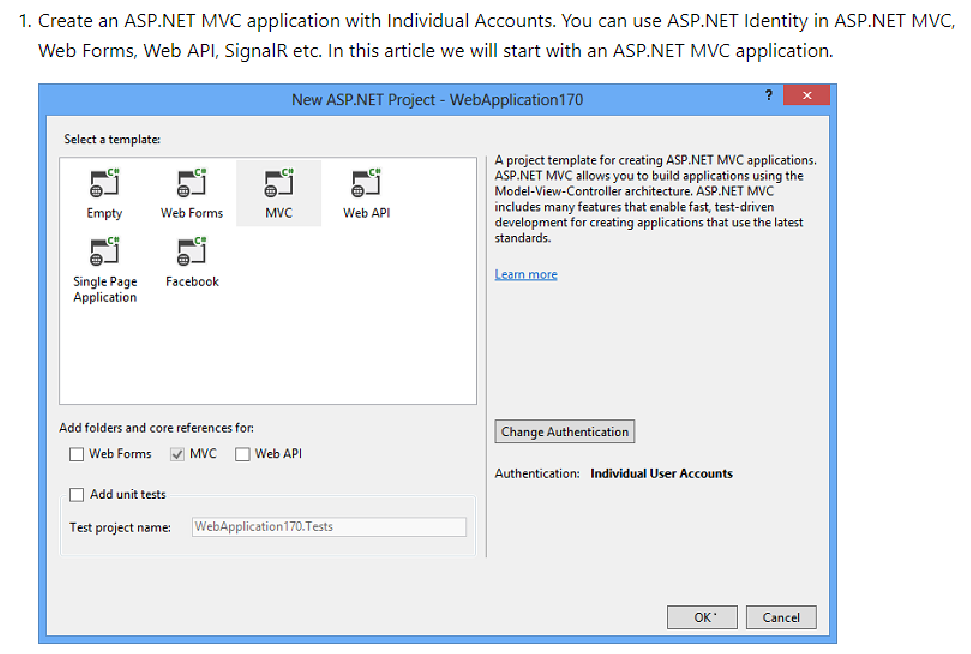
**ASP IDENTITY**

# Getting started with ASP.NET Identity:

1. Create an ASP.NET MVC application with Individual Accounts. You can use ASP.NET Identity in ASP.NET MVC, Web Forms, Web API, SignalR etc. In this article we will start with an ASP.NET MVC application.

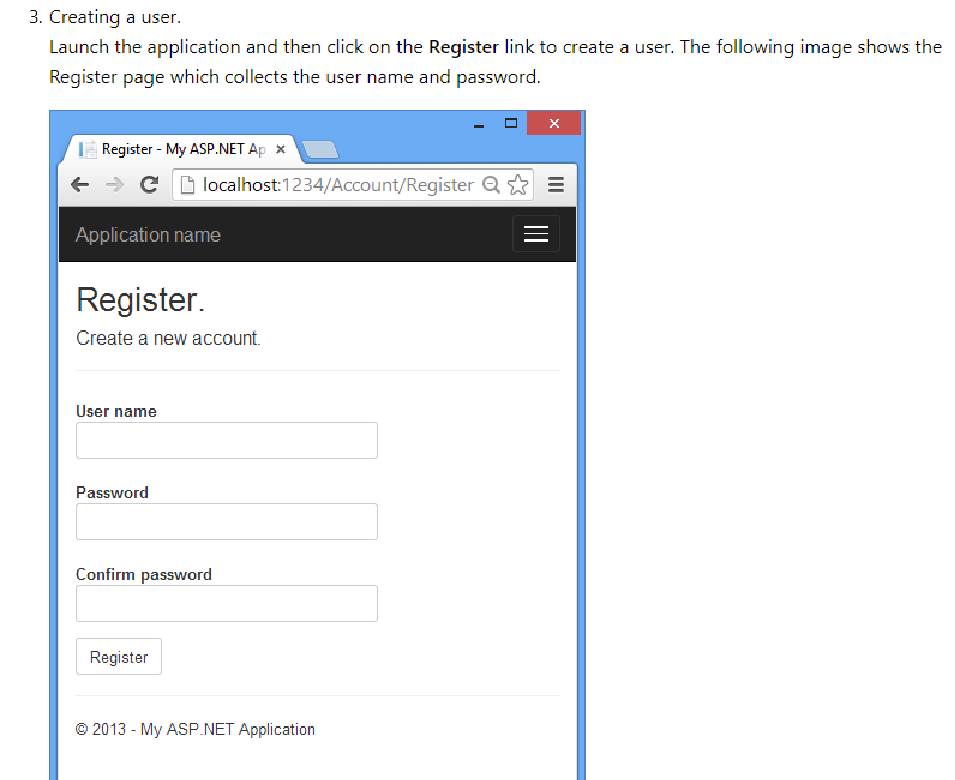


1. The created project contains the following three packages for ASP.NET Identity.

* [Microsoft.AspNet.Identity.EntityFramework](http://www.nuget.org/packages/Microsoft.AspNet.Identity.EntityFramework/)  
  This package has the Entity Framework implementation of ASP.NET Identity which will persist the ASP.NET Identity data and schema to SQL Server.
* [Microsoft.AspNet.Identity.Core](http://www.nuget.org/packages/Microsoft.AspNet.Identity.Core/)  
  This package has the core interfaces for ASP.NET Identity. This package can be used to write an implementation for ASP.NET Identity that targets different persistence stores such as Azure Table Storage, NoSQL databases etc.
* [Microsoft.AspNet.Identity.OWIN](http://www.nuget.org/packages/Microsoft.AspNet.Identity.Owin/)  
  This package contains functionality that is used to plug in OWIN authentication with ASP.NET Identity in ASP.NET applications. This is used when you add log in functionality to your application and call into OWIN Cookie Authentication middleware to generate a cookie.

1. Creating a user.

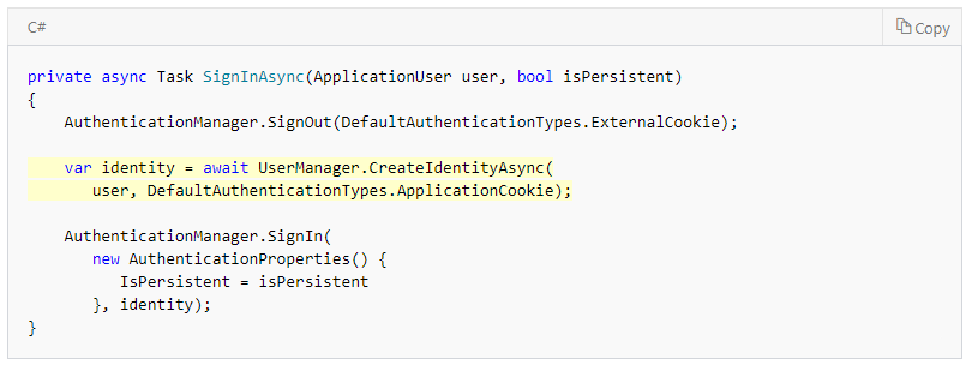
Launch the application and then click on the Register link to create a user. The following image shows the Register page which collects the user name and password.



When the user clicks the **Register** button, the Register action of the Account controller creates the user by calling the ASP.NET Identity API, as highlighted below:

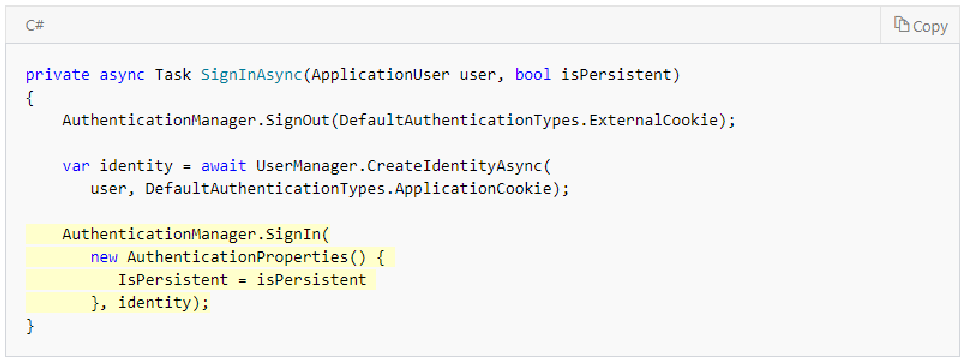
1. Log in.  
   If the user was successfully created, she is logged in by the SignInAsync method.





The highlighted code above in the SignInAsync method generates a [ClaimsIdentity](https://msdn.microsoft.com/library/system.security.claims.claimsidentity.aspx). Since ASP.NET Identity and OWIN Cookie Authentication are claims-based system, the framework requires the app to generate a ClaimsIdentity for the user. ClaimsIdentity has information about all the claims for the user, such as what roles the user belongs to. You can also add more claims for the user at this stage.

The highlighted code below in the SignInAsync method signs in the user by using the AuthenticationManager from OWIN and calling SignIn and passing in the ClaimsIdentity.



1. Log off. Clicking the Log off link calls the LogOff action in the account controller.

# AUTHENTICATION METHODS

* [Individual User Accounts](https://docs.microsoft.com/en-us/aspnet/visual-studio/overview/2013/creating-web-projects-in-visual-studio#indauth) (ASP.NET Identity, formerly known as ASP.NET membership)
* [Organizational Accounts](https://docs.microsoft.com/en-us/aspnet/visual-studio/overview/2013/creating-web-projects-in-visual-studio#orgauth) (Windows Server Active Directory or Azure Active Directory)
* [Windows Authentication](https://docs.microsoft.com/en-us/aspnet/visual-studio/overview/2013/creating-web-projects-in-visual-studio#winauth) (Intranet)

## Individual User Accounts

If you select **Individual User Accounts**, the sample application will be configured to use ASP.NET Identity (formerly known as ASP.NET membership) for user authentication. ASP.NET Identity enables a user to register an account, by creating a username and password on the site or by signing in with social providers such as Facebook, Google, Microsoft Account, or Twitter. The default data store for user profiles in ASP.NET Identity is a SQL Server LocalDB database, which you can deploy to SQL Server or Azure SQL Database for the production site.

In Visual Studio 2013 these features are the same as in Visual Studio 2012, but the underlying code for the ASP.NET membership system has been rewritten. Advantages of the new code base include the following:

* The new membership system is based on [OWIN](http://owin.org/) rather than the ASP.NET Forms Authentication module. This means that you can use the same authentication mechanism whether you're using Web Forms or MVC in IIS, or you're self-hosting Web API or SignalR.
* The new membership database is managed by Entity Framework Code First, and all of the tables are represented by entity classes that you can modify. This means that you can easily customize the database schema and profile-related web UI to fit your own needs, and you can easily deploy your updates using Code First Migrations.

The new membership system is implemented automatically in the new templates, and it can be implemented manually in any project that targets .NET 4.5 or later.

ASP.NET Identity is a good choice if you are creating an Internet web site which is mainly for external customers. If your organization uses Active Directory or Office 365 and you want to create a project that enables single-sign-on for employees and business partners, the **Organizational Accounts** option might be a better choice.

## Organizational Accounts

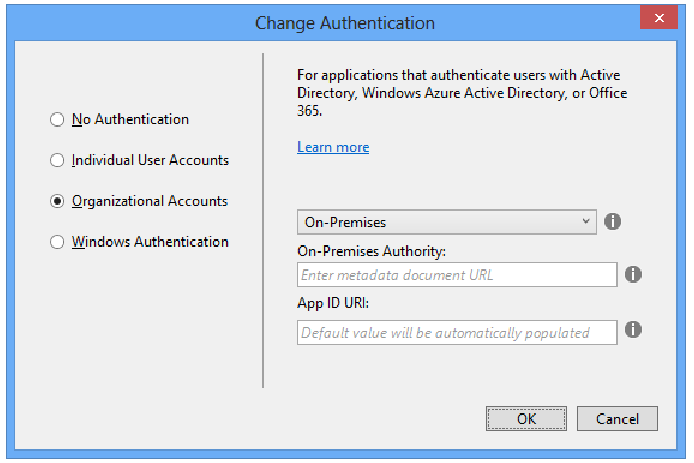
If you select **Organizational Accounts**, the sample application will be configured to use Windows Identity Foundation (WIF) for authentication based on user accounts in Azure Active Directory (Azure AD, which includes Office 365) or Windows Server Active Directory.

The **Configure Authentication** dialog gives you several options for Azure Active Directory (Azure AD, which includes Office 365) or Windows Server Active Directory (AD) account authentication:

* [Cloud - Single Organization](https://docs.microsoft.com/en-us/aspnet/visual-studio/overview/2013/creating-web-projects-in-visual-studio#orgauthsingle) (Azure AD, or AD using directory integration with Azure AD);
* [Cloud - Multi Organization](https://docs.microsoft.com/en-us/aspnet/visual-studio/overview/2013/creating-web-projects-in-visual-studio#orgauthmulti) (Azure AD, or AD using directory integration with Azure AD);
* [On-Premises](https://docs.microsoft.com/en-us/aspnet/visual-studio/overview/2013/creating-web-projects-in-visual-studio#orgauthonprem) (AD);

## On-Premises Organizational Authentication

The screen asks you for two pieces of information:



Choose this option if you want to enable authentication for user accounts that are defined in Windows Server Active Directory (AD), and you don't want to use Azure AD. You can use this option to create an Intranet site or an Internet site. For an Internet site, use Active Directory Federation Services (ADFS) to provide access to AD. For more information, see [Use the On-Premises Organizational Authentication Option (ADFS) With ASP.NET in Visual Studio 2013](http://www.cloudidentity.com/blog/2014/02/12/use-the-on-premises-organizational-authentication-option-adfs-with-asp-net-in-visual-studio-2013/).

For an Intranet site, as an alternative you can choose [Windows Authentication](https://docs.microsoft.com/en-us/aspnet/visual-studio/overview/2013/creating-web-projects-in-visual-studio#winauth) instead of this option. For the Windows Authentication option you don't have to provide a metadata document URL. However, Windows Authentication does not give you the ability to control application access in Active Directory or to query directory data.

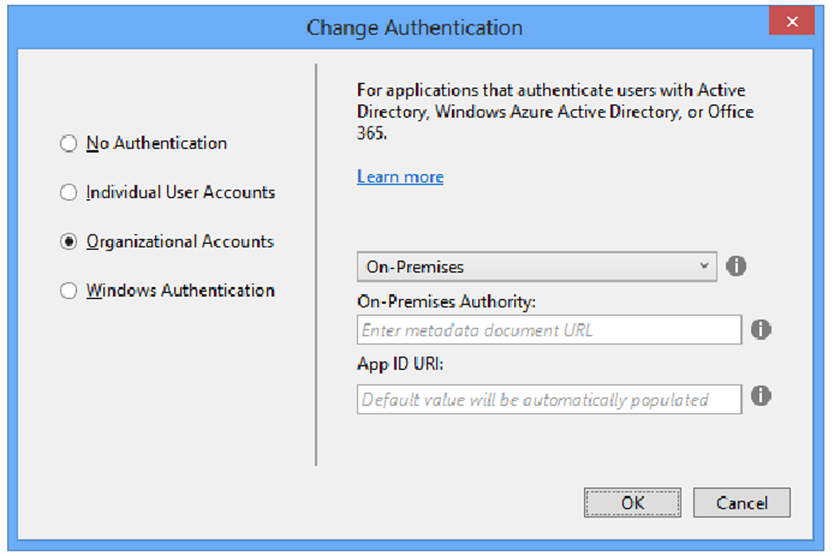
### On-Premises Authority

Enter a URL that points to the metadata document. The metadata document contains the coordinates of the authority. The application will use those coordinates to drive the web sign-on flow.

### Application ID URI

Provide a unique URI that AD can use to identify this application, or leave blank to let Visual Studio create one.

## Use the On-Premises Organizational Authentication Option (ADFS) With ASP.NET in Visual Studio 2013



The screen asks you for two pieces of information:

“On-Premises Authority” – this represents the URL of the metadata document of your authority. You’ll typically receive this value from your administrator. The template tool will use that document for discovering all of the relevant info about your ADFS (addresses, signing keys, identifiers, etc) and derive the project’s configuration from it

“App ID URI” – You can leave it blank, and the tool will fill it up for you (with the URL assigned to your project by IIS (express)).

Personally I find it is good practice to assign to your project something that helps me to remember what the app is about, even when I deploy it out of my dev environment, but that’s totally up to you. Also remember, this is a URI not an URL Smile.

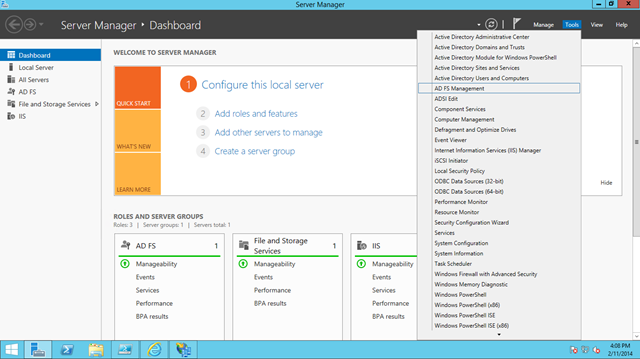
There is another case in which you might want to enter a value here, and that’s when your administrator already provisioned the app in ADFS: in that case, you want to enter here the same realm value he/she entered in ADFS. This will get clearer in the next section.

That’s it. “OK” your way through the various dialogs back to Visual Studio, and you’ll find yourself with a nicely configured MVC app – ready to hit ADFS. However we are not ready to run yet! Before we’re able to do so, we must pay a visit to ADFS and provision our app in it.

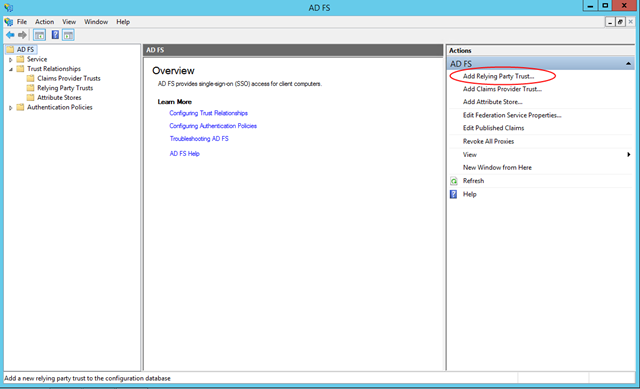
## Provisioning your Web App in ADFS

Whereas the “Cloud” options in the template can automatically provision your app in Windows Azure AD using the Graph API, for ADFS there is no such option. You have to get in touch with your administrator, and convince him/her to provision your application in the ADFS instance. In ADFS parlance, that’s called provisioning a Relying Party Trust. Until that does not happen, ADFS will NOT act on any token request it will receive from your application.

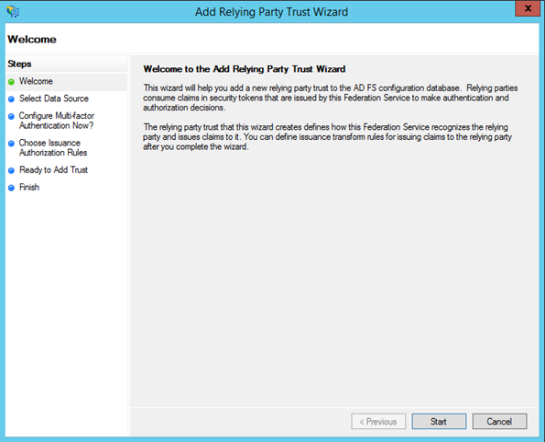
Here I’ll assume that you are an admin on your target ADFS instance. Log in on the box where ADFS is running. From the Server Manager dashboard, launch the ADFS MMC by choosing “Tools->AD FS Management”.



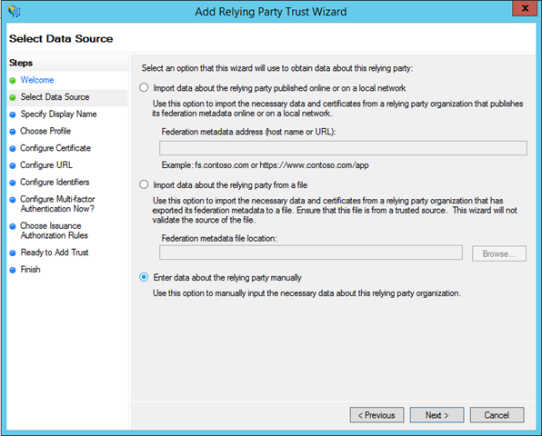
Take a look on the right pane, you’ll find a shortcut for “Add Relying Party Trust…”.



The process is pretty straightforward, ADFS guides you though it. Hit “Start”.

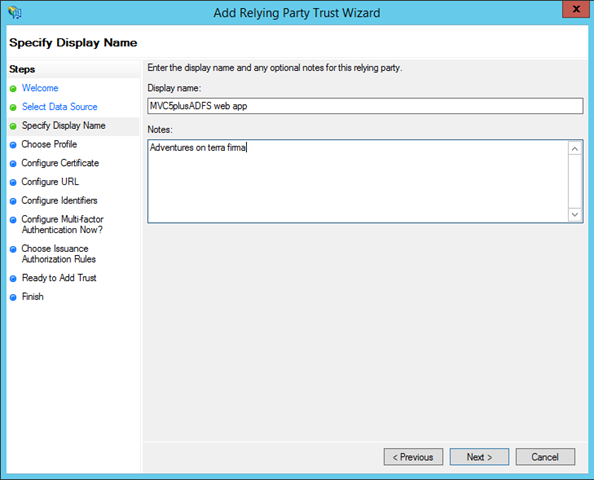


Choose “Enter data about the relying party manually”.



In the first screen you can write whatever and hit next, just remember that you never

know what ends up in production.

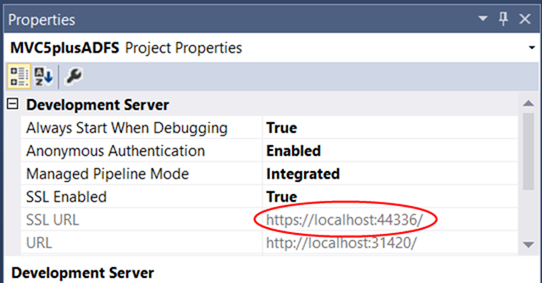


The next screen asks for the URL of your application. That’s easy to find out: switch

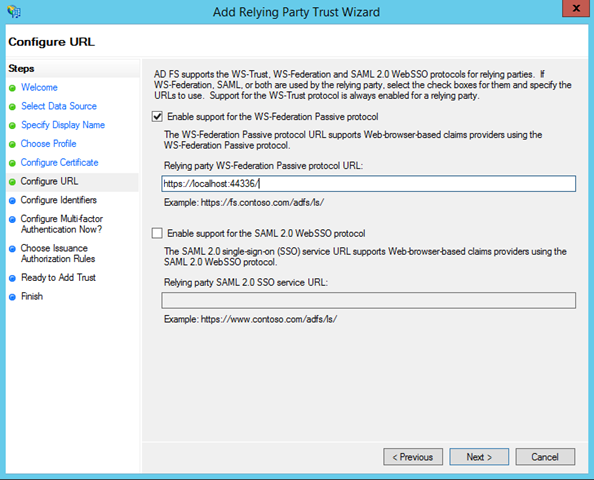
back to Visual Studio, select the project in the project explorer, and take a look at the

properties page (usually in the bottom right corner). There you should see the

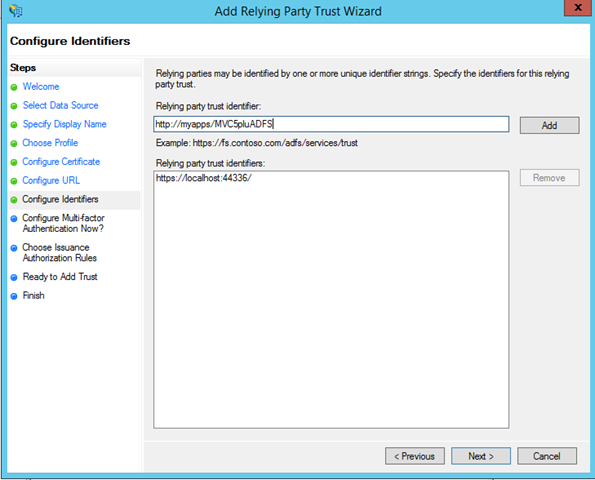
HTTPS address of your app, copy it to the clipboard.



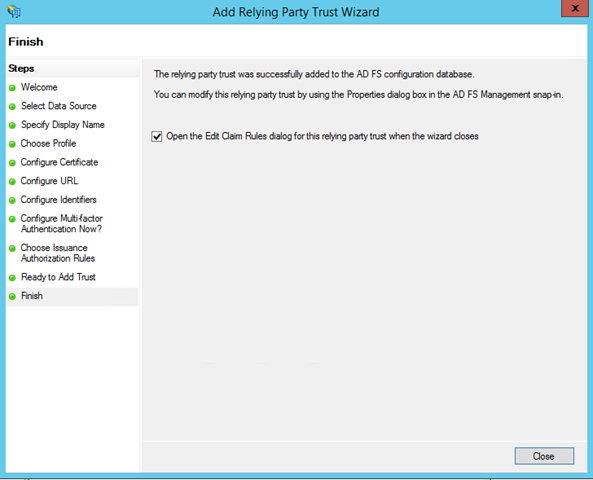
Check the “Enable support for the WS-Federation passive protocol”, and paste the app URL in the corresponding text box. Hit next.



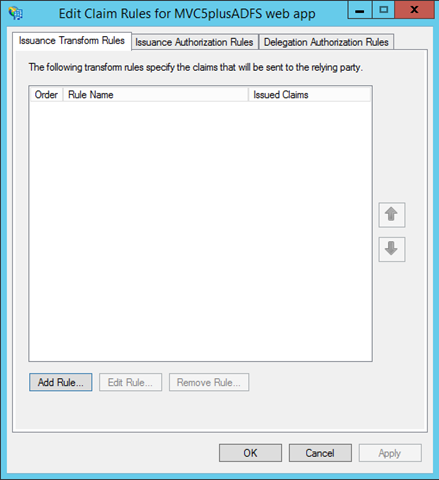
Here there’s where you specify the realm of your application. If you did not choose a custom realm, you are already done here. If you did, you have to paste it in and hit Add.



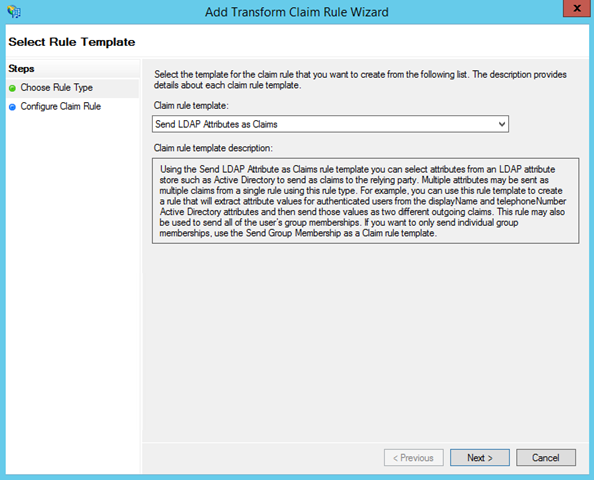
Done that, hit next as many times as it is necessary (without changing any of the defaults) to reach the Finish step.



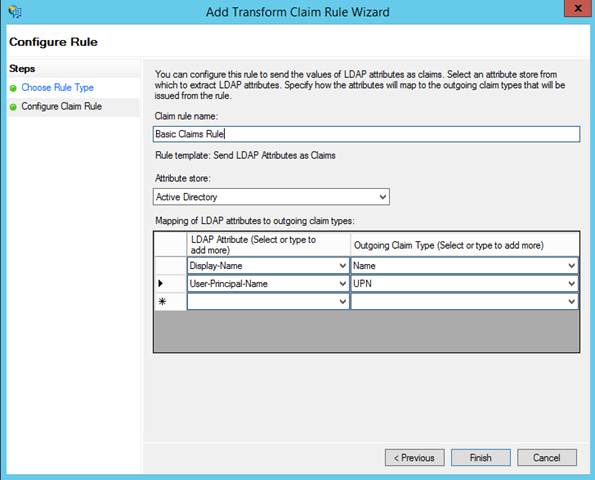
At this point your app is known to ADFS – but we’re not done yet. Here’s why. Whereas Windows Azure AD features a default set of claims that are issued at sign in time, ADFS does not issue ANY claims without your direct configuration. Luckily setting something up is super fast. Click “Close”.



You are presented with another dialog which helps you to set up some claims issuance rule. Stay on this tab and hit Add Rule.



Keep LDAP as the choice, and hit Next.



Name your rule, select Active Directory add the attribute store, and go wild with the claims you want to issue. I always suggest to ensure you are issuing at least one “Name” claim, given that often web apps use that claim for display purposes (the ASP.NET template is not exception) but that’s really up to you.

Once you are satisfied, hit Finish and get back to the MMC. Now we’re ready to run!