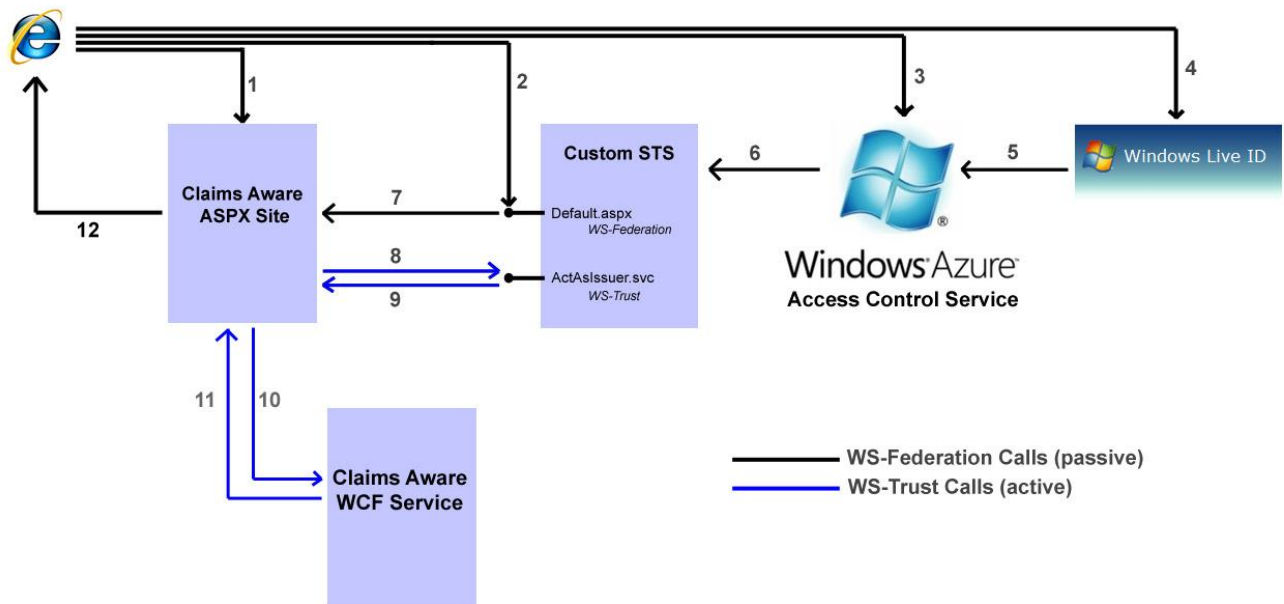


Overview

In this article we will walk you through building an application that federates the Windows Live ID. The application makes use of a custom security token service that exposes a WS-Federation passive endpoint (default.aspx) and also exposes a WS-Trust active endpoint (ActAsIssuer.svc). The passive endpoint federates to Windows Azure Access Control Service (ACS) and the active endpoint will consume the bootstrap token and issue additional claims for that user.

Here is a summarized diagram of the application architecture:



Message Flow

1. Client browser sends a HTTP GET request to a claims aware ASP.NET web application at <https://localhost/ClaimsAwareASPX/default.aspx>
2. WIF intercepts that request and detects that the request does not have the proper security token so WIF redirects the user to the configured issuer, CustomSTS, at <https://localhost/CustomSTS/default.aspx>
3. The CustomSTS federates with Windows Azure ACS so the user is redirected to Windows Azure
4. Windows Azure has a relying party trust for the CustomSTS, with Windows Live ID as the identity provider so the user is redirected to Windows Live to login for authentication
5. The user has been authenticated with Live ID and now has security token with a set of claims added by Windows Live ID. For step five here it's actually not a direct call from Live ID to Azure, that implementation actually goes back to the client browser and the immediately redirected back to Azure

6. Azure accepted the security token from LiveID, performs its authorization, adds or manipulates claims if configured and then redirects the browser back to the CustomSTS passive endpoint.
7. The browser presented a valid security token to our CustomSTS passive endpoint, so the CustomSTS performs any authorization and also has an opportunity to add more or manipulate the set of claims and then redirects the client browser back to the ASPX relying party application but this time with an issued and trusted security token
8. In this step the code inside default.aspx of our ASPX relying party begins to run. Inside the Page_Load event we attempt to call to the backend claims aware WCF service. The WIF and WCF configurations require the ASPX client to make a WS-Trust call to the active endpoint of our CustomSTS to get a required security token before talking with the backend WCF service. The ASPX page passes in the bootstrap token in that call to the CustomSTS active endpoint
9. The CustomSTS Active endpoint (ActAsIssuer.svc) authenticates the caller using ws2007httpbinding, does any desired authorization, and then issues a security token with a set of claims for the bootstrap token user. Our sample adds two more claims to this list and returns the security token back to the ASPX client
10. The ASPX client now has the required security token to call our backend WCF Service. The call is made.
11. The WCF service method executes, it simply enumerates the set of incoming claims that are populated by WIF using the incoming security token on the IClaimsIdentity object and returns the set of claims as a generic list
12. The ASPX relying party app now gets the set of claims returned from our WCF service method and then dumps out the set of claims it received from the CustomSTS passive endpoint and then also dumps out the set of claims it received from the WCF service method all and displays it on the page to the browser client.

Prerequisites

- Install WIF Runtime
- Install WIF SDK
- The Application Pools hosting your web apps have [Load User Profile set to true](#)
- IIS already installed
- Visual Studio .NET 2010
- An account setup with Windows Azure: <https://windows.azure.com>

Phase 1 – Build claims aware web app that requires Windows Live ID claim through ACS

Step 1 – Generate Certificates

For a production system you'll want to use official issued certificates from a trusted root authority or a third party authority like VeriSign. For this walkthrough we'll generate certificates using makecert.exe.

I'm going to generate four different certificates: A root authority certificate (CN=Repro Root Authority), a localhost certificate for SSL (CN=localhost), a signing certificate (CN=Repro Signing Cert), and an encryption certificate (CN=Repro Encryption Cert). Running makecert will install these certificates into the certificate stores during execution however I'll call out any scenario where we have to copy a public key (*.cer) or private key (*.pfx) to a different certificate store.

Use makecert.exe to generate our root authority certificate

1. Open up an Administrative level Visual Studio 2010 Command Prompt
2. Change directories to a working directory
3. Execute the following command to generate the root authority certificate and have it automatically installed into the LocalMachine -> Trusted Root Certification Authorities store

```
makecert -pe -n "CN=Repro Root Authority" -ss AuthRoot -sr LocalMachine -sky signature -r "Repro Root Authority.cer"
```

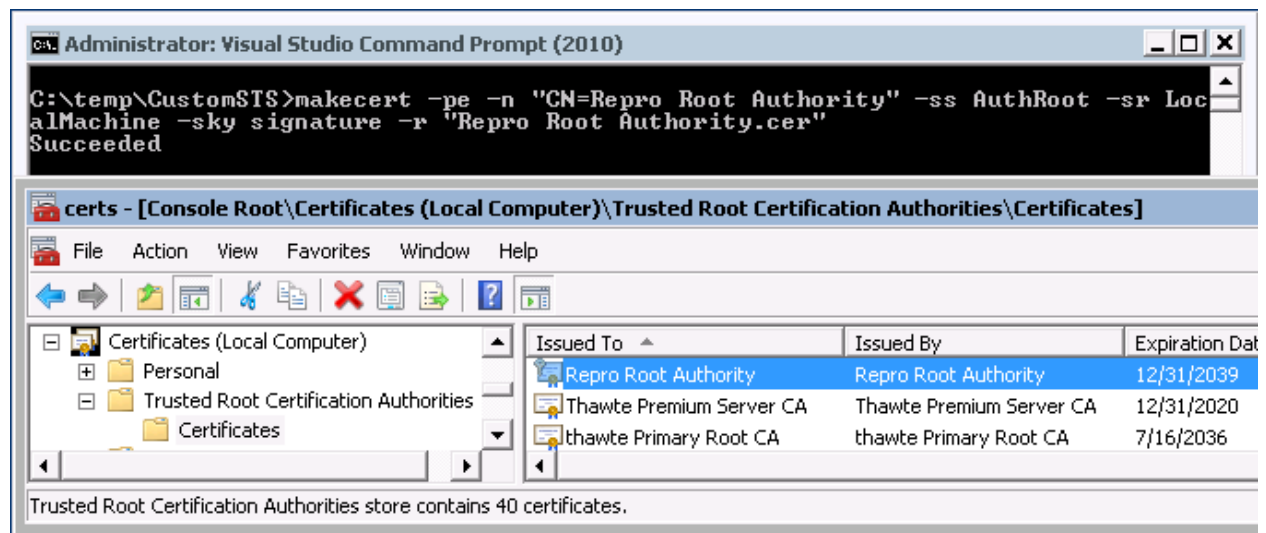


Figure 1 Generate Root Authority Cert. Shows up in Local Machine -> Trusted Root CA store

Use makecert.exe to generate our localhost certificate used for SSL in IIS. (NOTE: I assume that you do not already have a localhost certificate on this machine. If one already exists then you can export, with private key, to back up that existing localhost certificate before proceeding on with these steps)

1. In the same VS.NET command prompt execute the following command to generate the localhost certificate that we'll use in IIS for SSL. This certificate will install into the LocalMachine -> Personal store. Notice here that we sign the certificate with the root authority certificate we just created:

```
makecert -pe -n "CN=localhost" -ss my -sr LocalMachine -sky exchange -a sha1 -in "Repro Root Authority" -is AuthRoot -ir LocalMachine "localhost.cer"
```

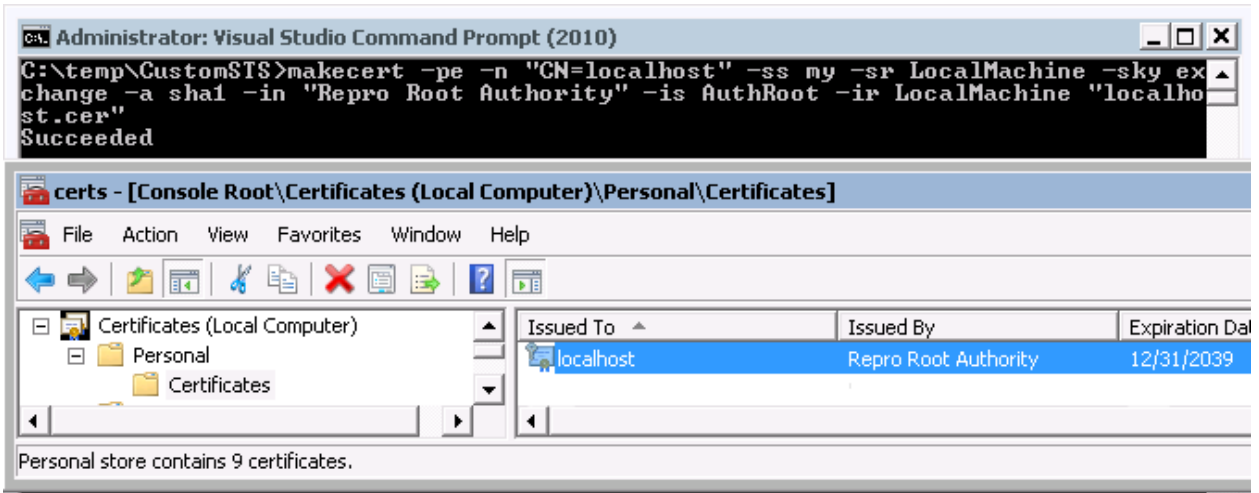


Figure 2 Generate the localhost certificate for SSL. Installed into LocalMachine -> Personal store

Use makecert.exe to generate our signing certificate used for signing tokens by our CustomSTS

1. In the same VS.NET command prompt execute the following command to generate the signing certificate. This certificate will install into the LocalMachine -> Personal store. Notice here that we sign the certificate with the root authority certificate we previously created:

makecert -pe -n "CN=Repro Signing Cert" -ss my -sr LocalMachine -sky exchange -a sha1 -in "Repro Root Authority" -is AuthRoot -ir LocalMachine "Repro Signing Cert.cer"

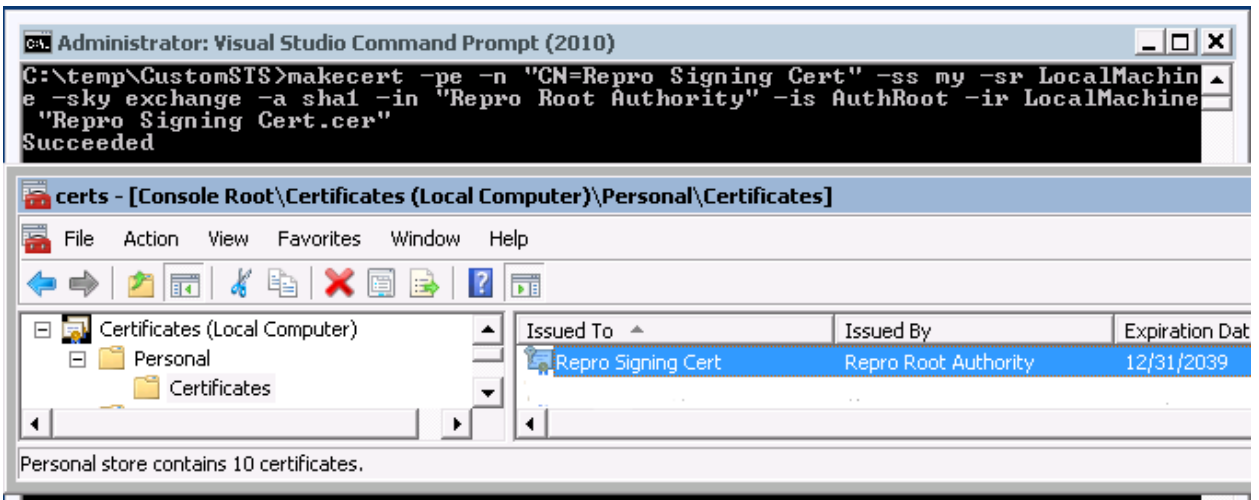


Figure 3 Generate signing cert. Installed into LocalMachine -> Personal store

Use makecert.exe to generate our encryption certificate used for encrypting tokens by our CustomSTS

1. In the same VS.NET command prompt execute the following command to generate the signing certificate. This certificate will install into the LocalMachine -> Personal store. Notice here that we sign the certificate with the root authority certificate we previously created:

makecert -pe -n "CN=Repro Encryption Cert" -ss my -sr LocalMachine -sky exchange -a sha1 -in "Repro Root Authority" -is AuthRoot -ir LocalMachine "Repro Encryption Cert.cer"

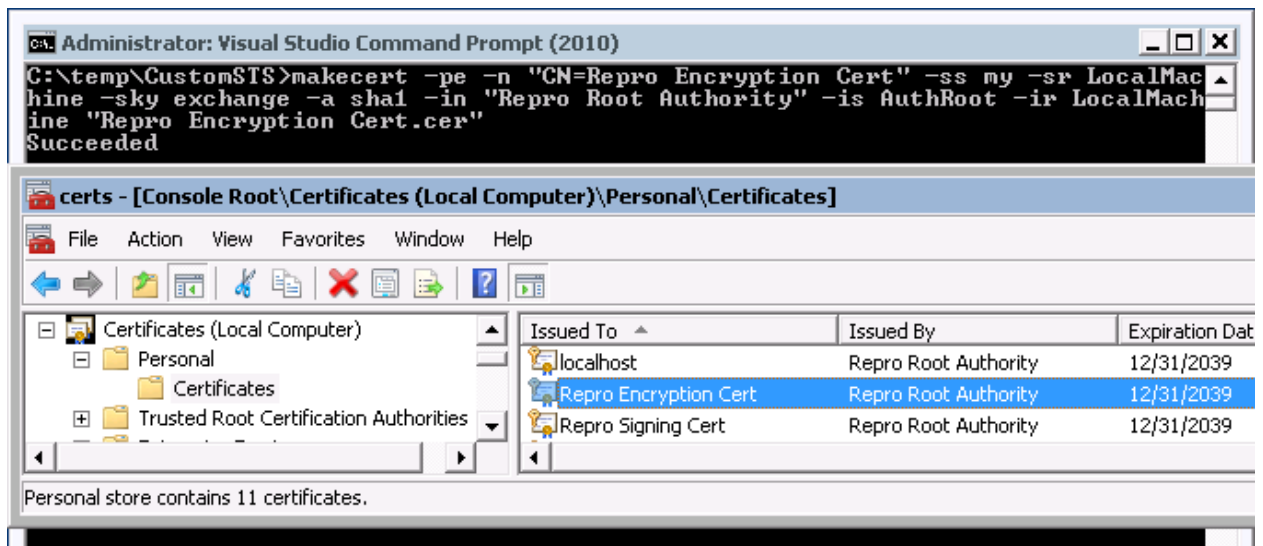
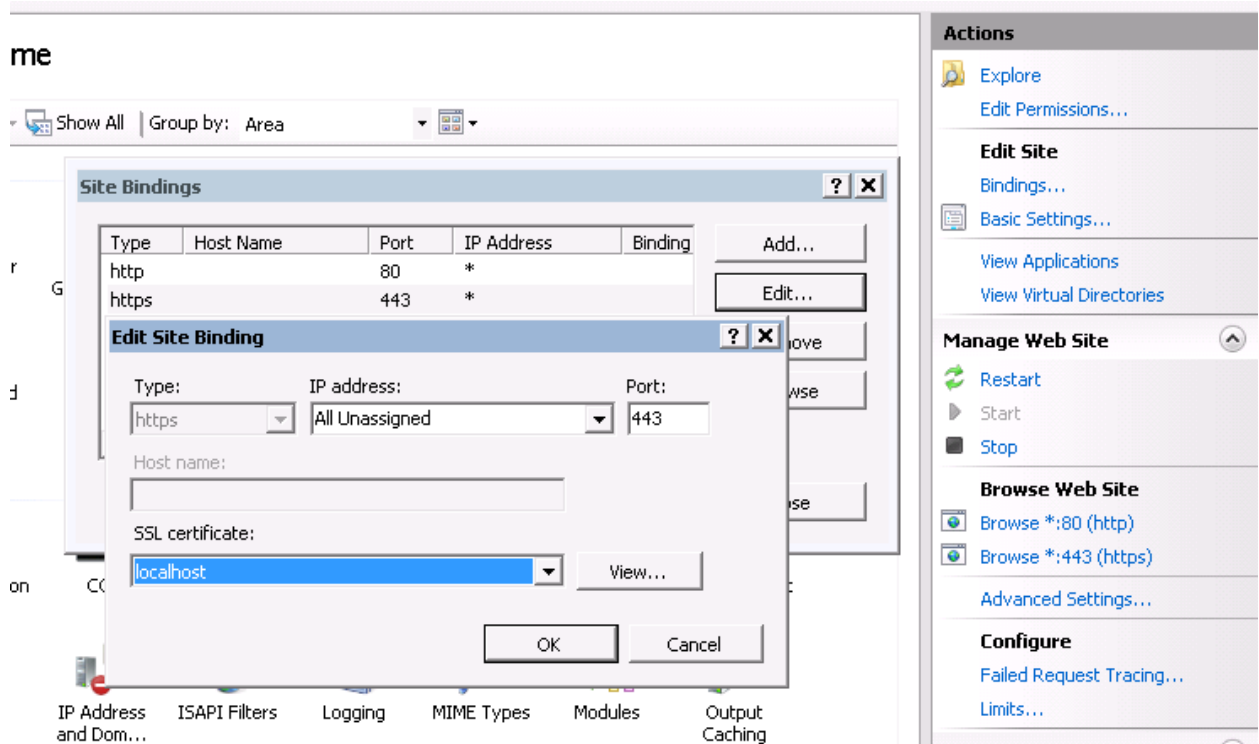


Figure 4 Generate encryption cert. Installed into LocalMachine -> Personal store

Step 2 – Configure IIS for SSL using our localhost certificate

In this step we will configure the local IIS to use the localhost certificate we just created for SSL encryption. In a production scenario you would use a certificate for your publically exposed website or a fully qualified domain name certificate instead.

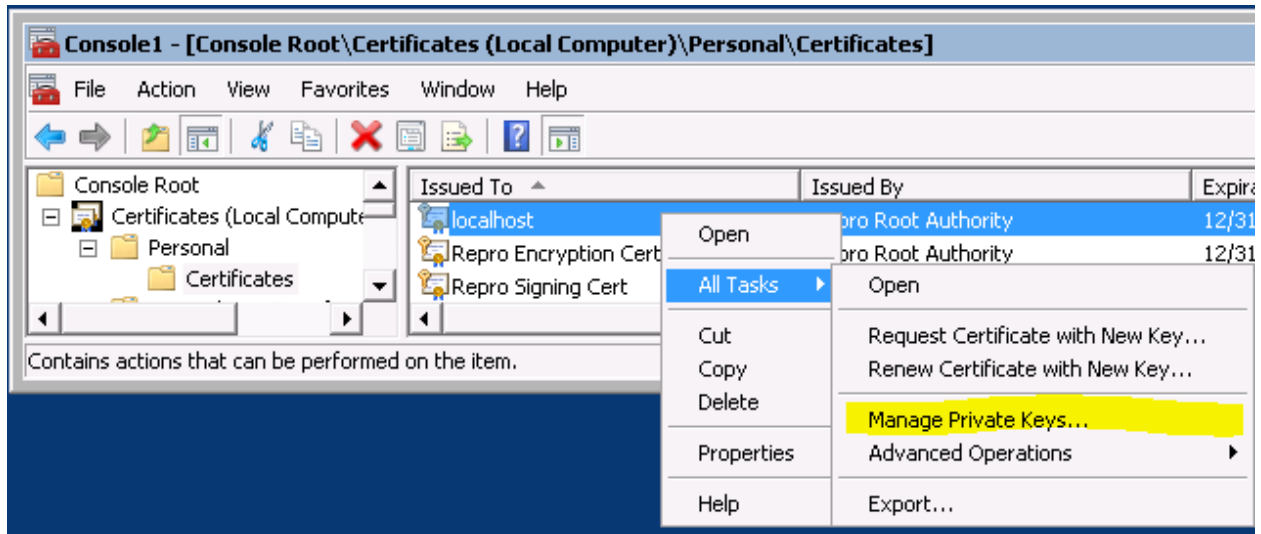
1. Click on Start -> Administrative Tools -> Internet Information Services (IIS) Manager. This launches the IIS management console
2. In IIS manager, click to expand the node containing your machine name, then expand Sites folder. Click to highlight the Default Web Site node and then in the Actions pane on the right hand side click the "Bindings..." link
3. In the Site Bindings dialog select, or click to Add and edit the binding for https type. Change the SSL certificate drop down option and select the localhost certificate that we created in previous steps above. Then click OK button and Close button to apply those changes.



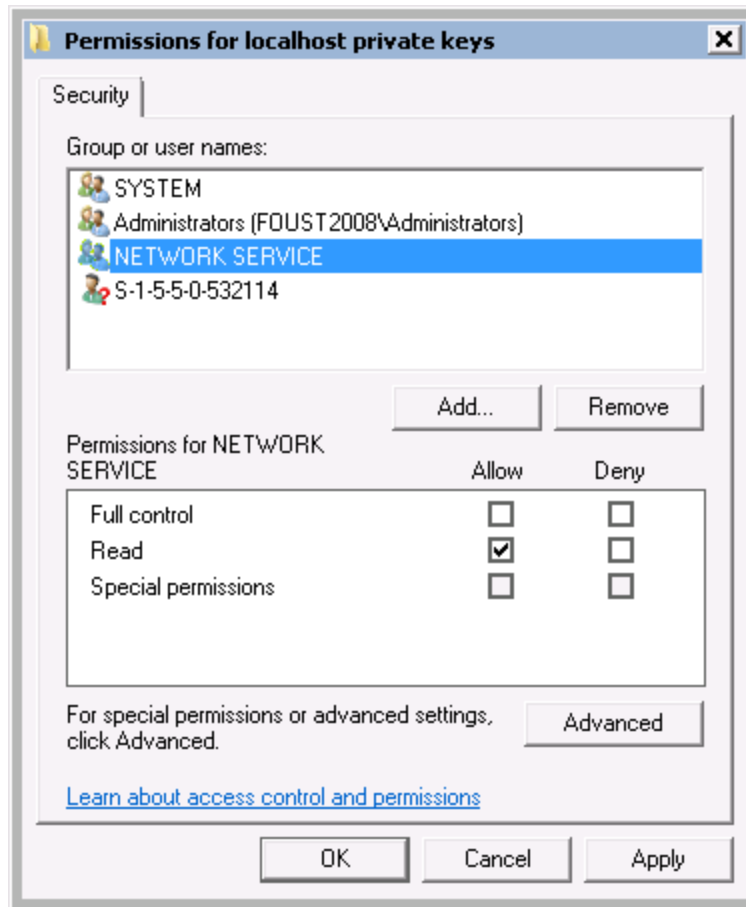
Step 3 – Give permissions of application pool to certificate private keys

We need to give the application pool identity read permissions to the certificate private key. There are several ways to adjust permissions to private keys but we'll use the built in functionality of the certificate manager UI as part of Windows 7 and Windows 2008.

1. Go to Start and enter mmc.exe, press enter to launch the Microsoft Management Console
2. In the console select the File menu and select Add/Remove Snap-in...
3. In the list of available snap-ins on the left hand side select Certificates and click the Add> button.
4. Select the Computer account radio button and click Next
5. Select the Local computer radio button and click Finish, then click OK
6. Expand the Certificates (Local Computer) node
7. Expand the Personal node then click on the Certificates folder for this store
8. You should now see the localhost and two Repro certs that we added to this store in previous steps
9. Right click on the localhost certificate in the list, select **All Tasks**, then click **Manage Private Keys...** option



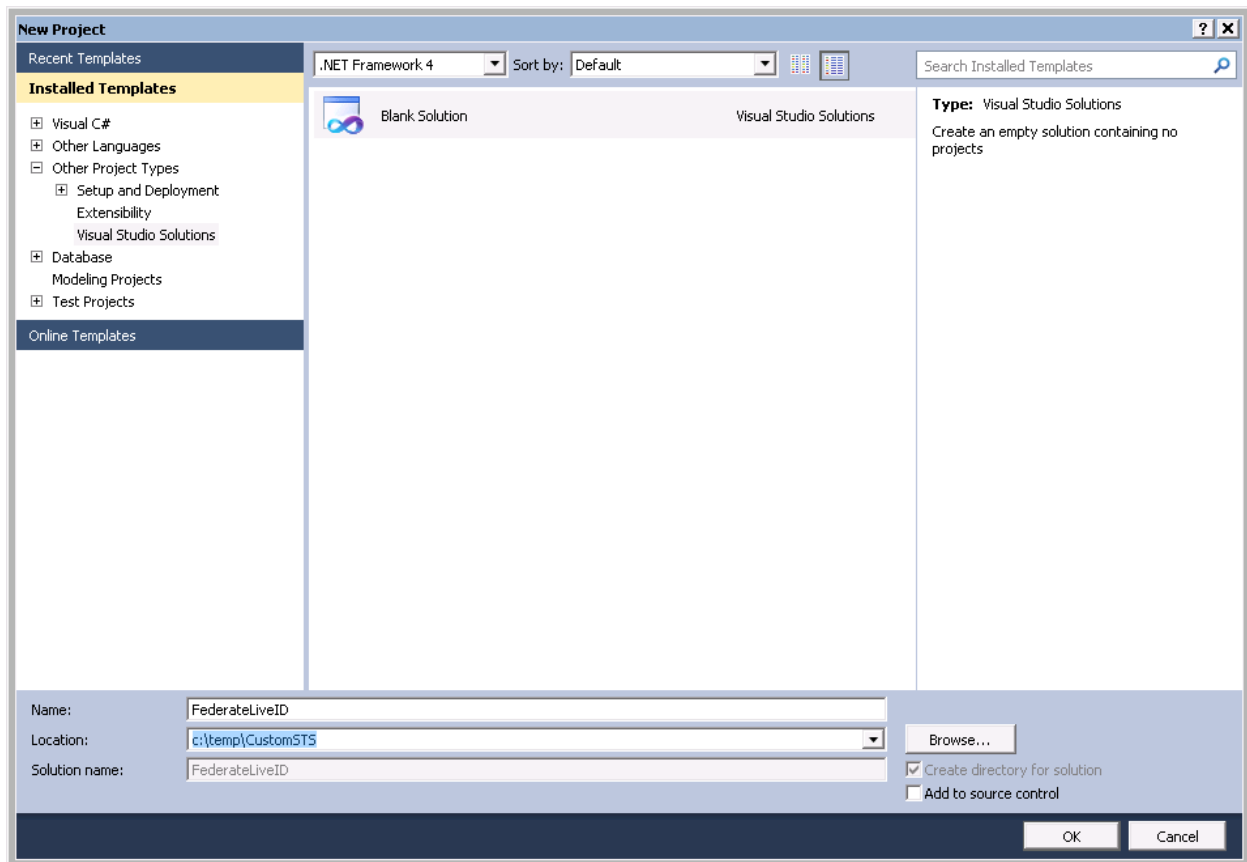
10. This launches the permissions available for this specific localhost private key container file. We need to give the application pool identity read permissions to this private key. My application pool identity is Network Service so I'll add that here. If your application pool is configured to run as ApplicationPoolIdentity in IIS then you would add this user using "IIS APPPOOL\{App Pool Name}" where you replace the {app pool name} with the name of your application pool, ie for the default app pool you would give permissions to the user "IIS APPPOOL\DefaultAppPool". Here is a screen shot of the permissions to my localhost private key after giving the Network Service read permissions:



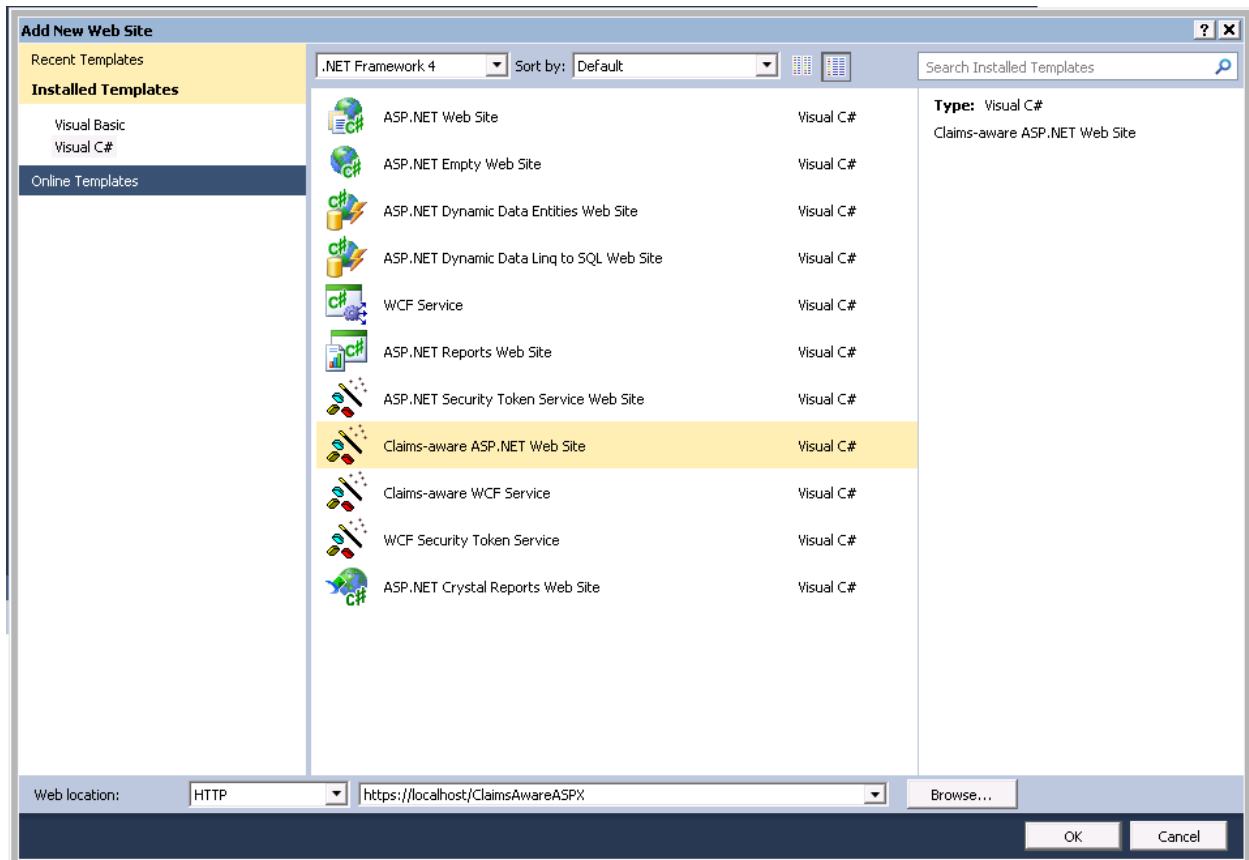
11. Repeat steps 9 and 10 for the **Repro Encryption Cert** and the **Repro Signing Cert** certificates, giving your application pool identity read permissions to those private keys as well.

Step 4 – Create the Claims Aware ASP.NET Web Application (relying party)

1. Launch Visual Studio .NET 2010 as an Administrator
2. Click File menu -> New -> Project...
3. In the New Project dialog expand Other Project Types and select Visual Studio Solutions
4. Select Blank Solution
5. Give the solution a name, for our walkthrough we've selected FederateLiveID, select the location and click OK



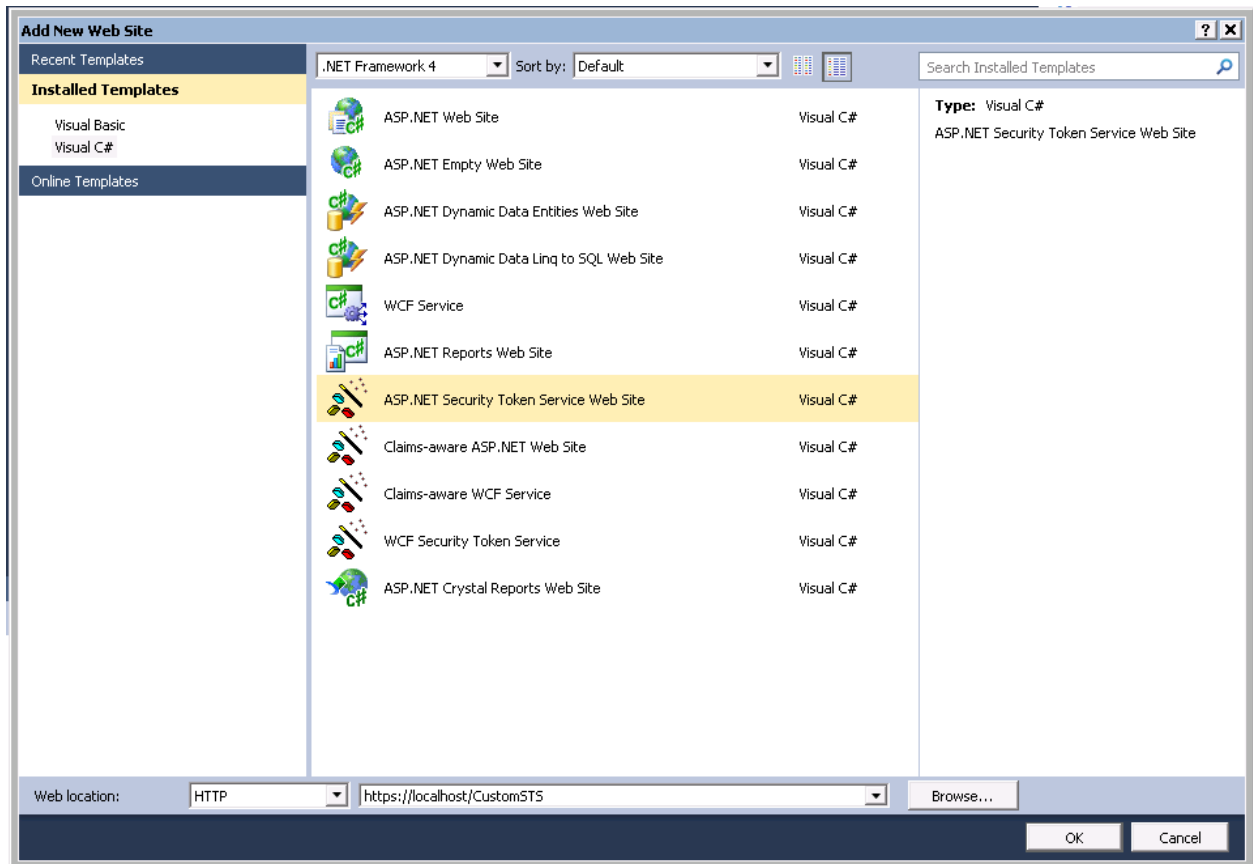
6. Next we will add the ASP.NET Relying Party application to our solution
7. Go to File Menu -> Add -> New Web Site...
8. Select the C# -> Claims-aware ASP.NET Web Site template (WIF 4.0 SDK should be installed if this project type does not show up)
9. Enter the web location of <https://localhost/ClaimsAwareASPX> and then click OK to add the relying party web app to our solution



Step 5 – Create the ASP.NET Security Token Service Web Site

Next we will add a Security Token Service (STS) that will serve as the basis for our CustomSTS. It will contain an endpoint for WS-Federation passive clients and then later we will add an endpoint for WS-Trust active clients.

1. Go to File Menu -> Add -> New Web Site...
2. Select the C# -> ASP.NET Security Token Service Web Site template
3. Enter the web location of `https://localhost/CustomSTS` and then click OK button to add the Security Token Service web app to our solution



4. Now go back to IIS manager
5. Refresh the Default Web Site node in the left hand connections pane
6. Click the ClaimsAwareASPX web app and then double-click on the Authentication icon in the features view. Make sure you have the following authentication modes enabled/disabled:

Anonymous Authentication – Enabled
 ASP.NET Impersonation – Disabled
 Basic Authentication – Disabled
 Forms Authentication – Disabled
 Windows Authentication - Enabled

7. Click the CustomSTS web app and then double-click on the Authentication icon in the features view. Make sure you have the following authentication modes enabled/disabled:

Anonymous Authentication – Enabled
 ASP.NET Impersonation – Disabled
 Basic Authentication – Disabled
 Forms Authentication – Enabled
 Windows Authentication – Enabled

8. Modify the CustomSTS to use our specific signing and encryption certificates. Open the web.config file of the CustomSTS project and in the <appSettings> section enter the following values:

```

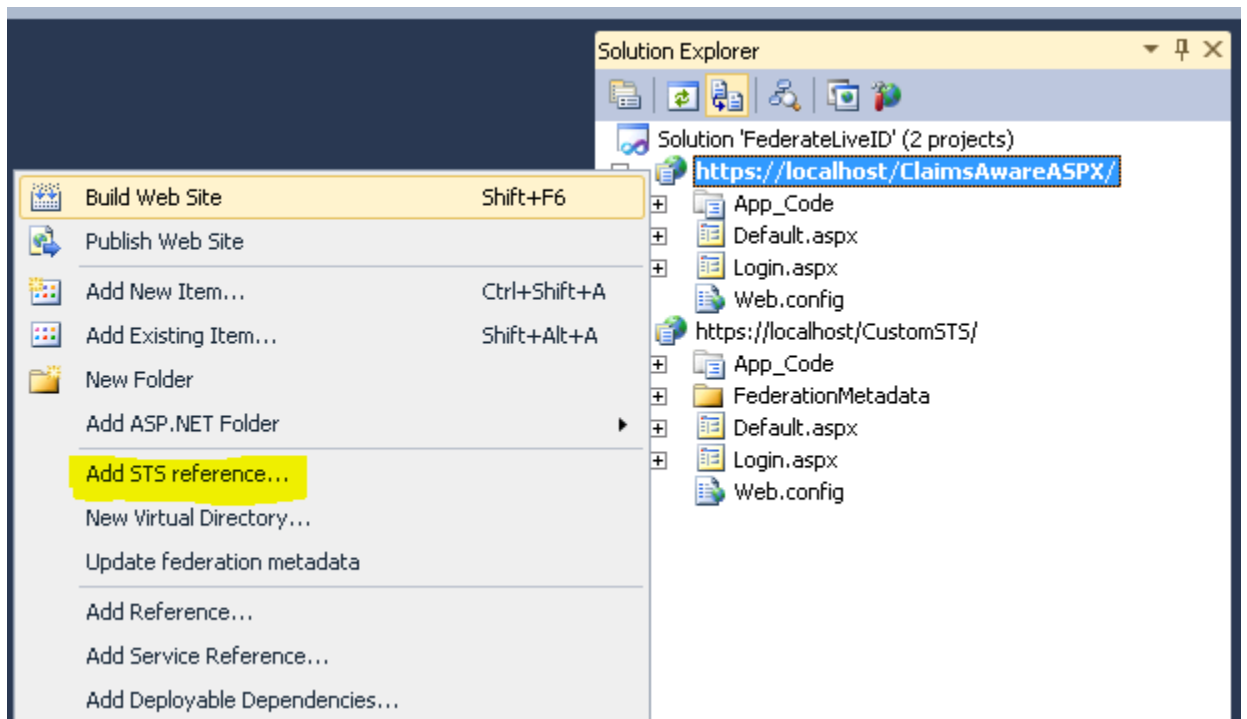
<appSettings>
  <add key="IssuerName" value="PassiveSigninSTS"/>
  <add key="SigningCertificateName" value="CN=Repro Signing Cert" />
  <add key="EncryptingCertificateName" value="CN=Repro Encryption Cert" />
</appSettings>

```

We are simply specifying that our CustomSTS will use certain certificates for signing and encrypting tokens.

Step 6 – Add STS Reference to our CustomSTS

1. In the open VS.NET 2010 select the Build menu and select Build Solution to build the solution in the current state
2. Next, in the solution explorer right click on the <https://localhost/ClaimsAwareASPX/> web app and then select Add STS Reference




3. This launches the Federation Utility (fedutil.exe) to configure this relying party application to require issued tokens from a security token service
4. On the welcome page leave the default settings. These values are picked up from the project properties and auto populate with the path to the relying party web.config file and the application URI for the relying party application. Click Next on this welcome screen
5. On the FedUtil Security Token Service page select the Use an existing STS radio button and then enter the full path to our CustomSTS federation metadata xml file:
<https://localhost/CustomSTS/FederationMetadata/2007-06/FederationMetadata.xml>

Administrator: Federation Utility

Security Token Service

Select a Security Token Service (STS) option.

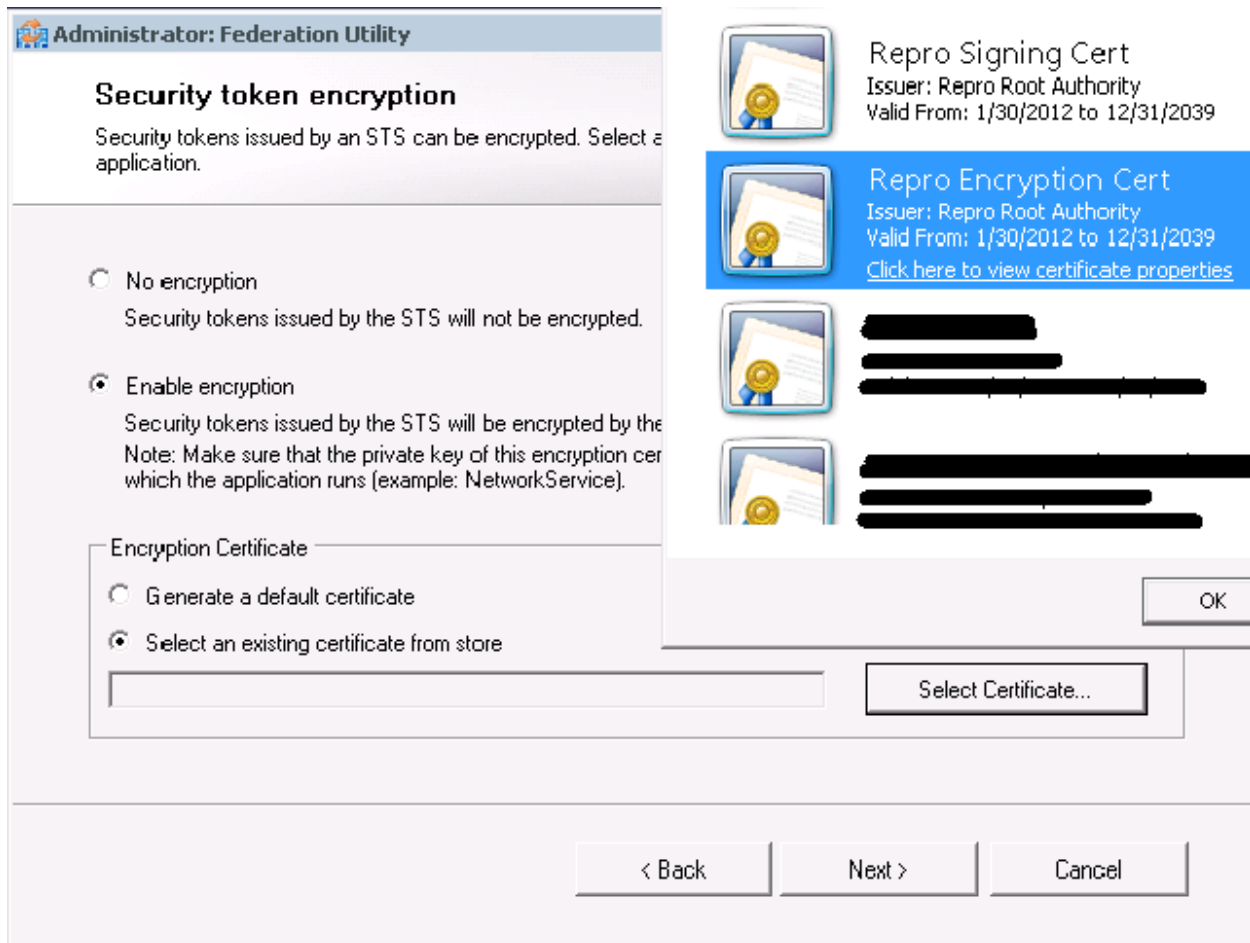


- No STS
Enables claims programming model for the selected application. This option does not require a Security Token Service.
- Create a new STS project in the current solution
A new STS project will be added to the current solution. The selected application's configuration will be modified to trust and accept claims issued by this STS. This option is only available through the 'Add STS reference...' menu item in Visual Studio.
- Use an existing STS
The selected application's configuration will be modified to trust and accept the claims issued by an existing STS. Specify the WS-Federation metadata document location for the existing STS.

STS WS-Federation metadata document location

[Example: <https://fabrikam.com/FederationMetadata/2007-06/FederationMetadata.xml>]

6. Click Next
7. Next you will see the Security Token encryption page. We will be applying encryption for both the passive and active endpoints of our CustomSTS so select Enable encryption radio button.
8. Click the **Select an existing certificate from store** radio button and click the **Select Certificate...** button. This prompts a dialog listing all the available certificates in the Local Machine -> Personal store that you can enable for encryption. Select the **Repro Encryption Cert** certificate that has the issuer **Repro Root Authority** that we created and then click OK.



9. Next you will see the Offered claims available from the CustomSTS. By default there are only two, the Name and Role claims. You can see this by reviewing the contents of the CustomSTS FederationMetadata.xml file that gets generated by the project template. It only shows the Name and Role claims in the available <fed:ClaimTypesOffered><auth:ClaimType...> elements. We can adjust all of these requirements through configuration changes later as needed. Simply click Next on this Offered claims dialog.
10. Review the FedUtil summary page and click Finish button to complete the wizard.

Quick Note: Our CustomSTS is configured to use an encryption certificate and a signing certificate. When you run through the FedUtil wizard it updates the relying party application with these certificate details. Open the config file of your relying party application and find the <microsoft.identityModel> section. The encryption certificate that we have configured for our CustomSTS will be referenced in the <microsoft.identityModel><serviceCertificate><certificateReference/> element of the relying party config file. The signing certificate that we configured for our CustomSTS will be reference in the <microsoft.identityModel><issuerNameRegistry><trustedIssuers><add/> element of the relying party config file. The one problem with this is that during the FedUtil wizard it will ask you what certificate to use during encryption/decryption process but it doesn't prompt you for which signing certificate to use. This is because the signing certificate information is acquired from the FederationMetadata.xml file of the CustomSTS which defaults to using the STSTestCert

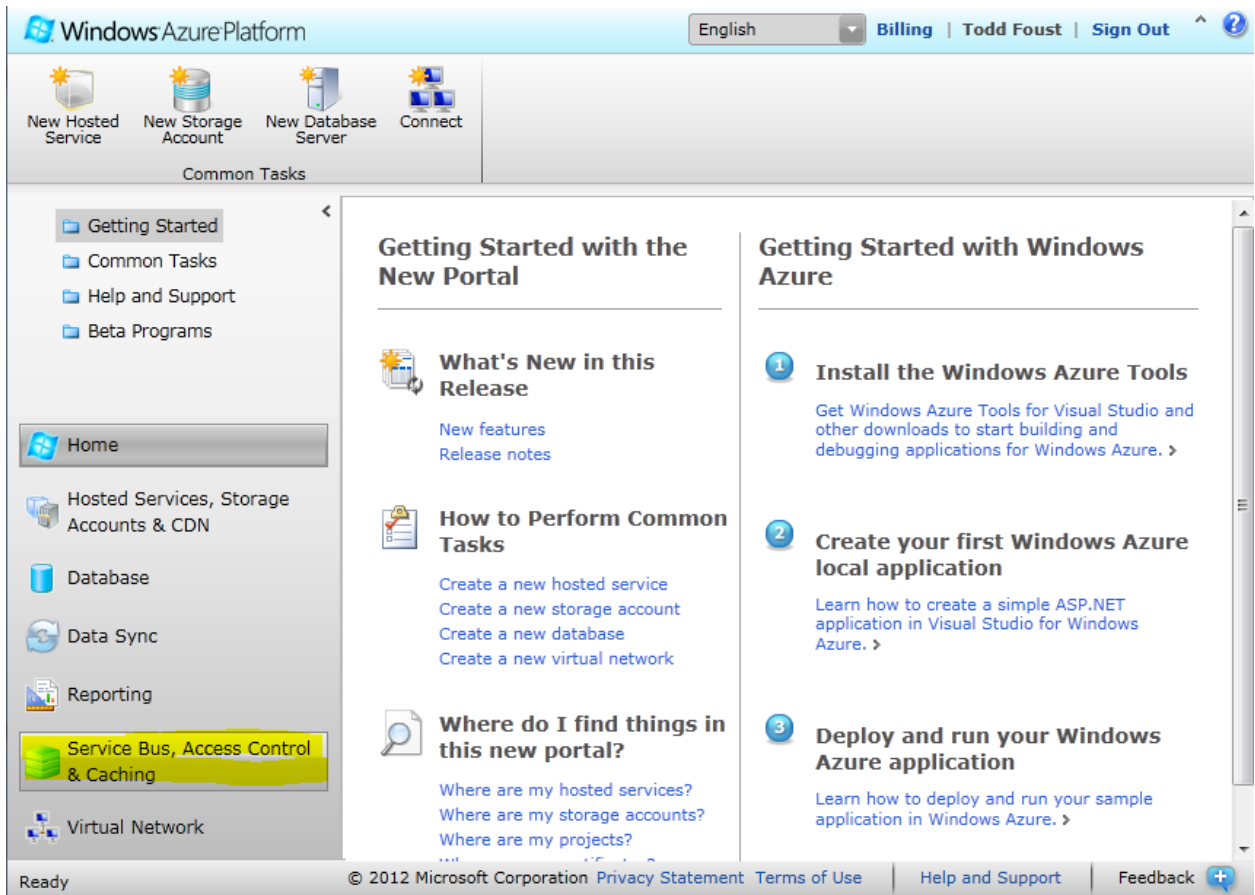
certificate that installs with the WIF SDK. This means that we will have to update the ASPX relying party web.config file, specifically the `<microsoft.identityModel><issuerNameRegistry><trustedIssuers><add/>` element to specify the thumbprint of the Repro Signing Cert instead of the STSTestCert

11. If you build the solution at this point and then attempt to browse to the <https://localhost/ClaimsAwareASPX/default.aspx> page you should see the browser is redirected to the login form of our CustomSTS web app where you can simply click submit to log in and then you'll see the list of default claims created for the user Adam Carter for the out of the box WIF web application templates.

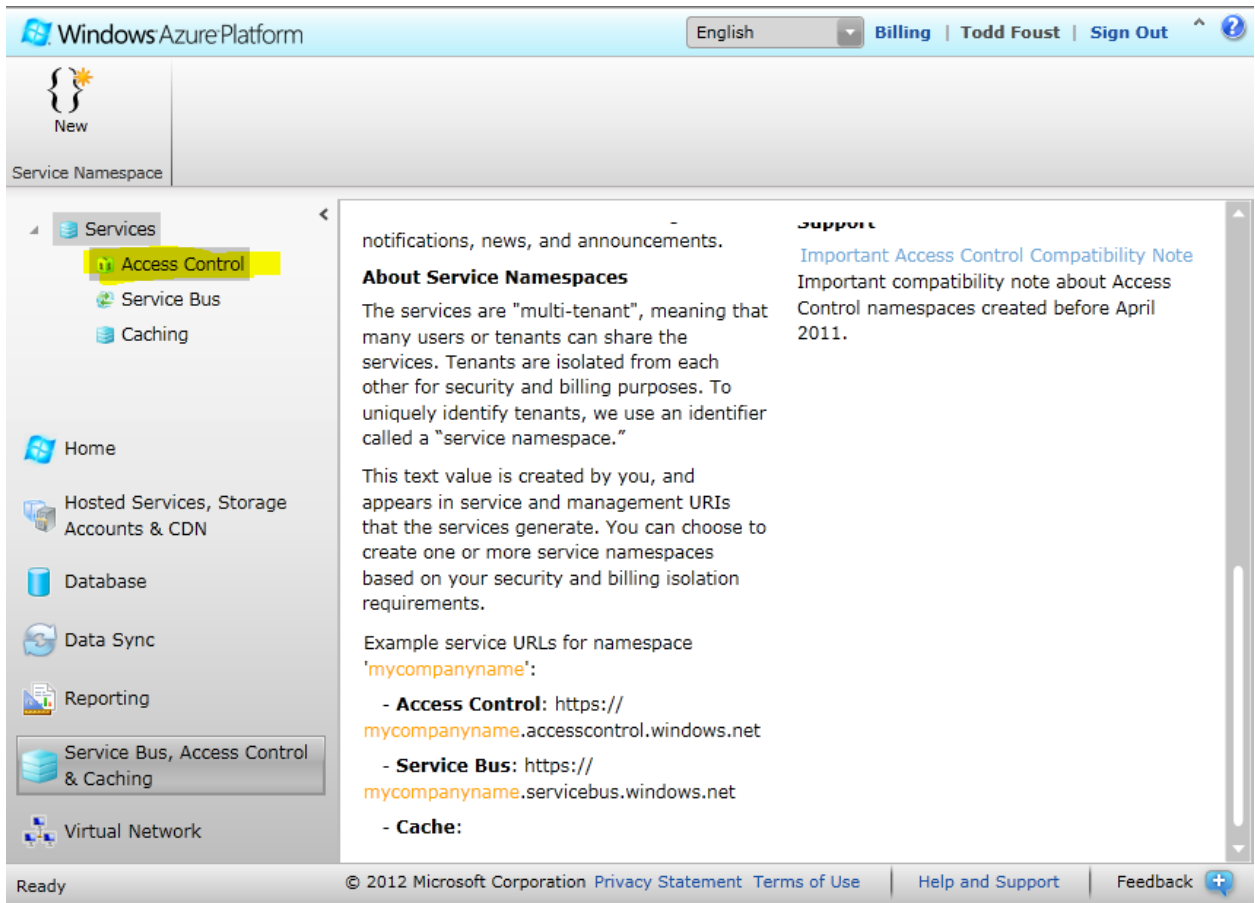
Step 7 – Add Relying Party trust in Azure Access Control Service and configure Windows Live ID as our Identity Provider.

During this step we will register the CustomSTS web application as a relying party application within Windows Azure ACS and we'll configure this trust use Windows Live ID as the identity provider so that users can log into your web application using their live.com credentials. For this step I will assume that you already have an account in Azure. If you don't already have an account, when you browse to this page <https://windows.azure.com/> and log in with your Live ID then you should see a Sign up now button to setup a Windows Azure account.

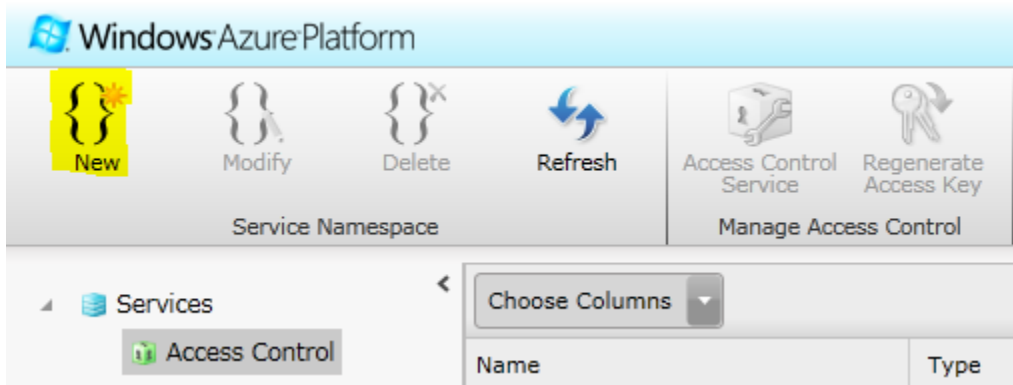
1. Open a browser and browse to <https://windows.azure.com/>
2. When the main page loads on the bottom left hand navigation menu select **Service Bus, Access Control & Caching** button



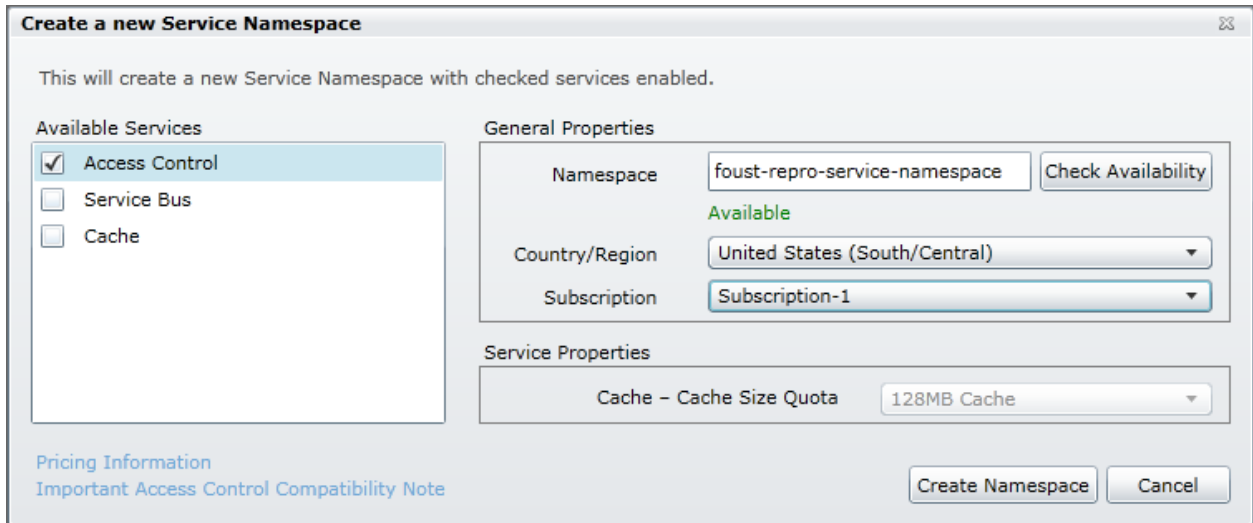
3. Then under the list of Services on the left hand top navigation panel select Access Control:



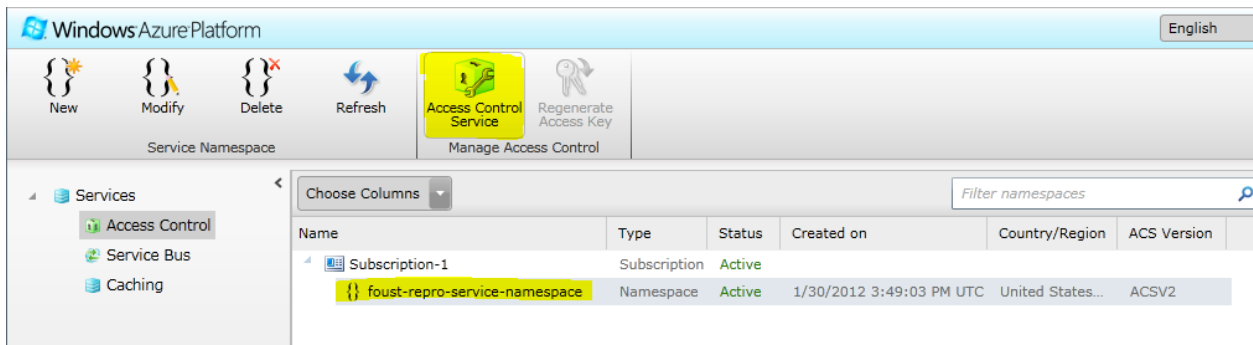
- In ACS we need to create a new Namespace if one does not already exist. Click the New button in the Service Namespace section at the top of the page:



- In the Create a new Service Namespace dialog select Access Control for available services, then enter a unique Namespace string, select appropriate country and subscription. You'll want to click on the Check Availability to make sure that namespace name isn't already taken. For this walkthrough I'm using the Namespace name of "foust-repro-service-namespace":



- Click the **Create Namespace** button. This will take a few minutes to activate the namespace. You can click the Refresh button on the page navigation bar at the top of this page while you wait for the namespace activation to complete. I had to wait two minutes before that namespace become active.
- Click on the namespace that we just created and then click the Access Control Service button at the top to configure ACS for this namespace:



- This takes you to the main Access Control Service (ACS) page. On the left hand side is a navigation column where you can adjust many settings for ACS.
- First click on the Identity Providers link. In ACS the Windows Live ID is already added as an Identity Provider. For our walkthrough we'll use this default identity provider, however you can add other identity providers like Facebook, Google, Yahoo! or even your own ADFS instance by clicking the Add link above the list of current Identity Providers:

Windows Azure Platform English

Access Control Service Service Namespace: foust-repro-service-namespace

Home

Trust relationships

Identity providers

Relying party applications

Rule groups

Service settings

Certificates and keys

Service identities

Administration

Portal administrators

Management service

Development

Application integration

Identity Providers

Add or manage identity providers with which you want to authenticate into your relying party application. To configure direct authentication with ACS, see [Service Identities](#).

[Add](#) | [Delete](#)

Identity Providers	
<input type="checkbox"/> Name	Type
<input type="checkbox"/> Windows Live ID	Windows Live ID

[Return to Home](#)

10. Click on the Relying party applications navigation link so that we can add our CustomSTS as a relying party to ACS. Click the Add button to add a new Relying Party Trust:

Windows Azure Platform English

Access Control Service Service Namespace: foust-repro-service-namespace

Home

Trust relationships

Identity providers

Relying party applications

Rule groups

Service settings

Certificates and keys

Service identities

Relying Party Applications

Add or manage relying party applications. Relying party applications are your websites, applications, and services for which you want to use ACS to implement federated authentication. Relying party applications consume claims from identity providers to make authentication and authorization decisions. [Learn more about relying party applications](#).

[Add](#) | [Delete](#)

Relying Party Applications

No relying party applications are configured in this service namespace. To configure a relying party application, click Add.

[Return to Home](#)

11. On the Add Relying Party Application window fill out the details of the form for our sample walkthrough:

- a. **Name:** https://localhost/CustomSTS/
- b. **Mode:** Enter settings manually
- c. **Realm:** https://localhost/CustomSTS/
- d. **Return URL:** https://localhost/CustomSTS/default.aspx
- e. **Token format:** SAML 1.1
- f. **Token encryption policy:** Require Encryption
- g. **Token lifetime (secs):** 600
- h. **Identity providers:** leave Windows Live ID checked
- i. **Rule groups:** leave Create new rule group checked
- j. **Token signing:** Use service namespace certificate (standard) Note: ACS will sign the SAML token that it issues. The public key, used in verifying signature, will be configured in our CustomSTS when we add an STS reference to this ACS instance.
- k. **Token Encryption:** Click Browse and find our encryption cert, currently at "c:\temp\customsts\Repro Encryption Cert.cer". Note: ACS will use the public key

(* .cer) of this certificate to encrypt the token it issues to our CustomSTS. The CustomSTS already has access to the private key needed to decrypt the token and get its claims:

Windows Azure Platform

Access Control Service Service Namespace: foust-repro-service-namespace > Relying party applications >

Home

Trust relationships

Identity providers

Relying party applications

Rule groups

Service settings

Certificates and keys

Service identities

Administration

Portal administrators

Management service

Development

Application integration

Add Relying Party Application

Use the following options to configure your relying party application in this service namespace.

Relying Party Application Settings

Name

Enter a display name for this relying party application.

Example: fabrikam.com

Mode

Click to configure your relying party application settings manually or to upload a WS-Federation metadata document with the settings for your relying party application. [Learn more](#)

Enter settings manually

Import WS-Federation metadata

Realm

Enter the URI for which the security token that ACS issues is valid. [Learn more](#)

Example: https://www.fabrikam.com (http://localhost is allowed.)

Return URL

Enter the URL to which ACS returns the security token. [Learn more](#)

Example: https://www.fabrikam.com/index.aspx (http://localhost is allowed.)

Error URL (optional)

Enter the URL to which ACS redirects users if an error occurs during the login process. [Learn more](#)

Example: https://www.fabrikam.com/error.aspx (http://localhost is allowed.)

Token format

Select a token format for ACS to use when it issues security tokens for this relying party application. [Learn more](#)

Token encryption policy

Select an encryption policy for tokens that ACS issues for this relying party application. Note: Encryption must be selected if the application is a web service that uses proof-of-possession tokens over the WS-Trust protocol because this scenario does not work without encryption. [Learn more](#)

Figure 5 Portion of our Relying Party trust registration page

12. Click the Save button at the bottom of this Add Relying Party Application page that we just filled out.
13. You should now see the relying party application we just added, the realm entered and the token format of the token returned by ACS:

Windows Azure Platform

Access Control Service Service Namespace: foust-repro-service-namespace

Home

Trust relationships
 Identity providers
 Relying party applications
 Rule groups

Service settings
 Certificates and keys
 Service identities

Administration
 Portal administrators

Relying Party Applications

Add or manage relying party applications. Relying party applications are your websites, applications, and services for which you want to use ACS to implement federated authentication. Relying party applications consume claims from identity providers to make authentication and authorization decisions. [Learn more about relying party applications.](#)

[Add](#) | [Delete](#)

Relying Party Applications			
<input type="checkbox"/>	Application Name	Realm	Token Format
<input type="checkbox"/>	https://localhost/CustomSTS/	https://localhost/CustomSTS/	SAML 1.1

[Return to Home](#)

14. Next we need to add a set of rules to the rule group created for our Relying Party application registration. In the left hand navigation click the **Rule groups** link. Then click the new rule group that was just added for us **Default Rule Group for https://localhost/CustomSTS/**:

Windows Azure Platform

Access Control Service Service Namespace: foust-repro-service-namespace

Home

Trust relationships
 Identity providers
 Relying party applications
 Rule groups

Service settings
 Certificates and keys
 Service identities

Administration

Rule Groups

Add or manage rule groups. Rule groups define how claims are passed from identity providers to your relying party applications. [Learn more about rule groups.](#)

[Add](#) | [Delete](#)

Rule Groups	
<input type="checkbox"/>	Name
<input type="checkbox"/>	Default Rule Group for https://localhost/CustomSTS/

[Return to Home](#)

15. In the **Edit Rule Group** page you can change the name of the rule group if desired and you can also add specific rules for this group. For more information about adding individual rules that govern what claims are issued in the token you can reference [Rule Groups and Rules](#). For this walkthrough we'll just use the default rules used for issuing LiveID claims
16. Click the Generate link to generate the default rules for LiveID identity provider:

Windows Azure Platform

Access Control Service Service Namespace: foust-repro-service-namespace > Rule groups >

Home

- Trust relationships
 - Identity providers
 - Relying party applications
 - Rule groups**
- Service settings
 - Certificates and keys
 - Service identities
- Administration
 - Portal administrators
 - Management service
- Development
 - Application integration

Edit Rule Group

Use the following options to specify how claims input into ACS are transformed into output claims delivered to your relying party application.


Rule Group Details

Name
Enter a name for the rule group.

Used by the following relying party applications

| |

Rules

 No rules have been added. Click **Generate** to generate rules automatically, or click **Add** to add rules manually.

- Then click the **Generate** button to generate the pass through rules that just pass the claims returned from Windows Live ID identity provider into the claims that ACS issues to our CustomSTS:

Windows Azure Platform

Access Control Service Service Namespace: foust-repro-service-namespace > Rule groups >

Home

- Trust relationships
 - Identity providers
 - Relying party applications
 - Rule groups**
- Service settings
 - Certificates and keys
 - Service identities
- Administration
 - Portal administrators

Generate Rules: Default Rule Group for https://localhost/CustomSTS/

Generate rules in this rule group that pass through the claims available from each identity provider to your relying party applications. [Learn more about rule generation.](#)

Rule Generation Options

Generate rules for:
Select the identity providers to generate rules for. Existing rules will not be modified or deleted. [Learn more](#)

Windows Live ID

- You will see only one rule added. Windows Live ID returns just the nameidentifier that represents the Windows Live ID user. Simply click the **Save** button to save these newly added rules to our rule group:

Home

Trust relationships

Identity providers

Relying party applications

Rule groups

Service settings

Certificates and keys

Service identities

Administration

Portal administrators

Management service

Development

Application integration

Edit Rule Group

Use the following options to specify how claims input into ACS are transformed into output claims delivered to your relying party application.

Rule Group Details

Name

Enter a name for the rule group.

Default Rule Group for https://localhost/CustomSTS/

Used by the following relying party applications

https://localhost/CustomSTS/

Save

Cancel

[Generate](#) | [Add](#) | [Delete](#)

Rules

<input type="checkbox"/>	Output Claim	Claim Issuer	Rule Description
<input type="checkbox"/>	nameidentifier	Windows Live ID	Passthrough "nameidentifier" claim from Windows Live ID as "nameidentifier"

1 of 1

- At this point we have completed our relying party application registration within ACS. ACS will sign the token using its own default certificate and encrypt the token using the public key of our encryption certificate. Our CustomSTS will then be able to decrypt using the private key of our encryption cert and verify the signature using the public key information (which will be found in web.config after we run Add STS Reference next) and then be able to consume the claims before issuing its own token to the ASPX relying party application.
- Still within the Access Control Service administration page click the Application Integration link in the left hand navigation panel. This will show you the specific Endpoint Reference urls you can use to access ACS. Copy the WS-Federation Metadata url link as we will use that in our next step:

Windows Azure Platform English Windows Azure Portal | Si

Access Control Service Service Namespace: foust-repro-service-namespace

Home

Trust relationships

- Identity providers
- Relying party applications
- Rule groups

Service settings

- Certificates and keys
- Service identities

Administration

- Portal administrators
- Management service

Development

- Application integration**

Application Integration

Get the code required to integrate Access Control Service with your relying party applications.

Login Pages

Learn how to configure your relying party applications to show a federated login page.

[▶ Login Pages](#)

SDKs and Documentation

Learn how to configure your relying party applications to consume identity tokens issued by Access Control Service.

[▶ SDKs and Documentation](#)

Endpoint Reference

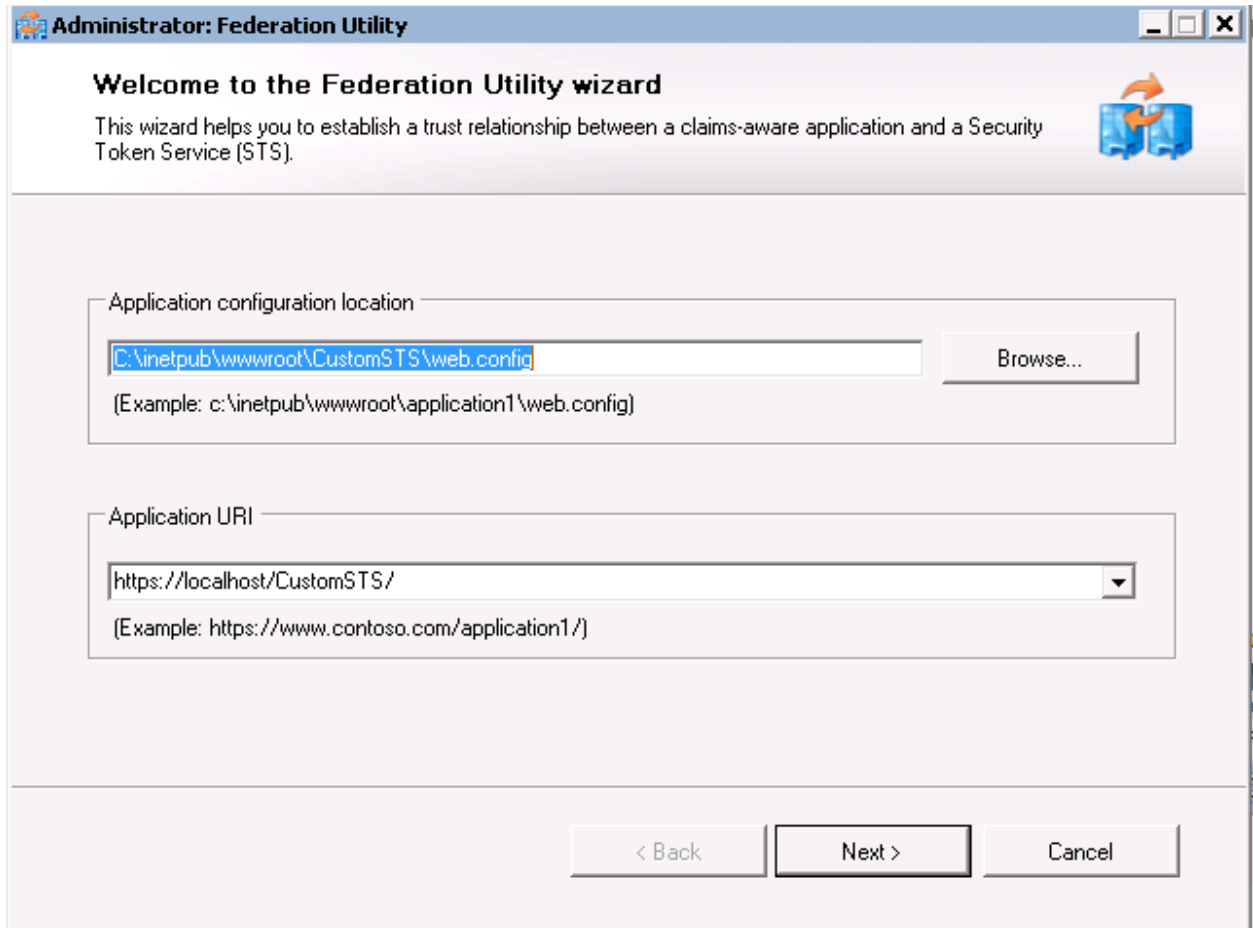
Management Service	https://foust-repro-service-namespace.accesscontrol.windows.net/v2/mgmt/service
Management Portal	https://foust-repro-service-namespace.accesscontrol.windows.net/
OAuth WRAP	https://foust-repro-service-namespace.accesscontrol.windows.net/WRAPv0.9
OAuth2	https://foust-repro-service-namespace.accesscontrol.windows.net/v2/OAuth2-13
WS-Federation Metadata	https://foust-repro-service-namespace.accesscontrol.windows.net/FederationMetadata/2007-06/FederationMetadata.xml
WS-Metadata Exchange	https://foust-repro-service-namespace.accesscontrol.windows.net/v2/wstrust/mex

[Return to Home](#)

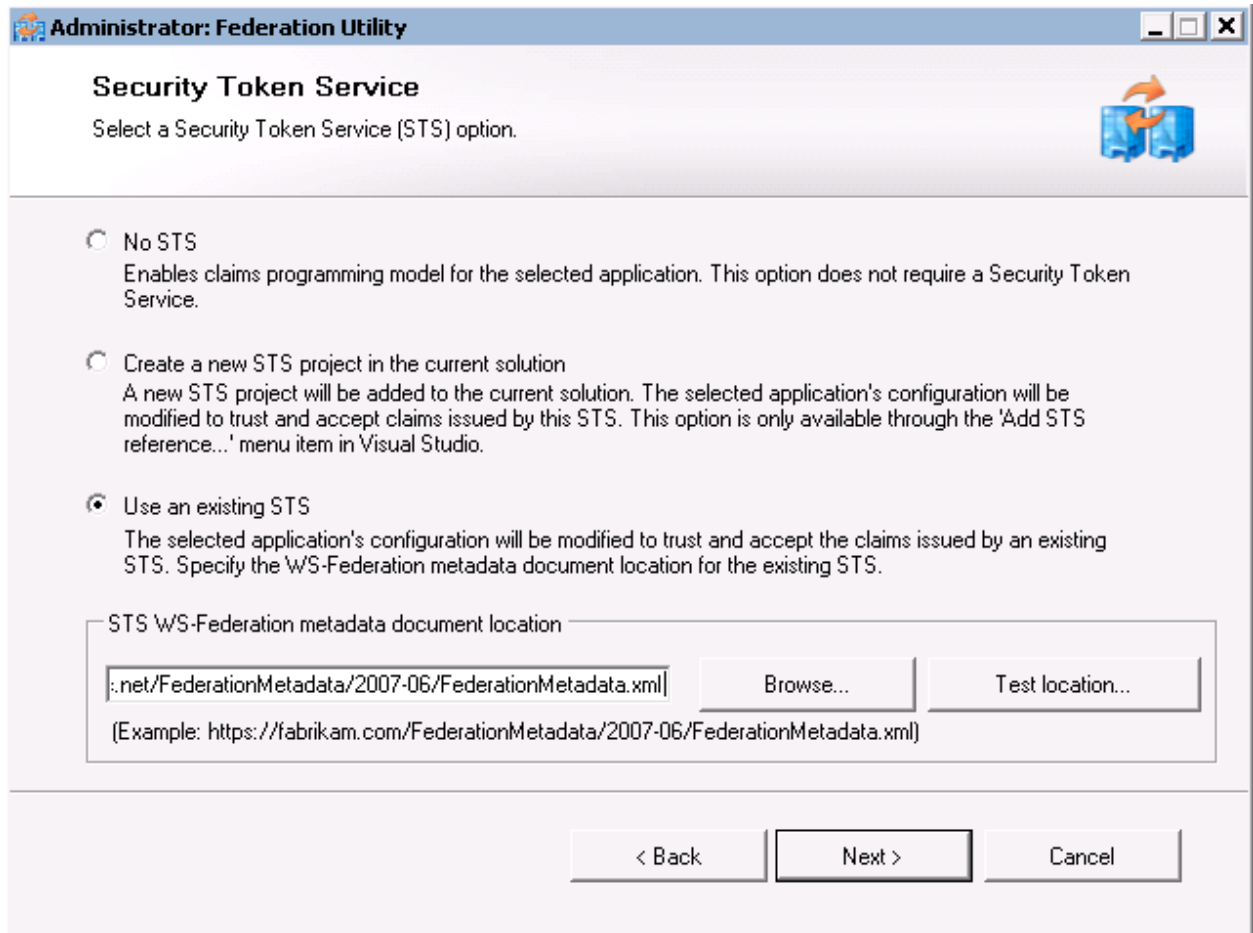
Step 8 – Add STS Reference from our CustomSTS to our Azure ACS endpoint

Our CustomSTS (Security Token Service) web application will get SAML 1.1 tokens from ACS, then it will add any other claims to that token before sending it back to the calling ASPX relying party application. In this step we will configure the CustomSTS to get tokens from Azure ACS.

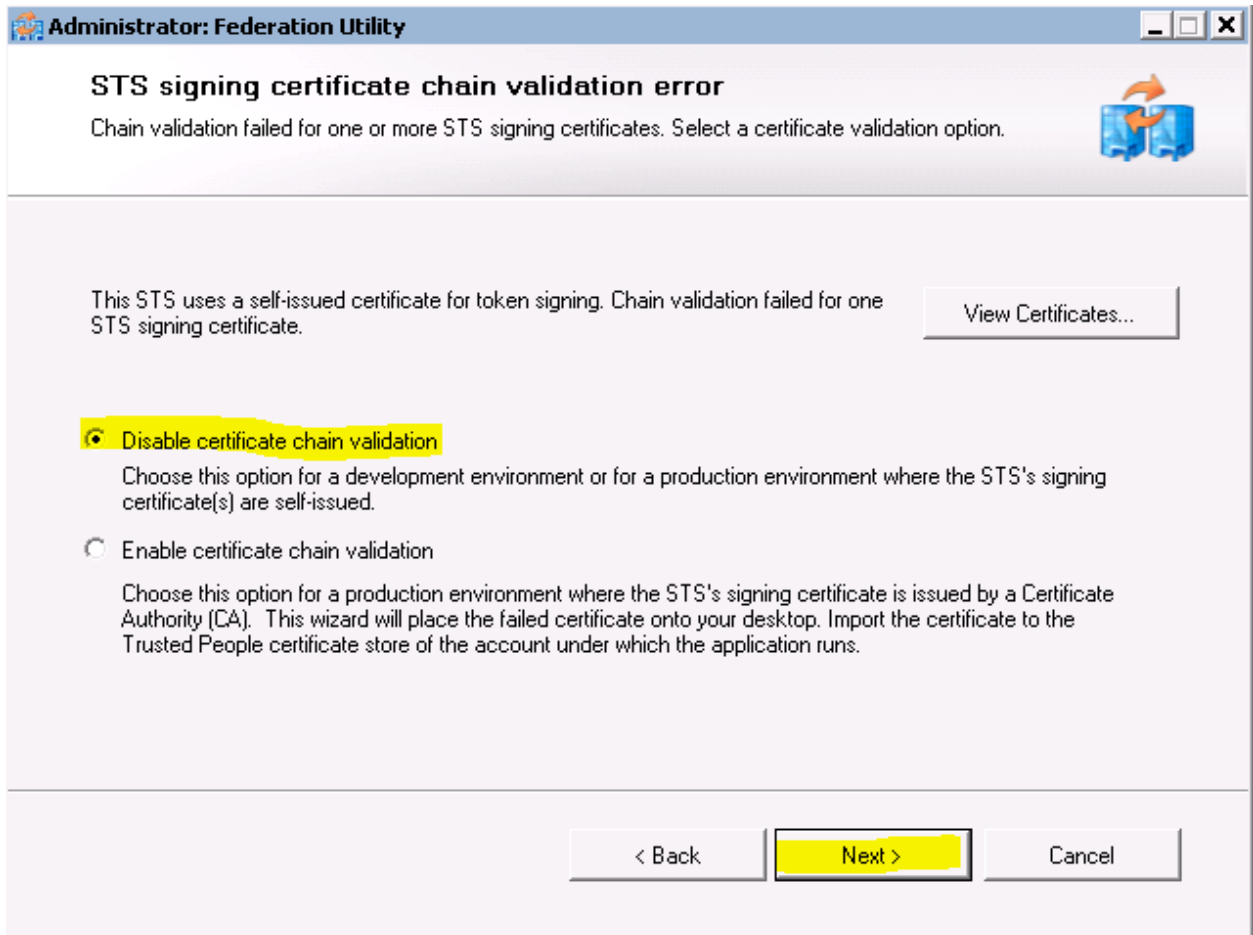
1. In the VS.NET 2010 open FederateLiveID solution, right-click on the <https://localhost/CustomSTS/> web application in the solution explorer and then select **Add STS Reference...**
2. This launches the Federation Utility wizard again, but this time we're going to federate the CustomSTS to our backend ACS instance.
3. Leave the default settings for Application Configuration location and Application Uri as fedutil.exe picks these details up from the existing web application and click Next



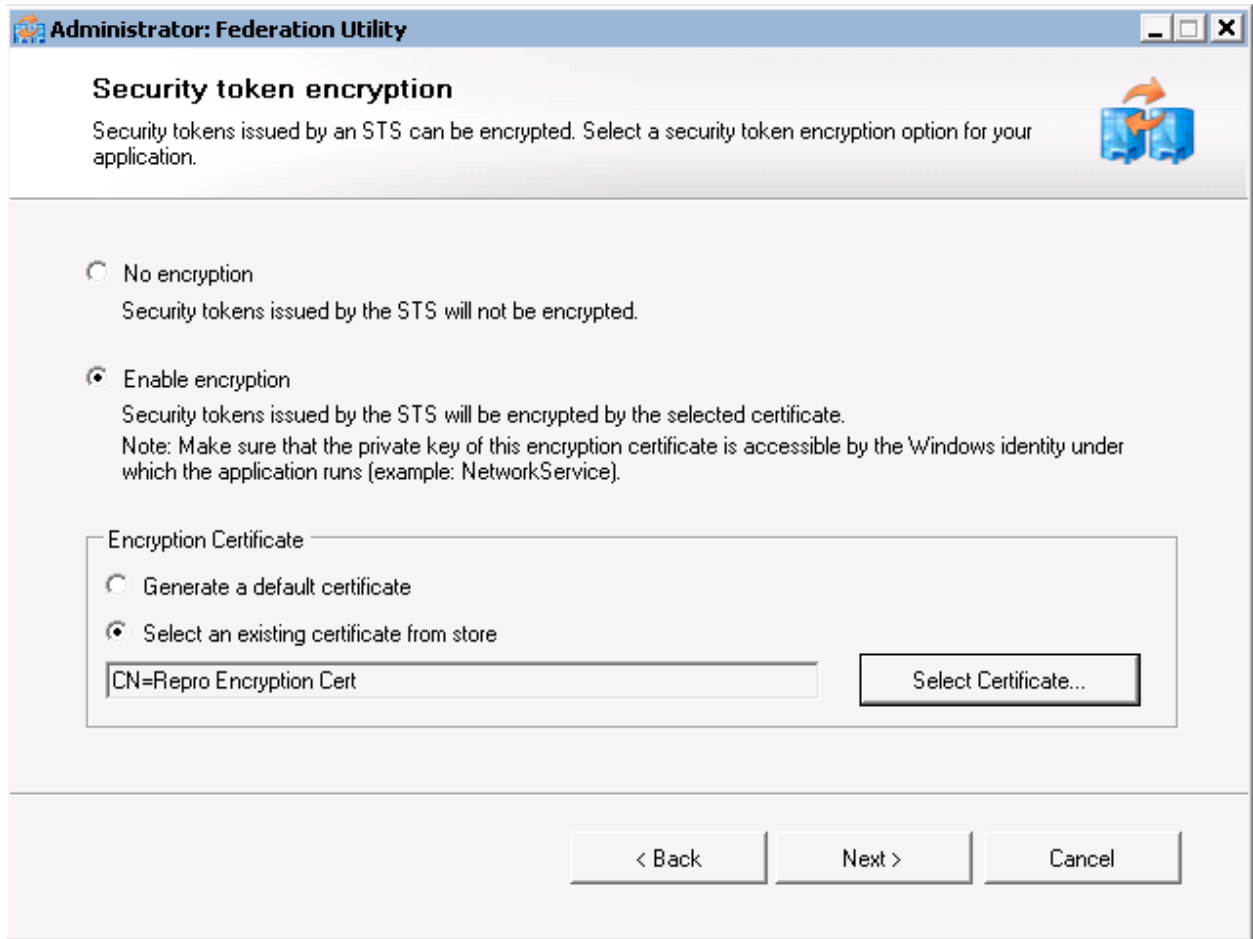
4. Select the Use an existing STS radio button and enter the full path to the Azure ACS WS-Federation Metadata link that we got from step 7-21 above. For my sample I entered <https://foust-repro-service-namespace.accesscontrol.windows.net/FederationMetadata/2007-06/FederationMetadata.xml>:



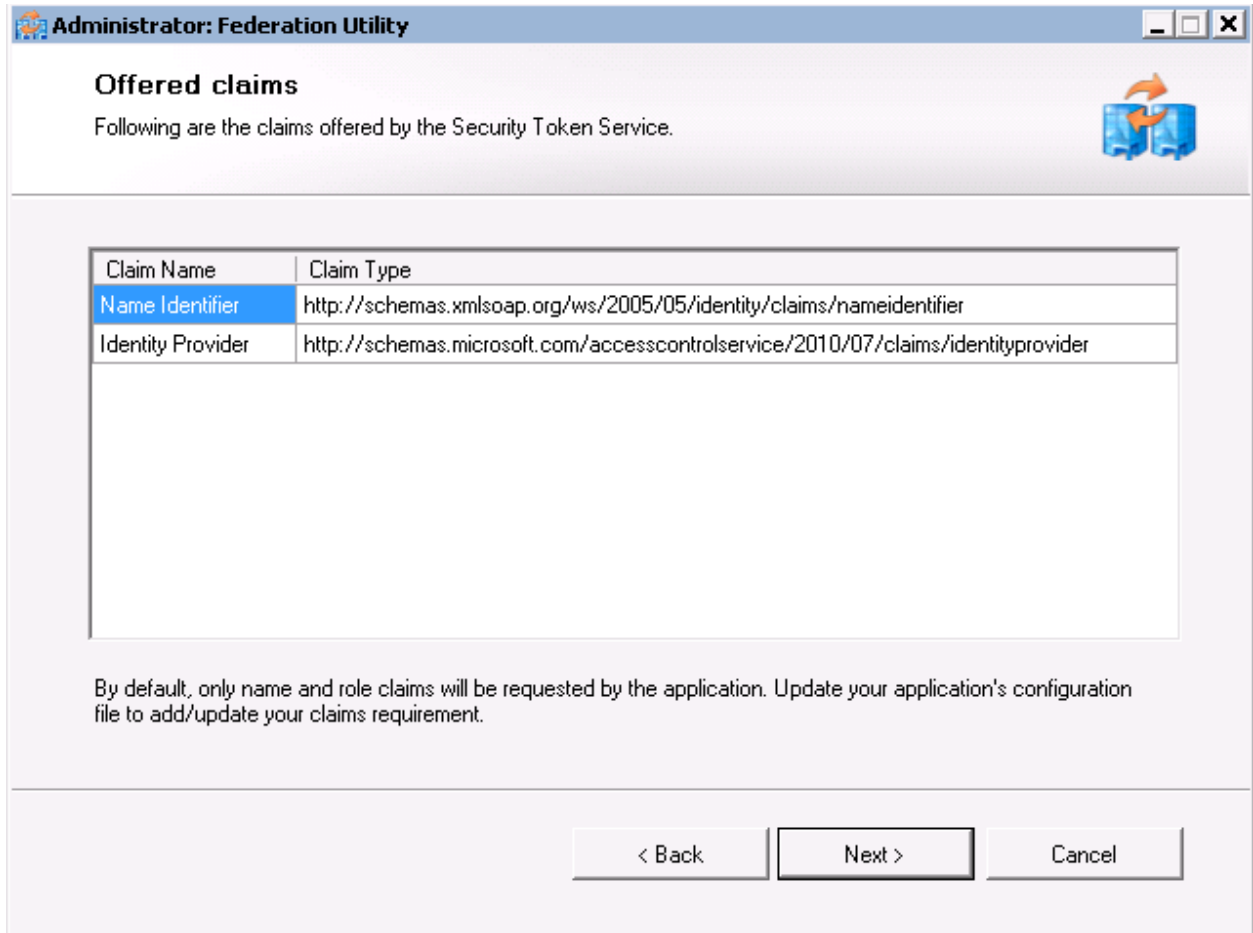
5. Click the Next button. This takes you to the STS signing certificate chain validation error page. The reason for this is that we have ACS configure to use its own self-signed cert for signing tokens. We don't have the certificate chain for that ACS cert on our local machine so we see this page. To avoid this we could have configured ACS to use our own Repro Signing Certificate certificate where we do have a certificate chain trust, however for this walkthrough we are simply going to select **Disable certificate chain validation** radio button and click **Next**:



- Now we see the Security token encryption page. We have already configured Azure ACS to encrypt the tokens that it issues with our Repro Encryption Certificate so we will need to select **Enable encryption** radio button here and then select that same Repro Encryption Certificate from the LocalMachine -> Personal store:



7. Click Next and you will now see the list of claims offered by our Windows Azure ACS relying party registration to our CustomSTS "client". There are only two claims returned: the first one is the nameidentifier that we get from Windows Live ID identity provider for the authenticated user. The second claims is the identity provider claim added by ACS to this claimset and so these two claims will be returned to our CustomSTS for processing:



8. Click **Next** to see the Summary page. Then click **Finish** to complete the Federation Utility wizard.
9. Based on all these configuration changes and FedUtil.exe steps, when browsing to our /ClaimsAwareASPX/default.aspx page, WIF will redirect the user to our CustomSTS for authentication. The CustomSTS is now configured to redirect the user to Windows Azure ACS for authentication. And Windows Azure ACS is configured to redirect the user to Windows Live ID for authentication, whereupon the user will be challenged for credentials to login and then the tokens will be returned all the way back to the original /ClaimsAwareASPX/default.aspx page for processing.

Step 9 - Update CustomSTS to pass through and add outgoing claims

The CustomSTS is configured to issue a ClaimTypes.Name and ClaimTypes.Role claims however we are only getting the claims ClaimTypes.NameIdentifier and a custom claim type called IdentityProvider from Windows Azure ACS. In this step we will modify the CustomSTS passive endpoint to pass through the claims returned from ACS but then also add a name and role claim required by the front end /ClaimsAwareASPX/ relying party application. I'm also going to add two additional custom claims that help us identify that the claims came from this passive endpoint:

1. In VS.Net 2010, in the solution explorer, expand the https://localhost/CustomSTS web application and then expand the App_Code folder

2. Double-click to open the CustomSecurityTokenService.cs code file
3. Find the method GetOutputClaimsIdentity and modify the existing code to the following. This code still copies all the claims returned from Azure ACS but then it adds the Name and Role claims, thus making this returned token contain four total claims.

```
protected override IClaimsIdentity GetOutputClaimsIdentity(IClaimsPrincipal
principal, RequestSecurityToken request, Scope scope)
{
    if (null == principal)
    {
        throw new ArgumentNullException("principal");
    }

    ClaimsIdentity outgoingIdentity = new ClaimsIdentity();
    IClaimsIdentity incomingIdentity = (IClaimsIdentity)principal.Identity;

    // Copy claims from incoming request which include ACS claims
    // to outgoing identity
    CopyClaims(incomingIdentity, outgoingIdentity);

    // Issue custom claims.
    // TODO: Change the claims below to issue custom claims required by your
    // application. Here you would add any business logic to lookup a
    // user based on their nameidentifier returned from ACS against a
    // database and then populate further claims for that user as needed.
    // Update the application's configuration file too to reflect
    // new claims requirement.

    // Hard coding the Name claim because I'm logging into Live.com using
    // my own personal ID. You'd want to replace this with any business logic
    // lookup to properly set the name claim of the user
    outgoingIdentity.Claims.Add(new
Claim(System.IdentityModel.Claims.ClaimTypes.Name, "Todd Foust"));
    outgoingIdentity.Claims.Add(new Claim(ClaimTypes.Role, "Administrator"));
    outgoingIdentity.Claims.Add(new
Claim("http://localhost/CustomSts/PassiveEndpoint/ROTYClaim", "Cam Newton"));
    outgoingIdentity.Claims.Add(new
Claim("http://localhost/CustomSts/PassiveEndpoint/SuperbowlWinnerClaim", "Giants"));

    return outgoingIdentity;
}
```

4. Add the CopyClaims method implementation to this same CustomSecurityTokenService.cs code file. This method just copies the incoming claims from the identity provider (Azure ACS/Windows LiveID) to the outgoing token of our CustomSTS:

```
/// <summary>
/// Do a deep-copy of IClaimsIdentity except the issuer.
/// </summary>
/// <param name="srcIdentity">Source Identity.</param>
/// <param name="dstIdentity">Destination Identity.</param>
private void CopyClaims(IClaimsIdentity srcIdentity, IClaimsIdentity dstIdentity)
{
```

```

foreach (Claim claim in srcIdentity.Claims)
{
    // We don't copy the issuer because it is not needed in this case.
    // The STS always issues claims using its own identity.
    Claim newClaim = new Claim(claim.ClaimType, claim.Value, claim.ValueType);

    // copy all claim properties
    foreach (string key in claim.Properties.Keys)
    {
        newClaim.Properties.Add(key, claim.Properties[key]);
    }

    // add claim to the destination identity
    dstIdentity.Claims.Add(newClaim);
}

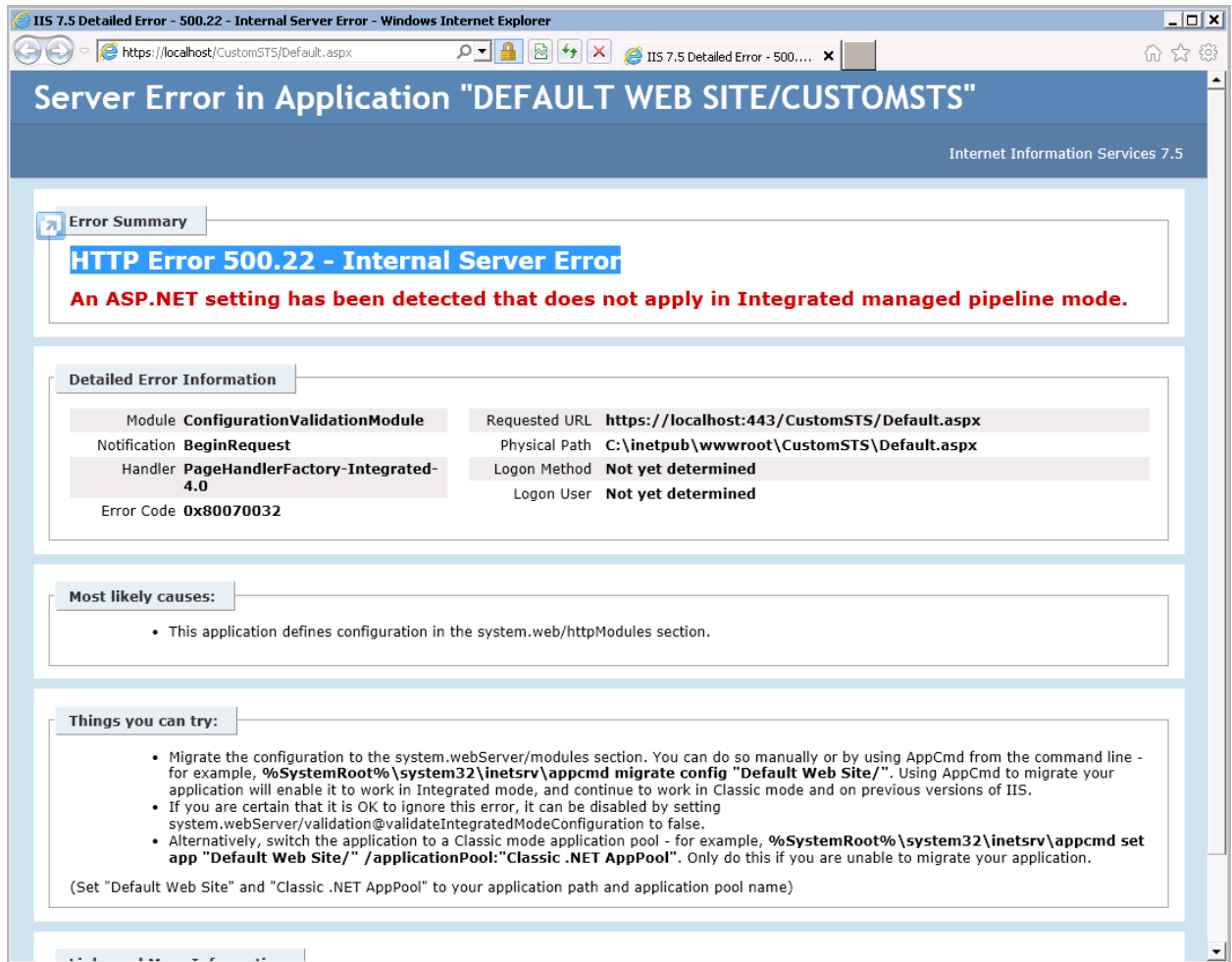
// Recursively copy claims from the source identity delegates
if (srcIdentity.Actor != null)
{
    dstIdentity.Actor = new ClaimsIdentity();
    CopyClaims(srcIdentity.Actor, dstIdentity.Actor);
}
}

```

- Build the entire solution. Build Menu -> Build Solution.
- Attempt to browse to the <https://localhost/ClaimsAwareASPX/default.aspx> page and watch as the browser redirects through your CustomSTS to Azure, to Live and back until the page is displayed showing you the following similar output. This output displays the two claims added by Windows LiveID and Windows Azure ACS as well as the two claims (name, role) added by our CustomSTS passive endpoint implementation:

Claim Type	Claim Value
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	IU [REDACTED]
http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider	uri:WindowsLiveID
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	Todd Foust
http://schemas.microsoft.com/ws/2008/06/identity/claims/role	Administrator
http://localhost/CustomSts/PassiveEndpoint/ROTYClaim	Cam Newton
http://localhost/CustomSts/PassiveEndpoint/SuperbowlWinnerClaim	Giants

- NOTE: If you get the following error when the browser tries to navigate to your CustomSTS, or if you open a browser and attempt to browse to <https://localhost/CustomSTS/Default.aspx> and see the following error...



This error indicates a conflict of hosting the CustomSTS in an Integrated Pipeline mode of the application pool and the existence of the `<system.web><httpModules/>` section. To get past this error you can take one of two actions:

- a. Comment out the `<system.web><httpModules/>` section of your web.config file. This section is read by IIS 6 and earlier versions. For IIS 7 and later we read in these settings from `<system.webServer><modules/>` section instead
- b. Leave the `<system.web><httpModules/>` section in place but then add the following `<validation>` element to the `<system.webServer>` section to bypass this check while in integration pipeline mode:

```
<system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <modules>
    <add name="WSFederationAuthenticationModule" ...../>
    <add name="SessionAuthenticationModule" ..... />
  </modules>
</system.webServer>
```


Phase Summary

During this phase of the walkthrough we created a claims aware web front end application that requires issued tokens from a CustomSTS. We added the CustomSTS security token service and configure it to federate with Windows Azure ACS. Within Azure ACS we configured a relying party application and configured that registration to use Windows Live ID to be the identity provider. We modified the CustomSTS implementation to pass through the claims returned from ACS and then also add the Name and Role claims in the issued token that the client sends to the claims aware front end web application.

Our next goal is to find a way to have our ASPX page call a claims aware WCF service. The call will 'delegate' the original security token used to access the ASPX page to the backend claims aware WCF service. The WCF relying party application will use the same CustomSTS as its security token service however this time the STS will be the identity provider, we won't be going back to Azure ACS on the call to the claims aware WCF service. This isn't possible in any case because Windows Live ID doesn't expose an active WS-Trust endpoint for authentication.

Phase 2 – Adding Active Endpoint to the CustomSTS Security Token Service

In Phase 1 we walked through the steps to support passive clients against our CustomSTS and then federate those calls against Windows Azure Access Control Service and Windows Live ID. That was the easy part. Over the remaining phases of this walk through we add an Active Endpoint to the CustomSTS, Add a claims aware WCF service, then delegate the token to the backend WCF service for additional claims processing. Identity delegation in this design makes use of a term called "ActAs token" which refers to a token that is issued by a STS and contains the user's identity (claims). The Actor property contains the STS's identity.

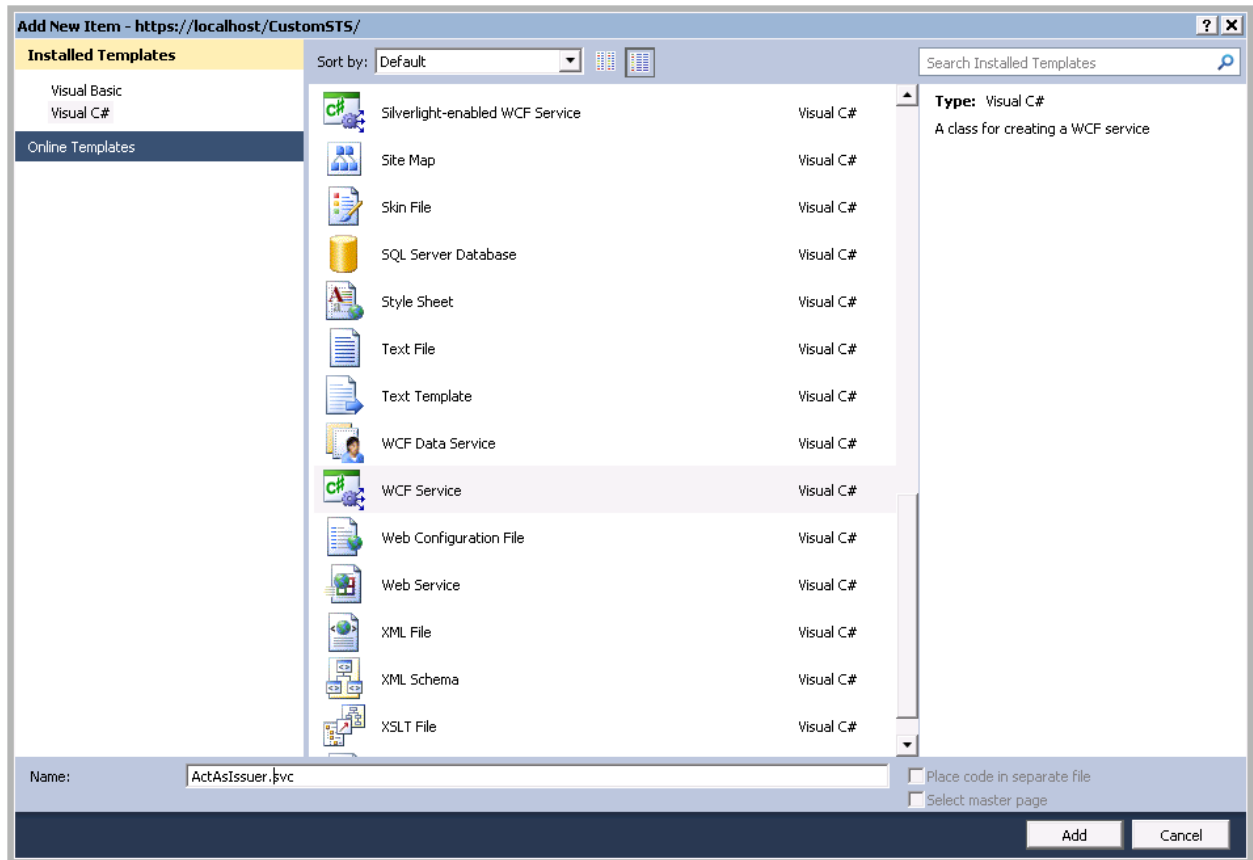
In my discussions with Vittorio Bertocci about this scenario he stated *"The ActAs (token) is probably the most complex thing you can do with WIF, building it bottom-up would be really hard."* In retrospect he is correct but I will try to walk you through the steps below. The main challenge is that we cannot depend on all the tooling support for configuring this scenario, so we have to do a lot of the work by hand.

I'll try to step through each piece of the remaining design architecture to hopefully simplify this process and help shine some light into how WIF processes these identity delegation scenarios. First we'll focus on adding the WS-Trust Active endpoint to our CustomSTS which will be responsible for getting the creating the delegated token and then issuing the token back to the client before the client sends the token on to the claims aware WCF service. We will programmatically add the WIF enabled Active endpoint.

Step 1 – Add ActAsIssuer.svc WCF service endpoint which will become our WS-Trust Active endpoint

1. In our open Visual Studio .NET 2010 FederateLiveID solution right-click on the <https://localhost/CustomSTS/> project and select **Add New Item...**

2. Select WCF Service in the list of templates and give it the name ActAsIssuer.svc then click the Add button to add this to our CustomSTS project



Step 2 – Implement the Active Endpoint

The ActAsIssuer.svc that we just added is a normal WCF service, with a code behind and interface code file. We are going to take the following steps to make this a WIF-based WS-Trust Active Endpoint. Most of the classes and code here was taken from the WIF 4.0 SDK sample “Web Sites And Identity – Exercise 4 - Invoking a WCF Service on the Backend via Delegated Access”. You can download this sample from the WIF 4.0 SDK: <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=4451>

1. In the CustomSTS project, expand the App_Code folder and delete both the **ActAsIssuer.cs** and **IActAsIssuer.cs** code files. These are the default WCF template code files that we’ll replace in further steps below.
2. Double-click to open the ActAsIssuer.svc file
3. Replace the current declarations from this...

```
<%@ ServiceHost Language="C#" Debug="true" Service="ActAsIssuer"
CodeBehind="~/App_Code/ActAsIssuer.cs" %>
```

To This...

```
<%@ ServiceHost Language="C#" Factory="ActAsSecurityTokenServiceFactory"
```

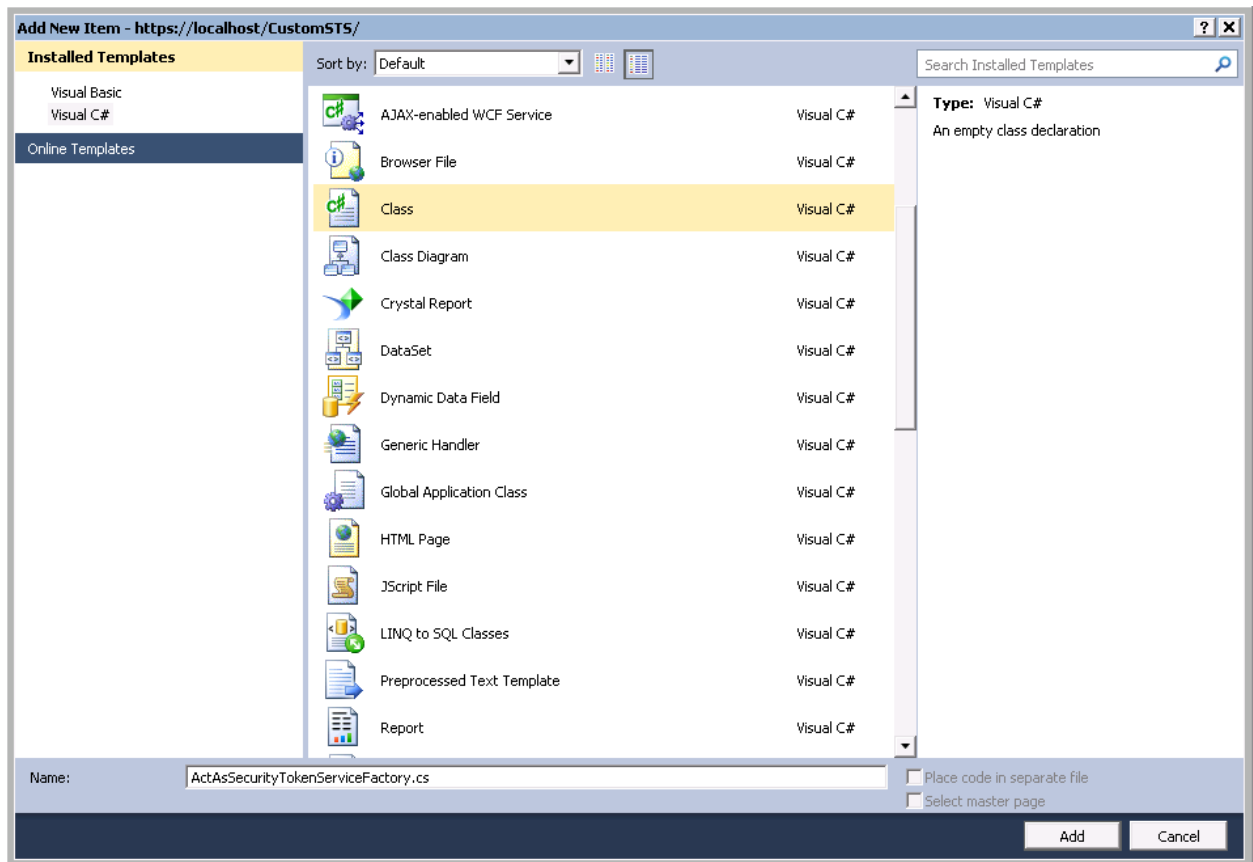
```
Service="ActAsCustomSecurityTokenServiceConfiguration" %>
```

Instead of the implicit default WCF host factory we are explicitly specifying the factory and service configuration classes. Next we will add those class files one at a time.

4. Add the service config information to the web.config file for this Active endpoint. Add or modify the following <system.serviceModel> section of the CustomSTS web.config file:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="ActAsStsBehavior">
        <serviceMetadata httpsGetEnabled="true" httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="true" />
        <useRequestHeadersForMetadataAddress/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <!--<serviceHostingEnvironment multipleSiteBindingsEnabled="true" />-->
  <services>
    <service name="Microsoft.IdentityModel.Protocols.WSTrust.WSTrustServiceContract"
behaviorConfiguration="ActAsStsBehavior">
      <!--
        This is the HTTPS endpoint that supports IMetadataExchange.
      -->
      <endpoint address="mex" binding="mexHttpsBinding" contract="IMetadataExchange"/>
    </service>
  </services>
</system.serviceModel>
```

5. Right-click on the App_Code folder of the https://localhost/CustomSTS/ project in the solution explorer and select **Add New Item....** Select a class file and name this class ActAsSecurityTokenServiceFactory.cs then click the Add button



- Replace the entire contents of the ActAsSecurityTokenServiceFactory.cs code file with the following code:

```
// -----
// Microsoft Developer & Platform Evangelism
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//
// THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
// EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES
// OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
// -----
// The example companies, organizations, products, domain names,
// e-mail addresses, logos, people, places, and events depicted
// herein are fictitious. No association with any real company,
// organization, product, domain name, email address, logo, person,
// places, or events is intended or should be inferred.
// -----

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.IdentityModel.Protocols.WSTrust;
using System.ServiceModel;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Selectors;
```

```

using Microsoft.IdentityModel.Tokens.Saml11;
using Microsoft.IdentityModel.Configuration;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.Web.Configuration;

/// <summary>
/// Summary description for ActAsSecurityTokenServiceFactory
/// </summary>
public class ActAsSecurityTokenServiceFactory : WSTrustServiceHostFactory
{
    public ActAsSecurityTokenServiceFactory()
    {
    }

    public override ServiceHostBase CreateServiceHost(string constructorString, Uri[]
baseAddresses)
    {
        SecurityTokenServiceConfiguration config = new
SecurityTokenServiceConfiguration(WebConfigurationManager.AppSettings[Common.ActiveIssuer
Name]);

        //Uri baseUri = baseAddresses.FirstOrDefault(a => a.Scheme == "http");
        //if (baseUri == null)
        //    throw new InvalidOperationException("The STS should be hosted under http");

        // Hard Coded value here so that it uses localhost instead of my FQDN domain
machine name for this sample
        Uri baseUri = new Uri("https://localhost/CustomSTS/ActAsIssuer.svc");
        // Set cert validation mode to none here since we're using simple localhost cert
config.CertificateValidationMode =
System.ServiceModel.Security.X509CertificateValidationMode.None;
        config.RevocationMode =
System.Security.Cryptography.X509Certificates.X509RevocationMode.NoCheck;

        config.TrustEndpoints.Add(new
ServiceHostEndpointConfiguration(typeof(IWSTrust13SyncContract),
GetWindowsCredentialsBinding(), baseUri.AbsoluteUri));

        // Set the STS implementation class type
config.SecurityTokenService = typeof(ActAsCustomSecurityTokenService);

        // Create a security token handler collection and then provide with a SAML11
security token
        // handler and set the Audience restriction to Never
        SecurityTokenHandlerCollection actAsHandlers = new
SecurityTokenHandlerCollection();
        Saml11SecurityTokenHandler actAsTokenHandler = new Saml11SecurityTokenHandler();
        // This is the token handler that will process on the ActAs token

        actAsHandlers.Add(actAsTokenHandler);
        actAsHandlers.Configuration.AudienceRestriction.AudienceMode =
AudienceUriMode.Never; // Here we ignore audience URI checks

        //Set the appropriate issuer name registry
        actAsHandlers.Configuration.IssuerNameRegistry = new ActAsIssuerNameRegistry();
        // we ignored audience URI but we still make sure the token was signed by trusted issuer
(our Passive STS endpoint)

```

```

        // Set the token handlers collection
    config.SecurityTokenHandlerCollectionManager[SecurityTokenHandlerCollectionManager.Usage.
    ActAs] = actAsHandlers;

    WSTrustServiceHost host = new WSTrustServiceHost(config, baseAddresses);
    return host;
}

static Binding GetWindowsCredentialsBinding()
{
    WS2007HttpBinding binding = new WS2007HttpBinding();
    binding.Security.Message.ClientCredentialType = MessageCredentialType.Windows;
    binding.Security.Message.EstablishSecurityContext = false;
    binding.Security.Transport.ClientCredentialType = HttpClientCredentialType.None;
    binding.Security.Mode = SecurityMode.TransportWithMessageCredential;
    binding.HostNameComparisonMode = HostNameComparisonMode.StrongWildcard;

    return binding;
}
}

```

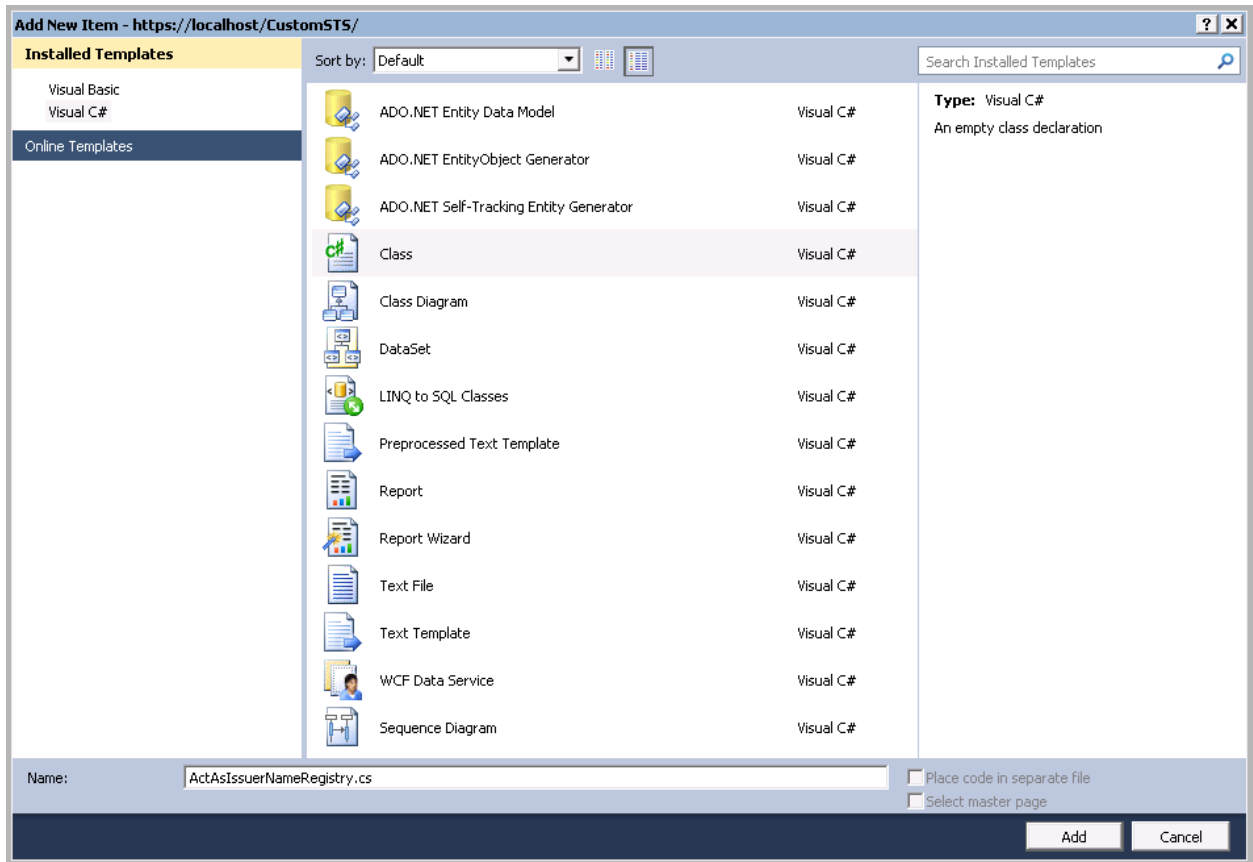
Let's talk through the code that we just added. We have added a custom implementation of the WIF provided [WSTrustServiceHostFactory](#) class which is responsible for providing instances of WSTrustServiceHost class in managed hosting environments.

We've provided an empty default constructor. We then overrode the default implementation of the CreateServiceHost method. This method basically programmatically sets the entire host configuration values that we would normally set in the <Microsoft.IdentityModel> section of the web.config file.

We expose an IWSTrust13SyncContract trust endpoint and pass in the binding instance that the ASPX Relying Party application will use to authenticate against this Active endpoint. If you review the GetWindowsCredentialsBinding() method you'll see that we just create a normal ws2007HttpBinding() and will use Windows client credential. Basically we will authenticate the ASPX RP app using the application pool identity once that call is authenticated then we proceed to process on the token that ASPX RP app passes in.

We add a Saml11SecurityTokenHandler to handle the SAML 1.1 token that ACS passed back to the ASXP RP app, then we add a custom IssuerNameRegistry class that is responsible for returning an issuer name for the incoming signing certificate. Lastly this method simply returns an instance of the WIF provided WSTrustServiceHost with all of these configurations applied.

7. Right-click on the App_Code folder of the https://localhost/CustomSTS/ project in the solution explorer and select **Add New Item...** Select a class file and name this class ActAsIssuerNameRegistry.cs then click the Add button. This is our custom IssuerNameRegistry, again pulled as a sample from the WIF 4.0 SDK Web Sites and Identity Exercise #4 sample.



8. Replace the default code in ActAsIssuerNameRegistry.cs with the following code:

```
// -----
// Microsoft Developer & Platform Evangelism
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//
// THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
// EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES
// OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
// -----
// The example companies, organizations, products, domain names,
// e-mail addresses, logos, people, places, and events depicted
// herein are fictitious. No association with any real company,
// organization, product, domain name, email address, logo, person,
// places, or events is intended or should be inferred.
// -----

using System;
using System.IdentityModel.Tokens;
using Microsoft.IdentityModel.Tokens;
using System.Web.Configuration;

/// <summary>
/// IssuerNameRegistry that validates the incoming token in RST.ActAs parameter.
/// </summary>
public class ActAsIssuerNameRegistry : IssuerNameRegistry
```

```

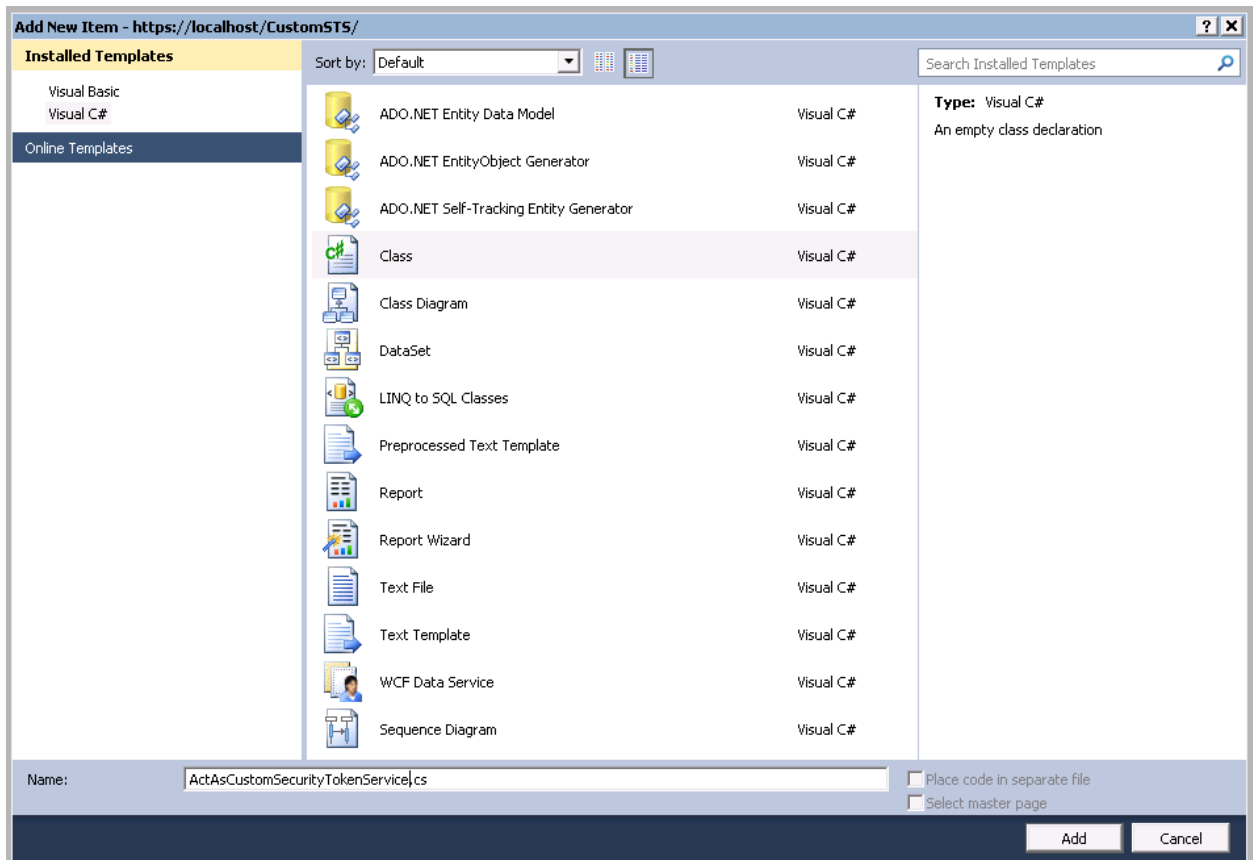
{
    /// <summary>
    /// Overrides the base class. Validates the given issuer token.
    /// For an incoming SAML token the issuer token is the
    /// Certificate that signed the SAML token.
    /// </summary>
    /// <param name="securityToken">Issuer token to be validated.</param>
    /// <returns>Friendly name representing the Issuer.</returns>
    public override string GetIssuerName(SecurityToken securityToken)
    {
        X509SecurityToken x509Token = securityToken as X509SecurityToken;
        if (x509Token != null)
        {
            // Warning: This sample does a simple compare of the Issuer Certificate
            // to a subject name. This is not appropriate for production use.
            // Check your validation policy and authenticate issuers based off the
policy.
            if (String.Equals(x509Token.Certificate.SubjectName.Name,
WebConfigurationManager.AppSettings[Common.SigningCertificateName]))
            {
                return x509Token.Certificate.SubjectName.Name;
            }
        }

        throw new SecurityTokenException("Untrusted issuer.");
    }
}

```

This method just returns a friendly name that represents the issuer of tokens provided by this ws-trust active endpoint of our CustomSTS project. You can see that we just return the subject name of the signing certificate as this issuer name.

9. Right-click on the App_Code folder of the <https://localhost/CustomSTS/> project in the solution explorer and select **Add New Item....** Select a class file and name this class ActAsCustomSecurityTokenService.cs then click the Add button.



10. Replace the default code in the ActAsCustomSecurityTokenService.cs with the following code:

```
// -----
// Microsoft Developer & Platform Evangelism
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//
// THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
// EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES
// OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
// -----
// The example companies, organizations, products, domain names,
// e-mail addresses, logos, people, places, and events depicted
// herein are fictitious. No association with any real company,
// organization, product, domain name, email address, logo, person,
// places, or events is intended or should be inferred.
// -----
```

```
using System.Linq;
using System.Security.Cryptography.X509Certificates;
using Microsoft.IdentityModel.Claims;
using Microsoft.IdentityModel.Configuration;
using Microsoft.IdentityModel.SecurityTokenService;
using Microsoft.IdentityModel.Protocols.WSTrust;
using System.Web.Configuration;
```

```
/// <summary>
```

```

/// Implementation of a Custom SecurityTokenService.
/// </summary>
public class ActAsCustomSecurityTokenService : SecurityTokenService
{
    static readonly string SigningCertificateName =
WebConfigurationManager.AppSettings[Common.SigningCertificateName];
    static readonly string EncryptingCertificateName =
WebConfigurationManager.AppSettings[Common.EncryptingCertificateName];

    /// <summary>
    /// Creates an instance of ActAsCustomSecurityTokenService.
    /// </summary>
    /// <param name="configuration">Configuration for this SecurityTokenService.</param>
    public ActAsCustomSecurityTokenService(SecurityTokenServiceConfiguration
configuration)
        : base(configuration)
    {
        // Setup our certificate the STS is going to use to sign the issued tokens
        configuration.SigningCredentials = new
X509SigningCredentials(CertificateUtil.GetCertificate(StoreName.My,
StoreLocation.LocalMachine, SigningCertificateName));
    }

    /// <summary>
    /// This method returns the configuration for the token issuance request. The
configuration
    /// is represented by the Scope class. In our case, we are only capable to issue a
token for a
    /// single RP identity represented by CN=Repro Signing Cert.
    /// </summary>
    /// <param name="principal">The caller's principal</param>
    /// <param name="request">The incoming RST</param>
    /// <returns></returns>
    protected override Scope GetScope(IClaimsPrincipal principal, RequestSecurityToken
request)
    {
        // Create the scope using the request AppliesTo address and the STS signing
certificate
        Scope scope = new Scope(request.AppliesTo.Uri.ToString(),
SecurityTokenServiceConfiguration.SigningCredentials);

        // We only support a single RP identity represented by CN=localhost. Set the RP
certificate for encryption
        scope.EncryptingCredentials = new X509EncryptingCredentials(
CertificateUtil.GetCertificate(StoreName.My,
StoreLocation.LocalMachine,
EncryptingCertificateName));

        // Set the replyTo address. In WS-Federation passive case this value is used as
the endpoint
        // where the user is redirected to.
        scope.ReplyToAddress = scope.AppliesToAddress;

        return scope;
    }
}

```

```

    /// <summary>
    /// This method returns the content of the issued token. The content is represented
as a set of
    /// IClaimIdentity instances, each instance corresponds to a single issued token.
    /// </summary>
    /// <param name="scope">The scope that was previously returned by GetScope
method.</param>
    /// <param name="principal">The caller's principal.</param>
    /// <param name="request">The incoming RST.</param>
    /// <returns></returns>
    protected override IClaimsIdentity GetOutputClaimsIdentity(IClaimsPrincipal
principal, RequestSecurityToken request, Scope scope)
    {
        IClaimsIdentity callerIdentity = (IClaimsIdentity)principal.Identity;
        ClaimsIdentityCollection outputClaimsCollection = new ClaimsIdentityCollection();

        // Create new identity and copy content of the caller's identity into it
(including the existing delegate chain)
        IClaimsIdentity outputIdentity = new ClaimsIdentity();
        CopyClaims(callerIdentity, outputIdentity);

        // If there is an ActAs token in the RST, add and return the claims from it as
the top-most identity
        // and put the caller's identity into the Delegate property of this identity.
        if (request.ActAs != null)
        {
            IClaimsIdentity actAsIdentity = new ClaimsIdentity();
            CopyClaims(request.ActAs.GetSubject()[0], actAsIdentity);

            // Find the last delegate in the actAs identity
            IClaimsIdentity lastActingVia = actAsIdentity;
            while (lastActingVia.Actor != null)
            {
                lastActingVia = lastActingVia.Actor;
            }

            // Put the caller's identity as the last delegate to the ActAs identity
            lastActingVia.Actor = outputIdentity;

            // Return the actAsIdentity instead of the caller's identity in this case
            outputIdentity = actAsIdentity;
        }

        // Add two more claims specific to this WS-Trust Active endpoint
        // that the claims aware ASPX web app can process on.
        // You could insert additional business logic to process on the incoming claims
        // to determine which additional claims you want to add to the token
        outputIdentity.Claims.Add(new
Claim("http://localhost/CustomSts/ActiveEndpoint/EmailClaim",
"mybogusemail@hotmail.com"));
        outputIdentity.Claims.Add(new
Claim("http://localhost/CustomSts/ActiveEndpoint/CustomClaim", "any value"));

        return outputIdentity;
    }

    /// <summary>
    /// Do a deep-copy of IClaimsIdentity except the issuer.

```

```

/// </summary>
/// <param name="srcIdentity">Source Identity.</param>
/// <param name="dstIdentity">Destination Identity.</param>
private void CopyClaims(IClaimsIdentity srcIdentity, IClaimsIdentity dstIdentity)
{
    foreach (Claim claim in srcIdentity.Claims)
    {
        // We don't copy the issuer because it is not needed in this case. The STS
always issues claims
        // using its own identity.
        Claim newClaim = new Claim(claim.ClaimType, claim.Value, claim.ValueType);

        // copy all claim properties
        foreach (string key in claim.Properties.Keys)
        {
            newClaim.Properties.Add(key, claim.Properties[key]);
        }

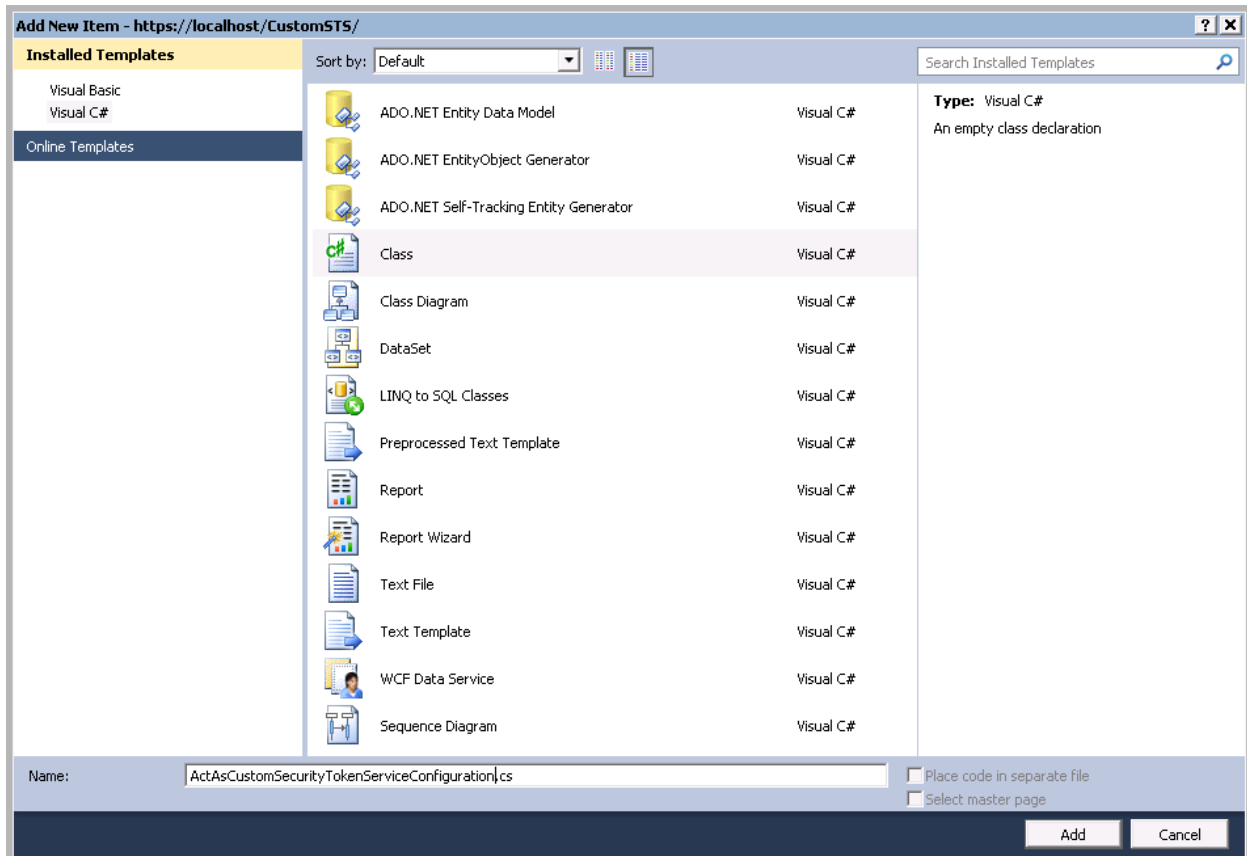
        // add claim to the destination identity
        dstIdentity.Claims.Add(newClaim);
    }

    // Recursively copy claims from the source identity delegates
    if (srcIdentity.Actor != null)
    {
        dstIdentity.Actor = new ClaimsIdentity();
        CopyClaims(srcIdentity.Actor, dstIdentity.Actor);
    }
}
}

```

You will see that this ActAsCustomSecurityTokenService class is very similar to our WS-Federation Passive endpoint implementation. The only difference here is that our Active endpoint adds two more claims to the token that are specific to this CustomSTS Active Endpoint.

- There is one more class that we need to add to our CustomSTS App_Code folder. If you look back at the ActAsIssuer.svc file you see we have configured the Service with the value: Service="ActAsCustomSecurityTokenServiceConfiguration". We need to add that class implementation as well. Right-click on the App_Code folder of the <https://localhost/CustomSTS/> project in the solution explorer and select **Add New Item....** Select a class file and name this class ActAsCustomSecurityTokenServiceConfiguration.cs then click the Add button.



12. Replace the default code found in `ActAsCustomSecurityTokenServiceConfiguration.cs` with the following contents. You can see that this class just lines up our `ActAsCustomSecurityTokenService` as the class to be loaded when processing incoming requests to the endpoint.

```
// -----
// Microsoft Developer & Platform Evangelism
//
// Copyright (c) Microsoft Corporation. All rights reserved.
//
// THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
// EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES
// OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
// -----
// The example companies, organizations, products, domain names,
// e-mail addresses, logos, people, places, and events depicted
// herein are fictitious. No association with any real company,
// organization, product, domain name, email address, logo, person,
// places, or events is intended or should be inferred.
// -----
using Microsoft.IdentityModel.Configuration;

/// <summary>
/// Summary description for CustomSecurityTokenServiceConfiguration
/// </summary>
class ActAsCustomSecurityTokenServiceConfiguration : SecurityTokenServiceConfiguration
{
```

```

/// <summary>
/// Creates an instance of CustomBookStoreSecurityTokenServiceConfiguration.
/// </summary>
public ActAsCustomSecurityTokenServiceConfiguration()
{
    SecurityTokenService = typeof(ActAsCustomSecurityTokenService);
}
}

```

- Next open the Common.cs code file in the CustomSTS App_Code folder and define the Common class as follows. This class is used to grab settings out of the web.config file used to configure our CustomSTS.

```

public static class Common
{
    public const string PassiveIssuerName = "PassiveIssuerName";
    public const string ActiveIssuerName = "ActiveIssuerName";
    public const string SigningCertificateName = "SigningCertificateName";
    public const string EncryptingCertificateName = "EncryptingCertificateName";
}

```

- We need to make sure that the Passive Security Token Service uses an IssuerName of PassiveIssuerName application setting. In the CustomSTS App_Code folder open the CustomSecurityTokenServiceConfiguration.cs code file and in the class constructor make sure we load the Common.PassiveIssuerName:

```

/// <summary>
/// CustomSecurityTokenServiceConfiguration constructor.
/// </summary>
public CustomSecurityTokenServiceConfiguration(
    : base( WebConfigurationManager.AppSettings[Common.PassiveIssuerName],
          new X509SigningCredentials( CertificateUtil.GetCertificate(
              StoreName.My, StoreLocation.LocalMachine,
              WebConfigurationManager.AppSettings[Common.SigningCertificateName] )
    ) )
{
    this.SecurityTokenService = typeof( CustomSecurityTokenService );
}

```

- Build the solution and make sure it compiles without any errors

Step 3 – Update IssuerName application settings

We have two token issuer endpoints in our CustomSTS web application. We need to add application settings to the web.config to specify the issuer name for both of those endpoints.

- In the https://localhost/CustomSTS/ web application open up the web.config file
- At the top of this config file change the appSettings to look like this:
NOTE: Your implementation will have a different FederationMetadataLocation value to your own custom Azure ACS namespace

```

<appSettings>
  <add key="PassiveIssuerName" value="PassiveSigninSTS" />
  <add key="ActiveIssuerName" value="ActiveSigninSTS" />
  <add key="SigningCertificateName" value="CN=Repro Signing Cert" />
  <add key="EncryptingCertificateName" value="CN=Repro Encryption Cert" />
  <add key="FederationMetadataLocation" value="https://foust-repro-service-
namespace.accesscontrol.windows.net/FederationMetadata/2007-06/FederationMetadata.xml "
/>
</appSettings>

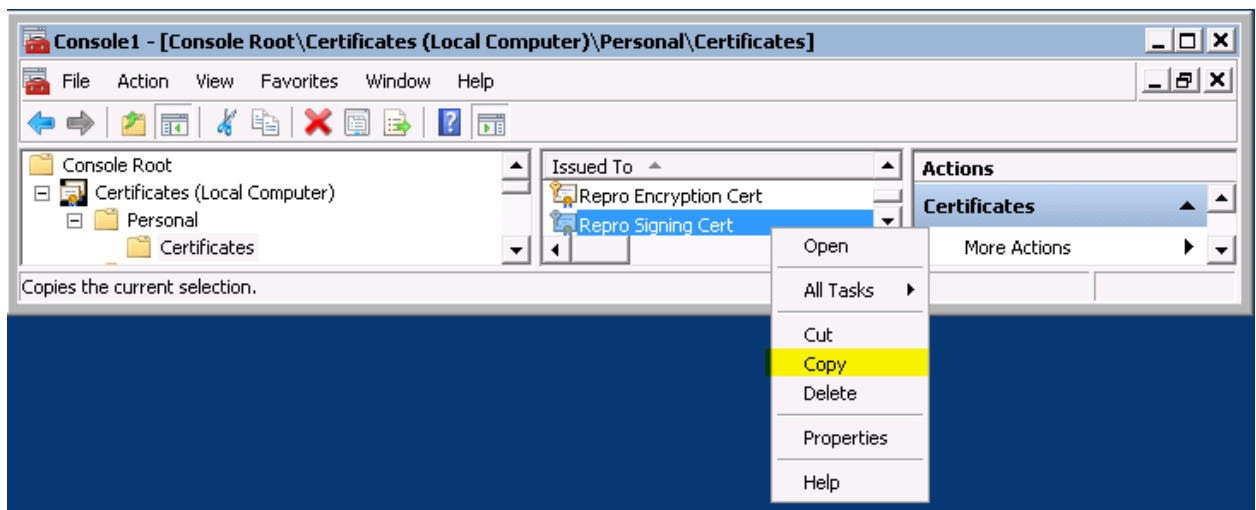
```

You can see here that we specify the IssuerName string for the Passive and Active STS endpoints as well as properly set the signing and encryption certificate used by the CustomSTS.

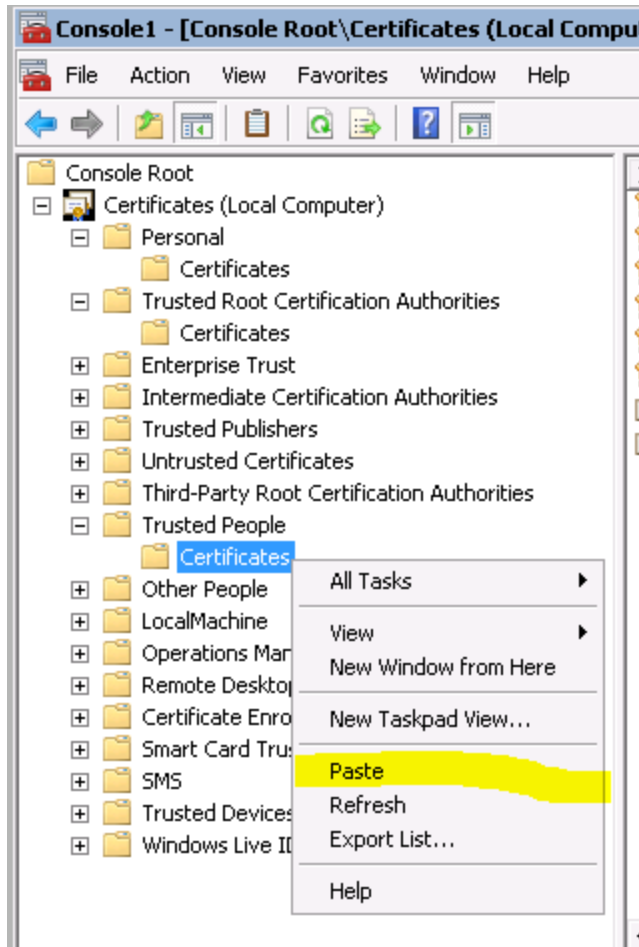
Step 4 – Copy signing certificate to Trusted People store

The Active endpoint of our CustomSTS uses the default Saml11SecurityTokenHandler class to process incoming SAML 1.1 tokens. This class defaults to a CertificateValidationMode of PeerOrChainTrust. If you want to change this to None or disable the revocation setting you will have to create a custom certificate validator class and set that for the Saml11SecurityTokenHandler class in the ActAsSecurityTokenServiceFactory.CreateServiceHost() method. See example validator class [here](#). For the walkthrough I'll just accept the default values, PeerOrChainTrust, which requires that we copy the **CN=Repro Signing Cert** into the LocalMachine -> Trusted People store.

1. Open up the certificate management console
2. Find the Repro Signing Cert in the Local Computer -> Personal store
3. Right-click on that certificate and select Copy:



4. Then navigate to Local Computer -> Trusted People store
5. Right-click on the Certificates folder and select Paste to copy this certificate from the Personal store to the Trusted People store:



Phase Summary

At this point we have a CustomSTS that exposes a WS-Federation passive endpoint, which uses a series of redirects to ACS and Live ID to authenticate the user and issue a token. The CustomSTS now also exposes a WS-Trust active endpoint (at ActAsIssuer.svc). The passive endpoint is mostly configured through the web.config file however the active endpoint is completely implemented through code using the ActAsSecurityTokenServiceFactory, ActAsCustomSecurityTokenService and other supporting classes.

The key take away from this phase is that we have two different WSTrustServiceHostFactory objects to respond to the Passive and then the Active endpoint. Both endpoints end up adding individual claims that will be processed by the front end ASPX relying party application. The CustomSTS web app is now complete at this point. We still have to build the claims aware backend WCF Service and then configure the client ASPX relying party app to call to that claims aware WCF RP service.

Phase 3 – Building Claims Aware WCF service which receives the ActAs token

In this phase we will build the backend Claims Aware WCF relying party application. This service will require issued tokens from the Active endpoint of our CustomSTS service. We will not use the Federation Utility wizard to configure WIF, but instead do it manually by simply making changes to the

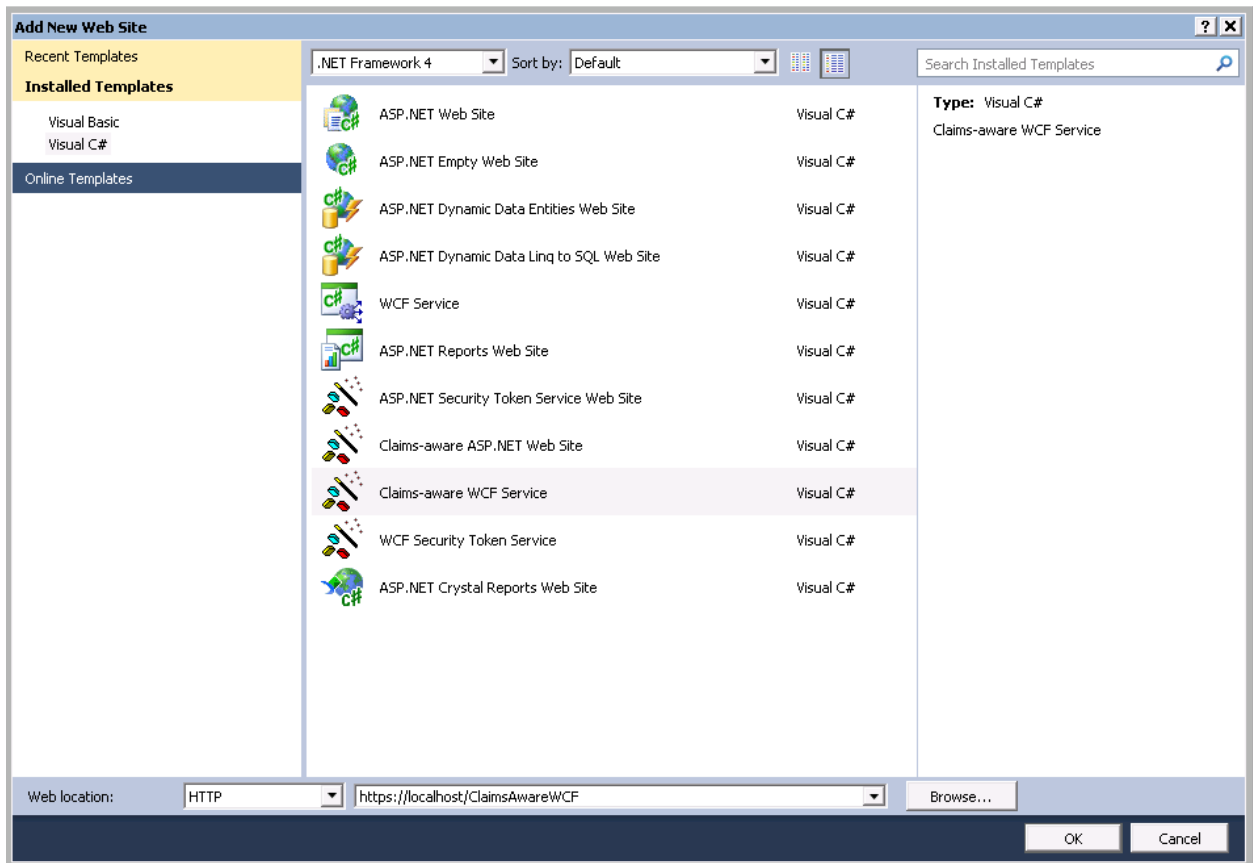
web.config file. In the config file we expose an endpoint for this WCF service that uses the IssuedTokenOverTransport authentication mode which basically tells the client to go authenticate against a specific issuer, and once they are authenticated they can call the WCF service over transport security.

This was another challenging piece of the entire application because we have to use customBindings instead of the normal ws2007FederationHttpBinding. When using customBindings the settings are pretty delicate and there is practically no documentation on what is required to support federation that still involves WIF.

To make up for the lack of documentation, in the web.config file I have commented 5 additional endpoints and their customBinding configurations required to expose this WCF RP app using all of the other IssuedToken* authentication types, some including secure conversations and others without. You can comment out the current endpoint, uncomment one of the other endpoints, then update the client service reference and the call should still work. You'll see this in the web.config file of our claims aware WCF application.

Step 1 – Add the Backend Claims Aware WCF Service to the Solution

1. In our open FederateLiveID solution, select the File menu -> Add -> New Web Site...
2. Select the Claims-aware WCF Service project template
3. Enter the address `https://localhost/ClaimsAwareWCF/` and click OK button to add this project to our solution:



4. Open up the IService.cs code behind file and replace the code there with the following. This code just exposes one operation called GetClaims that just enumerates the list of claims that this service received from the incoming call from our ASPX RP app:

```
//-----  
//  
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF  
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO  
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A  
// PARTICULAR PURPOSE.  
//  
// Copyright (c) Microsoft Corporation. All rights reserved.  
//  
//  
//-----  
  
using System.Runtime.Serialization;  
using System.ServiceModel;  
using System.Collections.Generic;  
  
namespace ClaimsAwareWCF  
{  
    // NOTE: If you change the interface name "IService" here,  
    // you must also update the reference to "IService" in Web.config.  
    [ServiceContract(Namespace = "urn:federateliveid:samples")]  
    public interface IService  
    {  
        [OperationContract]  
        List<ViewClaim> GetClaims();  
    }  
  
    // Use a data contract as illustrated in the sample below to  
    // add composite types to service operations.  
    [DataContract(Namespace = "urn:federateliveid:samples")]  
    public class ViewClaim  
    {  
        [DataMember]  
        public string ClaimType { get; set; }  
  
        [DataMember]  
        public string Value { get; set; }  
  
        [DataMember]  
        public string Issuer { get; set; }  
  
        [DataMember]  
        public string OriginalIssuer { get; set; }  
    }  
}
```

5. Open up the Service.cs code file in the App_Code folder and replace its entire contents with the following code which implements the interface and GetClaims method:

```
//-----  
//  
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF  
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO  
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A  
// PARTICULAR PURPOSE.  
//  
// Copyright (c) Microsoft Corporation. All rights reserved.  
//  
//  
//-----  
  
using System.Threading;  
using Microsoft.IdentityModel.Claims;  
using System.Linq;  
using System.Collections.Generic;  
  
namespace ClaimsAwareWCF  
{  
    // NOTE: If you change the class name "Service" here, you  
    // must also update the reference to "Service" in Web.config  
    // and in the associated .svc file.  
    public class Service : IService  
    {  
        public List<ViewClaim> GetClaims()  
        {  
            var id = Thread.CurrentPrincipal.Identity as IClaimsIdentity;  
  
            return (from c in id.Claims  
                    select new ViewClaim  
                    {  
                        ClaimType = c.ClaimType,  
                        Value = c.Value,  
                        Issuer = c.Issuer,  
                        OriginalIssuer = c.OriginalIssuer  
                    }).ToList();  
        }  
    }  
}
```

6. Go into IIS manager. Find the ClaimsAwareWCF service. Double-click on the Authentication feature and then configure the following authentication mode settings:

- Anonymous Authentication – Enabled
- ASP.NET Impersonation – Disabled
- Basic Authentication – Disabled
- Forms Authentication – Disabled
- Windows Authentication - Enabled

Step 2 – Populate the Claims Aware WCF RP app web.config file

At this point we need to configure this WCF service to use WIF and federate against our CustomSTS Active endpoint. This normally requires the step to add STS Reference against a FederationMetadata.xml file. The problem with this is that our CustomSTS FederationMetadata.xml file is currently signed to federate against ACS. In this case we don't want to go to ADFS. There may be a way to manually modify the FederationMetadata.xml file to expose a RoleDescriptor element with type `xsi:type="fed:SecurityTokenServiceType"`, and then a second RoleDescriptor element with type `xsi:type="fed:ApplicationServiceType"`, but I ran out of time to investigate if the FedUtility wizard can parse two RoleDescriptor elements and properly configure the RP app to the proper STS endpoint. Instead we will simply configure this WCF RP app to use WIF and our CustomSTS manually by modifying the web.config file.

1. Open up the web.config file for our `https://localhost/ClaimsAwareWCF/` project and replace the entire contents with the following configuration:

Backend claims aware WCF service web.config file

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Note: As an alternative to hand editing this file you can use the
    web admin tool to configure settings for your application. Use
    the Website->Asp.Net Configuration option in Visual Studio.
    A full list of settings and comments can be found in
    machine.config.comments usually located in
    \Windows\Microsoft.Net\Framework\v2.x\Config
-->
<configuration>
  <configSections>
    <section name="microsoft.identityModel"
type="Microsoft.IdentityModel.Configuration.MicrosoftIdentityModelSection,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <connectionStrings />
  <location path="FederationMetadata">
    <system.web>
      <authorization>
        <allow users="*" />
      </authorization>
    </system.web>
  </location>
  <system.web>
    <!--
      Set compilation debug="true" to insert debugging
      symbols into the compiled page. Because this
      affects performance, set this value to true only
      during development.
    -->
    <compilation debug="true" targetFramework="4.0">
      <assemblies>
```

```

        <add assembly="Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31BF3856AD364E35" />
    </assemblies>
</compilation>
<!--
    The <authentication> section enables configuration
    of the security authentication mode used by
    ASP.NET to identify an incoming user.
-->
<authentication mode="Windows" />
<!--
    The <customErrors> section enables configuration
    of what to do if/when an unhandled error occurs
    during the execution of a request. Specifically,
    it enables developers to configure html error pages
    to be displayed in place of an error stack trace.

    <customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
        <error statusCode="403" redirect="NoAccess.htm" />
        <error statusCode="404" redirect="FileNotFound.htm" />
    </customErrors>
-->
<pages>
    <controls>
        <add tagPrefix="asp" namespace="System.Web.UI" assembly="System.Web.Extensions,
Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
    </controls>
</pages>
    <identity impersonate="false" />

</system.web>
<system.web.extensions>
    <scripting>
        <webServices>
            <!--
                Uncomment this section to enable the authentication service. Include
                requireSSL="true" if appropriate.
            -->
            <!--
            <authenticationService enabled="true" requireSSL = "true|false"/>
            -->
            <!--
                Uncomment these lines to enable the profile service, and to choose the
                profile properties that can be retrieved and modified in ASP.NET AJAX
                applications.
            -->
            <!--
            <profileService enabled="true"
                readAccessProperties="propertyname1,propertyname2"
                writeAccessProperties="propertyname1,propertyname2" />
            -->
            <!--
                Uncomment this section to enable the role service.
            -->
            <!--
            <roleService enabled="true"/>
            -->
        </webServices>

```

```

        <!--
        <scriptResourceHandler enableCompression="true" enableCaching="true" />
        -->
    </scripting>
</system.web.extensions>
<microsoft.identityModel>
    <service name="ClaimsAwareWCF.Service">
        <audienceUri>
            <add value="https://localhost/ClaimsAwareWCF/Service.svc" />
        </audienceUri>
        <issuerNameRegistry
type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35">
            <trustedIssuers>
                <add thumbprint="70C883F888C017B3FB02B9887F7835794473C06F" name="CN=Repro
Signing Cert" />
            </trustedIssuers>
        </issuerNameRegistry>
        <serviceCertificate>
            <certificateReference x509FindType="FindByThumbprint"
findValue="7DD17B7807EDA96F1DDD687EB420A097294F0A77" storeLocation="LocalMachine"
storeName="My"/>
        </serviceCertificate>
        <certificateValidation certificateValidationMode="None" revocationMode="NoCheck" />
    </service>
</microsoft.identityModel>
<system.serviceModel>
    <services>
        <service name="ClaimsAwareWCF.Service"
behaviorConfiguration="ClaimsAwareWCF.ServiceBehavior">
            <!-- IssuedToken Endpoint: -->
            <!-- <endpoint address="http://localhost/ClaimsAwareWCF/Service.svc"
binding="customBinding" contract="ClaimsAwareWCF.IService"
bindingConfiguration="CustomBindingConfiguration_IssuedToken" />-->

            <!-- IssuedTokenOverTransport Endpoint: -->
            <endpoint address="https://localhost/ClaimsAwareWCF/Service.svc"
binding="customBinding" contract="ClaimsAwareWCF.IService"
bindingConfiguration="CustomBindingConfiguration_IssuedTokenOverTransport" />

            <!-- IssuedTokenOverTransport SecureConversation Endpoint: -->
            <!-- <endpoint address="https://localhost/ClaimsAwareWCF/Service.svc"
binding="customBinding" contract="ClaimsAwareWCF.IService"
bindingConfiguration="CustomBindingConfiguration_IssuedTokenOverTransport_SecureConversat
ion" />-->

            <!-- IssuedTokenForCertificate Endpoint: -->
            <!-- <endpoint address="http://localhost/ClaimsAwareWCF/Service.svc"
binding="customBinding" contract="ClaimsAwareWCF.IService"
bindingConfiguration="CustomBindingConfiguration_IssuedTokenForCertificate" />-->

            <!-- IssuedTokenForSslNegotiated Endpoint: -->
            <!-- <endpoint address="http://localhost/ClaimsAwareWCF/Service.svc"
binding="customBinding" contract="ClaimsAwareWCF.IService"
bindingConfiguration="CustomBindingConfiguration_IssuedTokenForSslNegotiated" />-->

            <!-- IssuedTokenForSslNegotiated SecureConversation Endpoint: -->

```

```

        <!-- <endpoint address="http://localhost/ClaimsAwareWCF/Service.svc"
binding="customBinding" contract="ClaimsAwareWCF.IService"
bindingConfiguration="CustomBindingConfiguration_IssuedTokenForSslNegotiated_SecureConver
sation" />-->

        <!--Commented out by FedUtil-->
        <!--<endpoint address="" binding="wsHttpBinding"
contract="ClaimsAwareWCF.IService"><!-- -
        Upon deployment, the following identity element should be removed or replaced
to reflect the
        identity under which the deployed service runs. If removed, WCF will infer
an appropriate identity
        automatically.
        - -><identity><dns value="localhost" /></identity></endpoint>-->
        <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
    </service>
</services>
<behaviors>
    <serviceBehaviors>
        <behavior name="ClaimsAwareWCF.ServiceBehavior">
            <!-- Behavior extension to make the service claims aware -->
            <federatedServiceHostConfiguration name="ClaimsAwareWCF.Service" />
            <!-- To avoid disclosing metadata information, set the value below to false and
remove the metadata endpoint above before deployment -->
            <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
            <!-- To receive exception details in faults for debugging purposes, set the
value below to true. Set to false before deployment to avoid disclosing exception
information -->
            <serviceDebug includeExceptionDetailInFaults="true" />
            <serviceCredentials>
                <!--Certificate added by FedUtil. Subject='CN=localhost', Issuer='CN=Repro
Root Authority'.-->
                <serviceCertificate findValue="F4323E79A8A50833C45E79547254564C8CA5601F"
storeLocation="LocalMachine" storeName="My" x509FindType="FindByThumbprint" />
            </serviceCredentials>
            <useRequestHeadersForMetadataAddress />
        </behavior>
    </serviceBehaviors>
</behaviors>
<extensions>
    <behaviorExtensions>
        <!-- This behavior extension will enable the service host to be Claims aware -->
        <add name="federatedServiceHostConfiguration"
type="Microsoft.IdentityModel.Configuration.ConfigureServiceHostBehaviorExtensionElement,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
    </behaviorExtensions>
</extensions>
<bindings>
    <customBinding>
        <binding name="CustomBindingConfiguration_IssuedToken">
            <security authenticationMode="IssuedToken">
                <issuedTokenParameters keyType="SymmetricKey" tokenType="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">
                    <issuer address="https://localhost/CustomSTS/ActAsIssuer.svc"
binding="ws2007HttpBinding" bindingConfiguration="IssuedTokenBinding" />
                    <issuerMetadata address="https://localhost/CustomSTS/ActAsIssuer.svc/mex"
/>

```

```

        </issuedTokenParameters>
    </security>
    <textMessageEncoding />
    <httpTransport />
</binding>
<binding name="CustomBindingConfiguration_IssuedTokenOverTransport">
    <security authenticationMode="IssuedTokenOverTransport">
        <issuedTokenParameters keyType="SymmetricKey" tokenType="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">
            <issuer address="https://localhost/CustomSTS/ActAsIssuer.svc"
binding="ws2007HttpBinding" bindingConfiguration="IssuedTokenBinding" />
            <issuerMetadata address="https://localhost/CustomSTS/ActAsIssuer.svc/mex"
/>
        </issuedTokenParameters>
    </security>
    <textMessageEncoding />
    <httpsTransport />
</binding>
<binding
name="CustomBindingConfiguration_IssuedTokenOverTransport_SecureConversation">
    <security authenticationMode="SecureConversation"
messageSecurityVersion="WSecurity11WSTrust13WSecureConversation13WSSecurityPolicy12Basi
cSecurityProfile10" requireSecurityContextCancellation="false">
        <secureConversationBootstrap authenticationMode="IssuedTokenOverTransport"
messageSecurityVersion="WSecurity11WSTrust13WSecureConversation13WSSecurityPolicy12Basi
cSecurityProfile10">
            <issuedTokenParameters keyType="SymmetricKey" tokenType="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">
                <issuer address="https://localhost/CustomSTS/ActAsIssuer.svc"
binding="ws2007HttpBinding" bindingConfiguration="IssuedTokenBinding" />
                <issuerMetadata address="https://localhost/CustomSTS/ActAsIssuer.svc/mex"
/>
            </issuedTokenParameters>
            <localClientSettings cacheCookies="true" detectReplays="true"
replayCacheSize="900000" maxClockSkew="00:05:00" replayWindow="00:05:00"
sessionKeyRenewalInterval="10:00:00" sessionKeyRolloverInterval="00:05:00"
reconnectTransportOnFailure="true" timestampValidityDuration="00:05:00"
cookieRenewalThresholdPercentage="60" />
            <localServiceSettings detectReplays="true" issuedCookieLifetime="10:00:00"
maxStatefulNegotiations="128" replayCacheSize="900000" maxClockSkew="00:05:00"
negotiationTimeout="00:01:00" replayWindow="00:05:00" inactivityTimeout="00:02:00"
sessionKeyRenewalInterval="15:00:00" sessionKeyRolloverInterval="00:05:00"
reconnectTransportOnFailure="true" maxPendingSessions="128" maxCachedCookies="1000"
timestampValidityDuration="00:05:00" />
        </secureConversationBootstrap>
    </security>
    <textMessageEncoding />
    <httpsTransport />
</binding>
<binding name="CustomBindingConfiguration_IssuedTokenForCertificate">
    <security authenticationMode="IssuedTokenForCertificate"
requireDerivedKeys="true" securityHeaderLayout="Strict" includeTimestamp="true"
keyEntropyMode="CombinedEntropy"
messageSecurityVersion="WSecurity11WSTrust13WSecureConversation13WSSecurityPolicy12Basi
cSecurityProfile10" requireSignatureConfirmation="true">
        <issuedTokenParameters keyType="SymmetricKey" tokenType="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">

```



```

        <issuer address="https://localhost/CustomSTS/ActAsIssuer.svc"
binding="ws2007HttpBinding" bindingConfiguration="IssuedTokenBinding">
        <identity>
            <certificateReference storeLocation="LocalMachine" storeName="My"
findValue="CN=localhost" />
        </identity>
        </issuer>
        <issuerMetadata address="https://localhost/CustomSTS/ActAsIssuer.svc/mex"
/>
    </issuedTokenParameters>
</security>
<textMessageEncoding />
<httpTransport />
</binding>
<binding name="CustomBindingConfiguration_IssuedTokenForSslNegotiated">
    <security authenticationMode="IssuedTokenForSslNegotiated">
        <issuedTokenParameters keyType="SymmetricKey" tokenType="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">
            <issuer address="https://localhost/CustomSTS/ActAsIssuer.svc"
binding="ws2007HttpBinding" bindingConfiguration="IssuedTokenBinding" />
            <issuerMetadata address="https://localhost/CustomSTS/ActAsIssuer.svc/mex"
/>
        </issuedTokenParameters>
    </security>
    <textMessageEncoding />
    <httpTransport />
</binding>
<binding
name="CustomBindingConfiguration_IssuedTokenForSslNegotiated_SecureConversation">
    <security authenticationMode="SecureConversation"
requireSecurityContextCancellation="false">
        <secureConversationBootstrap
authenticationMode="IssuedTokenForSslNegotiated">
            <issuedTokenParameters keyType="SymmetricKey" tokenType="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">
                <issuer address="https://localhost/CustomSTS/ActAsIssuer.svc"
binding="ws2007HttpBinding" bindingConfiguration="IssuedTokenBinding" />
                <issuerMetadata address="https://localhost/CustomSTS/ActAsIssuer.svc/mex"
/>
            </issuedTokenParameters>
        </secureConversationBootstrap>
    </security>
    <textMessageEncoding />
    <httpTransport />
</binding>
</customBinding>
<ws2007HttpBinding>
    <binding name="IssuedTokenBinding">
        <security mode="TransportWithMessageCredential">
            <message clientCredentialType="Windows" establishSecurityContext="false" />
        </security>
    </binding>
</ws2007HttpBinding>
</bindings>
</system.serviceModel>
</configuration>

```

NOTE: The configuration above again contains extra customBindings and commented out endpoints that allow you to support other authentication modes such as IssuedToken, IssuedTokenForCertificate, IssuedTokenOverTransport, etc.

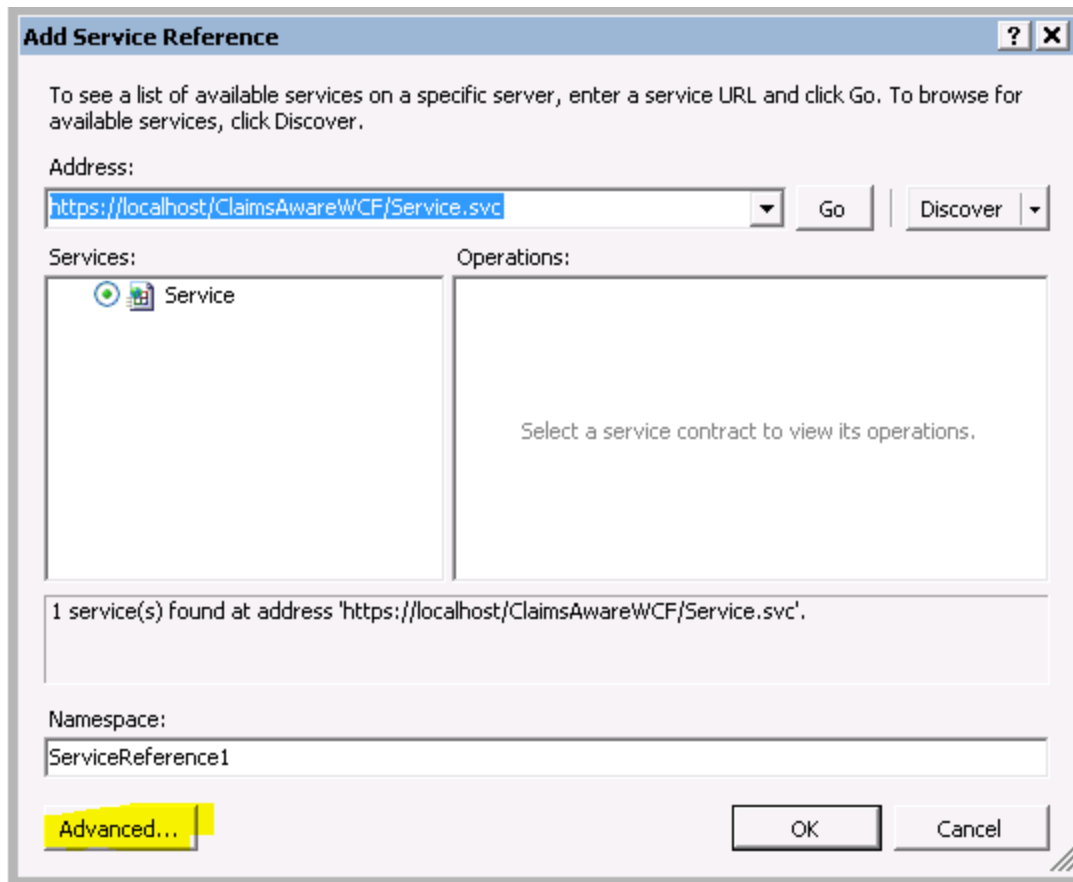
2. After adding in the above configurations to your claims aware WCF web.config file, for your environment you will want to make sure that you have changed the `<microsoft.IdentityModel><service><issuerNameRegistry><trustedIssuers><add>` elements thumbprint attribute so that it matches the thumbprint of your **Repro Signing Cert** certificate. Then you will want to make sure that the `<microsoft.IdentityModel><service><serviceCertificate><certificateReference>` element references the thumbprint value of your **Repro Encryption Cert** certificate. Lastly you'll have to do the same for the `<system.ServiceModel><behaviors><serviceBehaviors><behavior><serviceCredentials><serviceCertificate>` element's findValue attribute to be the thumbprint of your **localhost** certificate.

Phase 4 – Update the Claims Aware ASXP app to call our CustomSTS active endpoint

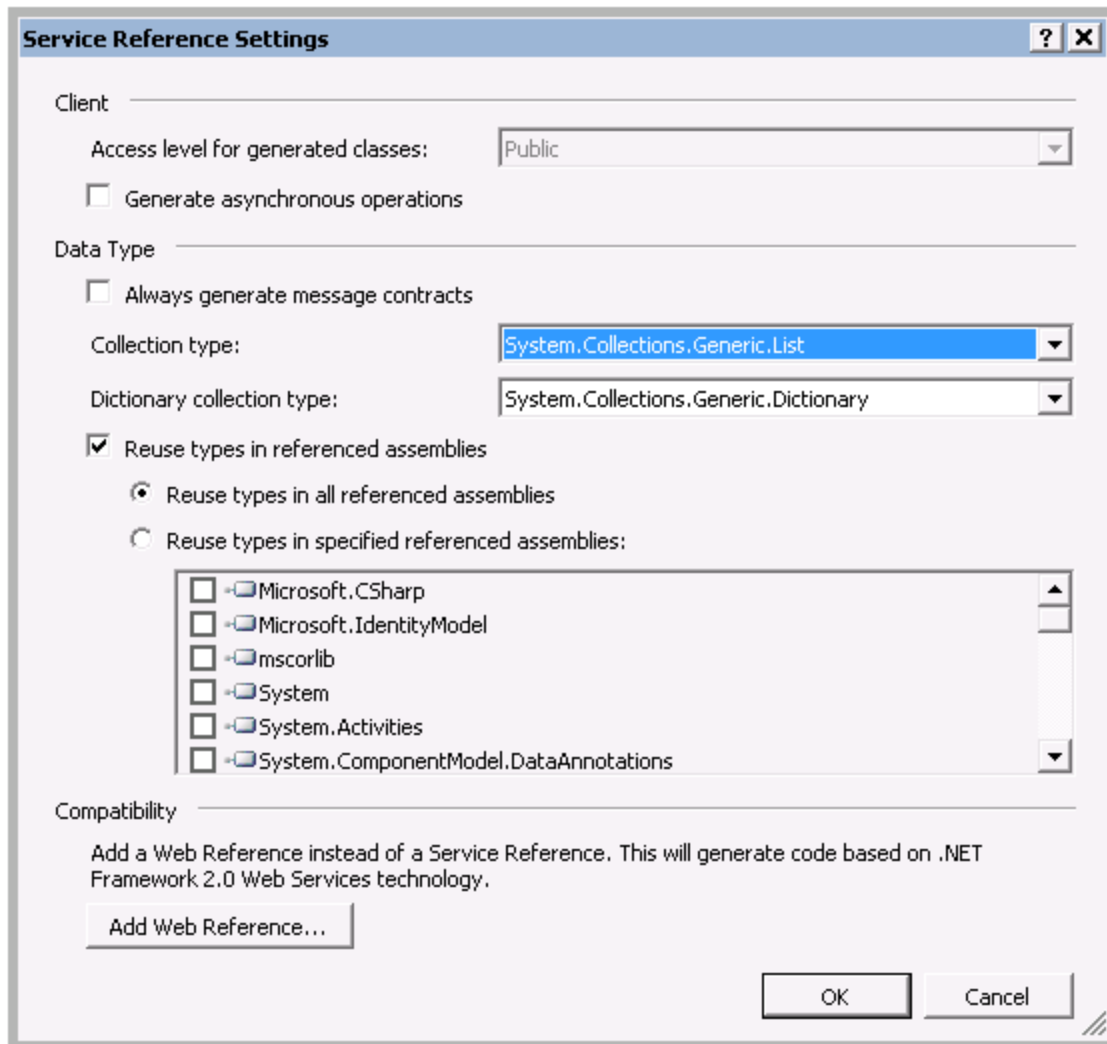
Our CustomSTS is now configured and complete. Our backend claims aware WCF service is also configured and complete. The last step is to update our front end claims aware ASPX web application so that it calls to the backend WCF service and then displays the list of claims returned in the token from the Passive endpoint and the list of claims that are available in the execution of the backend WCF Service method.

Step 1 – Add Service reference to WCF Service from ASPX Relying Party app

1. Right-click on the `https://localhost/ClaimsAwareASXP/` app in the solution explorer and select Add Service Reference...
2. Enter the path to our claims aware WCF service in the Address text box: `https://localhost/ClaimsAwareWCF/Service.svc` then click the OK button
3. Click the Advanced button at the bottom of this dialog box.



4. Change the collection type to `System.Collections.Generic.List` and click OK. We do this because our WCF RP app has a method call `GetClaims` that returns a generic list so we have to update the collection type to generate that generic list code in the proxy class.



5. Leave the namespace name as ServiceReference1 and click OK
6. The service proxy class is generated so that our claims aware ASPX can call the claims aware WCF Service

Step 2 - Modify the claims aware ASPX app to save bootstrap tokens and update web.config file

1. Open the web.config file of the <https://localhost/ClaimsAwareASPX> project
2. Next confirm that our ClaimsAwareASPX application is using the proper certificate reference for our decryption certificate. It might be using the localhost or STSTestCert thumbprint value instead of our encryption cert thumbprint value. Find the `<microsoft.identityModel><serviceCertificate><certificateReference/>` element. This certificate reference is what the relying party application uses to find the private key to decrypt the incoming token. Change the findValue attribute to the thumbprint value of our **CN=Root Encryption Cert** certificate:

```

<serviceCertificate>
  <certificateReference x509FindType="FindByThumbprint"
findValue="7DD17B7807EDA96F1DDD687EB420A097294F0A77" storeLocation="LocalMachine"
storeName="My" />
</serviceCertificate>

```

3. Confirm that the issuerNameRegistry, which references the signing certificate, is using the correct thumbprint value. It might be using the localhost or WIF SDK provided STSTestCert, as read from the FederationMetadata.xml file. That FederationMetadata.xml file is a static file that is created as part of the out of the box WIF project template. There are ways to dynamically create this metadata file so that it has updated details about the endpoints and their properly required certs but I won't go into that for this walkthrough. In the same ClaimsAwareASPX web.config file find the <microsoft.identityModel><issuerNameRegistry><trustedIssuers><add/> element and make sure the thumbprint attribute value matches the thumbprint value of our **CN=Root Signing Cert** certificate:

```

<issuerNameRegistry
type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35">
  <trustedIssuers>
    <add thumbprint="70C883F888C017B3FB02B9887F7835794473C06F"
name="https://localhost/CustomSTS/" />
  </trustedIssuers>
</issuerNameRegistry>

```

4. Find the <microsoft.identityModel> element and then add the saveBootstrapTokens attribute to the <service> element.

```

<microsoft.identityModel>
  <service saveBootstrapTokens="true">

```

When you set this attribute to true then the security token returned by our CustomSTS passive endpoint, the token that the client used to access the claims aware ASPX web app, is added as a bootstrap token to the IIdentity so that you can programmatically use that token to request claims as that user against the CustomSTS Active endpoint. This is a key piece to enable claims identity delegation

5. Update the certificate validation mode. [By default](#) the certificate validation mode is PeerOrChainTrust. Peer means that it will check the LocalMachine -> TrustedPeople store for the certificate. Also the default value for revocationMode is Online, but since we are using self-signed certificates there is no revocation process available for the certificate. We need to change the revocationMode so add the following element under the <microsoft.identityModel><service> section:

```
<certificateValidation certificateValidationMode="PeerOrChainTrust"
revocationMode="NoCheck"/>
```

6. If you modify the WCF relying party application endpoints, and select one of the endpoints that uses the authentication mode of IssuedTokenForSslNegotiated then you will have the following behavior element to your <system.serviceModel> section of the ASPX RP app web.config file. This is because during the SSL negotiation these makecert certificates will fail on certificate revocation checks:

```
<behaviors>
  <endpointBehaviors>
    <behavior>
      <clientCredentials>
        <serviceCertificate>
          <authentication certificateValidationMode="None" revocationMode="NoCheck"/>
        </serviceCertificate>
      </clientCredentials>
    </behavior>
  </endpointBehaviors>
</behaviors>
```

7. If you are using the WCF RP endpoint that uses the IssuedTokenOverTransport authentication mode, then add a service reference and make the changes to the config file suggested above then your web.config file of the ClaimsAwareASPX relying party application will look similar to the following (of course certificate thumbprints values will need to be changed to use your own certificates):

Front end claims aware ASPX web.config file (IssuedTokenOverTransport)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Note: As an alternative to hand editing this file you can use the
  web admin tool to configure settings for your application. Use
  the Website->Asp.Net Configuration option in Visual Studio.
  A full list of settings and comments can be found in
  machine.config.comments usually located in
  \Windows\Microsoft.Net\Framework\v2.x\Config
-->
<configuration>
  <configSections>
    <section name="microsoft.identityModel"
type="Microsoft.IdentityModel.Configuration.MicrosoftIdentityModelSection,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <appSettings>
    <add key="FederationMetadataLocation"
value="https://localhost/CustomSTS/FederationMetadata/2007-06/FederationMetadata.xml" />
  </appSettings>
</configuration>
```

```

</appSettings>
<connectionStrings />
<location path="FederationMetadata">
  <system.web>
    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>
</location>
<system.web>
  <authentication mode="None" />
  <!--
    Set compilation debug="true" to insert debugging
    symbols into the compiled page. Because this
    affects performance, set this value to true only
    during development.
  -->
  <compilation debug="true" targetFramework="4.0">
    <assemblies>
      <add assembly="Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
      PublicKeyToken=31BF3856AD364E35" />
    </assemblies>
  </compilation>
  <!--
    The <authentication> section enables configuration
    of the security authentication mode used by
    ASP.NET to identify an incoming user.
  -->
  <!--Commented out by FedUtil-->
  <!--<authentication mode="Forms"><forms loginUrl="Login.aspx" protection="All"
  timeout="30" name=".ASPXAUTH" path="/" requireSSL="false" slidingExpiration="true"
  defaultUrl="default.aspx" cookieless="UseDeviceProfile" enableCrossAppRedirects="false"
  /></authentication-->
  <!-- Deny Anonymous users. -->
  <authorization>
    <deny users="?" />
  </authorization>
  <!--
    The <customErrors> section enables configuration
    of what to do if/when an unhandled error occurs
    during the execution of a request. Specifically,
    it enables developers to configure html error pages
    to be displayed in place of an error stack trace.

    <customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
      <error statusCode="403" redirect="NoAccess.htm" />
      <error statusCode="404" redirect="FileNotFound.htm" />
    </customErrors>
  -->
  <pages controlRenderingCompatibilityVersion="3.5" />
  <httpRuntime requestValidationType="SampleRequestValidator" />
  <httpModules>
    <add name="ClaimsPrincipalHttpModule"
    type="Microsoft.IdentityModel.Web.ClaimsPrincipalHttpModule, Microsoft.IdentityModel,
    Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
    <add name="WSFederationAuthenticationModule"
    type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule,

```

```

Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
    <add name="SessionAuthenticationModule"
type="Microsoft.IdentityModel.Web.SessionAuthenticationModule, Microsoft.IdentityModel,
Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
    </httpModules>
    <identity impersonate="false" />
</system.web>
<system.codedom>
</system.codedom>
<!--
    The system.webServer section is required for running ASP.NET AJAX under Internet
    Information Services 7.0. It is not necessary for previous version of IIS.
-->
<system.webServer>
    <validation validateIntegratedModeConfiguration="false" />
    <modules>
        <add name="ClaimsPrincipalHttpModule"
type="Microsoft.IdentityModel.Web.ClaimsPrincipalHttpModule, Microsoft.IdentityModel,
Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
preCondition="managedHandler" />
        <add name="WSFederationAuthenticationModule"
type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" preCondition="managedHandler" />
        <add name="SessionAuthenticationModule"
type="Microsoft.IdentityModel.Web.SessionAuthenticationModule, Microsoft.IdentityModel,
Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
preCondition="managedHandler" />
    </modules>
</system.webServer>
<microsoft.identityModel>
    <service saveBootstrapTokens="true">
        <audienceUri>
            <add value="https://localhost/ClaimsAwareASPX/" />
        </audienceUri>
        <federatedAuthentication>
            <wsFederation passiveRedirectEnabled="true" issuer="https://localhost/CustomSTS/"
realm="https://localhost/ClaimsAwareASPX/" requireHttps="true" />
            <cookieHandler requireSsl="true" />
        </federatedAuthentication>
        <serviceCertificate>
            <certificateReference x509FindType="FindByThumbprint"
findValue="7DD17B7807EDA96F1DDD687EB420A097294F0A77" storeLocation="LocalMachine"
storeName="My" />
        </serviceCertificate>
        <certificateValidation certificateValidationMode="PeerOrChainTrust"
revocationMode="NoCheck"/>
        <applicationService>
            <claimTypeRequired>
                <!--Following are the claims offered by STS 'https://localhost/CustomSTS/'. Add
or uncomment claims that you require by your application and then update the federation
metadata of this application.-->
                <claimType type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
optional="true" />
                <claimType type="http://schemas.microsoft.com/ws/2008/06/identity/claims/role"
optional="true" />
            </claimTypeRequired>

```



```

        </applicationService>
        <issuerNameRegistry
type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35">
            <trustedIssuers>
                <add thumbprint="70C883F888C017B3FB02B9887F7835794473C06F"
name="https://localhost/CustomSTS/" />
            </trustedIssuers>
        </issuerNameRegistry>
    </service>
</microsoft.identityModel>
<system.serviceModel>
    <bindings>
        <customBinding>
            <binding name="CustomBinding IService">
                <security defaultAlgorithmSuite="Default"
authenticationMode="IssuedTokenOverTransport"
                    requireDerivedKeys="false" securityHeaderLayout="Strict"
includeTimestamp="true"
                    keyEntropyMode="CombinedEntropy"
messageSecurityVersion="WSecurity11WSTrustFebruary2005WSecureConversationFebruary2005WS
SecurityPolicy11BasicSecurityProfile10">
                    <issuedTokenParameters keyType="SymmetricKey" tokenType="http://docs.oasis-
open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1">
                        <issuer address="https://localhost/CustomSTS/ActAsIssuer.svc"
binding="ws2007HttpBinding"
bindingConfiguration="https://localhost/CustomSTS/ActAsIssuer.svc" />
                        <issuerMetadata address="https://localhost/CustomSTS/ActAsIssuer.svc/mex"
/>
                    </issuedTokenParameters>
                    <localClientSettings cacheCookies="true" detectReplays="false"
replayCacheSize="900000" maxClockSkew="00:05:00"
maxCookieCachingTime="Infinite"
replayWindow="00:05:00" sessionKeyRenewalInterval="10:00:00"
sessionKeyRolloverInterval="00:05:00" reconnectTransportOnFailure="true"
timestampValidityDuration="00:05:00" cookieRenewalThresholdPercentage="60"
/>
                    <localServiceSettings detectReplays="false" issuedCookieLifetime="10:00:00"
maxStatefulNegotiations="128" replayCacheSize="900000"
maxClockSkew="00:05:00"
negotiationTimeout="00:01:00" replayWindow="00:05:00"
inactivityTimeout="00:02:00"
sessionKeyRenewalInterval="15:00:00" sessionKeyRolloverInterval="00:05:00"
reconnectTransportOnFailure="true" maxPendingSessions="128"
maxCachedCookies="1000" timestampValidityDuration="00:05:00" />
                </secureConversationBootstrap />
            </security>
            <textMessageEncoding maxReadPoolSize="64" maxWritePoolSize="16"
messageVersion="Default" writeEncoding="utf-8">
                <readerQuotas maxDepth="32" maxStringContentLength="8192"
maxArrayLength="16384"
                    maxBytesPerRead="4096" maxNameTableCharCount="16384" />
            </textMessageEncoding>
            <httpsTransport manualAddressing="false" maxBufferPoolSize="524288"
maxReceivedMessageSize="65536" allowCookies="false"
authenticationScheme="Anonymous"

```

```

        bypassProxyOnLocal="false" decompressionEnabled="true"
        hostNameComparisonMode="StrongWildcard"
        keepAliveEnabled="true" maxBufferSize="65536"
        proxyAuthenticationScheme="Anonymous"
        realm="" transferMode="Buffered" unsafeConnectionNtlmAuthentication="false"
        useDefaultWebProxy="true" requireClientCertificate="false" />
    </binding>
</customBinding>
<ws2007HttpBinding>
    <binding name="https://localhost/CustomSTS/ActAsIssuer.svc"
closeTimeout="00:01:00"
        openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
        bypassProxyOnLocal="false" transactionFlow="false"
        hostNameComparisonMode="StrongWildcard"
        maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
        messageEncoding="Text"
        textEncoding="utf-8" useDefaultWebProxy="true" allowCookies="false">
        <readerQuotas maxDepth="32" maxStringContentLength="8192"
maxArrayLength="16384"
            maxBytesPerRead="4096" maxNameTableCharCount="16384" />
        <reliableSession ordered="true" inactivityTimeout="00:10:00"
            enabled="false" />
        <security mode="TransportWithMessageCredential">
            <transport clientCredentialType="None" proxyCredentialType="None"
                realm="" />
            <message clientCredentialType="Windows" negotiateServiceCredential="true"
                algorithmSuite="Default" establishSecurityContext="false" />
        </security>
    </binding>
</ws2007HttpBinding>
</bindings>

<!-- NOTE: If your backend claims aware WCF Service is configured to use
the IssuedTokenForSslNegotiated authentication mode then you'll need
to uncomment the following behavior section so that the SSL negotiation
ignores revocation problem of our makecert certs -->
<!--<behaviors>
    <endpointBehaviors>
        <behavior>
            <clientCredentials>
                <serviceCertificate>
                    <authentication certificateValidationMode="None" revocationMode="NoCheck"/>
                </serviceCertificate>
            </clientCredentials>
        </behavior>
    </endpointBehaviors>
</behaviors-->
<client>
    <endpoint address="https://localhost/ClaimsAwareWCF/Service.svc"
        binding="customBinding" bindingConfiguration="CustomBinding IService"
        contract="ServiceReference1.IService" name="CustomBinding IService" />
</client>
</system.serviceModel>
</configuration>

```

Step 3 – Add code to call backend WCF service and display claims

1. Modify the default.aspx.cs code file. Our updated code will display the list of claims that are available to the ASPX page and are issued by our CustomSTS passive endpoint. This list will include the LiveID Claims, the Azure ACS claims, and the claims added in our passive custom security token service.

The code will then get access to the bootstrap token and use that to build a channel to call our backend claims aware WCF service. The backend WCF service enumerates the claims available to it and return that claimset. The claims available to the backend should include the claims from the original bootstrap token (Live ID claims, Azure ACS claims, passive endpoint claims) and then it will also include the claims added in our CustomSTS active endpoint.

Finally the code displays the two sets of claims to the page for viewing.

Default.aspx.cs code behind file

```
//-----  
//  
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF  
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO  
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A  
// PARTICULAR PURPOSE.  
//  
// Copyright (c) Microsoft Corporation. All rights reserved.  
//  
//  
//-----  
  
using System;  
using System.Web.UI;  
using System.Web.UI.WebControls;  
  
using Microsoft.IdentityModel.Claims;  
using System.Collections.Generic;  
using System.IdentityModel.Tokens;  
using System.Threading;  
using System.ServiceModel;  
using System.Security.Cryptography.X509Certificates;  
using Microsoft.IdentityModel.Protocols.WSTrust;  
using System.ServiceModel.Security;  
using System.Text;  
  
public partial class _Default : System.Web.UI.Page  
{  
    Label errorLabel = new Label();  
  
    protected void Page_Load(object sender, EventArgs e)
```

```

{
    errorLabel.Text = "";
    errorLabel.ForeColor = System.Drawing.Color.Red;

    IClaimsPrincipal claimsPrincipal = Page.User as IClaimsPrincipal;
    IClaimsIdentity claimsIdentity = ( IClaimsIdentity )claimsPrincipal.Identity;

    Table PassiveClaimsTable = GetTable();
    Table ActiveClaimsTable = GetTable();
    TableRow newRow;
    TableCell newClaimTypeCell, newClaimValueCell;

    // Populate the list of incoming claims from the CustomSTS
    // passive endpoint into the PassiveClaimsTable
    foreach ( Claim claim in claimsIdentity.Claims )
    {
        newRow = new TableRow();
        newClaimTypeCell = new TableCell();
        newClaimTypeCell.Text = claim.ClaimType;

        newClaimValueCell = new TableCell();
        newClaimValueCell.Text = claim.Value;

        newRow.Cells.Add(newClaimTypeCell);
        newRow.Cells.Add(newClaimValueCell);

        PassiveClaimsTable.Rows.Add(newRow);
    }

    // Call the backend claims aware WCF Service
    // Which returns a list of ViewClaim objects
    List<ServiceReference1.ViewClaim> vc = CallBackendClaimsAwareWCFService();

    // Populate the list of claims from the CustomSTS
    // active endpoint and available inside the WCF
    // service method into the ActiveClaimsTable
    foreach (ServiceReference1.ViewClaim c in vc)
    {
        newRow = new TableRow();
        newClaimTypeCell = new TableCell();
        newClaimTypeCell.Text = c.ClaimType;

        newClaimValueCell = new TableCell();
        newClaimValueCell.Text = c.Value;

        newRow.Cells.Add(newClaimTypeCell);
        newRow.Cells.Add(newClaimValueCell);

        ActiveClaimsTable.Rows.Add(newRow);
    }

    // Create and populate web controls
    Label passiveLabel = new Label();
    passiveLabel.Text = "Claims returned from CustomSTS Passive endpoint and
available inside ASPX code";

```

```

        passiveLabel.Font.Size = 12;
        passiveLabel.Font.Bold = true;
        Label activeLabel = new Label();
        activeLabel.Text = "<br/><br/>Claims returned from CustomSTS Active endpoint and
available inside WCF relying party app";
        activeLabel.Font.Size = 12;
        activeLabel.Font.Bold = true;

        // Build page output to display claim sets:
        this.Controls.Add(passiveLabel);
        this.Controls.Add(PassiveClaimsTable);
        this.Controls.Add(activeLabel);
        this.Controls.Add(ActiveClaimsTable);
        this.Controls.Add(errorLabel);
    }

    protected Table GetTable()
    {
        Table claimsTable = new Table();
        TableRow headerRow = new TableRow();

        TableCell claimTypeCell = new TableCell();
        claimTypeCell.Text = "Claim Type";
        claimTypeCell.BorderStyle = BorderStyle.Solid;

        TableCell claimValueCell = new TableCell();
        claimValueCell.Text = "Claim Value";
        claimValueCell.BorderStyle = BorderStyle.Solid;

        headerRow.Cells.Add(claimTypeCell);
        headerRow.Cells.Add(claimValueCell);
        claimsTable.Rows.Add(headerRow);
        return claimsTable;
    }

    protected List<ServiceReference1.ViewClaim> CallBackendClaimsAwareWCFService()
    {
        List<ServiceReference1.ViewClaim> vc = new List<ServiceReference1.ViewClaim>();

        SecurityToken bootstrapToken =
        ((IClaimsPrincipal)Thread.CurrentPrincipal).Identities[0].BootstrapToken;

        string tmpResult = "Call failed";

        if (bootstrapToken == null)
        {
            errorLabel.Text = tmpResult;
        }

        // Get the channel factory to the backend service from the application state
        ChannelFactory<ServiceReference1.IServiceChannel> factory = new
ChannelFactory<ServiceReference1.IServiceChannel>("CustomBinding_IService");
        //factory.Credentials.ServiceCertificate.SetDefaultCertificate("CN=localhost",
StoreLocation.LocalMachine, StoreName.My);
        factory.Credentials.ServiceCertificate.SetDefaultCertificate("CN=Repro Signing
Cert", StoreLocation.LocalMachine, StoreName.My);
    }

```

```

factory.ConfigureChannelFactory();
factory.Credentials.SupportInteractive = false;

// Create and setup channel to talk to the backend service
ServiceReference1.IServiceChannel channel;

// Setup the ActAs to point to the caller's token so that we perform a delegated
call to the backend service
// on behalf of the original caller.
channel =
factory.CreateChannelActingAs<ServiceReference1.IServiceChannel>(bootstrapToken);

// Call the backend service and handle the possible exceptions
try
{
    vc = channel.GetClaims();

    channel.Close();
}
catch (SecurityAccessDeniedException)
{
    channel.Abort();
    tmpResult = "Access is denied";
}
catch (CommunicationException exception)
{
    StringBuilder sb = new StringBuilder();
    sb.AppendLine(exception.Message);
    sb.AppendLine(exception.StackTrace);
    Exception ex = exception.InnerException;
    while (ex != null)
    {
        sb.AppendLine("=====");
        sb.AppendLine(ex.Message);
        sb.AppendLine(ex.StackTrace);
        ex = ex.InnerException;
    }
    channel.Abort();
    errorLabel.Text = sb.ToString();
}
catch (TimeoutException)
{
    channel.Abort();
    errorLabel.Text = "Timed out...";
}
catch (Exception exception)
{
    StringBuilder sb = new StringBuilder();
    sb.AppendLine("An unexpected exception occurred.");
    sb.AppendLine(exception.StackTrace);
    channel.Abort();
    errorLabel.Text = sb.ToString();
}

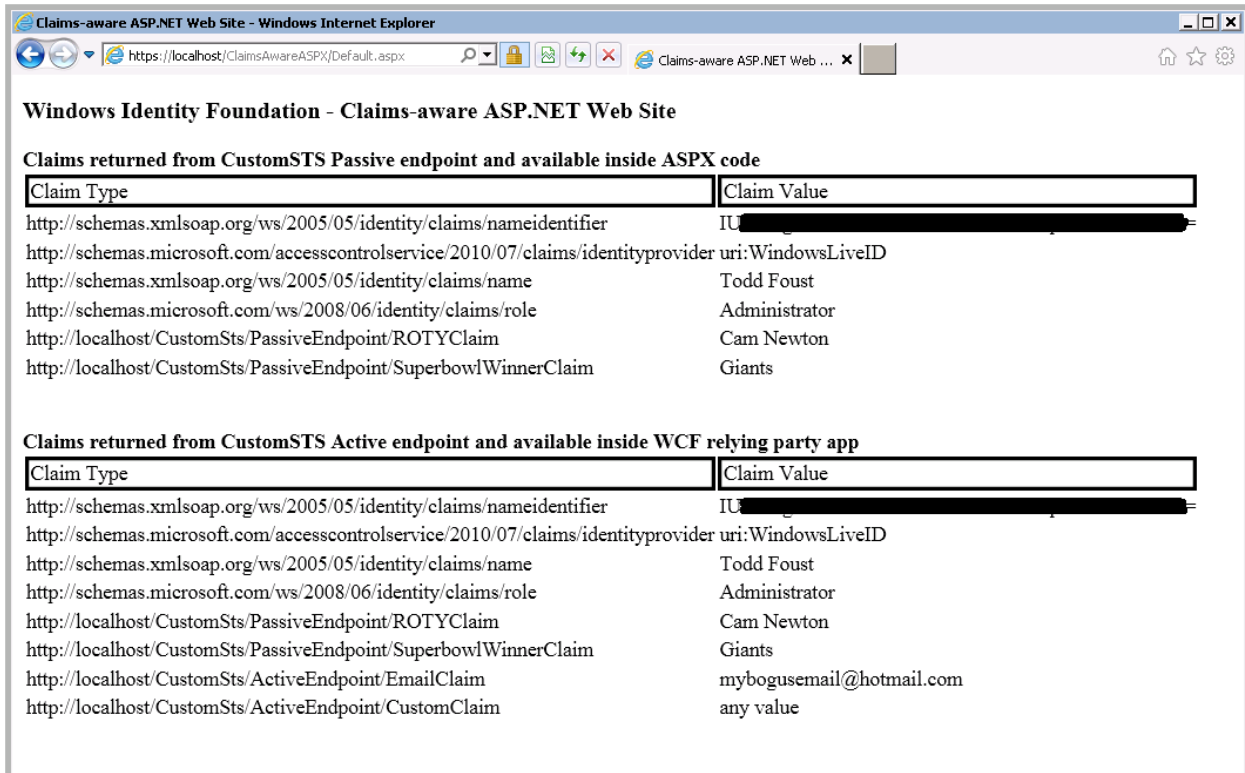
return vc;
}
}

```

Step 4 - Final output

If the application is configured correctly then when you browse to the default.aspx web page of the front end ASPX relying party then you should see two tables of claims. The top table are the list of claims available in the ASPX page, the claims returned from the CustomSTS passive endpoint which include claims from LiveID, any claims added by Azure ACS, and any claims added by the passive endpoint of our STS.

The second table dumps the list of claims that were available inside the execution of the claims aware WCF service method. Because we configured identity delegation using bootstrap tokens and CreateChannelActingAs<> calls the same set of claims are available however this time the token also includes any claims added by the CustomSTS Active endpoint.



Windows Identity Foundation - Claims-aware ASP.NET Web Site

Claims returned from CustomSTS Passive endpoint and available inside ASPX code

Claim Type	Claim Value
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	IU [REDACTED]
http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider	uri:WindowsLiveID
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	Todd Foust
http://schemas.microsoft.com/ws/2008/06/identity/claims/role	Administrator
http://localhost/CustomSts/PassiveEndpoint/ROTYClaim	Cam Newton
http://localhost/CustomSts/PassiveEndpoint/SuperbowlWinnerClaim	Giants

Claims returned from CustomSTS Active endpoint and available inside WCF relying party app

Claim Type	Claim Value
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier	IU [REDACTED]
http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider	uri:WindowsLiveID
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	Todd Foust
http://schemas.microsoft.com/ws/2008/06/identity/claims/role	Administrator
http://localhost/CustomSts/PassiveEndpoint/ROTYClaim	Cam Newton
http://localhost/CustomSts/PassiveEndpoint/SuperbowlWinnerClaim	Giants
http://localhost/CustomSts/ActiveEndpoint/EmailClaim	mybogusemail@hotmail.com
http://localhost/CustomSts/ActiveEndpoint/CustomClaim	any value