**Log Users in to Your Web Application with OpenID or OAuth**

By **Michiel Van Otegem**

[](https://www.codemag.com/subscribe/freearticle6b1?utm_source=articleviewer&utm_medium=banner&utm_campaign=Article-Viewer-Free-Subs)

Share

Syntax Highlight Theme:



**Michiel Van Otegem**

Michiel van Otegem is Senior Software Architect at Sogeti Netherlands. He started programming at age 10, and has nearly 20 years of professional experience as a developer and architect. As a pioneer in ASP and ASP.NET, Michiel still has a strong focus on the Microsoft platform, but he combines that with extensive knowledge of integration and security.  
  
  
Michiel is the author of several books in Dutch and English, starting with Teach Yourself XSLT in 21 Days (SAMS) back in 2002, and most recently contributing to Programming Microsoft’s Clouds: Windows Azure and Office 365 (Wrox).  
  
  
Michiel enjoys sharing his knowledge through books, articles, and speaking at conferences and user group events. He also loves to learn, especially life lessons from his two children.

Users already have many usernames and passwords for different popular online services, and with OpenID and OAuth, you can leverage those. Why burden users with yet another set of credentials for your site if they can use their Google or Facebook account, or any other OpenID or OAuth account? In this article, I will show you how to do this with ASP.NET 4.5, but more importantly help you understand what’s going on behind the scenes.

Ever since .NET 1.0, Forms Authentication has been the norm when it comes to authenticating users for your website. In .NET 2.0, Microsoft added the Membership API to make it even simpler. The Membership API contains everything you need to enable users to register for an account and login without a single line of code. It just works. There must be scores of websites out there using ASP.NET Membership with a database.

Although Membership security is definitely better than no security, there are at least two major drawbacks to having a user store in your own database. The first, as I already mentioned, is the number of accounts people have across all those sites on the Internet. Last time I tried to figure out how many accounts I have, I stopped counting after 150, and I’m sure I have at least a couple of hundred more. Because people have so many accounts, they are prone to reuse usernames and passwords, and that’s not good for security. Some less reputable site may try to use the harvested usernames and passwords on other sites.

The attack surface of an application increases as the number of functions increases.

The second drawback is the security of the user store itself. The attack surface of an application increases as the number of functions increases. In itself this is not the biggest problem in the world, but because the user store is part of the application, a hacker can also potentially gain access to numerous accounts through a weakness is some completely unrelated function.

**Centralizing Identity**

The obvious solution to the many passwords users have is centralizing the user store, so applications can enable users to use the same credentials across applications. One of the first attempts to do this was Microsoft Passport, but it didn’t go down well with the majority of users. A major reason Passport didn’t get a lot of traction was the fear that Microsoft was storing all data about the user, from simple name and email to credit card number. Imagine what would happen if that data fell into the wrong hands. With Microsoft not being synonymous with security, as well as privacy concerns, Passport was dead in the water. Microsoft more or less retracted and only used Passport for authentication with its own sites: Hotmail, MSN, MSDN, etc. It also rebranded Passport a couple of times, and now it is known as Live ID. The lesson learned: the trust we have in an organization determines how much data we are willing to entrust to it. Hence, a single centralized user store with key information about you has a low chance at success, especially if you don’t trust the organization behind it.

The trust we have in an organization determines how much data we are willing to entrust to it.

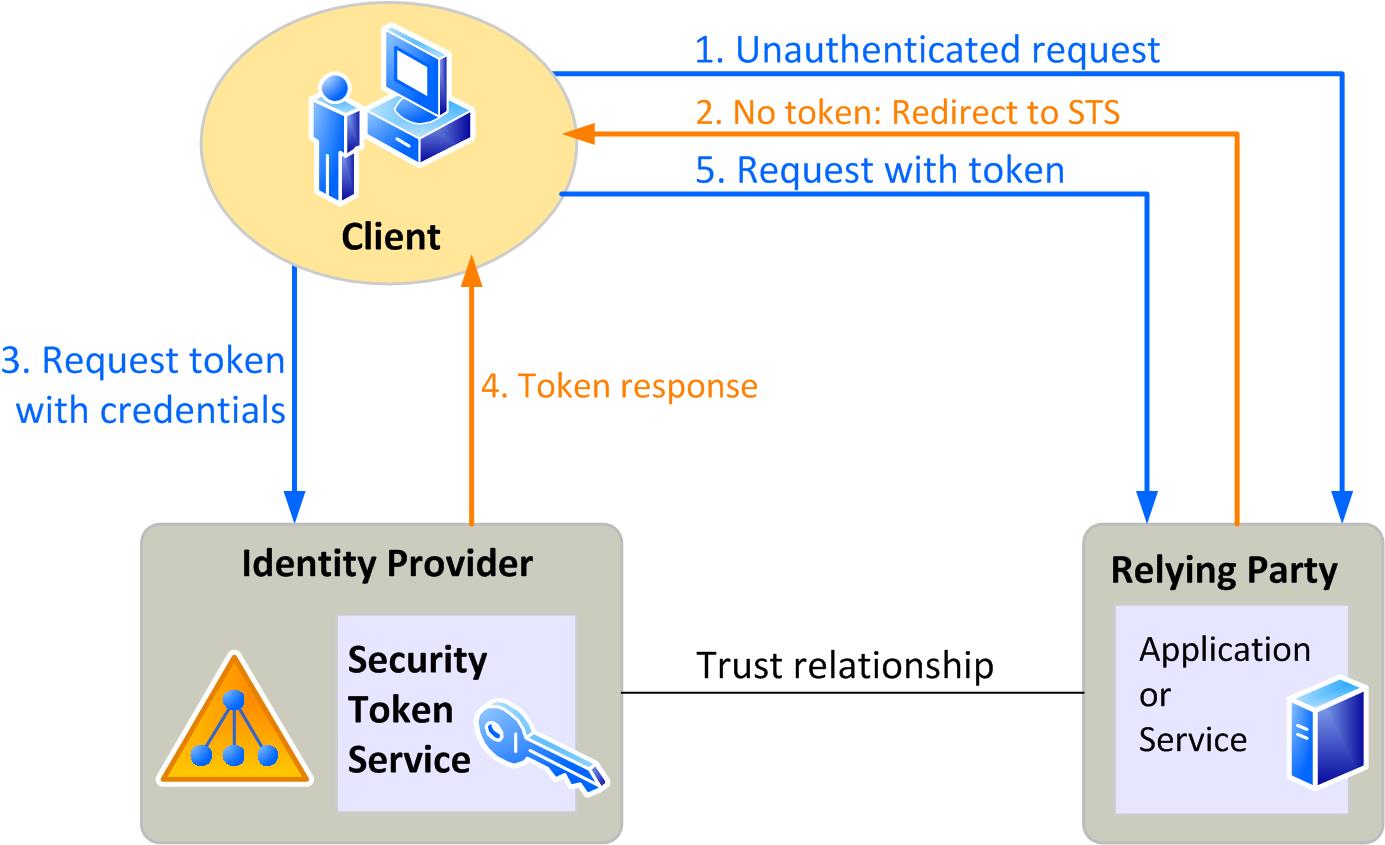
**Federating Identity**

Even though a single centralized system failed, the concepts behind it haven’t. Being able to reuse an identity you have with multiple applications remains a good idea. You may, however, want to have different identities to use with different applications. Also, as a user you should be in control of what data is stored about you, and what is shared with an application you authenticate with. In practice, this means that the most sensitive data, such as your credit card number (as yet) isn’t stored centrally, but rather with each application that requires it, which stores it locally. That is exactly what identity federation is all about. First, it means you can use an identity provider external to an application to authenticate, so you can reuse the credentials. Second, it means that data about you is not stored in a central location, but rather scattered over different locations, depending on what information is needed where. Some data is, in fact, stored centrally, like your email address, while other data is stored elsewhere. For a specific application, the external data and the local data is combined to make a single identity. So, how does it work?

**Identity Federation Basics**

Sharing identities with multiple sites is easy, right? Just have the user store expose a few web services to applications to check credentials. Although that idea looks good at first sight, it unfortunately doesn’t solve the problem of a malicious site harvesting credentials. The user can’t be sure the site is actually authenticating against the centralized service, and can’t know what the site does with the username and password entered.

A better solution is a concept introduced by the Kerberos protocol decades ago: have the user authenticate with the Authentication Server (in Kerberos lingo), and provide the user with a ticket (also Kerberos lingo) they can present to applications. **Figure 1**shows how this works using more generic terminology surrounding identity federation. You can compare the process in **Figure 1**with the way your driver’s license works. It is issued by the state (the Security Token Service or STS), which establishes your identity before giving it to you (the Identity Provider or IP). The police (the Relying Party or RP) will accept it to identify you if you made a traffic violation. Night clubs will also accept it, not necessarily to identify you, but to ensure you are not under age. To them, that is the only aspect of your identity they are interested in. This all works because both the police and the bouncer at the night club can check the authenticity of the driver’s license. It has certain characteristics that should be able to tell you whether it’s the real thing. The same is true for a token presented to an application. Applications need a means to check if the token is indeed from the STS it trusts, and that it hasn’t been tampered with. The solution to this is signing and/or encryption.

**Figure 1**: Identity Federation works through a series of redirects.

The example of the night club doesn’t work with Kerberos. It was only designed to authenticate. Applications get a unique identifier of the user, and that’s it. Data the application needs is up to the application to acquire from the user. This is exactly where identity federation differs. It is not just about authenticating the user. It’s about the information that an application gets about the user. The night club doesn’t need a unique identifier (well maybe for marketing purposes); it just wants to make sure you’re old enough to frequent the place. Other organizations accepting a driver’s license may need other information that’s on it.

**Competing Protocols**

These days there are several protocols that implement the same concept, but where Kerberos was limited to a single network, protocols such as OpenID, OAuth and WS-Federation work equally well over the Internet. In addition, where Kerberos wasn’t suitable to the night club example, these protocols are. These protocols can communicate attributes of the user to the application. These attributes can range from roles or groups the user belongs to, to email, age, allergies, you name it. The attributes communicated to an application can differ per application, so an application only gets to know what it needs to know. Even though the protocols are more or less the same, the terminology used for each of them might differ slightly. Once you understand the concepts, you will find you will understand the different terminology as well.

Microsoft originally focused its efforts on the WS-Federation protocol, which is at the heart of Windows Identity Foundation, available since .NET Framework 3.5, and its flagship Identity Federation product, Active Directory Federation Services 2.0. In Visual Studio 2012, integrating with that stack has become far easier than it was (but that is a topic for another time). Although WS-Federation remains important, especially in business-to-business scenarios, the release of Visual Studio 2012 shows that OpenID and OAuth are the preferred protocols for authentication to consumer-oriented web applications. If you create an application using the Internet Application template with WebForms or MVC, you can easily modify the login experience to allow login with the following providers:

* Facebook
* Google
* LinkedIn
* Twitter
* Windows Live ID
* Yahoo

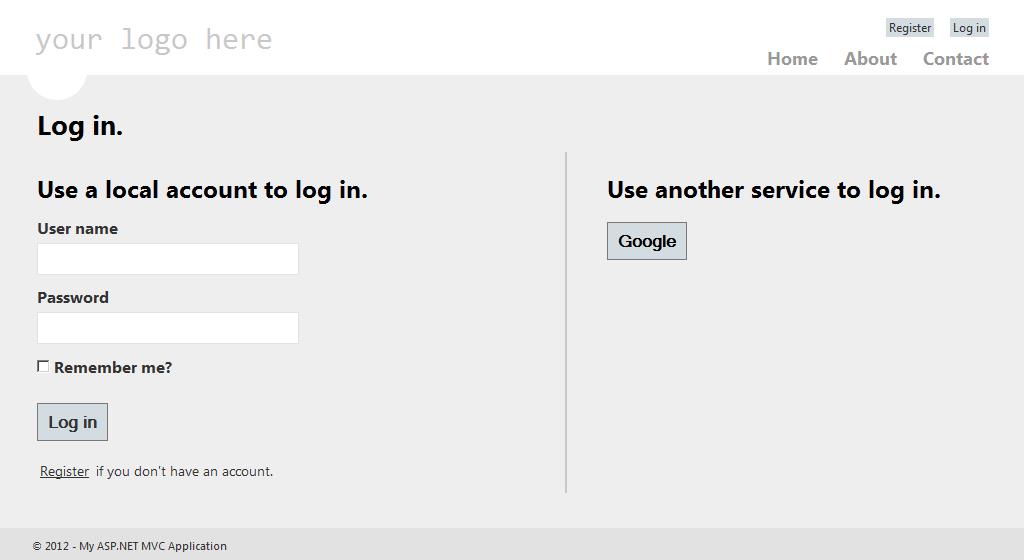
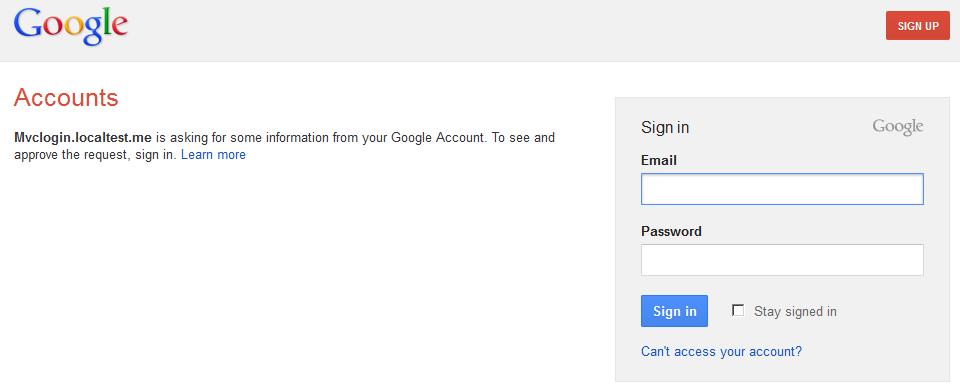
All you have to do is activate the provider(s) you want in the AuthConfig.cs file, which you can find in the App\_Start folder. For most of the providers mentioned, there’s already a commented line of code to activate it. You will find additional methods on the OAuthWebSecurity class for the other providers. The OAuthWebSecurity class is a wrapper around DotNetOpenAuth, an open source library for authentication with OpenID, OAuth, and ICard (see [http://www.dotnetopenauth.net](http://www.dotnetopenauth.net/)). It has been available through NuGet for quite a while, but comes with Visual Studio 2012 by default.

**Authentication via Google**

For Google you just have to uncomment the line of code in the RegisterAuth method in AuthConfig.cs that reads:

OAuthWebSecurity.RegisterGoogleClient();

Google is an OpenID provider, so you could also use the more generic RegisterOpenIDClient method, but then you would have to provide information about the provider. This information is preset for Google when using the Google method (and this goes for all predefined providers). The information is used to perform a discovery of all the required information to make the authentication process work. Discovery of the OpenID endpoint Google provides ensures that whatever your application gets back from Google is verifiable from Google. This is possible because OpenID uses Public Key Encryption to sign the token. When you uncomment the line of code above, the login screen of your application will look like **Figure 2**. Notice the Google button on the right side of the screen under *Use another service to login.*When the user clicks that, he’s sent to Google to login, which will show the page in **Figure 3**. Notice that under the Accounts heading, Google tells the user which application (URL) he’s signing into.

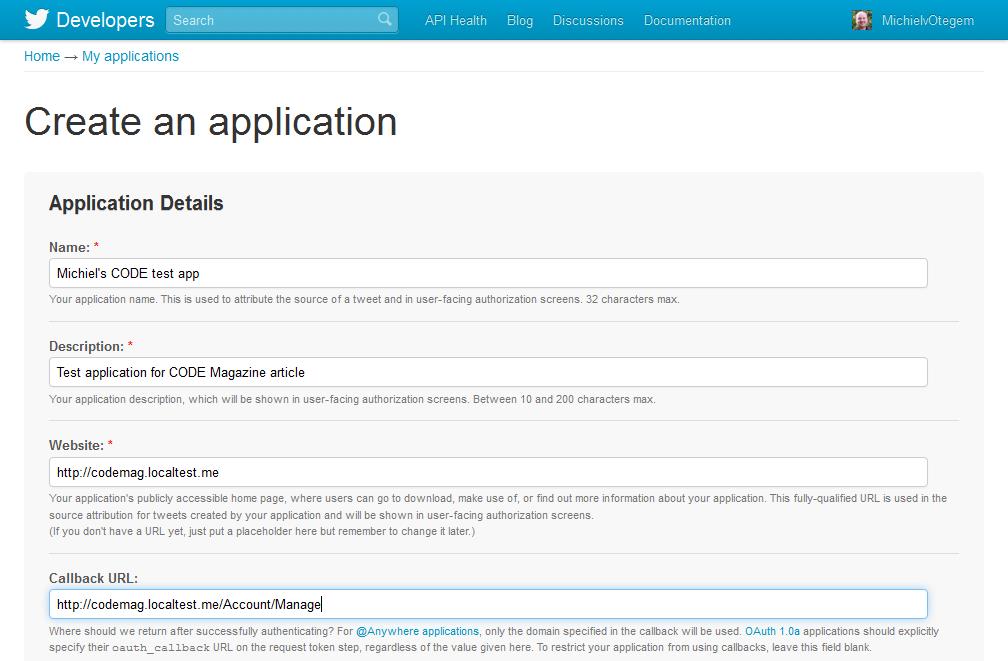
**Figure 2**: The login screen of the Internet Application template is automatically updated when you add additional login providers.**Figur e 3**: Google shows the user its familiar login screen, and information about where the user is signing into.

**Authentication via Twitter**

As you’ve seen, authentication via Google is easy enough. Just a single line of code and you’re done. With other providers, you need to do some more work. In order to work with Twitter, Facebook, and the other providers built-in to OAuthWebSecurity, you need to setup a trust relationship. To do this, you must register with the provider to get a unique identifier and an associated key. This key acts as a shared secret between your application and the provider, so the provider can encrypt the data it sends to you about the user. This is safer for the user, because personal data isn’t flying over the web unencrypted, and it is also safer for your web application, because you can be sure the data is coming from the selected provider. If it isn’t, the data can’t be properly decrypted. The other way around, the provider doesn’t need to worry that it’s authenticating the user for the wrong application, because it knows the data can only be decrypted by the application having the correct key. The shared key effectively establishes a trust relationship between your application and the identity provider.

Showing you the registration process and setting up your application to be able to authenticate with each of the built-in providers is beyond the scope of this article, but to show you how this works in general, I’ll walk you through authentication via Twitter. You can find a walkthrough for each of the providers on my blog, [http://michiel.vanotegem.nl](http://michiel.vanotegem.nl/).

To get a key from Twitter, you need to create a Twitter App. You can do this from the developer site at [https://dev.twitter.com](https://dev.twitter.com/). There you need to click *Create an app*and sign in with your Twitter account. You then need to provide some information about your application, as shown in **Figure 4**. First are name and description, which are shown to users that want to authenticate for your application. Then you need to provide the URL of the home page of your application. If your site is already on a public domain, this is where you would fill in that domain. More likely, at this point you’re working a local development environment. Unfortunately, the form will reject localhost or 127.0.0.1. You can solve this by creating an entry in your hosts file for some thought up server name, such as my.test.local, but there is an easier way. Several smart guys registered the localtest.me domain, and set up a DNS that always points to 127.0.0.1, regardless of the subdomain (except for [readme.localtest.me](http://readme.localtest.me/), where you can find more information). This means that for instance, [codemag.localtest.me](http://codemag.localtest.me/) is a perfectly valid domain according to the Twitter form, but it does in fact point to your local machine.

**Figure 4**: Twitter requires you to create an application so users can be informed about what they are signing into.

There’s one more gotcha here, and that’s that the form doesn’t allow you to specify a port number. This means you have to operate your development site on port 80. This will not work with IIS Express out-of-the-box, and at http://bit.ly/iisexpressport80 you can find how to get that working. Much easier is hooking your development site into IIS by either having the Default Web Site point to the folder where your site lives, or by adding another site with a host header corresponding to whatever localtest.me domain you entered in the Twitter form.

Finally, you can provide a callback URL that points to the page you want users to see after they’ve logged in through Twitter. If you don’t provide it, the user is redirected to the home page. Setting it, however, like in **Figure 4**, doesn’t help, as ASP.NET overrides this behavior. Once you’ve filled out all fields, and accepted the *Rules of the Road*, you can click *Create Twitter application*, and you are shown the management page for the App. At the bottom of the Details page is a button named *Create my access token*. Contrary to what you might think (and other resources will tell you to do), you don’t need to do this. This only comes into play if you want your application to be able to read tweets from the user’s timeline (see sidebar for more information). From the Details page grab the Consumer key and Consumer secret, and use these to register Twitter as authentication provider in AuthConfig.cs, as follows:

public static void RegisterAuth()

{

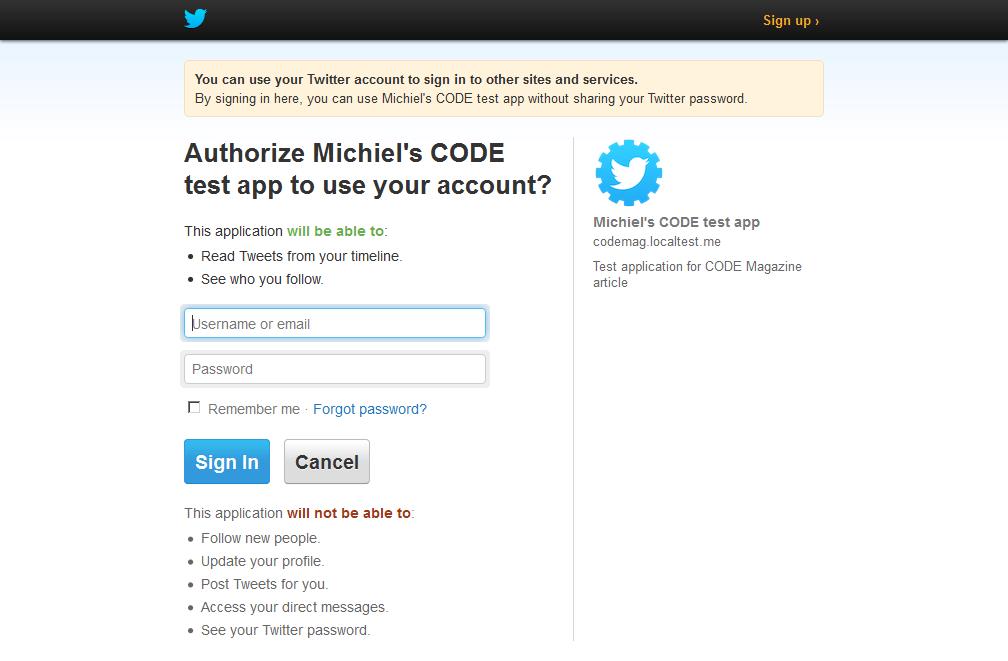
OAuthWebSecurity.RegisterTwitterClient(

consumerKey: "fIPRhs89Nshhaz998YQ"

consumerSecret: "VH2ajS89...");

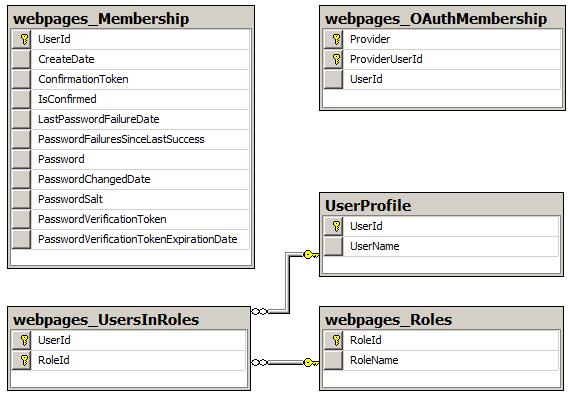
}

Now if you browse to the site and go to the Login page, Twitter will be available as alternative login service. Like Google, Twitter will show you a login page that tells the user that he’s logging into your application. The page also details what the user is allowing your application to do, and what it can’t, as shown in **Figure 5**. Note that unlike Google, there is no additional screen after logging in where the user has to explicitly authorize the application. As with Google, and all the other providers, the user has to create an account in your application corresponding to the Twitter account.

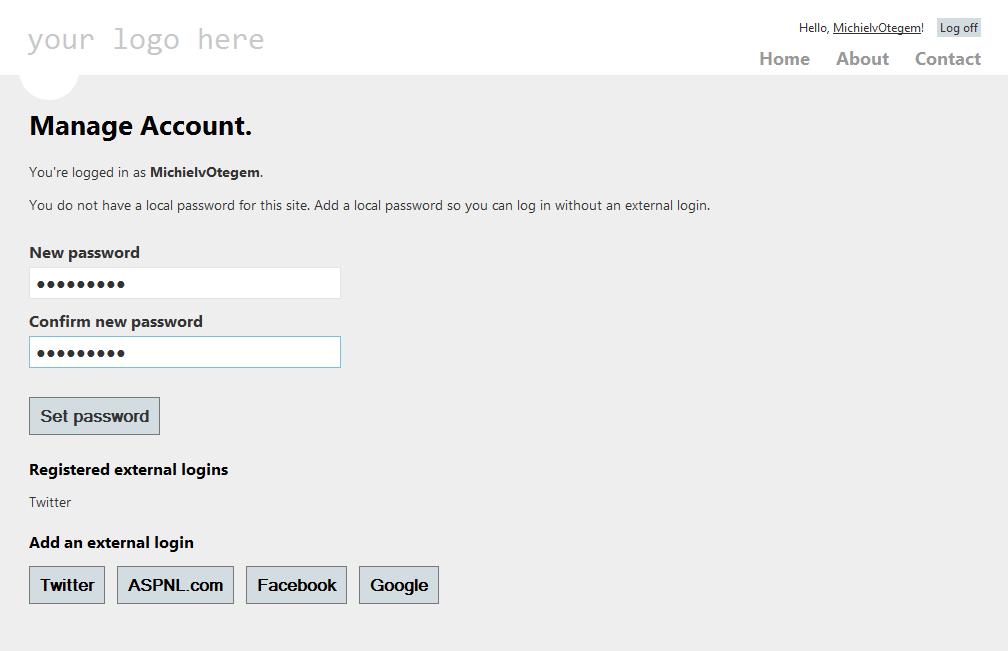
**Figure 5**: Twitter displays information about the application, and what the application can do on behalf of the user.

**How does it Work?**

Behind the scenes the new ASP.NET Simple Membership is used, which was introduced for ASP.NET WebPages. It doesn’t replace ASP.NET Membership that was introduced in ASP.NET 2.0, but solves some common issues associated with ASP.NET Membership. This makes it the preferred API for membership functionality in ASP.NET moving forward. One major improvement Simple Membership brings to the table is the option to work with your own database schema when it comes to storing data about the user, and as such, it works much better with the Entity Framework. Out of the box, the database schema looks like **Figure 6**. At the heart of the schema in **Figure 6**is the UserProfile table with the UserId. That UserId is the internal identifier of the user, to which other tables can be linked, like the webpages\_UserInRoles table. The tables webpages\_OAuthMembership and webpages\_Membership are auxiliary tables that refer to the UserProfile table, even though there is no foreign key relationship in the database by default. These each serve a different purpose. The former is used for authentication through an OAuth or OpenID provider, whereas the latter is for “local” accounts; accounts that have username and password stored in the database. That one is used if you don’t configure any other providers.

**Figure 6**: Simple Membership uses a database schema, which is extensible.

A subtle characteristic of this model is the fact that you can tie multiple login methods to the same UserId. The Internet Application template actually supports this out of the box if you configure additional providers. Once the user is signed in, he can manage his account and the logins he can use to access it, as you can see in **Figure 7**.

**Figure 7**: Users can create a single account and add multiple external logins to it.

**Simple Membership Weakness**

Earlier I said that having a user store inside your application isn’t a good idea, because hackers can gain access to it more easily through other functionality. However, the Simple Membership system still stores information about the user, including Role authorizations of the users locally. How big a deal that is depends on what you do with user information. If there is no sensitive data about the user in your application, the problem is minor. All the hacker gets is some unique identifier with some properties. By exchanging in his own unique identifier, he can gain access to the account, but gain no information or additional privileges. If you do authorize and/or store sensitive information, you need to take extra care that your database remains secure. Encrypting the contents of the database, or at least signing it so it can’t be tampered with is a good idea in that case. There are more elaborate options, like working with an intermediary authentication provider, but that is beyond the scope of this article.

**Conclusion**

In this article, you’ve learned how easy it is to enable users to login into your application with Google and Twitter, and other popular providers. Facebook alone represents roughly a billion users, so think of the reach this gives you. It makes the barrier for people to login to your site much lower. You’ve also learned what goes on behind the scenes, and you got a 101 on ASP.NET Simple Membership that’s now part of ASP.NET. I sincerely hope you use this knowledge to make the web a safer and more fluid experience for your users.

**OpenID vs OAuth**

OAuth can be used to authenticate a user for an application, as is demonstrated in this article. In that sense, it is the same as OpenID (and WS-Federation). However, it is actually designed for a different purpose: provide other applications access to data and operations of the application authenticating the user. For instance, allowing your Twitter client to read and post messages on your behalf. This means that if you enable your application to authenticate users through Twitter or Facebook, users are actually giving your application access to their information thereby authenticating. In essence, OAuth does more than OpenID, and you need to be aware of that fact. It probably means you need to tell your users what you will do with the powers they grant you.