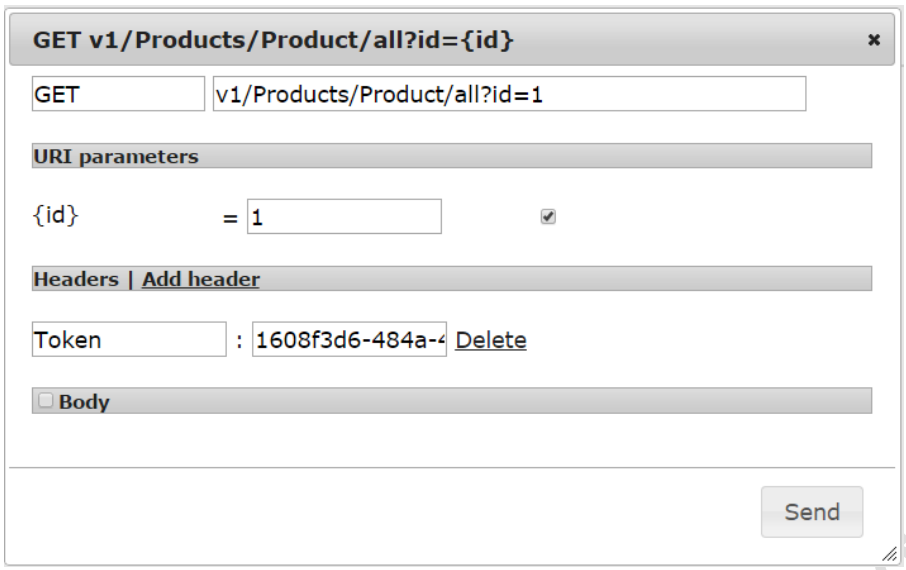
Provide a general description of your APIs here.

Authenticate

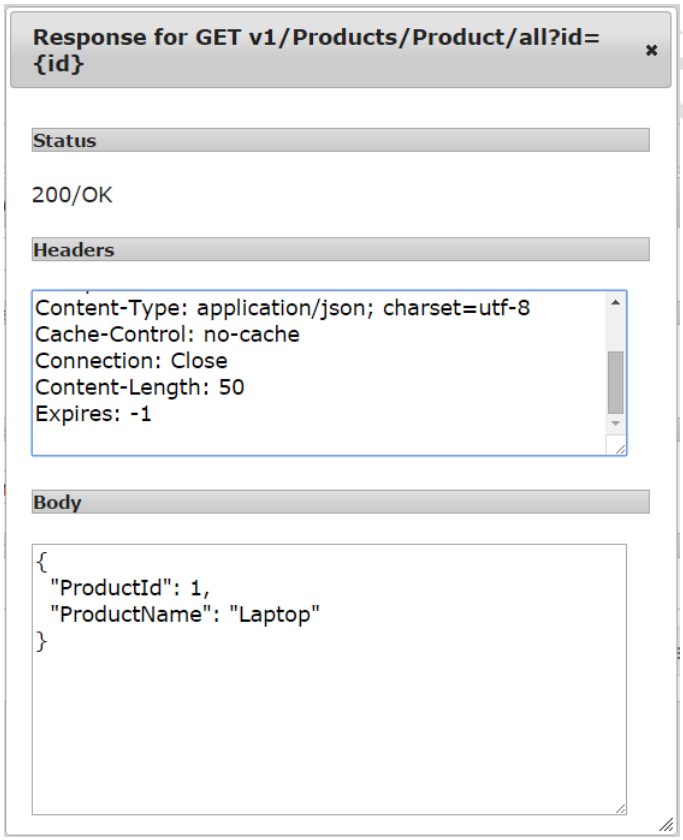
API
POST get/token
POST authenticate
POST login

Product

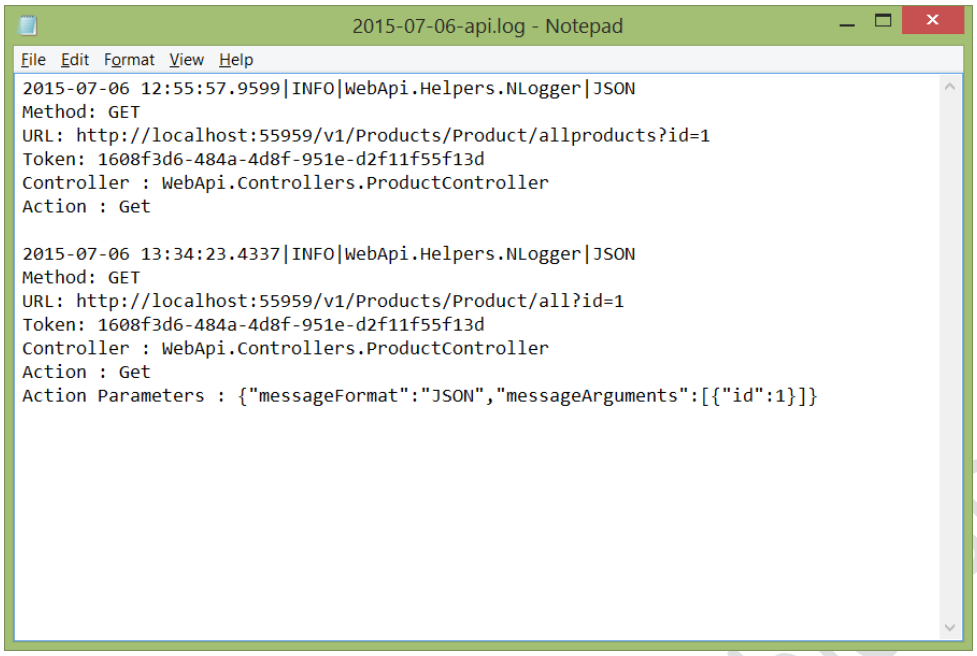
API
GET v1/Products/Product/allproducts
GET v1/Products/Product/allproducts?id={id}
GET v1/Products/Product/all
GET v1/Products/Product/all?id={id}
GET v1/Products/Product/myproduct/{id}



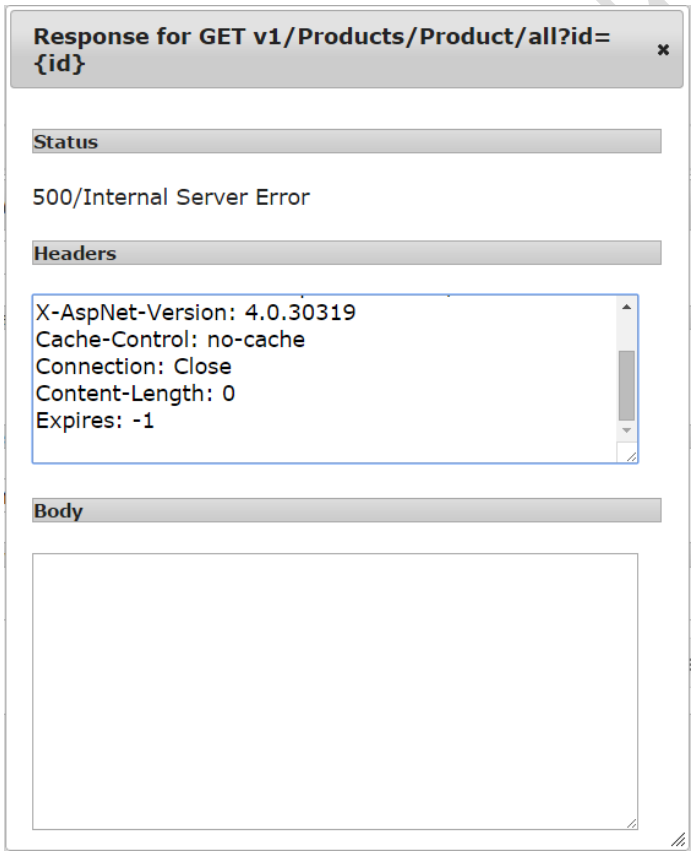
Add the parameter id value to 1 and header Token with it's current value, click on send button to get the result –



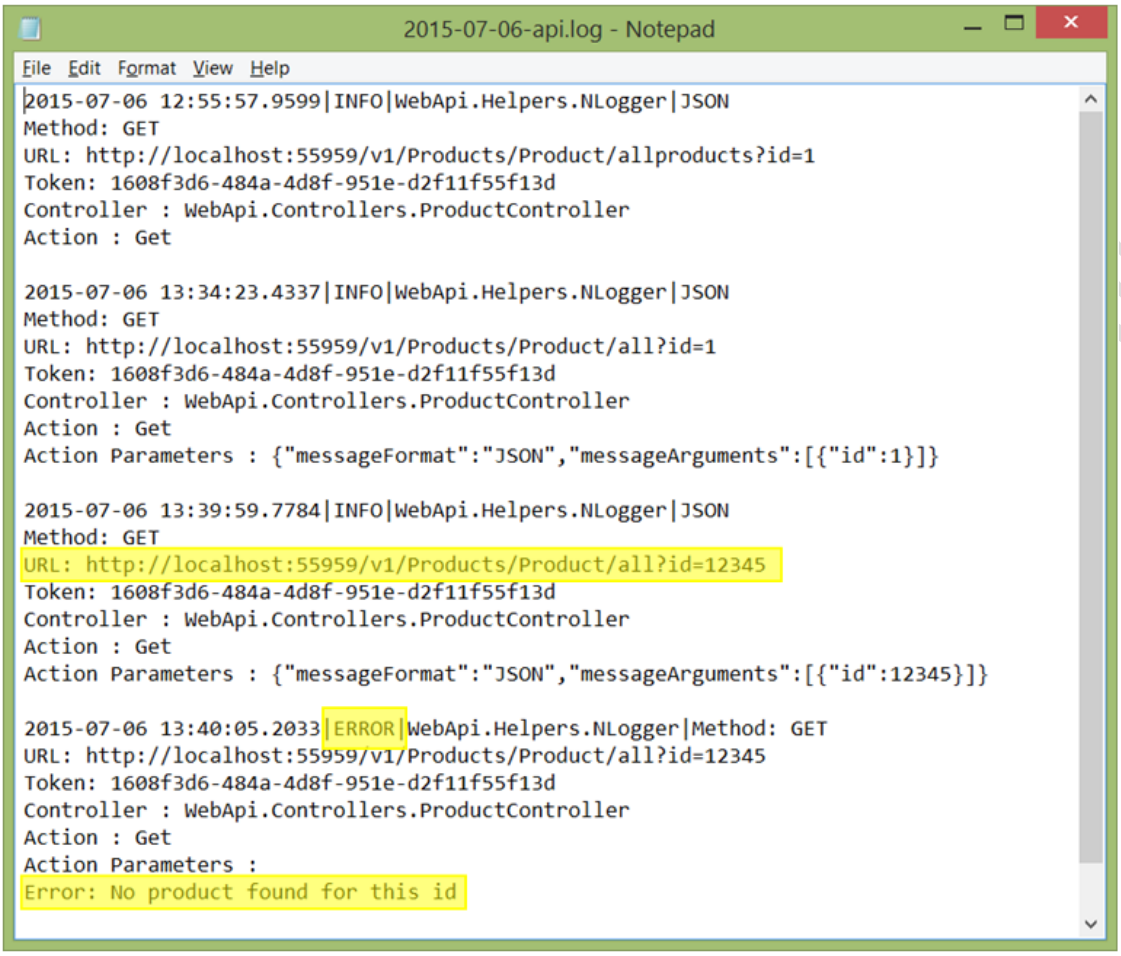
Now we can see that the Status is 200/OK, and we also get a product with the provided id in the response body. Let's see the API log now –



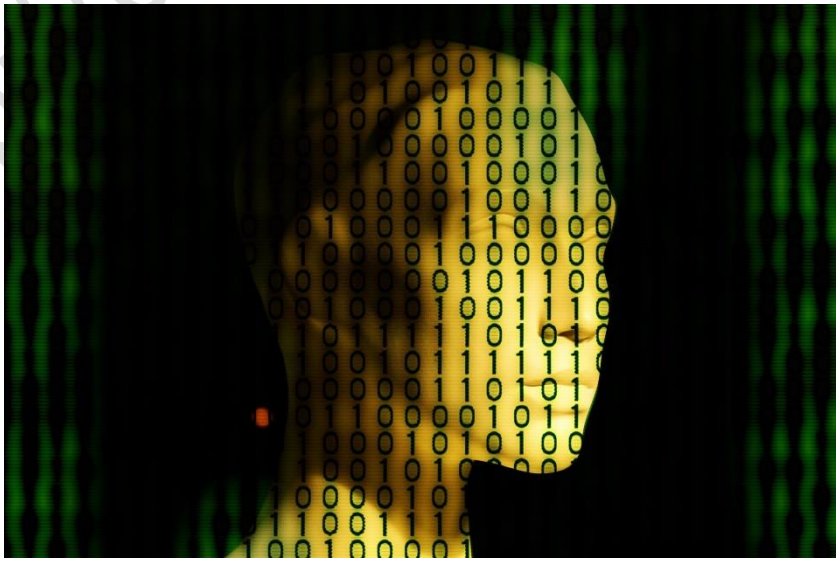
The log has captured the call of Product API, now provide a new product id as parameter, which is not there in database, I am using 12345 as product id and result is –



We can see there is an 500/Internal Server Error now in response status, lets check the API Log-



Well, now the log has captured both the event and error of same call on the server, you can see call log details and the error with provided error message in the log.

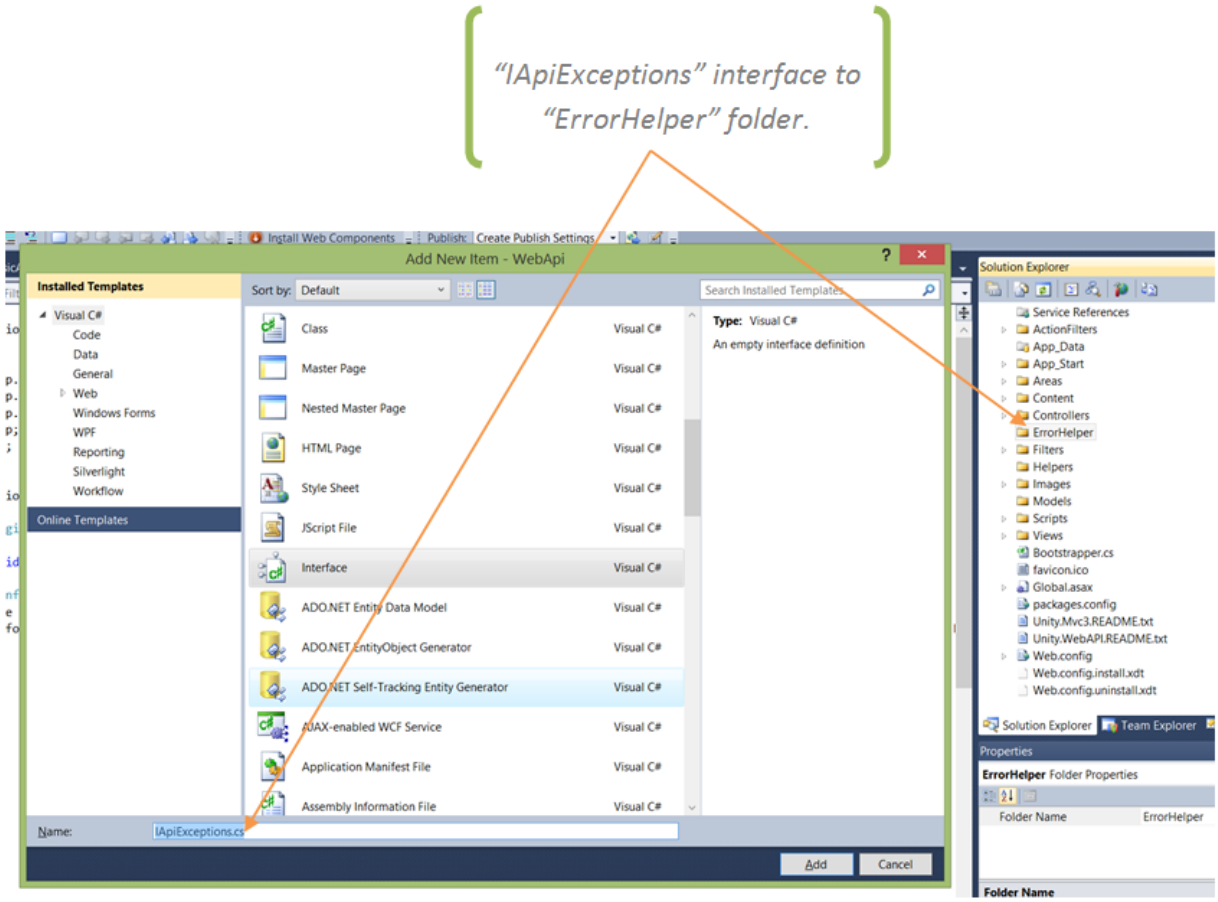


Custom Exception logging

In the above section we have implemented exception logging, but there is default system response and status (i. e. 500/Internal Server Error) , It will be always good to have your own custom response and exceptions for your API. That will be easier for client to consume and understand the API responses.

Step 1: Add Custom Exception Classes

Add “Error Helper” folder to application to maintain our custom exception classes separately and add “IApiExceptions” interface to newly created “ErrorHelper” folder -



Add following code the `IApiExceptions` interface, this will serve as a template for all exception classes, I have added four common properties for our custom classes to maintain

Error Code, ErrorDescription, `HttpStatus` (Contains the values of status codes defined for HTTP) and `ReasonPhrase`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net;

namespace WebApi.ErrorHelper
{
    /// <summary>
    /// IApiExceptions Interface
    /// </summary>
    public interface IApiExceptions
    {
        /// <summary>
        /// ErrorCode
        /// </summary>
        int ErrorCode { get; set; }
        /// <summary>
        /// ErrorDescription
        /// </summary>
        string ErrorDescription { get; set; }
        /// <summary>
        /// HttpStatus
        /// </summary>
        HttpStatusCode HttpStatus { get; set; }
        /// <summary>
        /// ReasonPhrase
        /// </summary>
        string ReasonPhrase { get; set; }
    }
}
```

Here, I divided our exceptions in three categories

1. API Exceptions – for API level exceptions.
2. Business Exceptions – for exceptions at business logic level.
3. Data Exceptions – Data related exceptions.

To implement this create a three new classes ApiException.cs, ApiDataException.cs and ApiBusinessException classes to same folder which implements IApiExceptions interface with following code to the classes –

```
#region Using namespaces.
using System;
using System.Net;
using System.Runtime.Serialization;
#endregion


namespace WebApi.ErrorHelper
{
    /// <summary>
    /// Api Exception
    /// </summary>
    [Serializable]
    [DataContract]
    public class ApiException : Exception, IApiExceptions
    {
        #region Public Serializable properties.
        [DataMember]
        public int ErrorCode { get; set; }
        [DataMember]
        public string ErrorDescription { get; set; }
        [DataMember]
        public HttpStatusCode HttpStatus { get; set; }

        string reasonPhrase = "ApiException";

        [DataMember]
        public string ReasonPhrase
        {
            get { return this.reasonPhrase; }

            set { this.reasonPhrase = value; }
        }
        #endregion
    }
}
```

I have initialized ReasonPhrase property with different default values in these classes to differentiate the implementation, you can use implement your custom classes as per your application needs.

The directives applied on class as Serializable and DataContract to make sure that the class defines or implements a data contract is serializable and can be serialize by a serializer.
Note – *Add reference of "System.Runtime.Serialization.dll" dll if you facing any assembly issue.*

In the same way add "ApiBusinessException" and "ApiDataException" classes into the same folder, with the following code –

```
#region Using namespaces.
using System;
using System.Net;
using System.Runtime.Serialization;
#endregion

namespace WebApi.ErrorHelper
{
    /// <summary>
    /// Api Business Exception
    /// </summary>
    [Serializable]
    [DataContract]
    public class ApiBusinessException : Exception, IApiExceptions
    {
        #region Public Serializable properties.
        [DataMember]
        public int ErrorCode { get; set; }
        [DataMember]
        public string ErrorDescription { get; set; }
        [DataMember]
        public HttpStatusCode HttpStatus { get; set; }

        string reasonPhrase = "ApiBusinessException";

        [DataMember]
        public string ReasonPhrase
        {
            get { return this.reasonPhrase; }

            set { this.reasonPhrase = value; }
        }
        #endregion

        #region Public Constructor.
        /// <summary>
        /// Public constructor for Api Business Exception
        /// </summary>
        /// <param name="errorCode"></param>
        /// <param name="errorDescription"></param>
        /// <param name="httpStatus"></param>
        public ApiBusinessException(int errorCode, string errorDescription,
HttpStatusCode httpStatus)
        {
            ErrorCode = errorCode;
            ErrorDescription = errorDescription;
            HttpStatus = httpStatus;
        }
```

```
        #endregion

    }
}

#region Using namespaces.
using System;
using System.Net;
using System.Runtime.Serialization;
#endregion

namespace WebApi.ErrorHelper
{
    /// <summary>
    /// Api Data Exception
    /// </summary>
    [Serializable]
    [DataContract]
    public class ApiDataException : Exception, IApiExceptions
    {
        #region Public Serializable properties.
        [DataMember]
        public int ErrorCode { get; set; }
        [DataMember]
        public string ErrorDescription { get; set; }
        [DataMember]
        public HttpStatusCode HttpStatus { get; set; }

        string reasonPhrase = "ApiDataException";

        [DataMember]
        public string ReasonPhrase
        {
            get { return this.reasonPhrase; }

            set { this.reasonPhrase = value; }
        }

        #endregion

        #region Public Constructor.
        /// <summary>
        /// Public constructor for Api Data Exception
        /// </summary>
        /// <param name="errorCode"></param>
        /// <param name="errorDescription"></param>
        /// <param name="httpStatus"></param>
        public ApiDataException(int errorCode, string errorDescription,
HttpStatusCode httpStatus)
        {
```

```
            ErrorCode = errorCode;
            ErrorDescription = errorDescription;
            HttpStatus = httpStatus;
        }
        #endregion
    }
}
```

JSon Serializers

There are some objects need to be serialized in json, to log and to transfer through the modules, for this I have add some extension methods to Object class –

For that add "*System.Web.Extensions.dll*" reference to project and add "JSONHelper" class to Helpers folder, with following code –

```
#region Using namespaces.
using System.Web.Script.Serialization;
using System.Data;
using System.Collections.Generic;
using System;

#endregion

namespace WebApi.Helpers
{
  public static class JSONHelper
  {
     #region Public extension methods.
    /// <summary>
    /// Extened method of object class, Converts an object to a json string.
    /// </summary>
    /// <param name="obj"></param>
    /// <returns></returns>
    public static string ToJSON(this object obj)
    {
      var serializer = new JavaScriptSerializer();
      try
      {
        return serializer.Serialize(obj);
      }
      catch(Exception ex)
      {
```

```
            return "";
        }
    }
    #endregion
  }
}
```

In the above code “ToJSON()” method is an extension of base Object class, which serializes supplied the object to a JSon string. The method using “JavaScriptSerializer” class which exist in “System.Web.Script.Serialization”.

Modify NLogger Class

For exception handling I have modified the Log() method of NLogger, which will now handle the different API exceptions.

```
/// <summary>
/// Logs info/Error to Log file
/// </summary>
/// <param name="record"></param>
private void Log(TraceRecord record)
{
      var message = new StringBuilder();

      if (!string.IsNullOrWhiteSpace(record.Message))
                message.Append("").Append(record.Message +
Environment.NewLine);

            if (record.Request != null)
            {
                if (record.Request.Method != null)
                    message.Append("Method: " + record.Request.Method +
Environment.NewLine);

                if (record.Request.RequestUri != null)
                    message.Append("").Append("URL: " +
record.Request.RequestUri + Environment.NewLine);

                if (record.Request.Headers != null &&
record.Request.Headers.Contains("Token") &&
record.Request.Headers.GetValues("Token") != null &&
record.Request.Headers.GetValues("Token").FirstOrDefault() != null)
                    message.Append("").Append("Token: " +
record.Request.Headers.GetValues("Token").FirstOrDefault() +
Environment.NewLine);
            }
```

```
            if (!string.IsNullOrWhiteSpace(record.Category))
                message.Append("").Append(record.Category);

            if (!string.IsNullOrWhiteSpace(record.Operator))
                message.Append(" ").Append(record.Operator).Append("
").Append(record.Operation);

            if (record.Exception != null &&
!string.IsNullOrWhiteSpace(record.Exception.GetBaseException().Message))
            {
                var exceptionType = record.Exception.GetType();
                message.Append(Environment.NewLine);
                if (exceptionType == typeof(ApiException))
                {
                    var exception = record.Exception as ApiException;
                    if (exception != null)
                    {
                        message.Append("").Append("Error: " +
exception.ErrorDescription + Environment.NewLine);
                        message.Append("").Append("Error Code: " +
exception.ErrorCode + Environment.NewLine);
                    }
                }
                else if (exceptionType == typeof(ApiBusinessException))
                {
                    var exception = record.Exception as ApiBusinessException;
                    if (exception != null)
                    {
                        message.Append("").Append("Error: " +
exception.ErrorDescription + Environment.NewLine);
                        message.Append("").Append("Error Code: " +
exception.ErrorCode + Environment.NewLine);
                    }
                }
                else if (exceptionType == typeof(ApiDataException))
                {
                    var exception = record.Exception as ApiDataException;
                    if (exception != null)
                    {
                        message.Append("").Append("Error: " +
exception.ErrorDescription + Environment.NewLine);
                        message.Append("").Append("Error Code: " +
exception.ErrorCode + Environment.NewLine);
                    }
                }
                else
                    message.Append("").Append("Error: " +
record.Exception.GetBaseException().Message + Environment.NewLine);
            }
```

```
            Logger[record.Level](Convert.ToString(message) +
Environment.NewLine);
        }
```

The code above checks the exception object of TraceRecord and updates the logger as per the exception type.

Modify GlobalExceptionAttribute

As we have created GlobalExceptionAttribute to handle all exceptions and create response in case of any exception. Now I have added some new code to this in order to enable the GlobalExceptionAttribute class to handle custom exceptions. I am adding only modified method here for your reference .

```
public override void OnException(HttpActionExecutedContext context)
{

GlobalConfiguration.Configuration.Services.Replace(typeof(ITraceWriter), new
NLogger());
            var trace =
GlobalConfiguration.Configuration.Services.GetTraceWriter();
            trace.Error(context.Request, "Controller : " +
context.ActionContext.ControllerContext.ControllerDescriptor.ControllerType.F
ullName + Environment.NewLine + "Action : " +
context.ActionContext.ActionDescriptor.ActionName, context.Exception);

            var exceptionType = context.Exception.GetType();

            if (exceptionType == typeof(ValidationException))
            {
                var resp = new HttpResponseMessage(HttpStatusCode.BadRequest)
{ Content = new StringContent(context.Exception.Message), ReasonPhrase =
"ValidationException", };
                throw new HttpResponseException(resp);

            }
            else if (exceptionType == typeof(UnauthorizedAccessException))
            {
                throw new
HttpResponseException(context.Request.CreateResponse(HttpStatusCode.Unauthori
zed, new ServiceStatus() { StatusCode = (int)HttpStatusCode.Unauthorized,
StatusMessage = "UnAuthorized", ReasonPhrase = "UnAuthorized Access" }));
            }
            else if (exceptionType == typeof(ApiException))
            {
```

```
                var webapiException = context.Exception as ApiException;
                if (webapiException != null)
                    throw new
HttpResponseException(context.Request.CreateResponse(webapiException.HttpStat
us, new ServiceStatus() { StatusCode = webapiException.ErrorCode,
StatusMessage = webapiException.ErrorDescription, ReasonPhrase =
webapiException.ReasonPhrase }));
            }
            else if (exceptionType == typeof(ApiBusinessException))
            {
                var businessException = context.Exception as
ApiBusinessException;
                if (businessException != null)
                    throw new
HttpResponseException(context.Request.CreateResponse(businessException.HttpSt
atus, new ServiceStatus() { StatusCode = businessException.ErrorCode,
StatusMessage = businessException.ErrorDescription, ReasonPhrase =
businessException.ReasonPhrase }));
            }
            else if (exceptionType == typeof(ApiDataException))
            {
                var dataException = context.Exception as ApiDataException;
                if (dataException != null)
                    throw new
HttpResponseException(context.Request.CreateResponse(dataException.HttpStatus
, new ServiceStatus() { StatusCode = dataException.ErrorCode, StatusMessage =
dataException.ErrorDescription, ReasonPhrase = dataException.ReasonPhrase
}));
            }
            else
            {
                throw new
HttpResponseException(context.Request.CreateResponse(HttpStatusCode.InternalS
erverError));
            }
        }
```

In the above code I have modified the overrided method OnExeption() and created new Http response exception based on the different exception types.

Modify Product Controller

Now modify the Product controller to throw our custom exception form, please look into the Get method I have modified to throw the APIDataException in case if data is not found and APIException in any other kind of error.

```
// GET api/product/5
[GET("productid/{id?}")]
[GET("particularproduct/{id?}")]
[GET("myproduct/{id:range(1, 3)}")]
public HttpResponseMessage Get(int id)
{
      if (id != null)
      {
            var product = _productServices.GetProductById(id);
            if (product != null)
                  return Request.CreateResponse(HttpStatusCode.OK, product);

            throw new ApiDataException(1001, "No product found for this id.",
      HttpStatusCode.NotFound);
      }
      throw new ApiException() { ErrorCode = (int)HttpStatusCode.BadRequest,
ErrorDescription = "Bad Request..." };
}
```

Run the application

Run the application and click on Product/all API -

ASP.NET Web API Help Page

Introduction

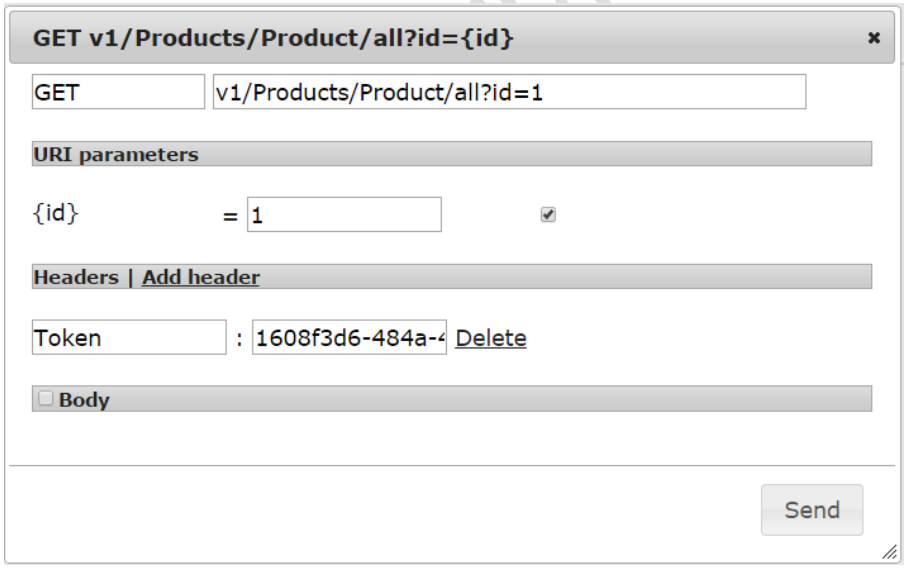
Provide a general description of your APIs here.

Authenticate

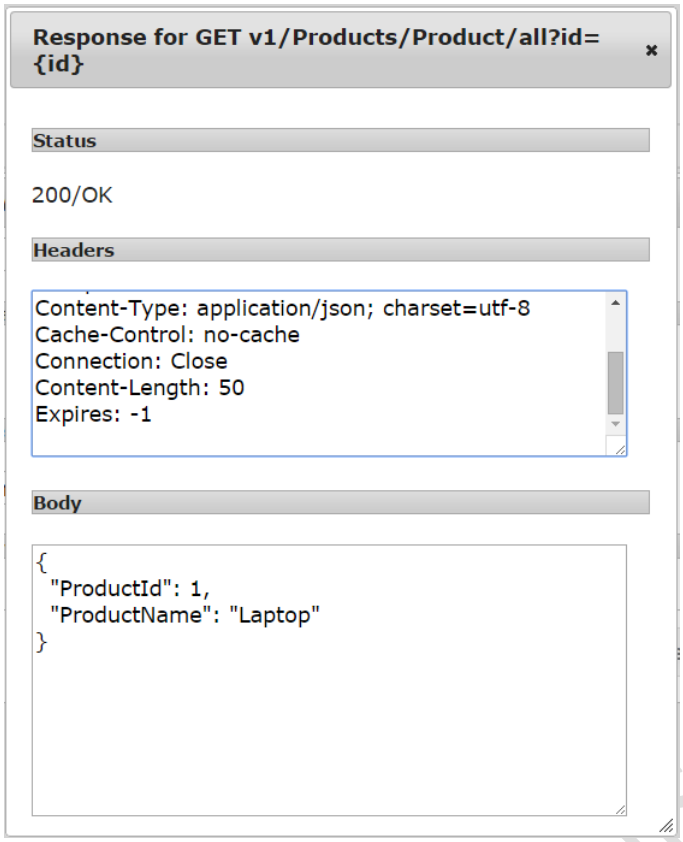
API
POST get/token
POST authenticate
POST login

Product

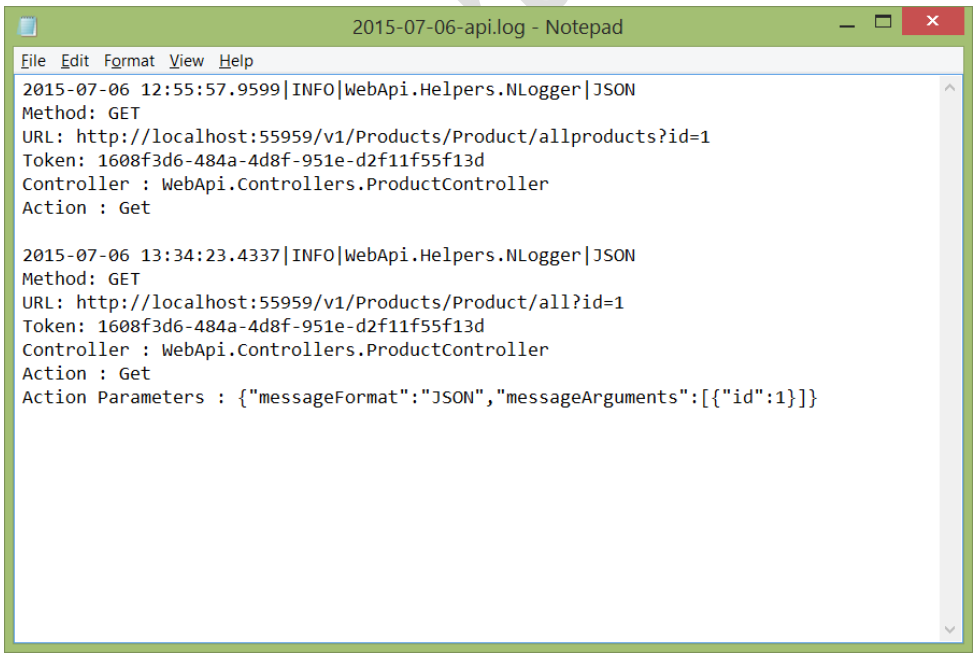
API
GET v1/Products/Product/allproducts
GET v1/Products/Product/allproducts?id={id}
GET v1/Products/Product/all
GET v1/Products/Product/all?id={id}
GET v1/Products/Product/myproduct/{id}



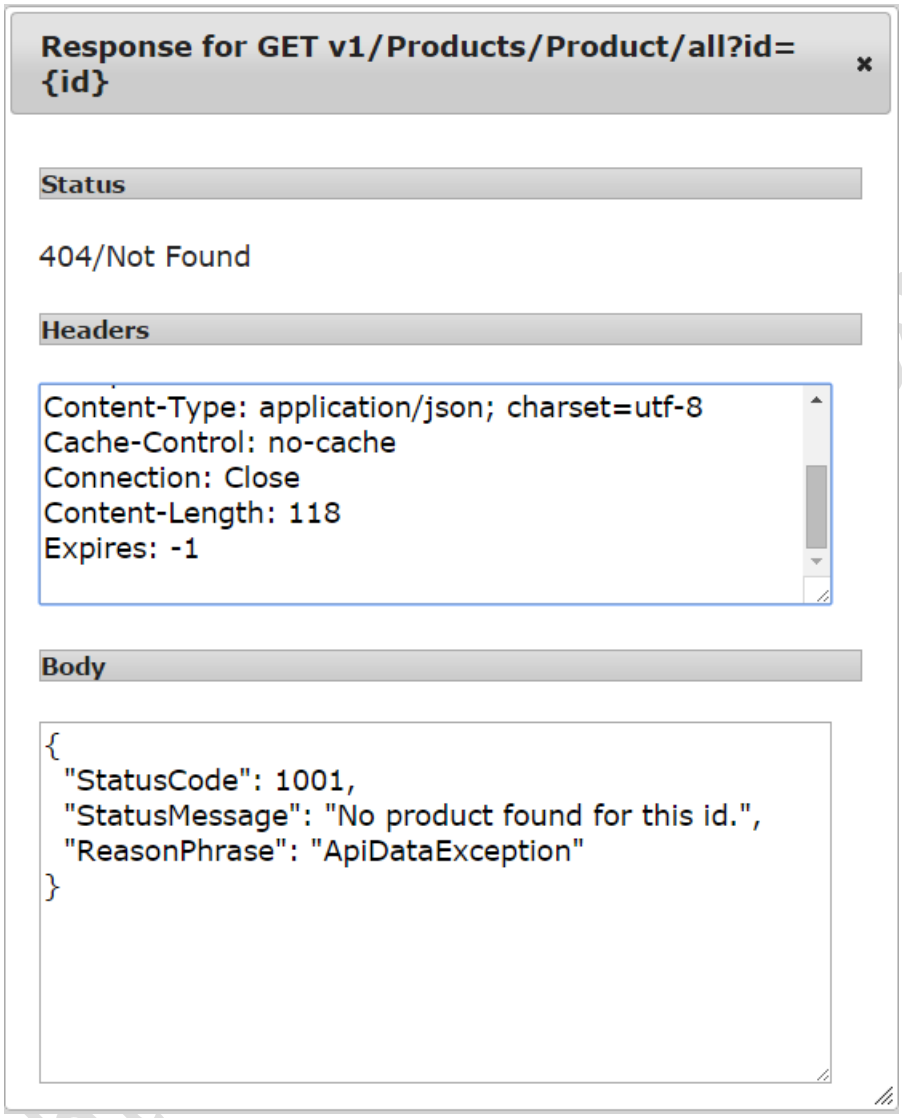
Add the parameter id value to 1 and header Token with its current value, click on send button to get the result –



Now we can see that the Status is 200/OK, and we also get a product with the provided id in the response body. Lets see the API log now –

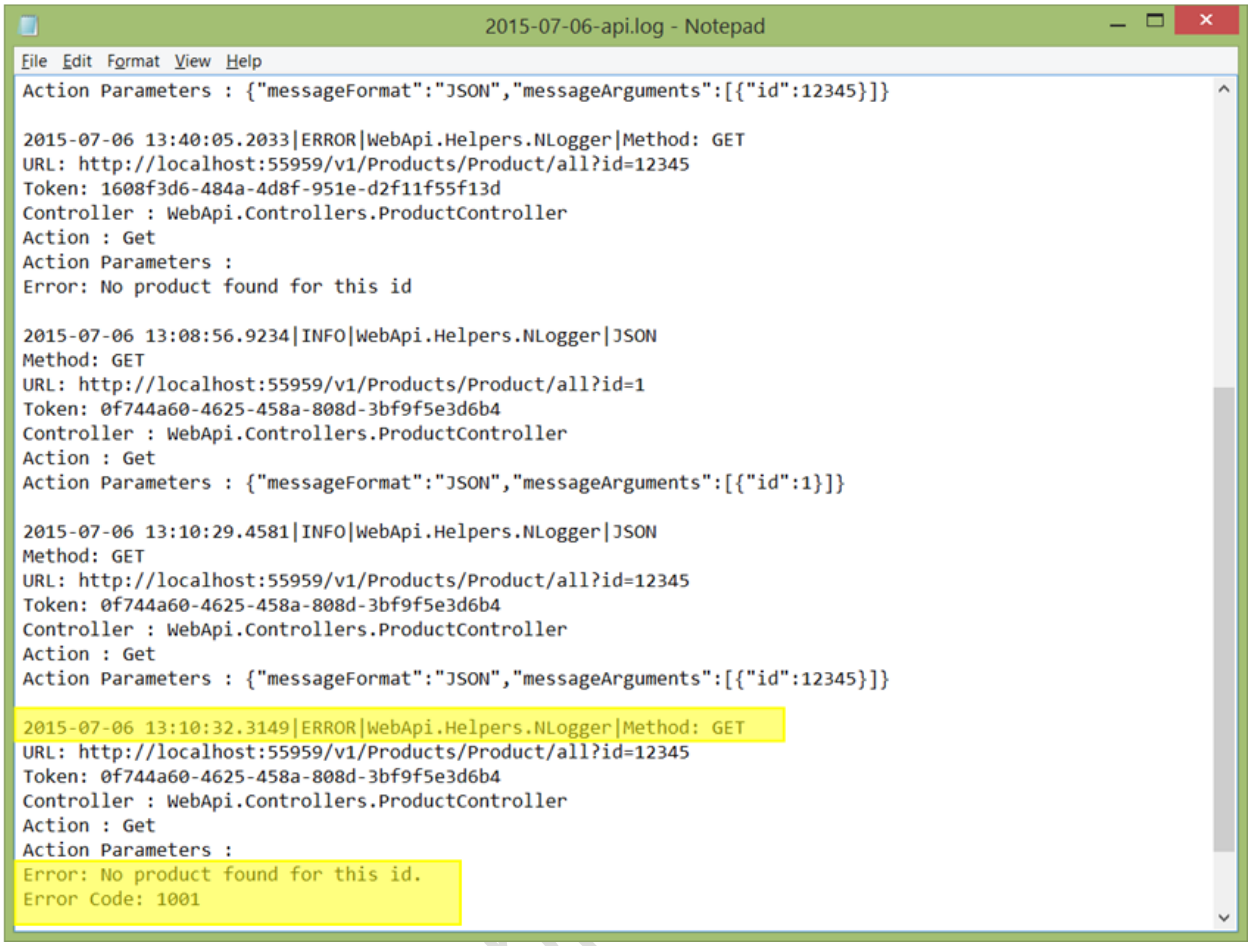


The log has captured the call of Product API, now provide a new product id as parameter, which is not there in database, I am using 12345 as product id and result is –



We can see, now there is a custom error status code “1001” and messages “No product found for this id.” And the generic status code “500/Internal Server Error” is now replaced with our supplied code “404/ Not Found”, which is more meaningful for the client or consumer.

Lets see the APILog now -



Well, now the log has captured both the event and error of same call on the server, you can see call log details and the error with provided error message in the log with our custom error code, I have only captured error description and error code, but you can add more details in the log as per your application needs.

Update the controller for new Exception Handling

Following is the code for controllers with implementation of custom exception handling and logging –

Product Controller

```
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using AttributeRouting;
using AttributeRouting.Web.Http;
using BusinessEntities;
using BusinessServices;
using WebApi.ActionFilters;
using WebApi.Filters;
using System;
using WebApi.ErrorHelper;

namespace WebApi.Controllers
{
    [AuthorizationRequired]
    [RoutePrefix("v1/Products/Product")]
    public class ProductController : ApiController
    {
        #region Private variable.

        private readonly IProductServices _productServices;

        #endregion

        #region Public Constructor

        /// <summary>
        /// Public constructor to initialize product service instance
        /// </summary>
        public ProductController(IProductServices productServices)
        {
            _productServices = productServices;
        }

        #endregion

        // GET api/product
        [GET("allproducts")]
        [GET("all")]
        public HttpResponseMessage Get()
        {
```

```
            var products = _productServices.GetAllProducts();
            var productEntities = products as List<ProductEntity> ??
products.ToList();
            if (productEntities.Any())
                return Request.CreateResponse(HttpStatusCode.OK,
productEntities);
            throw new ApiDataException(1000, "Products not found",
HttpStatusCode.NotFound);
        }

        // GET api/product/5
        [GET("productid/{id?}")]
        [GET("particularproduct/{id?}")]
        [GET("myproduct/{id:range(1, 3)}")]
        public HttpResponseMessage Get(int id)
        {
            if (id != null)
            {
                var product = _productServices.GetProductById(id);
                if (product != null)
                    return Request.CreateResponse(HttpStatusCode.OK,
product);

                throw new ApiDataException(1001, "No product found for this
id.", HttpStatusCode.NotFound);
            }
            throw new ApiException() { ErrorCode =
(int)HttpStatusCode.BadRequest, ErrorDescription = "Bad Request..." };
        }

        // POST api/product
        [POST("Create")]
        [POST("Register")]
        public int Post([FromBody] ProductEntity productEntity)
        {
            return _productServices.CreateProduct(productEntity);
        }

        // PUT api/product/5
        [PUT("Update/productid/{id}")]
        [PUT("Modify/productid/{id}")]
        public bool Put(int id, [FromBody] ProductEntity productEntity)
        {
            if (id > 0)
            {
                return _productServices.UpdateProduct(id, productEntity);
            }
            return false;
        }
```

```
        // DELETE api/product/5
        [DELETE("remove/productid/{id}")]
        [DELETE("clear/productid/{id}")]
        [PUT("delete/productid/{id}")]
        public bool Delete(int id)
        {
            if (id != null && id > 0)
            {
                var isSuccess = _productServices.DeleteProduct(id);
                if (isSuccess)
                {
                    return isSuccess;
                }
                throw new ApiDataException(1002, "Product is already deleted
or not exist in system.", HttpStatusCode.NoContent );
            }
            throw new ApiException() {ErrorCode = (int)
HttpStatusCode.BadRequest, ErrorDescription = "Bad Request..."};
        }
    }
}
```

Now you can see, our application is so rich and scalable that none of the exception or transaction can escape logging.Once setup is inplaced, now you don't have to worry about writing code each time for logging or requests and exceptions, but you can relax and focus on business logic only.

Image credit: https://pixabay.com/en/kermit-frog-meadow-daisy-concerns-383370/

Conclusion

In this article we learnt about how to perform request logging and exception logging in WebPI. There could be numerous ways in which you can perform these operations but I tried to present this in as simple way as possible. My approach was to take our enterprise level to next level of development, where developers should not always be worried about exception handling and logging. Our solution provides a generic approach of centralizing the operations in one place; all the requests and exceptions are automatically taken care of. In my new articles, I'll try to enhance the application by explaining unit testing in WebAPI and OData in WebAPI. You can download the complete source code of this article with packages from **GitHub**. Happy coding ☺

Other Series

My other series of articles:

- Introduction to MVC Architecture and Separation of Concerns: Part 1
- Diving Into OOP (Day 1): Polymorphism and Inheritance (Early Binding/Compile Time Polymorphism)

For more informative articles visit my Blog.