Web application security is always an important part of how you design and implement a solution. For many years, the standard method for securing a C# ASP .NET MVC web application was to use session for storing the user object, in combination with traditional .NET forms-based authentication cookies. This has worked well for many years, and still does. However, more recently, single page web applications have gained significant traction. Since these types of web applications run almost entirely in the client web browser, they often call a REST Web API or .NET MVC controller, serving as a web service API, in order to get and send data to display in the views. In this scenario, a large portion of the business logic has moved from the server-side to the client-side. Still, the API calls to access data need to be secured.

In this tutorial, we’ll walk through how to create a simple, but effective token-based authentication framework to secure a .NET REST API. The framework is based upon a detailed post, using the hash-based message authentication code (HMAC).

Our API will be created from a simple MVC controller. Each call on the API will be protected by token-based authentication, which requires the client to provide a token key in the URL for each request. Each token will be unique and have a limited duration of time that it is valid. We’ll complete the authentication framework by including client-side Javascript for generating the token and making calls to the .NET API service.

Migrating Away From Session

What’s so bad about .NET session anyway? Session allows you to store objects on the server, in memory, or persisted on a database. Aside from the potential clutter it can cause if mismanaged, it has a more recent shortcoming, largely involving the cloud.

Companies are migrating to the cloud en masse. Along with this migration, certain features used by traditional C# ASP .NET MVC web applications are becoming more difficult to migrate. One of these is session-backed storage.

Cloud servers, by nature, are often temporary instances. They can be transparently brought up and taken down, sometimes without notice. When a server instance is removed or re-created, it can lose persisted data, which includes session storage - both in memory and on a database. While there are workarounds for migrating session storage to cloud compatibility, many companies are choosing to re-architect their web applications to a REST design, providing web service API calls to the user interface. Along with this change, they’re choosing to move away from session, and instead use token-based authentication.

A Brief Overview of Token-Based Authentication

Token-based authentication involves providing a token or key in the url or HTTP request header, which contains all necessary information to validate a user’s request. Some examples of information included in the token are username, timestamp, ip address, and any other information pertinent towards checking if a request should be honored.

In this tutorial, we’ll focus on a simple implementation of token-based authentication. Let’s see how it works.

Generating a Token

First, we’ll need to decide exactly what we want in our token. We’ll use the following schema:

A token is generated by hashing a message with a key, using HMAC SHA256.

The message will consist of: username:ip:userAgent:timeStamp

The key will consist of a hashed combination of: password:salt

We include the client IP address and user-agent string as part of the message in order to bind a token to a specific client. This helps prevent an attacker from replaying a token request, as both pieces of data are verified at the server and must match the requesting client information. If the client web browser making the API request is different than the one that generated the token, then the token will fail to authenticate.

The resulting token is concatenated with username:timeStamp and the final result is base64-encoded. This final concatenation is necessary so that we are able to validate the timestamp, as well as check the user’s credentials on the server.

A Note on Token Strength .. and Weakness

It’s important to keep in mind the strength and weakness of a token-based system. The most important rule is - never transmit the token key across the wire. That is, you select a key that will be used to generate a token for each request. This key must never be sent to the server and must always remain on the client (ie., held in javascript memory). With that in mind, here are some descriptions of potential weaknesses for token-based authentication.

There are several vectors of attack that need to be considered. The first, is a simple replay attack or man-in-the-middle attack. This involves an attacker capturing a token API request and replaying the same exact request again. We can prevent this type of attack by validating client-specific data as part of the token (IP address and user-agent string). In addition, adding a token expiration date helps to limit the duration that such an attack is viable. You could further prevent this type of attack by keeping a server log (MemoryCache, etc) of recently used tokens and invalidate them once used. Depending on how short the token expiration time is (5-10 minutes), invalidation may not be necessary.

Of course, SSL is another way to help protect tokens. SSL encrypts url query string parameters and post data. Since the token is included as a parameter within the url, SSL can be an effective tool to boost token authentication security.

A second type of attack is physically obtaining the token key. Since the javascript that generates a token is publicly available, an attacker could attempt to generate his own token, impersonating another user. To do this, he would need to provide another user’s username as part of the token body. However, this requires using the user’s key to hash the token message. To generate the user’s key, an attacker would need either the user’s password or their actual key. We’ll have to assume that an attacker would not have access to the user’s password (otherwise, the user could change his password to invalidate the token key). Since we never transmit the token key to the server (the token key never goes over the wire), it’s possible instead that the user’s PC could be physically compromised or stolen. An attacker could access the localStorage, obtain the user’s key, and generate his own token on behalf of the user. Really, this is no different than auto-sign-in, which many web applications already do for the physical computer. We have to consider this case a more remote possibility and leave it to other security mechanisms to protect the physical PC.

Code for Generating a Token

The method for generating a token is included below. Keep in mind, this code will need to be mirrored on the client (such as with javascript) in order to make API calls.

private const string \_alg = "HmacSHA256";

private const string \_salt = "rz8LuOtFBXphj9WQfvFh"; // Generated at https://www.random.org/strings

public static string GenerateToken(string username, string password, string ip, string userAgent, long ticks)

{

string hash = string.Join(":", new string[] { username, ip, userAgent, ticks.ToString() });

string hashLeft = "";

string hashRight = "";

using (HMAC hmac = HMACSHA256.Create(\_alg))

{

hmac.Key = Encoding.UTF8.GetBytes(GetHashedPassword(password));

hmac.ComputeHash(Encoding.UTF8.GetBytes(hash));

hashLeft = Convert.ToBase64String(hmac.Hash);

hashRight = string.Join(":", new string[] { username, ticks.ToString() });

}

return Convert.ToBase64String(Encoding.UTF8.GetBytes(string.Join(":", hashLeft, hashRight)));

}

public static string GetHashedPassword(string password)

{

string key = string.Join(":", new string[] { password, \_salt });

using (HMAC hmac = HMACSHA256.Create(\_alg))

{

// Hash the key.

hmac.Key = Encoding.UTF8.GetBytes(\_salt);

hmac.ComputeHash(Encoding.UTF8.GetBytes(key));

return Convert.ToBase64String(hmac.Hash);

}

}

In the above code for GenerateToken(), notice that we provide a username, password, IP address, user-agent, and timestamp (in ticks). Using this information, we can generate a time-sensitive token that is bound to a specific IP address and web browser. It’s also important to note that we utilize a hashed password, rather than the actual user password.

Notice that the final token actually has two parts. There is a hashed portion, which includes sensitive details. There is also a public portion (base64-encoded) that includes necessary public data for the server to validate the token. When validating, we’ll extract the username and timestamp and lookup the remaining fields in order to authenticate.

Validating a Token

Every API call will contain a token as part of the url. To validate a token we can follow a series of steps. First, we base64-decode the string. This provides us with the token, along with the username and timestamp. We can then validate the timestamp to ensure it has not yet expired. If the timestamp is valid, we’ll next lookup the user in our database to obtain their hashed password or other unique identifying key. With this information, we can now compute a token and verify that it matches the one passed to our API. We do this by hashing the username:ip:userAgent:timeStamp with the key hashed password:salt. We can then compare the computed token and ensure that it matches the provided one.

The method for checking a valid token is included below.

private const int \_expirationMinutes = 10;

public static bool IsTokenValid(string token, string ip, string userAgent)

{

bool result = false;

try

{

// Base64 decode the string, obtaining the token:username:timeStamp.

string key = Encoding.UTF8.GetString(Convert.FromBase64String(token));

// Split the parts.

string[] parts = key.Split(new char[] { ':' });

if (parts.Length == 3)

{

// Get the hash message, username, and timestamp.

string hash = parts[0];

string username = parts[1];

long ticks = long.Parse(parts[2]);

DateTime timeStamp = new DateTime(ticks);

// Ensure the timestamp is valid.

bool expired = Math.Abs((DateTime.UtcNow - timeStamp).TotalMinutes) > \_expirationMinutes;

if (!expired)

{

//

// Lookup the user's account from the db.

//

if (username == "john")

{

string password = "password";

// Hash the message with the key to generate a token.

string computedToken = GenerateToken(username, password, ip, userAgent, ticks);

// Compare the computed token with the one supplied and ensure they match.

result = (token == computedToken);

}

}

}

}

catch

{

}

return result;

}

To validate a token, we must be able to compute one ourselves and compare it to the one supplied by the client. Our GenerateToken() method requires 5 parameters: username, password, ip, userAgent, and ticks. So, where do we get this information?

First, note that the IsValidToken() method receives a token, IP address, and user-agent string. All three fields are passed to us by the .NET MVC controller. The token is passed as part of the url, while the IP and user-agent are determined from the HTTP request object. This ensures that we are attempting to compute a token based upon the actual client making the API call. This effectively gives us two of the required parameters for computing a token.

Since the provided token, itself, consists of two pieces (a hashed part and a public username:timestamp part), we can easily extract two more parameters. This gives us the username and ticks.

We now only need one more parameter: the user’s hashed password (or other unique identifier). We can obtain this on the server by looking up the user’s account from the database. It’s important to note that the user’s password is never transferred over the wire. It is looked up on the server or entered by the user in the client web browser, but never transferred between.

With all 5 parameters obtained, we can now call GenerateToken() and compare the result.

Securing the Web Service Controller

We now have methods for generating and validating tokens. However, how does the token get passed to these methods? We know that our client will be calling REST web service API methods. Since our REST web service is an MVC .NET controller, we can check the token parameter in each controller method.

An example C# MVC .NET web service API controller method is included below:

[RESTAuthorize]

public class ApiController : Controller

{

[HttpGet]

public JsonResult Person(string query)

{

var data = PersonRepository.Find(query);

return Json(data, JsonRequestBehavior.AllowGet);

}

}

The above code shows an example of a simple MVC controller method. The method takes a string as a parameter and returns a JSON response. We can assume the REST url for calling this method would be, as follows:

http://localhost/api/person?query=Luke?token=R2D2Sqr7c3V0R2dTMxMDAzMTA3MDAwMAC3P0==

The controller method is only concerned with receiving a parameter, “query”. We can still pass the token as a url parameter and have it validated as well. We do this with the RESTAuthorize attribute. It’s added at the top of the MVC controller class and will be effective on all controller methods within the class.

The RESTAuthorizeAttribute

The RESTAuthorizeAttribute specifically looks for the token url parameter and calls the IsTokenValid() method to authorize it. It handles sending the appropriate data from the client request, including the IP address, user-agent, and any other data pieces that need to be validated.

The attribute is defined as follows:

public class RESTAuthorizeAttribute : AuthorizeAttribute

{

private const string \_securityToken = "token"; // Name of the url parameter.

public override void OnAuthorization(AuthorizationContext filterContext)

{

if (Authorize(filterContext))

{

return;

}

HandleUnauthorizedRequest(filterContext);

}

protected override void HandleUnauthorizedRequest(AuthorizationContext filterContext)

{

base.HandleUnauthorizedRequest(filterContext);

}

private bool Authorize(AuthorizationContext actionContext)

{

try

{

HttpRequestBase request = actionContext.RequestContext.HttpContext.Request;

string token = request.Params[\_securityToken];

return SecurityManager.IsTokenValid(token, CommonManager.GetIP(request), request.UserAgent);

}

catch (Exception)

{

return false;

}

}

}

Notice, in the above code we call IsTokenValid() from within the Authorize() method. This allows us to protect our C# ASP .NET MVC controller, while keeping the token-based authentication parameter separate from the actual controller methods.

We now have a fully functioning token-based authentication framework. The only part missing is to allow a token to be generated by the client.

Generating a Token on the Client

The C# .NET method for generating a token is only used by the IsValidToken() method, as part of computing a token to authenticate. However, the client still needs to generate his own token in order to call the API methods. We can do this in a variety of programming languages, but for this tutorial, we’ll focus on Javascript. This is especially handy for single page applications created with AngularJs or other MVC client frameworks.

We can create a javascript file, which can be included in the main page layout or parent view. The contents of this file will allow generation of a token, as shown below.

security.js

var SecurityManager = {

salt: 'rz8LuOtFBXphj9WQfvFh', // Generated at https://www.random.org/strings

username: localStorage['SecurityManager.username'],

key: localStorage['SecurityManager.key'],

ip: null,

generate: function (username, password) {

if (username && password) {

// If the user is providing credentials, then create a new key.

SecurityManager.logout();

}

// Set the username.

SecurityManager.username = SecurityManager.username || username;

// Set the key to a hash of the user's password + salt.

SecurityManager.key = SecurityManager.key || CryptoJS.enc.Base64.stringify(CryptoJS.HmacSHA256([password, SecurityManager.salt].join(':'), SecurityManager.salt));

// Set the client IP address.

SecurityManager.ip = SecurityManager.ip || SecurityManager.getIp();

// Persist key pieces.

if (SecurityManager.username) {

localStorage['SecurityManager.username'] = SecurityManager.username;

localStorage['SecurityManager.key'] = SecurityManager.key;

}

// Get the (C# compatible) ticks to use as a timestamp. http://stackoverflow.com/a/7968483/2596404

var ticks = ((new Date().getTime() \* 10000) + 621355968000000000);

// Construct the hash body by concatenating the username, ip, and userAgent.

var message = [SecurityManager.username, SecurityManager.ip, navigator.userAgent.replace(/ \.NET.+;/, ''), ticks].join(':');

// Hash the body, using the key.

var hash = CryptoJS.HmacSHA256(message, SecurityManager.key);

// Base64-encode the hash to get the resulting token.

var token = CryptoJS.enc.Base64.stringify(hash);

// Include the username and timestamp on the end of the token, so the server can validate.

var tokenId = [SecurityManager.username, ticks].join(':');

// Base64-encode the final resulting token.

var tokenStr = CryptoJS.enc.Utf8.parse([token, tokenId].join(':'));

return CryptoJS.enc.Base64.stringify(tokenStr);

},

logout: function () {

SecurityManager.ip = null;

localStorage.removeItem('SecurityManager.username');

SecurityManager.username = null;

localStorage.removeItem('SecurityManager.key');

SecurityManager.key = null;

},

getIp: function () {

var result = '';

$.ajax({

url: '/ip',

method: 'GET',

async: false,

success: function (ip) {

result = ip;

}

});

return result;

}

};

Also include the crypto-js library in your main page:

<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.2/rollups/hmac-sha256.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.2/components/enc-base64.js"></script>

The above javascript code allows the client to generate a token for each web service API call that is made. While we’re using javascript, you can generate a token with HMAC in a variety of programming languages. Here’s how it’s used.

When the user initially logs into the web application, they will provide their username and password for authentication. Since we’re not validating with traditional session ASP .NET forms-based authentication, we will authenticate via a token instead.

The javascript method generate() may be called, by passing in the username and password, provided by the client. This information is not transmitted to the server and is held completely in the web browser client. The method then computes a hashed token key (bound to an IP address), and uses this to generate a token. For single-page applications (AngularJs, etc) the key can be stored in memory via javascript. For multi-page applications that require a page reload or for further persistence, the key can be stored in localStorage. In either case, the key never leaves the client.

Getting the Client IP Address

The javascript code for generating a token makes a call to the web application to obtain the client IP address. We can use a simple C# MVC .NET helper method and controller with the following code:

public static class CommonManager

{

public static string GetIP(HttpRequestBase request)

{

string ip = request.Headers["X-Forwarded-For"]; // AWS compatibility

if (string.IsNullOrEmpty(ip))

{

ip = request.UserHostAddress;

}

return ip;

}

}

public class IpController : Controller

{

[HttpGet]

public string Index()

{

return CommonManager.GetIP(Request);

}

}

The above method allows us to determine the web browser client IP address. Note, we include a check on the HTTP header value “X-Forwarded-For”, in case the web application is behind an AWS cloud instance or load balancer. If not, we default to using the standard Request.UserHostAddress value.

Example Token Requests

The following example shows how to use the code to generate a token. The initial call from the login page generates the key:

1

$http.get('/api/login?token=' + SecurityManager.generate('user', 'password')).success(function() { ... });

Of course, to login, an API call isn’t even needed. Simply call the generate(username, password) method to have a key created on the client, usable by subsequent API calls:

1

SecurityManager.generate('user', 'password');

Subsequent calls to the web service simply omit the username and password, as upon leaving the login page, the password is no longer in memory.

1

$http.get('/api/person?q=some\_query&token=' + SecurityManager.generate()).success(function() { ... });

Notice that each call to SecurityManager.generate() creates a unique token, each with its own timestamp. Each token is only valid for a short duration of time (ie., 5-10 minutes) before it is expired. If the client provides a different timestamp as part of the public portion of the token, or if the client provides a different IP address or user-agent than the one contained within the token body, it will fail to match the hashed message portion of the token, and thus, access will be denied.

Conclusion

Developers have a variety of options for securing web applications. Two popular options include session-backed forms authentication with cookies and token-based authentication via the url. While both options offer a secure solution for a C# ASP .NET MVC web application, token-based authentication excels, in particular, with cloud-compatibility. Token-based frameworks also offer an advantage in striving for a stateless REST web service, compared with utilizing session for maintaining application/user state.

http://www.primaryobjects.com/2015/05/08/token-based-authentication-for-web-service-apis-in-c-mvc-net/

The Open Web Interface for .NET (OWIN) defines a standard interface between .NET web servers and web applications. Katana is open-source components for building and hosting OWIN-based web applications. It provides the implementation of the OWIN specification. The OAuth authorization framework enables a third-party application to obtain limited access to a HTTP service.

Currently the preferred approach to authenticate the users is to use a signed token and this token is sent to the server with each request. The following are the benefits for using this approach.

Scalability of Servers

the token itself contains all the information of the user that is needed for authentication, so Web Farm extension is an easy task. There is no dependence on shared session stores.

Loosely Coupling

Our front-end application is not coupled with a specific authentication mechanism. The token is generated from the server and our web API has a built-in way to understand this token and perform authentication.

Mobile Friendly

This type of authentication does not require cookies, so this authentication type can be used with mobile applications.

Example

In the following demo application, the OAuth authorization server and the Web API endpoints will be hosted inside the same host.

The following is the procedure to do Token Based Authentication using ASP.NET Web API, OWIN and Identity.

Step 1

Create and configure a Web API project.

Create an empty solution for the project template "ASP.NET Web Application" and add a core reference of the Web API and set the authentication to “No Authentication”.

Create and configur

Update the current version of the Web API using the Nuget package with the following command.

PM> Update-package Microsoft.AspNet.WebApi

WebApi

Step 2

Instal the required OWIN component using Nuget Packages.

In this step, we need to install Nuget packages that are required to set up our OWIN server and configure the Web API to be hosted within the OWIN server. The "Microsoft.AspNet.Identity.Owin" package provides many useful extensions and we will use this while working with ASP.Net Identity on top of OWIN. It also downloads some other dependency packages. One of those dependency packages is "Microsoft.Owin.Security.OAuth". This is a core package required to support any standard OAuth 2.0 authentication workflow. The “Microsoft.Owin.Host.SystemWeb” namespace contains the types related to handling OWIN requests. It helps us to run OWIN-based applications on IIS using the ASP.NET request pipeline.

Use the following commands to instal the OWIN server.

PM> Install-Package Microsoft.AspNet.Identity.Owin

PM> Install-Package Microsoft.Owin.Host.SystemWeb

ASP.NET Identity also supports the Entity Framework. Here we will use ASP.Net identity with Entity Framework, so we need to install this via Nuget packages.

PM> Install-Package Microsoft.AspNet.Identity.EntityFramework

This will also install the Entity Framework as a dependency.

Step 3

Create a DbContext class. The IdentityDbContext Class uses the default entity types for ASP.NET Identity Users, Roles, Claims and Logins. We can overload this to add our own entity types.

public class OwinAuthDbContext : IdentityDbContext

{

public OwinAuthDbContext()

: base("OwinAuthDbContext")

{

}

}

Step 4

Do the migrations (optional step).

Entity Framework supports the database migration to create the database and insert some initial values. Migration commands can be executed from the Package Manager Console.

Migration commands

Step 4

Define an OWIN Startup Class.

Every OWIN application has a startup class in which we specify components for the application pipeline. Here we are using the OwinStartup Attribute to connect to the startup class with the hosting runtime.

using Microsoft.Owin;

using Owin;

[assembly: OwinStartup(typeof(OwinAuthentication.Startup))]

namespace OwinAuthentication

{

public class Startup

{

public void Configuration(IAppBuilder app)

{

}

}

}

Step 5

Configure the OAuth Authorization Server.

AppBuilderExtensions has the method "CreatePerOwinContext<T>" that registers a callback that will be invoked to create an instance of type T and it will be stored in the OwinContext. Later on we can retrieve it using the context.Get method. This method creates one instance of the given type per request. Here we are using ASP.Net Identity with Entity Framework, so we must create the instance of our DbContext class and to do this we use this extension method.

The Microsoft.AspNet.Identity namespace has the class UserManager that exposes the use related to the API that automatically saves the changes to the UserStore. Here we will interact with our database using the UserManager class.

The following code is required to use the UserManager class inside our OWIN component efficiently.

private void ConfigureOAuth(IAppBuilder app)

{

app.CreatePerOwinContext<OwinAuthDbContext>(() => new OwinAuthDbContext());

app.CreatePerOwinContext<UserManager<IdentityUser>>(CreateManager);

}

private static UserManager<IdentityUser> CreateManager (IdentityFactoryOptions<UserManager<IdentityUser>> options, IOwinContext context)

{

var userStore = new UserStore<IdentityUser>(context.Get<OwinAuthDbContext>());

var owinManager = new UserManager<IdentityUser>(userStore);

return owinManager;

}

The ConfigureOAuth method will be called inside the Configuration method of the OWIN startup class.

The UseOAuthAuthorizationServer extension method of OWIN is used to setup the authorization server. Following are the setup options:

TokenEndpointPath

requests the path on which the client application directly communicates to obtain the access token. It must begin with a leading slash, for example "/oauth/token".

AuthorizeEndpointPath

It is the request path where the client application will redirect the user-agent to obtain the user's consent to issue a token or code. It must begin with a leading slash (the same as TokenEndpointPath).

AllowInsecureHttp

Set to true to allow authorize and token requests to arrive on HTTP URI addresses.

Provider

the object provided by the application to process events raised by the Authorization Server. It may the instance of OAuthAuthorizationServerProvider and assign delegates necessary for the OAuth flow.

AuthorizationCodeProvider

produces a single-use authorization code to return to the client application. It is required where the token is produced by the OnCreate/OnCreateAsync event.

RefreshTokenProvider

produces a refresh token that may produce a new access token when required. If this option is not provided then the authorization server will not return refresh tokens from the Token endpoint.

ApplicationCanDisplayErrors

Set to true when the web application is able to render error messages on the Authorize endpoint. This is required only in cases where the browser is not redirected back to the client application.

AccessTokenExpireTimeSpan

The time period the access token remains valid after it was generated. The default value is 20 minutes.

AccessTokenFormat

The data format used to protect the information contained by the access token. If it is not provided then the application will use the default data protection provider depending on the host server.

AccessTokenProvider

It produces a bearer token.

AuthorizationCodeExpireTimeSpan

The time period the authorization code remains valid after it was generated. The default value is 5 minutes

AuthorizationCodeFormat

The data format used to protect and unprotect the information contained in the authorization code.

RefreshTokenFormat

The data format used to protect and unprotect the information contained in the refresh token.

SystemClock

Used to know what the current clock time is when calculating or validating token expiration. The default value is DateTimeOffset.UtcNow.

private void ConfigureOAuth(IAppBuilder app)

{

….

….

app.UseOAuthAuthorizationServer(new OAuthAuthorizationServerOptions

{

TokenEndpointPath = new PathString("/oauth/token"),

Provider = new AuthorizationServerProvider(),

AccessTokenExpireTimeSpan = TimeSpan.FromMinutes(30),

AllowInsecureHttp = true,

});

app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());

}

An Authorization Server uses a default implementation of IOAuthAuthorizationServerProvider to communicate with a web application while processing the request. OAuthAuthorizationServerProvider provides some default behavior like as used as a virtual base class and offers delegate properties that may be used to handle individual calls without creating the instance.

Here I just override the methods that I need here: ValidateClientAuthentication and GrantResourceOwnerCredentials.

ValidateClientAuthentication

It is called to validate that the requester (origin of the request) is a registered client\_id and the correct credentials for that client are present on the request. "OAuthValidateClientAuthenticationContext.TryGetBasicCredentials" may be able to retrieve the values of the client credential request header if the web application accepts basic authentication credentials. If the web application accepts a client id and secret as form-encoded POST parameters, "OAuthValidateClientAuthenticationContext.TryGetFormCredentials" can be used to retrieve this value. Finally, if "OAuthValidateClientAuthenticationContext.Validated” is not called then the request will not proceed further.

GrantResourceOwnerCredentials

It is called when the request to the token endpoint arrives with a "grant\_type" of "password". This occurs when the user provides a user id and password directly to the client application using the client application user interface. If the web application supports the resource owner credentials grant type, it must validate the username and password property of context. To issue the access token, the request must end with the “OAuthGrantResourceOwnerCredentialsContext.Validated" method. The default behavior is to reject this grant type.

public class AuthorizationServerProvider : OAuthAuthorizationServerProvider

{

public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)

{

}

public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)

{

}

}

The following is a probable implementation of the ValidateClientAuthentication method.

public override async Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)

{

string clientId;

string clientSecret;

if (context.TryGetBasicCredentials(out clientId, out clientSecret))

{

// validate the client Id and secret against database or from configuration file.

context.Validated();

}

else

{

context.SetError("invalid\_client", "Client credentials could not be retrieved from the Authorization header");

context.Rejected();

}

}

The following is an implementation of the GrantResourceOwnerCredentials method.

public override async Task GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)

{

UserManager<IdentityUser> userManager = context.OwinContext.GetUserManager<UserManager<IdentityUser>>();

IdentityUser user;

try

{

user = await userManager.FindAsync(context.UserName, context.Password);

}

catch

{

// Could not retrieve the user due to error.

context.SetError("server\_error");

context.Rejected();

return;

}

if (user != null)

{

ClaimsIdentity identity = await userManager.CreateIdentityAsync(

user,

DefaultAuthenticationTypes.ExternalBearer);

context.Validated(identity);

}

else

{

context.SetError("invalid\_grant", "Invalid User Id or password'");

context.Rejected();

}

An HTTP POST request is made to the URL "/oauth/token" endpoint with grant\_type parameter "password"; it will first arrive at the ValidateClientAuthentication method. In this place we can retrieve the client credentials and validate it. If the client credential is invalid, we need to return an unauthorized request using the context.Rejected method. If we grant the request in the ValidateClientAuthentication method, the request will arrive at the GrantResourceOwnerCredentials method. Inside this method, we need to validate that the user is using resource-owner credentials.

Step 6

Test the Project.

To test the preceding approach I created a console project in my solution. Create the following Token class within the console application.

using Newtonsoft.Json;

namespace OWINTest

{

public class Token

{

[JsonProperty("access\_token")]

public string AccessToken { get; set; }

[JsonProperty("token\_type")]

public string TokenType { get; set; }

[JsonProperty("expires\_in")]

public int ExpiresIn { get; set; }

[JsonProperty("refresh\_token")]

public string RefreshToken { get; set; }

[JsonProperty("error")]

public string Error { get; set; }

}

}

Program.cs code

using System;

using System.Collections.Generic;

using System.Net.Http;

using System.Net.Http.Formatting;

namespace OWINTest

{

class Program

{

static void Main(string[] args)

{

string baseAddress = "http://localhost:4312";

using (var client = new HttpClient())

{

var form = new Dictionary<string, string>

{

{"grant\_type", "password"},

{"username", "jignesh"},

{"password", "user123456"},

};

var tokenResponse = client.PostAsync(baseAddress + "/oauth/token", new FormUrlEncodedContent(form)).Result;

//var token = tokenResponse.Content.ReadAsStringAsync().Result;

var token = tokenResponse.Content.ReadAsAsync<Token>(new[] { new JsonMediaTypeFormatter() }).Result;

if (string.IsNullOrEmpty(token.Error))

{

Console.WriteLine("Token issued is: {0}", token.AccessToken);

}

else

{

Console.WriteLine("Error : {0}", token.Error);

}

Console.Read();

}

}

}

}

Using the following script, insert some dummy data into the AspNetUser table. Here I have encrypted my password using Abstraction for the password hashing methods of the Microsoft.AspNet.Identity namespace available with the UserManger class.

INSERT [dbo].[AspNetUsers] ([Id], [Email], [EmailConfirmed], [PasswordHash], [SecurityStamp],

[PhoneNumber], [PhoneNumberConfirmed], [TwoFactorEnabled], [LockoutEndDateUtc], [LockoutEnabled],

[AccessFailedCount], [UserName])

VALUES (N'9f15bdd0fcd5423190c2e877ba0228ee', N'abc@gail.com', 1,

N'ALkHGax/i5KBYWJ7q4jhJmMKmm2quBtnnqS8KcmLWd2kQpN6FaGVulDmmX12s7YAyQ==',

N'a7bc5c5c-6169-4911-b935-6fc4df01d313', NULL, 0, 0, NULL, 0, 0, N'Jignesh')

Output

JSON

We receive the response in the form of JSON and we convert it into our Token class using the Content.ReadAsAsync method. The Result Token class contains either the access token or an error. If we pass the wrong credentials, the system will generate the error:

error

In the next request we use this token for the authentication and the token will be sent in the request header. The Token is valid up to its expiry time.

To test it, I added a controller to my Web API project and created a test method as in the following.

using System.Web.Http;

namespace OwinAuthentication.Controllers

{

[Authorize]

public class POCController : ApiController

{

[HttpGet]

[Route("api/TestMethod")]

public string TestMethod()

{

return "Hello, C# Corner Member. ";

}

}

}

Here we must mark our controller or action method with the Authorize attribute to check whether the request has a valid token.

The following is needed to append in a client application (console application in our case). First of all we need to get the token using the code described in the preceding section and then use this token to process the request.

using (HttpClient httpClient1 = new HttpClient())

{

httpClient1.BaseAddress = new Uri(baseAddress);

httpClient1.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("bearer", token.AccessToken);

HttpResponseMessage response = httpClient1.GetAsync("api/TestMethod").Result;

if (response.IsSuccessStatusCode)

{

System.Console.WriteLine("Success");

}

string message = response.Content.ReadAsStringAsync().Result;

System.Console.WriteLine("URL responese : " + message);

}

Output

Output

Summary

Using this article we can set up our authorization server and we have a working OAuth 2.0 token endpoint that only supports "Resource Owner Password Credentials Grant" as of now.

http://www.c-sharpcorner.com/UploadFile/ff2f08/token-based-authentication-using-Asp-Net-web-api-owin-and-i/

In my previous article, we saw an overview of Token based authentication using ASP.net web API and OWIN. In this post, I will explain how to use Token based authentication in AngularJS.

In this post, I will build sample Single Page Application (SPA) using AngularJS and this application will do the following:

Allow user to login using "Admin" and "Jignesh" user ID

Token keep alive 30 minutes

Authenticated user will access the certain views.

Prerequisites

Before reading this article, you must have some basic knowledge about AngularJS and Token base authentication using OWIN.

The following are the steps to create AngularJS Token Authentication using ASP.NET Web API 2 and OWIN

Step 1

Include 3rd party libraries

To get started, we required to include the following libraries:

AngularJS

We can download the latest AngularJS version using NuGet package manager.

PM> Install-Package angularjs

Preceding command includes all available AngularJS libraries including minified version. So delete script files which are not required.

UI Bootstrap

We can download the latest Bootstrap version using NuGet package manager.

PM> Install-Package bootstrap -Version 3.3.5

Step 2

Organize Project Structure

We can use any IDE to build the web application because this web app totally decouples with backend API and it develops using HTML, AngularJS, and CSS. Here I am using Visual Studio 2013. I have created project using empty project template.

In this project structure, I have created a folder named "Modules", this contains all AngularJS application files and resources files and "Asset" folder contains the asset of this project i.e. AngularJS libraries file, CSS files, etc.

Step 3

Boot Strapping Angular Application

Boot Strapping Angular Application means creating angular application and modules (modules are nothing but collection of services, directives, filters which are used by the application). Each module has configuration block and applied to the application during this process. To do this, I have added a file called "app.js" in the root folder. This file also contains the route and interceptor.

Interceptor is a regular service and it allows us to capture every XHR request and we can also manipulate it before sending it to server end point (web API). It also captures all response and response error.

var serviceBase = 'http://localhost:49707/';

var app = angular.module('AngularApp', ['ngRoute', 'LocalStorageModule']);

app.config(function ($routeProvider) {

$routeProvider.when("/home", {

controller: "homeController",

templateUrl: "/Modules/views/home.html"

});

$routeProvider.when("/login", {

controller: "loginController",

templateUrl: "/Modules/views/login.html"

});

$routeProvider.when("/next", {

controller: "nextController",

templateUrl: "/Modules/views/Next.html"

});

$routeProvider.when("/myInfo", {

templateUrl: "/Modules/views/Info.html"

});

$routeProvider.otherwise({ redirectTo: "/home" });

})

.config(['$httpProvider', function ($httpProvider) {

$httpProvider.interceptors.push(function ($q, $rootScope, $window, $location) {

return {

request: function (config) {

return config;

},

requestError: function (rejection) {

return $q.reject(rejection);

},

response: function (response) {

if (response.status == "401") {

$location.path('/login');

}

//the same response/modified/or a new one need to be returned.

return response;

},

responseError: function (rejection) {

if (rejection.status == "401") {

$location.path('/login');

}

return $q.reject(rejection);

}

};

});

}]);

Here, I have defined four views with their corresponding controllers

Home: It is home page. It can be also access by the anonymous users.

Login: It shows the login form. It can be also access by the anonymous users.

next: It shows after user has been logged-in.

myInfo: It shows my details.

Step 4

Add Index.html (Shell Page)

Single page application contains the Shell page which is container for the application. It will contain the navigation menus which contains all the available links for the application. It also contains reference of all the 3rd party JavaScript and CSS files which are required by the application.

<!DOCTYPE html>

<html data-ng-app="AngularApp">

<head>

<meta content="IE=edge, chrome=1" http-equiv="X-UA-Compatible" />

<title>AngularJS - OWIN Authentication</title>

<link href="Asset/Content/bootstrap.min.css" rel="stylesheet" />

<link href="Asset/Content/ProjectStyle.css" rel="stylesheet" />

</head>

<body>

<div class="navbar navbar-inverse navbar-fixed-top" role="navigation" data-ng-controller="indexController">

<div class="container">

<div class="collapse navbar-collapse" data-collapse="!navbarExpanded">

<ul class="nav navbar-nav navbar-right">

<li data-ng-hide="!authentication.IsAuthenticated"><a href="#">Welcome, {{authentication.userName}}</a></li>

<li data-ng-hide="!authentication.IsAuthenticated"><a href="#/myInfo">My Info</a></li>

<li data-ng-hide="!authentication.IsAuthenticated"><a href="" data-ng-click="logOut()">Logout</a></li>

<li data-ng-hide="authentication.IsAuthenticated"> <a href="#/login">Login</a></li>

</ul>

</div>

</div>

</div>

<div class="jumbotron">

<div class="container">

<div class="page-header text-center">

<h3>AngularJS Owin Authentication</h3>

</div>

</div>

</div>

<div class="container">

<div data-ng-view="">

</div>

</div>

<hr />

<div id="footer">

<div class="container">

<div class="row">

AngularJS - OAuth Bearer Token Implementation Example

</div>

</div>

</div>

<!-- 3rd party libraries -->

<script src="Asset/Scripts/angular.js"></script>

<script src="Asset/Scripts/angular-route.js"></script>

<script src="Asset/Scripts/angular-local-storage.min.js"></script>

<!-- Load app main script -->

<script src="Modules/app.js"></script>

<!-- Load Angular services -->

<script src="Modules/Services/loginService.js"></script>

<script src="Modules/Services/AuthenticationService.js"></script>

<script src="Modules/Services/AuthData.js"></script>

<!-- Load Angular controllers -->

<script src="Modules/Controllers/indexController.js"></script>

<script src="Modules/Controllers/homeController.js"></script>

<script src="Modules/Controllers/loginController.js"></script>

<script src="Modules/Controllers/nextController.js"></script>

</body>

</html>

Indexcontroller.js

Now we need to add index controller under "Controller" folder which will responsible to change the layout of for index page i.e. when user is not logged-in, it displays only "login" menu, else displays welcome text and logout menu.

(function () {

'use strict';

app.controller('indexController', ['$scope', '$location', 'authData','LoginService', function ($scope, $location, authData, loginService) {

$scope.logOut = function () {

loginService.logOut();

$location.path('/home');

}

$scope.authentication = authData.authenticationData;

}]);

})();

Step 5:

Add AngularJS Authentication Data (Factory)

This AngularJS service will be responsible for storing the authentication values. It contains the object called "authentication", which will store two values (IsAuthenticated and username). This object will be used to change the layout of the Index page (mainly menu option).

'use strict';

app.factory('authData', [ function () {

var authDataFactory = {};

var \_authentication = {

IsAuthenticated: false,

userName: ""

};

authDataFactory.authenticationData = \_authentication;

return authDataFactory;

}]);

Step 6:

Add AuthenticationService

This AngularJS service will be responsible for get and set token data in to client windows session, remove token from the client windows session and set http header. We have to configure the http request header for the end point: content type as “application/x-www-form-urlencoded” and sent the data as string not JSON object and also need to set Bearer token.

(function () {

'use strict';

app.service('AuthenticationService', ['$http', '$q', '$window',

function ($http, $q, $window) {

var tokenInfo;

this.setTokenInfo = function (data) {

tokenInfo = data;

$window.sessionStorage["TokenInfo"] = JSON.stringify(tokenInfo);

}

this.getTokenInfo = function () {

return tokenInfo;

}

this.removeToken = function () {

tokenInfo = null;

$window.sessionStorage["TokenInfo"] = null;

}

this.init = function () {

if ($window.sessionStorage["TokenInfo"]) {

tokenInfo = JSON.parse($window.sessionStorage["TokenInfo"]);

}

}

this.setHeader = function (http) {

delete http.defaults.headers.common['X-Requested-With'];

if ((tokenInfo != undefined) && (tokenInfo.accessToken != undefined) && (tokenInfo.accessToken != null) && (tokenInfo.accessToken != "")) {

http.defaults.headers.common['Authorization'] = 'Bearer ' + tokenInfo.accessToken;

http.defaults.headers.common['Content-Type'] = 'application/x-www-form-urlencoded;charset=utf-8';

}

}

this.validateRequest = function () {

var url = serviceBase + 'api/home';

var deferred = $q.defer();

$http.get(url).then(function () {

deferred.resolve(null);

}, function (error) {

deferred.reject(error);

});

return deferred.promise;

}

this.init();

}

]);

})();

Step 7

Add Login service, controller and its view

LoginService.js

Login service is added under the folder “Services”. It contains the method for login and logout. The function "login" is responsible to send HTTP request to the end point with user credential and end point will validate the user credential and generate token. This method also set authorized token to browser window session. The function "logout" is responsible to clear browser window session and redirect to home page.

function () {

'use strict';

app.service('LoginService', ['$http', '$q', 'AuthenticationService', 'authData',

function ($http, $q, authenticationService, authData) {

var userInfo;

var loginServiceURL = serviceBase + 'token';

var deviceInfo = [];

var deferred;

this.login = function (userName, password) {

deferred = $q.defer();

var data = "grant\_type=password&username=" + userName + "&password=" + password;

$http.post(loginServiceURL, data, {

headers:

{ 'Content-Type': 'application/x-www-form-urlencoded' }

}).success(function (response) {

var o = response;

userInfo = {

accessToken: response.access\_token,

userName: response.userName

};

authenticationService.setTokenInfo(userInfo);

authData.authenticationData.IsAuthenticated = true;

authData.authenticationData.userName = response.userName;

deferred.resolve(null);

})

.error(function (err, status) {

authData.authenticationData.IsAuthenticated = false;

authData.authenticationData.userName = "";

deferred.resolve(err);

});

return deferred.promise;

}

this.logOut = function () {

authenticationService.removeToken();

authData.authenticationData.IsAuthenticated = false;

authData.authenticationData.userName = "";

}

}

]);

})();

loginController.js

Now I have added logincontroller.js under the folder "Controllers". Generally controller is simple and will contain the client side business logic. It is a bridge between service and HTML view.

(function () {

'use strict';

app.controller('loginController', ['$scope', 'LoginService', '$location', function ($scope, loginService, $location) {

$scope.loginData = {

userName: "",

password: ""

};

$scope.login = function () {

loginService.login($scope.loginData.userName, $scope.loginData.password).then(function (response) {

if (response != null && response.error != undefined) {

$scope.message = response.error\_description;

}

else {

$location.path('/next');

}

});

}

}]);

})();

Logincontroller responsible to redirect authenticated users only to the "next" view else system will redirect to login page. To do this we need to write some code in controller side and catch the "401" response code at interceptor. I have defined interceptor in app.js file. AuthenticationService has method called "validateRequest", it help us to validate whether user is logged-in or not by sending request to server and server will sent "401" status code if it is unauthorized. Here I have added one example code in "nextController".

nextController.js

(function ()

{

'use strict';

app.controller('nextController', ['$scope', 'AuthenticationService', function ($scope, authenticationService) {

authenticationService.validateRequest();

}]);

})();

Login.html

View for the log-in is very simple. A new file named “login.html” created under the view folder.

<form role="form">

<div class="row">

<div class="col-md-2">

</div>

<div class="col-md-4">

<h2 class="form-login-heading col-md-12">Login</h2>

<div class="col-md-12 PaddingTop">

<input type="text" class="form-control" placeholder="Username" data-ng-model="loginData.userName" required autofocus>

</div>

<div class="col-md-12 PaddingTop">

<input type="password" class="form-control" placeholder="Password" data-ng-model="loginData.password" required>

</div>

<div class="col-md-12 PaddingTop">

<button class="btn btn-md btn-info btn-block" type="submit" data-ng-click="login()">Login</button>

</div>

<div data-ng-hide="message == ''">

{{message}}

</div>

</div>

<div class="col-md-2">

</div>

</div>

</form>

Step 8

Add Home controller and its view

Lastly, I have added home controller and its view. It has very simple view and empty controller which is used to display text "Home page".

HomeController.js

(function () {

'use strict';

app.controller('homeController', ['$scope', function ($scope) {

}]);

})();

Summary

Now we have SPA which authenticates users by using token based approach. Here redirection for anonymous users to the login page is done by client side code. So it is very important to secure all server side (web API) methods. This is not done in this article.

http://www.c-sharpcorner.com/UploadFile/ff2f08/token-based-authentication-using-Asp-Net-web-api-in-angularj/

In this article, I want to show you an example of how the authentication can be implemented using the ASP.NET Web API as a back-end and Angular 2 as the front-end technology.

For this purpose, I’m going to use an already implemented application and show you just most important pieces of this.

First, let’s take a look at the front-end. In app.module, we will register a new provider which is going to be responsible for providing information about authentication (checking whether user is logged in or not) and also for performing the authentication. This is the example of the code.

Latest Update In 2017: MBuild Alpha Release .Net Core Tools in Visual Studio 2017- The new release of .Net Core Tools based on MSBuild or .Net Core can be tried in Mac Visual Studio or Visual Studio version 2017 RC or at commandline. These tools can run in both .Net Core 1.1 and 1.0 versions.

providers: [

AuthGuard,

AuthenticationService,

IdentityService,

{

provide: AuthHttp,

useFactory: getAuthHttp,

deps: [Http]

},

First lets take a look at the IdentityService, this is the code for it:

import { Injectable } from '@angular/core';

import { Http, Headers, RequestOptions, Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/catch';

import 'rxjs/add/observable/throw';

import { AuthHttp } from 'angular2-jwt';

/\*\*

\* Identity service (to Identity Web API controller).

\*/

@Injectable()

export class IdentityService {

headers: Headers;

options: RequestOptions;

constructor(private authHttp: AuthHttp, private http: Http) {

// Creates header for post requests.

this.headers = new Headers({ 'Content-Type': 'application/json' });

this.options = new RequestOptions({ headers: this.headers });

}

/\*\*

\* Gets all users through AuthHttp.

\*/

public GetAll(): Observable<any> {

// Sends an authenticated request.

return this.authHttp.get("/api/identity/GetAll")

.map((res: Response) => {

return res.json();

})

.catch((error: any) => {

// Error on get request.

return Observable.throw(error);

});

}

/\*\*

\* Creates a new user.

\*

\* @param model User's data

\* @return An IdentityResult

\*/

public Create(model: any): Observable<any> {

let body: string = JSON.stringify(model);

return this.http.post("/api/identity/Create", body, this.options)

.map((res: Response) => {

return res.json();

})

.catch((error: any) => {

// Error on post request.

return Observable.throw(error);

});

}

/\*\*

\* Deletes a user through AuthHttp.

\*

\* @param username Username of the user

\* @return An IdentityResult

\*/

public Delete(username: string): Observable<any> {

let body: string = JSON.stringify(username);

// Sends an authenticated request.

return this.authHttp.post("/api/identity/Delete", body, this.options)

.map((res: Response) => {

return res.json();

})

.catch((error: any) => {

// Error on post request.

return Observable.throw(error);

});

}

// Add other methods.

}

Here, we have functions which allow us to Create, Delete, and Get All users of the current application. Please note that for this purpose, we are using the http get/post methods pointing to the URI (Unique Resource Identifier) of API.

Read more at Parameterized URL Routing in Asp.Net MVC Application.

Now, let's take a look at the Authentication Service.

import { Injectable } from '@angular/core';

import { Http, Headers, RequestOptions, Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/catch';

import 'rxjs/add/observable/throw';

import { JwtHelper, tokenNotExpired } from 'angular2-jwt';

import { Config } from '../config';

/\*\*

\* ROPC Authentication service.

\*/

@Injectable() export class AuthenticationService {

/\*\*

\* Stores the URL so we can redirect after signing in.

\*/

public redirectUrl: string;

/\*\*

\* User's data.

\*/

private user: any = {};

private headers: Headers;

private options: RequestOptions;

constructor(private http: Http) {

// On bootstrap or refresh, tries to get the user's data.

this.decodeToken();

// Creates header for post requests.

this.headers = new Headers({ 'Content-Type': 'application/x-www-form-urlencoded' });

this.options = new RequestOptions({ headers: this.headers });

}

/\*\*

\* Tries to sign in the user.

\*

\* @param username

\* @param password

\* @return The user's data

\*/

public signin(username: string, password: string): Observable<any> {

// Token endpoint & params.

let tokenEndpoint: string = Config.TOKEN\_ENDPOINT;

let params: any = {

client\_id: Config.CLIENT\_ID,

grant\_type: Config.GRANT\_TYPE,

username: username,

password: password,

scope: Config.SCOPE

};

// Encodes the parameters.

let body: string = this.encodeParams(params);

return this.http.post(tokenEndpoint, body, this.options)

.map((res: Response) => {

let body: any = res.json();

// Sign in successful if there's an access token in the response.

if (typeof body.access\_token !== 'undefined') {

// Stores access token & refresh token.

this.store(body);

}

}).catch((error: any) => {

// Error on post request.

return Observable.throw(error);

});

}

/\*\*

\* Tries to get a new token using refresh token.

\*/

public getNewToken(): void {

let refreshToken: string = localStorage.getItem('refresh\_token');

if (refreshToken != null) {

// Token endpoint & params.

let tokenEndpoint: string = Config.TOKEN\_ENDPOINT;

let params: any = {

client\_id: Config.CLIENT\_ID,

grant\_type: "refresh\_token",

refresh\_token: refreshToken

};

// Encodes the parameters.

let body: string = this.encodeParams(params);

this.http.post(tokenEndpoint, body, this.options)

.subscribe(

(res: Response) => {

let body: any = res.json();

// Successful if there's an access token in the response.

if (typeof body.access\_token !== 'undefined') {

// Stores access token & refresh token.

this.store(body);

}

});

}

}

/\*\*

\* Revokes token.

\*/

public revokeToken(): void {

let token: string = localStorage.getItem('id\_token');

if (token != null) {

// Revocation endpoint & params.

let revocationEndpoint: string = Config.REVOCATION\_ENDPOINT;

let params: any = {

client\_id: Config.CLIENT\_ID,

token\_type\_hint: "access\_token",

token: token

};

// Encodes the parameters.

let body: string = this.encodeParams(params);

this.http.post(revocationEndpoint, body, this.options)

.subscribe(

() => {

localStorage.removeItem('id\_token');

});

}

}

/\*\*

\* Revokes refresh token.

\*/

public revokeRefreshToken(): void {

let refreshToken: string = localStorage.getItem('refresh\_token');

if (refreshToken != null) {

// Revocation endpoint & params.

let revocationEndpoint: string = Config.REVOCATION\_ENDPOINT;

let params: any = {

client\_id: Config.CLIENT\_ID,

token\_type\_hint: "refresh\_token",

token: refreshToken

};

// Encodes the parameters.

let body: string = this.encodeParams(params);

this.http.post(revocationEndpoint, body, this.options)

.subscribe(

() => {

localStorage.removeItem('refresh\_token');

});

}

}

/\*\*

\* Removes user and revokes tokens.

\*/

public signout(): void {

this.redirectUrl = null;

this.user = {};

// Revokes token.

this.revokeToken();

// Revokes refresh token.

this.revokeRefreshToken();

}

/\*\*

\* Gets user's data.

\*

\* @return The user's data

\*/

public getUser(): any {

return this.user;

}

/\*\*

\* Decodes token through JwtHelper.

\*/

private decodeToken(): void {

if (tokenNotExpired()) {

let token: string = localStorage.getItem('id\_token');

let jwtHelper: JwtHelper = new JwtHelper();

this.user = jwtHelper.decodeToken(token);

}

}

/\*\*

\* // Encodes the parameters.

\*

\* @param params The parameters to be encoded

\* @return The encoded parameters

\*/

private encodeParams(params: any): string {

let body: string = "";

for (let key in params) {

if (body.length) {

body += "&";

}

body += key + "=";

body += encodeURIComponent(params[key]);

}

return body;

}

/\*\*

\* Stores access token & refresh token.

\*

\* @param body The response of the request to the token endpoint

\*/

private store(body: any): void {

// Stores access token in local storage to keep user signed in.

localStorage.setItem('id\_token', body.access\_token);

// Stores refresh token in local storage.

localStorage.setItem('refresh\_token', body.refresh\_token);

// Decodes the token.

this.decodeToken();

}

}

Here, we can note the usage of JWT (JavaScript Web Token); this service is responsible for performing the login/logout functionalities, and also for managing the life cycle of the logged-in user object.

The method getUser() retrieves user which is currently logged in, in order to be able to access information of that user.

Now, let’s take a look at the AuthGuard.

import { Injectable } from '@angular/core';

import { Router, CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

import { tokenNotExpired } from 'angular2-jwt';

import { AuthenticationService } from './authentication.service';

/\*\*

\* Decides if a route can be activated.

\*/

@Injectable() export class AuthGuard implements CanActivate {

constructor(public authenticationService: AuthenticationService, private router: Router) { }

public canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {

if (tokenNotExpired()) {

// Signed in.

return true;

}

// Stores the attempted URL for redirecting.

let url: string = state.url;

this.authenticationService.redirectUrl = url;

// Not signed in so redirects to signin page.

this.router.navigate(['/signin']);

return false;

}

}

This guard performs validation, to see whether the user is logged in or not. Guards are normally used and associated with routes of the Angular2 application, to check whether user can access the route or not, and in our case the condition to pass the guard is that the user should be logged in.

Also, this guard redirects a user to the login component if the user is currently not logged in.

Here is the example of how we can apply the guard to some specific routes.

Routes = [

{ path: '', redirectTo: 'home', pathMatch: 'full' },

{ path: 'home', component: HomeComponent },

{ path: 'resources', component: ResourcesComponent, canActivate: [AuthGuard] },

{ path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] },

{ path: 'signin', component: SigninComponent },

{ path: 'signup', component: SignupComponent }

];

So the main idea here is to separate public routes from ones which requires user to be logged in.

Now let’s take a look at the asp.net web api example:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Authorization;

using \*\*\*\*\*\*.DataCore.Models;

using Microsoft.AspNetCore.Identity;

using \*\*\*\*\*\*.ApiCore.Services;

using \*\*\*\*\*\*.DataCore.Access;

using Microsoft.Extensions.Logging;

using System.Security.Claims;

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using IdentityModel;

using \*\*\*\*\*\*.ApiCore.Models.AccountViewModels;

using IdentityServer4.Extensions;

using Microsoft.EntityFrameworkCore;

namespace \*\*\*\*\*\*.ApiCore.Controllers

{

/// <summary>

/// Identity Web API controller.

/// </summary>

[Route("api/[controller]")]

//[Authorize(Policy = "Manage Accounts")] // Authorization policy for this API.

public class IdentityController : ControllerBase

{

private readonly UserManager<ApplicationUser> \_userManager;

private readonly SignInManager<ApplicationUser> \_signInManager;

private readonly IEmailSender \_emailSender;

private readonly ISmsSender \_smsSender;

private readonly ILogger \_logger;

private readonly ApplicationDbContext \_context;

public IdentityController(

UserManager<ApplicationUser> userManager,

SignInManager<ApplicationUser> signInManager,

IEmailSender emailSender,

ISmsSender smsSender,

ILoggerFactory loggerFactory,

ApplicationDbContext context)

{

\_userManager = userManager;

\_signInManager = signInManager;

\_emailSender = emailSender;

\_smsSender = smsSender;

\_logger = loggerFactory.CreateLogger<IdentityController>();

\_context = context;

}

[HttpGet]

[AllowAnonymous]

public void Get()

{

var sports = \_context.Sports.Include(x => x.Associations);

var sport = \_context.Associations;

}

/// <summary>

/// Gets all the users (user role).

/// </summary>

/// <returns>Returns all the users</returns>

// GET api/identity/getall

[HttpGet("GetAll")]

public async Task<IActionResult> GetAll()

{

var claim = new Claim("role", "user");

var users = await \_userManager.GetUsersForClaimAsync(claim);

return new JsonResult(users);

}

/// <summary>

/// Registers a new user.

/// </summary>

/// <returns>IdentityResult</returns>

// POST: api/identity/register

[HttpPost]

[AllowAnonymous]

public async Task<IActionResult> Register([FromBody]RegisterViewModel model)

{

var user = new ApplicationUser

{

UserName = model.Username,

Email = model.Email

};

var claims = new IdentityUserClaim<string>[]

{

new IdentityUserClaim<string> { ClaimType = JwtClaimTypes.PreferredUserName, ClaimValue = model.Username },

new IdentityUserClaim<string> { ClaimType = JwtClaimTypes.Email, ClaimValue = model.Email },

new IdentityUserClaim<string> { ClaimType = JwtClaimTypes.Role, ClaimValue = "user" }

};

foreach (var claim in claims)

{

user.Claims.Add(claim);

}

var result = await \_userManager.CreateAsync(user, model.Password);

// Option: enable account confirmation and password reset.

return new JsonResult(result);

}

/// <summary>

/// Deletes a user.

/// </summary>

/// <returns>IdentityResult</returns>

// POST: api/identity/delete

[HttpPost("delete")]

public async Task<IActionResult> Delete([FromBody]string username)

{

var user = await \_userManager.FindByNameAsync(username);

var result = await \_userManager.DeleteAsync(user);

return new JsonResult(result);

}

// Add other methods.

}

}

Here, you have the mapping of paths and methods from identityService. I’m using the standard authentication from .NET, based on .NET identity.

The very same could be done for the sign in and sign out. We just need to generate and store JWT in our front-end Angular 2 application, and then use the stored token to make sure if the user is logged in or not.

You must try this guide. If you face any issue while implementing authentication with Angular 2 apps and ASP.NET Web API, ask me in the comments. Your feedback is valuable, so share your thoughts regarding this post.

http://www.c-sharpcorner.com/article/authentication-with-angular-2-app-front-end-and-asp-net-webapi-backend/

Weak session management and authentication is a cause for a web application's security flaw, which results in the failure to protect the session tokens and the credentials through their lifecycle. These flaws can lead to the stealing of the administrative or user accounts and privacy violations.

What is this vulnerability?

If authentication and session management are not implemented correctly, it allows the attackers to compromise session tokens or passwords, keys or to exploit the other implementation flaws to assume other users' identities. Authentication verifies the identity for the given credentials such as a username and password. Authentication and session management breaks the reasons such as insecure communication channels, password cracking etc.

Risks

Undermined authorization and accountability controls.

Cause privacy violation.

Identity theft.

We can implement authentication in two ways as below

Cookie Based Authentication(old approach), which uses the server side cookies to authenticate the user on each request.

Token Based Authentication(new approach) depends on a signed token, which is sent to the Server on each request.

Token based approach and Cookie based approach

As we know, when we use the cookie based approach, the generated ASP\_SessionId will be stored in the Browser cookies. When we use a token-based approach, the generated token will be stored in the Browser Cookies and with every request, the generated token will be passed as an attribute of the request header.

The following diagram explains how both the methods work:

diagram

Advantages of using a token-based approach

CORS

Cookies are a Token-based approach, which allows you to make the AJAX calls to any domain, any Server because you use HTTP header to transfer the user data.

Stateless (Server side scalability)

There is no need to keep a session store. The token is a self-contained entity, which passes all the user information and the rest of the state lives in the cookies or local storage on the client side.

CDN

You can refer to the required client side scripts from a CDN (e.g. JavaScript, HTML, images, etc.) and your Server side is just API.

Mobile ready

When you start working on a native platform like Windows 8, Android, iOS etc., cookies are not perfect, when consuming a secure API. Using a Token-based approach simplifies this a lot.

CSRF

As you are not depending on the cookies, you have no need to protect against cross site requests.

Standard-based

Your API methods can accept a JSON Web Token (JWT). This is standard and there are multiple frameworks like .NET, Ruby, Java, Python and PHP etc. For instance, Firebase allows its customers to use any authentication approach, as long as you produce a JSON Web Token with the certain pre-defined attributes and signed with a shared secret to call their API.

Advantages of using a cookie-based approach

Compared to Token-based authentication, I don't find any good advantages in Cookie-based authentication.

The Problems with Cookie Based Authentication

A few major problems may arise with this method of authentication.

Sessions

Scalability

CORS

CSRF

Malicious attack when we use cookie-based approach

Before starting the malicious attack example, we have to add few add-ons in the Browsers: Here, I prefer Firefox and Chrome Browsers for the example, given below:

Steps to add a few add-ons in browsers

Step 1: In Google, browse for “edit cookie in Firefox” and you will get many links, as given in the screenshot, below:

edit cookie in Firefox

I have added one add-on in Firebox Browser from the screenshot, shown above. After you add the add-on, you will find one option in the Browser cookie with the name “Edit”. Refer to the screenshot, given below:

edit cookie in Firefox

Step 2: This is the same way you have to do it for Chrome Browser.

In Google, browse for “edit cookie chrome”, you will get many links, as given below. From the screenshot, given y can add the top two add-ons.

edit cookie chrome

I have added the top two add-ons in Chrome Browser from the screenshot, shown above. After you add the add-ons, you will find two options in your Chrome Browser, as shown below:

browser

From the screenshot, shown above, you can select any option to edit your cookie. For reference, find the screenshot, given below, to see, how the editors will look:

browser

Now, it’s time to show you how to hijack the session key from one Browser. Let's say Firefox to another Browser.

Step 1

Let's say, you logged in with the user credential "Test1 / test123" in Firefox Browser, the unique session key will generate with the name "ASP.NET\_SessionId", make a note as each time the login session key will differ. After Login, how the session key looks is depicted below:

Login screenshot

Login

After Login

Login

After logging in to Firefox Browser, if you observe the screenshot, shown above, ASP.NET\_SessionId with the value “2e3yo3htzmddvi5uw3xzukls”got created in browser cookie.

Step 2

If you copy the URL from Firefox Browser, paste in Chrome Browser, hit enter key and it will send you to Login screen, as shown below:

Firefox browser URL: http://localhost:62887/EditPOInjection.aspx?POID=1

After pasting the URL, shown above, from Firefox Browser to Chrome Browser, it will send you to the login screen:

Login

If you observe the screenshot, shown above, the session Id is “z5egra3t0rbxymb1gu5qi4rf”, which is different from Firefox Browser. As the session Id is different that is why it automatically redirected to login screen.

Step 3

Now if you copy ASP.NET\_SessionId from Firefox browser and change the Chrome browser's session id with copied ASP.NET\_SessionId value and again request to the URL, given below, in Chrome browser then:

URL request

http://localhost:62887/EditPOInjection.aspx?POID=1

Screenshot after changes made for ASP.NET\_SessionId:

Login

URL request after ASP.NET\_SessionId changes made:

http://localhost:62887/EditPOInjection.aspx?POID=1

Output

output

Prevention mechanism for cookie-based token

Step 1

Generate a Hash key with the logged in Browser information like the Browser version, platform, Browser minimum version and Browser maximum version etc. In the code, given below, “GenerateHashKey()” method generates a Hash key for the logged in user and we are assigning the generated Hash key into one session variable with the name “SessionHashKey”. As we know, the session variable keeps the assigned value, until the session is in the active state.

code

Destination page code behind file

In our destination page code at the backend file, we are validating with the “SessionHashKey” session key value. If the request matches with the “SessionHashKey” session key value, it will redirect to the destination page. If it does not match with the “SessionHashKey” session key value, it will redirect to the login page.

code

Note: To prevent this type of attack, we can validate in all the pages. As “GenerateHashKey()” method is needed in all the pages to validate the request. We can place “GenerateHashKey()” method in our “Utility” project so that it will be available in all projects.

Malicious attack when we use token-based approach

If you are not using strongly typed encrypted token then you are giving an opportunity to decrypt the token. Once the token is decrypted then you lost your control and the hacker will take the control. So we have to use strong encryption mechanism like Bearer or JWT token with Sign mechanism.

Let’s take an example, if you are using weekly typed token with the token based basic authentication, see how the attacker can break it.

Here, I have written the custom attribute to validate the user:

code

In our login API method, we are generating a token by taking user logged in credential as an input.

code

Once the login is successful, in the client side success event, we are storing the generated token in the session storage variable, as shown below:

code

Now, we add this generated token in the header part of all the corresponding requests to validate the authenticated user. Sample code is given below:

code

When we execute API method and if you see in the Browser developer tool in the request header, we will find one property with the name “Authorization” with the generated token. Refer to the screenshot, given below, for reference:

reference

Now, if you change the generated token, using the Browser developer tool, it will send you to the login screen again. Screenshot to change the generated token, using the developer tool is given below:

login

Let's say, this token is generated in Firefox Browser for one logged in user. Now, login with the different user in Chrome Browser and change the generated token with Firefox token which is generated for different user login. It will allow you to perform CRUD operations, even though you are an not authentic user.

Prevention mechanism for token-based approach

Create a custom token and apply strong encryption mechanism, using MD5, SHA etc. With every request, validate the token by decrypting it in the code at the backend file and comparing it with the logged in user information. If everything is fine, process the given request otherwise respond with the proper warning message. Make sure, while creating your custom token, you are considering MAC Id also. If you consider MAC Id, while creating your custom token, it will prevent CSRF attack as well. The main idea to consider with MAC Id is, when we consider MAC Id; the hacker cannot hack your data either from the same system (where the authenticated user is working) or from the different system (from where the hacker is trying to hack).

http://www.c-sharpcorner.com/article/broken-authentication-and-session-management-and-its-prevention-mechanism/

The JWT AuthProvider is what Issues the JWT token which it populates based on the User Session. You can add your own metadata in the tokens and inspect it with the CreatePayloadFilter and PopulateSessionFilter.

JWT is enabled in both the AngularJS http://techstacks.io Example by just making a call to /session-to-token after the user successfully authenticates with their OAuth Provider, e.g:

$http.post("/session-to-token");

This converts their currently authenticated session into a JWT Token which it uses for future subsequent requests.

Likewise JWT is also used in http://gistlyn.com which uses a Customized JwtAuthProvider to embed the Github OAuth Access Token Secret into the JWT Token then uses the PopulateSessionFilter to extract it from the JWT Token and populate it back in the Users Session:

appHost.Plugins.Add(new AuthFeature(() => new AuthUserSession(),

new IAuthProvider[] {

new GithubAuthProvider(appHost.AppSettings),

//Use JWT so sessions survive across AppDomain restarts, redeployments, etc

new JwtAuthProvider(appHost.AppSettings)

{

CreatePayloadFilter = (payload, session) =>

{

var githubAuth = session.ProviderOAuthAccess.Safe()

.FirstOrDefault(x => x.Provider == "github");

payload["ats"] = githubAuth != null

? githubAuth.AccessTokenSecret : null;

},

PopulateSessionFilter = (session, obj, req) =>

{

session.ProviderOAuthAccess = new List<IAuthTokens>

{

new AuthTokens { Provider = "github", AccessTokenSecret = obj["ats"] }

};

}

},

}));

Gistlyn uses a similar approach to TechStacks to using JWT Tokens by calling /session-to-token after the User has authenticated with Github OAuth using JavaScript's new fetch API

fetch("/session-to-token", { method:"POST", credentials:"include" });

JWT Stateless Auth Tests

For other examples you can look at JWT RSA Tests which uses CreateJwtPayload which shows examples of manually creating JWT Tokens in code.

https://stackoverflow.com/questions/40109578/how-to-issue-and-consume-jwt-using-servicestacks-jwtauthprovider

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What Is a Token-Based Authentication System?

The token-based authentication systems allow users to enter their username and password in order to obtain a token which allows them to fetch a specific resource - without entering their username and password at each request. Once their token has been obtained, the user can use the token to access specific resources for a set time period.

JWT (pronounced 'jot') is a token based authentication system. It is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed using JSON Web Signature. The JWT is a self-contained token which has authentication information, expire time information, and other user defined claims digitally signed.

How We Used to Do Authentication

HTTP is a stateless protocol. That means it doesn't store any state from request to response. If you login for one request, you'll be forgotten and will need to login again to make another request. As you can imagine, this can get very annoying, very fast.

The old-school solution was to create what's called a "session." A session is implemented in two parts:

An object stored on the server that remembers if a user is still logged in, a reference to their profile, etc.

A cookie on the client-side that stores some kind of ID that can be referenced on the server against the session object's ID.Image title

This solution still works, but nowadays we have different requirements, i.e. hybrid application or single page application contacting multiple backends (split up into separate micro-service authetication servers, databases, image processing servers, etc). In these types of scenarios, the session cookie we get from one server won't correspond to another server.

Image title

Image title

JWTs don't use sessions and have no problem with micro-service architectures. Instead of making a session and setting a cookie, the server will send you a JSON Web Token instead. Now you can use that token to do whatever you want to do with the server (that you have authorization to do).

Think of it like a hotel key: you register at the front-desk, and they give you one of those plastic electronic keys with which you can access your room, the pool, and the garage, but you can't open other people's rooms or go into the manager's office. And, like a hotel key, when your stay has ended, you're simply left with a useless piece of plastic (i.e., the token doesn't do anything anymore after it's expired).

Advantages of JWTs

No Session to Manage (stateless): The JWT is a self contained token which has authetication information, expire time information, and other user defined claims digitally signed.

Portable: A single token can be used with multiple backends.

No Cookies Required, So It's Very Mobile Friendly

Good Performance: It reduces the network round trip time.

Decoupled/Decentralized: The token can be generated anywhere. Authentication can happen on the resource server, or easily separated into its own server.

https://dzone.com/articles/jwtjson-web-tokens-are-better-than-session-cookies

https://auth0.com/blog/cookies-vs-tokens-definitive-guide/

Token-Based Authentication

Token-based authentication has gained prevalence over the last few years due to rise of single page applications, web APIs, and the Internet of Things (IoT). When we talk about authentication with tokens, we generally talk about authentication with JSON Web Tokens (JWTs). While there are different ways to implement tokens, JWTs have become the de-facto standard. With this context in mind, the rest of the article will use tokens and JWTs interchangeably.

Token-based authentication is stateless. The server does not keep a record of which users are logged in or which JWTs have been issued. Instead, every request to the server is accompanied by a token which the server uses to verify the authenticity of the request. The token is generally sent as an addition Authorization header in form of Bearer {JWT}, but can additionally be sent in the body of a POST request or even as a query parameter. Let's see how this flow works:

User enters their login credentials

Server verifies the credentials are correct and returns a signed token

This token is stored client-side, most commonly in local storage - but can be stored in session storage or a cookie as well

Subsequent requests to the server include this token as an additional Authorization header or through one of the other methods mentioned above

The server decodes the JWT and if the token is valid processes the request

Once a user logs out, the token is destroyed client-side, no interaction with the server is necessary

Advantages of Token-Based Authentication

Understanding how something works is only half the battle. Next, we'll cover the reasons why token authentication is preferable over the traditional cookie based approach.

Stateless, Scalable and Decoupled

Perhaps the biggest advantage to using tokens over cookies is the fact that token authentication is stateless. The back-end does not need to keep a record of tokens. Each token is self-contained, containing all the data required to check its validity as well as convey user information through claims.

The server's only job then, becomes to sign tokens on a successful login request and verify that incoming tokens are valid. In fact, the server does not even need to sign tokens. Third party services such as Auth0 can handle the issuing of tokens and then the server only needs to verify the validity of the token.

https://hashedin.com/2016/07/05/choosing-right-authentication-for-rest-apis/

Is your API meant for a single page web application?

SPAs have two choices – Session based authentication and Token based authentication. Ask the following questions

Do you want to invalidate sessions before they expire? If yes, prefer sessions.

Do you want to expire sessions based on inactivity, as opposed to expiring after a fixed time? If yes, prefer sessions.

Do you want remember me functionality? If yes, prefer sessions.

Are you reusing the same APIs for a mobile application? If yes, prefer token based authentication (but consider building a separate API for these use cases)

Does your web framework protect against CSRF? If no, or if you don’t know what CSRF is – prefer token based authentication.

If you do decide to use token based authentication, avoid JSON Web Tokens. You should use JWT as a short, one-time token, and not as something that is reused multiple times. Instead of JWT, create a random token, store it in redis/memcached, and validate it on every request.

Q2. Is your API meant for Mobile Apps?

Prefer token based authentication.

Session based mechanisms are painful because a mobile app doesn’t automatically maintain and send the session cookie. For a mobile app developer, it is far easier to set an authentication token as opposed to setting a session cookie.

Signature based mechanisms are generally not useful because you cannot embed secret keys in a mobile application. If you have another channel to pass secrets to the mobile app – say via a QR code, then signature based mechanisms can be useful.

The ideal approach is a random token that is stored in a cache such as redis or memcached. Just remember to use a cryptographically secure random generator in your programming language.

Q3. Are you building a service that will be used by server side code only?

APIs that are called from a server-side application have 3 choices – basic authentication over https, token based authentication and signature based authentication.

Is your API meant for internal use only, and by applications you control, and the API themselves are low value? If yes, basic authentication is acceptable, as long as you are using HTTPS.

Do you want your keys to be long lived? If yes, prefer signature based authentication, because the keys themselves are never sent over the wire.

Do you want to reuse the same APIs with mobile or web apps? If yes, prefer token based authentication, because signature based auth requires clients to store secrets.

Are you using OAuth? If yes, the decision is made for you. OAuth 1 uses signature based authentication, whereas OAuth 2 uses token based authentication.

Do you provide a client library to access your APIs? If yes, prefer signature based auth, because you can then write the cryptography code once and provide it to all your clients.

Q4. When is it acceptable to use JSON Web Tokens?

JWT are best for single use tokens. You should ideally generate a new JWT for each use. Acceptable use cases:

Server-to-server API calls, where the client can store a shared secret and generate a new JWT for each API call.

Generate links that expire shortly – such as those used for email verification

As a way for one system to provide a logged in user limited access to another system.

Q5. Should I use OAuth for my APIs?

Only use OAuth if your APIs meet the following criteria:

Your API exposes consumer or user specific data

Third party developers that you don’t trust want access to some of the user specific data

You want to ask your users if they want to share their data with the third party developer

If the answer is “yes” to ALL the 3 questions, you require OAuth.

Q6. Is there a quick way to compare all the different authentication techniques?

The Authentication Techniques for APIs Spreadsheet has pros and cons for each approach.

JWT doesn't have a benefit over using "sessions" per say. JWTs provide a means of maintaining session state on the client in stead of doing it on the server.

What people often mean when asking this is "What are the benefits of using JWTs over using Server-side sessions"

With server-side sessions you will either have to store the session identifier in a database, or else keep it in memory and make sure that the client always hits the same server. Both of these have drawbacks. In the case of the database (or other centralised storage), this becomes a bottleneck and a thing to maintain - essentially an extra query to be done with every request.

With an in-memory solution you limit your horizontal scaling, and sessions will be affected by network issues (clients roaming between Wifi and mobile data, servers rebooting, etc)

Moving the session to the client means that you remove the dependency on a server-side session, but it imposes its own set of challenges.

- Storing the token securely

- transporting it securely

- JWTs Sessions can sometimes be hard to invalidate.

- Trusting the client's claim.

These issues are shared by JWTs and other client-side session mechanisms alike.

JWT in particular addresses the last of these. It may help to understand what a JWT is:

It is a bit of information. For user sessions you could include the username and the time when the token expires. But it could conceivably be anything, even the session ID or the user's entire profile. (Please don't do that though) It has got a secure signature that prevents malicious parties from generating fake tokens (You need access to the server's private key to sign them and you can verify that they were not modified after they were signed) You send them with every request, just like a cookie or Authorization Header would be sent. In fact they are commonly sent in the HTTP Authorization header but using a cookie is fine too.

The token is signed and so the server can verify its origin. We will assume that the server trusts its own ability to sign securely (you should use a standard library: don't try to do it yourself, and secure the server properly)

On the issue with securely transporting the token the answer is commonly to send it via an encrypted channel, usually httpS.

Regarding securely storing the token in the client, you need to ensure that the bad guys can't get to it. This (mostly) means preventing JS from bad web sites from reading the token to send it back to them. This is mitigated using the same strategies used to mitigate other kinds of XSS attacks.

If you have a need to invalidate JWTs there are definitely ways this can be achieved. Storing a per-user epoch for only users who have requested to have their "other sessions terminated" is a very efficient method that will probably be good enough. If a application needs per-session invalidation, then a session ID can be maintained in the same way and the "killed tokens" table can still be maintained to be much smaller than the full user table (You only need to retain records newer than the longest allowed token lifetime.) So the ability to invalidate the token partially negates the benefit of client-side sessions in that you would have to maintain this session killed state. This will more than likely be a much smaller table than the original session state table, so the lookups are still more efficient though.

One other benefit of using JWT tokens is that it is reasonably easy to implement using libraries available in probably every language you can expect to have it. It is also completely divorced from your initial user authentication scheme - if you move to a finger print based system you do not need to make any changes to the session management scheme.

A more subtle benefit: Because the JWT can carry "information" and this can be accessed by the client you can now start do some smart things. For example remind the user that their session will be expiring a few days before they are logged out, giving them the option to re-authenticate, based on the expiry date in the token. Whatever you can imagine.

So in short: JWTs answers some of the questions and shortcomings of other session techniques.

1. "Cheaper" authentication because you can eliminate a DB round trip (or at least have a much smaller table to query!), which in turns enable horizontal scalability.

2. Tamper-proof client-side claims.

While JWTs does not answer the other issues like secure storage or transport, it does not introduce any new security issues.

A lot of negativity exists around JWTs but if you implement the same security that you would for other types of authentication, you will be fine.

One final note: It is also not Cookies vs Tokens. Cookies is a mechanism for storing and transporting bits of information and can be used to store and transport JWT tokens too.

https://stackoverflow.com/questions/43452896/authentication-jwt-usage-vs-session

https://ponyfoo.com/articles/json-web-tokens-vs-session-cookies

Prosper from Auth0 approached us about writing a guest post on Pony Foo, and he’ll be addressing exactly those questions. 🍪

What Are JSON Web Tokens?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way to securely transmit information between parties as a JSON Object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with HMAC algorithm) or a public/private key pair using RSA.

JWT Anatomy

JWTs basically consist of three parts separated by a . . This is the header, payload and signature. Check out this excellent article for a comprehensive explanation of the JWT Structure.

How JSON Web Tokens Work

In authentication, when the user successfully logs in using his credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used), instead of the traditional approach of creating a session in the server and returning a cookie.

Whenever the user wants to access a protected route, it should send the JWT, typically in the Authorization header using the Bearer schema. Therefore, the content of the header should look like the following:

Authorization: Bearer <token>

This is a stateless authentication mechanism as the user state is never saved in the server memory. The server’s protected routes will check for a valid JWT in the Authorization header, and if it is there, the user will be allowed. As JWTs are self-contained, all the necessary information is there, reducing the need to go back and forth to the database.

This allows the user to fully rely on data APIs that are stateless and even make requests to downstream services. It doesn’t matter which domains are serving your APIs, as Cross-Origin Resource Sharing (CORS) won’t be an issue since it doesn’t use cookies.

Authentication flow

image description

image description

Why Should You Use JWTs

There are several reasons that you should use JSON Web Tokens.

They are easy to scale horizontally

They are easier to maintain and debug

They have the ability to create truly RESTful Services

They have built-in expiration functionality.

JSON Web Tokens are self-contained.

The points highlighted above will be explained in detail in the next section.

JWTs vs. Sessions

Before the emergence of JSON Web Tokens, we had the predominant server-based authentication. As we all know, HTTP Protocol is stateless, this means that if we authenticate a user with a username and password, then on the next request, our application won’t know who we are. We would have to authenticate again. So there was a need to ensure that after a user has logged in, the user’s authentication status can still be verified on every subsequent HTTP request.

image description

image description

A user’s credentials are sent as a POST request to the server. The server authenticates the user. If the credentials are valid, the server responds with a cookie, which is set on the user’s browser and includes a SESSION ID to identify the user. The user sessions are stored in memory either via files or in the database on the server. In this section, I’ll elaborate on several points that will be used as a basis for comparing JWTs with sessions in practice.

1. Scalability: As your application grows and your user base increases, you’ll have to start scaling either horizontally or vertically. Session data is stored in memory on the server either via files or in a database. In a horizontal scaling scenario, where you have to start replicating servers, you have to come up with a separate central session storage system that all of your application servers have access to. Otherwise, you won’t be able to scale your application because of the session-store drawback. Another way to solve this challenge is to use the concept of sticky sessions. You can also store your sessions on disk to make your application easy to scale in a cloud environment. These types of workarounds don’t really play well with modern large applications. Setting up and maintaining this type of distributed system involves in-depth technical knowledge and subsequently incurs higher financial costs. Using JWTs, in this case, is seamless; there is no need to store user information in the session since token-based authentication is stateless. Our application can scale easily because we can use tokens to access resources from different servers without worrying if the user was actually logged in on a particular server. You also save costs because you don’t need a dedicated server to store your sessions. Why? Because there are no sessions!

Note: If you are building small applications that absolutely don’t need to scale up to running on multiple servers and have no need for RESTful APIs, sessions will definitely work fine for you. And if you can use a dedicated server to run a tool like Redis for your session storage, then sessions might also work perfectly for you!

2. Security: Signing JWTs already aim to prevent tampering on the client side, but they can also be encrypted to ensure that the claim that the token carries is very secure. Now, JWTs are mostly either directly stored in web storage ( local/session storage) or in cookies. And JavaScript has access to web storage on the same domain. This simply means that your JWTs might be vulnerable to XSS (Cross-site Scripting). Malicious JavaScript can be embedded on a page to read and compromise the contents of your Web Storage. In fact, a lot of people advocate that very sensitive data shouldn’t be stored in Web Storage because of XSS attacks. A very typical example is ensuring that your JWTs are not encoded with very sensitive/trusted data, such as a user’s Social Security Number.

Initially, I mentioned that JWTs can be stored in cookies. In fact, JWTs are stored as cookies on many occasions, and cookies are vulnerable/susceptible to CSRF (Cross-site Request Forgery) attacks. One of the many ways to prevent CSRF attacks is to ensure that your cookie is accessible by only your domain. As a developer, ensure that necessary CSRF protections are put in place to avoid these attacks, regardless of the use of JWTs.

Now, JWTs and session ids can also be exposed to unmitigated replay attacks. It is totally up to the developers to establish what replay-mitigation techniques are appropriate for their systems. One way of solving this problem is ensuring that JWTs rely on short expiration times. Although, this technique doesn’t totally solve the problem. However, other alternatives for solving this challenge are issuing JWTs to specific IP addresses and using browser fingerprinting.

Note: Use HTTPS/SSL to ensure that your cookies and JWTs are encrypted by default during client and server transmission. This helps avoid man-in-the-middle attacks!

3. RESTful API Services: A common pattern for modern applications is to retrieve and consume JSON data from a RESTful API. Most applications these days have RESTful APIs for other developers or applications to consume. Serving data from an API has several distinct advantages, one of them which is the ability for data to be used in more than just one application. The traditional approach of using sessions and cookies for the user’s identity doesn’t work well in this case because they introduce the state to the application.

One of the tenets of a RESTful API is that it should be stateless, which means that when a request is made, a response within certain parameters can always be anticipated without side effects. A user’s authentication state introduces such a side effect, which breaks this principle. Keeping the API stateless and therefore without side effects means that maintainability and debugging are made much easier.

Another challenge here is that it is quite common for an API to be served from one server and for the actual application to consume it from another. To make this happen, we need to enable Cross-Origin Resource Sharing (CORS). Since cookies can only be used for the domain from which they originated, they aren’t much help for APIs on different domains than the application. Using JWTs for authentication in this case ensures that the RESTful API is stateless, and you also don’t have to worry about where the API or the application is being served from!

4. Performance: A critical analysis of this is very necessary. When making requests from the client to the server, if a lot of data is encoded within the JWT, it creates a significant amount of overhead with every HTTP request. However, with sessions, there is only a tiny amount of overhead because SESSION IDs are actually very small. Look at this example below:

A JWT has 5 claims like so:

{

"sub": "1234567890",

"name": "Prosper Otemuyiwa",

"admin": true,

"role": "manager",

"company": "Auth0"

}

When encoded, the size of the JWT will be several times the size of a SESSION ID (identifier), thus making this JWT add more overhead than a SESSION ID with every HTTP request. With sessions, there is also a server side lookup to find and deserialize the session on each request.

JWTs trade size for latency by keeping the data on the client side. The data model of your application is a significant factor here because latency is saved by preventing incessant calls and queries to the database on the server. The idea here is to be careful not to store too many claims in a JWT to avoid huge, over-bloated requests.

Worthy of mention is the fact that tokens may require access to the database on the backend. This is particularly the case for refresh tokens. They may require access to a database on the authorization server for blacklisting. Get more info about refresh tokens and when to use them. Also, check out this article for more information on blacklisting.

Note: Developers need to strike a balance to make the usage of JWTs really worth it!

image description

image description

Source: Quora

5. Downstream Services: Another common pattern seen with modern web applications is that they often rely on downstream services. For example, a call to the main application server might make a request to a downstream server before the original request is resolved. The issue here is that cookies don’t flow easily to the downstream servers and can’t tell those servers about the user’s authentication state. Since each server has its own scheme for cookies, there is a lot of resistance to flow, and connecting to them is difficult. JSON web tokens again makes these a breeze!

Authentication with Auth0 using JWTs

In Auth0, we issue JWTs as a result of the authentication process. When the user logs in using Auth0, a JWT is created, signed, and sent to the user. Auth0 supports signing JWT with both HMAC and RSA algorithms. The user has the flexibility to actually choose any of these two algorithms from the dashboard. This token will be then used to authenticate and authorize with APIs, which will grant access to their protected routes and resources.

We also use JWTs to perform authentication and authorization in Auth0’s API v2, replacing the traditional usage of regular opaque API keys. Regarding authorization, JSON Web Tokens allow granular security, which is the ability to specify a particular set of permissions in the token, thus improving debuggability.

Conclusion

JSON Web Tokens (JWTs) are lightweight and can easily be used across platforms and languages. They are a clever way to authenticate & authorize without sessions. There are several JWT libraries available for signing and verifying the tokens. There are also many reasons to use tokens, and Auth0 can help implement token authentication in an easy and secure way.

https://float-middle.com/json-web-tokens-jwt-vs-sessions/

The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure

The former gives us just the signature and the data it contains (or the "claims" as they call it in JWT nomenclature) is readable to anyone. The latter offers encryption, so only someone with a key can decrypt it. The JWS implementation is much easier and the basic usage does not require encryption - after all if you have a key on the client you may as well leave the whole thing unencrypted. Therefore JWS is used in most cases and so I am going to focus on it here.

So what goes into the JWT/JWS?

Header - information about the signing algorithm, the type of payload (JWT) and so on in JSON format.

Payload - the actual data (or claims if you like) in JSON format.

Signature - well... the signature.

I will explain the details later on. For now let's analyse the basics.

Each part mentioned above (header, payload and signature) is base64url-encoded, then they are glued together with a . to form JWT. Here is how the implementation could look like:

var header = {

// The signing algorithm.

"alg": "HS256",

// The type (typ) property says it's "JWT",

// because with JWS you can sign any type of data.

"typ": "JWT"

},

// Base64 representation of the header object.

headerB64 = btoa(JSON.stringify(header)),

// The payload here is our JWT claims.

payload = {

"name": "John Doe",

"admin": true

},

// Base64 representation of the payload object.

payloadB64 = btoa(JSON.stringify(payload)),

// The signature is calculated on the base64 representation

// of the header and the payload.

signature = signatureCreatingFunction(headerB64 + '.' + payloadB64),

// Base64 representation of the signature.

signatureB64 = btoa(signature),

// Finally, the whole JWS - all base64 parts glued together with a '.'

jwt = headerB64 + '.' + payloadB64 + '.' + signatureB64;

The resulting JWS looks neat and sweet and somewhat like this:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSm9obiBEb2UiLCJhZG1pbiI6dHJ1ZX0.OLvs36KmqB9cmsUrMpUutfhV52\_iSz4bQMYJjkI\_TLQ

You can play around with creating the tokens on jwt.io website.

Quite an important thing is that the the signature is calculated for both the header and the payload in one go. Thus the authenticity of the header and the payload can be easily checked in one go as well:

[headerB64, payloadB64, signatureB64] = jwt.split('.');

if (atob(signatureB64) === signatureCreatingFunction(headerB64 + '.' + payloadB64) {

// good

} else

// no good

}

What can be in the JWT header?

As a matter of fact, the JWT header is called JOSE header. JOSE stands for JSON Object Signing and Encryption. As you would expect, both JWS and JWE have such a header, however each has slightly different set of registered parameters. Below's the list of the header parameters registered for JWS. All except the first one (alg) are optional:

alg Algorithm (compulsory)

typ Type (for JWT it has a value JWT, if present)

kid Key ID

cty Content Type

jku JWK Set URL

jwk JSON Web Key

x5u X.509 URL

x5c X.509 Certificate Chain

x5t X.509 Certificate SHA-1 Thumbprint

x5t#S256 X.509 Certificate SHA-256 Thumbprint

crit Critical

The first two are the most commonly used, therefore the typical header looks somewhat along these lines:

{

"alg": "HS256",

"typ": "JWT"

}

The third header parameter listed above, kid, proves to be handy for security reasons. cty on the other hand should be used only when dealing with nested JWTs. The rest of them you can read upon in the spec, as I believe they don't fit in the scope of this post.

alg (algorithm)

The value of the alg parameter can be anything specified in JSON Web Algorithms (JWA) - yet another spec, I know. Here's the registered list for JWS:

HS256 - HMAC using SHA-256

HS384 - HMAC using SHA-384

HS512 - HMAC using SHA-512

RS256 - RSASSA-PKCS1-v1\_5 using SHA-256

RS384 - RSASSA-PKCS1-v1\_5 using SHA-384

RS512 - RSASSA-PKCS1-v1\_5 using SHA-512

ES256 - ECDSA using P-256 and SHA-256

ES384 - ECDSA using P-384 and SHA-384

ES512 - ECDSA using P-521 and SHA-512

PS256 - RSASSA-PSS using SHA-256 and MGF1 with SHA-256

PS384 - RSASSA-PSS using SHA-384 and MGF1 with SHA-384

PS512 - RSASSA-PSS using SHA-512 and MGF1 with SHA-512

none - No digital signature or MAC performed

Please note the last one, none, which is the most interesting from the security perspective. It's been known to be used for a downgrade attack angle. How does it work? Imagine a JWT is generated by the client with some made up claims. It specifies the none signature algorithm in the header and it sends it for verification. If the issuer is naive it takes the alg parameter as true and grants access where it shouldn't.

The baseline is, the security layer of your app should always be suspicious about the alg parameter from the header. That's where the kid comes handy.

typ (type)

This one's pretty straightforward. If it's known that it is a JWT, because the application does not expect anything else, this parameter would be ignored. Therefore it's optional. If specified though, it should be spelled in capitals - JWT.

In some cases when the app accepts non-JWTs containing JWT it's important to specify it, so that the app does not go bonkers.

kid (key id)

If the security layer in your app uses just one algorithm for signing the JWTs, you don't have to worry about the alg parameter, because you are always checking integrity of the token with the same key and algorithm. If however, your app uses a bunch of different algorithms and keys, you need to be able to figure out which the token was signed with.

As we saw earlier, relying on the alg parameter alone may lead to some... inconveniences. However, if your application maintained a list of key/algorithm pairs, and each of the pairs had a name (id), you could add that key id to the header and then during verification of the JWT you would have more confidence in picking the algorithm. That's what goes into the kid header parameter - the id of the key your app used to sign the token. The id is arbitrary and it's up to you to assign it. What's most important here - you gave the id, so you can verify it.

cty (content type)

The spec is pretty clear here, so I will just quote:

In the normal case in which nested signing or encryption operations are not employed, the use of this Header Parameter is NOT RECOMMENDED. In the case that nested signing or encryption is employed, this Header Parameter MUST be present; in this case, the value MUST be "JWT", to indicate that a Nested JWT is carried in this JWT. While media type names are not case sensitive, it is RECOMMENDED that "JWT" always be spelled using uppercase characters for compatibility with legacy implementations.

What can be in the JWT claims?

Doesn't the name "claims" bother you? It did bother me at first. I believe you need to repeat that a few times in order to get used to it. In simple terms, the claims are the meat of the JWT - this is the data that we care so much about as to sign it. It's called "claims" because usually that's what it is - the client claims that the username, user role or whatever else is such that it would grant him access to the resources he's after.

Remember that lovely story I told you at the beginning? Your citizenship was the claim and your passport - the JWT.

You can put whatever you wish in the claims, there is however a registered list, which should be universally recognised amongst implementations. Please note that each of them is optional and processing for most of them is application specific. Here's the list:

exp - Expiration Time

nbf - Not Before

iat - Issued At

sub - Subject

iss - Issuer

aud - Audience

jti - JWT ID

It's worth noting that except for the last three (issuer, audience and JWT ID) are usually used in more complex cases, e.g. with multiple issuers. Let's get on with it, then.

exp (expiration date)

Timestamp indicating when the token becomes invalid. The spec says that "the current date/time MUST be before" the value specified in the exp claim in order to allow processing of the token. It is also indicated that some leeway (a few minutes) is allowed in order to account for clock skew.

nbf (not before)

Timestamp indicating when the token becomes valid. The spec says that "the current date/time MUST be after or equal" the value specified in the nbf claim in order to allow processing of the token. It is also indicated that some leeway (a few minutes) is allowed in order to account for clock skew.

iat (issued at)

Timestamp indicating when the token has been issued.

sub (subject)

As the spec says "the claims in a JWT are normally statements about the subject". Subject must be unique within the context of the issuer or globally unique. The sub claim can be used to identify the user, for example JIRA does that.

iss (issuer)

String value identifying the issuer of the token. If the value contains : it has to be an URI. It can be useful if there are many issuers with one security layer and the application needs to identify the issuer. For example Salesforce require to use OAuth client\_id as the iss value.

aud (audience)

String or an array of strings identifying the intended recipient(s) of the token. If the string contains : it has to be an URI. Often used as an URI of the resource for which the claims are valid. For example, in OAuth the audience is the authorisation server. The application processing the token must verify that the audience is correct or reject the token if it is intended for different audience.

jti (JWT id)

Unique identifier for the token. This value must be unique for each issued token, even if there are many issuers. The jti claim can be used for one-time tokens, which cannot be replayed.

How to use JWT in my application?

In the most common scenario, the client in the browser will authenticate in the authentication service and receive JWT in return. Then the client stores the token somehow (e.g. memory, localStorage) and sends it back with every request for a protected resource. Usually the token is sent as a cookie or Authorization header in HTTP request:

GET /api/secured-resource HTTP/1.1

Host: example.com

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSm9obiBEb2UiLCJhZG1pbiI6dHJ1ZX0.OLvs36KmqB9cmsUrMpUutfhV52\_iSz4bQMYJjkI\_TLQ

The header method is preferred for security reasons - cookies would be susceptible to CSRF (Cross Site Request Forgery) unless CSRF tokens were used.

Secondly, the cookies can be sent back only to the same domain (or at most second level domain) they were issued from. If the authentication service resides on a different domain, cookies require much more wild creativeness.

How to log out with JWT?

Because there is no session date stored on the service side, logging out cannot be performed by destroying the session. Therefore logging out is the client's responsibility - as soon as the client forgets the token it cannot be authorised any more and therefore can be considered logged out.

Conclusion

I think JWTs are a very clever way of authorising without sessions. They allow for creating truly RESTful services with no state remembered on the service side, meaning no session storage is required either.

service Discovery

Secure Interprocess COmmunication

Load Balancing

Resilience

FAbric Model, Service Model, Proxy Model

Code Base, Dependencies, Config, Backing Services, Build/Release/Run, Processes, Data Isolation, Concurrency, Disposibility, Dev/Prod Parity, Logs, Admin Processes