CODE PROJECT® For those who code

Web API Thoughts 1 of 3 - Data Streaming

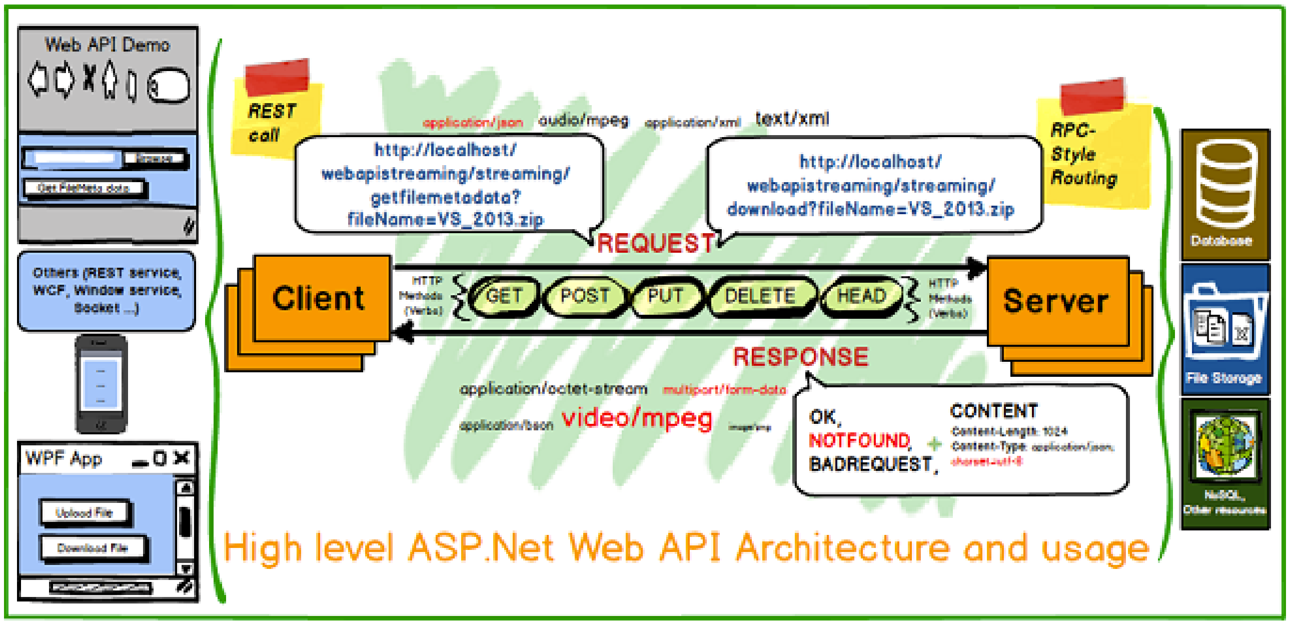
Wonde Tadesse, 18 Oct 2015

★★★★★ 4.89 (102 votes)

ASP.NET Web API related projects. It touches most parts of the technology through the articles on Data Streaming, Working with HTTPS and Extending Web API Documentation.

Download source code Web API Thoughts

GitHub repository link available in History section



BackGround

There are numerous articles written about ASP.NET Web API technology. A lot can be found in CodeProject (ASP.NET Web API), the ASP.NET Web API official web site, as well as additional places. The following series of articles will try to explain the technology by applying on Data Streaming, HTTPS and extending Web API documentation. The articles assume you have basic knowledge of **C#, ASP.NET, OOP, ASP.NET MVC, REST service, IIS and some technical terms of .NET framework**. With that being said, the articles will be presented with the following order.

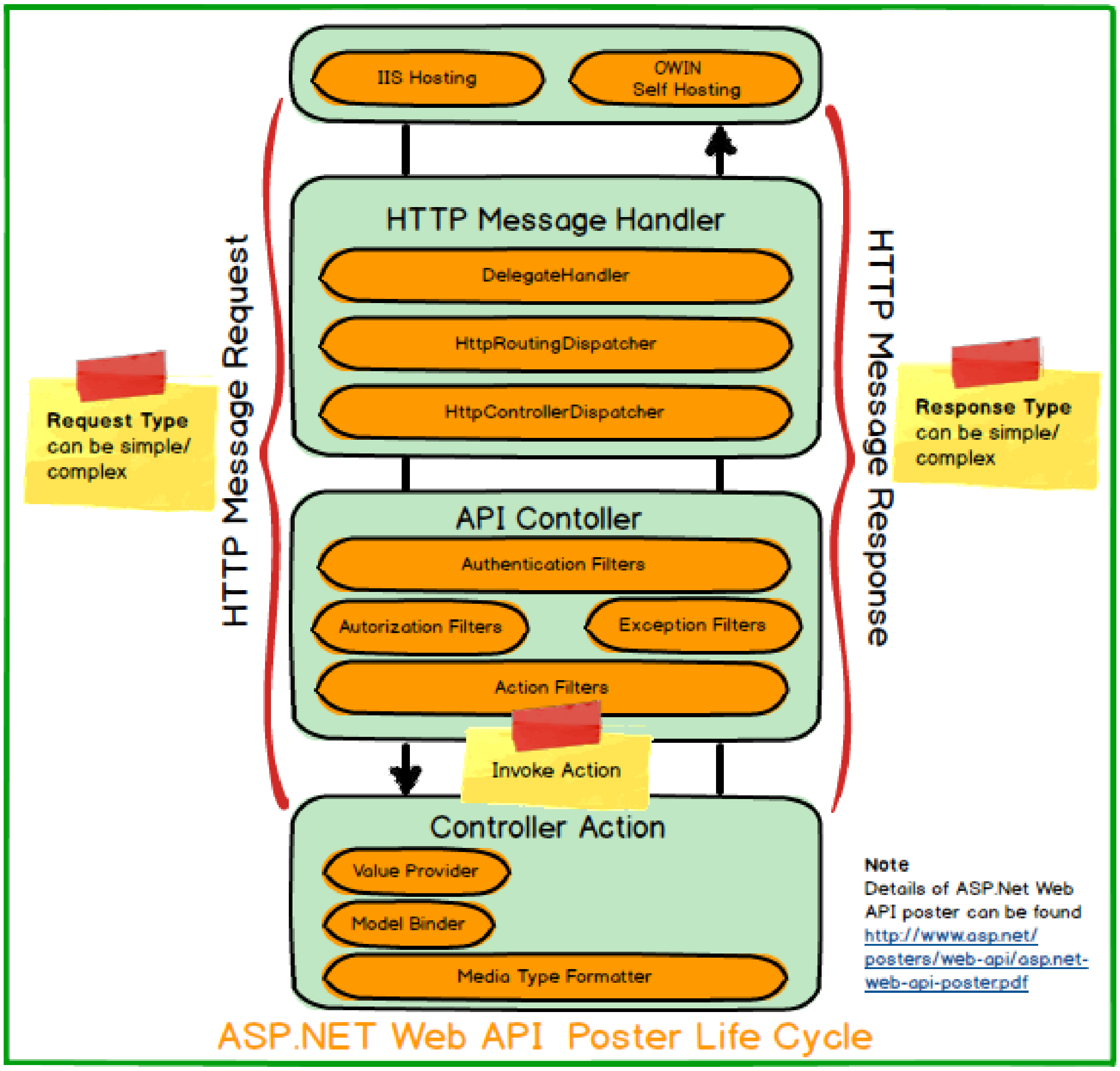
Requirements and Dependencies

- The solution is best viewed by VS 2012(SP4) Ultimate and above. But it's also possible to view each individual project using VS Professional or Express 2012 and above.
- .NET framework 4.5.1 and above
- A lot of Nuget packages. See each project Nuget **packages.config** file

Things that will be covered in the articles

- Related to Asp.NET Web API Technology
    1. ActionFilter
    2. AuthorizationFilter
    3. DelegateHandler
    4. Different Web API routing attributes
    5. MediaTypeFormatter
    6. OWIN
    7. Self Hosting
    8. Web API documentation and its extension
- Related to .NET framework and other
    1. Async/Await
    2. .NET reflection
    3. Serialization
    4. ASP.NET Web API/MVC Error handling
    5. IIS ,HTTPS and Certificate
    6. Design principles and techniques

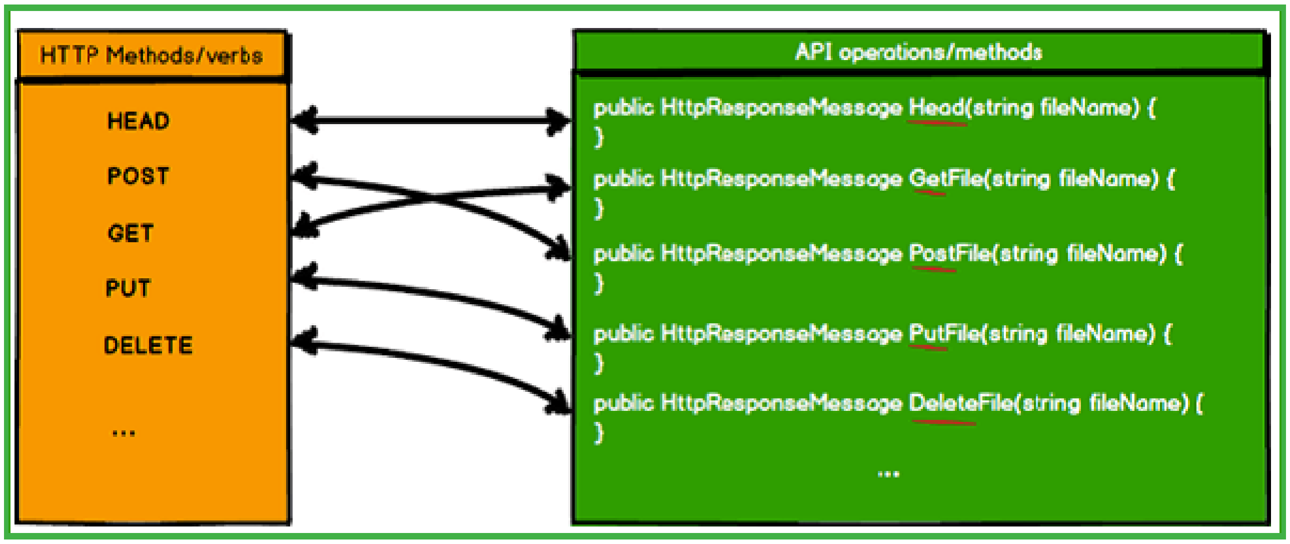
Introduction



It’s been years since ASP.NET Web API was added to the .NET framework. ASP.NET Web API is built on top of HTTP protocol that exposes RESTful services and data to external systems. The target is to reach services to various platforms using HTTP technology, which is supported by many end user applications, browsers, mobile devices, other services. ASP.NET Web API is a request-response message exchange pattern, in which a client can request certain information from a server and a server responses the request to the client. The response can be expected synchronously or asynchronously. When we think about a Web API usually several things pop up in your mind. Personally I'll point out these three basic key points regardless of the Web API implementation.

- The purpose of the service and its methods.
- Each method input(s) i.e. Request
- Each method output i.e. Response.

By convention, ASP.NET Web API lets you define method with a matching semantic to HTTP methods(verbs). For example, if we have a method name `GetPysicians()`, then its matching name to the HTTP method will be `GET`. In short, the following diagram below shows the matching of each method to the HTTP methods (verbs).

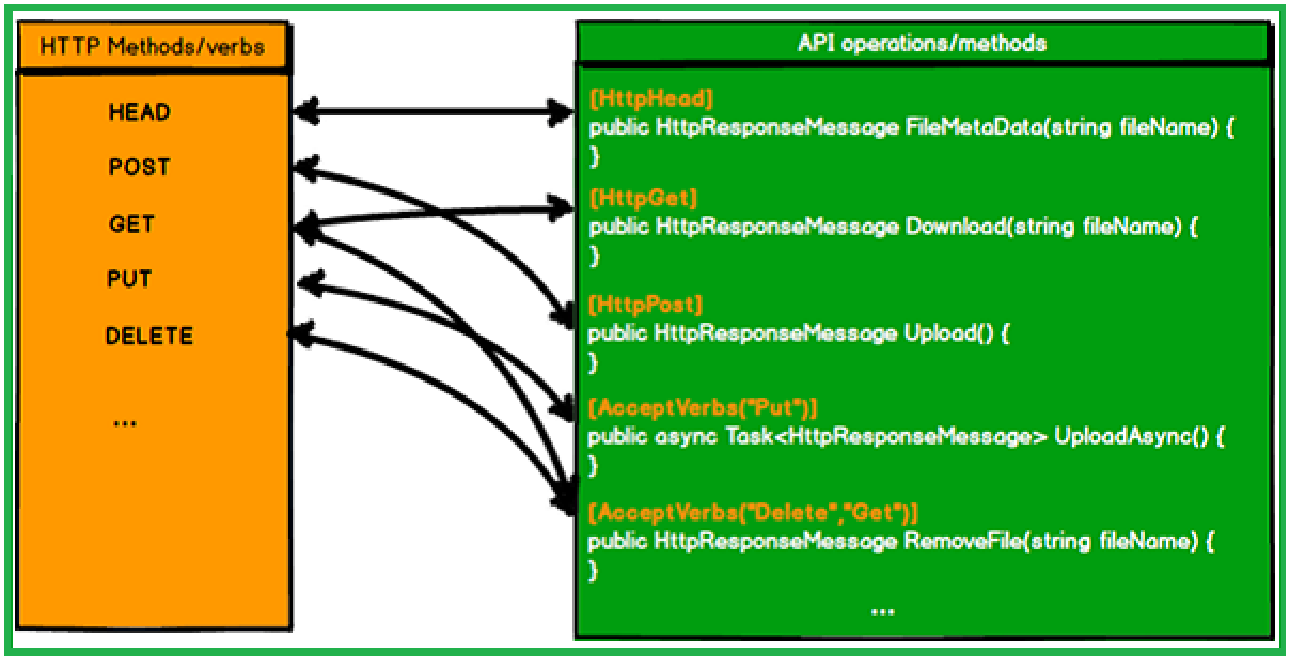

HTTP Methods/verbs
HEAD
POST
GET
PUT
DELETE
...
API operations/methods
public HttpResponseMessage Head(string fileName) {
}
public HttpResponseMessage GetFile(string fileName) {
}
public HttpResponseMessage PostFile(string fileName) {
}
public HttpResponseMessage PutFile(string fileName) {
}
public HttpResponseMessage DeleteFile(string fileName) {
}
...

But this approach may not be convenient for different scenarios. For example, you may want to define multiple `GET` or `POST` methods within a single API controller class. In this case, the framework lets you define an action level route to include the action as part of a request URI. The following code shows how this can be configured.

```
public static void Register(HttpConfiguration config)
{
    // Web API configuration and services
    // Web API routes
     config.MapHttpAttributeRoutes();

     config.Routes.MapHttpRoute(name: "PhysicianApi",
                routeTemplate: "{controller}/{action}/{id}",
                defaults: new { id = RouteParameter.Optional });
}
```

Still this may not be sufficient to satisfy other scenarios. Suppose you want a method that deletes a file from a centralized repository and you want to use the same method to get the file as well. In a situation like this, the the Web API framework lets you to decorate the action with `Get` and `Delete` HTTP method attributes. See the picture below.


HTTP Methods/verbs
HEAD
POST
GET
PUT
DELETE
...
API operations/methods
[HttpHead]
public HttpResponseMessage FileMetaData(string fileName) {
}
[HttpGet]
public HttpResponseMessage Download(string fileName) {
}
[HttpPost]
public HttpResponseMessage Upload() {
}
[AcceptVerbs("Put")]
public async Task<HttpResponseMessage> UploadAsync() {
}
[AcceptVerbs("Delete","Get")]
public HttpResponseMessage RemoveFile(string fileName) {
}
...

Notice that `RemoveFile` method can be invoked by using Delete(`HttpDelete`) or Get(`HttpGet`) verbs. HTTP method attribue is also helpful to define a suitable name for a service method.

The other feature the framework provided is that the ability to facilitate an attribute routing template. This is similar to ASP.NET MVC routing except it relay on HTTP Methods (verbs) not the URI of the action. This lets you define several types of actions under the Web API service. For example, you can define a parameterized URI's service methods. So when a request is made to a service method, you can pass a parameter as part of the request URI. The following example code shows a Web API attribute routing on `GetPhysicianBySpeciality` method.

```
[Route("physician/{speciality}/{physicianID}")]
public PhysicianBase GetPhysicianBySpeciality(string speciality, int physicianID)
{
    // Todo : Process the request
}
```

So this will facilitate the client to send a request like http://localhost/physician/eyecare/1. There are various types of routing attributes that helps to decorate the Web API controller and its methods. These are:

ActionName	lets you define action name routing
Http Methods(HttpGet, HttpPost, AcceptVerbs ...)	lets you define HTTP methods(verbs)
NonAction	prevent the action not being invoked
Route	lets you define template along with/without parameter
RoutePrefix	lets you define a controller prefix name

ASP.NET Web API is also enriched with useful libraries and classes like `HttpClient` , `HttpRequestMessage`, and `HttpResponseMessage`. See the reference section for list of Web API references. This is by far enough as introduction. In the following section, I'll explain one of the main areas of the article.

Data Streaming

One of frequently performed operation through the internet is data streaming. ASP.NET Web API is capable of processing large size of stream data from/to the server/client. The stream data can be a file located on a directory or binary data stored on database. In this article, there are two basic methods, namely `Download` and `Upload` are involved to accomplish data streaming. The download is responsible for pulling stream data from the server whereas upload is responsible for saving stream data to server.

Note: This article explains data streaming on resources that are located in specific(configurable) directories and be able to streamed through Web API service.

Participating projects

- `WebAPIDataStreaming`
- `WebAPIClient`
- `POCOLibrary`

Before explaining the code section, there are certain configurations that need to be done on IIS (7.5) and Web API service web.config file.

1. Make sure *Downloads/Uploads* directories/files are granted the necessary permissions (Read/Write) to a user (**IIS_IUSRS**)
2. Make sure there is enough memory (RAM) and Hard Disk space available to process large sized files.
3. For larger sized file data,
    a. Make sure `maxRequestLength` along with a reasonable `executionTimeout` are configured in

web.config file. The value can vary depending on the size allowed to be streamed. The maximum allowed file size is 2GB.

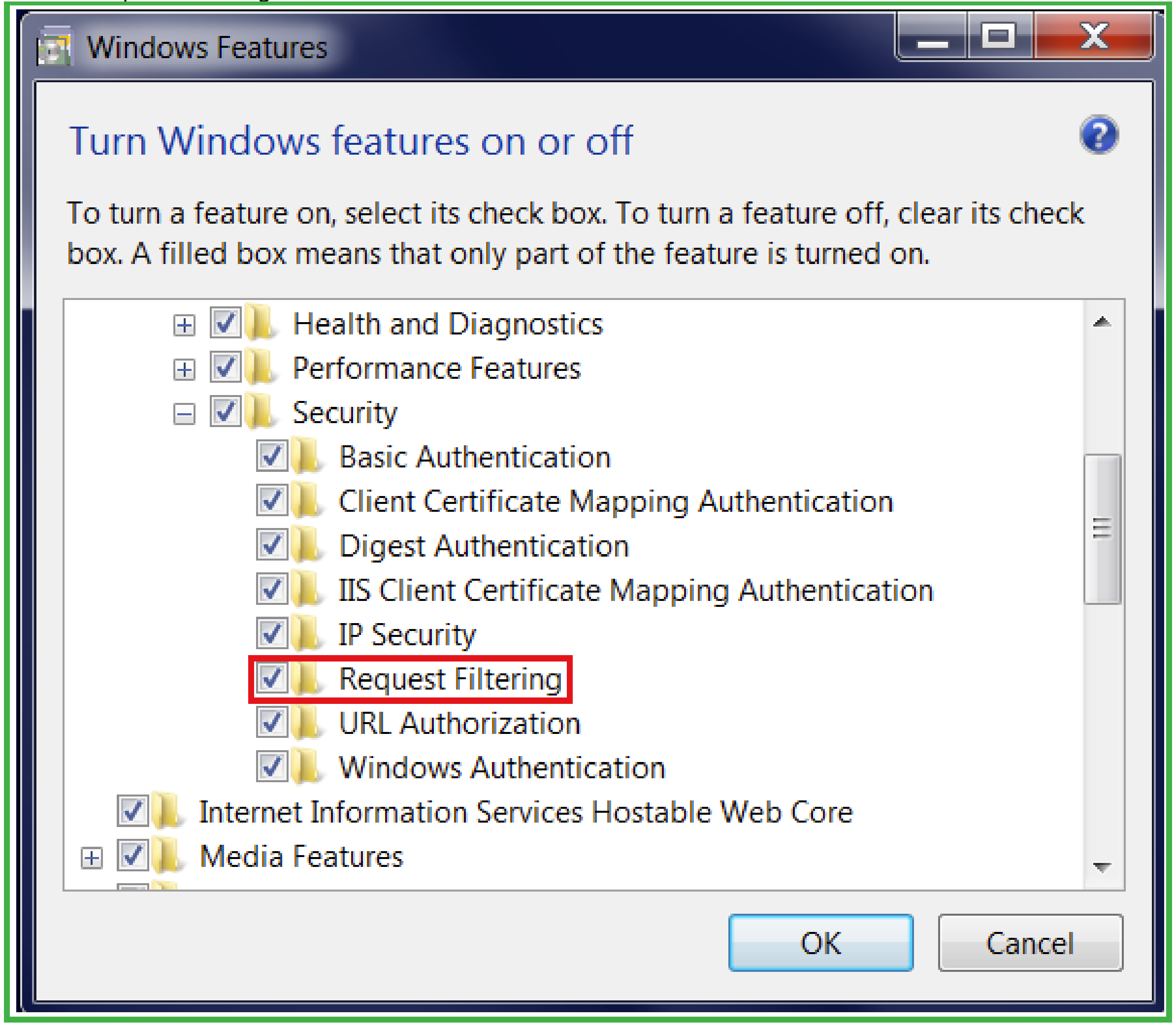
```
<system.web>
    <httpRuntime targetFramework="4.5.1"
                 maxRequestLength="2147483647" executionTimeout="600"/>
</system.web>
```

b. Make sure `maxAllowedContentLength` is configured under `requestFiltering` configuration section of web.config file. The default value for this setting is approximately 30MB and the max value is 4GB.

```
<system.webServer>
   <security>
     <requestFiltering>
       <requestLimits maxAllowedContentLength="4294967295"/>
     </requestFiltering>
   </security>
 </system.webServer>
```

Note: In order to use `maxAllowedContentLength` configuration, Request filtering feature should be enabled on IIS. Here is how to do so.

- Go to Control Panel
- Select Turn Windows Features on or off on the left side of the Programs and Features menu
- Select Internet Information Services > World Wide Web Services > Security
- Enable Request Filtering



Windows Features

Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- ✓ Health and Diagnostics
- ✓ Performance Features
- ✓ Security
  - ✓ Basic Authentication
  - ✓ Client Certificate Mapping Authentication
  - ✓ Digest Authentication
  - ✓ IIS Client Certificate Mapping Authentication
  - ✓ IP Security
  - ✓ Request Filtering
  - ✓ URL Authorization
  - ✓ Windows Authentication
- ✓ Internet Information Services Hostable Web Core
- ✓ Media Features

Details of request filtering can be found in reference section

Once these pre-configuration are completed, it is easy to create and consume data stream Web API service. The first thing to do is to define the file streaming `ApiController` along with the necessary operations. As I stated earlier, the main methods of the file streaming service are download and upload. But, it also contains other methods related to file streaming.

```
/// <summary>
/// File streaming API
/// </summary>
[RoutePrefix("filestreaming")]
[RequestModelValidator]
public class StreamFilesController : ApiController
{
    /// <summary>
    /// Get File meta data
    /// </summary>
    /// <param name="fileName">FileName value</param>
    /// <returns>FileMeta data response.</returns>
    [Route("getfilemetadata")]
    public HttpResponseMessage GetFileMetaData(string fileName)
    {
        // .........................................
        // Full code available in the source control
        // .........................................

    }

    /// <summary>
    /// Search file and return its meta data in all download directories
    /// </summary>
    /// <param name="fileName">FileName value</param>
    /// <returns>List of file meta datas response</returns>
    [HttpGet]
    [Route("searchfileindownloaddirectory")]
    public HttpResponseMessage SearchFileInDownloadDirectory(string fileName)
    {
        // .........................................
        // Full code available in the source control
        // .........................................
    }

    /// <summary>
    /// Asynchronous Download file
    /// </summary>
    /// <param name="fileName">FileName value</param>
    /// <returns>Tasked File stream response</returns>
    [Route("downloadasync")]
    [HttpGet]
    public async Task<HttpResponseMessage> DownloadFileAsync(string fileName)
    {
        // .........................................
        // Full code available in the source control
        // .........................................
    }

    /// <summary>
    /// Download file
    /// </summary>
    /// <param name="fileName">FileName value</param>
    /// <returns>File stream response</returns>
    [Route("download")]
    [HttpGet]
    public HttpResponseMessage DownloadFile(string fileName)
    {
        // .........................................
        // Full code available in the source control
        // .........................................
    }

    /// <summary>
    /// Upload file(s)
    /// </summary>
    /// <param name="overWrite">An indicator to overwrite a file if it exist in the server</param>
    /// <returns>Message response</returns>
    [Route("upload")]
    [HttpPost]
    public HttpResponseMessage UploadFile(bool overWrite)
    {
        // ........................................
        // Full code available in the source control
        // ........................................
    }

    /// <summary>
    /// Asynchronous Upload file
    /// </summary>
    /// <param name="overWrite">An indicator to overwrite a file if it exist in the server</param>
    /// <returns>Tasked Message response</returns>
    [Route("uploadasync")]
    [HttpPost]
    public async Task<HttpResponseMessage> UploadFileAsync(bool overWrite)
    {
        // ........................................
        // Full code available in the source control
        // ........................................
    }
}
```

The `Download` service method works first by checks the requested file name existence in the desired file path. If file is not found,it returns an error response object saying "*file is not found*". If it succeeds, it reads the content as bytes and attaches to the response object as an `application/octet-stream` MIMI content type.

```
/// <summary>
/// Download file
/// </summary>
/// <param name="fileName">FileName value<param>
/// <returns>File stream response<returns>
[Route("download")]
[HttpGet]
public HttpResponseMessage DownloadFile(string fileName)
{
    HttpResponseMessage response = Request.CreateResponse();
    FileMetaData metaData = new FileMetaData();
    try
    {
        string filePath = Path.Combine(this.GetDownloadPath(), @"\", fileName);
        FileInfo fileInfo = new FileInfo(filePath);

        if (!fileInfo.Exists)
        {
            metaData.FileResponseMessage.IsExists = false;
            metaData.FileResponseMessage.Content = string.Format("{0} file is not found !",
fileName);
            response = Request.CreateResponse(HttpStatusCode.NotFound, metaData, new
MediaTypeHeaderValue("text/json"));
        }
        else
        {
            response.Headers.AcceptRanges.Add("bytes");
            response.StatusCode = HttpStatusCode.OK;
            response.Content = new StreamContent(fileInfo.ReadStream());
            response.Content.Headers.ContentDisposition = new
ContentDispositionHeaderValue("attachment");
            response.Content.Headers.ContentDisposition.FileName = fileName;
            response.Content.Headers.ContentType = new MediaTypeHeaderValue("application/octet-
stream");
            response.Content.Headers.ContentLength = fileInfo.Length;
        }
    }
    catch (Exception exception)
    {
        // Log exception and return gracefully
        metaData = new FileMetaData();
        metaData.FileResponseMessage.Content = ProcessException(exception);
        response = Request.CreateResponse(HttpStatusCode.InternalServerError, metaData, new
MediaTypeHeaderValue("text/json"));
    }
    return response;
}
```

The `Upload` service method works on top of a `multipart/form-data` MIMI content type. First it checks the HTTP request content type is a type of multipart. If it succeeds, it compares the content length to the maximum allowed file size to be uploaded. If it succeeds, it starts to upload the request content to the desired location. When the operation is completed, it notifies the user with appropriate response message. The code fragment that performs upload is shown below.

```
/// <summary>
/// Upload file(s)
/// </summary>
/// <param name="overWrite">An indicator to overwrite a file if it exist in the server.</param>
/// <returns>Message response</returns>
[Route("upload")]
[HttpPost]
public HttpResponseMessage UploadFile(bool overWrite)
{
    HttpResponseMessage response = Request.CreateResponse();
    List<FileResponseMessage> fileResponseMessages = new List<FileResponseMessage>();
    FileResponseMessage fileResponseMessage = new FileResponseMessage { IsExists = false };

    try
    {
        if (!Request.Content.IsMimeMultipartContent())
        {
            fileResponseMessage.Content = "Upload data request is not valid !";
            fileResponseMessages.Add(fileResponseMessage);
            response = Request.CreateResponse(HttpStatusCode.UnsupportedMediaType,
fileResponseMessages, new MediaTypeHeaderValue("text/json"));
        }

        else
        {
            response = ProcessUploadRequest(overWrite);
        }
    }
    catch (Exception exception)
    {
        // Log exception and return gracefully
        fileResponseMessage = new FileResponseMessage { IsExists = false };
        fileResponseMessage.Content = ProcessException(exception);
        fileResponseMessages.Add(fileResponseMessage);
        response = Request.CreateResponse(HttpStatusCode.InternalServerError, fileResponseMessages,
new MediaTypeHeaderValue("text/json"));

    }
    return response;
}
```

```
/// <summary>
/// Asynchronous Upload file
/// </summary>
/// <param name="overWrite">An indicator to overwrite a file if it exist in the server.<param>
/// <returns>Tasked Message response</returns>
[Route("uploadasync")]
[HttpPost]
public async Task<HttpResponseMessage> UploadFileAsync(bool overWrite)
{
    return await new TaskFactory().StartNew(
       () =>
       {
           return UploadFile(overWrite);
       });
}

/// <summary>
/// Process upload request in the server
/// </summary>
/// <param name="overWrite">An indicator to overwrite a file if it exist in the server.</param>
/// </returns>List of message object</returns>
private HttpResponseMessage ProcessUploadRequest(bool overWrite)
{
    // .........................................
    // Full code available in the source control
    // .........................................
}
```

The client app that calls the download and upload file method is a console app. The app consumes the file streaming Web API service through `HttpClient` and related classes. Basically the download file code creates a download HTTP request object with a proper file name and sends the request to the server.

```
/// <summary>
/// Download file
/// </summary>
/// <returns>Awaitable Task object</returns>
private static async Task DownloadFile()
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine("Please specify file name  with extension and Press Enter :- ");
    string fileName = Console.ReadLine();
    string localDownloadPath = string.Concat(@"c:\", fileName); // the path can be configurable
    bool overWrite = true;
    string actionURL = string.Concat("downloadasync?fileName=", fileName);

    try
    {
        Console.WriteLine(string.Format("Start downloading @ {0}, {1} time ",
            DateTime.Now.ToLongDateString(),
            DateTime.Now.ToLongTimeString()));


        using (HttpClient httpClient = new HttpClient())
        {
            httpClient.BaseAddress = baseStreamingURL;
            HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, actionURL);

            await httpClient.SendAsync(request, HttpCompletionOption.ResponseHeadersRead).
                ContinueWith((response)
                    =>
                {
                    Console.WriteLine();
                    try
                    {
                        ProcessDownloadResponse(localDownloadPath, overWrite, response);
                    }
                    catch (AggregateException aggregateException)
                    {
                        Console.ForegroundColor = ConsoleColor.Red;
                        Console.WriteLine(string.Format("Exception : ", aggregateException));
                    }
                });
            }
        }
    catch (Exception ex)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(ex.Message);
    }
}


/// <summary>
/// Process download response object
/// </summary>
/// <param name="localDownloadFilePath">Local download file path</param>
/// <param name="overWrite">An indicator to overwrite a file if it exist in the client.</param>
/// <param name="response">Awaitable HttpResponseMessage task value</param>
private static void ProcessDownloadResponse(string localDownloadFilePath, bool overWrite,
    Task<HttpResponseMessage> response)
{
    if (response.Result.IsSuccessStatusCode)
    {
        response.Result.Content.DownloadFile(localDownloadFilePath, overWrite).
            ContinueWith((downloadmessage)
                =>
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine(downloadmessage.TryResult());
            });
    }
    else
    {
        ProcessFailResponse(response);
    }
}
```

Notice the code above. The `HttpClient` object sends the request and waits the response to send only the response header (`HttpCompletionOption.ResponseHeadersRead`), not the entire response content of the file. Once the response header is read, it performs the necessary validation on the content and if it succeeds, the actual file downloading method will be executed.

And here is the code that calls the upload file streaming Web API service method. Similar to download method, it creates a request object with a multipart form data content type and sends the request to the server. The content is validated and sent to server to be processed further.

```
/// <summary>
/// Upload file
/// </summary>
/// <returns>Awaitable task object</returns>
private static async Task UploadFile()
{
    try
    {
        string uploadRequestURI = "uploadasync?overWrite=true";
        MultipartFormDataContent formDataContent = new MultipartFormDataContent();

        // Validate the file and add to MultipartFormDataContent object
        formDataContent.AddUploadFile(@"c:\nophoto.png");
        formDataContent.AddUploadFile(@"c:\ReadMe.txt");

        if (!formDataContent.HasContent()) // No files found to be uploaded
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write(formDataContent.GetUploadFileErrorMesage());
            return;
        }
        else
        {
            string uploadErrorMessage = formDataContent.GetUploadFileErrorMesage();
            if (!string.IsNullOrWhiteSpace(uploadErrorMessage)) // Some files couldn't be found
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.Write(uploadErrorMessage);
            }

            HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Post, uploadRequestURI);
            request.Content = formDataContent;

            using (HttpClient httpClient = new HttpClient())
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine(string.Format("Start uploading @ {0}, {1} time ",
                DateTime.Now.ToLongDateString(),
                DateTime.Now.ToLongTimeString()));

                httpClient.BaseAddress = baseStreamingURL;
                await httpClient.SendAsync(request).
                      ContinueWith((response)
                          =>
                          {
                              try
                              {
                                  ProcessUploadResponse(response);
                              }
                              catch (AggregateException aggregateException)
                              {
                                  Console.ForegroundColor = ConsoleColor.Red;
                                  Console.WriteLine(string.Format("Exception : ",
aggregateException));
                              }
                          });
            }
        }
    }
    catch (Exception ex)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(ex.Message);
    }
}

/// <summary>
/// Process download response object
/// </summary>
/// <param name="response">Awaitable HttpResponseMessage task value</param>
private static void ProcessUploadResponse(Task<HttpResponseMessage> response)
{
    if (response.Result.IsSuccessStatusCode)
    {
        string uploadMessage = string.Format("\nUpload completed @ {0}, {1} time ",
                    DateTime.Now.ToLongDateString(),
                    DateTime.Now.ToLongTimeString());
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine(string.Format("{0}\nUpload Message : \n{1}", uploadMessage,

JsonConvert.SerializeObject(response.Result.Content.ReadAsAsync<List<FileResponseMessage>>
().TryResult(), Formatting.Indented)));
    }
    else
    {
        ProcessFailResponse(response);
    }
}
```

The data streaming project is also consists of extension classes and methods which are not explicitly explained in the article. Download the source code and explore it.

What's Next

The next section explains about Working with HTTPS

References

- http://en.wikipedia.org/wiki/Representational_state_transfer
- http://en.wikipedia.org/wiki/Request-response
- http://www.ietf.org/rfc/rfc2616
- http://msdn.microsoft.com/en-us/library/hh849329(v=vs.108).aspx
- http://www.iis.net/configreference/system.webserver/security/requestfiltering/requestlimits
- http://www.ASP.NET/web-api/
- http://technet.microsoft.com/en-us/library/bb727008.aspx

History and GitHub Version

- WebAPI Thoughts @ GitHub
- Update on Nov 4, 2014
- Update on Nov 18, 2014
- Update on Nov 20, 2014
- Update on Oct 18, 2015 - Article code format issue

License

Share

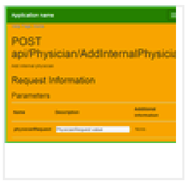
About the Author

Wonde Tadesse

Software Developer (Senior)
United States

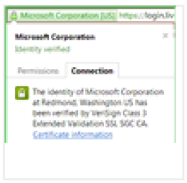
MSCS, MCTS, Senior Software Engineer, Architect, Craftsman ...

You may also be interested in...



Web API Thoughts 3 of 3 - Extending Web API Documentation

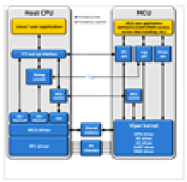
AvePoint & Amazon Web Services: SharePoint in AWS



Web API Thoughts 2 of 3 - Working with HTTPS

Announcing 18 New How-To Intel IoT Code Samples

ASP.NET Web API



Using an MCU on the Intel® Edison Board with the Ultrasonic Range Sensor

Comments and Discussions

32 messages have been posted for this article Visit **http://www.codeproject.com/Articles/838274/Web-API-Thoughts-of-Data-Streaming** to post and view comments on this article, or click **here** to get a print view with messages.