**Building an OAuth2 Authorization Server with DotNetOpenAuth**

1. **Scenario**

* You’ve got an existing web app for which you’d like to expose a Web API for external apps (i.e., third-party apps) to integrate with.
* You’ve like to allow users of the existing web app to authorize these external apps to access their personal data via the Web API.
* You’d like to make the solution flexible enough to easily work with new APIs you might create in the future and with new apps that you might want to grant access to some of your APIs.
* You’ve got a single-sign-on process in place for your web apps that you’d like to continue to work with, to avoid users needing to maintain many sets of credentials.

The scenario described here fits the intention of the OAuth2 protocol very well. This is the protocol that is typically in play when, for example, you grant an application access to display your Twitter feeds. As an application developer, you first register your application with Twitter and receive some credentials that are unique to your app. Then, when a user is interacting with your app and wants to see their Twitter feeds, your app first submits an authorization request to Twitter. Twitter then provides a login page to prompt the user to enter their Twitter credentials, and after doing so, gives the user the opportunity to authorize your application to access his/her Twitter data. The result of this process is ultimately an access token that is returned to your app, and then your app sends back this token with subsequent requests to Twitter’s API to facilitate access to the user’s data.

To support the scenario outlined above, we need to stand up what’s known in OAuth2 terminology as an authorization server (as Twitter has done). This is the mechanism will accepts requests from clients to access our resources, optionally on behalf of a user.

This post is not intended to be a comprehensive description of OAuth2. If you’re not too familiar with it, please refer to Appendix A at the end before continuing.

1. **Authorization Server Options**

As I started to look into how to go about putting an authorization server in place, I encountered a couple of good candidates to help me out. One of these was a fully functional authorization server from Thinktecture, who seem to be quite knowledgeable in this space. Unfortunately, their solution is built on top of the new features of .NET 4.5, and I was constrained to use .NET 4. The other library I found was DotNetOpenAuth, which is what is being used under the covers when you create an ASP.NET application that uses other identity providers. It hides most of the guts of the OAuth2 protocol specifics, although you still definitely need to know how the protocol flows. Unfortunately, I found the DotNetOpenAuth samples around building an authorization server to be a bit confusing and cluttered.

In the end, I decided to start with one of the DotNetOpenAuth samples and extend it to build a more modularized proof-of-concept solution. While I did attempt to build in some flexibility for the authorization server implementation, I didn’t go the whole nine yards of fully abstracting into interfaces, building unit tests, building a stylish UI, and the like. The scope of the initial effort was to provide a proof-of-concept with minimal clutter to demonstrate that such a solution could be implemented. The remainder of this summary will walk through the major building blocks of this solution and attempt to better explain how the various pieces fit together. I’ll start with the project containing the core of the authorization server, including the data that it needs to do its thing.

1. **Authorization Server Implementation**

First, let’s outline the major players in the implementation of the authorization server. You’ll see there’s not really a ton of code; we’re helped a lot by DotNotOpenAuth.

* 1. **OAuth.AuthorizationServer.Core**

This project contains the main logic of the authorization server process, along with the data model used to track clients, resources, and other information needed by the authorization server

* + 1. **Data**

The following classes are the primary components of the data model.

* Client: Represents a client (external app) that is recognized by the authorization server
  + Id: Client identifier, used in requests from client
  + ClientSecret: Secret used to verify client
  + Callback: Optional, if specified enforces a match during the code/implicit flows
  + Name: Display name of client, used when user authorizes the client
  + Scopes: Scopes this client is allowed to use
* Resource: Represents a resource (e.g., API) recognized by the authorization server
  + AuthenticationTokenName: Token that identifies whether user has authenticated with the resource
  + AuthenticationUrl: Where the authorization server should redirect the user to authenticate with the resource
  + AuthenticationKey: Shared secret with resource login mechanism to verify token provided is really from there
  + PublicTokenEncryptionKey: Encryption key for resource server to use in encrypting the authorization token. Resource server has corresponding private key to decrypt
  + Scopes: Scopes supported by this resource
* Scope: Represents a scope recognized by the authorization server. Allows us to relate clients with scopes they are allowed to use, and to relate resources with scopes they support.
  + Identifier: This is the name of the scope that gets passed around in requests
  + Description: For display to user on authorization page to describe what this scope allows the client to do
* User: Represents a user recognized by the authorization server
  + Id: User name, comes from resource authentication process
* Authorization: Represents an authorization for a client to access a particular resource, optionally by a particular user
  + ClientId: Client being authorized
  + UserId: Authorizing used
  + ResourceId: Resource being authorized
  + Scope: This is a space-delimited list of scopes, for simplicity here
* Nonce, SymmetricCryptoKey: Used internally by DotNetOpenAuth
* OAuthDataContext, OAuthDbInitializer: EntityFramework hooks

The authorization server accesses the model by means of repository classes, some of when cache the read-only data to limit database accesses. There’s not too much interesting going on in those classes to describe further here.

* + 1. **Server**

The primary logic of our authorization server is housed in an implementation of the DotNetOpenAuth IAuthorizationServerHost interface. The members implemented are used by the DotNetOpenAuth AuthorizationServer class during its request processing. I’ll include the code for the class here; hopefully the included comments adequately describe what’s going on. Much of the content is the similar to that of the DotNetOpenAuth sample. An important aspect to note is that when an access token is created, it uses the public key of the intended resource to encrypt the token and the private key of the authorization server to sign the token. When the protected resource accepts a token, it will use its private key to decrypt the token and the authorization server’s public key to verify its signature. These steps verify that the token came from our authorization server and is intended to allow access to the resource receiving the token.

// Our implementation of the authorization server, which is passed to the ctor of the

// DotNetOpenAuth AuthroizationServer class. The DotNetOpenAuth impl calls methods in this

// class when it sees fit.

public class AuthorizationServerHost : IAuthorizationServerHost

{

// Ideally, IOC these dependencies. OK, ideally, refactor this a bit more, although

// we're our hands are tied somewhat by the IAuthorizationServerHost interface

private readonly ClientRepository \_clientRepository = new ClientRepository();

private readonly ResourceRepository \_resourceRepository = new ResourceRepository();

private readonly ICryptoKeyStore \_cryptoKeyRepository = new SymmetricCryptoKeyRepository();

private readonly INonceStore \_nonceRepository = new NonceRepository();

private readonly AuthorizationRepository \_authorizationRepository = new AuthorizationRepository();

private readonly AuthorizationServerSigningKeyManager \_tokenSigner = new

AuthorizationServerSigningKeyManager();

public ICryptoKeyStore CryptoKeyStore { get { return \_cryptoKeyRepository; } }

public INonceStore NonceStore { get { return \_nonceRepository; } }

// Generate an access token, given parameters in request that tell use what scopes to include,

// and thus what resource's encryption key to use in addition to the authroization server key

public AccessTokenResult CreateAccessToken(IAccessTokenRequest accessTokenRequestMessage)

{

// could parameterize lifetime

var accessToken = new AuthorizationServerAccessToken {Lifetime = TimeSpan.FromMinutes(10)};

var targetResource = \_resourceRepository.FindWithSupportedScopes(accessTokenRequestMessage.Scope);

accessToken.ResourceServerEncryptionKey = targetResource.PublicTokenEncrypter;

accessToken.AccessTokenSigningKey = \_tokenSigner.GetSigner();

var result = new AccessTokenResult(accessToken);

return result;

}

// Lookup client given an identifier

public IClientDescription GetClient(string clientIdentifier)

{

IClientDescription client = \_clientRepository.GetById(clientIdentifier);

if (client == null)

{

throw new ArgumentOutOfRangeException("clientIdentifier");

}

return client;

}

// Determine whether the given authorization is still ok

public bool IsAuthorizationValid(IAuthorizationDescription authorization)

{

// If db precision exceeds token time precision (which is common), the following query would

// often disregard a token that is minted immediately after the authorization record is

// stored in the db.

// To compensate for this, we'll increase the timestamp on the token's issue date by 1 second.

var grantedAuths = \_authorizationRepository.FindCurrent(

authorization.ClientIdentifier, authorization.User,

authorization.UtcIssued + TimeSpan.FromSeconds(1)).ToList();

if (!grantedAuths.Any())

{

// No granted authorizations prior to the issuance of this token,

// so it must have been revoked.

// Even if later authorizations restore this client's ability to call in, we can't allow

// access tokens issued before the re-authorization because the revoked authorization should

// effectively and permanently revoke all access and refresh tokens.

return false;

}

// Determine the set of all scopes the user has authorized for this client

var grantedScopes = new HashSet<string>(OAuthUtilities.ScopeStringComparer);

foreach (var auth in grantedAuths)

{

grantedScopes.UnionWith(OAuthUtilities.SplitScopes(auth.Scope));

}

// See if what's requested is authorized

return authorization.Scope.IsSubsetOf(grantedScopes);

}

// Used during client credentials flow. Before we get here, the client and secret will

// already have been verified

// We're also ensuring the scopes requested are ok to give the client

public AutomatedAuthorizationCheckResponse CheckAuthorizeClientCredentialsGrant(

IAccessTokenRequest accessRequest)

{

// Find the client

var client = \_clientRepository.GetById(accessRequest.ClientIdentifier);

// Determine the scopes the client is authorized for

var scopesClientIsAuthorizedFor = client.SupportedScopes;

// Check if the scopes that are being requested are a subset of the scopes the user is

// authorized for.

// If not, that means that the user has requested at least one scope it is not authorized for

var clientIsAuthorizedForRequestedScopes =

accessRequest.Scope.IsSubsetOf(scopesClientIsAuthorizedFor);

// The token request is approved when the client is authorized for the requested scopes

var isApproved = clientIsAuthorizedForRequestedScopes;

return new AutomatedAuthorizationCheckResponse(accessRequest, isApproved);

}

public AutomatedUserAuthorizationCheckResponse CheckAuthorizeResourceOwnerCredentialGrant(

string userName, string password, IAccessTokenRequest accessRequest)

{

// Not supporting this flow, as it's not normally a good idea to have user give

// their credentials directly to the client

throw new **NotImplementedException**();

}

}

The authorization server built here is used by the endpoint that clients use to obtain an access token to use when accessing our resources. We’ll look at that endpoint next.

* 1. **OAuth.AuthorizationServer.API**

This project provides client applications an interface into the authorization server. There are a couple of key pieces worth detailing.

* + 1. **Endpoint**

The endpoint used by clients is an MVC Controller class. When the sample client is described later you’ll see how this endpoint is used. The controller consists of three actions:

* Token: Used in the client credentials flow and the second half of the code flow when exchanging an access code for a token
* Authorize: Used in the first half of the code flow and the implicit flow so the user can authorize the client. Note that this method is decorated with a ResourceAuthenticated attribute (described in next section) to ensure that only users that have authenticated with the resource can get inside. Presents the user with a form that shows which scopes the client is requesting and allows the user to allow or deny the request.
* ProcessAuthorization The form in the Authorize view (which indicated whether the user decided to allow the request) posts back to this action to grant or deny the authorization, and redirect the browser back to the client’s specified redirect URL.

The code for the controller is included here; hopefully the included comments adequately describe what’s going on.

// Exposed endpoint by which clients can request access to a resource via the OAuth2 protocol

public class OAuthController : Controller

{

// Ideally, IOC these dependencies.

private readonly DNOA.AuthorizationServer \_authorizationServer =

new DNOA.AuthorizationServer(new AuthorizationServerHost());

private readonly ClientRepository \_clientRepository = new ClientRepository();

private readonly AuthorizationRepository \_authorizationRepository = new AuthorizationRepository();

private readonly ResourceRepository \_resourceRepository = new ResourceRepository();

private readonly UserRepository \_userRepository = new UserRepository();

// Provides authorization token to the client based on information in the request

// DotNetOpenAuth is doing all the heavy lifting here. Request must contain all of the

// necessary info to grant a token

public ActionResult Token()

{

return \_authorizationServer.HandleTokenRequest(Request).AsActionResult();

}

// Prompts the user to authorize a client to access the user's private data.

// If user is not already authenticated by the resource, user will be redirected to login first

// and then come back here to authorize the client

[ResourceAuthenticated, AcceptVerbs(HttpVerbs.Get | HttpVerbs.Post)]

public ActionResult Authorize()

{

// Have DotNetOpenAuth read the info we need out of the request

EndUserAuthorizationRequest pendingRequest = \_authorizationServer.ReadAuthorizationRequest();

if (pendingRequest == null)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest),

"Missing authorization request.");

}

// Make sure the client is one we recognize

Client requestingClient = \_clientRepository.GetById(pendingRequest.ClientIdentifier);

if (requestingClient == null)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest), "Invalid request");

}

// Ensure client is allowed to use the requested scopes

if (!pendingRequest.Scope.IsSubsetOf(requestingClient.SupportedScopes))

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest), "Invalid request");

}

// Consider auto-approving if safe, so user doesn't have to authorize repeatedly.

// Leaving this step out for now

// Show user the authorization page by which they can authorize this client to access their

// data within the resource determined by the requested scopes

var model = new AccountAuthorizeModel

{

Client = requestingClient,

Scopes = requestingClient.Scopes.Where(x =>

pendingRequest.Scope.Contains(x.Identifier)).ToList(),

AuthorizationRequest = pendingRequest

};

return View(model);

}

// Processes the user's response as to whether to authorize a Client to access

// his/her private data.

[ResourceAuthenticated(Order = 1), HttpPost, ValidateAntiForgeryToken(Order = 2)]

public ActionResult ProcessAuthorization(bool isApproved)

{

// Have DotNetOpenAuth read the info we need out of the request

EndUserAuthorizationRequest pendingRequest = \_authorizationServer.ReadAuthorizationRequest();

if (pendingRequest == null)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest),

"Missing authorization request.");

}

// Make sure the client is one we recognize

Client requestingClient = \_clientRepository.GetById(pendingRequest.ClientIdentifier);

if (requestingClient == null)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest), "Invalid request");

}

// Make sure the resource is defined, it definitely should be due to the

// ResourceAuthenticated attribute

Resource requestedResource =

\_resourceRepository.FindWithSupportedScopes(pendingRequest.Scope);

if (requestedResource == null)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest), "Invalid request");

}

// See if authorization of this client was approved by the user

// At this point, the user either agrees to the entire scope requested by the client or

// none of it.

// If we gave capability for user to reduce scope to give client less access, some changes

// would be required here

IDirectedProtocolMessage authRequest;

if (isApproved)

{

// Add user to our repository if this is their first time

var requestingUser = \_userRepository.GetById(User.Identity.Name);

if (requestingUser == null)

{

requestingUser = new User

{ Id = User.Identity.Name, CreateDateUtc = DateTime.UtcNow };

\_userRepository.Insert(requestingUser);

\_userRepository.Save();

}

// The authorization we file in our database lasts until the user explicitly revokes it.

// You can cause the authorization to expire by setting the ExpirationDateUTC

// property in the below created ClientAuthorization.

\_authorizationRepository.Insert(new Authorization

{

ClientId = requestingClient.Id,

Scope = OAuthUtilities.JoinScopes(pendingRequest.Scope),

UserId = requestingUser.Id,

ResourceId = requestedResource.Id,

CreatedOnUtc = DateTime.UtcNow

});

\_authorizationRepository.Save();

// Have DotNetOpenAuth generate an approval to send back to the client

authRequest = \_authorizationServer.PrepareApproveAuthorizationRequest(

pendingRequest, User.Identity.Name);

}

else

{

// Have DotNetOpenAuth generate a rejection to send back to the client

authRequest = \_authorizationServer.PrepareRejectAuthorizationRequest(pendingRequest);

// The PrepareResponse call below is giving an error of "The following required parameters

// were missing from the DotNetOpenAuth.OAuth2.Messages.EndUserAuthorizationFailedResponse

// message: error" unless I do this.....

var msg = (EndUserAuthorizationFailedResponse) authRequest;

msg.Error = "User denied your request";

}

// This will redirect to the client app using their defined callback, so they can handle

// the approval or rejection as they see fit

return \_authorizationServer.Channel.PrepareResponse(authRequest).AsActionResult();

}

}

* + 1. **ResourceAuthenticatedAttribute**

As mentioned earlier, an attribute is placed on the authorization methods of the OAuthController to ensure the user has been authenticated with the resource. The OAuth2 spec does not include any mechanism for authentication, it solely focuses on authorization. This attribute checks for a token that came from the resource’s login mechanism, and if not found, redirects the browser there so the user can authenticate. After the user is authenticated, it is assumed the login mechanism provides a token indicating such and redirects back to the action on which this attribute lies. So there is some coupling here that forces the authorization server to be able to understand the format of the authentication token provided by the resource. The code for this attribute is shown below.

// This authorization attribute is applied to the authorization methods in our OAuthController

// to ensure the user has been authenticated by the resource being requested

public class ResourceAuthenticatedAttribute : AuthorizeAttribute

{

private readonly DNOA.AuthorizationServer \_authorizationServer =

new DNOA.AuthorizationServer(new AuthorizationServerHost());

private readonly ResourceRepository \_resourceRepository = new ResourceRepository();

private Resource \_targetResource;

protected override bool AuthorizeCore(HttpContextBase httpContext)

{

// Figure out what resource the request is intending to access to see if the

// user has already authenticated to with it

EndUserAuthorizationRequest pendingRequest = \_authorizationServer.ReadAuthorizationRequest();

if (pendingRequest == null)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest),

"Missing authorization request.");

}

try

{

\_targetResource = \_resourceRepository.FindWithSupportedScopes(pendingRequest.Scope);

// Above will return null if no resource supports all of the requested scopes

if (\_targetResource == null)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest),

"Bad authorization request.");

}

}

catch (Exception)

{

throw new HttpException(Convert.ToInt32(HttpStatusCode.BadRequest),

"Bad authorization request.");

}

// User is considered authorized if in possession of token that originated from the

// resource's login page,

// Name of token is determined by the resource configuration

string tokenName = \_targetResource.AuthenticationTokenName;

// token could be in cookie if previously logged in, or querystring if just logged in

string encryptedToken = httpContext.Request[tokenName];

if (string.IsNullOrWhiteSpace(encryptedToken))

{

// No token, so unauthorized

return false;

}

// Validate this thing came from us via shared secret with the resource's login page

// The implementation here ideally could be generalized a bit better or standardized

string encryptionKey = \_targetResource.AuthenticationKey;

string decryptedToken = EncodingUtility.Decode(encryptedToken, encryptionKey);

string[] tokenContentParts = decryptedToken.Split(';');

string name = tokenContentParts[0];

DateTime loginDate = DateTime.Parse(tokenContentParts[1]);

bool storeCookie = bool.Parse(tokenContentParts[2]);

if ((DateTime.Now.Subtract(loginDate) > TimeSpan.FromDays(7)))

{

// Expired, remove cookie if present and flag user as unauthorized

httpContext.Response.Cookies.Remove(tokenName);

return false;

}

// Things look good.

// Set principal for the authorization server

IIdentity identity = new GenericIdentity(name);

httpContext.User = new GenericPrincipal(identity, null);

// If desired, persist cookie so user doesn't have to authenticate with the resource

// over and over

var cookie = new HttpCookie(tokenName, encryptedToken);

if (storeCookie)

{

cookie.Expires = DateTime.Now.AddDays(7); // could parameterize lifetime

}

httpContext.Response.AppendCookie(cookie);

return true;

}

protected override void HandleUnauthorizedRequest(AuthorizationContext filterContext)

{

// Unauthorized, redirect user to the target resource's login page, telling it to

// redirect back here when complete

filterContext.Result = new RedirectResult(string.Format("{0}?returnUrl={1}",

\_targetResource.AuthenticationUrl, new

UrlHelper(filterContext.RequestContext).Encode(

filterContext.RequestContext.HttpContext.Request.Url.ToString())));

}

}

* + 1. **SSL Requirements**

As mentioned earlier, for development purposes, the SSL requirements for DotNetOpenAuth can be relaxed. This is done via the relaxSslRequirements attribute on the messaging element of the dotNetOpenAuth element in web.config, as see below.

<dotNetOpenAuth>

<oauth2>

<authorizationServer></authorizationServer>

</oauth2>

<!-- Relaxing SSL requirements is useful for simple samples, but NOT a good idea in production. -->

<messaging relaxSslRequirements="true">

<untrustedWebRequest>

<whitelistHosts>

<!-- since this is a sample, and will often be used with localhost -->

<add name="localhost" />

</whitelistHosts>

</untrustedWebRequest>

</messaging>

</dotNetOpenAuth>

I repeat, **DO NOT** deploy code to production with the attribute set to true.

This completes the authorization server itself. Now we need a mechanism to allow our Web API resource to handle the access tokens that are generated by the authorization server. That will be described next.

* 1. **OAuth.ResourceServer.Core**

This project has only one item of major interest, namely an attribute named OAuthAuthorizeScopeAttribute that can be applied to Web API methods to indicate which scope(s) are required to access it. It uses the public key of the authorization server and a private encryption key to ensure validity of the token, and then checks that the scopes contained in the token match up with those that the method requires for access. The DotNetOpenAuth samples seem to implement this process instead in a DelegatingHandler and use URIs for scopes and match on the resource target URI. This version allows us to use scopes more like roles and apply attributes with the same scope name to various parts of our API. The code for this attribute is shown below.

// Attribute to apply to methods on WebAPI controller methods to restrict access to

// those in possession of an authorization token with specified scopes

public class OAuthAuthorizeScopeAttribute : AuthorizeAttribute

{

private static readonly RSACryptoServiceProvider Decrypter;

private static readonly RSACryptoServiceProvider SignatureVerifier;

// Get the keys from wherever they are stored

static OAuthAuthorizeScopeAttribute()

{

Decrypter = new ResourceServerKeyManager().GetDecrypter();

SignatureVerifier = new AuthorizationServerKeyManager().GetSignatureVerifier();

}

// Which scopes are required to gain access

public string[] RequiredScopes { get; set; }

public OAuthAuthorizeScopeAttribute(params string[] requiredScopes)

{

RequiredScopes = requiredScopes;

}

public override void OnAuthorization(HttpActionContext actionContext)

{

try

{

base.OnAuthorization(actionContext);

// Bail if no auth header or the header isn't bearing a token for us

var authHeader = actionContext.Request.Headers.FirstOrDefault(x =>

x.Key == "Authorization");

if (authHeader.Value == null || !authHeader.Value.Any())

{

actionContext.Response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

return;

}

var authHeaderValue = authHeader.Value.FirstOrDefault(x => x.StartsWith("Bearer "));

if (authHeaderValue == null)

{

actionContext.Response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

return;

}

// Have the DotNetOpenAuth resource server inspect the provided request using the

// configured keys

// This checks both that the token is ok and that the token grants the scope required by

// the required scope parameters to this attribute

var resourceServer = new DNOA.ResourceServer(

new StandardAccessTokenAnalyzer(SignatureVerifier, Decrypter));

var principal = resourceServer.GetPrincipal(actionContext.Request,RequiredScopes);

if (principal != null)

{

// Things look good. Set principal for the resource to use in identifying the user so

// it can act accordingly

Thread.CurrentPrincipal = principal;

HttpContext.Current.User = principal;

// Dont understand why the call to GetPrincipal is setting actionContext.Response

// to be unauthorized even when the principal returned is non-null

// If I do this code the same way in a delegating handler, that doesn't happen

actionContext.Response = null;

}

else

{

actionContext.Response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

}

}

catch (SecurityTokenValidationException)

{

actionContext.Response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

}

catch (ProtocolFaultResponseException)

{

actionContext.Response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

}

catch (Exception)

{

actionContext.Response = new HttpResponseMessage(HttpStatusCode.BadRequest);

}

}

}

1. **Sample Client/Resource Implementations**

To show usage of the authorization server code, I built sample implementations of resources protected by OAuth2 tokens and clients that obtain access to these resources.

* 1. **Resource Implementation**

Project Resource1API consists of a very simple ASP.NET Web API. There’s not much to see here, other than that controller actions are protected by the OAuthAuthorizeScopeAttribute described earlier. The resource can then use the User.Identity.Name value to determine what user is making the request and act accordingly. The encryption keys needed by the attribute are included via web.config. Here’s a sample:

// Sample controller with methods accessible by those with Resource1-Read or Resource1-Write scopes

public class NumberController : ApiController

{

private static int \_number;

[OAuthAuthorizeScope("Resource1-Read")]

public IEnumerable<string> Get()

{

// passing the name back just to show we've got it

return new [] { User.Identity.Name, \_number.ToString(CultureInfo.InvariantCulture) };

}

[OAuthAuthorizeScope("Resource1-Write")]

public void Put(int id, int value)

{

\_number = value;

}

}

Project Resource2API is the same code with a different resource encryption key, to demonstrate accessing multiple resources via the same authorization server.

* 1. **Client Implementation**

Project Client1 is an ASP.NET MVC project exhibiting how a client might go about accessing a resource via the code flow, implicit flow, and client credentials flow. A single client wouldn’t normally implement all three of these flows like this (especially the implicit flow on the server side), but I wanted to show them all without building separate client apps.

The project has 2 controllers of interest. The MainController is the one that’s primarily responsible for accessing the resource after a token has been obtained. There’s not too much interesting in there except for how the token is included in the request to the resource. The code for the MainController is shown below.

public class MainController : Controller

{

private static int \_number = 1;

// Show user main screen indicating their authorization status and what they can do

// in current state

public ActionResult Index(string msg)

{

var tokenCookie = Request.Cookies[ClientConfig.AccessTokenName];

return View(new TokenInfoModel { TokenCookie = tokenCookie, Message = msg });

}

// Remove current authorization token so reauthorization process can happen

public ActionResult RemoveToken()

{

var tokenCookie = Request.Cookies[ClientConfig.AccessTokenName];

if (tokenCookie != null)

{

tokenCookie.Expires = DateTime.Now.AddDays(-1);

Response.Cookies.Add(tokenCookie);

}

return View();

}

// Make a call to an api using the authorization token

public ActionResult CallApi(string apiAction)

{

var tokenCookie = Request.Cookies[ClientConfig.AccessTokenName];

if (tokenCookie == null)

{

// Need to get a token

return View("Index", new TokenInfoModel {

TokenCookie = null, Message = "Token is missing. It likely expired."});

}

// Assemble the API URL. This won't be as ugly for real apps.

var relativePath = apiAction.Contains("Number") ? "Number" : "Shape";

var method = apiAction.Contains("Update") ? "PUT" : "GET";

var baseUrl = apiAction.Contains("Resource1")

? ResourceConfig.Resource1BaseUrl : ResourceConfig.Resource2BaseUrl;

if (method == "PUT")

{

relativePath += "/1?value=" + \_number++;

}

// Submit the request to the API with our token

var req = WebRequest.Create(baseUrl + (baseUrl.EndsWith("/") ? "" : "/") + relativePath);

req.Headers.Add("Authorization", "Bearer " + tokenCookie.Value);

req.Method = method;

req.ContentLength = 0;

string content;

try

{

var resp = req.GetResponse();

content = new StreamReader(resp.GetResponseStream()).ReadToEnd();

}

catch (WebException ex)

{

content = String.Format("An error occurred, message was {0}", ex.Message);

}

return View((object)content);

}

}

}

The other controller is the TokenRequestController, which is responsible for obtaining an access token from the authorization server. It starts with a simple page which allows specification of scopes and the flow to use. Normally the user of the client app wouldn’t directly specify these things like this; instead they would be baked into the client app based on its knowledge of the resource it’s accessing. Depending on which flow is selected, the appropriate action is invoked on the controller to start the flow. This code interacts with the client side piece of the DotNotOpenAuth library to implement the code and client credentials flows, and you can see how the callbacks on the code and implicit flows maps back to corresponding actions on this controller to process the results. Code for this controller is shown below.

public class TokenRequestController : Controller

{

// Show user view to allow selection of scopes and flow

public ActionResult Index()

{

return View();

}

// Kick off desired flow

public ActionResult StartFlow(string[] scopes, string flowType)

{

var scopeList = scopes ?? Enumerable.Empty<string>();

if (flowType == "code")

{

return invokeCodeFlow(scopeList);

}

else if (flowType == "implicit")

{

return invokeImplicitFlow(scopeList);

}

else if (flowType == "clientCredentials")

{

return invokeClientCredentialsFlow(scopeList);

}

else

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

}

#region CodeFlow

// First step in code flow is getting an access code, which will be passed to our

// callback action on this controller

// Note that the user will be prompted by the authorization server to authorize the

// client (and login to the resource if necessary)

// before control is given to our callback

private ActionResult invokeCodeFlow(IEnumerable<string> scopes)

{

var state = new AuthorizationState();

foreach (var s in scopes)

{

state.Scope.Add(s);

}

state.Callback = new Uri(Request.Url, Url.Action("ExchangeAccessCodeForAuthToken",

"TokenRequest"));

// Here DotNetOpenAuth figures out what the request should look like

// (i.e., builds appropriate url)

var r = ClientConfig.AuthorizationServerClient.PrepareRequestUserAuthorization(state);

return r.AsActionResult();

}

// When this callback is invoked in the code flow, the user has chosen to approve or deny

// access for this client

// If approved, store token in cookie for use by the rest of the app

public ActionResult ExchangeAccessCodeForAuthToken()

{

var accessTokenResponseState =

ClientConfig.AuthorizationServerClient.ProcessUserAuthorization(this.Request);

var gotToken = accessTokenResponseState.AccessToken != null;

if (gotToken)

{

var cookie = new HttpCookie(ClientConfig.AccessTokenName,

accessTokenResponseState.AccessToken)

{Expires = accessTokenResponseState.AccessTokenExpirationUtc.Value, Path = "/"};

Response.Cookies.Add(cookie);

}

return RedirectToAction("Index", "Main", new {

msg = gotToken ? "Token Granted" : "No Access Token Was Granted"});

}

#endregion

#region Implicit Flow

// Here we request the token directly, which is passed back on a url fragment

// Typically this flow wouldn't be used from the server side and would instead be done

// via javascript in the browser, but doing it here in this sample to reduce the maze a bit

// Our callback will parse the fragment with javascript, as the token doesn't get sent

// to the server

// Note that with this flow there are no client credentials involved, so the callback

// provided here must match the one configured for the client in the authorization server

// (if one is configured)

// Again, note that the user will be prompted to authorize the client (and potentially login

// to the resource) before control is given to our callback

private ActionResult invokeImplicitFlow(IEnumerable<string> scopes)

{

var scopesQueryString = HttpUtility.UrlEncode(String.Join(" ",scopes));

var callbackQueryString = HttpUtility.UrlEncode(new Uri(Request.Url,

Url.Action("CacheTokenFromImplicitFlow", "TokenRequest")).AbsoluteUri);

return Redirect(String.Format(

"{0}?scope={1}&redirect\_uri={2}&response\_type=token&client\_id={3}",

AuthorizationServerConfig.AuthorizationEndpoint,

scopesQueryString,callbackQueryString,ClientConfig.ClientId));

}

// Process auth token provided by authorization server

// Has to be done by javascript on the browser side because

// it is passed as a fragment and doesn't come down to the server

public ActionResult CacheTokenFromImplicitFlow()

{

// View needs to know this clients auth token name as the js is actually putting the

// token in a cookie for access by the rest of the app

return View((object)ClientConfig.AccessTokenName);

}

#endregion

#region Client Credentials Flow

// Client Credentials flow obtains a token that is independent of user

// Client is verified using the client secret included when we configured the

// ClientConfig.AuthorizationServerClient

private ActionResult invokeClientCredentialsFlow(IEnumerable<string> scopes)

{

string msg = null;

bool gotToken = false;

try

{

// This seems to throw ProtocolException if client is unauthorized rather than

// just returning no token

var state = ClientConfig.AuthorizationServerClient.GetClientAccessToken(scopes);

gotToken = state.AccessToken != null;

if (gotToken)

{

var cookie = new HttpCookie(ClientConfig.AccessTokenName, state.AccessToken)

{ Expires = state.AccessTokenExpirationUtc.Value, Path="/" };

Response.Cookies.Add(cookie);

}

}

catch (ProtocolException ex)

{

msg = ", Message was: " + (ex.InnerException != null

? ex.InnerException.Message : ex.Message);

}

return RedirectToAction("Index", "Main",

new { msg = gotToken ? "Token Granted" : "No Access Token Was Granted" + msg });

}

#endregion

}

Information needed to access the authorization server is stored in some classes in the Config folder, including the client-specific id/secret, and the endpoint URLs of the authorization server. For this sample, this information all comes from the web.config for the project.

Project Client2API is the same code with a different resource encryption key, to demonstrate having multiple clients accessing resources via the same authorization server. For Client2 the seed data is set to disallow access to the scopes for resource2.

* 1. **ResourceLogin**

This is a simple MVC project containing a single controller that allows a user to login. There’s nothing special to see here. The configuration of the seed data for the resources mimics a single-sign-on scenario by having both of the resources use the login action on this controller as their authentication mechanism.

1. **Other Possible Extensions**

The solution presented here seems to serve its purpose of providing a single authorization server that can support multiple clients and multiple resources. There are a number of areas that could be extended further but have been left out for now for simplicity.

* Allow user to specify the duration for the authorization of a client, and thus the length of the access token, instead of the arbitrary value I’ve chosen.
* Allow user to authorize a client once instead of repeatedly.
* Support concept of refresh tokens. Currently once a token expires a new one must be retrieved by the client.
* Support consideration of the roles a user has for a given resource when deciding whether the client should be granted access to scopes on behalf of the user (e.g., if a user is restricted to read-only access, don’t allow clients to perform updates via the API on behalf of the user)

**Appendix A: OAuth2 Terminology**

But it will be helpful to clarify the terminology a bit before going further. For more details on the protocol, the OAuth2 spec can be found at <http://tools.ietf.org/html/rfc6749>.

**A.1 Roles**

Quoting from section 1.1 of the OAuth2 spec, the following roles are involved in the protocol:

* “resource owner”: “An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.” Think of this as the user that has data that he/she would like to expose to an external app.
* “resource server”: “The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.” Think of this as the Web API we want to expose.
* “client”: “An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).”
* “authorization server”: “The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.”

**A.2 Flows**

There are 4 flows that can be used by a client to obtain an access token to use in accessing a resource. Again, quoting from the spec, these flows include:

* “Authorization Code” [a.k.a. Code Flow]: “The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server (via its user-agent as defined in [RFC2616]), which in turn directs the resource owner back to the client with the authorization code. Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner and obtains authorization. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client. The authorization code provides a few important security benefits, such as the ability to authenticate the client, as well as the transmission of the access token directly to the client without passing it through the resource owner's user-agent and potentially exposing it to others, including the resource owner.”
* “Implicit”: “The implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In the implicit flow, instead of issuing the client an authorization code, the client is issued an access token directly (as the result of the resource owner authorization). The grant type is implicit, as no intermediate credentials (such as an authorization code) are issued (and later used to obtain an access token). When issuing an access token during the implicit grant flow, the authorization server does not authenticate the client. In some cases, the client identity can be verified via the redirection URI used to deliver the access token to the client. The access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent. Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application), since it reduces the number of round trips required to obtain an access token. However, this convenience should be weighed against the security implications of using implicit grant…”
* “Resource Owner Password Credentials”: “The resource owner password credentials (i.e., username and password) can be used directly as an authorization grant to obtain an access token. The credentials should only be used when there is a high degree of trust between the resource owner and the client (e.g., the client is part of the device operating system or a highly privileged application), and when other authorization grant types are not available (such as an authorization code). Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.”
* “Client Credentials”: “The client credentials (or other forms of client authentication) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used as an authorization grant typically when the client is acting on its own behalf (the client is also the resource owner) or is requesting access to protected resources based on an authorization previously arranged with the authorization server.” Note that in this flow, there is no user context at all.

**A.3 IMPORTANT NOTE**

Anyone who possesses an access token from the authorization server has access to the scopes and user information defined by that token. For this reason, it is important that these tokens do not fall into the wrong hands. Given this, all pieces of a solution involving OAuth2 **MUST** run under the context of HTTPS with properly configured SSL certificates. Not doing so is just asking for trouble. The framework we’re working with here (DotNetOpenAuth) will, by default, throw exceptions when not using HTTPS. For development purposes, you can relax this requirement (as I’ll show later), but **DO NOT** deploy code to production with these settings relaxed.