

RESTful Day #5: Security in Web APIs-Basic Authentication and Token based custom Authorization in Web APIs using Action Filters.

Table of Contents

Table of Contents	1
Introduction	2
Roadmap	2
Security in WebAPI	3
Authentication	4
Authorization	4
Maintaining Session	4
Basic Authentication	4
Pros and Cons of Basic Authentication	4
Token Based Authorization	4
WebAPI with Basic Authentication and Token Based Authorization	5
Creating User Service	5
UserServices:	6
Resolve dependency of UserService:	8
Implementing Basic Authentication	9
Step 1: Create generic Authentication Filter	9
Step 2: Create Basic Authentication Identity	12
Step 3: Create a Custom Authentication Filter	12
Step 4: Basic Authentication on Controller	14
Running the application	16
Design discrepancy	22
Implementing Token based Authorization	22
Set Database	23
Set Business Services	23
Setup WebAPI/Controller:	28
AuthenticateController:	28
Setup Authorization Action Filter	33
Mark Controllers with Authorization filter	34
Maintaining Session using Token	35
Running the application	36
Test Authentication	36
Test Authorization	37

Conclusion	41
References	41
Other Series	41

Introduction

Security has always been a major concern we talk about enterprise level applications, especially when we talk about exposing our business through services. I have already explained a lot on WebAPI in my earlier articles of the series. I explained, how do we create a WebAPI, how to resolve dependencies to make it a loosely coupled design, defining custom routes, making use of attribute routing. My article will explain how we can achieve security in a WebAPI. This article will explain how to make WebAPI secure using Basic Authentication and Token based authorization. I'll also explain how we can leverage token based authorization and Basic authentication in WebAPI to maintain sessions in WebAPI. There is no standard way of achieving security in WebAPI. We can design our own security technique and structure which suits best to our application.

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 handling/logging in Web APIs using Action Filters, Exception Filters and nLog.
- RESTful Day #7: Unit testing ASP.NET Web APIs 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.

Security in WebAPI

Security in itself is very complicated and tricky topic. I'll try to explain how we can achieve it in WebAPI in my own way. When we plan to create an enterprise level application, we especially want to take care of authentication and authorization. These are two techniques if used well makes our application secure, in our case makes our WebAPI more secure.



Image credit: https://pixabay.com/static/uploads/photo/2014/12/25/10/56/road-sign-579554_180.jpg

Authentication

Authentication is all about the identity of an end user. It's about validating the identity of a user who is accessing our system, that he is authenticated enough to use our resources or not. Does that end user have valid credentials to log in our system? Credentials can be in the form of a user name and password. We'll use Basic Authentication technique to understand how we can achieve authentication in WebAPI.

Authorization

Authorization should be considered as a second step after authentication to achieve security. Authorization means what all permissions the authenticated user has to access web resources. Is allowed to access/ perform action on that resource? This could be achieved by setting roles and permissions for an end user who is authenticated, or can be achieved through providing a secure token, using which an end user can have access to other services or resources.

Maintaining Session

RESTful services work on a stateless protocol i.e. HTTP. We can achieve maintaining session in Web API through token based authorization technique. An authenticated user will be allowed to access resources for a particular period of time, and can re-instantiate the request with an increased session time delta to access other resource or the same resource. Websites using WebAPIs as RESTful services may need to implement login/logout for a user, to maintain sessions for the user, to provide roles and permissions to their user, all these features could be achieved using basic authentication and token based authorization. I'll explain this step by step.

Basic Authentication

Basic authentication is a mechanism, where an end user gets authenticated through our service i.e. RESTful service with the help of plain credentials such as user name and password. An end user makes a request to the service for authentication with user name and password embedded in request header. Service receives the request and checks if the credentials are valid or not, and returns the response accordingly, in case of invalid credentials, service responds with 401 error code i.e. unauthorized. The actual credentials through which comparison is done may lie in database, any config file like web.config or in the code itself.

Pros and Cons of Basic Authentication

Basic authentication has its own pros and cons. It is advantageous when it comes to implementation, it is very easy to implement, it is nearly supported by all modern browsers and has become an authentication standard in RESTful / Web APIs. It has disadvantages like sending user credentials in plain text, sending user credentials inside request header, i.e. prone to hack. One has to send credentials each time a service is called. No session is maintained and a user cannot logout once logged in through basic authentication. It is very prone to CSRF (Cross Site Request Forgery).

Token Based Authorization

Authorization part comes just after authentication, once authenticated a service can send a token to an end user through which user can access other resources. The token could be any encrypted key, which only server/service understands and when it fetches the token from the request made by end user, it validates the token and authorizes user into the system. Token generated could be stored in a database or an external file as well i.e. we need to persist the

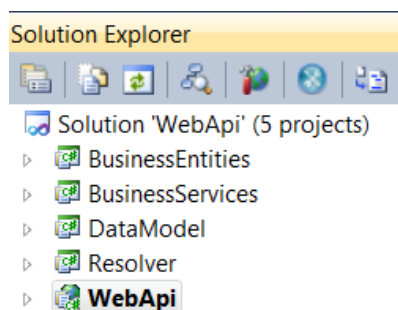
token for future references. Token can have its own lifetime, and may expire accordingly. In that case user will again have to be authenticated into the system.

WebAPI with Basic Authentication and Token Based Authorization

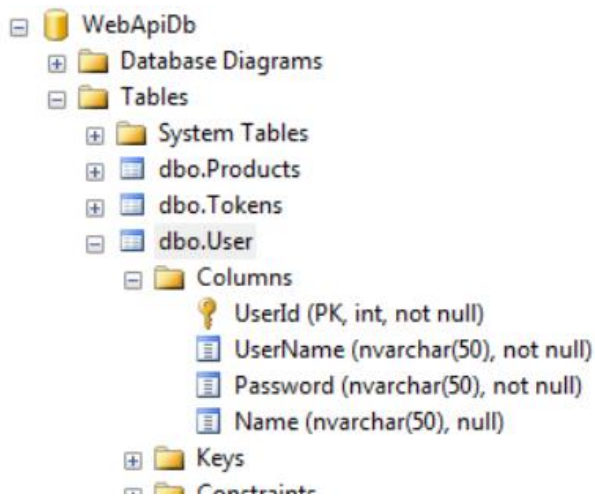


Creating User Service

Just open your WebAPI project or the WebAPI project that we discussed in the [last](#) part of learning WebAPI.



We have BusinessEntities, BusinessServices, DataModel, DependencyResolver and a WebApi project as well. We already have a User table in database, or you can create your own database with a table like User Table as shown below,

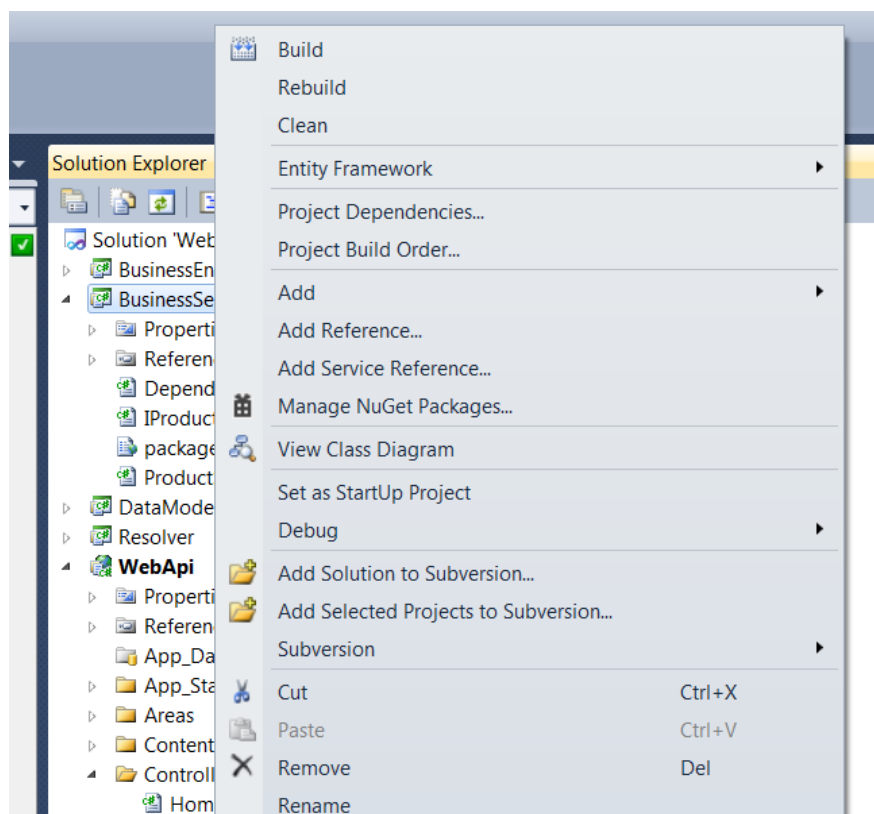


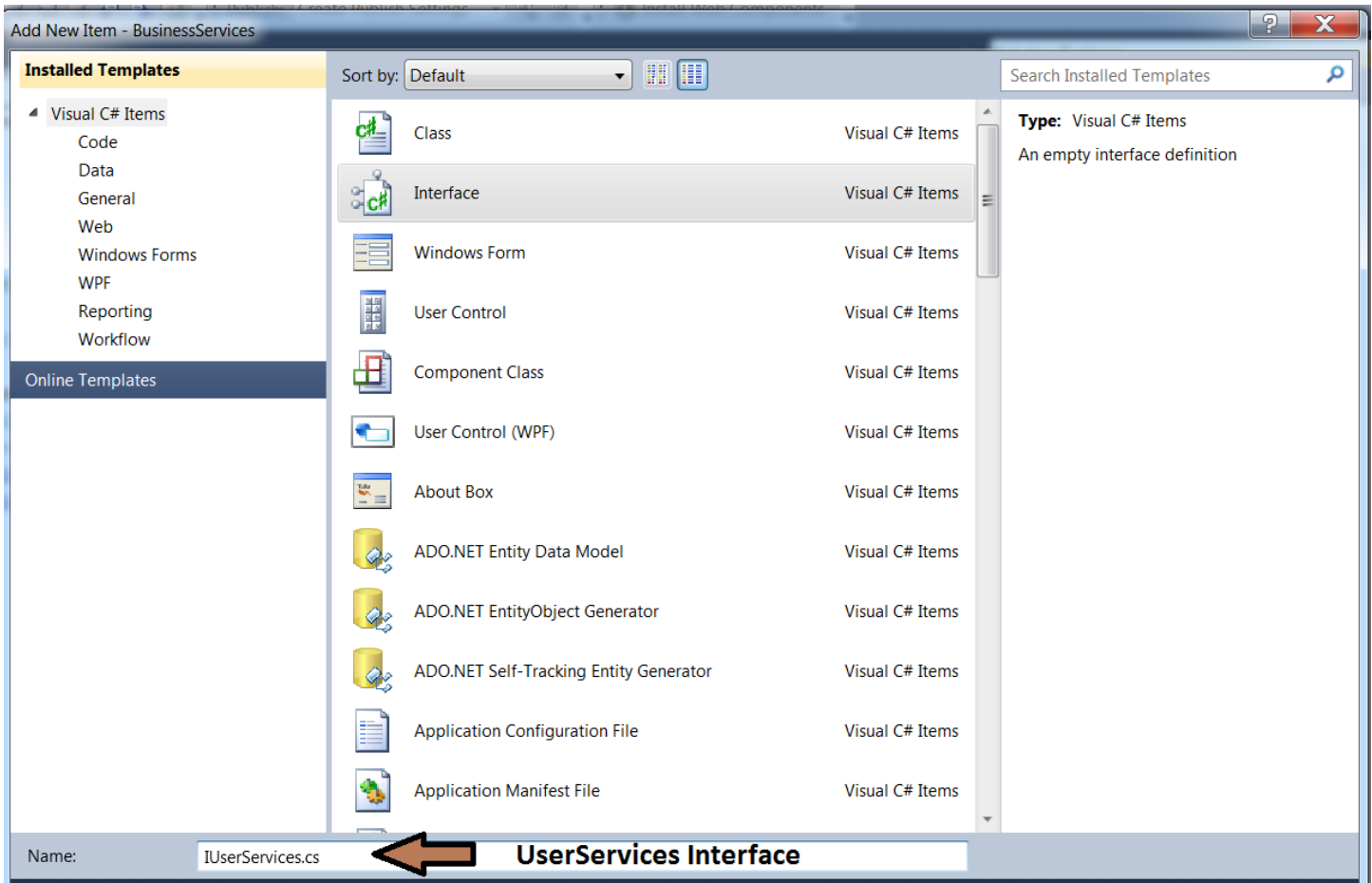
	UserId	UserName	Password	Name
1	1	akhil	akhil	Akhil Mittal
2	2	api	api	API User

I am using WebAPI database, scripts I have attached for download.

UserServices:

Go to BusinessServices project and add a new interface IUserService and a service named UserServices implementing that interface,





Just define one method named Authenticate in the interface.

```
namespace BusinessServices
{
    public interface IUserServices
    {
        int Authenticate(string userName, string password);
    }
}
```

This method takes username and password as a parameter and returns particular userId if the user is authenticated successfully.

Just implement this method in UserServices.cs class, just like we created services earlier in the series,

```
using DataModel.UnitOfWork;

namespace BusinessServices
{
    /// <summary>
    /// Offers services for user specific operations
    /// </summary>
    public class UserServices : IUserServices
    {
        private readonly UnitOfWork _unitOfWork;
```

```

    /// <summary>
    /// Public constructor.
    /// </summary>
    public UserServices(UnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    /// <summary>
    /// Public method to authenticate user by user name and password.
    /// </summary>
    /// <param name="userName"></param>
    /// <param name="password"></param>
    /// <returns></returns>
    public int Authenticate(string userName, string password)
    {
        var user = _unitOfWork.UserRepository.Get(u => u.UserName == userName &&
u.Password == password);
        if (user != null && user.UserId > 0)
        {
            return user.UserId;
        }
        return 0;
    }
}

```

You can clearly see that Authenticate method just checks user credentials from UserRepository and returns the values accordingly. The code is very much self-explanatory.

Resolve dependency of UserService:

Just open DependencyResolver class in BusinessServices project itself and add its dependency type so that we get UserServices dependency resolved at run time,so add registerComponent.RegisterType<IUserServices, UserServices>(); line to SetUp method.Our class becomes,

```

using System.ComponentModel.Composition;
using DataModel;
using DataModel.UnitOfWork;
using Resolver;

namespace BusinessServices
{
    [Export(typeof(IComponent))]
    public class DependencyResolver : IComponent
    {
        public void SetUp(IRegisterComponent registerComponent)
        {
            registerComponent.RegisterType<IProductServices, ProductServices>();
            registerComponent.RegisterType<IUserServices, UserServices>();
        }
    }
}

```


Implementing Basic Authentication

Step 1: Create generic Authentication Filter

Add a folder named Filters to the WebAPI project and add a class named GenericAuthenticationFilter under that folder. Derive that class from [AuthorizationFilterAttribute](#), this is a class under System.Web.Http.Filters. I have created the generic authentication filter will be like,

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false)]
public class GenericAuthenticationFilter : AuthorizationFilterAttribute
{
    /// <summary>
    /// Public default Constructor
    /// </summary>
    public GenericAuthenticationFilter()
    {
    }

    private readonly bool _isActive = true;

    /// <summary>
    /// parameter isActive explicitly enables/disables this filter.
    /// </summary>
    /// <param name="isActive"></param>
    public GenericAuthenticationFilter(bool isActive)
    {
        _isActive = isActive;
    }

    /// <summary>
    /// Checks basic authentication request
    /// </summary>
    /// <param name="filterContext"></param>
    public override void OnAuthorization(HttpActionContext filterContext)
    {
        if (!_isActive) return;
        var identity = FetchAuthHeader(filterContext);
        if (identity == null)
        {
            ChallengeAuthRequest(filterContext);
            return;
        }
        var genericPrincipal = new GenericPrincipal(identity, null);
        Thread.CurrentPrincipal = genericPrincipal;
        if (!OnAuthorizeUser(identity.Name, identity.Password, filterContext))
        {
            ChallengeAuthRequest(filterContext);
            return;
        }
        base.OnAuthorization(filterContext);
    }
}
```

```

    /// <summary>
    /// Virtual method.Can be overridden with the custom Authorization.
    /// </summary>
    /// <param name="user"></param>
    /// <param name="pass"></param>
    /// <param name="filterContext"></param>
    /// <returns></returns>
    protected virtual bool OnAuthorizeUser(string user, string pass, HttpContext
filterContext)
    {
        if (string.IsNullOrEmpty(user) || string.IsNullOrEmpty(pass))
            return false;
        return true;
    }

    /// <summary>
    /// Checks for authentication header in the request and parses it, creates user
credentials and returns as BasicAuthenticationIdentity
    /// </summary>
    /// <param name="filterContext"></param>
    protected virtual BasicAuthenticationIdentity FetchAuthHeader(HttpContext
filterContext)
    {
        string authHeaderValue = null;
        var authRequest = filterContext.Request.Headers.Authorization;
        if (authRequest != null && !string.IsNullOrEmpty(authRequest.Scheme) &&
authRequest.Scheme == "Basic")
            authHeaderValue = authRequest.Parameter;
        if (string.IsNullOrEmpty(authHeaderValue))
            return null;
        authHeaderValue =
Encoding.Default.GetString(Convert.FromBase64String(authHeaderValue));
        var credentials = authHeaderValue.Split(':');
        return credentials.Length < 2 ? null : new
BasicAuthenticationIdentity(credentials[0], credentials[1]);
    }

    /// <summary>
    /// Send the Authentication Challenge request
    /// </summary>
    /// <param name="filterContext"></param>
    private static void ChallengeAuthRequest(HttpContext filterContext)
    {
        var dnsHost = filterContext.Request.RequestUri.DnsSafeHost;
        filterContext.Response =
filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized);
        filterContext.Response.Headers.Add("WWW-Authenticate", string.Format("Basic
realm=\"{0}\"", dnsHost));
    }
}

```

Since this is an AuthorizationFilter derived class, we need to override its methods to add our custom logic. Here "OnAuthorization" method is overridden to add a custom logic. Whenever we get ActionContext on OnAuthorization, we'll check for its header, since we are pushing our service to follow BasicAuthentication, the request headers should

contain this information. I have used `FetchAuthHeader` to check the scheme, if it comes to be "Basic" and thereafter store the credentials i.e. user name and password in a form of an object of class `BasicAuthenticationIdentity`, therefore creating an identity out of valid credentials.

```
protected virtual BasicAuthenticationIdentity FetchAuthHeader(HttpContext
filterContext)
{
    string authHeaderValue = null;
    var authRequest = filterContext.Request.Headers.Authorization;
    if (authRequest != null && !string.IsNullOrEmpty(authRequest.Scheme) &&
authRequest.Scheme == "Basic")
        authHeaderValue = authRequest.Parameter;
    if (string.IsNullOrEmpty(authHeaderValue))
        return null;
    authHeaderValue =
Encoding.Default.GetString(Convert.FromBase64String(authHeaderValue));
    var credentials = authHeaderValue.Split(':');
    return credentials.Length < 2 ? null : new
BasicAuthenticationIdentity(credentials[0], credentials[1]);
}
```

I am expecting values to be encrypted using Base64 string; You can use your own encryption mechanism as well. Later on in `OnAuthorization` method we create a `genericPrincipal` with the created identity and assign it to current Thread principal,

```
var genericPrincipal = new GenericPrincipal(identity, null);
Thread.CurrentPrincipal = genericPrincipal;
if (!OnAuthorizeUser(identity.Name, identity.Password, filterContext))
{
    ChallengeAuthRequest(filterContext);
    return;
}
base.OnAuthorization(filterContext);
```

Once done, a challenge to that request is added, where we add response and tell the Basic realm,

```
filterContext.Response.Headers.Add("WWW-Authenticate", string.Format("Basic realm=\"{0}\"",
dnsHost));
in ChallengeAuthRequest method.
```

If no credentials is provided in the request, this generic authentication filter sets generic authentication principal to the current thread principal.

Since we know the drawback that in basic authentication credentials are passed in a plain text, so it would be good if our service uses SSL for communication or message passing.

We have an overridden constructor as well that allows to stop the default behavior of the filter by just passing in a parameter i.e. true or false.

```
public GenericAuthenticationFilter(bool isActive)
{
    _isActive = isActive;
}
```

We can use `OnAuthorizeUser` for custom authorization purposes.

Step 2: Create Basic Authentication Identity

Before we proceed further, we also need the `BasicIdentity` class, that takes hold of credentials and assigns it to `GenericPrincipal`. So just add one more class named `BasicAuthenticationIdentity` deriving from `GenericIdentity`. This class contains three properties i.e. `UserName`, `Password` and `UserId`. I purposely added `UserId` because we'll need that in future. So our class will be like,

```
using System.Security.Principal;

namespace WebApi.Filters
{
    /// <summary>
    /// Basic Authentication identity
    /// </summary>
    public class BasicAuthenticationIdentity : GenericIdentity
    {
        /// <summary>
        /// Get/Set for password
        /// </summary>
        public string Password { get; set; }
        /// <summary>
        /// Get/Set for UserName
        /// </summary>
        public string UserName { get; set; }
        /// <summary>
        /// Get/Set for UserId
        /// </summary>
        public int UserId { get; set; }

        /// <summary>
        /// Basic Authentication Identity Constructor
        /// </summary>
        /// <param name="userName"></param>
        /// <param name="password"></param>
        public BasicAuthenticationIdentity(string userName, string password)
            : base(userName, "Basic")
        {
            Password = password;
            UserName = userName;
        }
    }
}
```

Step 3: Create a Custom Authentication Filter

Now you are ready to use your own Custom Authentication filter. Just add one more class under that Filters project and call it `ApiAuthenticationFilter`, this class will derive from `GenericAuthenticationFilter`, that we created in first step. This class overrides `OnAuthorizeUser` method to add custom logic for authenticating a request. It makes use of `UserService` that we created earlier to check the user,

```
protected override bool OnAuthorizeUser(string username, string password, HttpContext
actionContext)
{
    var provider = actionContext.ControllerContext.Configuration
```



```

        .DependencyResolver.GetService(typeof(IUserServices)) as
IUserServices;
    if (provider != null)
    {
        var userId = provider.Authenticate(username, password);
        if (userId > 0)
        {
            var basicAuthenticationIdentity = Thread.CurrentPrincipal.Identity as
BasicAuthenticationIdentity;
            if (basicAuthenticationIdentity != null)
                basicAuthenticationIdentity.UserId = userId;
            return true;
        }
    }
    return false;
}

```

Complete class:

```

using System.Threading;
using System.Web.Http.Controllers;
using BusinessServices;

namespace WebApi.Filters
{
    /// <summary>
    /// Custom Authentication Filter Extending basic Authentication
    /// </summary>
    public class ApiAuthenticationFilter : GenericAuthenticationFilter
    {
        /// <summary>
        /// Default Authentication Constructor
        /// </summary>
        public ApiAuthenticationFilter()
        {
        }

        /// <summary>
        /// AuthenticationFilter constructor with isActive parameter
        /// </summary>
        /// <param name="isActive"></param>
        public ApiAuthenticationFilter(bool isActive)
            : base(isActive)
        {
        }

        /// <summary>
        /// Protected overridden method for authorizing user
        /// </summary>
        /// <param name="username"></param>
        /// <param name="password"></param>
        /// <param name="actionContext"></param>
        /// <returns></returns>
        protected override bool OnAuthorizeUser(string username, string password,
HttpActionContext actionContext)
        {

```

```

        var provider = actionContext.ControllerContext.Configuration
            .DependencyResolver.GetService(typeof(IUserServices)) as
IUserServices;
        if (provider != null)
        {
            var userId = provider.Authenticate(username, password);
            if (userId > 0)
            {
                var basicAuthenticationIdentity = Thread.CurrentPrincipal.Identity as
BasicAuthenticationIdentity;
                if (basicAuthenticationIdentity != null)
                    basicAuthenticationIdentity.UserId = userId;
                return true;
            }
        }
        return false;
    }
}
}

```

Step 4: Basic Authentication on Controller

Since we already have our products controller,

```

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);
        return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
    }
}

```

```

    }

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

    // 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 > 0)
            return _productServices.DeleteProduct(id);
        return false;
    }
}

```

There are three ways in which you can use this authentication filter.

Just apply this filter to ProductController. You can add this filter at the top of the controller, for all API requests to be validated,

```

[ApiAuthenticationFilter]
[RoutePrefix("v1/Products/Product")]
public class ProductController : ApiController

```

You can also globally add this in Web API configuration file , so that filter applies to all the controllers and all the actions associated to it,

```
GlobalConfiguration.Configuration.Filters.Add(new ApiAuthenticationFilter());
```

You can also apply it to Action level too by your wish to apply or not apply authentication to that action,

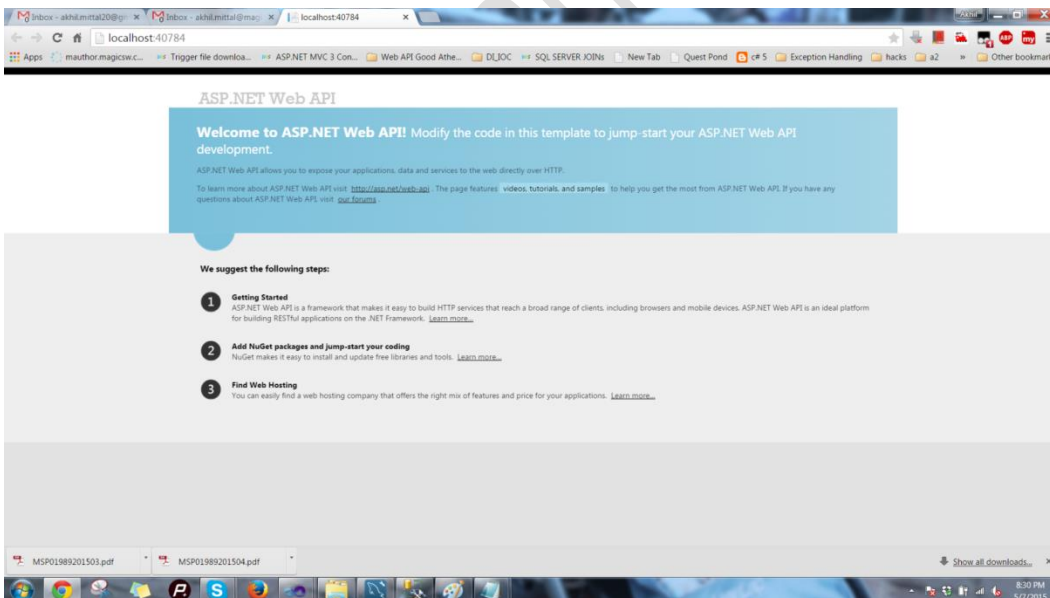
```
// GET api/product
[ApiAuthenticationFilter(true)]
[GET("allproducts")]
[GET("all")]
public HttpResponseMessage Get()
{
    .....
}

// GET api/product/5
[ApiAuthenticationFilter(false)]
[GET("productid/{id?}")]
[GET("particularproduct/{id?}")]
[GET("myproduct/{id:range(1, 3)}")]
public HttpResponseMessage Get(int id)
{
    .....
}
```

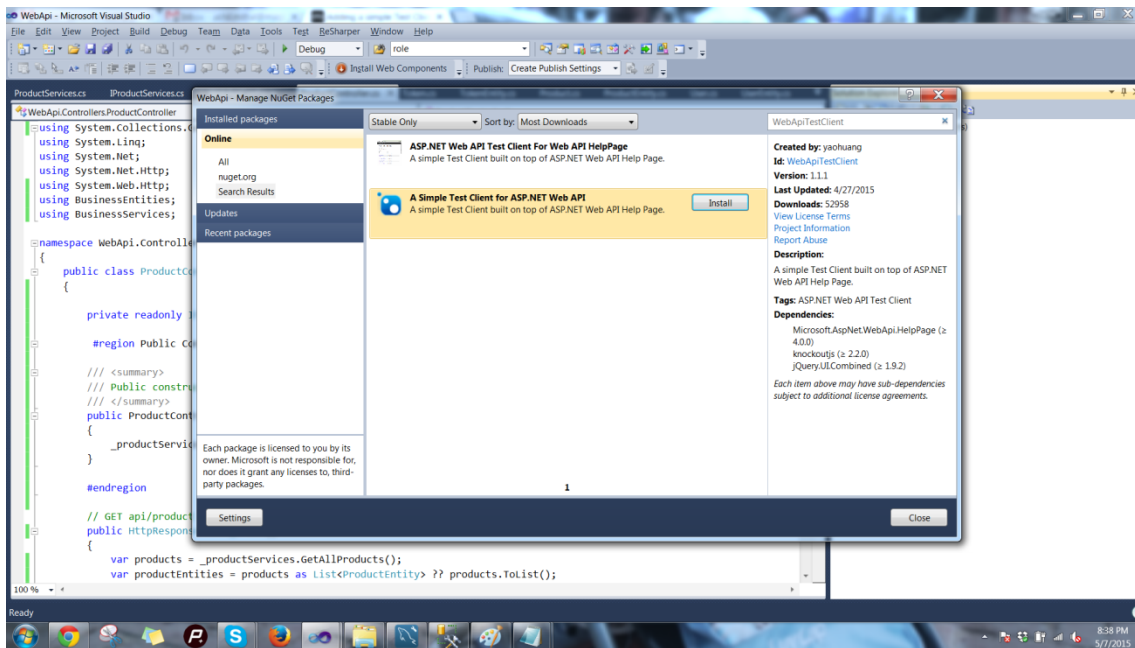
Running the application

We have already implemented Basic Authentication, just try to run the application to test if it is working

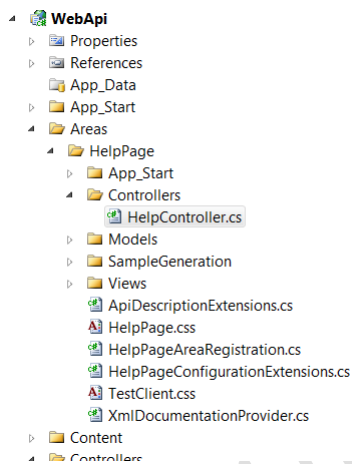
Just run the application, we get,



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}](#)

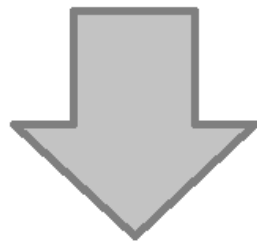
[Documentation for 'Delete'.](#)

[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,

**Press this button to
test the API**



Test API

Service for Get All products,

[Help](#) [Page](#) [Home](#)

GET v1/Products/Product/all

Documentation for 'Get'.

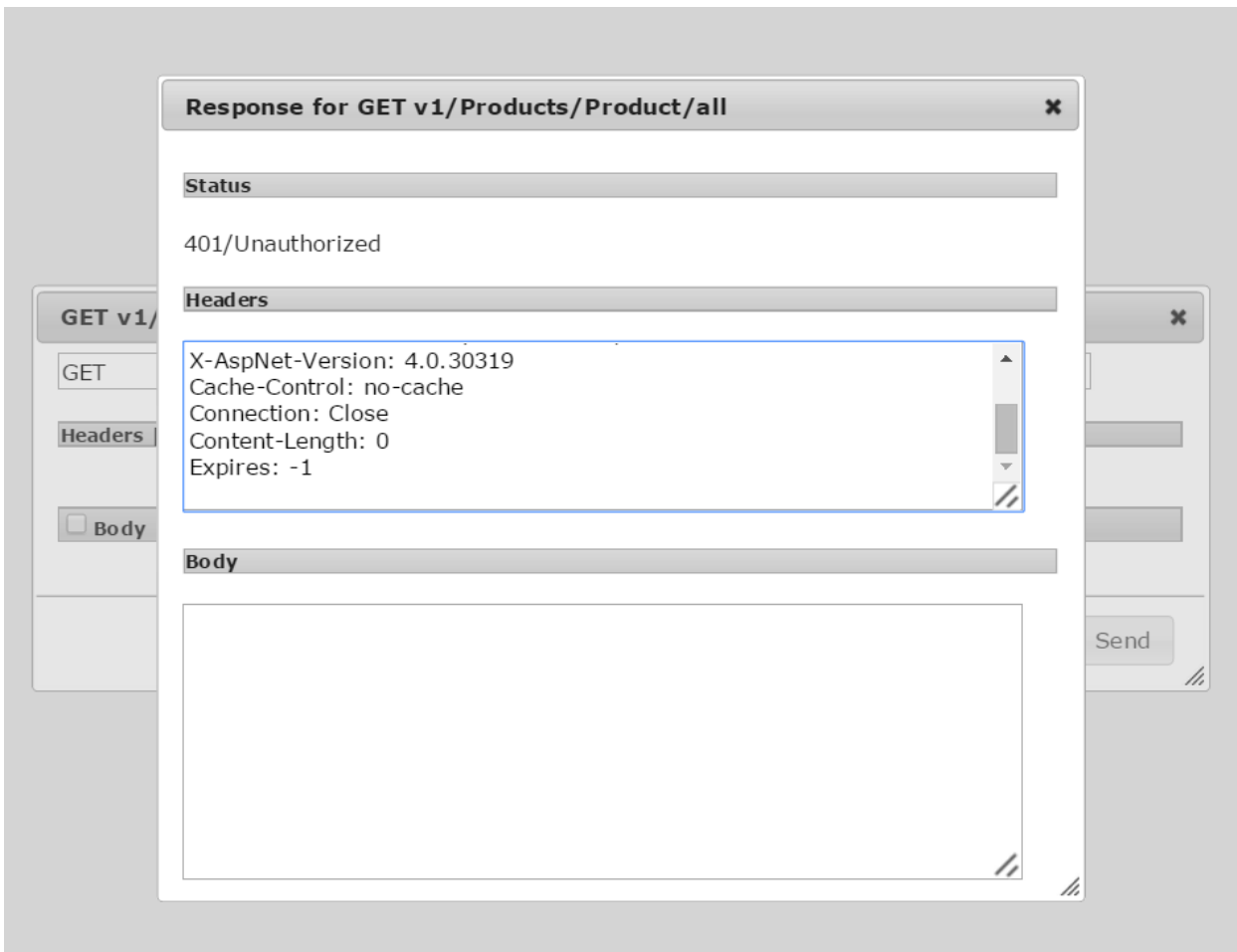
GET v1/Products/Product/all ×

GET

Headers | [Add header](#)

☐ **Body**

When you click on Send request, a popup will come asking Authentication required. Just cancel that popup and let request go without credentials. You'll get a response of Unauthorized i.e. 401,



This means our authentication mechanism is working.



Just to double sure it, let's send the request with credentials now. Just add a header too with the request. Header should be like ,

Authorization : Basic YWtoaWw6YWtoaWw=

Here "YWtoaWw6YWtoaWw=" is my Base64 encoded user name and password i.e. akhil:akhil

GET v1/Products/Product/all

GETv1/Products/Product/all

Headers | Add header

Authorization : Basic YWtoaWw6YWtoDelete

☐ Body

Send

Click on Send and we get the response as desired,

Response for GET v1/Products/Product/all

Status

200/OK

Headers

Server: ASP.NET Development Server/10.0.0.0
Date: Sun, 31 May 2015 09:11:10 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8

Body

```
"ProductId": 1,
"ProductName": "Laptop"
},
{
"ProductId": 2,
"ProductName": "computer"
},
{
"ProductId": 4,
"ProductName": "iPhone"
}
].
```



Likewise you can test all the service endpoints.
This means our service is working with Basic Authentication.

Design discrepancy

This design when running on SSL is very good for implementing Basic Authentication. But there are few scenarios in which, along with Basic Authentication I would like to leverage authorization too and not even authorization but sessions too. When we talk about creating an enterprise application, it just does not limit to securing our endpoint with authentication only.

In this design, each time I'll have to send user name and password with every request. Suppose I want to create such application, where authentication only occurs only once as my login is done and after successfully authenticated i.e. logging in I must be able to use other services of that application i.e. I am authorized now to use those services. Our application should be that robust that it restricts even authenticated user to use other services until he is not authorized. Yes I am talking about Token based authorization.

I'll expose only one endpoint for authentication and that will be my login service .So client only knows about that login service that needs credentials to get logged in to system.

After client successfully logs in I'll send a token that could be a GUID or an encrypted key by any xyz algorithm that I want when user makes request for any other services after login, should provide this token along with that request. And to maintain sessions, our token will have an expiry too, that will last for 15 minutes, can be made configurable with the help of web.config file. After session is expired, user will be logged out, and will again have to use login service with credentials to get a new token. Seems exciting to me, let's implement this ☺

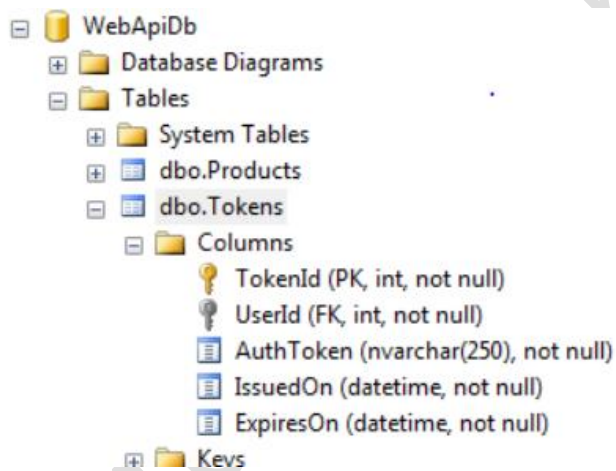
Implementing Token based Authorization

To overcome above mentioned scenarios, let's start developing and giving our application a shape of thick client enterprise architecture.



Set Database

Let's start with setting up a database. When we see our already created database that we had set up in [first](#) part of the series, we have a token table. We require this token table for token persistence. Our token will persist in database with an expiry time. If you are using your own database, you can create token table as,



Set Business Services

Just navigate to BusinessServices and create one more Interface named ITokenServices for token based operations,

```
using BusinessEntities;

namespace BusinessServices
{
    public interface ITokenServices
```

```

{
    #region Interface member methods.
    /// <summary>
    /// Function to generate unique token with expiry against the provided userId.
    /// Also add a record in database for generated token.
    /// </summary>
    /// <param name="userId"></param>
    /// <returns></returns>
    TokenEntity GenerateToken(int userId);

    /// <summary>
    /// Function to validate token against expiry and existence in database.
    /// </summary>
    /// <param name="tokenId"></param>
    /// <returns></returns>
    bool ValidateToken(string tokenId);

    /// <summary>
    /// Method to kill the provided token id.
    /// </summary>
    /// <param name="tokenId"></param>
    bool Kill(string tokenId);

    /// <summary>
    /// Delete tokens for the specific deleted user
    /// </summary>
    /// <param name="userId"></param>
    /// <returns></returns>
    bool DeleteByUserId(int userId);
    #endregion
}
}

```

We have four methods defined in this interface. Let's create TokenServices class which implements ITokenServices and understand each method.

GenerateToken method takes userId as a parameter and generates a token, encapsulates that token in a token entity with Token expiry time and returns it to caller.

```

public TokenEntity GenerateToken(int userId)
{
    string token = Guid.NewGuid().ToString();
    DateTime issuedOn = DateTime.Now;
    DateTime expiredOn = DateTime.Now.AddSeconds(
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
    var tokendomain = new Token
    {
        UserId = userId,
        AuthToken = token,
        IssuedOn = issuedOn,
        ExpiresOn = expiredOn
    };

    _unitOfWork.TokenRepository.Insert(tokendomain);
    _unitOfWork.Save();
}

```



```

        var tokenModel = new TokenEntity()
        {
            UserId = userId,
            IssuedOn = issuedOn,
            ExpiresOn = expiredOn,
            AuthToken = token
        };

        return tokenModel;
    }

```

While generating token, it names a database entry into Token table.

ValidateToken method just validates that the token associated with the request is valid or not i.e. it exists in the database within its expiry time limit.

```

public bool ValidateToken(string tokenId)
{
    var token = _unitOfWork.TokenRepository.Get(t => t.AuthToken == tokenId &&
t.ExpiresOn > DateTime.Now);
    if (token != null && !(DateTime.Now > token.ExpiresOn))
    {
        token.ExpiresOn = token.ExpiresOn.AddSeconds(
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
        _unitOfWork.TokenRepository.Update(token);
        _unitOfWork.Save();
        return true;
    }
    return false;
}

```

It just takes token Id supplied in the request.

Kill Token just kills the token i.e. removes the token from database.

```

public bool Kill(string tokenId)
{
    _unitOfWork.TokenRepository.Delete(x => x.AuthToken == tokenId);
    _unitOfWork.Save();
    var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.AuthToken ==
tokenId).Any();
    if (isNotDeleted) { return false; }
    return true;
}

```

DeleteByUserId method deletes all token entries from the database w.r.t particular userId associated with those tokens.

```

public bool DeleteByUserId(int userId)
{
    _unitOfWork.TokenRepository.Delete(x => x.UserId == userId);
    _unitOfWork.Save();

    var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.UserId == userId).Any();
}

```

```

return !isNotDeleted;
}

```

So with `_unitOfWork` and along with Constructor our class becomes,

```

using System;
using System.Configuration;
using System.Linq;
using BusinessEntities;
using DataModel;
using DataModel.UnitOfWork;

namespace BusinessServices
{
    public class TokenServices:ITokenServices
    {
        #region Private member variables.
        private readonly UnitOfWork _unitOfWork;
        #endregion

        #region Public constructor.
        /// <summary>
        /// Public constructor.
        /// </summary>
        public TokenServices(UnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }
        #endregion

        #region Public member methods.

        /// <summary>
        /// Function to generate unique token with expiry against the provided userId.
        /// Also add a record in database for generated token.
        /// </summary>
        /// <param name="userId"></param>
        /// <returns></returns>
        public TokenEntity GenerateToken(int userId)
        {
            string token = Guid.NewGuid().ToString();
            DateTime issuedOn = DateTime.Now;
            DateTime expiredOn = DateTime.Now.AddSeconds(
                Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
            var tokendomain = new Token
            {
                UserId = userId,
                AuthToken = token,
                IssuedOn = issuedOn,
                ExpiresOn = expiredOn
            };

            _unitOfWork.TokenRepository.Insert(tokendomain);
            _unitOfWork.Save();
            var tokenModel = new TokenEntity()

```

```

{
    UserId = userId,
    IssuedOn = issuedOn,
    ExpiresOn = expiredOn,
    AuthToken = token
};

return tokenModel;
}

/// <summary>
/// Method to validate token against expiry and existence in database.
/// </summary>
/// <param name="tokenId"></param>
/// <returns></returns>
public bool ValidateToken(string tokenId)
{
    var token = _unitOfWork.TokenRepository.Get(t => t.AuthToken == tokenId && t.ExpiresOn >
    DateTime.Now);
    if (token != null && !(DateTime.Now > token.ExpiresOn))
    {
        token.ExpiresOn = token.ExpiresOn.AddSeconds(
        Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
        _unitOfWork.TokenRepository.Update(token);
        _unitOfWork.Save();
        return true;
    }
    return false;
}

/// <summary>
/// Method to kill the provided token id.
/// </summary>
/// <param name="tokenId">true for successful delete</param>
public bool Kill(string tokenId)
{
    _unitOfWork.TokenRepository.Delete(x => x.AuthToken == tokenId);
    _unitOfWork.Save();
    var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.AuthToken == tokenId).Any();
    if (isNotDeleted) { return false; }
    return true;
}

/// <summary>
/// Delete tokens for the specific deleted user
/// </summary>
/// <param name="userId"></param>
/// <returns>true for successful delete</returns>
public bool DeleteByUserId(int userId)
{
    _unitOfWork.TokenRepository.Delete(x => x.UserId == userId);
    _unitOfWork.Save();

    var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.UserId == userId).Any();
    return !isNotDeleted;
}

```

```
#endregion
```

```
}  
}
```

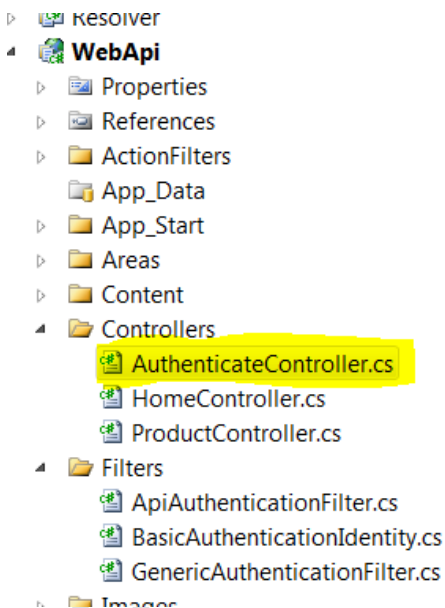
Do not forget to resolve the dependency of this Token service in DependencyResolver class. Add `registerComponent.RegisterType<ITokenServices, TokenServices>();` to the `SetUp` method of `DependencyResolver` class in `BusinessServices` project.

```
[Export(typeof(IComponent))]  
public class DependencyResolver : IComponent  
{  
    public void SetUp(IRegisterComponent registerComponent)  
    {  
        registerComponent.RegisterType<IProductServices, ProductServices>();  
        registerComponent.RegisterType<IUserServices, UserServices>();  
        registerComponent.RegisterType<ITokenServices, TokenServices>();  
    }  
}
```

Setup WebAPI/Controller:

Now since we decided, that we don't want authentication to be applied on each and every API exposed, I'll create a single Controller/API endpoint that takes authentication or login request and makes use of Token Service to generate token and respond client/caller with a token that persists in database with expiry details.

Add a new Controller under `Controllers` folder in `WebAPI` with a name `Authenticate`,



AuthenticateController:

```
using System.Configuration;  
using System.Net;  
using System.Net.Http;  
using System.Web.Http;
```

```

using AttributeRouting.Web.Http;
using BusinessServices;
using WebApi.Filters;

namespace WebApi.Controllers
{
    [ApiAuthenticationFilter]
    public class AuthenticateController : ApiController
    {
        #region Private variable.

        private readonly ITokenServices _tokenServices;

        #endregion

        #region Public Constructor

        /// <summary>
        /// Public constructor to initialize product service instance
        /// </summary>
        public AuthenticateController(ITokenServices tokenServices)
        {
            _tokenServices = tokenServices;
        }

        #endregion

        /// <summary>
        /// Authenticates user and returns token with expiry.
        /// </summary>
        /// <returns></returns>
        [POST("login")]
        [POST("authenticate")]
        [POST("get/token")]
        public HttpResponseMessage Authenticate()
        {
            if (System.Threading.Thread.CurrentPrincipal != null &&
                System.Threading.Thread.CurrentPrincipal.Identity.IsAuthenticated)
            {
                var basicAuthenticationIdentity =
                    System.Threading.Thread.CurrentPrincipal.Identity as BasicAuthenticationIdentity;
                if (basicAuthenticationIdentity != null)
                {
                    var userId = basicAuthenticationIdentity.UserId;
                    return GetAuthToken(userId);
                }
            }
            return null;
        }

        /// <summary>
        /// Returns auth token for the validated user.
        /// </summary>
        /// <param name="userId"></param>
        /// <returns></returns>
        private HttpResponseMessage GetAuthToken(int userId)

```

```

    {
        var token = _tokenServices.GenerateToken(userId);
        var response = Request.CreateResponse(HttpStatusCode.OK, "Authorized");
        response.Headers.Add("Token", token.AuthToken);
        response.Headers.Add("TokenExpiry",
ConfigurationManager.AppSettings["AuthTokenExpiry"]);
        response.Headers.Add("Access-Control-Expose-Headers", "Token,TokenExpiry" );
        return response;
    }
}
}

```

The controller is decorated with our authentication filter,

```

[ApiAuthenticationFilter]
public class AuthenticateController : ApiController

```

So, each and every request coming through this controller will have to pass through this authentication filter, that check for BasicAuthentication header and credentials. Authentication filter sets CurrentThread principal to the authenticated Identity.

There is a single Authenticate method / action in this controller. You can decorate it with multiple endpoints like we discussed in [fourth](#) part of the series,

```

[POST("login")]
[POST("authenticate")]
[POST("get/token")]

```

Authenticate method first checks for CurrentThreadPrincipal and if the user is authenticated or not i.e job done by authentication filter,

```

if (System.Threading.Thread.CurrentPrincipal!=null &&
System.Threading.Thread.CurrentPrincipal.Identity.IsAuthenticated)

```

When it finds that the user is authenticated, it generates an auth token with the help of TokenServices and returns user with Token and its expiry,

```

response.Headers.Add("Token", token.AuthToken);
response.Headers.Add("TokenExpiry", ConfigurationManager.AppSettings["AuthTokenExpiry"]);
response.Headers.Add("Access-Control-Expose-Headers", "Token,TokenExpiry" );
return response;

```

In our BasicAuthenticationIdentity class, I purposely used userId property so that we can make use of this property when we try to generate token, that we are doing in this controller's Authenticate method,

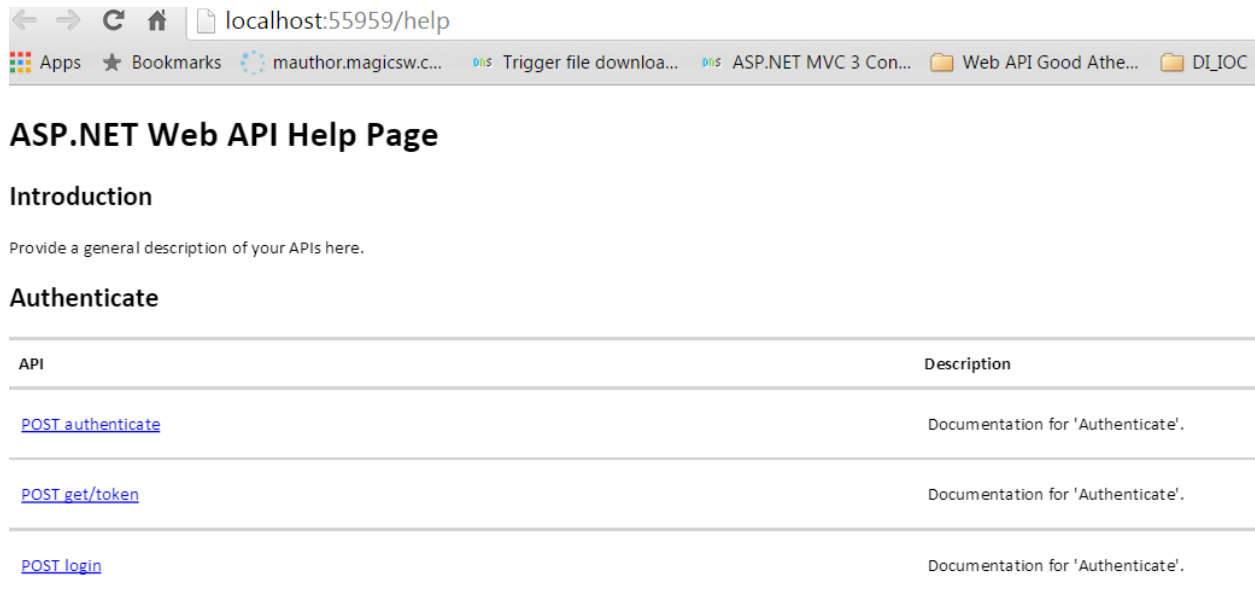
```

var basicAuthenticationIdentity = System.Threading.Thread.CurrentPrincipal.Identity as
BasicAuthenticationIdentity;
if (basicAuthenticationIdentity != null)
{
    var userId = basicAuthenticationIdentity.UserId;
    return GetAuthToken(userId);
}

```

Now when you run this application, You'll see Authenticate api as well, just invoke this api with Basic Authentication and User credentials, you'll get the token with expiry, let's do this step by step.

1. Run the application.



The screenshot shows a web browser window with the address bar at `localhost:55959/help`. The page title is "ASP.NET Web API Help Page". Under the "Introduction" section, it says "Provide a general description of your APIs here." The "Authenticate" section contains a table with two columns: "API" and "Description".

API	Description
POST authenticate	Documentation for 'Authenticate'.
POST get/token	Documentation for 'Authenticate'.
POST login	Documentation for 'Authenticate'.

2. Click on first api link i.e. [POST authenticate](#). You'll get the page to test the api,

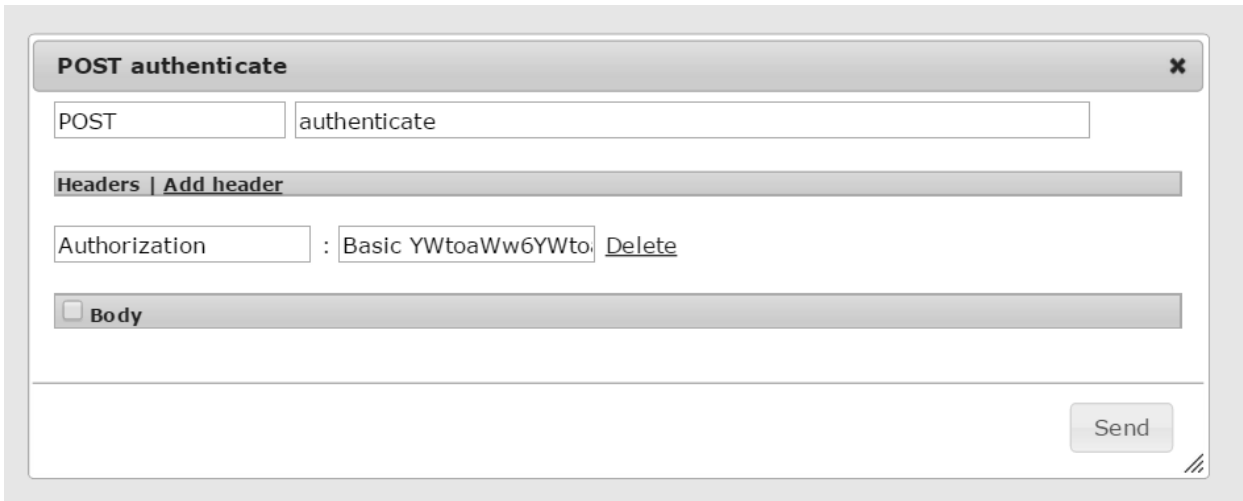
[Help Page Home](#)

POST authenticate

Documentation for 'Authenticate'.

Test API

3. Press the TestAPI button in the right corner. In the test console, provide Header information with Authorization as Basic and user credentials in Base64 format, like we did earlier. Click on Send.



POST authenticate [X]

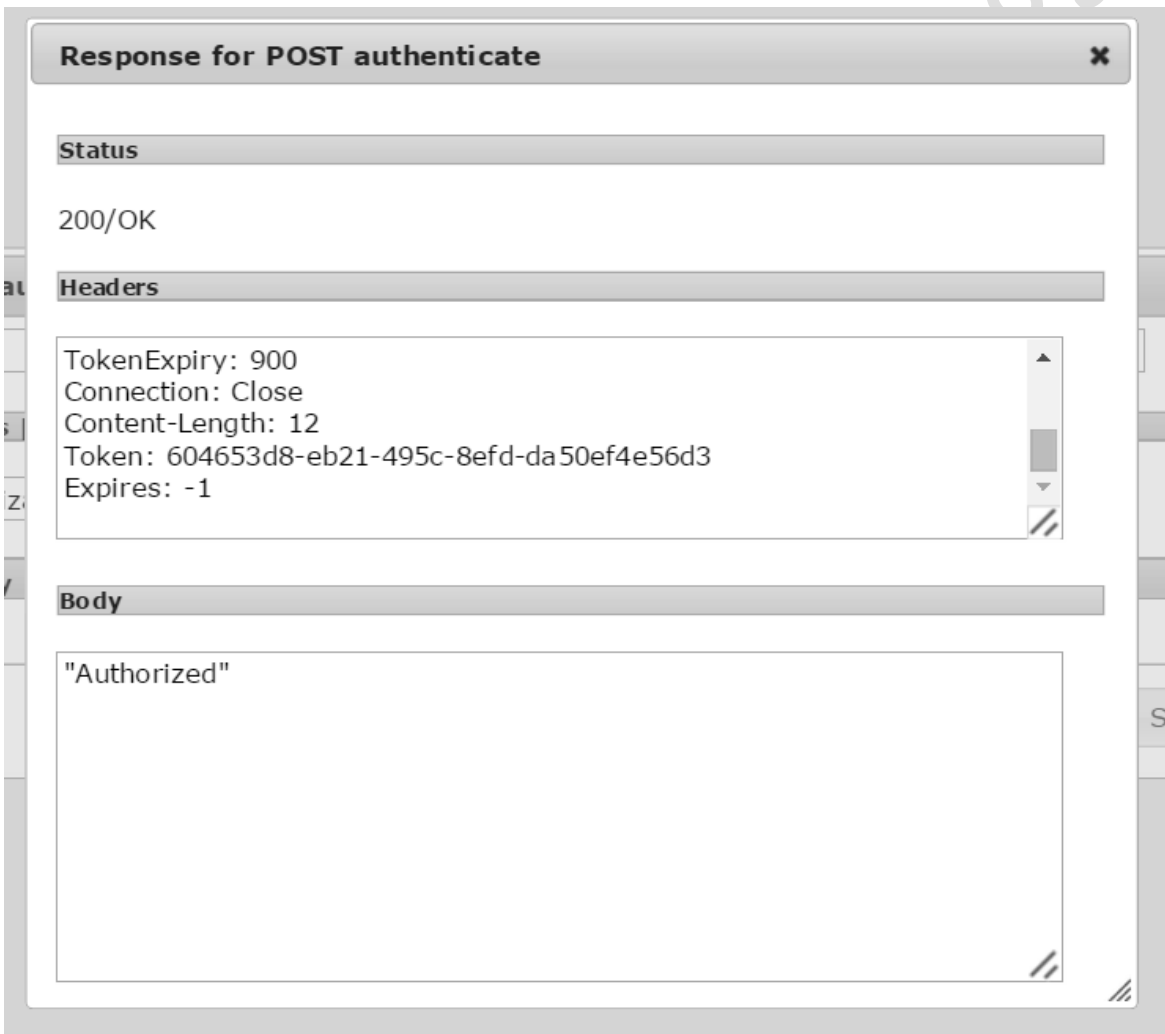
POST

Headers | [Add header](#)

: [Delete](#)

☐ **Body**

4. Now since we have provided valid credentials ,we'll get a token from the Authenticate controller, with its expiry time,



Response for POST authenticate [X]

Status

200/OK

Headers

TokenExpiry: 900
Connection: Close
Content-Length: 12
Token: 604653d8-eb21-495c-8efd-da50ef4e56d3
Expires: -1

Body

"Authorized"

In database,

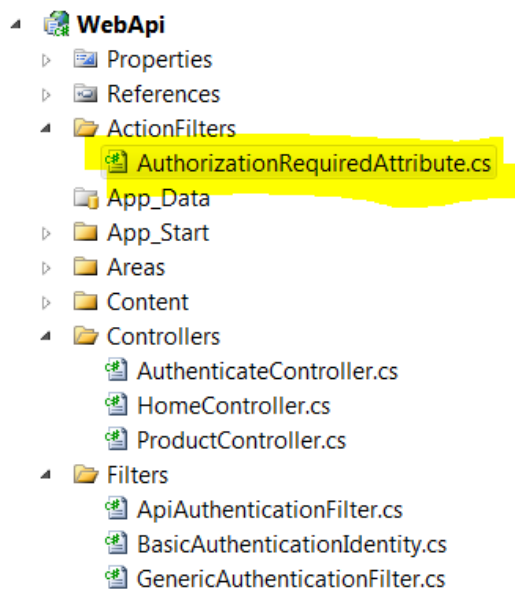
69B5ZR1.WebApiDb - dbo.Tokens SQLQuery1.sql - 6...aster (akhil (53))					
	TokenId	UserId	AuthToken	IssuedOn	ExpiresOn
▶	1	1	604653d8-eb21-495c-8efd-da50ef4e56d3	2015-06-23 13:54:21.520	2015-06-23 14:09:21.520
*	NULL	NULL	NULL	NULL	NULL

Here we get response 200, i.e. our user is authenticated and logged into system. TokenExpiry in 900 i.e. 15 minutes. Note that the time difference between IssuedOn and ExpiresOn is 15 minutes, this we did in TokenServices class method GenerateToken, you can set the time as per your need. Token is 604653d8-eb21-495c-8efd-da50ef4e56d3. Now for 15 minutes we can use this token to call our other services. But before that we should mark our other services to understand this token and respond accordingly. Keep the generated token saved so that we can use it further in calling other services that I am about to explain. So let's setup authorization on other services.

Setup Authorization Action Filter

We already have our Authentication filter in place and we don't want to use it for authorization purposes. So we have to create a new Action Filter for authorization. This action filter will only recognize Token coming in requests. It assumes that, requests are already authenticated through our login channel, and now user is authorized/not authorized to use other services like Products in our case, there could be n number of other services too, which can use this authorization action filter. For request to get authorized, now we don't have to pass user credentials. Only token(received from Authenticate controller after successful validation) needs to be passed through request.

Add a folder named ActionFilters in WebAPI project. And add a class named [AuthorizationRequiredAttribute](#) Deriving from [ActionFilterAttribute](#),



Override the OnActionExecuting method of ActionFilterAttribute, this is the way we define an action filter in API project.

```
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http.Controllers;
using System.Web.Http.Filters;
using BusinessServices;
```

```

namespace WebApi.ActionFilters
{
    public class AuthorizationRequiredAttribute : ActionFilterAttribute
    {
        private const string Token = "Token";

        public override void OnActionExecuting(HttpContext filterContext)
        {
            // Get API key provider
            var provider = filterContext.ControllerContext.Configuration
                .DependencyResolver.GetService(typeof(ITokenServices)) as ITokenServices;

            if (filterContext.Request.Headers.Contains(Token))
            {
                var tokenValue = filterContext.Request.Headers.GetValues(Token).First();

                // Validate Token
                if (provider != null && !provider.ValidateToken(tokenValue))
                {
                    var responseMessage = new HttpResponseMessage(HttpStatusCode.Unauthorized) { ReasonPhrase = "Invalid Request" };
                    filterContext.Response = responseMessage;
                }
            }
            else
            {
                filterContext.Response = new HttpResponseMessage(HttpStatusCode.Unauthorized);
            }

            base.OnActionExecuting(filterContext);
        }
    }
}

```

The overridden method checks for “Token” attribute in the Header of every request, if token is present, it calls ValidateToken method from TokenServices to check if the token exists in the database. If token is valid, our request is navigated to the actual controller and action that we requested, else you’ll get an error message saying unauthorized.

Mark Controllers with Authorization filter

We have our action filter ready. Now let’s mark our controller ProductController with this attribute. Just open Product controller class and at the top just decorate that class with this ActionFilter attribute,

```

[AuthorizationRequired]
[RoutePrefix("v1/Products/Product")]
public class ProductController : ApiController
{

```

We have marked our controller with the action filter that we created, now every request coming to the actions of this controller will have to be passed through this ActionFilter, that checks for the token in request.

You can mark other controllers as well with the same attribute, or you can do marking at action level as well. Suppose you want certain actions should be available to all users irrespective of their authorization then you can just mark only those actions which require authorization and leave other actions as they are like I explained in Step 4 of Implementing Basic Authentication.

Maintaining Session using Token

We can certainly use these tokens to maintain session as well. The tokens are issues for 900 seconds i.e. 15 minutes. Now we want that user should continue to use this token if he is using other services as well for our application. Or suppose there is a case where we only want user to finish his work on the site within 15 minutes or within his session time before he makes a new request. So while validating token in TokenServices, what I have done is, to increase the time of the token by 900 seconds whenever a valid request comes with a valid token,

```
/// <summary>
/// Method to validate token against expiry and existence in database.
/// </summary>
/// <param name="tokenId"></param>
/// <returns></returns>
public bool ValidateToken(string tokenId)
{
    var token = _unitOfWork.TokenRepository.Get(t => t.AuthToken == tokenId &&
t.ExpiresOn > DateTime.Now);
    if (token != null && !(DateTime.Now > token.ExpiresOn))
    {
        token.ExpiresOn = token.ExpiresOn.AddSeconds(
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
        _unitOfWork.TokenRepository.Update(token);
        _unitOfWork.Save();
        return true;
    }
    return false;
}
```

In above code for token validation, first we check if the requested token exists in the database and is not expired. We check expiry by comparing it with current date time. If it is valid token we just update the token into database with a new ExpiresOn time that is adding 900 seconds.

```
if (token != null && !(DateTime.Now > token.ExpiresOn))
{
    token.ExpiresOn = token.ExpiresOn.AddSeconds(
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
    _unitOfWork.TokenRepository.Update(token);
    _unitOfWork.Save();
}
```

By doing this we can allow end user or client to maintain session and keep using our services/application with a session timeout of 15 minutes. This approach can also be leveraged in multiple ways, like making different services with different session timeouts or many such conditions could be applied when we work on real time application using APIs.

Running the application

Our job is almost done.



We just need to run the application and test if it is working fine or not. If you have saved the token you generated earlier while testing authentication you can use same to test authorization. I am just again running the whole cycle to test application.

Test Authentication

Repeat the tests we did earlier to get Auth Token. Just invoke the Authenticate controller with valid credentials and Basic authorization header. I get,

Response for POST authenticate ✕

Status

200/OK

Headers

TokenExpiry: 900
Connection: Close
Content-Length: 12
Token: d843cdef-6858-4651-8407-65e4cb61b3b7
Expires: -1

Body

"Authorized"

And without providing Authorization header as basic with credentials I get,



I just saved the token that I got in first request.

Now try to call ProductController actions.

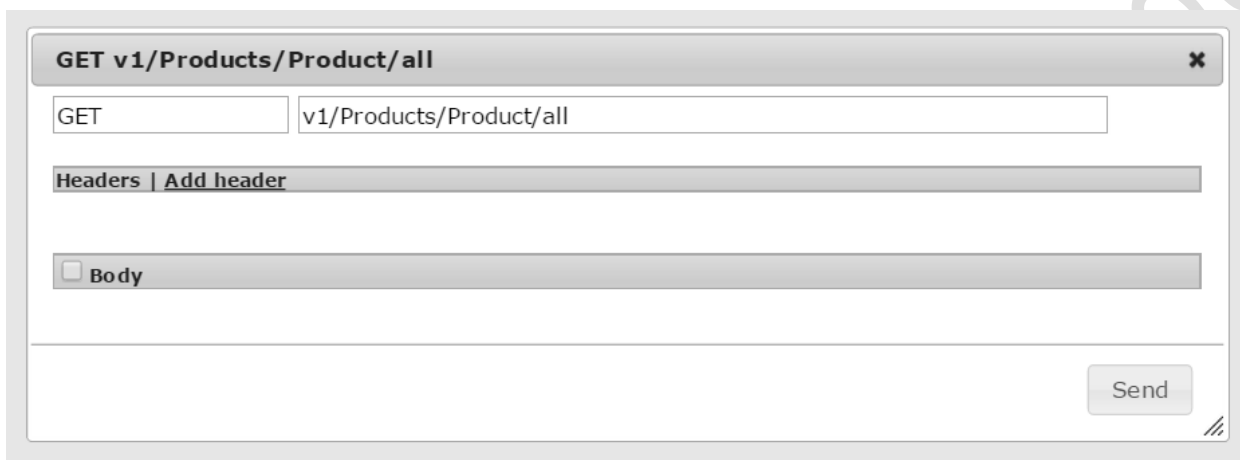
Test Authorization

Run the application to invoke Product Controller actions. Try to invoke them without providing any Token,

Product

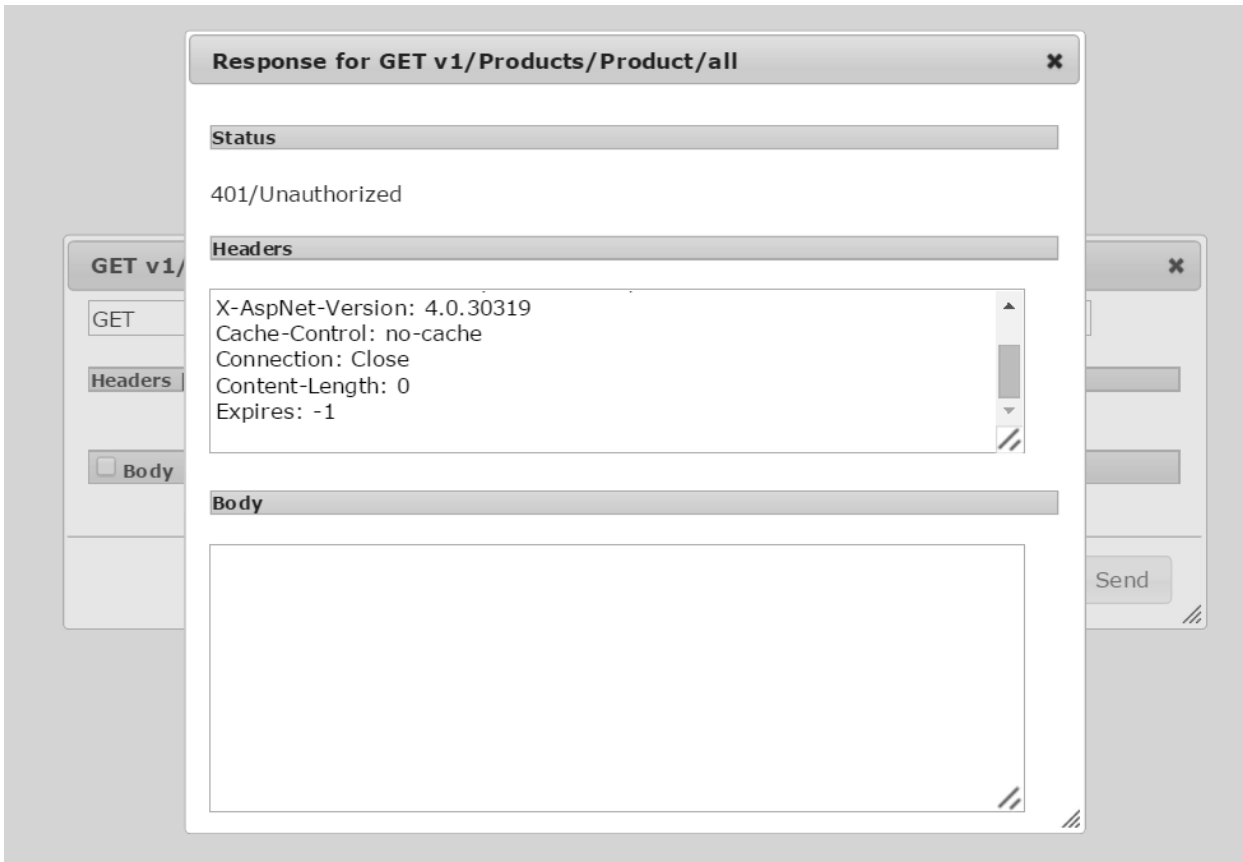
API	Description
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.

Invoke first service in the list,

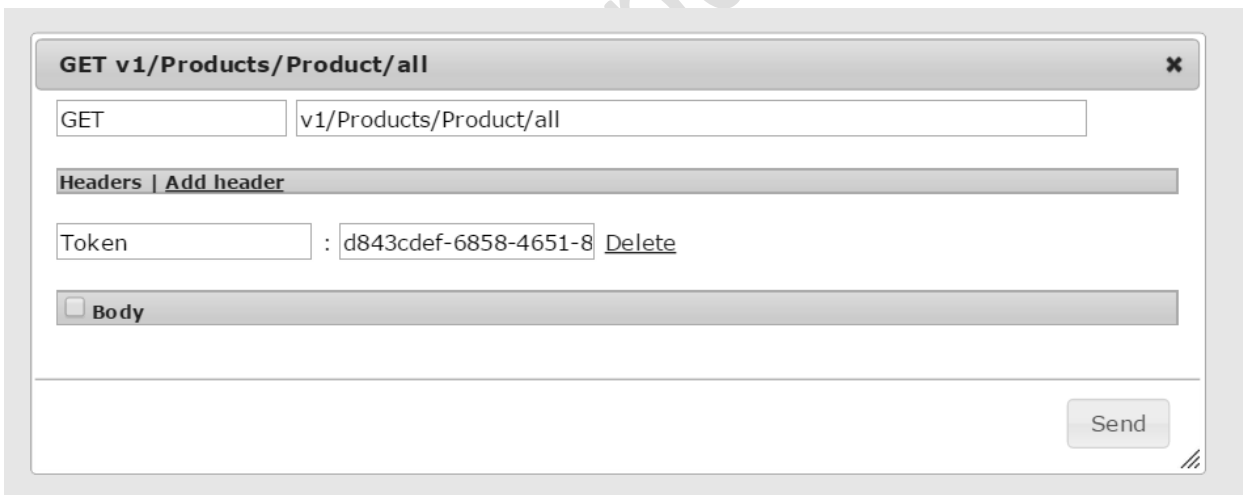


The screenshot shows an API client window titled "GET v1/Products/Product/all". It features a dropdown menu set to "GET" and a text input field containing the URL "v1/Products/Product/all". Below the URL field, there is a section for "Headers" with an "Add header" button. A "Body" section is also present but is currently unchecked. At the bottom right of the window, there is a "Send" button and a small icon representing a document or list.

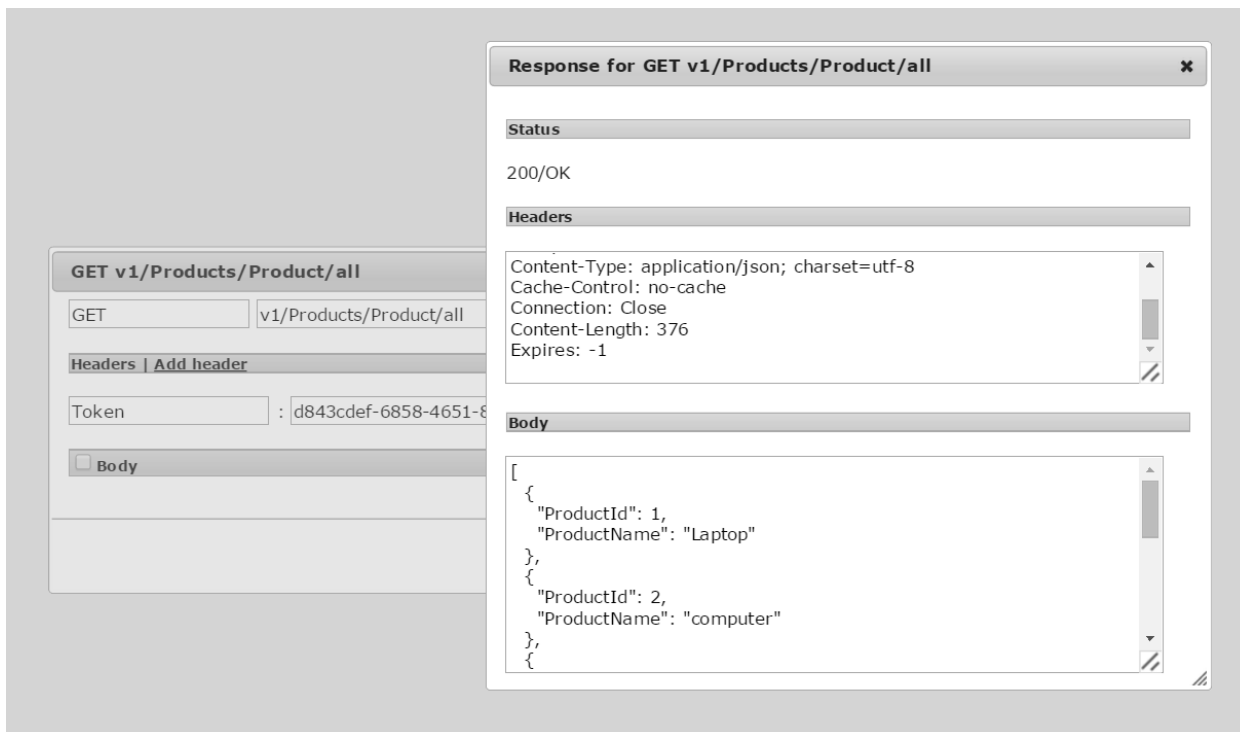
Click send,



Here we get Unauthorized, i.e. because our ProductController is marked with authorization attribute that checks for a Token. So here our request is invalid. Now try calling this action by providing the token that we saved,



Click on Send and we get,



That means we got the response and our token was valid. Now we see our Authentication and Authorization, both functionalities are working fine. You can test Sessions by your own.



Likewise you can test all actions. You can create other controllers and test the security, and play around with different set of permutations and combinations.

Conclusion

We covered and learnt a lot. In this article I tried to explain about how we can build an API application with basic Authentication and Authorization. One can mould this concept to achieve the level of security needed. Like token generation mechanism could be customized as per one's requirement. Two level of security could be implemented where authentication and authorization is needed for every service. One can also implement authorization on actions based on Roles.



Image credit: http://www.greencountryfordofparsons.com/images/secure_application.jpg

I already stated that there is no specific way of achieving security, the more you understand the concept, the more secure system you can make. The techniques used in this article or the design implemented in this article should be leveraged well if you use it with SSL (Secure Socket Layer), running the REST apis on https. In my next article I'll try to explain some more beautiful implementations and concepts. Till then Happy Coding ☺
You can also download the complete source code with all packages from [Github](#).

References

<https://msdn.microsoft.com/en-us/magazine/dn781361.aspx>

<http://weblog.west-wind.com/posts/2013/Apr/18/A-WebAPI-Basic-Authentication-Authorization-Filter>

Other Series

My other series of articles:

MVC: <http://www.codeproject.com/Articles/620195/Learning-MVC-Part-Introduction-to-MVC-Architectu>

OOP: <http://www.codeproject.com/Articles/771455/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Ear>

For more informative articles visit my [blog](#).