# Chapter 6. Security

# Using OAuth Providers with MVC 4

This tutorial shows you how to build an ASP.NET MVC 4 web application that enables users to log in with credentials from an external provider, such as Facebook, Twitter, Microsoft, or Google, and then integrate some of the functionality from those providers into your web application. For simplicity, this tutorial focuses on working with credentials from Facebook.

Enabling these credentials in your web sites provides a significant advantage because millions of users already have accounts with these external providers. These users may be more inclined to sign up for your site if they do not have to create and remember a new set of credentials. Also, after a user has logged in through one of these providers, you can incorporate social operations from the provider.

## What you’ll build

There are two main goals in this tutorial:

1. Enable a user to log in with credentials from an OAuth provider.
2. Retrieve account information from the provider and integrate that information with the account registration for your site.

Although the examples in this tutorial focus on using Facebook as the authentication provider, you can modify the code to use any of the providers. The steps to implement any provider are very similar to the steps you will see in this tutorial. You will only notice significant differences when making direct calls to the provider’s API set.

## Prerequisites

* [Microsoft Visual Studio 2012](http://www.microsoft.com/visualstudio/eng/downloads#vs) or [Microsoft Visual Studio Express 2012 for Web](http://www.microsoft.com/visualstudio/eng/downloads#d-2012-express)

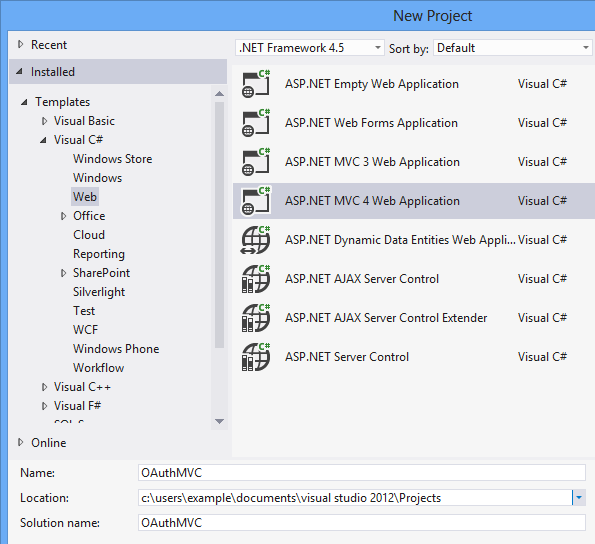
Or

* Microsoft Visual Studio 2010 SP1 or [Visual Web Developer Express 2010 SP1](http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express)
* [ASP.NET MVC 4](http://go.microsoft.com/fwlink/?LinkId=243392)

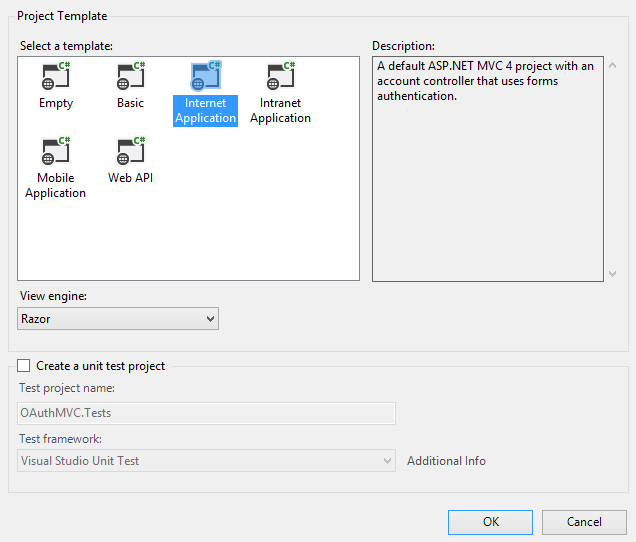
Furthermore, this topic assumes you have basic knowledge about ASP.NET MVC and Visual Studio. If you need an introduction to ASP.NET MVC 4, see [Intro to ASP.NET MVC 4](http://www.asp.net/mvc/tutorials/mvc-4/getting-started-with-aspnet-mvc4/intro-to-aspnet-mvc-4).

## Create the project

In Visual Studio, create a new ASP.NET MVC 4 Web Application, and name it "OAuthMVC". You can target either .NET Framework 4.5 or 4.

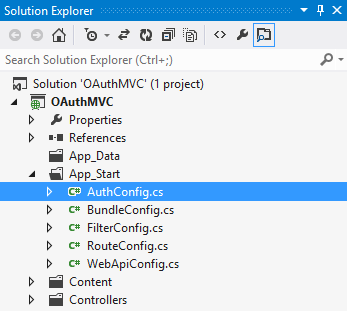


In the New ASP.NET MVC 4 Project window, select **Internet Application** and leave **Razor** as the view engine.



## Enable a provider

When you create an MVC 4 web application with the Internet Application template, the project is created with a file named AuthConfig.cs in the App\_Start folder.



The AuthConfig file contains code to register clients for external authentication providers. By default, this code is commented out, so none of the external providers are enabled.

public static class AuthConfig

{

public static void RegisterAuth()

{

// To let users of this site log in using their accounts from other sites such as Microsoft, Facebook, and Twitter,

// you must update this site. For more information visit http://go.microsoft.com/fwlink/?LinkID=252166

//OAuthWebSecurity.RegisterMicrosoftClient(

// clientId: "",

// clientSecret: "");

//OAuthWebSecurity.RegisterTwitterClient(

// consumerKey: "",

// consumerSecret: "");

//OAuthWebSecurity.RegisterFacebookClient(

// appId: "",

// appSecret: "");

//OAuthWebSecurity.RegisterGoogleClient();

}

}

You must uncomment this code to use the external authentication client. You uncomment only the providers you want to include in your site. For this tutorial, you will only enable the Facebook credentials.

public static class AuthConfig

{

public static void RegisterAuth()

{

//OAuthWebSecurity.RegisterMicrosoftClient(

// clientId: "",

// clientSecret: "");

//OAuthWebSecurity.RegisterTwitterClient(

// consumerKey: "",

// consumerSecret: "");

OAuthWebSecurity.RegisterFacebookClient(

appId: "",

appSecret: "");

//OAuthWebSecurity.RegisterGoogleClient();

}

}

Notice in the above example that the method includes empty strings for the registration parameters. If you try to run the application now, the application throws an argument exception because empty strings are not allowed for the parameters. To provide valid values, you must register your web site with the external providers, as shown in the next section.

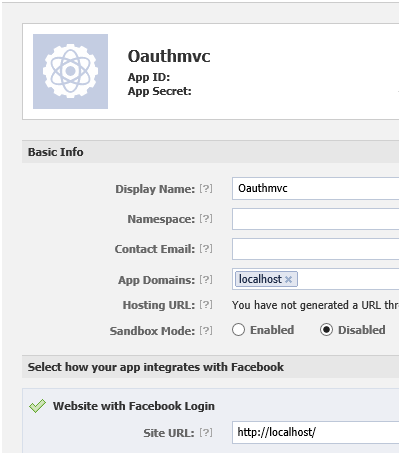
## Registering with an external provider

To authenticate users with credentials from an external provider, you must register your web site with the provider. When you register your site, you will receive the parameters (such as key or id, and secret) to include when registering the client. You must have an account with the providers you wish to use.

This tutorial does not show all of the steps you must perform to register with these providers. The steps are typically not difficult. To successfully register your site, follow the instructions provided on those sites. To get started with registering your site, see the developer site for:

* [Facebook](http://developers.facebook.com/)
* [Google](http://developers.google.com/)
* [Microsoft](http://manage.dev.live.com/)
* [Twitter](http://dev.twitter.com/)

When registering your site with Facebook, you can provide "localhost" for the site domain and "http://localhost/" for the URL, as shown in the image below. Using localhost works with most providers, but currently does not work with the Microsoft provider. For the Microsoft provider, you must include a valid web site URL.



In the previous image, the values for the app id, app secret, and contact email have been removed. When you actually register your site, those values will be present. You will want to note the values for app id and app secret because you will add them to your application.

## Creating test users

If you do not mind using an existing Facebook account to test your site, you can skip this section.

You can easily create test users for your application within the Facebook app management page. You can use these test accounts to log in to your site. You create test users by clicking the **Roles** link in the left navigation pane and the clicking the **Create** link.



The Facebook site automatically creates the number of test accounts that you request.

## Adding application id and secret from the provider

Now that you have received the id and secret from Facebook, go back to the AuthConfig file and add them as the parameter values. The values shown below are not real values.

public static class AuthConfig

{

public static void RegisterAuth()

{

//OAuthWebSecurity.RegisterMicrosoftClient(

// clientId: "",

// clientSecret: "");

//OAuthWebSecurity.RegisterTwitterClient(

// consumerKey: "",

// consumerSecret: "");

OAuthWebSecurity.RegisterFacebookClient(

appId: "111111111111111",

appSecret: "a1a1aa111111111a111a111aaa111111");

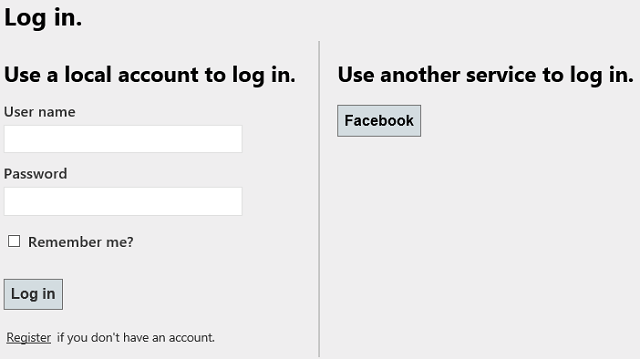
//OAuthWebSecurity.RegisterGoogleClient();

}

}

## Log in with external credentials

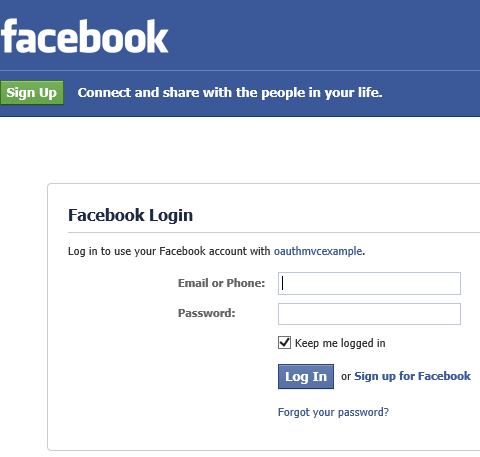
That is all you have to do to enable external credentials in your site. Run your application and click the login link in the upper right corner. The template automatically recognizes that you have registered Facebook as a provider and includes a button for the provider. If you register multiple providers, a button for each one is automatically included.



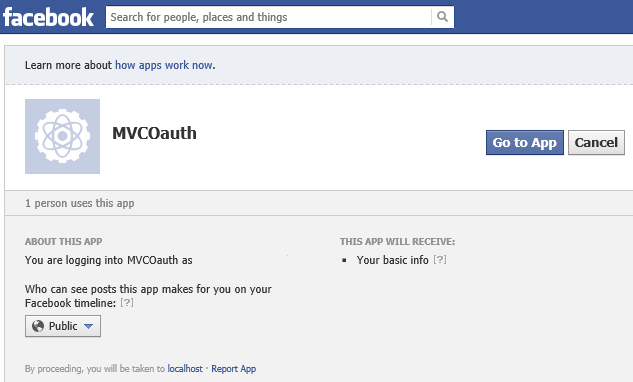
This tutorial does not cover how to customize the log in buttons for the external providers. For that information, see [Customizing the login UI when using OAuth/OpenID](http://blogs.msdn.com/b/pranav_rastogi/archive/2012/08/24/customizing-the-login-ui-when-using-oauth-openid.aspx).

Click on the Facebook button to log in with Facebook credentials. When you select one of the external providers, you are redirected to that site and prompted by that service to log in.

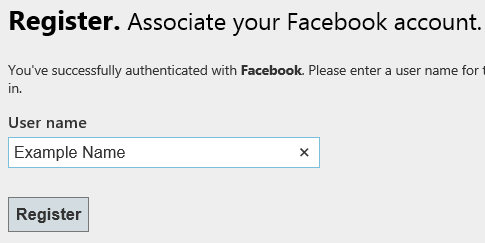
The following image shows the login screen for Facebook. It notes that you are using your Facebook account to log in to a site named oauthmvcexample.



After logging in with Facebook credentials, a page informs the user that the site will have access to basic information.

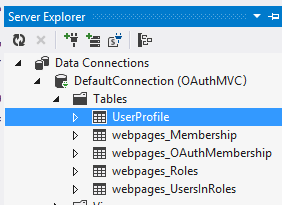


After selecting **Go to App**, the user must register for the site. The following image shows the registration page after a user has logged in with Facebook credentials. The user name is typically pre-filled with a name from the provider.

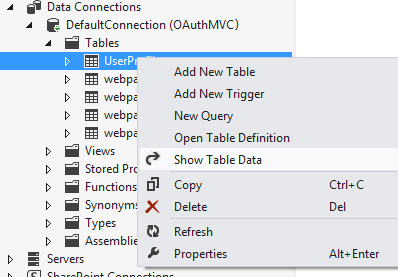


Click **Register** to complete registration. Close the browser.

You can see that the new account has been added to your database. In Server Explorer, open the **DefaultConnection** database and open the **Tables** folder.



Right-click the **UserProfile** table and select **Show Table Data**.



You will see the new account you added. Look at the data in **webpage\_OAuthMembership** table. You will see more data related to the external provider for the account you just added.

If you only want to enable external authentication, you are done. However, you can further integrate information from the provider into the new user registration process, as shown in the following sections.

## Create models for additional user information

As you noticed in the previous sections, you do not need to retrieve any additional information for the built-in account registration to work. However, most external providers pass back additional information about the user. The following sections show how to retain that information and save it to a database. Specifically, you will retain values for the user’s full name, the URI of the user’s personal web page, and a value that indicates whether Facebook has verified the account.

You will use [Code First Migrations](http://msdn.microsoft.com/en-us/data/jj591621) to add a table for storing additional user information. You are adding the table to an existing database, so you will first need to create a snapshot of the current database. By creating a snapshot of the current database, you can later create a migration that contains only the new table. To create a snapshot of the current database:

1. Open the **Package Manager Console**
2. Run the command **enable-migrations**
3. Run the command **add-migration initial –IgnoreChanges**
4. Run the command **update-database**

Now, you will add the new properties. In the Models folder, open the AccountModels.cs file, and find the RegisterExternalLoginModel class. The RegisterExternalLoginModel class holds values that come back from the authentication provider. Add properties named FullName and Link, as highlighted below.

public class RegisterExternalLoginModel

{

[Required]

[Display(Name = "User name")]

public string UserName { get; set; }

public string ExternalLoginData { get; set; }

[Display(Name = "Full name")]

public string FullName { get; set; }

[Display(Name = "Personal page link")]

public string Link { get; set; }

}

Also in AccountModels.cs, add a new class called ExtraUserInformation. This class represents the new table which will be created in the database.

[Table("ExtraUserInformation")]

public class ExternalUserInformation

{

public int Id { get; set; }

public int UserId { get; set; }

public string FullName { get; set; }

public string Link { get; set; }

public bool? Verified { get; set; }

}

In the UsersContext class, add the highlighted code below to create a DbSet property for the new class.

public class UsersContext : DbContext

{

public UsersContext()

: base("DefaultConnection")

{

}

public DbSet<UserProfile> UserProfiles { get; set; }

public DbSet<ExternalUserInformation> ExternalUsers { get; set; }

}

You are now ready to create the new table. Open the Package Manager Console again and this time:

1. Run the command **add-migration AddExtraUserInformation**
2. Run the command **update-database**

The new table now exists in the database.

## Retrieve the additional data

There are two ways to retrieve additional user data. The first way is to retain user data that is passed back, by default, during the authentication request. The second way is to specifically call the provider API and request more information. Values for FullName and Link are automatically passed back by Facebook. A value that indicates whether Facebook has verified the account is retrieved through a call to the Facebook API. First, you will populate values for FullName and Link, and then later, you will get the verified value.

To retrieve the additional user data, open the **AccountController.cs** file in the **Controllers** folder.

This file contains the logic for logging, registering, and managing accounts. In particular, notice the methods called **ExternalLoginCallback** and **ExternalLoginConfirmation**. Within these methods, you can add code to customize external login operations for your application. The first line of the **ExternalLoginCallback** method contains:

AuthenticationResult result = OAuthWebSecurity.VerifyAuthentication(

Url.Action("ExternalLoginCallback", new { ReturnUrl = returnUrl }));

Additional user data is passed back in the **ExtraData** property of the **AuthenticationResult** object that is returned from the **VerifyAuthentication** method. The Facebook client contains the following values in the **ExtraData** property:

* id
* name
* link
* gender
* accesstoken

Other providers will have similar but slightly different data in the ExtraData property.

If the user is new to your site, you will retrieve some the additional data and pass that data to the confirmation view. The last block of code in the method is run only if the user is new to your site. Replace the following line:

return View("ExternalLoginConfirmation", new RegisterExternalLoginModel

{

UserName = result.UserName,

ExternalLoginData = loginData

});

With this line:

return View("ExternalLoginConfirmation", new RegisterExternalLoginModel

{

UserName = result.UserName,

ExternalLoginData = loginData,

FullName = result.ExtraData["name"],

Link = result.ExtraData["link"]

});

This change merely includes values for the FullName and Link properties.

In the **ExternalLoginConfirmation** method, modify the code as highlighted below to save the additional user information.

if (user == null)

{

// Insert name into the profile table

UserProfile newUser = db.UserProfiles.Add(new UserProfile { UserName = model.UserName });

db.SaveChanges();

db.ExternalUsers.Add(new ExternalUserInformation

{

UserId = newUser.UserId,

FullName = model.FullName,

Link = model.Link

});

db.SaveChanges();

OAuthWebSecurity.CreateOrUpdateAccount(provider, providerUserId, model.UserName);

OAuthWebSecurity.Login(provider, providerUserId, createPersistentCookie: false);

return RedirectToLocal(returnUrl);

}

else

{

ModelState.AddModelError("UserName", "User name already exists. Please enter a different user name.");

}

## Adjusting the view

The additional user data that you retrieve from the provider will be displayed in the registration page.

In the **Views**/**Account** folder, open **ExternalLoginConfirmation.cshtml**. Below the existing field for user name, add fields for FullName, Link, and PictureLink.

<li>

@Html.LabelFor(m => m.FullName)

@Html.TextBoxFor(m => m.FullName)

</li>

<li>

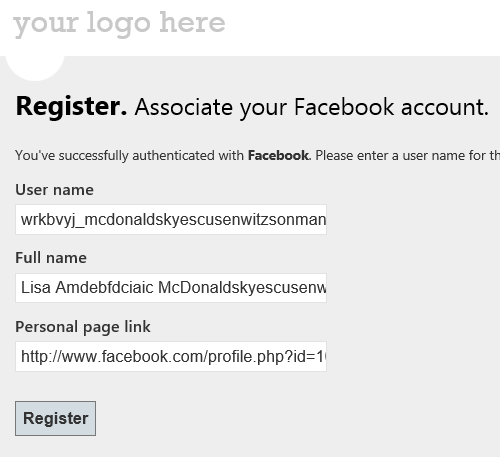
@Html.LabelFor(m => m.Link)

@Html.TextBoxFor(m => m.Link)

</li>

You are now almost ready to run the application and register a new user with the additional information saved. You must have an account that is not already registered with the site. You can either use a different test account, or delete the rows in the **UserProfile** and **webpages\_OAuthMembership** tables for the account you wish to reuse. By deleting those rows, you will ensure that the account is registered again.

Run the application and register the new user. Notice that this time the confirmation page contains more values.

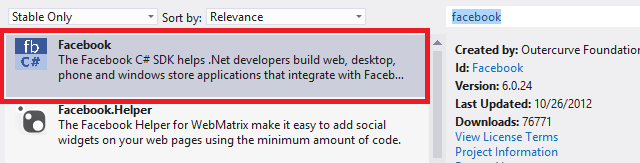


After completing registration, close the browser. Look in the database to notice the new values in the **ExtraUserInformation** table.

## Install NuGet package for Facebook API

Facebook provides an [API](http://developers.facebook.com/docs/reference/apis/) that you can call to perform operations. You can call the Facebook API either by directing sending HTTP requests, or by using installing a NuGet package that facilitates sending those requests. Using a NuGet package is shown in this tutorial, but installing NuGet package is not essential. This tutorial shows how to use the Facebook C# SDK package. There are other NuGet packages that assist with calling the Facebook API.

From the **Manage NuGet Packages** windows, select the Facebook C# SDK package.



You will use the Facebook C# SDK to call an operation that requires the access token for the user. The next section shows how to get the access token.

## Retrieve access token

Most external providers pass back an access token after the user’s credentials are verified. This access token is very important because it enables you to call operations that are only available to authenticated users. Therefore, retrieving and storing the access token is essential when you want to provide more functionality.

Depending on the external provider, the access token may be valid for only a limited amount of time. To ensure that you have a valid access token, you will retrieve it each time the user logs in, and store it as a session value rather than save it to a database.

In the **ExternalLoginCallback** method, the access token is also passed back in the **ExtraData** property of the **AuthenticationResult** object. Add the highlighted code to **ExternalLoginCallback** to save the access token in the **Session** object. This code is run every time the user logs in with a Facebook account.

[AllowAnonymous]

public ActionResult ExternalLoginCallback(string returnUrl)

{

AuthenticationResult result = OAuthWebSecurity.VerifyAuthentication(

Url.Action("ExternalLoginCallback", new { ReturnUrl = returnUrl }));

if (!result.IsSuccessful)

{

return RedirectToAction("ExternalLoginFailure");

}

if (result.ExtraData.Keys.Contains("accesstoken"))

{

Session["facebooktoken"] = result.ExtraData["accesstoken"];

}

if (OAuthWebSecurity.Login(

result.Provider,

result.ProviderUserId,

createPersistentCookie: false))

{

return RedirectToLocal(returnUrl);

}

if (User.Identity.IsAuthenticated)

{

// If the current user is logged in add the new account

OAuthWebSecurity.CreateOrUpdateAccount(

result.Provider,

result.ProviderUserId,

User.Identity.Name);

return RedirectToLocal(returnUrl);

}

else

{

// User is new, ask for their desired membership name

string loginData = OAuthWebSecurity.SerializeProviderUserId(

result.Provider,

result.ProviderUserId);

ViewBag.ProviderDisplayName =

OAuthWebSecurity.GetOAuthClientData(result.Provider).DisplayName;

ViewBag.ReturnUrl = returnUrl;

return View("ExternalLoginConfirmation", new RegisterExternalLoginModel

{

UserName = result.UserName,

ExternalLoginData = loginData,

FullName = result.ExtraData["name"],

Link = result.ExtraData["link"]

});

}

}

Although this example retrieves an access token from Facebook, you can retrieve the access token from any external provider through the same key named "accesstoken".

To prevent the token from persisting after the user has logged off, you can add the following highlighted code to the **LogOff** method in the AccountController.

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult LogOff()

{

WebSecurity.Logout();

Session.Remove("facebooktoken");

return RedirectToAction("Index", "Home");

}

## Retrieve user information that requires the access token

Now that you have stored the access token and installed the Facebook C# SDK package, you can use them together to request additional user information from Facebook. In the **ExternalLoginConfirmation** method, create an instance of the **FacebookClient** class by passing the value of the access token. Request the value of the **verified** property for the current, authenticated user. The **verified** property indicates whether Facebook has validated the account through some other means, such as sending a message to a cell phone. Save this value in the database.

if (user == null)

{

// Insert name into the profile table

UserProfile newUser = db.UserProfiles.Add(new UserProfile { UserName = model.UserName });

db.SaveChanges();

bool facebookVerified;

var client = new Facebook.FacebookClient(Session["facebooktoken"].ToString());

dynamic response = client.Get("me", new { fields = "verified" });

if (response.ContainsKey("verified"))

{

facebookVerified = response["verified"];

}

else

{

facebookVerified = false;

}

db.ExternalUsers.Add(new ExternalUserInformation

{

UserId = newUser.UserId,

FullName = model.FullName,

Link = model.Link,

Verified = facebookVerified

});

db.SaveChanges();

OAuthWebSecurity.CreateOrUpdateAccount(provider, providerUserId, model.UserName);

OAuthWebSecurity.Login(provider, providerUserId, createPersistentCookie: false);

return RedirectToLocal(returnUrl);

}

You will again need to either delete the records in the database for the user, or use a different Facebook account.

Run the application, and register the new user. Look at the **ExtraUserInformation** table to see the value for the Verified property.

## Conclusion

In this tutorial, you created a site that is integrated with Facebook for user authentication and registration data. You learned about the default behavior that is set up for MVC 4 web application, and how to customize that default behavior.

# Preventing Open Redirection Attacks (C#)

This tutorial explains how you can prevent open redirection attacks in your ASP.NET MVC applications. This tutorial discusses the changes that have been made in the AccountController in ASP.NET MVC 3 and demonstrates how you can apply these changes in your existing ASP.NET MVC 1.0 and 2 applications.

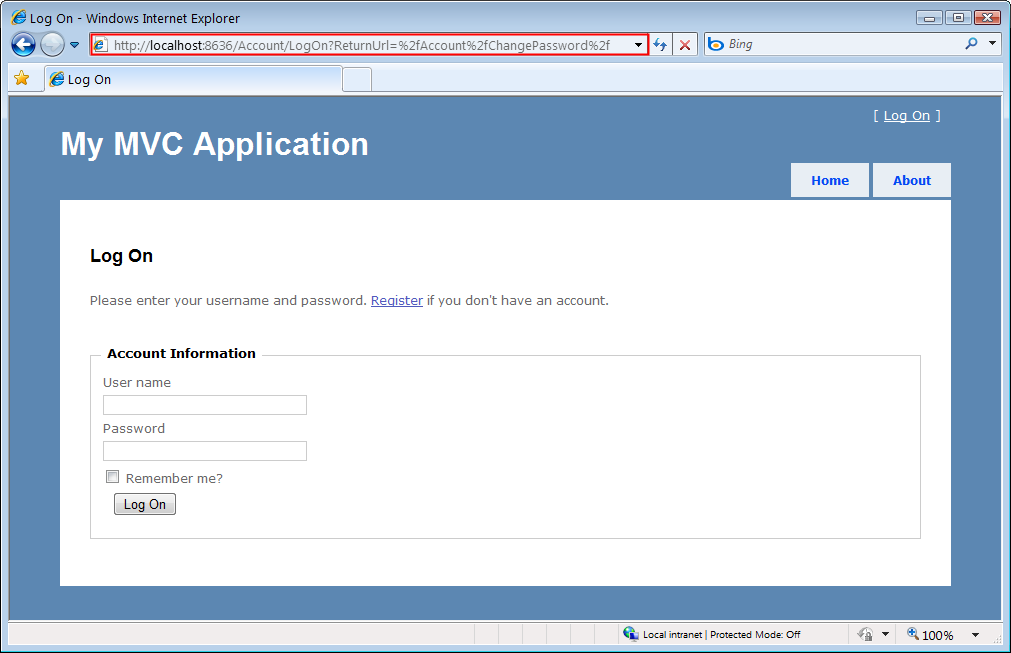
## What is an Open Redirection Attack?

Any web application that redirects to a URL that is specified via the request such as the querystring or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack.

Whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn’t been tampered with. The login used in the default AccountController for both ASP.NET MVC 1.0 and ASP.NET MVC 2 is vulnerable to open redirection attacks. Fortunately, it is easy to update your existing applications to use the corrections from the ASP.NET MVC 3 Preview.

To understand the vulnerability, let’s look at how the login redirection works in a default ASP.NET MVC 2 Web Application project. In this application, attempting to visit a controller action that has the [Authorize] attribute will redirect unauthorized users to the /Account/LogOn view. This redirect to /Account/LogOn will include a returnUrl querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in.

In the screenshot below, we can see that an attempt to access the /Account/ChangePassword view when not logged in results in a redirect to /Account/LogOn?ReturnUrl=%2fAccount%2fChangePassword%2f.

[](http://i2.asp.net/asp.net/images/whitepapers/security/Login%20Screen.png)

**Figure 01**: Login page with an open redirection

Since the ReturnUrl querystring parameter is not validated, an attacker can modify it to inject any URL address into the parameter to conduct an open redirection attack. To demonstrate this, we can modify the ReturnUrl parameter to [http://bing.com](http://bing.com/), so the resulting login URL will be /Account/LogOn?ReturnUrl=http://www.bing.com/. Upon successfully logging in to the site, we are redirected to [http://bing.com](http://bing.com/). Since this redirection is not validated, it could instead point to a malicious site that attempts to trick the user.

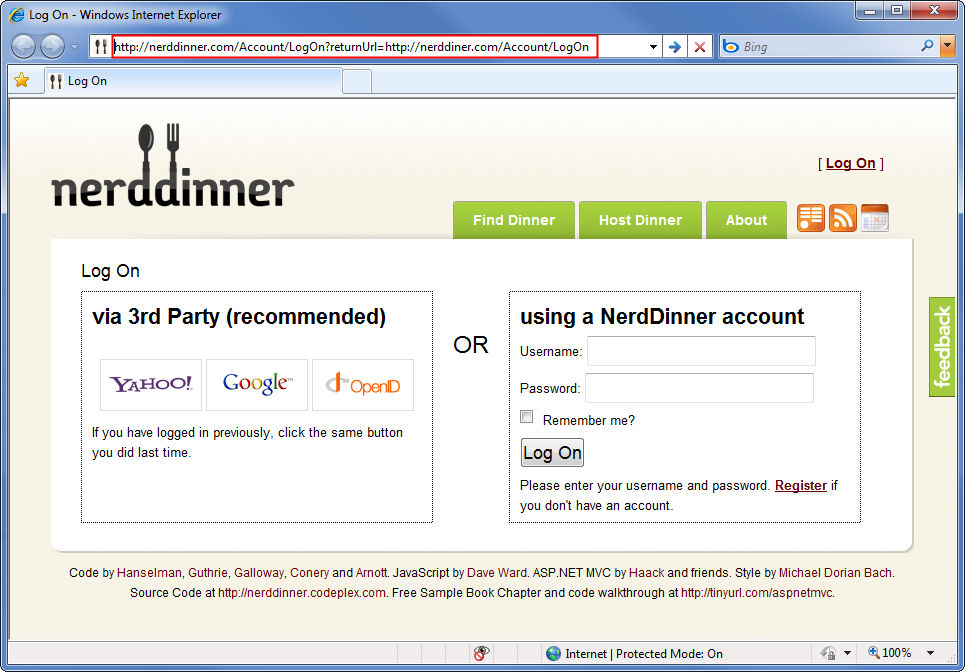
## A more complex Open Redirection Attack

Open redirection attacks are especially dangerous because an attacker knows that we’re trying to log into a specific website, which makes us vulnerable to a [phishing attack](http://www.microsoft.com/protect/fraud/phishing/symptoms.aspx). For example, an attacker could send malicious e-mails to website users in an attempt to capture their passwords. Let’s look at how this would work on the NerdDinner site. (Note that the live NerdDinner site has been updated to protect against open redirection attacks.)

First, an attacker sends us a link to the login page on NerdDinner that includes a redirect to their forged page:

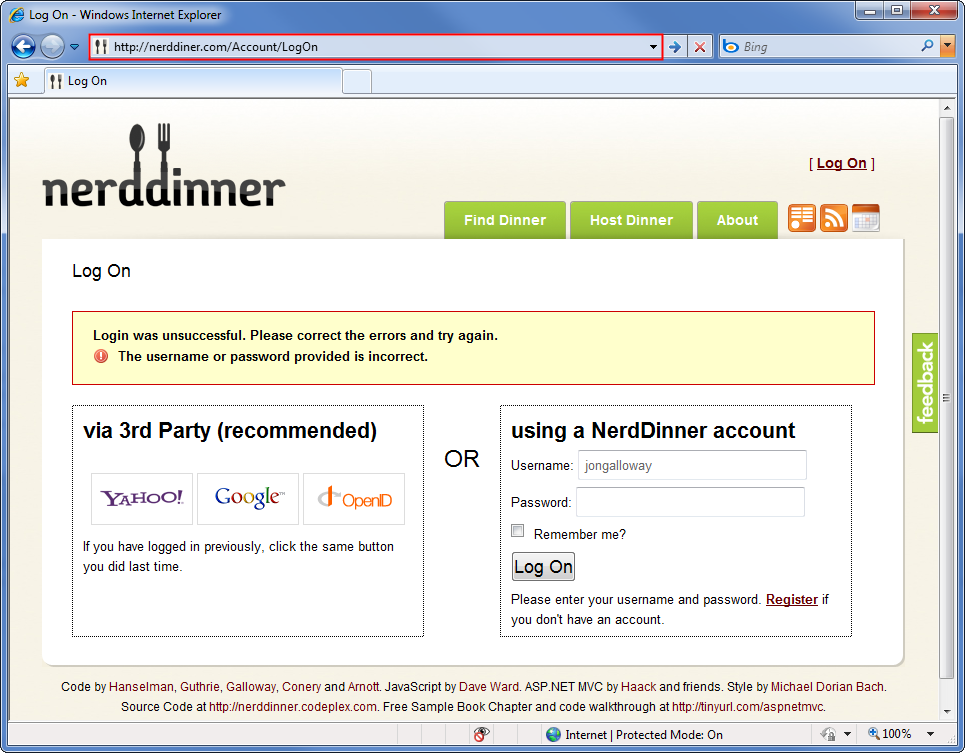
<http://nerddinner.com/Account/LogOn?returnUrl=http://nerddiner.com/Account/LogOn>

Note that the return URL points to nerddiner.com, which is missing an “n” from the word dinner. In this example, this is a domain that the attacker controls. When we access the above link, we’re taken to the legitimate NerdDinner.com login page.

[](http://i2.asp.net/asp.net/images/whitepapers/security/NerdDinner%20Login%20Screen.png)

**Figure 02**: NerdDinner login page with an open redirection

When we correctly log in, the ASP.NET MVC AccountController’s LogOn action redirects us to the URL specified in the returnUrl querystring parameter. In this case, it’s the URL that the attacker has entered, which is <http://nerddiner.com/Account/LogOn>. Unless we’re extremely watchful, it’s very likely we won’t notice this, especially because the attacker has been careful to make sure that their forged page looks exactly like the legitimate login page. This login page includes an error message requesting that we login again. Clumsy us, we must have mistyped our password.

[](http://i2.asp.net/asp.net/images/whitepapers/security/NerdDinner%20-%20Forged%20Login%20Screen.png)

**Figure 03**: Forged NerdDinner Login screen

When we retype our user name and password, the forged login page saves the information and sends us back to the legitimate NerdDinner.com site. At this point, the NerdDinner.com site has already authenticated us, so the forged login page can redirect directly to that page. The end result is that the attacker has our user name and password, and we are unaware that we’ve provided it to them.

## Looking at the vulnerable code in the AccountController LogOn Action

The code for the LogOn action in an ASP.NET MVC 2 application is shown below. Note that upon a successful login, the controller returns a redirect to the returnUrl. You can see that no validation is being performed against the returnUrl parameter.

**Listing 1 – ASP.NET MVC 2 LogOn action in AccountController.cs**

[HttpPost]

public ActionResult LogOn(LogOnModel model, string returnUrl)

{

if (ModelState.IsValid)

{

if (MembershipService.ValidateUser(model.UserName, model.Password))

{

FormsService.SignIn(model.UserName, model.RememberMe);

if (!String.IsNullOrEmpty(returnUrl))

{

return Redirect(returnUrl);

}

else

{

return RedirectToAction("Index", "Home");

}

}

else

{

ModelState.AddModelError("", "The user name or password provided is incorrect.");

}

}

// If we got this far, something failed, redisplay form

return View(model);

}

Now let’s look at the changes to the ASP.NET MVC 3 LogOn action. This code has been changed to validate the returnUrl parameter by calling a new method in the System.Web.Mvc.Url helper class named IsLocalUrl().

**Listing 2 – ASP.NET MVC 3 LogOn action in AccountController.cs**

[HttpPost]

public ActionResult LogOn(LogOnModel model, string returnUrl)

{

if (ModelState.IsValid)

{

if (MembershipService.ValidateUser(model.UserName, model.Password))

{

FormsService.SignIn(model.UserName, model.RememberMe);

if (Url.IsLocalUrl(returnUrl))

{

return Redirect(returnUrl);

}

else

{

return RedirectToAction("Index", "Home");

}

}

else

{

ModelState.AddModelError("",

"The user name or password provided is incorrect.");

}

}

// If we got this far, something failed, redisplay form

return View(model);

}

This has been changed to validate the return URL parameter by calling a new method in the System.Web.Mvc.Url helper class, IsLocalUrl().

## Protecting Your ASP.NET MVC 1.0 and MVC 2 Applications

We can take advantage of the ASP.NET MVC 3 changes in our existing ASP.NET MVC 1.0 and 2 applications by adding the IsLocalUrl() helper method and updating the LogOn action to validate the returnUrl parameter.

The UrlHelper IsLocalUrl() method actually just calling into a method in System.Web.WebPages, as this validation is also used by ASP.NET Web Pages applications.

**Listing 3 – The IsLocalUrl() method from the ASP.NET MVC 3 UrlHelper class**

public bool IsLocalUrl(string url) {

return System.Web.WebPages.RequestExtensions.IsUrlLocalToHost(

RequestContext.HttpContext.Request, url);

}

The IsUrlLocalToHost method contains the actual validation logic, as shown in Listing 4.

**Listing 4 – IsUrlLocalToHost() method from the System.Web.WebPages RequestExtensions class**

public static bool IsUrlLocalToHost(this HttpRequestBase request, string url) {

if (url.IsEmpty()) {

return false;

}

Uri absoluteUri;

if (Uri.TryCreate(url, UriKind.Absolute, out absoluteUri)) {

return String.Equals(request.Url.Host, absoluteUri.Host,

StringComparison.OrdinalIgnoreCase);

}

else {

bool isLocal = !url.StartsWith("http:", StringComparison.OrdinalIgnoreCase)

&& !url.StartsWith("https:", StringComparison.OrdinalIgnoreCase)

&& Uri.IsWellFormedUriString(url, UriKind.Relative);

return isLocal;

}

}

In our ASP.NET MVC 1.0 or 2 application, we’ll add a IsLocalUrl() method to the AccountController, but you’re encouraged to add it to a separate helper class if possible. We will make two small changes to the ASP.NET MVC 3 version of IsLocalUrl() so that it will work inside the AccountController. First, we’ll change it from a public method to a private method, since public methods in controllers can be accessed as controller actions. Second, we’ll modify the call that checks the URL host against the application host. That call makes use of a local RequestContext field in the UrlHelper class. Instead of using this.RequestContext.HttpContext.Request.Url.Host, we will use this.Request.Url.Host. The following code shows the modified IsLocalUrl() method for use with a controller class in ASP.NET MVC 1.0 and 2 applications.

**Listing 5 – IsLocalUrl() method, which is modified for use with an MVC Controller class**

//Note: This has been copied from the System.Web.WebPages RequestExtensions class

private bool IsLocalUrl(string url)

{

if (string.IsNullOrEmpty(url))

{

return false;

}

Uri absoluteUri;

if (Uri.TryCreate(url, UriKind.Absolute, out absoluteUri))

{

return String.Equals(this.Request.Url.Host, absoluteUri.Host,

StringComparison.OrdinalIgnoreCase);

}

else

{

bool isLocal = !url.StartsWith("http:", StringComparison.OrdinalIgnoreCase)

&& !url.StartsWith("https:", StringComparison.OrdinalIgnoreCase)

&& Uri.IsWellFormedUriString(url, UriKind.Relative);

return isLocal;

}

}

Now that the IsLocalUrl() method is in place, we can call it from our LogOn action to validate the returnUrl parameter, as shown in the following code.

**Listing 6 – Updated LogOn method which validates the returnUrl parameter**

[HttpPost]

public ActionResult LogOn(LogOnModel model, string returnUrl)

{

if (ModelState.IsValid)

{

if (MembershipService.ValidateUser(model.UserName, model.Password))

{

FormsService.SignIn(model.UserName, model.RememberMe);

if (IsLocalUrl(returnUrl))

{

return Redirect(returnUrl);

}

else

{

return RedirectToAction("Index", "Home");

}

}

else

{

ModelState.AddModelError("",

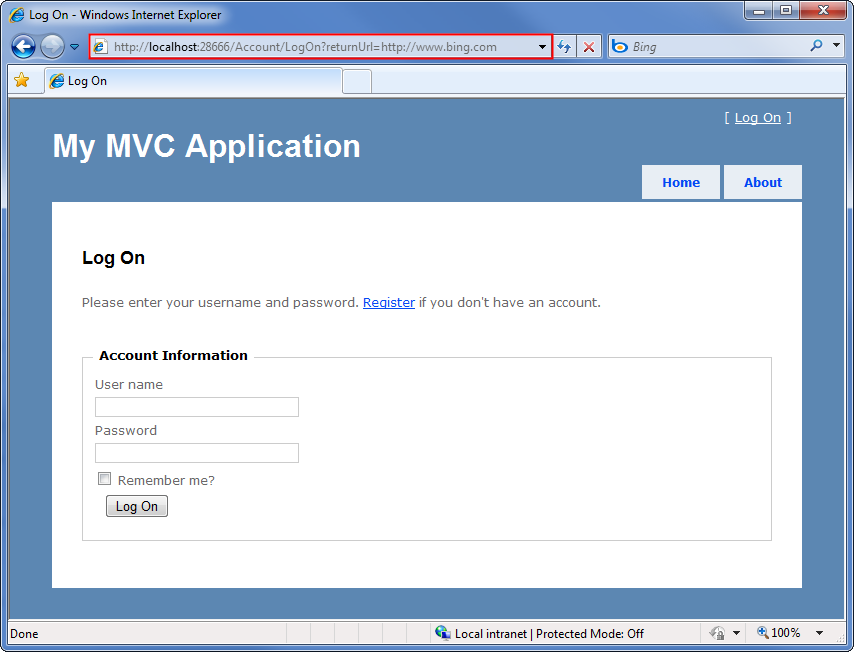
"The user name or password provided is incorrect.");

}

}

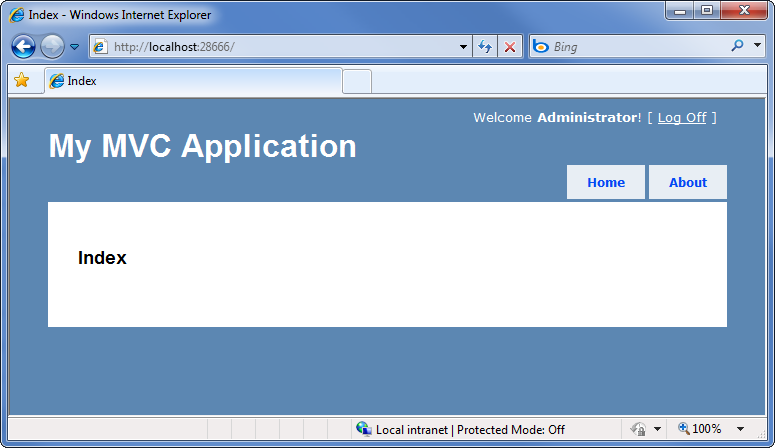
}

Now we can test an open redirection attack by attempting to log in using an external return URL. Let’s use /Account/LogOn?ReturnUrl=http://www.bing.com/ again.

[](http://i2.asp.net/asp.net/images/whitepapers/security/Corrected%20Login%20Screen.png)

**Figure 04**: Testing the updated LogOn Action

After successfully logging in, we are redirected to the Home/Index Controller action rather than the external URL.

[](http://i2.asp.net/asp.net/images/whitepapers/security/Validated%20Login.png)

**Figure 05**: Open Redirection attack defeated

## Summary

Open redirection attacks can occur when redirection URLs are passed as parameters in the URL for an application. The ASP.NET MVC 3 template includes code to protect against open redirection attacks. You can add this code with some modification to ASP.NET MVC 1.0 and 2 applications. To protect against open redirection attacks when logging into ASP.NET 1.0 and 2 applications, add a IsLocalUrl() method and validate the returnUrl parameter in the LogOn action.