****

**Basic Authentication and Authorization and JWT Authentication**

Date: 23-05-2018

Name: Shamil Sulaiman(M1044312) & Shashanka(M1044327) & Brahmaiah Shaik

Project: The Sanctuary & DriveSafe

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Authentication**:

In ASP.NET, authentication means to identify the user or in other words its nothing but to validate that he exists in the database and he is a proper user. i.e., Authentication is the process of obtaining some sort of credentials from the users and using those credentials to verify the user's identity.

**Types of Authentication**:

1. Windows Authentication
2. Forms Authentication
3. Passport Authentication
4. Anonymous Access

**Windows Authentication**:-

In this methodology ASP.NET web pages will use local windows users and groups to authenticate and authorize resources. i.e., when you configure your ASP.NET application as windows authentication it will use local windows user and groups to do authentication and authorization for your ASP.NET pages.

**Forms Authentication**:-

Forms authentication is a cookie/URL based authentication where username and password are stored on client machines as cookie files. So, the user enters “**userid**” and “**password**” through a custom login screen developed for authentication and authorization. Then, this userid and password is validated from the custom database and if the user is valid then a cookie text file is generated on the client end. This cookie test file signifies that the

user has been authenticated. Hence forth when the client computer browses other resources of your ASP.NET site, this cookie file will indicates that the user has logged in.

**Passport Authentication**:-

Passport authentication is based on the passport website provided  
by the Microsoft. So when user logins with credentials it will be reached to the passport website (for e.g.: Gmail) where authentication will happen. If Authentication is successful it will return a token to our website.

**Anonymous Access**:-

If we do not want any kind of authentication then we will go for Anonymous access (for e.g. a Guest user).

**Authorization:**

Authorization is the process of allowing an authenticated users to access the resources by checking whether the user has access rights to the system. So, Authentication is always precedes to Authorization; even if your application lets anonymous users connect and use the application, it still authenticates them as being anonymous. Authorization helps you to control access rights by granting or denying specific permissions to an authenticated user.

In ASP.NET, there are two ways to authorize access to a given resource:

1. File Authorization
2. URL Authorization

**File Authorization:-**

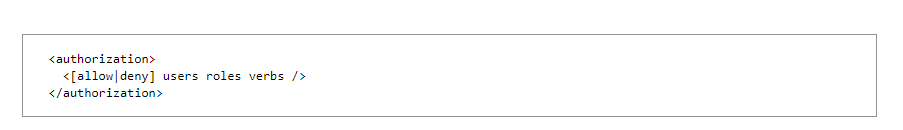
File authorization is performed by the FileAuthorizationModule. It validate the ACL (access control list) of the .aspx or .asmx handler file to determine whether a user should have access to the file.

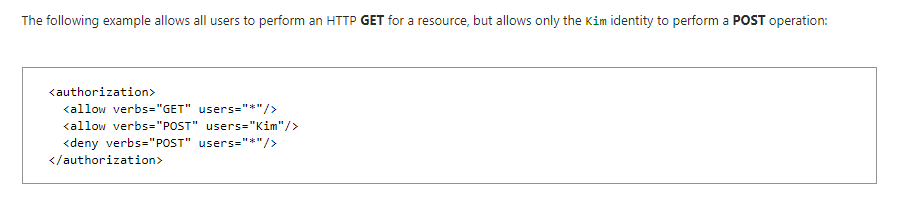
**URL Authorization:-**

URL authorization is performed by the UrlAuthorizationModule, which maps users and roles to URLs in ASP.NET applications. This module can be used to selectively allow or deny access to arbitrary parts of an application for specific users or roles.

With URL authorization, we explicitly allow or deny access to a particular directory by user name or role. To do so, we create an authorization section in the configuration file for that directory. To enable URL authorization, we specify a list of users or roles in the allow or deny elements of the authorization section of a configuration file as shown below:

The following shows the syntax for the authorization section:





**Steps to Implement Basic Authentication :**

In order to implement basic authentication, the steps are listed below.

Step1:

Method to validate a user

Add a class called ApiSecurity and add a method called ValidateUser(string username, string password), which takes two parameters - username and password. It checks the username and password with the database value, if it succeeds it returns boolean value as true, else false.

|  |
| --- |
| public static bool VaidateUser(string username, string password)  {  // Check if it is valid credential  if(true)//CheckUserInDB(username, password))  {  return true;  }  else  {  return false;  }  } |

Step2:

In the second step, add a class, which will used as Authorization filter. The class BasicAuthenticationAttribute inherits from BasicAuthenticationAttribute abstarct class.

It contains an override method *OnAuthorization*(), which performs all the validations. Inside method checks whether the header is present or not: if no, it sends an unauthorized, else it goes ahead to gets the values from the header. Here, we are using 64 bit encoding format to encrypt the username/password. Once you get the value from the header, it converts to original string, which contains the username and the password. Subsequently, it calls the *VaidateUser*() of ApiSecurity class(discussed in Step 1) with passing the required parameters to get the Boolean result. If it returns false, it sends an unauthorized error to the user.

|  |
| --- |
| public class BasicAuthenticationAttribute : AuthorizationFilterAttribute  {  public override void OnAuthorization(System.Web.Http.Controllers.HttpActionContext actionContext)  {  if (actionContext.Request.Headers.Authorization == null)  {  actionContext.Response = actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized);  }  else  {  // Gets header parameters  string authenticationString = actionContext.Request.Headers.Authorization.Parameter;  string originalString = Encoding.UTF8.GetString(Convert.FromBase64String(authenticationString));  // Gets username and password  string usrename = originalString.Split(':')[0];  string password = originalString.Split(':')[1];  // Validate username and password  if (!ApiSecurity.VaidateUser(usrename, password))  {  // returns unauthorized error  actionContext.Response = actionContext.Request.CreateResponse(HttpStatusCode.Unauthorized);  }  }  base.OnAuthorization(actionContext);  }  } |

**Step3**Our authorization filter is ready and we need to register it. You can register at global, controller or action level. Here, we have added for controller level.

|  |
| --- |
| [BasicAuthentication]  public class BlogController : ApiController  {  // Add your action here  } |

**Note**You can add the filter in either WebApiConfig or FilterConfig class file.

* In WebApiConfig.cs  
  config.Filters.Add(new BasicAuthenticationAttribute());
* In FilterConfig.cs  
  filters.Add(new BasicAuthenticationAttribute());

**JSON WEB TOKENS:-**

A JSON Web Token, or JWT, is a cryptographically signed token which can be used to send information that can be verified and trusted by means of a digital signature. This token can be verified against the signature to verify its authenticity. Since the token is encrypted we can include sensitive information like authorization headers in the payload.

Characteristics of JWT:

JWTs are self-contained: A JWT mainly has 3 parts, basic information about the token itself, a payload which can be sensitive information, and a signature. All these information are passed along with the JWT itself.

JWTs can be passed around easily: Since JWTs are self-contained, and encrypted they can be easily transmitted inside a HTTP header.

### JSON Web Token structure

JSON Web Tokens are self-contained and consist of three parts separated by dots (.), which are:

* **Header**
* **Payload**
* **Signature**

Therefore, a typical JWT looks like the following.

header.payload.signature

##### **Header**

The header part of a JWT mainly consists of information related to the token itself. It has two parts:

* the type of the token, which is JWT,
* the hashing algorithm being used, such as HMAC SHA256 or RSA.
* {
* "alg": "HS256",
* "typ": "JWT"
* }

##### **Payload**

Payload is the second part of the JWT. It may include confidential information like user details, any other additional metadata etc. These are also called claims.

There are three types of claims: reserved, public, and private claims.

###### Reserved claims:

The standard defines a set of predefined claims. These are are not mandatory but hightly recommended.

###### Public claims:

Any custom claims can be defined in the public claims section.

###### Private claims:

These are the custom claims created to share information between parties that agree on using them.

##### **Signature**

To create the signature for a JWT, we first need to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and using that sign that.

The commonly used algorithm is HMAC SHA256.

### When should you use JSON Web Tokens?

Some of the most common use case of JWT are authentication and information exchange.

### Advantages of Token-Based Authentication and JWT

**Stateless, easier to scale:** The JWT contains all the necessary information to identify a particular user. It can be sent to any server among a cluster of servers under a load balancer, not necessarily to the server that issued the token, or the server that the user logged in initially with.

**Reusability:** We can reuse the token generated across multiple servers and multiple platforms and domains. Applications using **Single Sign On (SSO)** are typical examples.

Security: Since the tokens are signed we can transmit the information securely over HTTP. Additionally, we can encrypt the tokens using JWE and also use an SSL and transmit over HTTPS

Performance: We can completely avoid deserializing the sessions back on the server on each request. We can use any of the servers to authenticate the token, validate it and parse the contents

**Code Snippet for Generating JSON WEB Token:**

|  |
| --- |
| public static string GenerateToken(User user) {  var symmetricKey = Convert.FromBase64String(Secret);  var tokenHandler = new JwtSecurityTokenHandler();  var tokenDescriptor = new SecurityTokenDescriptor  {  Subject = new ClaimsIdentity(new[]  {  new Claim(ClaimTypes.Name, user.FirstName),  new Claim(ClaimTypes.Email, user.Email)  }),  SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(symmetricKey), SecurityAlgorithms.HmacSha256Signature)  };  var stoken = tokenHandler.CreateToken(tokenDescriptor);  var token = tokenHandler.WriteToken(stoken);  return token;  } |