Lab 1

Single Sign on for Web Applications

1. This lab extends the guidance in Chapter 2, "*Claims-Based Single Sign-On for the Web and Windows Azure*" of the book "*A Guide to Claims–based Identity and Access Control, 2nd Edition*" (<http://msdn.microsoft.com/en-us/library/ff423674.aspx>). It demonstrates how you can make an existing application claims-aware and perform authorization based on claims presented by users when they are authenticated. It also demonstrates how you can work with WIF sessions, and publish the application to Windows Azure.

# Objectives

After working though this lab, you will understand how claims can be used to implement authentication and authorization for applications, how to enable single sign-on and single sign-out, how WIF Session Mode affects application and network performance, and how you can publish a claims-aware application to Windows Azure. The final optional exercise in this lab shows you how to integrate a claims-based application with Microsoft Active Directory Federation Services (ADFS).

* 1. **Note:** Before you start working with these exercises ensure you have run the dependency checking utility. You must also run Visual Studio as an administrator when opening the solutions in these exercises. If you simply double-click the solution file, Visual Studio may fail to load the projects. See the "*Introduction*" document for information about the dependency checker utility and how to run Visual Studio as an administrator.

This lab contains the following exercises:

* + [Exercise 1](#Ex01): **Making Applications Claims-aware**. In this exercise you will modify two Adatum web applications (a-Order and a-Expense) that currently use forms-based authentication to make them claims-aware, and to provide the user with a single sign-on (SSO) experience.
  + [Exercise 2](#Ex02): **Enabling Single Sign-Out**. In this exercise you will add code to the applications so that users logging out of one are automatically logged out of the other.
  + [Exercise 3](#Ex03): **Using WIF Session Mode**. In this exercise you will modify the applications to change the behavior of the WIF modules so that token information is stored in the session instead of the authentication cookie.
  + [Exercise 4](#Ex04): **Publishing a Claims-Aware Application to Windows Azure**. In this exercise you will modify the a-Expense application so that it can be deployed to Windows Azure while still using the same token issuer as in previous exercises.
  + [Exercise 5](#Ex05): **Integrating the Application with ADFS**. In this optional additional exercise you will modify the applications to authenticate through ADFS 2.0, and then configure ADFS to act as the identity provider and token issuer for the applications.

# Exercise 1: Making Applications Claims-aware

* 1. In this exercise you will modify two Adatum web applications (a-Order and a-Expense) that currently use forms-based authentication to make them claims-aware, and to provide the user with a single sign-on (SSO) experience.

This exercise contains the following tasks:

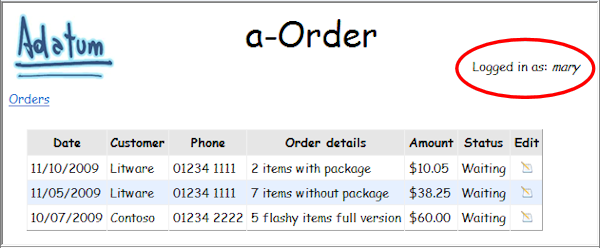
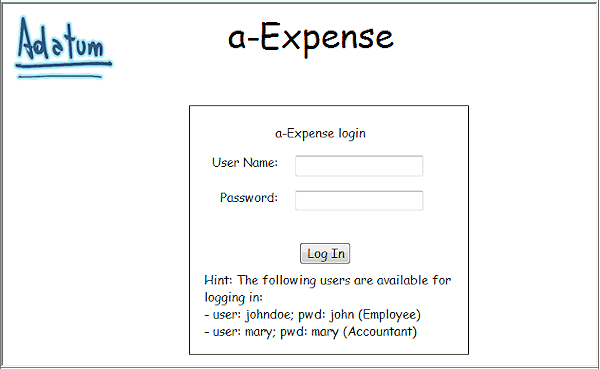
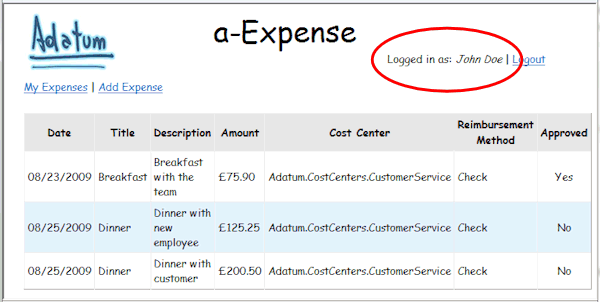
* + [Task 1](#Ex01Task01): Understand the no-claims scenario.
  + [Task 2](#Ex01Task02): Add the Adatum simulated issuer to the solution.
  + [Task 3](#Ex01Task03): Make the a-Expense application claims-aware.
  + [Task 4](#Ex01Task04): Make the a-Order application claims-aware.
  + [Task 5](#Ex01Task05): Analyze the authentication sign-in flow.

This is the longest exercise in this lab, and you should be able to complete it in approximately 60 minutes.

## Task 1: Understand the No-claims Scenario

* 1. In this task you will explore the existing a-Order and a-Expense applications to understand how they implement authentication and authorization.

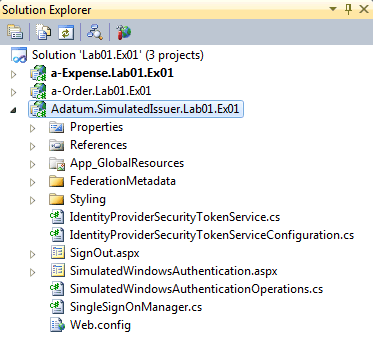
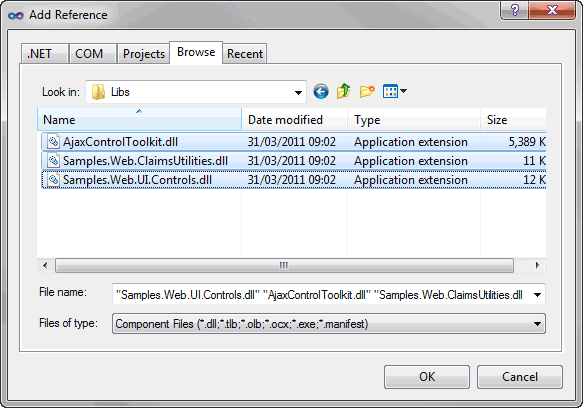
To explore the existing a-Order and a-Expense applications

* 1. Start Visual Studio as an administrator and open the solution named **Lab01.Ex01.sln** from the **Lab01-SingleSignOn\Source\Ex01\Begin** folder.
  2. In Solution Explorer right-click the top-level **Lab01.Ex01** solution item and click **Rebuild solution**.
  3. Open a web browser and navigate to the URL <https://localhost/a-Order.Lab01.Ex01> to run the a-Order application. You do not need to enter any credentials because this application simulates Windows Integrated authentication using Active Directory. You will see that you are logged into the application as the user named **mary**.
     1. 
  4. In Solution Explorer double-click the file **web.config** in the **a-Order.Lab01.Ex01** project to open it in the editor. You will see that it contains the following sections that specify the authentication parameters for the application.
     1. XML
     2. <authentication mode="None" />
     3. <authorization>
     4. <allow roles="Employee, Order Approver" />
     5. <deny users="\*" />
     6. </authorization>
     7. These settings specify that the application will allow anonymous access, but that the user must be a member of one of the Windows groups named **Employee** or **Order Approver**. In a production scenario you would set the authentication mode to "Windows" to force the user to be authenticated. However, to avoid the need to create specific users in your own Active Directory repository the example uses a hard-coded user defined in **Global.asax**.
  5. In Solution Explorer double-click the file **Global.asax** in the **a-Order.Lab01.Ex01** project to open its code file (**Global.asax.cs**) in the code editor window. You will see that it contains the following code that defines a property named **MaryMay** that returns an instance of an **IPrincipal** implementation with the name **mary** and membership of the roles named **OrderApprover** and **Employee**.
     1. C#
     2. private static IPrincipal MaryMay
     3. {
     4. get
     5. {
     6. IIdentity identity = new GenericIdentity("mary");
     7. string[] roles = { Roles.OrderApprover, Roles.Employee };
     8. return new GenericPrincipal(identity, roles);
     9. }
     10. }
     11. The roles used in the code above are defined in the file **Roles.cs** in the **Data** subfolder of the **a-Order.Lab01.Ex01** project.
  6. The **Global.asax.cs** file also contains the following handler for the **Application.AuthenticateRequest** event that sets the **User** property of the current ASP.NET context to the value of the **MaryMay** property. This causes the application to run in the context of the account for the user named **mary**, as you saw earlier.
     1. C#
     2. protected void Application\_AuthenticateRequest(object sender, EventArgs e)
     3. {
     4. this.Context.User = MaryMay;
     5. }
  7. Go back to your web browser and navigate to the URL <https://localhost/a-Expense.Lab01.Ex01> to run the a-Expense application. This application uses ASP.NET Forms authentication with a mock user repository database. You must enter the appropriate credentials to log into the application.
     1. 
  8. Enter the user name **johndoe** and the password **john**, and then click **Log In**. You will see the main page of the application (Default.aspx), and the indication that you are logged in as **John Doe**.
     1. 
  9. This application represents a typical legacy application that implements the authentication logic throughout the classes and files of the project. In Visual Studio open the file **web.config** from the **a-Expense.Lab01.Ex01** project into the code editor window. You will see that it contains the following sections that specify ASP.NET Forms authentication and denies access to all unauthenticated users.
     1. XML
     2. <authentication mode="Forms">
     3. <forms name=".ASPXAUTH" loginUrl="~/login.aspx" defaultUrl="~/default.aspx"
     4. requireSSL="true"></forms>
     5. </authentication>
     6. <authorization>
     7. <deny users="?" />
     8. </authorization>
  10. Open the file **BasePage.cs** from the **a-Expense.Lab01.Ex01** project into the code editor window. This is the base class for the web pages used in the application. You will see that it contains the following code that declares an abstract field names **AuthorizedRoles**, and it also overrides the **OnLoad** method to call another method named **Authorize**.
      1. C#
      2. protected abstract IEnumerable<string> AuthorizedRoles { get; }
      3. protected override void OnLoad(EventArgs e)
      4. {
      5. this.Authorize();
      6. base.OnLoad(e);
      7. }
  11. The **Authorize** method looks in the current session for a logged-on user. If one is not found, it redirects the request to the login page. If there is a logged-on user it checks that the user name is in the list of users in the **AuthorizedRoles** property of this page. If not, the code redirects the request to another page that displays an access denied message.
      1. C#
      2. private void Authorize()
      3. {
      4. var user = (User)this.Session["LoggedUser"];
      5. if (user == null)
      6. {
      7. this.Response.Redirect("~/Login.aspx");
      8. return;
      9. }
      10. if (user.Roles.Where(r => this.AuthorizedRoles.Contains(r.Name)).Count() == 0)
      11. {
      12. this.Response.Redirect("~/AccessDenied.aspx");
      13. return;
      14. }
      15. }
      16. The **Authorize** method runs when a page that inherits this base class loads to ensure the request came from a valid user, and prevent access if not.
  12. Right-click the file **Default.aspx** in the **a-Expense.Lab01.Ex01** project and click **View Code** to open the code behind page for the default page the user sees when accessing the application. You will see that it inherits the base class **BasePage** you just looked at. The code in **Default.aspx.cs** also includes the following overrides of the **AuthorizedRoles** property and the **OnLoad** method. These specify the roles named **Employee** and **Accountant** as being the required role membership for users, and save the current user in the Session.
      1. C#
      2. protected override IEnumerable<string> AuthorizedRoles
      3. {
      4. get { return new[] { Role.Employee, Role.Accountant }; }
      5. }
      6. protected override void OnLoad(EventArgs e)
      7. {
      8. var user = (User)this.Session["LoggedUser"];
      9. ... Code here to retrieve and display the list of expenses ...
      10. base.OnLoad(e);
      11. }
      12. The sample application uses mock repositories for both the list of users (**UserRepository.cs** in the **Data** subfolder) and the list of expenses displayed in the page. This removes the requirement for the example to include a database. In a production scenario, the application would use a suitable database server to store the information.
  13. Close all of the files open in the Visual Studio editor, but keep the solution open so that you are ready to use it in the next task.
  14. You have now completed this task and you should appreciate how authentication and authorization is typically implemented using a mix of techniques common in existing (and, in particular, legacy) applications. In the next few tasks of this lab you will add a suitable claims issuer (STS) to the solution you've used, and then modify the existing a-Order and a-Expense applications to use claims instead.

## Task 2: Add the Adatum Simulated Issuer to the Solution

* 1. In this task you will add a mock claims issuer (STS) to the solution. This will allow you to authenticate users against the STS and then use the claims in the tokens it issues within the applications. We provide a project that implements the simulated issues with the samples.

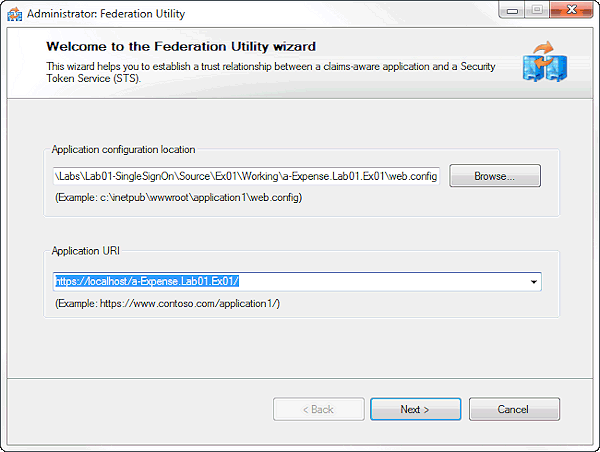
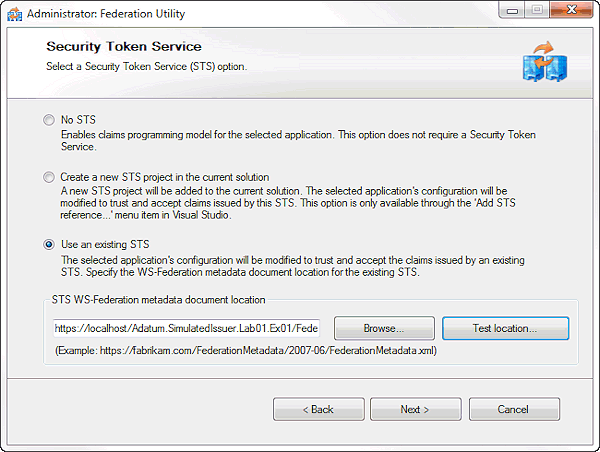
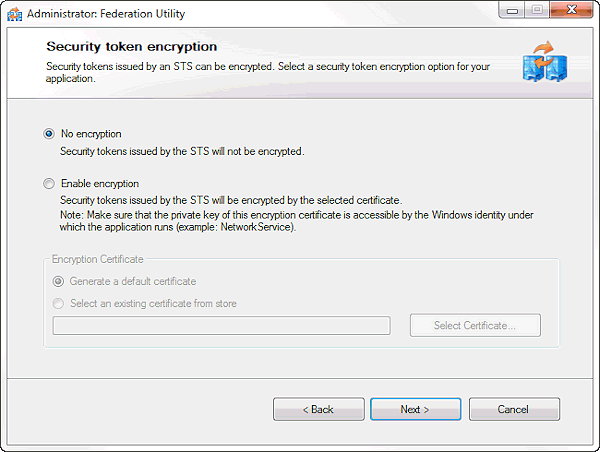
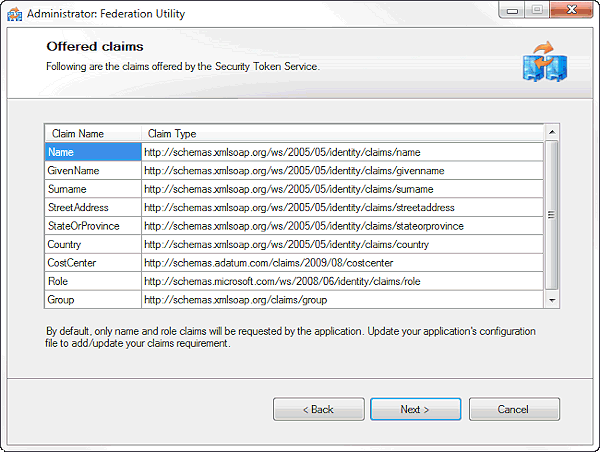
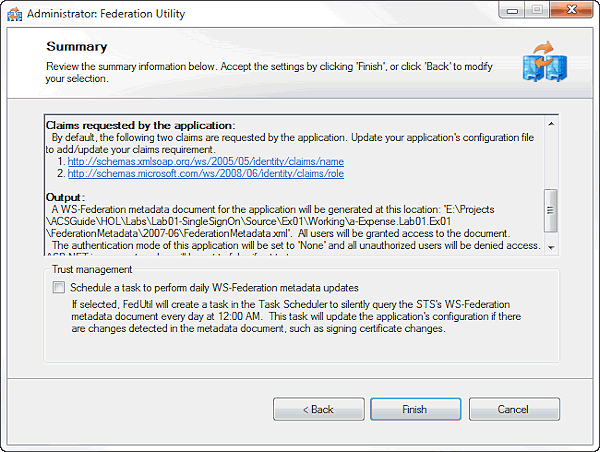
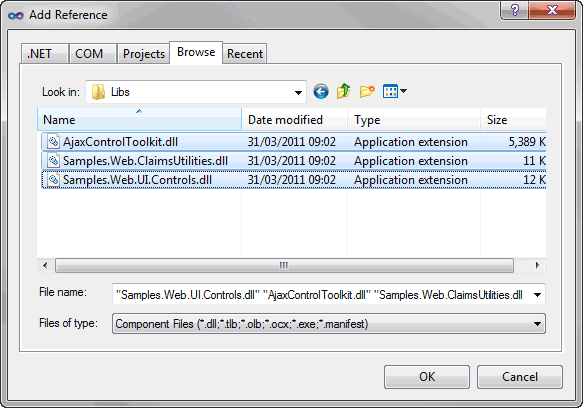
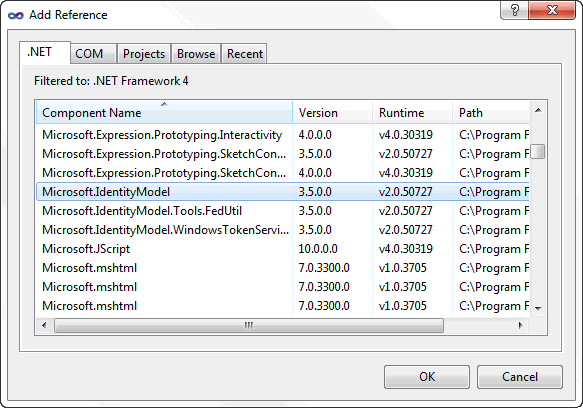
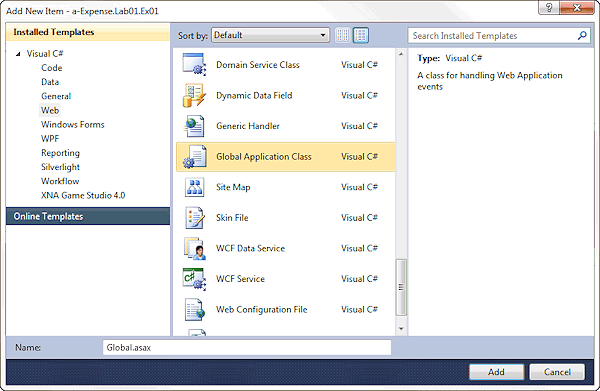
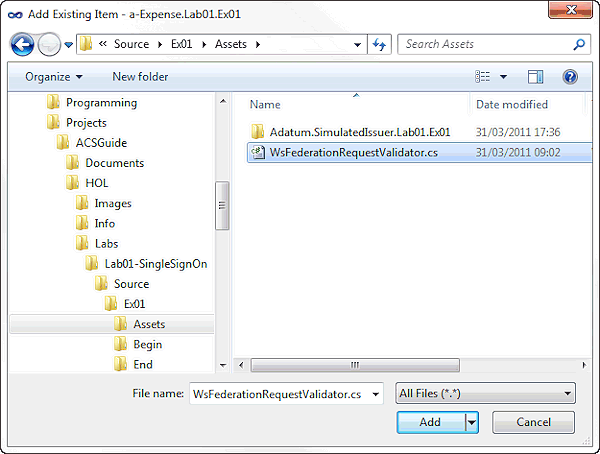
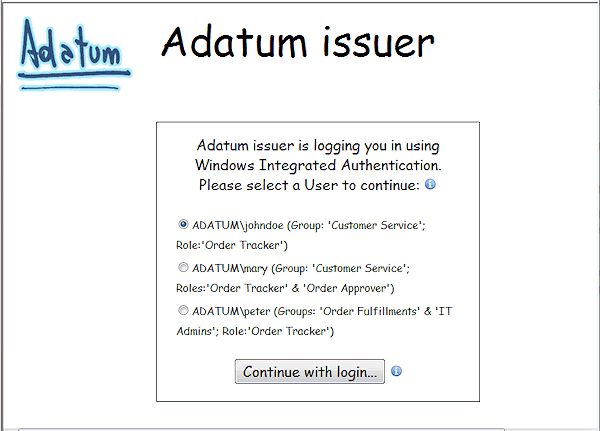
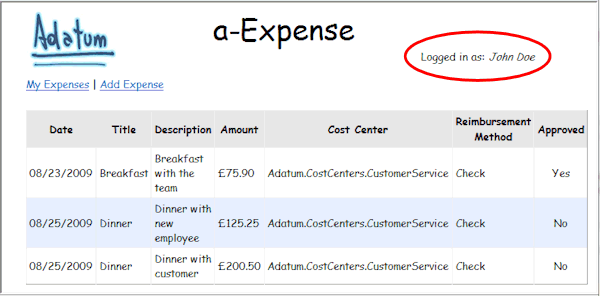
To add the Adatum simulated issuer to the solution

* 1. Continue with the **Lab01.Ex01** solution you used in the previous task.
  2. **In Windows Explorer, navigate to the folder Lab01-SingleSignOn\Source\Ex01\Assets and copy the folder named AdatumSimulatedIssuer.Lab01.Ex01 (including all of its contents) into the Lab01-SingleSignOn\Source\Ex01\Begin** folder.
  3. **In Visual Studio Solution Explorer, right-click on the top-level Lab01.Ex01 solution item, point to Add, and click Existing Project. Navigate to the folder Lab01-SingleSignOn\Source\Ex01\Begin\AdatumSimulatedIssuer.Lab01.Ex01, select the file named AdatumSimulatedIssuer.Lab01.Ex01.csproj, and click Open. The AdatumSimulatedIssuer.Lab01.Ex01** project now appears in Solution Explorer.
     1. 
  4. **In Solution Explorer, right-click on the AdatumSimulatedIssuer.Lab01.Ex01 project and click Add Reference. In the Add Reference dialog click the Browse tab and navigate to the folder Lab01-SingleSignOn\Source\Shared Code\Libs.** Select the three assemblies **AjaxControlToolkit.dll**, **Samples.WebClaimsUtilities.dll**, **and Samples.Web.UI.Controls.dll**.
     1. 
  5. **Click OK to add the references to the project.**
  6. In Solution Explorer double-click the file **SimulatedWindowsAuthentication.aspx** to open it in the code editor. You will see that it contains the following declaration of an option button list where a user can select which account to use when the simulated issuer is invoked from an application, and a button to initiate the logon process.
     1. XML
     2. <div id="UserOptions">
     3. <asp:RadioButtonList ID="UserList" runat="server">
     4. <asp:ListItem Text="ADATUM\johndoe (Group: 'Customer Service';
     5. Role:'Order Tracker')" Value="ADATUM\johndoe" Selected="True" />
     6. <asp:ListItem Text="ADATUM\mary (Group: 'Customer Service';
     7. Roles:'Order Tracker' & 'Order Approver')" Value="ADATUM\mary" />
     8. <asp:ListItem Text="ADATUM\peter (Groups: 'Order Fulfillments' & 'IT Admins';
     9. Role:'Order Tracker')" Value="ADATUM\peter" />
     10. </asp:RadioButtonList>
     11. </div>
     12. <asp:Button ID="ContinueButton" runat="server" class="tooltip"
     13. Text="Continue with login..." OnClick="ContinueButtonClick" />
  7. In Solution Explorer right-click the file **SimulatedWindowsAuthentication.aspx** and click **View Code** to open the code behind file for this page. It contains the handler for the "Continue" button, which calls a method named **LogOnUser** to log on the user, and a method named **HandleSignInRequest** that uses other classes within the issuer to generate the token for the response.
     1. C#
     2. protected void ContinueButtonClick(object sender, EventArgs e)
     3. {
     4. SimulatedWindowsAuthenticationOperations.LogOnUser(this.UserList.SelectedValue,
     5. this.Context, this.Request, this.Response);
     6. this.HandleSignInRequest();
     7. }
     8. The **LogOnUser** method creates a **GenericIdentity** and a **GenericPrincipal** instance, and then creates the required authentication cookie and adds it to the response. To see the claims that are added for the selected user, open the file **IdentityProviderSecurityTokenService**.cs and look at the **GetOutputClaimsIdentity** method.
  8. Look at the **OnLoad** method of the **SimulatedWindowsAuthentication.aspx.cs** file. This method runs whenever a user accesses the issuer. It examines the request to see if this is a sign-in or sign-out action request. For a sign-in request the code attempts to log in the user. For a sign-out request (or a sign-out and clean up request) the code redirects the request to another page that logs out the user. All of these actions use the methods exposed by the Windows Identity Foundation (WIF) framework.
  9. In Solution Explorer right-click the top-level **Lab01.Ex01** solution item and click **Rebuild solution**. Ensure that the solution builds with no errors.
  10. You have now completed this task. The solution contains an STS that you can use to authenticate users for the a-Order and a-Expense applications. In the next task you will modify the a-Expense application to use this STS for authentication

## Task 3: Make the a-Expense Application Claims-aware

* 1. In this task you will modify the a-Expense application to use the simulated issuer you added in the previous task. The a-Expense application stores the current user in the ASP.NET Session, and so you will add code to the application that uses the claims in the token provided by the simulated issuer to generate a user instance and store this in the Session so that the remainder of the application works as it did previously.

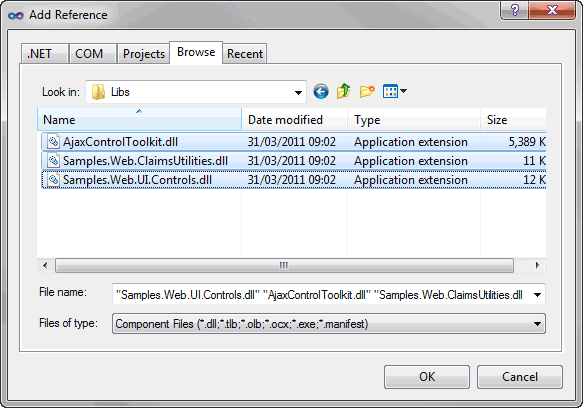
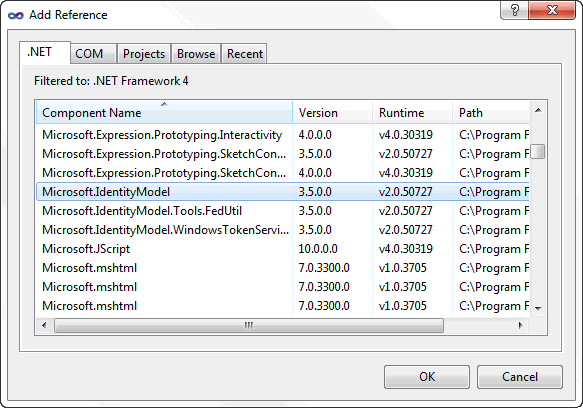
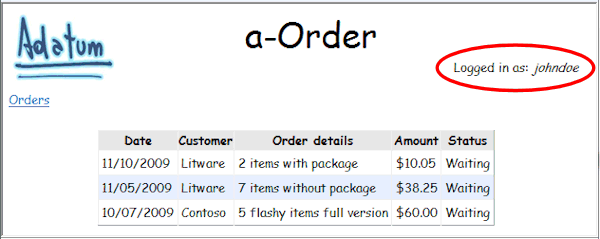
To make the a-Expense application claims-aware

* 1. Continue with the **Lab01.Ex01** solution you used in the previous task.
  2. The first stage of the process for making the application claims-aware is to add a reference to a token issuer. In Solution Explorer, right-click the **a-Expense.Lab01.Ex01** project item and click **Add STS Reference**. This menu option is added to Visual Studio when you install the Windows Identity Framework.
  3. In the first page of the Wizard, enter **https://localhost/a-Expense.Lab01.Ex01/** as the **Application URI**. Do not change the application configuration location value.
     1. 
  4. Click **Next** and select **Use an existing STS**. Enter the value **https://localhost/Adatum.SimulatedIssuer.Lab01.Ex01/FederationMetadata/2007-06/FederationMetadata.xml** for the STS WS-Federation metadata document location.
     1. 
  5. Click **Test location**, click **Yes** in the certificate warning dialog that appears, and you will see an XML document displayed in your web browser. This is the federation document that defines the service and claims exposed by the STS.
  6. Close your web browser and click **Next** in the STS wizard. Select **No** **encryption** in the Security token encryption page.
     1. 
  7. **Click Next and you will see a list of the claims found in the federation metadata document. Notice the message that indicates that the application will only request the name and role claims from the STS when authenticating.** 
     1. 
  8. **Click Next and review the information in the Summary page of the wizard. Notice that you can specify that the wizard automatically updates the federation metadata by downloading it from the STS on a regular schedule. Do not set this option in the sample.** 
     1. 
  9. **Click Finish to complete the process of adding the STS to the application. A Success dialog appears to confirm that the process is complete.**
  10. **In Solution Explorer double-click the file Web.config in the a-Expense.Lab01.Ex01 project to open it in the editor. You will see that the FedUtil utility that you used to add the STS reference to the project has commented out the <authentication mode="Forms"> element as the application will now use the STS to perform authentication. The FedUtil utility also inserts two WIF modules into the web server request pipeline, as shown here.**
      1. ****XML****
      2. **<system.webServer>**
      3. **<modules runAllManagedModulesForAllRequests="true">**
      4. **<add name="WSFederationAuthenticationModule"**
      5. **type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule, ..." />**
      6. **<add name="SessionAuthenticationModule"**
      7. **type="Microsoft.IdentityModel.Web.SessionAuthenticationModule, ... " />**
      8. **</modules>**
      9. **</system.webServer>**
      10. **These two modules automatically interact with requests to the application and perform authentication and authorization based on claims received from a token issuer. You will see more about how these modules and WIF are used later in this and subsequent tasks.**
  11. **FedUtil also adds the location of the federation metadata document to the <appSettings> section of Web.config (some parts of the full file path have been removed for clarity here).**
      1. ****XML****
      2. **<appSettings>**
      3. **<add key="FederationMetadataLocation"**
      4. **value="https://localhost/.../2007-06/FederationMetadata.xml" />**
      5. **</appSettings>**
  12. **Look at the <microsoft.identityModel> section. This contains information about the STS you added to the project. Notice the list of required claim types generated from the federation metadata file by the wizard. You can see how, by default, it only requires the name and role claims form the STS. If you want to require other claims that are available from the STS you can modify the configuration here.**
      1. ****XML****
      2. **<claimTypeRequired>**
      3. **<!--Following are the claims offered by STS**
      4. **'https://localhost/Adatum.SimulatedIssuer.Lab01.Ex01/'. Add or uncomment claims**
      5. **that you require by your application and then update the federation metadata of**
      6. **this application.-->**
      7. **<claimType type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"**
      8. **optional="true" />**
      9. **<claimType type="http://schemas.microsoft.com/ws/2008/06/identity/claims/role"**
      10. **optional="true" />**
      11. **<!--<claimType**
      12. **type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname"**
      13. **optional="true" />-->**
      14. **<!--<claimType**
      15. **type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname"**
      16. **optional="true" />-->**
      17. **<!--<claimType**
      18. **type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/streetaddress"**
      19. **optional="true" />-->**
      20. **... other claims here ...**
      21. **</claimTypeRequired>**
      22. **The WIF modules check that the claims listed as being required are present in the token returned by the STS. It is good practice to specify those that your application depends on to avoid errors when it comes to use claims that may not be present.**
  13. **Now you can begin the process of using the claims in the a-Expense application. In Solution Explorer right-click on the a-Expense.Lab01.Ex01 project and click Add Reference. In the Add Reference dialog click the Browse tab and navigate to the folder Lab01-SingleSignOn\Source\Shared Code\Libs.** Select the three assemblies **AjaxControlToolkit.dll**, **Samples.WebClaimsUtilities.dll**, **and Samples.Web.UI.Controls.dll**.
      1. 
  14. **In Solution Explorer, right-click on the a-Expense.Lab01.Ex01 project again and click Add Reference. This time, in the Add Reference dialog, click the .NET tab and s**elect the assembly **Microsoft.IdentityModel.**
      1. 
  15. **Click OK to add the reference.**
  16. **In Solution Explorer, right-click on the a-Expense.Lab01.Ex01 project once more, point to Add**, and click **New Item**. Select **Web** in the list of Installed Templates, and select **Global Application Class** in the central column. **The name is automatically set to Global.asax.**
      1. 
  17. **Click Add to add the new file to the project. The code behind file named Global.asax.cs automatically opens in the code editor.**
  18. **Move the using statements that are added to the file by default so that they are inside the AExpense namespace and add the following highlighted using statements to the end of the list to reference the types you will use in your code. Your code should look like that shown here**
      1. ****C#****
      2. **namespace AExpense**
      3. **{**
      4. **using System;**
      5. **using System.Collections.Generic;**
      6. **using System.Linq;**
      7. **using System.Web;**
      8. **using System.Web.Security;**
      9. **using System.Web.SessionState;**
      10. **using Samples.Web.ClaimsUtilities;**
      11. **using System.Globalization;**
      12. **using AExpense.Data;**
      13. **using Microsoft.IdentityModel.Protocols.WSIdentity;**
      14. **using Microsoft.IdentityModel.Web;**
      15. **using Microsoft.IdentityModel.Tokens;**
      16. **using Microsoft.IdentityModel.Web.Configuration;**
      17. **public class Global : System.Web.HttpApplication**
  19. **Delete all of the event handler outlines except for** **Application\_Start and Session\_Start from the Global class (do not delete the class definition). Your class definition should look like the following.**
      1. ****C#****
      2. **public class Global : System.Web.HttpApplication**
      3. **{**
      4. **protected void Application\_Start(object sender, EventArgs e)**
      5. **{**
      6. **}**
      7. **protected void Session\_Start(object sender, EventArgs e)**
      8. **{**
      9. **}**
      10. **}**
  20. **You will now add code to the Session\_Start method to populate the current User object and store it in the Session. Add the following highlighted code to the Session\_Start method.**
      1. ****C#****
      2. protected void Session\_Start(object sender, EventArgs e)
      3. {
      4. **if (this.Context.User.Identity.IsAuthenticated)**
      5. **{**
      6. // Access the authenticated user information using the Thread.CurrentPrincipal.
      7. **string issuer = ClaimHelper.GetCurrentUserClaim(**
      8. **WSIdentityConstants .ClaimTypes.Name).OriginalIssuer;**
      9. **string givenName = ClaimHelper.GetCurrentUserClaim(**
      10. **WSIdentityConstants.ClaimTypes.GivenName).Value;**
      11. **string surname = ClaimHelper.GetCurrentUserClaim(**
      12. **WSIdentityConstants.ClaimTypes.Surname).Value;**
      13. **string costCenter = ClaimHelper.GetCurrentUserClaim(**
      14. **Adatum.ClaimTypes.CostCenter).Value;**
      15. **var repository = new UserRepository();**
      16. **var user = repository.GetUser(this.User.Identity.Name);**
      17. **user.CostCenter = costCenter;**
      18. **user.FullName = givenName + " " + surname;**
      19. // Store the user reference in the ASP.NET Session.
      20. **this.Context.Session["LoggedUser"] = user;**
      21. **}**
      22. }
      23. The application will no longer use the local repository to authenticate users. Authentication will be performed by the STS. The WS-Federation Authentication Module automatically parses the token received from the STS and sets the user information as the current thread principal. This means that the **IsAuthenticated** property of the **Context.User.Identity** will return true.
      24. **Notice that the code makes use of methods such as GetCurrentUserClaim that are defined in the ClaimHelper class in the Samples.Web.ClaimsUtilities assembly you imported into the project earlier in this task. .**
  21. **You must also add some code to encrypt the authentication cookies sent to clients that use the application. Add the following highlighted code to the Application\_Start method so that the method named OnServiceConfigurationCreated will be executed when the application starts.**
      1. ****C#****
      2. **protected void Application\_Start(object sender, EventArgs e)**
      3. **{**
      4. **FederatedAuthentication.ServiceConfigurationCreated**
      5. **+= OnServiceConfigurationCreated;**
      6. **}**
  22. Now add the following implementation of the **OnServiceConfigurationCreated** method to the **Global.asax** file.
      1. C#
      2. private static void OnServiceConfigurationCreated(object sender,
      3. ServiceConfigurationCreatedEventArgs e)
      4. {
      5. // Use the <serviceCertificate> to protect cookies sent to the client.
      6. var sessionTransforms =
      7. new List<CookieTransform>(
      8. new CookieTransform[]
      9. {
      10. new DeflateCookieTransform(),
      11. new RsaEncryptionCookieTransform(e.ServiceConfiguration.ServiceCertificate),
      12. new RsaSignatureCookieTransform(e.ServiceConfiguration.ServiceCertificate)
      13. });
      14. var sessionHandler = new SessionSecurityTokenHandler(
      15. sessionTransforms.AsReadOnly());
      16. e.ServiceConfiguration.SecurityTokenHandlers.AddOrReplace(sessionHandler);
      17. }
      18. **This code uses the methods of the WIF (Microsoft.Identity) classes to create a set of transformations that will be applied to cookies to deflate, encrypt, and sign them. It then creates an instance of a security token handler that will apply these transformations to cookies, and replaces the default security token handler for the application with this new one. This means that all cookies will be encrypted and signed by the security token handler.**
  23. **Save and close the Global.asax file. Then, in Solution Explorer, right-click on the a-Expense.Lab01.Ex01 project, point to Add**, and click **Existing Item**. Navigate to the **Lab01-SingleSignOn\Source\Ex01\Assets** folder and select the file **WsFederationRequestValidator.cs.**
      1. 
  24. **Click Add. Then, in Solution Explorer, double-click the new file WsFederationRequestValidator.cs to open it in the Visual Studio editor. You will see that it contains just the following method named IsValidRequestString.**
      1. ****C#****
      2. **protected override bool IsValidRequestString(HttpContext context, string value,**
      3. **RequestValidationSource requestValidationSource,**
      4. **string collectionKey, out int validationFailureIndex)**
      5. **{**
      6. **validationFailureIndex = 0;**
      7. **if (requestValidationSource == RequestValidationSource.Form &&**
      8. **collectionKey.Equals(WSFederationConstants.Parameters.Result,**
      9. **StringComparison.Ordinal))**
      10. **{**
      11. **if (WSFederationMessage.CreateFromFormPost(context.Request)**
      12. **as SignInResponseMessage != null)**
      13. **{**
      14. **return true;**
      15. **}**
      16. **}**
      17. **return base.IsValidRequestString(context, value, requestValidationSource,**
      18. **collectionKey, out validationFailureIndex);**
      19. **}**
      20. **This method first checks if the response is a Form post containing the result of an authentication request. If it is, the code then checks if there is a response message from the STS in the response. If so, it returns true allowing the response to be processed. If not, it returns the result of processing the response through the remainder of the HTTP pipeline.**
  25. **In Solution Explorer double-click the file Web.config file in the a-Expense.Lab01.Ex01 project to open it in the editor. Add the following highlighted line to the main <system.web> section (not the <system.web> section inside the preceding <location>section).** 
      1. XML
      2. </location>
      3. <system.web>
      4. **<httpRuntime requestValidationType="Adatum.WsFederationRequestValidator" />**
      5. <authentication mode="None" />
      6. <compilation debug="true" targetFramework="4.0">
      7. **This forces the application to use the custom request validation class you just added to the project instead of the default ASP.NET request validator. The default ASP.NET request validator may reject as dangerous content some values commonly used when performing claims-based authentication.**
  26. **The cookie encryption code you added to Global.asax earlier requires a certificate for encrypting the cookies. Add the following highlighted lines to the <service> section of the <Microsoft.identitymodel> section in the Web.config file to specify the server certificate the code will use.**
      1. ****XML****
      2. <microsoft.identityModel>
      3. <service>
      4. **<serviceCertificate>**
      5. **<certificateReference x509FindType="FindByThumbprint"**
      6. **findValue="5a074d678466f59dbd063d1a98b1791474723365" />**
      7. **</serviceCertificate>**
      8. **<certificateValidation certificateValidationMode="None" />**
      9. <audienceUris>
      10. <add value="https://localhost/a-Expense.Lab01.Ex01/" />
      11. You have now completed all of the steps required to implement authentication for the application. The remaining tasks are to remove the login feature (it is no longer required because the STS will perform authentication), and update the authorization code in the application to allow users in roles specified on the simulated issuer to access the application. The application will perform authorization in exactly the same way as before, but the role names it will uses are those defined in **Roles** enumeration in the common assembly named **Samples.Web.ClaimsUtilities**.
  27. In Solution Explorer, expand the **Data** folder and double-click the file **Role.cs**. This defines the role names that are valid for the application. To be able to use the **Roles** enumeration values, add the following highlighted **using** statement to the file.
      1. C#
      2. namespace AExpense.Data
      3. {
      4. using System;
      5. **using Samples.Web.ClaimsUtilities;**
      6. public class Role
  28. Now edit the declaration of the two valid role names as shown in the following highlighted code so that they use the **Roles** enumeration values exposed by the utility assembly, instead of being simple string values.
      1. C#
      2. public class Role
      3. {
      4. public Guid Id { get; set; }
      5. public string Name { get; set; }
      6. public static readonly string Employee = **Adatum.Roles.Employee**;
      7. public static readonly string Accountant = **Adatum.Roles.Accountant**;
      8. }
  29. Save and close the **Role.cs** file and then, in Solution Explorer, expand the **Styling** folder and double-click the file **Site.Master** to open it in the editor.
  30. Scroll to about half way down this page, find the following **<div>** element, and delete the highlighted content. The **LoginStatus** control serves no purpose when the application is using claims-based authentication instead of forms-based authentication.
      1. XML
      2. <div id="topuserinformation">
      3. Logged in as: <i><%=((User)this.Session["LoggedUser"]).FullName%></i> **|**
      4. **<asp:LoginStatus ID="LoginStatus1" runat="server" OnLoggedOut="..." />**
      5. </div>
  31. Save and close the **Site.master** file. Then. Back in Solution Explorer, find the file **Login.aspx** and delete it. It is no longer required as users will present their logon credentials to the identity provider and not to the application.
      1. In the sample, the simulated issuer acts as the identity provider, and simply allows users to select one of the existing identities it defines.
  32. In Solution Explorer right-click the top-level **Lab01.Ex01** solution item and click **Rebuild solution**.
  33. Open your web browser and navigate to the URL <https://localhost/a-Expense.Lab01.Ex01/> to run the claims-aware a-Expense application. It now uses the simulated issuer for authentication, as you see the Adatum Issuer page where you can select the account to use to log on.
      1. 
      2. Of course, in a real issuer, you would see the logon page where you enter your credentials to be authenticated. The simulated issuer makes it easy to work with test accounts while you are developing solutions that use claims-based authentication.
  34. Select the **ADATUM\johndoe** account and click **Continue with login**. You see the same list of expenses as you did in Task 1 of this exercise. Notice that you are logged in as John Doe, just as before, but this time using the claims in the token issued by the simulated issuer.
      1. 
  35. You have now completed this task. The a-Expense application now uses the simulated token issuer for authentication (you added this token issuer to the solution in the previous task of this exercise). The a-Expense application continues to use the same code for *authorization* as it did before. This shows how you can often add claims authentication to an existing application without needing to change the authorization code. In the next task you will do much the same with the a-Order application.

## Task 4: Make the a-Order Application Claims-aware

* 1. In this task you will modify the a-Order application to use the simulated issuer you added in Task 2 of this exercise. At the moment, the a-Order application simulates Windows Integrated authentication using Active Directory. You will add code to the application that uses the claims in the token provided by the simulated issuer instead. The process is similar to that you carried out in the previous task of this exercise.

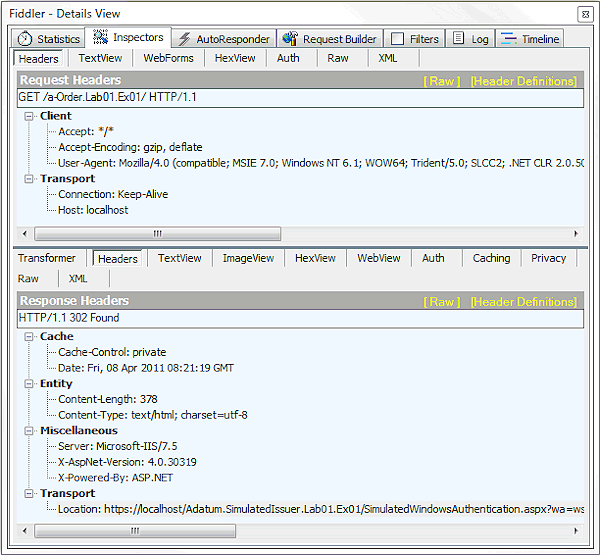
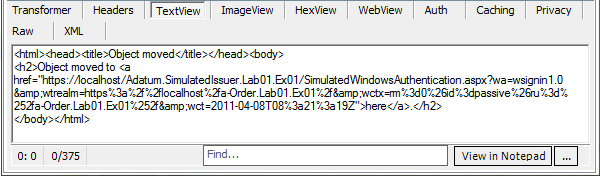
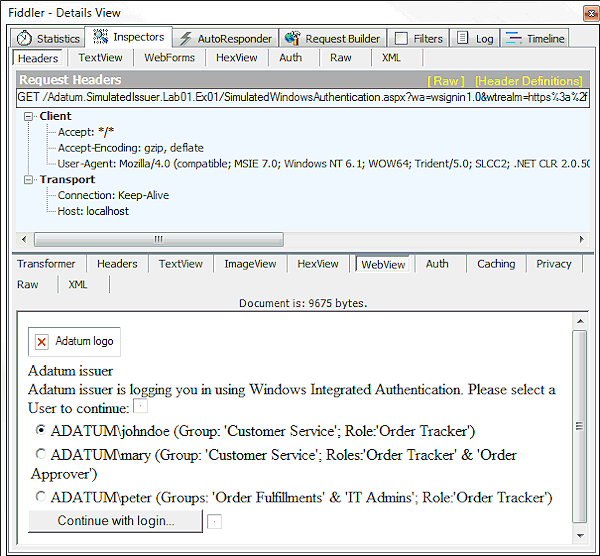
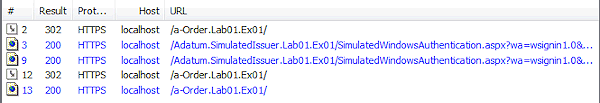
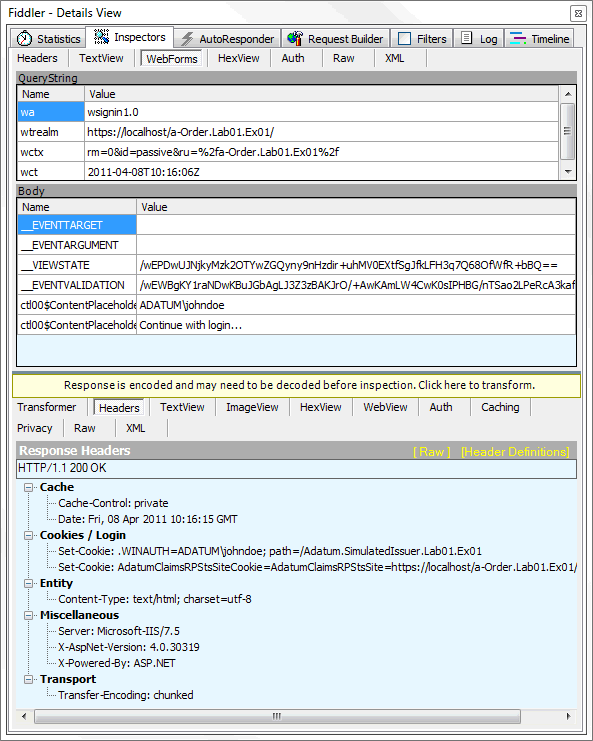
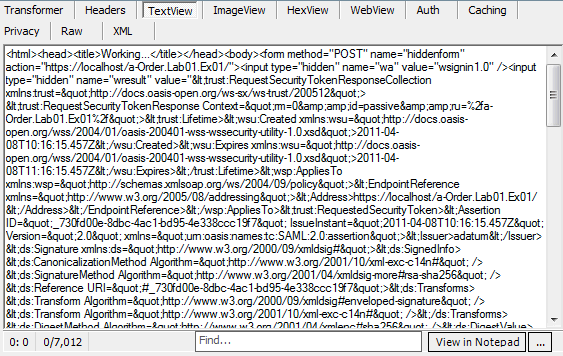
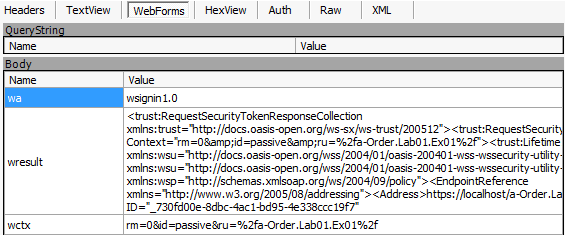
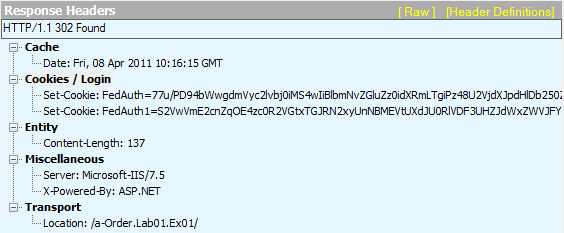
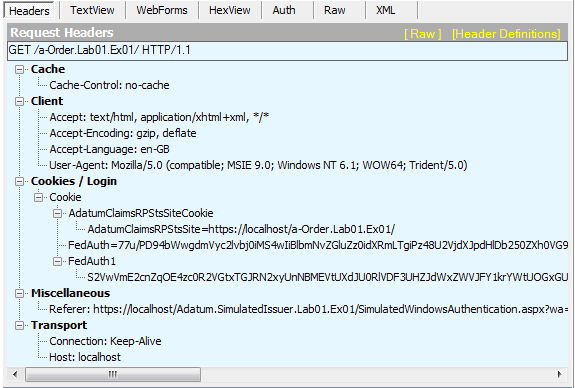
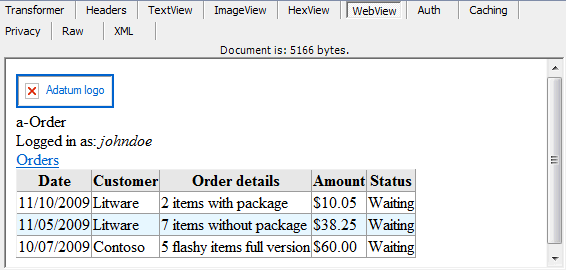
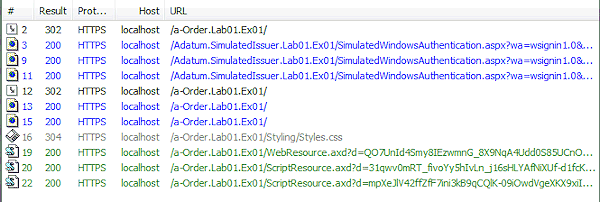
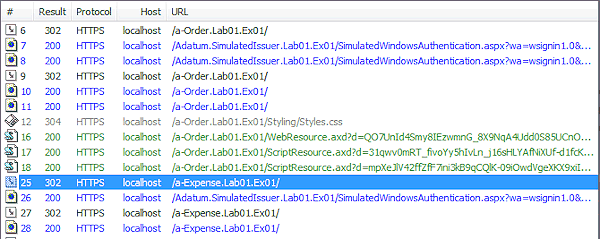
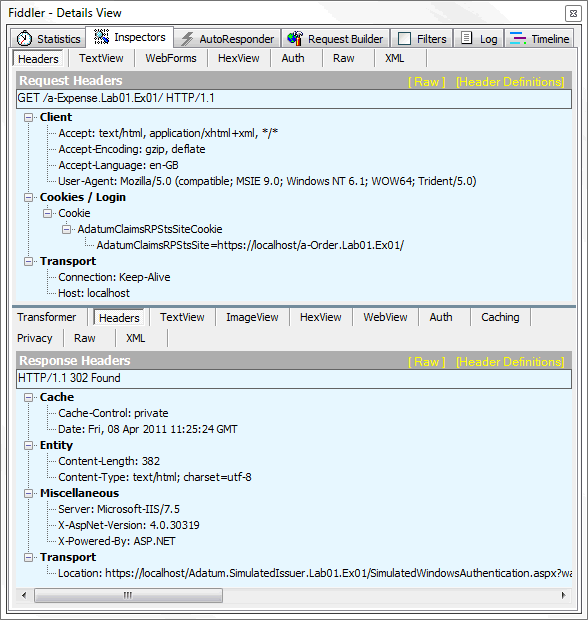
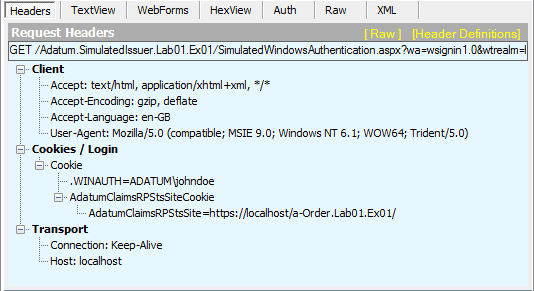
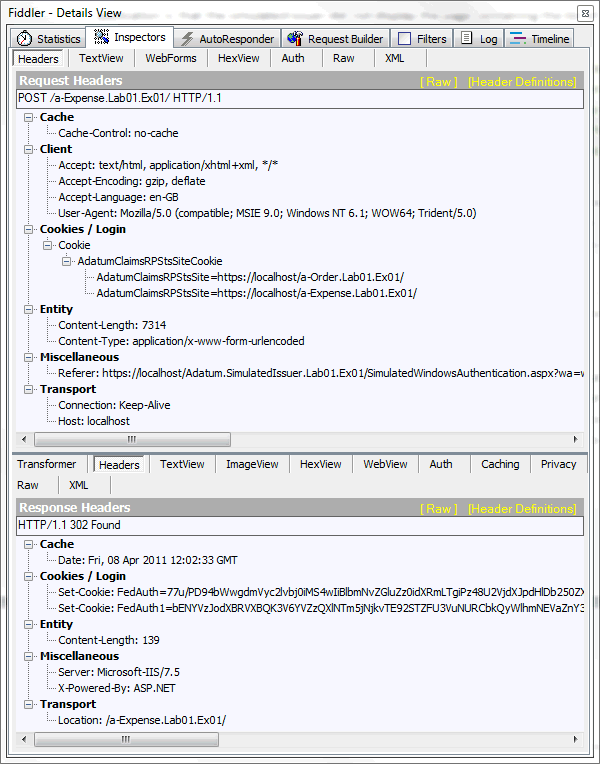
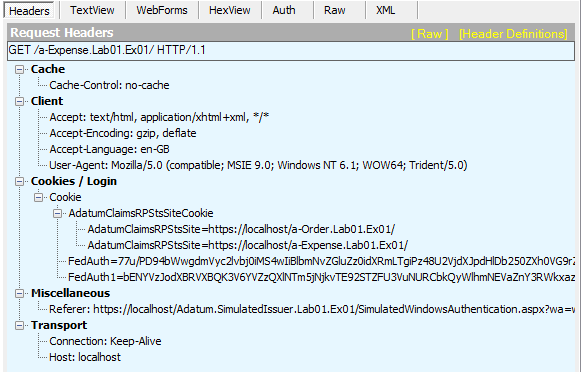
To make the a-Order application claims-aware

* 1. Continue with the **Lab01.Ex01** solution you used in the previous task.
  2. The first stage of the process for making the application claims-aware is to add a reference to a token issuer. In Solution Explorer, right-click the **a-Order.Lab01.Ex01** project item and click **Add STS Reference**.
  3. Work through the wizard in the same way as you did in the previous task, using the following values for the settings in the wizard:
     + Application URI: **https://localhost/a-Order.Lab01.Ex01/**
     + STS WS-Federation metadata document location: **https://localhost/Adatum.SimulatedIssuer.Lab01.Ex01/FederationMetadata/2007-06/FederationMetadata.xml**
     + Security token encryption: **No** **encryption**
  4. **In Solution Explorer right-click on the a-Order.Lab01.Ex01 project and click Add Reference. In the Add Reference dialog click the Browse tab and navigate to the folder Lab01-SingleSignOn\Source\Shared Code\Libs.** Select the three assemblies **AjaxControlToolkit.dll**, **Samples.WebClaimsUtilities.dll**, **and Samples.Web.UI.Controls.dll**.
     1. 
  5. **In Solution Explorer, right-click on the a-Order.Lab01.Ex01 project again and click Add Reference. This time, in the Add Reference dialog, click the .NET tab and s**elect the assembly **Microsoft.IdentityModel.**
     1. 
  6. **Click OK to add the reference. Then, in Solution Explorer double-click the file Global.asax to open its code file in the editor and add the following to the end of the existing list of using statements inside the namespace declaration to reference the types you will use in your code.**
     1. ****C#****
     2. **using Samples.Web.ClaimsUtilities;**
     3. **using System.Globalization;**
     4. **using System.Collections.Generic;**
     5. **using Microsoft.IdentityModel.Protocols.WSIdentity;**
     6. **using Microsoft.IdentityModel.Web;**
     7. **using Microsoft.IdentityModel.Tokens;**
     8. **using Microsoft.IdentityModel.Web.Configuration;**
  7. **Delete all of the existing content of Global class, and then add the following highlighted code to define an event handler for the Application\_Start event.**
     1. ****C#****
     2. **public class Global : HttpApplication**
     3. **{**
     4. **private void Application\_Start(object sender, EventArgs e)**
     5. **{**
     6. **FederatedAuthentication.ServiceConfigurationCreated**
     7. **+= OnServiceConfigurationCreated;**
     8. **}**
     9. **}**
  8. **Now add** the following implementation of the **OnServiceConfigurationCreated** method to the **Global.asax.cs** file. This is the same as the code you added to the a-Expense application in the previous task.
     1. C#
     2. private static void OnServiceConfigurationCreated(object sender,
     3. ServiceConfigurationCreatedEventArgs e)
     4. {
     5. // Use the <serviceCertificate> to protect cookies sent to the client.
     6. var sessionTransforms =
     7. new List<CookieTransform>(
     8. new CookieTransform[]
     9. {
     10. new DeflateCookieTransform(),
     11. new RsaEncryptionCookieTransform(e.ServiceConfiguration.ServiceCertificate),
     12. new RsaSignatureCookieTransform(e.ServiceConfiguration.ServiceCertificate)
     13. });
     14. var sessionHandler = new SessionSecurityTokenHandler(
     15. sessionTransforms.AsReadOnly());
     16. e.ServiceConfiguration.SecurityTokenHandlers.AddOrReplace(sessionHandler);
     17. }
  9. **Save and close the Global.asax file.**
  10. **In Solution Explorer, right-click on the a-Order.Lab01.Ex01 project, point to Add**, and click **Existing Item**. **As you did in the previous task**, navigate to the **Lab01-SingleSignOn\Source\Ex01\Assets** folder and select the file **WsFederationRequestValidator.cs. Click Add to add it to the project.**
  11. **In Solution Explorer double-click the Web.config file to open it in the editor, and add the following highlighted line to the main <system.web> section (not the <system.web> section inside the preceding <location>section).** This is the same as the modification you made to the **Web.config** file in the a-Expense project in the previous task.
      1. XML
      2. </location>
      3. <system.web>
      4. **<httpRuntime requestValidationType="Adatum.WsFederationRequestValidator" />**
      5. <authorization>
      6. <deny users="?" />
  12. **Now add the following highlighted lines to the <service> section of the <Microsoft.identitymodel> section in the Web.config file to specify the server certificate the code will use.** This is also the same as the modification you made to the **Web.config** file in the a-Expense project in the previous task.
      1. ****XML****
      2. <microsoft.identityModel>
      3. <service>
      4. **<serviceCertificate>**
      5. **<certificateReference x509FindType="FindByThumbprint"**
      6. **findValue="5a074d678466f59dbd063d1a98b1791474723365" />**
      7. **</serviceCertificate>**
      8. **<certificateValidation certificateValidationMode="None" />**
      9. <audienceUris>
      10. <add value="https://localhost/a-Order.Lab01.Ex01/" />
      11. You have now completed all of the steps required to implement authentication for the application. As in the a-Expense application, the remaining task is update the authorization code in the application to allow users in roles specified on the simulated issuer to access the application. The application will perform authorization in exactly the same way as before, but the role names it will uses are those defined in **Roles** enumeration in the common assembly named **Samples.Web.ClaimsUtilities**.
  13. In Solution Explorer, expand the **Data** folder and double-click the file **Roles.cs**. Add the following highlighted **using** statement inside the namespace of the file.
      1. C#
      2. namespace AOrder.Data
      3. {
      4. **using Samples.Web.ClaimsUtilities;**
      5. public class Roles
  14. Now edit the declaration of the two valid role names as shown in the following highlighted code so that they use the **Roles** enumeration values exposed by the utility assembly, instead of being simple string values. Notice that you also need to change the types from constants to static strings.
      1. C#
      2. public class Roles
      3. {
      4. public **static** readonly string OrderApprover = **Adatum.Roles.OrderApprover**;
      5. public **static** readonly string Accountant = **Adatum.Roles.Employee**;
      6. }
  15. Save and close the **Roles.cs** file. Then, in Solution Explorer, right-click the top-level **Lab01.Ex01** solution item and click **Rebuild solution**.
  16. Open your web browser and navigate to the URL <https://localhost/a-Order.Lab01.Ex01/> to run the claims-aware a-Order application. As with the a-Expense application you modified in the previous task of this lab, the a-Order application now uses the simulated issuer for authentication. You see the same **Adatum Issuer** page where you can select the account to use to log on.
  17. Select the **ADATUM\johndoe** account and click **Continue with login**. You see the same list of orders as you did in Task 1 of this exercise. Notice that you are logged in as the user **johndoe**, but this time through the claims in the token issued by the STS (the simulated issuer).
      1. 
      2. The a-Expense application you worked with in the previous task stores a custom implementation of a **User** object in the session after a successful logon, and this object has a **FullName** property containing the full user name. The a-Order application you worked with in this task just sets the **Identity** property of the application's current user context to an **IPrincipal** instance that does not contain the full user name. This is why the a-Expense application can display the user's full name, while the a-Order application (as you can see in the screenshot) only displays the username.
  18. Now, without closing your browser, navigate to <https://localhost/a-Expense.Lab01.Ex01/> to open the a-Expense application. You will see that you are not prompted to log on again. The token you obtained from the simulated issuer when you logged on to the a-Order application indicates that you have already been authenticated, and so the simulated issuer will generate a suitable token for the a-Expense application automatically (both applications trust this issuer). You can switch between the applications without needing to be re-authenticated.
      1. To understand how this works, remember the code you examined in the simulated issuer in Task 2 of this exercise. The **OnLoad** event handler in the **SimulatedWindowsAuthentication** class looks for a WIF **Action** parameter in the query string of the request. If this is a sign-in request (**Actions.SignIn**) for a user that was already authenticated, the code automatically generates a suitable token for the target application.
  19. Close your web browser and then open the URL <https://localhost/a-Expense.Lab01.Ex01/> again. You will see that you are once again prompted to log in. The cookie sent to your browser containing the authentication token is not persistent, and is destroyed when you close the browser.
  20. You have now completed this task. The a-Orders and a-Expense applications are now claims aware and support single sign-on. In the next task you will examine the authentication sign-in flow in more detail using an HTTP packet inspector utility to see the series of requests, tokens, and cookies that are part of the authentication process.

## Task 5: Analyze the Authentication Sign-in Flow

* 1. In this task you will examine the authentication requests, tokens, and cookies that are part of the authentication process. The step-by-step procedure in this task uses the **Fiddler** intercepting proxy utility with Internet Explorer 9. However, you can use any other HTTP packet inspection utility instead.
  2. **Note**: If you are using Internet Explorer 8 to work with the examples in these labs, you will find that Fiddler does not detect packets when you specify the URL of the application using **localhost**. The suggested way to resolve this in the Fiddler help files is to use the DNS or network name of your computer. However, this will not work with the examples we provide because the authentication configuration specifies **localhost** URLs. If you decide to use Fiddler, you must also use Internet Explorer 9 or a non-Microsoft browser to view the example applications.

To analyze the authentication sign-in flow using Fiddler

* 1. Continue with the **Lab01.Ex01** solution you used in the previous task.
  2. Start Fiddler from your **Start** menu, or by clicking the icon it adds to the browser toolbar. Ensure you have configured Fiddler for decrypting HTTPS traffic.
     1. See the section "*Configuring Fiddler for the Labs*" in the *Introduction* document provided with these labs for instructions on how to configure Fiddler to decrypt HTTPS traffic. You can download Fiddler free from <http://www.fiddler2.com/fiddler2/version.asp>.
  3. In your web browser, navigate to the URL <https://localhost/a-Order.Lab01.Ex01/> to open the a-Order application. If you are prompted by Fiddler to ignore the remote certificate error, click **Yes**. You will also see a warning in the browser that there is a problem with the website certificate. Select the option to continue into the site. You will see the **Adatum issuer** page.
     1. The warning occur because Fiddler captures the HTTP traffic, decrypts the contents, and then re-encrypts it using its own certificate before passing the traffic on to the browser.
  4. In the Fiddler list of web requests, you will see two requests. The first is your request to the a-Order application. This is followed by a request to the simulated token issuer you added to the solution in the earlier tasks of this exercise. For each request/response, the list shows the HTTP response code, the protocol, the host computer, and the URL requested. There are also columns containing useful debugging and performance information that are not shown in the screenshot.
     1. 
     2. The numbering of the requests (in the first column) is not sequential because Fiddler is filtering out the extraneous ones that are of no interest here.
  5. Double-click the first item in the list to show the request and response headers in the Details View pane of Fiddler (ensure that the **Inspectors** tab is selected). This view shows information about the request sent from the browser and the response received. In the **Request Headers** section you can see this was a **GET** request for the path **/a-Order.Lab01.Ex01/** (the a-Order application).
     1. 
     2. By default the Details View pane is part of the main window alongside the list of requests, but you can click the **Tearoff** icon in the toolbar to display it as a separate window as shown here.
  6. Look at the **Response Headers** section of the Details View pane. It shows that the server returned an HTTP **302** (Found) response. The headers also specify the **Location** that the browser should navigate (redirect) to in order to. This is the URL of the simulated issuer. If you right-click the **Location** value and click **Copy only Value** you can paste it into a text editor to see the complete URL more easily. The URL it redirects to will look something like this (some values such as the date and time will, of course, be different): **https://localhost/Adatum.SimulatedIssuer.Lab01.Ex01/SimulatedWindowsAuthentication.aspx?wa=wsignin1.0&wtrealm=https%3a%2f%2flocalhost%2fa-Order.Lab01.Ex01%2f&wctx=rm%3d0%26id%3dpassive%26ru%3d%252fa-Order.Lab01.Ex01%252f&wct=2011-04-07T10%3a04%3a50Z**.
     1. This initial redirection from the a-Order application to the simulated issuer occurs because the WIF authentication module in the a-Order application pipeline detects that the user is not authenticated (there is no cookie containing an authentication token in the request), and so it redirects the browser to the URL specified for authentication. This URL is defined in the **<federatedAuthentication>** element in the a-Order application's **web.config** file. The **wa** parameter in the request is set to **wsignin1.0** (equivalent to the WIF **Actions.SignIn** value you saw in previous tasks of this lab when you examined the simulated issuer code). The **wtrealm** parameter specifies the URL of the application that is requesting authentication. The **wctx** parameter specifies the original request URL within the context of the request. The **wct** parameter specifies the date and time of the request.
  7. Click the "view" buttons (**TextView**, **WebForms**, **HexView**, **Auth**, **Raw**, and **XML**) in the request and response sections of the Details View pane to understand how you can get more information about requests and responses. For the initial request/response, the **TextView** and **WebView** of the response section show the redirection page that the server generates, which contains a hyperlink to the redirect location in case the browser does not support redirection (or it is disabled in the browser settings).
     1. 
  8. Go back to the main Fiddler window and double-click the second item in the list of requests (the first request to the simulated issuer). This is the browser responding to the Object Moved response it received from the previous request by navigating to the URL specified in the **Location** response header. The simulated issuer checks if the user is already authenticated, and - if not - sends back the **Adatum issuer** page you see in the browser. Click the **WebView** button in the response section of the Details View to see the contents of the response.
     1. 
  9. Back in the **Adatum issuer** page in the browser, ensure that the account **ADATUM\johndoe** is selected and click the **Continue with login** button. The a-Order application is shown with a list of orders and it indicates that the logged on user is **johndoe**.
  10. Look at the Fiddler main window. You will see another request to the simulated issuer, followed by two requests to the a-Order application.
      1. 
  11. Double-click the third item in the list of requests (the second request to the simulated issuer). This is the request initiated by clicking the **Continue with login** button in the **Adatum issuer** page. Click the **WebForms** button in the request section to see the decoded content of the values sent in the query string. Also notice that the response contains two cookies generated by the simulated issuer and sent to the browser. The first is the authentication cookie (**.WINAUTH**) that indicates authentication succeeded and the user is logged in as the user **ADATUM\johndoe**. The second (**AdatumClaimsRPStsSiteCookie**) contains a list of the relying parties (applications) for which this user is logged on. 
      1. The **.WINAUTH** cookie is used only by the simulated issuer to confirm that the user was authenticated. The browser sends this cookie only in requests it makes to the simulated issuer; it does not send it to the relying party applications a-Order and a-Expense.
      2. The **AdatumClaimsRPStsSiteCookie** has its path set to "/" and so it will be sent to all applications.
      3. The simulated issuer must also deliver the token containing the claims for the newly authenticated user to the browser so that the browser can submit it to the a-Order application. Because the token may be larger than the limit for query strings and headers, the browser must use an HTTP **POST** to pass the token to the application.
  12. Click the **TextView** button in the response section. You will see an encoded block of text. Click the yellow bar that indicates the response is encoded (shown in the previous screenshot) to decode it. You will see that it is a page containing an HTML form with the **action** attribute set to the URL of the a-Order application at **https://localhost/a-Order.Lab01.Ex01/**. The form contains a series of hidden controls containing values to submit to the a-Order application. One of the hidden controls contains the authentication token generated by the simulated issuer as an XML document, encoded so that it can be properly submitted to the server. The page also contains client-side script that automatically submits the form to the a-Order application.
      1. 
      2. If you click the **WebView** button you will see what this page looks like in a browser. The token and other values are not visible, but the page contains instructions and a submit button so that it will work with clients that have scripting disabled.
  13. Now double-click the fourth item in the main Fiddler window list of requests (the first request to the a-Order application after the two requests to the token issuer). This is the browser submitting the form you saw in the previous step to the a-Order application. The request headers section (click the **Headers** button if it is not shown) indicates that the browser submitted the cookie named **AdatumClaimsRPStsSiteCookie**. Click the **WebForms** button for the request to see the values sent in the HTML form controls. You can clearly see the authentication token.
      1. 
  14. Look at the response headers in the lower section of the Details View pane. You can see that this POST request resulted in an HTTP **302** (Found) response with the redirect location being the same URL (the a-Order application). The application sent back two new cookies (**FedAuth** and **FedAuth1**) to be stored on the browser. These contain the claims issued by the simulated issuer and the **ClaimsPrincipal** instance created by the WIF authentication module in the a-Order application pipeline.
      1. 
      2. This request to the a-Order application is captured by the WIF authentication module in the a-Order application pipeline. It receives the authentication token from the issuer and processes it to generate a **ClaimsPrincipal** instance. It then redirects the browser to the same URL (the default page in the a-Order application).
      3. The **FedAuth** cookies are generated by the relaying party application (in this case, a-Order) and are specific to this application. The browser will send these cookies only to the a-Order application, and not to the simulated issuer or any other application.
  15. In the Fiddler main window double-click the final request/response item for the a-Order application and look at the request headers. You can see that the browser sent the relying party list cookie (**AdatumClaimsRPStsSiteCookie**) that contains just the URL of the a-Order application. It also sent the two authentication cookies **FedAuth** and **FedAuth1**.
      1. 
  16. Click the **WebView** button in the response section. You will see the HTML page (the list of orders) sent to the browser. The WIF authentication module accepted the authentication cookies logged you into the application as **johndoe**.
      1. 
  17. In your web browser, reload (refresh) the a-Order application. In the main Fiddler request list window you see just a single request to the application, and no request to the token issuer (the other requests are for components of the page). The request headers section in the Fiddler Details View pane shows that the browser sent the **AdatumClaimsRPStsSiteCookie** and the existing valid **FedAuth** and **FedAuth1FedAuth** cookies, which the WIF authentication module validated and accepted. There was no requirement to re-authenticate.
      1. 
  18. In your web browser, navigate to the URL <https://localhost/a-Expense.Lab01.Ex01/> to open the a-Expense application. The a-Expense application opens to show a list of expenses, and indicates that the logged on user is **John Doe**. The single sign-on feature you added to the two applications in this exercise means that there is no requirement to re-authenticate.
  19. Look in the main Fiddler request list. It shows the request you made to the a-Expense application, followed by a request to the simulated issuer, and then two more requests to the a-Expense application.
      1. 
  20. Double-click the first request to the a-Expense application and look at the Details View pane. You will see that the browser sent the **AdatumClaimsRPStsSiteCookie** but not the **FedAuth** cookies. The path value set for these cookies means they are not valid for the a-Expense application. The response is an HTTP **302** (Found) code, and the redirect location is the URL of the simulated issuer. This is a similar process to that you saw with the a-Order application earlier in this task, although the values of the request parameters specify the URL and realm of the a-Expense application this time.
      1. 
  21. In the main Fiddler request list window double-click the request to the simulated issuer and examine the request information in the Details View pane. In the request section you will see that the browser sends the authentication cookie named **.WINAUTH** that it received from the simulated issuer when the user originally authenticated for the a-Order application. This means that the issuer does not need to reconfirm the user identity and so it does not display the page showing the list of available accounts. Notice that the browser also sends the **AdatumClaimsRPStsSiteCookie**, which currently contains only the URL of the a-Order application (the only application for which the user was previously authenticated).
      1. 
      2. **Note**: The **.WINAUTH** authentication cookie generated by the simulated issuer in these examples is not encrypted, so that you can see the account that logged on. However, in a real-world issuer it is vital to encrypt these cookies to prevent spoofing of the issuer whereby an attacker would attempt to convince the issuer that he or she had already been authenticated by submitting a non-valid cookie.
  22. Now look at the response section. You will see that, as with the a-Order application, the simulated issuer generates an HTML form containing the authentication token that will be submitted to the a-Expense application. The difference this time compared to opening the a-Order application is that the simulated issuer did not display the page containing the list of accounts (where the user would, in a real world token issuer, enter login credentials). Also notice that the simulated issuer updated the **AdatumClaimsRPStsSiteCookie** so that it contains both the a-Order and the a-Expense URLs because the user is now logged on to both of these applications.
      1. Recall from your examination of the code in the simulated issuer that it handles the **OnLoad** event and, for a sign-in action, checks if the user is already authenticated. If so, it just generates the authentication token containing the user's claims and creates the HTML form page to submit this token to the requesting application. The simulated issuer has no need to prompt for credentials because the original request contains the authentication token it previously issued, which is stored in the browser and submitted with all requests to the domain for which it is valid.
  23. The remaining two requests you see in the main Fiddler request list work in the same way as you saw for the a-Order application in previous steps of this task. Look at the details of the first one. It submits the HTML form returned from the issuer to the a-Expense application, which returns a **302** (Found) response that sets the two **FedAuth** cookies (which are specific to the a-Expense application). You can clearly see in this view the two **FedAuth** cookies that the WIF authentication handler in the a-Expense application pipeline generates and sends to the browser in the response. You can also see the two application URLs in the **AdatumClaimsRPStsSiteCookie** submitted with the request.
      1. 
  24. Now double-click the final request to the a-Expense application. This is the browser request caused by the previous **302** response. It is a **GET** request that also sends the two **FedAuth** cookies to the a-Expense application, which validates them and displays the page containing the list of expenses.
      1. 

1. You have now completed this task and this exercise. You have seen how you can easily modify applications that use non-claims-based authentication methods, such as Windows Authentication and ASP.NET Forms-based authentication, so that they use a trusted token issuer and identity provider for authentication. In many cases, as you saw in the example, there is no requirement to change the authorization code. It only requires changes to the authentication approach and implementation.

## Running the "End" Solution

1. If you did not complete all of the tasks in this exercise, you can run the "end" solution we provide.

To run the end solution

* 1. Start Visual Studio 2010 as an administrator.
  2. Open the solution named **Lab01.Ex01.End** from the folder **Lab01-SingleSignOn\Source\Ex01\End**.
  3. Open the **Build** menu and click **Rebuild Solution**.
  4. Open a web browser and navigate to the URL <https://localhost/a-Order.Lab01.Ex01.End/> to run the a-Order application.
  5. Select the user **ADATUM/johndoe** and click **Continue with login**.
  6. Navigate to the URL [https://localhost/a-Expense.Lab01.Ex01.End/](https://localhost/a-Order.Lab01.Ex01.End/) to run the a-Expense application.
  7. Use the links at the bottom of the pages to navigate between the applications. Notice that you only see the simulated issuer once; there is no requirement to re-authenticate in the other application.

# Exercise 2: Enabling Single Sign-Out

* 1. As you saw in the previous exercise, the a-Order and a-Expense applications now support claims-based authentication and single sign-on. If you sign in to one of the applications, you can access the other using the same identity that was previously validated by the STS (the simulated issuer) that authenticated your original request.
  2. However, the applications currently contain no sign-out capability. Signing out of an application that uses claims and Windows Identity Framework (WIF) is easy to implement by using code or a special control that is part of WIF. The WIF modules and the control hide the complexity of managing the tokens that are issued by the identity provider and the token issuer, but often there are other sign-out and clean-up tasks that your code must perform. In this exercise you will learn how single sign-out works, and how you can implement it in your own applications.

This exercise contains the following tasks:

* + [Task 1](#Ex02Task01): Modify the default simulated token issuer.
  + [Task 2](#Ex02Task02): Add sign-out capabilities to the applications and explore sign-out flow.
  + [Task 3](#Ex02Task03): Explore and update the simulated issuer to enable single sign-out.
  + [Task 4](#Ex02Task04): Add code to the a-Expense application to perform clean-up.

You should be able to complete this exercise in approximately 30 minutes.

## Task 1: Modify the Default Simulated Token Issuer

* 1. In this task you will modify the Adatum simulated issuer class so that you can explore the way that single-sign on operates as you work through the tasks in this exercise. As you complete these tasks, you will see how the token issuer enables single sign-on capabilities for applications that use Windows Identity Framework for authentication.

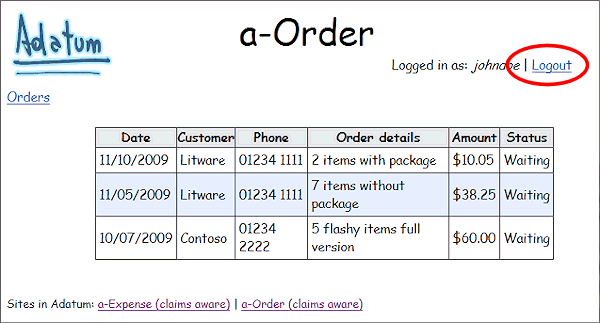
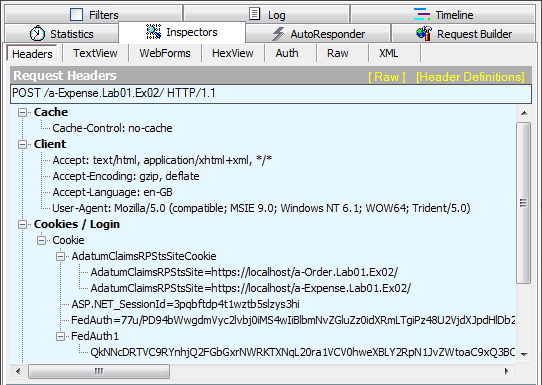
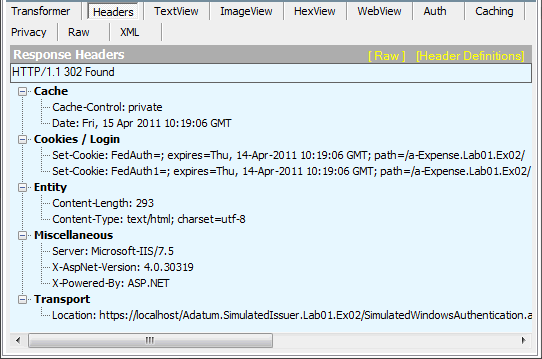
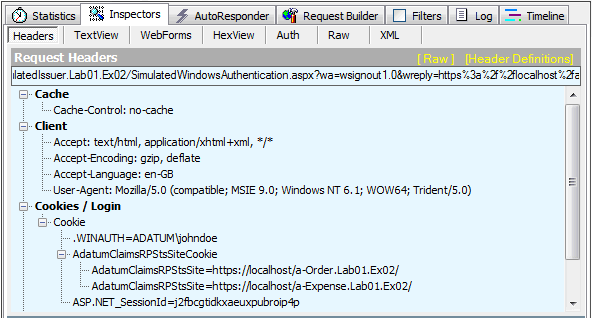
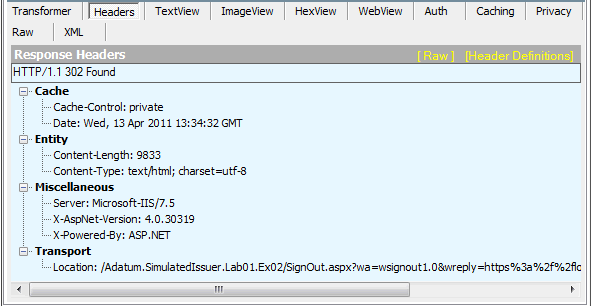
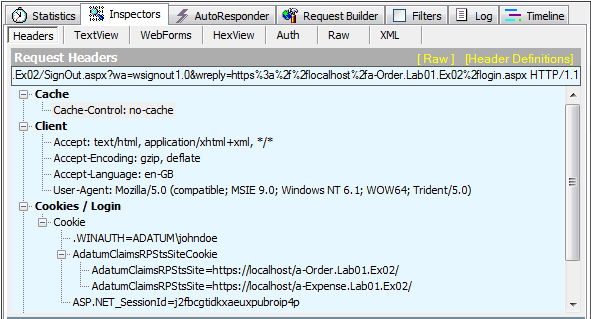
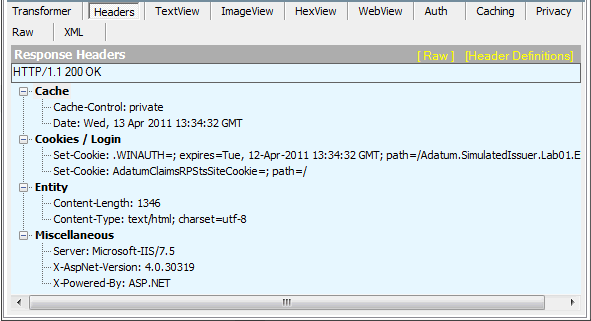
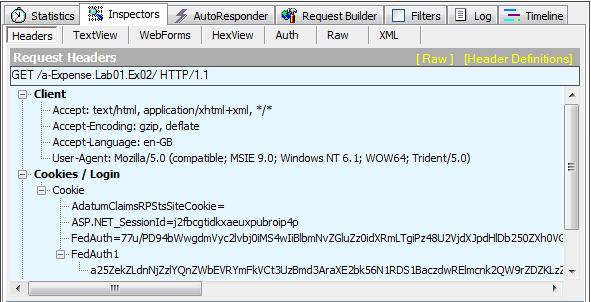
To modify the default simulated token issuer

* 1. Start Visual Studio as an administrator and open the solution named **Lab01.Ex02.sln** from the **Lab01-SingleSignOn\Source\Ex02\Begin** folder.
  2. In Solution Explorer expand the **Adatum.SimulatedIssuer.Lab01.Ex02** project, right-click on the file **SignOut.aspx**, and click **View Code**. This is the code-behind file for the page that a client requests when signing out of an application.
  3. Scroll to the end of the page and comment out the body of the method named **SignOutRelyingParties** as shown in the following highlighted code.
     1. C#
     2. private void SignOutRelyingParties()
     3. {
     4. **//**var signedInUrls = SingleSignOnManager.SignOutRelyingParties();
     5. **//**if (signedInUrls.Length > 0)
     6. **//**{
     7. **//** this.RelyingPartyLabel.Visible = true;
     8. **//** foreach (string url in signedInUrls)
     9. **//** {
     10. **//** this.RelyingPartySignOutLinks.Controls.Add(
     11. **//** new LiteralControl(string.Format("<p><a href='{0}'>...</p>", url)));
     12. **//** }
     13. **//**}
     14. }
     15. This project implements the simulated token issuer you've been using in this lab. The **SignOutRelyingParties** method is a core part of the mechanism for enabling single sign-out. In this exercise you will explore how it works, and what you must do in your applications to enable single-sign out.
  4. In Solution Explorer right-click the top-level **Lab01.Ex02** solution item and click **Rebuild solution**. Ensure that the solution builds without error.
  5. You have now completed this task. In the next task you will modify the a-Order and a-Expense applications to add sign-out capabilities to them, and explore how this works.

## Task 2: Add Sign-out Capabilities to the Applications and Explore Sign-out Flow

* 1. In this task you will modify the a-Order and a-Expense applications so that they support sign-out. You will start with the simplest case that requires only the WIF sign-out control to be inserted into the application page. You will then see an alternative approach that uses code to call the WIF module methods directly, instead of using the control. This is a more flexible approach better suited to types of applications other than ASP.NET Web Forms. Finally you will explore how the WIF sign-out mechanism interacts with the simulated issuer.

To add sign-out capabilities to the applications and explore sign-out flow

* 1. Continue with the **Lab01.Ex02** solution you used in the previous task.
  2. In Solution Explorer expand the **a-Order.Lab01.Ex02** project, open the **Styling** folder, and double-click the file **Site.Master** to open it in the Visual Studio editor pane. This is the master page for the web pages in the a-Order application.
  3. Add the following **Register** directive to the top of the page to register the WIF assembly and define an element prefix to use when inserting the WIF control into the page.
     1. HTML
     2. <%@ Register assembly="Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" namespace="Microsoft.IdentityModel.Web.Controls" tagprefix="idfx" %>
  4. Add the following highlighted code to the body of the page to insert the WIF **FederatedPassiveSignInStatus** control. Place it inside the **<div>** that contains the logged in user information, with a vertical bar to separate it from the logged in user name as shown here.
     1. HTML
     2. <div id="topuserinformation">
     3. Logged in as: <i><%=this.Context.User.Identity.Name%></i> **|**
     4. **<idfx:FederatedPassiveSignInStatus runat="server"**
     5. **ID="FederatedPassiveSignInStatus"**
     6. **SignOutText="Logout"**
     7. **FederatedPassiveSignOut="true"**
     8. **SignOutAction="FederatedPassiveSignOut" />**
     9. </div>
     10. Notice that the attributes of the **FederatedPassiveSignInStatus** control element allow you to specify the text for the link that the control will insert into the page, and the type of sign-out action that it should implement. Browser clients typically use the Passive sign-out method that takes advantage of browser redirection; as you saw at the end of Exercise 1 in this lab. You can use other attributes of the control to display an image instead of a text link, and to style the output of the control.
  5. Save and close the **Site.Master** file.
     1. Next, you will add a sign-out capability to the a-Expense application. However, instead of using the UI control you will add code that accesses the WIF module classes directly. This provides you with more flexibility and freedom to manage the sign-out process in applications that use other UI styles and architectures (such as MVC).
  6. In Solution Explorer expand the **a-Expense.Lab01.Ex02** project, expand the **Styling** folder, and double-click the file **Site.Master** to open it in the Visual Studio editor.
  7. Add the following highlighted definition of an ASP.NET **LinkButton** to the page, placing it after the code that displays the name of the currently logged on user as shown here. This will create a "Logout" link.
     1. HTML
     2. <div id="topuserinformation">
     3. Logged in as: <i><%= ((User)Session["LoggedUser"]).FullName %></i>
     4. (<%= this.NameClaim.OriginalIssuer %>) |
     5. **<asp:LinkButton ID="btnSignout" runat="server"**
     6. **onclick="btnSignout\_Click">Logout</asp:LinkButton>**
     7. </div>
  8. In Solution Explorer right-click on the **Site.Master** file in the **a-Expense.Lab01.Ex02** project and click **View Code** to open the code-behind file in the Visual Studio editor. Add the following namespace references to the end of the list of existing **using** statements in the file. These namespaces contain the WIF and ASP.NET authentication classes you will use in this page.
     1. C#
     2. using Microsoft.IdentityModel.Web;
     3. using System.Web.Security;
  9. Add the following event handler for the **LinkButton** to the **Site** class.
     1. C#
     2. protected void btnSignout\_Click(object sender, EventArgs e)
     3. {
     4. string signOutUrl =
     5. WSFederationAuthenticationModule.GetFederationPassiveSignOutUrl(
     6. FederatedAuthentication.WSFederationAuthenticationModule.Issuer,
     7. null, null);
     8. WSFederationAuthenticationModule.FederatedSignOut(new Uri(signOutUrl),
     9. new Uri("https://localhost/" + FormsAuthentication.LoginUrl));
     10. }
     11. This code performs the same tasks as the WIF sign-out control you used earlier in the a-Order application. The **GetFederationPassiveSignOutUrl** method creates a string that is the URL of the simulated issuer with the required parameters appended to request sign-out at the issuer. The code then calls the **FederatedSignOut** method to submit the sign-out request. Notice that the URL of the application's login page is obtained from configuration using the **FormsAuthentication** class, even though the login page name is not actually used by the simulated issuer in this example.
  10. Save and close the **Site.Master** and **Site.Master.cs** files. Then right-click the top-level **Lab01.Ex02** solution item in Solution Explorer and click **Rebuild solution**. Ensure that the solution builds without error.
  11. Open a web browser and navigate to <https://localhost/a-Order.Lab01.Ex02/> to run the a-Order application. You will see the Adatum simulated issuer login page.
  12. Ensure that **ADATUM\johndoe** is selected in the list of accounts and click **Continue with login**. You see the a-Order application showing a list of orders. The page now contains the **Logout** link generated by the WIF **FederatedPassiveSignInStatus** control.
      1. 
  13. Click the **a-Expense (claims aware)** link at the bottom of the page to confirm that you are now authenticated for both applications by the single sign-on mechanism. Then, in the a-Expense application, click the **a-Order (claims aware)** link to go back to the a-Order application.
  14. Open **Fiddler** from your **Start** menu. For information about installing and using Fiddler, see the section "*Pre-requisites for the Labs*" in the *Introduction* document for these labs.
  15. Back in your web browser click the **Logout** link in the a-Order application. You see the Adatum issuer again, but this time with a message that sign-out has been requested.
      1. 
      2. You may need click **Yes** in the Fiddler certificate warning dialog, and **Continue to this website** in your browser.
  16. In the Fiddler list of request/response items you will see a request to the a-Order application followed by two requests to the simulated issuer. Double-click the request to the a-Order application to view the details of the request and response. The headers for the request show that this is a **POST** request back to the a-Order application. It was initiated by the WIF control that implements the **Logout** link (you can click the **WebForms** tab in the request section to see the name of the logout control in the **EVENTTARGET** parameter).
      1. 
  17. Look at the headers in the response section. The WIF modules in the a-Order pipeline detect the post generated by the **Logout** link and return a **302** (Found) redirection response, with the location set to **https://localhost/Adatum.SimulatedIssuer.Lab01.Ex02/SimulatedWindowsAuthentication.aspx?wa=wsignout1.0&wreply=https%3a%2f%2flocalhost%2fa-Order.Lab01.Ex02%2flogin.aspx**.You can also see that the WIF module invalidates the two **FedAuth** cookies so that the user is no longer authenticated for this application.
      1. 
      2. The WIF sign-out control creates the redirection URL using values in the **<microsoft.identityModel>** section of the a-Order application's Web.config file. The **wa=signout1.0** parameter signifies that this is a sign-out request, and the **wreply** parameter value specifies the URL that the issuer should redirect to after logging out the user. The name of the login page (**login.aspx** in this example) is obtained from the Forms authentication section of Web.config. This allows the issuer, if required, to redirect the user back to the login page in the application. However, this is not mandatory; as you can see in the example in this task the simulated issuer just displays a message when the user logs out. In the real world, the action will vary depending on the issuer that is used.
  18. Double-click the first of the two requests to the simulated issuer. This is the request initiated by the previous **302** response from the a-Order application. The browser sends to the issuer the **.WINAUTH** cookie that confirms the user's identity, and the **AdatumClaimsRPStsSiteCookie** that contains the list of applications for which the user is authenticated.
      1. 
  19. Look at the response headers for this request. You will see that the response is a **302** (Found) redirection with the **Location** header set to the URL of the page **SignOff.aspx** within the issuer: **Adatum.SimulatedIssuer.Lab01.Ex02/SignOut.aspx?wa=wsignout1.0&wreply=https%3a%2f%2flocalhost%2fa-Order.Lab01.Ex02%2flogin.aspx**. This is the page that generates the response the simulated issuer will send back to the user after it completes the sign-out process.
      1. 
  20. In the Fiddler list of request/response items double-click the second request to the simulated issuer. This is the request for the page **SignOut.aspx** initiated by the previous **302** status code redirection. Again the **.WINAUTH** and **AdatumClaimsRPStsSiteCookie** cookies are sent with the request.
      1. 
  21. Now look at the response headers for this item. You can see that the issuer "cleans out" the **.WINAUTH** and **AdatumClaimsRPStsSiteCookie** cookies that are sent back to the browser by removing the values and setting the timestamp to a date and time in the past so that the browser does not store them.
      1. 
  22. Click the **WebView** tab and you will see the HTML sent back to the browser. This is the message that the user has been signed out. At this point, if required, the issuer could redirect the user to the application's login page instead using the value of the **wreply** parameter sent with the request for the **SignOut.aspx** page.
  23. Now go back to your browser and navigate to the URL <https://localhost/a-Order.Lab01.Ex02/> to run the a-Order application again. You see the Adatum issuer login page, which confirms that you have been signed out of the a-Order application. Do *not* click the login button.
  24. Look at the most recent request to the **a-Order** application in Fiddler. There are no **FedAuth** cookies that can authenticate you to the WIF module in this application's pipeline.
      1. The WIF sign-out control you added to the application removed these from the browser before it sent the sign-out request to the simulated issuer. Therefore your latest request to the a-Order application initiates a redirection to the simulated issuer. In addition, because your browser no longer contains a **.WINAUTH** cookie, you see the list of accounts and so would need to log on again.
  25. Now navigate to the URL <https://localhost/a-Expense.Lab01.Ex02/> to run the a-Expense application again. You will see that you are still logged into this application. It displays the list of expenses without redirecting to the simulated issuer and without requiring you to sign in again.
  26. Look at the most recent request to the **a-Expense** application in Fiddler. You will see that the browser still has the two **FedAuth** cookies that authenticate it to the WIF module in this application's pipeline, and these are sent to the application. You have not been signed out of the a-Expense application even though you signed out of the a-Order application.
      1. 
  27. You have now completed this task and you should understand how the sign-out process with WIF and a token issuer works. In particular, you should realize why most token issuers (STSs) support single sign-out in order to avoid the situation you have just seen where signing out of one application does not sign you out of both. In the next task you will see how the simulated issuer (and most other production issuers) implement single sign-out.

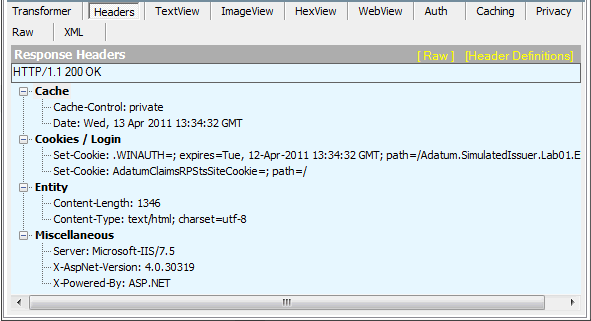
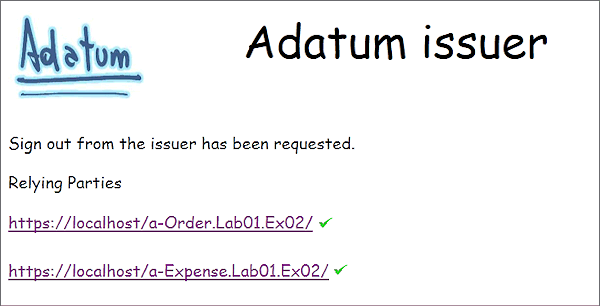
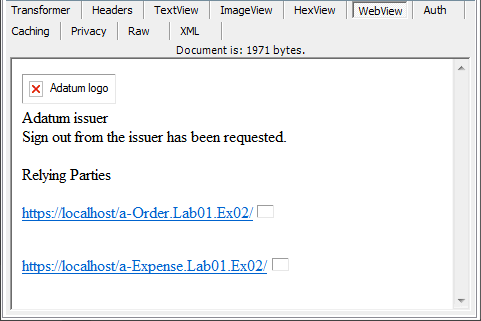
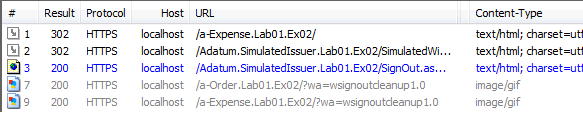
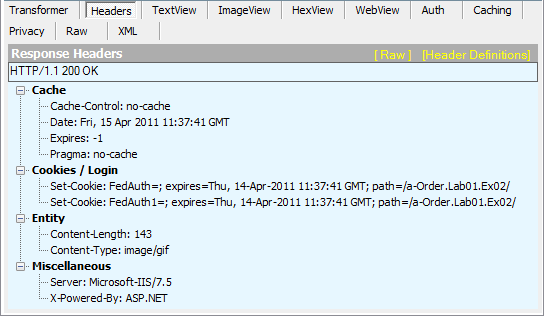
## Task 3: Explore and Update the Simulated Issuer to Enable Single Sign-out

* 1. In this task you will enable single sign-out so that, when a user signs out of one application where he or she was authenticated by the issuer, that user is automatically signed out of all other applications where the same issuer was used for authentication. For example, in the scenario of the examples you are working with, if you sign out of the a-Order application you will automatically be signed out of a-Expense.

You should understand that single sign-out is a session-based activity. If you open applications in different browsers that authenticate with the same issuer, single sign-out does not apply across these browsers. Recall that single sign-on for web browser-based applications uses tokens stored in the browser. Signing out in one browser will only sign you out of applications opened in that browser.

Also consider what users expect from single sign-out. When you sign out of an application, are you expecting to be signed out of all applications running the browser? You should only enable single sign-out if this actually is the case. What single sign-out really means is that you are signing out at the issuer, not at application level. After you sign out, the issuer will attempt to sign you out of all other applications for which it issued an authentication token within the current session. You will see how this is accomplished in this task.

To explore update the simulated issuer to enable single sign-out

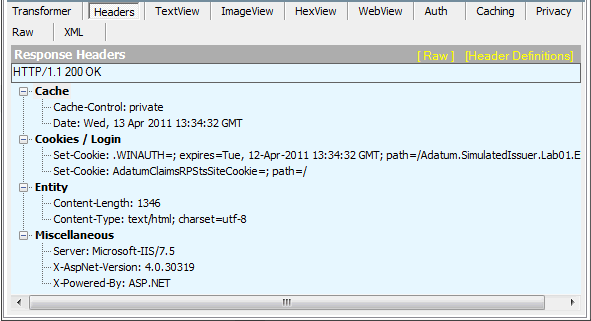
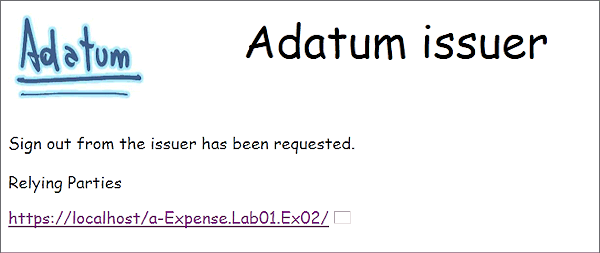
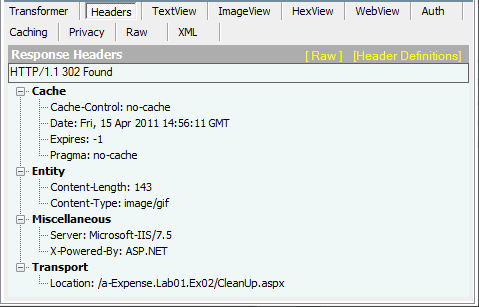
* 1. Continue with the **Lab01.Ex02** solution you used in the previous task.
  2. In Solution Explorer expand the **Adatum.SimulatedIssuer.Lab01.Ex02** project, right-click on **SimulatedWindowsAuthentication.aspx**, and click **View Source** to open the main simulated issuer page in the Visual Studio editor.
  3. Look at the **OnLoad** event handler. As you saw in the previous exercise when you examined the simulated issuer, this code checks the value of the **Action** parameter sent in the request from the client. The following code in the **OnLoad** method redirects the request to the page **SignOut.aspx** if the **Action** is either **SignOut** or **SignOutCleanup**.
     1. C#
     2. ...
     3. else if (action == WSFederationConstants.Actions.SignOut
     4. || action == WSFederationConstants.Actions.SignOutCleanup)
     5. {
     6. this.Response.Redirect("~/SignOut.aspx?" + this.Request.QueryString, false);
     7. }
     8. ...
  4. In Solution Explorer right-click on **SignOut.aspx** and click **View Source**. The **OnLoad** event handler of this page also checks that this is a sign-out request and, if so, calls some methods to perform the sign-out:
     + It calls the method **LogOutUser** of the **SimulatedWindowsAuthenticationOperations** class to invalidate the user's **.WINAUTH** cookie by setting the expiry time to the previous day.
     + It uses the WIF classes to generate the request that will sign out the user, and displays a message in the page to indicate that sign-out is taking place.
     + It calls the method named **SignOutRelyingParties**, which you commented out in the first task of this exercise.
     + It calls the **Clear** method of the **SingleSignOnManager** helper class to remove all cookies that apply to this issuer and to all relying parties that authenticated with this issuer.
     1. The Fiddler trace you saw in the previous task of this exercise (shown again here) indicates the effect of this sign-out code. The **Set-Cookie** headers invalidate the **.WINAUTH** cookie and clear the **AdatumClaimsRPStsSiteCookie**.
     2. X
  5. Scroll down to the **SignOutRelyingParties** method in the **SignOut.aspx.cs** file. This method is called before the code you just examined clears the cookies. Un-comment all of the commented lines of this method, right-click on the call to the **SignOutRelyingParties** method of the **SingleSignOnManager** class, and click **Go To Definition**.
  6. Look at the code in the **SignOutRelyingParties** method of the **SingleSignOnManager** class (the relevant section is shown below). It simply returns a string array containing the URLs of all of the relying parties that have authenticated with this issuer.
     1. C#
     2. ...
     3. HttpCookie siteCookie =
     4. HttpContext.Current.Request.Cookies[RelyingPartySiteCookieName];
     5. if (siteCookie != null)
     6. {
     7. var urls = siteCookie.Values.GetValues(RelyingPartySiteName);
     8. return urls ?? new string[0];
     9. }
     10. ...
  7. Back in the **SignOut.aspx.cs** file look at the remainder of the code in the **SignOutRelyingParties** method (shown below; but note that we had to split the string literal over three lines here due to page width limitations).
     1. C#
     2. private void SignOutRelyingParties()
     3. {
     4. var signedInUrls = SingleSignOnManager.SignOutRelyingParties();
     5. if (signedInUrls.Length > 0)
     6. {
     7. this.RelyingPartyLabel.Visible = true;
     8. foreach (string url in signedInUrls)
     9. {
     10. this.RelyingPartySignOutLinks.Controls.Add(
     11. new LiteralControl(string.Format("<p><a href='{0}'>{0}</a>&nbsp;<img
     12. src='{0}?wa=wsignoutcleanup1.0' title='Signout request:
     13. {0}?wa=wsignoutcleanup1.0'/></p>", url)));
     14. }
     15. }
     16. }
  8. You can see that this code generates a list of the relying party URLs obtained from the **AdatumClaimsRPStsSiteCookie**. For each one it adds an **<image>** element with the **src** attribute set to the URL of the relying party and the query string **wa=wsignoutcleanup1.0**. This list of URLs and images is inserted into the UI content of the **SignOut.aspx** page
  9. Right-click the top-level **Lab01.Ex02** solution item in Solution Explorer and click **Rebuild solution**. Ensure that the solution builds without error.
  10. Open a web browser and navigate to <https://localhost/a-Order.Lab01.Ex02/> to run the a-Order application. You will see the Adatum simulated issuer login page. Click **Continue with login** to see the list of orders in the a-Order page.
  11. Use the link at the bottom of the a-Order page to navigate to the a-Expense application. You are automatically signed in by the single sign-on mechanism, and you see the list of expenses.
  12. Open **Fiddler** from your **Start** menu. For information about installing and using Fiddler, see the section "*Pre-requisites for the Labs*" in the *Introduction* document for these labs.
  13. In Fiddler open the **Rules** menu and make sure that **Hide Image Requests** is *not* checked. You will need to see traces for image files in this task.
  14. Back in your web browser click the **Logout** link in the a-Expense application. You see the Adatum issuer again with the message that sign-out has been requested. You can also see the list of relying parties that were stored in the **AdatumClaimsRPStsSiteCookie**, and the "tick" images in the **<image>** elements that the **SignOut.aspx** page in the issuer added for each relying party.
      1. 
      2. You may need click **Yes** in the Fiddler certificate warning dialog, and **Continue to this website** in your browser.
  15. Look at the list of requests and responses in Fiddler. As you saw in the previous task of this exercise, the logout process initiates a **POST** request to the a-Expense application followed by two requests to the simulated issuer. The request to the a-Expense application was initiated by the WIF control that implements the **Logout** link, and the response created by the WIF modules in the a-Expense pipeline returns a **302** (Found) response that redirects the browser to the simulated issuer with the **wa=wsignout1.0** parameter in the query string. The WIF module also invalidates the two **FedAuth** cookies so that the user is no longer authenticated for this application.
  16. Double-click on the first request to the simulated issuer and confirm that this is the same as you saw in the previous task. The browser sends to the issuer the **.WINAUTH** cookie that confirms the user's identity, and the **AdatumClaimsRPStsSiteCookie** that contains the list of applications for which the user is authenticated. The simulated issuer responds with a **302** (Found) redirection with the **Location** header set to the URL of the page **SignOut.aspx** within the issuer.
  17. Double-click the second request to the simulated issuer. This is the request for the page **SignOut.aspx** initiated by the previous **302** status code redirection. Again the **.WINAUTH** and **AdatumClaimsRPStsSiteCookie** cookies are sent with the request; and, as in the previous task, the response headers show that the issuer invalidates the **.WINAUTH** and **AdatumClaimsRPStsSiteCookie** cookies that are sent back to the browser.
  18. Now click the **WebView** tab to see the HTML sent back to the browser. This shows the list of relying parties, just as you saw in the browser earlier.
      1. 
  19. Click the **TextView** tab to see the actual HTML code for this response. It contains no line-breaks and so is hard to assimilate. However, the following listing shows the content with line breaks added so you can more easily see what it contains. Notice that the **src** of the mages specifies a file in each of the relying party applications, and *not* files within the simulated issuer. The URL of the image also has the **wa=wsignoutcleanup1.0** parameter in the query string.
      1. HTML
      2. <p>
      3. <a href='https://localhost/a-Order.Lab01.Ex02/'>
      4. https://localhost/a-Order.Lab01.Ex02/
      5. </a>&nbsp;
      6. <img src='**https://localhost/a-Order.Lab01.Ex02/?wa=wsignoutcleanup1.0**'
      7. title='Signout request: https://localhost/a-
      8. Order.Lab01.Ex02/?wa=wsignoutcleanup1.0'/>
      9. </p>
      10. <p>
      11. <a href='https://localhost/a-Expense.Lab01.Ex02/'>
      12. https://localhost/a-Expense.Lab01.Ex02/
      13. </a>&nbsp;
      14. <img src='**https://localhost/a-Expense.Lab01.Ex02/?wa=wsignoutcleanup1.0**'
      15. title='Signout request: https://localhost/a-
      16. Expense.Lab01.Ex02/?wa=wsignoutcleanup1.0'/>
      17. </p>
      18. This is the mechanism by which STSs and token issuers can call back into every relying party application that has previously authenticated with this issuer during the current session. As you saw earlier, the list of authenticated relying parties comes from the cookie named **AdatumClaimsRPStsSiteCookie** that the issuer updates each time an application authenticates.
  20. To find out where the "tick" images come from, look at the list of requests and responses in Fiddler. Immediately after the two requests to the simulated issuer you will see requests for images that return responses of type **image/gif** for each relying party application (a-Order application and a-Expense).
      1. 
      2. Internet Explorer 9 may show two requests for each relying party application image if a request is initially aborted. This does not affect the working of the sign-out mechanism because all of the authentication cookies are invalidated with the first response.
  21. Click the first image request to the a-Order application and, in the response details section of Fiddler, click the **ImageView** tab. You will see the "tick" image. Click the **Headers** tab to see that the content type is set to **image/gif**, and that the response headers invalidate the two **FedAuth** cookies that are sent with the request (you can see the cookies in the request headers section). Recall that you logged out of the a-Expense application and so, up till this point, you were still logged into the a-Order application.
      1. 
      2. By default the WIF authentication module generates and returns the "tick" image automatically when it intercepts a request that contains the **wa=wsignoutcleanup1.0** parameter.
  22. The next request in the list is sent to the a-Expense application, and again the response is the "tick" image. However, if you examine the request headers you can see that there are no **FedAuth** cookies. The WIF modules in the a-Expense application issuer invalidated these before it sent the logout request to the issuer.
  23. Go back to your web browser and click the link in the Adatum issuer's sign-out page for the a-Order application. You will see the list of users and the **Continue with login** button, which confirms that you have been signed out of the a-Order application even though you only actually logged out of the a-Expense application.
  24. If you are using Internet Explorer 8 to view the example web pages, you may find that it caches pages after you log out of the applications so that it appears you are still logged on. To resolve this you can add directives to the applications to prevent Internet Explorer caching pages. You can add the following ASP.NET directive immediately after the existing **<%@Page ...>** directive in the Default.aspx pages:
  25. **<%@ OutputCache Location="None" %>**
  26. Alternatively, add all of the following META elements to any ASP.NET or HTML page:
  27. **<META Http-Equiv="Cache-Control" Content="no-cache">**
  28. **<META Http-Equiv="Pragma" Content="no-cache">**
  29. **<META Http-Equiv="Expires" Content="0">**
  30. You have now completed this task and you have seen how the combination of code in the issuer and the WIF modules in the application pipeline work together to sign a user out of all of the relying party applications for which that the user was authenticated by the issuer.
  31. When single-sign on is enabled, a token issuer will attempt to sign users off in every previously-authenticated relying party application; but this may not always succeed. Bear in mind that there is no guarantee that an application that originally authenticated (and is listed in the **AdatumClaimsRPStsSiteCookie**) will still be available or respond to sign-off requests. To minimize the risk of users still being authenticated, the issuer and the WIF mechanism must work together.
  32. In response to a sign-out request initiated within a client application by the WIF sign-out control, or by code that executes the **FederatedSignOut** method of the **WSFederationAuthenticationModule**, the WIF modules in that application pipeline first invalidate the **FedAuth** cookies on the client so that the user cannot access the application again.
  33. Next, the WIF mechanism sends a sign-out request to the token issuer. Instead of attempting to initiate the sign-out action on each relying party in turn, issuer generates a page containing an image for each relying party where the source of each image is the URL of the corresponding relying party application's sign-out page.
  34. If the application responds by sending back an image, the issuer sign-out page shows the image. If not, there is no image in the page for this relying party. This also means that an application can send back a different image if (for some other reason) it didn't manage to log off the user.

However, there is one more issue to consider, and you will investigate this in the next and final task of this exercise.

## Task 4: Add Code to the a-Expense Application to Perform Clean-up

* 1. In the previous tasks you have implemented single sign-out for the a-Order and a-Expense applications. The a-Order application contains no information in the ASP.NET session or any other store that requires additional clean-up when a user signs out, and so the simple approach of using the WIF sign-out control is sufficient to accomplish single sign-out.
  2. However, the a-Expense application requires some additional attention because, unlike the a-Order application, the a-Expense application stores the currently authenticated user instance in the ASP.NET session. This means that simply using the WIF module to initiate a sign-out and destroy the **FedAuth** authentication cookies is not sufficient. You must also clear the ASP.NET session, and you will see how to do that in this task.

To add code to the a-Expense application to perform clean-up

* 1. Continue with the **Lab01.Ex02** solution you used in the previous task.
  2. In Solution Explorer, expand the **a-Expense.Lab01.Ex02** project item, double-click the file **Global.asax** to open the code in the Visual Studio editor, and look at the end of the **Session\_Start** method. You can see that the a-Expense application saves the currently logged on user in the ASP.NET session. When the user signs out you must remove this object from the session.
  3. The WIF authentication module in the a-Expense application's pipeline raises the **SignedOut** event after a user is signed out (when the WIF module receives a request that contains the **wa=wsignoutcleanup1.0** parameter). You can handle this event to clean up the session and any other resources. Add the following code to the end of the **Global** class defined in the **Global.asax.cs** file to redirect the request to another page named **CleanUp.aspx** when the **SignedOut** event is raised.
     1. C#
     2. private void WSFederationAuthenticationModule\_SignedOut(object sender, EventArgs e)
     3. {
     4. Response.Redirect("~/CleanUp.aspx", false);
     5. }
     6. When using the WIF **FederatedPassiveSignInStatus** control, as you did in the a-Order application, you can specify an event handler using the **SignOutAction** attribute of the control. This is the equivalent of handling the **SignedOut** event of the **WSFederationAuthenticationModule**.
     7. You could, of course, simply write code in the event handler method to clean up the application and remove the user from the session, then display a suitable message. However, with many issuers this redirection is necessary to properly implement single sign-out because your application must replicate the behavior of the default WIF authentication module action. In other words, when it receives a request from the issuer that contains the **wa=wsignoutcleanup1.0** parameter, it must return an image that indicates sign out and clean-up was completed.
  4. Now you will create the **CleanUp.aspx** page. In Solution Explorer right-click the **a-Expense.Lab01.Ex02** project item, point to **Add**, and click **New Item**. In the Add New Item dialog select **Web** in the list if Installed Templates, select **Web Form** in the central list, and change the name to **CleanUp.aspx**.
     1. 
  5. Click **Add**. The new page opens in the Visual Studio editor. Delete all of the content except for the **<%@Page...>** directive at the top of the page.
  6. In Solution Explorer right-click **CleanUp.aspx** and click **View Code** to open the code-behind file for this page. Visual Studio automatically includes in the code an outline handler method for the **Page\_Load** event. Add the following highlighted code to this method to abandon the current ASP.NET session (which removes the stored user instance and any other data in the session).
     1. C#
     2. protected void Page\_Load(object sender, EventArgs e)
     3. {
     4. **if (this.User.Identity.IsAuthenticated)**
     5. **{**
     6. **this.Response.Redirect("~/Default.aspx", false);**
     7. **}**
     8. **else**
     9. **{**
     10. **this.Session.Abandon();**
     11. **}**
     12. }
  7. Right-click the top-level **Lab01.Ex02** solution item in Solution Explorer and click **Rebuild solution**. Ensure that the solution builds without error.
  8. Open a web browser and navigate to the URL <https://localhost/a-Expense.Lab01.Ex02/>. You are redirected to the Adatum issuer. Click **Continue with login**.
  9. Open **Fiddler** from your **Start** menu. Open the **Rules** menu and make sure that **Hide Image Requests** in *not* checked. You will need to see traces for image files in this task.
  10. Back in your web browser click the **Logout** link in the a-Expense application. You see the Adatum issuer again with the message that sign-out has been requested. As you only signed into the a-Expense application, this is the only one listed. However, notice that there is no "tick" image this time.
      1. 
  11. Look at the list of requests and responses in Fiddler. After the second request to the simulated issuer is the request for the image file specified in the **src** attribute of the **<image>** element in the page generated by the issuer. Look at the response headers and you will see that the a-Expense application returned a **302** (Found) status with the redirect location **CleanUp.aspx** (your new clean-up page that clears the ASP.NET session).
      1. 
  12. Now look at the following request/response item. The code you added to the **CleanUp.aspx.cs** file returns nothing (look in the **TextView** and **ImageView** tabs in the response section to confirm this). However, the issuer is expecting the "tick" image that WIF would have generated if you hadn't interrupted the process by handling the **SignedOut** event.
  13. Go back to Visual Studio and add the following code to the **Page\_Load** event handler. This code generates the standard "tick" GIF image, turns off proxy server caching for the response, clears any content from the response, sets the required content type, and the writes the image bytes to the response.
      1. C#
      2. protected void Page\_Load(object sender, EventArgs e)
      3. {
      4. if (this.User.Identity.IsAuthenticated)
      5. {
      6. this.Response.Redirect("~/Default.aspx", false);
      7. }
      8. else
      9. {
      10. this.Session.Abandon();
      11. **var signOutImage = new byte[]**
      12. **{**
      13. **71, 73, 70, 56, 57, 97, 17, 0, 13, 0, 162, 0, 0, 255, 255, 255,**
      14. **169, 240, 169, 125, 232, 125, 82, 224, 82, 38, 216, 38, 0, 0, 0, 0,**
      15. **0, 0, 0, 0, 0, 33, 249, 4, 5, 0, 0, 5, 0, 44, 0, 0,**
      16. **0, 0, 17, 0, 13, 0, 0, 8, 84, 0, 11, 8, 28, 72, 112, 32,**
      17. **128, 131, 5, 19, 22, 56, 24, 128, 64, 0, 0, 10, 13, 54, 116, 8,**
      18. **49, 226, 193, 1, 4, 6, 32, 36, 88, 113, 97, 0, 140, 26, 11, 30,**
      19. **68, 8, 64, 0, 129, 140, 29, 5, 2, 56, 73, 209, 36, 202, 132, 37,**
      20. **79, 14, 112, 73, 81, 97, 76, 150, 53, 109, 210, 36, 32, 32, 37, 76,**
      21. **151, 33, 35, 26, 20, 16, 84, 168, 65, 159, 9, 3, 2, 0, 59**
      22. **};**
      23. **this.Response.Cache.SetCacheability(HttpCacheability.NoCache);**
      24. **this.Response.ClearContent();**
      25. **this.Response.ContentType = "image/gif";**
      26. **this.Response.BinaryWrite(signOutImage);**
      27. }
      28. }
  14. Save and close the **CleanUp.aspx.cs** file, then right-click the top-level **Lab01.Ex02** solution item in Solution Explorer and click **Rebuild solution**. Ensure that the solution builds without error.
  15. Go back to your web browser and click the link in the Adatum issuer sign-out page to navigate to the URL <https://localhost/a-Expense.Lab01.Ex02/>. You are redirected to the Adatum issuer. Click **Continue with login**.
  16. In the a-Expense application, click the **Logout** button. Now you see the "tick" image generated by the **CleanUp.aspx** page after the URL of the a-Expense application in the issuer's sign-out page.
  17. You have now completed this task and this exercise. You have seen how single sign-out works through a combination of features implemented in the STS (the simulated issuer in these exercises) and the WIF modules in the relying party application pipelines.
  18. You have also seen how you can write code that performs clean-up, such as abandoning the ASP.NET session, in an application. You can include code that performs other operations such as closing connections or removing data from a local cache if this is required.
  19. Single sign-out is a powerful feature that is implemented in most commercial STSs, and so it is vital to understand how you can use it. However, remember that you must also consider when it is appropriate. In some scenarios, users may want only to log out of the current application and not all open applications.
  20. If you are using Internet Explorer 8 to view the example web pages, you may find that it caches pages after you log out of the applications so that it appears you are still logged on. To resolve this you can add directives to the applications to prevent Internet Explorer caching pages. You can add the following ASP.NET directive immediately after the existing **<%@Page ...>** directive in the Default.aspx pages:
  21. **<%@ OutputCache Location="None" %>**
  22. Alternatively, add all of the following META elements to any ASP.NET or HTML page:
  23. **<META Http-Equiv="Cache-Control" Content="no-cache">**
  24. **<META Http-Equiv="Pragma" Content="no-cache">**
  25. **<META Http-Equiv="Expires" Content="0">**

## Running the "End" Solution

1. If you did not complete all of the tasks in this exercise, you can run the "end" solution we provide.

To run the end solution

* 1. Start Visual Studio 2010 as an administrator.
  2. Open the solution named **Lab01.Ex02.End** from the folder **Lab01-SingleSignOn\Source\Ex02\End**.
  3. Open the **Build** menu and click **Rebuild Solution**.
  4. Open a web browser and navigate to the URL <https://localhost/a-Order.Lab01.Ex02.End/> to run the a-Order application.
  5. Select the user **ADATUM/johndoe** and click **Continue with login**.
  6. Use the links at the bottom of the pages to navigate between the applications. Notice that you only see the simulated issuer once; there is no requirement to re-authenticate in the other application.
  7. Click the **Logout** link in the **a-Order** application. You see the simulated issuer log-out page which indicates the applications for which the issuer has signed you out.
  8. Navigate to the URL <https://localhost/a-Expense.Lab01.Ex02.End/> again to conform that you need to re-authenticate.

# Exercise 3: Using WIF Session Mode

* 1. In this exercise you will modify the a-Expense application to change the behavior of the WIF modules so that token information is stored in the session instead of in the authentication cookie. By default, WIF encodes all of the claims information received from the issuer in the authentication cookie that it passes to the client to indicate that the client is authenticated. The browser stores this cookie and sends it to the application with every request.
  2. However, the authentication token can become quite large, especially if it contains many claims. This requires additional bandwidth, and may even exceed the limit on the size of cookies that browsers will accept. If the browser fails to store and return the cookie, the application will repeatedly prompt the user to sign in.
  3. To resolve this issue you can instruct WIF to store the claims information in the application's session and send to the client only a small cookie that identifies the stored information. You will see how to enable WIF session mode in this exercise.

This exercise contains the following tasks:

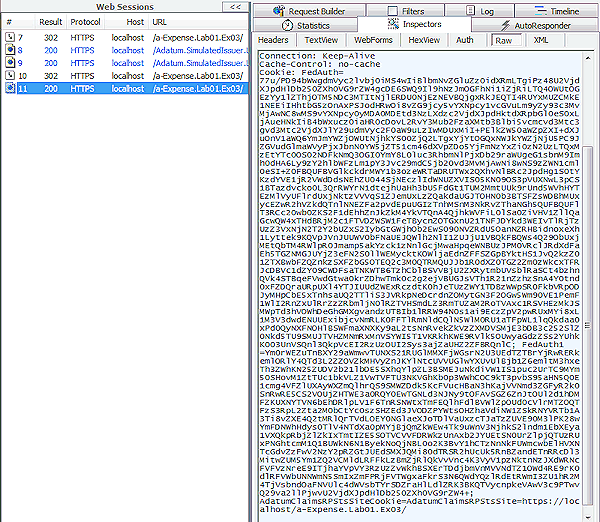
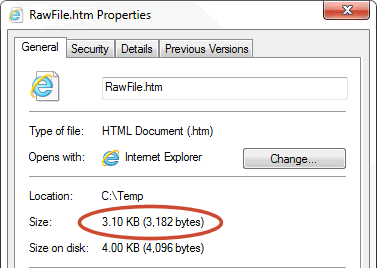
* + [Task 1](#Ex03Task01): Discover the default cookie size
  + [Task 2](#Ex03Task02): Modify the applications to enable session mode

You should be able to complete this exercise in approximately 20 minutes.

## Task 1: Discover the Default Cookie Size

In this task you will investigate the contents of the headers sent to the a-Expense application after authentication and discover the size of the default WIF authentication cookie for this application.

To discover the default cookie size

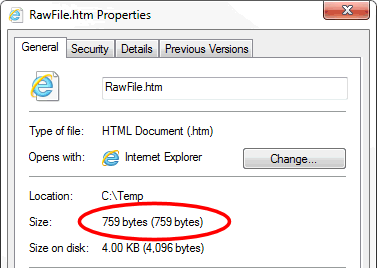
* 1. Start Visual Studio 2010 as an administrator.
  2. Open the solution named **Lab01.Ex03** from the folder **Lab01-SingleSignOn\Source\Ex03\Begin**.
  3. Open the **Build** menu and click **Rebuild Solution**.
  4. Open **Fiddler** from your **Start** menu. For information about installing and using Fiddler, see the section "*Pre-requisites for the Labs*" in the *Introduction* document for these labs.
  5. In Fiddler open the **Rules** menu and make sure that **Hide Image Requests** is checked. You will not need to see traces for image files in this task.
  6. Open a web browser and navigate to the URL <https://localhost/a-Expense.Lab01.Ex03/> to run the a-Expense application.
     1. You may need click **Yes** in the Fiddler certificate warning dialog, and **Continue to this website** in your browser.
  7. Select the user **ADATUM/johndoe** and click **Continue with login**.
  8. In the Fiddler list of requests/responses select the last request to the a-Expense application (ignore any requests for the style sheet or other resources if they appear in the list). In the request headers section click the **Raw** tab, right-click inside the text box containing the header information, and turn off **AutoTruncate** to see the entire contents of the **FedAuth** cookie.
     1. 
  9. Click the **View in Notepad** button below the text box. Notepad opens containing the entire content of the headers. Delete everything except the value of the **FedAuth** cookie and then save the file to a temporary folder on your computer.
  10. In Windows Explorer, right-click the file you just saved and click **Properties**. You will see is it over 3 KB in size, and yet it only contains a few simple text claim values.
      1. 

You have now completed this task. In the next task you will modify the a-Expense application so that it stores the claims information in the session instead of in the **FedAuth** cookie.

## Task 2: Modify the a-Expense Application to Enable Session Mode

In this task you will modify the a-Expense application so that it stores the claims information in the session instead of in the **FedAuth** cookie. This requires only a simple modification to the Global.asax file. You will then examine the authentication cookie again to see the difference this makes.

To modify the a-Expense application to enable session mode

* 1. Continue with the **Lab01.Ex03** solution you used in the previous task.
  2. In Solution Explorer expand the **a-Expense.Lab01.Ex03** project and double-click **Global.asax** to open its code file in the Visual Studio editor pane.
  3. Scroll to the end of file and add the following highlighted code to the **Global** class.
     1. C#
     2. private void WSFederationAuthenticationModule\_SignedOut(object sender,
     3. EventArgs e)
     4. {
     5. Response.Redirect("~/CleanUp.aspx", false);
     6. }
     7. **void WSFederationAuthenticationModule\_SessionSecurityTokenCreated(**
     8. **object sender, SessionSecurityTokenCreatedEventArgs e)**
     9. **{**
     10. **FederatedAuthentication.SessionAuthenticationModule.IsSessionMode = true;**
     11. **}**
     12. }
     13. }
     14. This code handles the event that the WIF exposes after creating the security token that will, by default, be stored in the **FedAuth** cookie. It instructs WIF to store the claims information in the session instead of the cookie, and then store only a key to the session information in the **FedAuth** cookie.
  4. Save and close the **Global.asax.cs** file. Then open the **Build** menu and click **Rebuild Solution**. Ensure that it builds without error.
  5. Make sure that you have logged out of the a-Expense application by clicking the **Logout** link. Then open the "properties" or "options" dialog of your web browser and delete all of the stored cookies. Now close your web browser. This ensures that the existing **FedAuth** cookie is destroyed.
     1. In Internet Explorer, you delete the stored cookies by opening the **Tools** | **Internet options** menu and clicking the **Delete** button on the **General** tab. Ensure that **Cookies** is selected in the list of items to delete. Note that you may have to repeat this process more than once to force the WIF modules to change mode after your modify the application.
  6. Open your web browser again and navigate to the URL <https://localhost/a-Expense.Lab01.Ex03/>. Select the user **ADATUM/johndoe** and click **Continue with login**.
  7. In Fiddler select the last request to the a-Expense application, click the **Raw** tab, right-click inside the text box containing the header information, and click **AutoTruncate**.
  8. Click **View in Notepad**, delete everything except the value of the **FedAuth** cookie in the new document, and then save the file to a temporary folder on your computer.
  9. In Windows Explorer, right-click the file you just saved and click **Properties**. You will see that is it now only a few hundred bytes.
     1. 

You have now completed this task and this exercise. You have seen how you can reduce the size of the **FedAuth** cookie that is stored on the client and passed over the network with each request to the application.

* 1. Note that if you use WIF session mode on a web farm or when you have multiple role instances running in Windows Azure you must either ensure that requests are sent to the same instance that originally authenticated the user, or use a session store that is suitably configured for web farm scenarios. You can also extend the WIF **TokenCache** class for these kinds of scenarios. For more details about using WIF Session Mode, see the blog post "*Your FedAuth Cookies on a Diet: IsSessionMode=true*" at <http://blogs.msdn.com/b/vbertocci/archive/2010/05/26/your-fedauth-cookies-on-a-diet-issessionmode-true.aspx>.

## Running the "End" Solution

1. If you did not complete all of the tasks in this exercise, you can run the "end" solution we provide.

To run the end solution

* 1. Start Visual Studio 2010 as an administrator.
  2. Open the solution named **Lab01.Ex03.End** from the folder **Lab01-SingleSignOn\Source\Ex03\End**.
  3. Open the **Build** menu and click **Rebuild Solution**.
  4. Close your web browser if it is still open. This ensures that the existing **FedAuth** cookie is destroyed.
  5. Start Fiddler from your Start menu.
  6. Open your web browser again and navigate to the URL <https://localhost/a-Expense.Lab01.Ex03.End/>. Select the user **ADATUM/johndoe** and click **Continue with login**.
  7. In Fiddler select the last request to the a-Expense application, click the **Raw** tab, right-click inside the text box containing the header information, and turn off **AutoTruncate**.
  8. Click **View in Notepad**, delete everything except the value of the **FedAuth** cookie in the new document, and then save the file to a temporary folder on your computer.
  9. In Windows Explorer, right-click the file you just saved and click **Properties**. You will see that is it now only a few hundred bytes.

# Exercise 4: Publishing a Claims-Aware Application to Windows Azure

* 1. In this exercise you will see how to modify an application that uses claims-based authentication so that it will run in Windows Azure. To do this, you must modify the application so that it sends its URL to the issuer using the **wreply** parameter of the WS-Federation request. This is necessary because the example applications for this lab use the Adatum issuer running on **localhost**. When the application is running in Windows Azure it will have a different URL, and the issuer requires this URL to be able to redirect the browser to the application after it authenticates with the issuer.
  2. You do not require a Windows Azure account to perform this exercise because you will deploy the application only to the local Visual Studio Azure Development Fabric.

This exercise contains the following tasks:

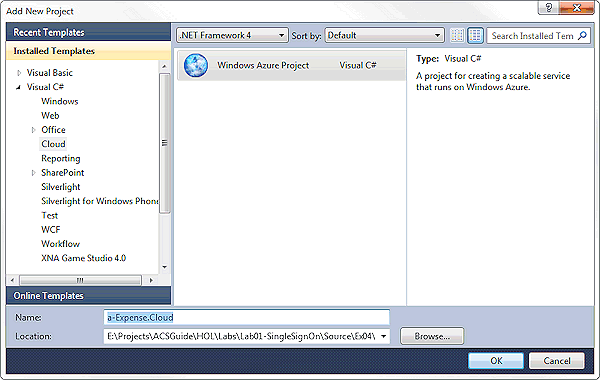
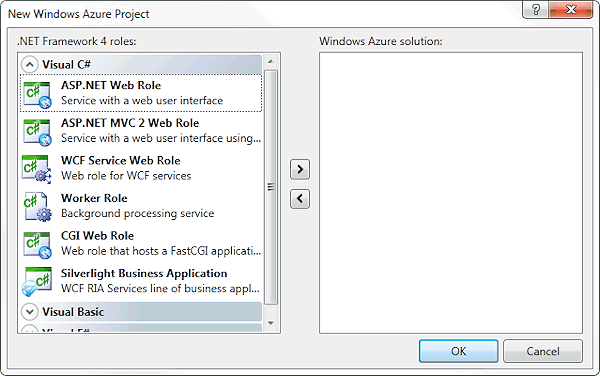
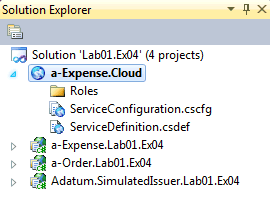
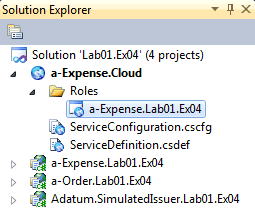
* + [Task 1](#Ex04Task01): Add a Windows Azure project to the example solution
  + [Task 2](#Ex04Task02): Modify the a-Expense application to run in Windows Azure
  + [Task 3](#Ex04Task03): Deploy the a-Expense application to the local Azure development fabric

You should be able to complete this exercise in approximately 30 minutes.

## Task 1: Add a Windows Azure Project to the Example Solution

* 1. In this task you will add an Azure cloud project to the begin solution, and add the a-Expense project to this.

To add a Windows Azure project to the example solution

* 1. Start Visual Studio 2010 as an administrator.
  2. Open the solution named **Lab01.Ex04** from the folder **Lab01-SingleSignOn\Source\Ex04\Begin**.
  3. In Solution Explorer right-click on the top-level solution item, point to **Add**, and click **New Project**. In the Add New Project dialog expand **Visual C#** in the list of Installed Templates and click **Cloud**. In the central pane of the dialog select the select **Windows Azure Project** and change the Name to **a-Expense.Cloud**.
     1. 
  4. Click **OK**. Visual Studio displays the New Windows Azure Project dialog where you can add roles to the new project if required. You will add the existing a-Expense application as a role in later step, and you do not need any other roles.
     1. 
  5. Click **OK** in the New Windows Azure Project dialog without selecting any roles. Visual Studio creates the new **a-Expense Cloud** project within the **Lab01.Ex04** solution and it appears in Solution Explorer.
     1. 
  6. In Solution Explorer right-click the **Roles** folder in the **a-Expense Cloud** project, point to **Add**, and click **Web Role Project in solution**. In the Associate with Role Project dialog select the project named **a-Expense.Lab01.Ex04** and click **OK**. The a-Expense project appears in the **Roles** folder of the **a-Expense Cloud** project in Solution Explorer.
     1. 
  7. Open the **Build** menu and click **Rebuild Solution**. Ensure that the solution builds without error.

You have now completed this task. You have a Windows Azure cloud project that references the a-Expense application project. In the next task you will modify the a-Expense application code so that it runs correctly in Windows Azure.

## Task 2: Modify the a-Expense Application to Run in Windows Azure

In this task you will modify the a-Expense application code so that it sends the appropriate **wreply** parameter to the issuer when the application is running in Windows Azure instead of on **localhost**.

To modify the a-Expense application to run in Windows Azure

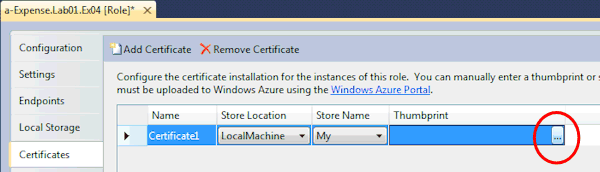
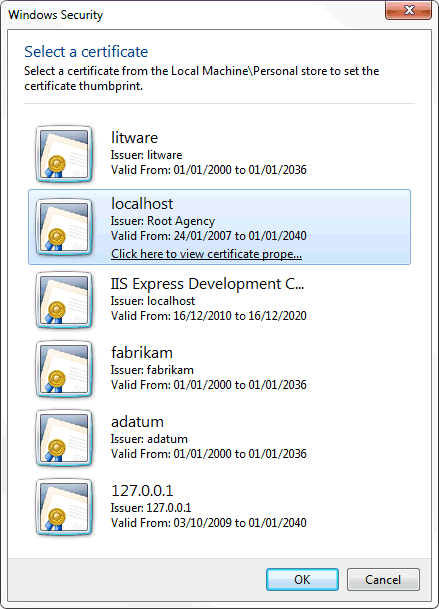
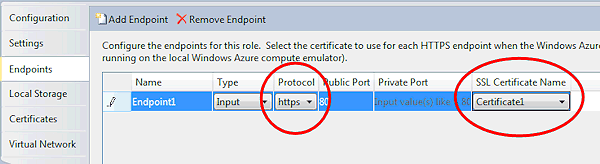
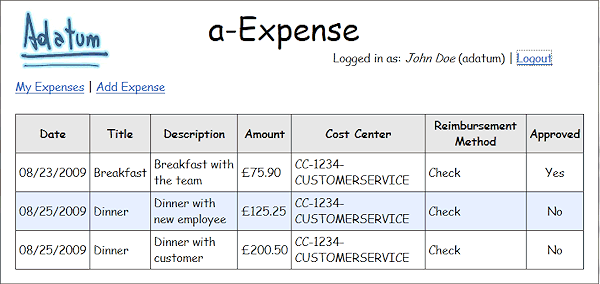
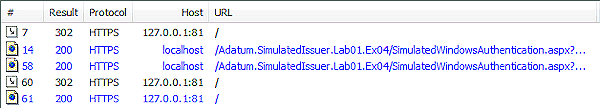
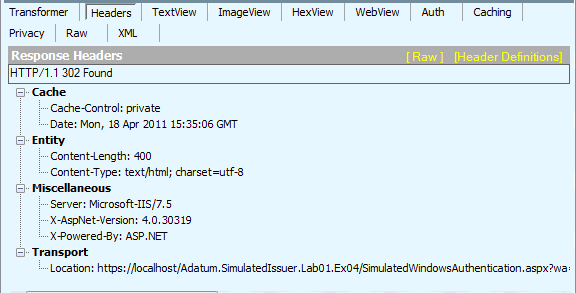
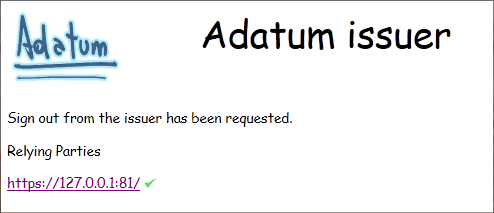
* 1. Continue with the **Lab01.Ex04** solution you used in the previous task.
  2. In Solution Explorer expand the **a-Expense.Lab01.Ex04** project and double-click the file **Global.asax** to open its code file in the Visual Studio editor pane.
  3. Scroll to the end of file and add the following highlighted code to the **Global** class.
     1. C#
     2. private void WSFederationAuthenticationModule\_SignedOut(object sender,
     3. EventArgs e)
     4. {
     5. Response.Redirect("~/CleanUp.aspx", false);
     6. }
     7. **private void WSFederationAuthenticationModule\_RedirectingToIdentityProvider(**
     8. **object sender, RedirectingToIdentityProviderEventArgs e)**
     9. **{**
     10. **}**
     11. }
     12. This code defines a method that will handle the event that WIF exposes before it begins redirecting the client's browser to the configured identity provider or token issuer (this is the Adatum simulated issuer in this example).
  4. To set the **wreply** parameter of the authentication request, the code simply needs to set the value of the **Reply** property of the current **SignInRequestMessage** instance passed to the method in the event argument parameter. Add the following highlighted code to your new method to build the required URL from the properties exposed by current request context, and then assign this to the **Reply** property.
     1. C#
     2. private void WSFederationAuthenticationModule\_RedirectingToIdentityProvider(
     3. object sender, RedirectingToIdentityProviderEventArgs e)
     4. {
     5. // In the Windows Azure environment, build a wreply parameter for the
     6. // SignIn request that reflects the real address of the application.
     7. **HttpRequest request = HttpContext.Current.Request;**
     8. **Uri requestUrl = request.Url;**
     9. **var wreply = new StringBuilder();**
     10. **wreply.Append(requestUrl.Scheme); // for example, "http" or "https"**
     11. **wreply.Append("://");**
     12. **wreply.Append(request.Headers["Host"] ?? requestUrl.Authority);**
     13. **wreply.Append(request.ApplicationPath);**
     14. **if (!request.ApplicationPath.EndsWith("/"))**
     15. **{**
     16. **wreply.Append("/");**
     17. **}**
     18. **e.SignInRequestMessage.Reply = wreply.ToString();**
     19. }
  5. Save and close the **Global.asax.cs** file. Then open the **Build** menu and click **Rebuild Solution**. Ensure that it builds without error.

You have now completed this task. In the final task of this exercise you will set the properties of the a-Expense web role and then deploy it to Windows Azure.

## Task 3: Deploy the a-Expense Application to the Local Azure Development Fabric

* 1. In this task you will set the properties of the a-Expense web role to specify the certificate and endpoint, and then then deploy it to Windows Azure. You will then test authentication against the Adatum simulated issuer and explore how the contents of the request differ from previous examples in this lab.

To deploy the a-Expense application to the local Azure development fabric

* 1. Continue with the **Lab01.Ex04** solution you used in the previous task.
  2. In Solution Explorer right-click the **a-Expense.Lab01.Ex04** web role in the **Roles** folder of the **a-Expense Cloud** project and click **Properties**.
  3. Click the **Certificates** tab and click the **Add Certificate** icon at the top of the Properties pane to add a new certificate for communication with the application when it is running in Windows Azure. Then click the ellipses (**...**) button in the new certificate entry.
     1. 
  4. The Windows Security dialog opens with a list of the certificates installed in the Personal store of your computer. Select the **localhost** certificate. This is the "Root Agency" certificate installed for the examples by the Dependency Checker.
     1. 
  5. Click **OK**. The thumbprint of this certificate appears in the certificate item in the Properties pane.
  6. Click the **Endpoints** tab and change the **Protocol** setting for the default endpoint to **https**, and select the certificate you just added in the **SSL Certificate Name** column.
     1. 
  7. Save your changes and close the Properties pane. Then open the **Build** menu and click **Rebuild Solution**. Ensure that it builds without error.
  8. In Solution Explorer, check that the a-Expense Cloud project is set as the start-up project for the solution. If the **a-Expense Cloud** item is not displayed in bold text, right-click it and click **Set as StartUp Project**.
  9. Open **Fiddler** from your **Start** menu. For information about installing and using Fiddler, see the section "*Pre-requisites for the Labs*" in the *Introduction* document for these labs.
  10. Press *F5* to deploy the application to the local development fabric and start it running. You will see the Adatum issuer page. Notice that the URL displayed in the browser starts with **https://localhost/Adatum.SimulatedIssuer**. The issuer is running locally on port 80 of your machine just as it did in previous examples.
      1. You may need click **Yes** in the Fiddler certificate warning dialog and **Continue to this website** in your browser more than once because you are now running two separate applications. Fiddler will capture the requests and responses for both of them.
  11. Select the user **ADATUM/johndoe** and click **Continue with login**. You see the list of expenses in the a-Expense application. Notice that now the browser address bar shows **https://127.0.0.1:81/**. The a-Expense application is running on port 81 of your computer within the Windows Azure local development fabric.
      1. In some circumstances, such as when you run the a-Expense application more than once, you may find that it runs on a port other than 81. In these steps we assume that it runs on port 81.
      2. 
  12. Look at the Fiddler list of requests/responses (ignore any for style sheets and other resource files). You can see that, as in earlier examples, the application redirects the browser to the simulated issuer, which generates the sign-in form page with the list of accounts. Clicking the **Continue with login** button submits the form to the issuer, which redirects the browser back to the application. The WIF modules in the application receive the token sent by the issuer and generate the **FedAuth** cookies, then send them in a **GET** request to the same application URL.
      1. 
  13. In the list of requests/responses, double-click the first request to the a-Expense application at 127.0.0.1:81 and look at the response headers in the Details View pane. The initial request to the a-Expense application received a **302** (Found) status code. The redirect location is the simulated issuer running on **localhost**.
      1. 
  14. Right-click on the **Location** header in the **Transport** section and click **Copy only Value**, and then paste it into a new Notepad document so that you can examine it. You will see it contains the **wreply** parameter with a value that is the URL of the a-Expense application running in the Windows Azure local development environment.
      1. The value of the Location header is **https://localhost/Adatum.SimulatedIssuer.Lab01.Ex04/SimulatedWindowsAuthentication.aspx?wa=wsignin1.0&wtrealm=https%3a%2f%2flocalhost%2fa-Expense.Lab01.Ex04%2f&wctx=rm%3d0%26id%3dpassive%26ru%3d%252f&wct=2011-04-18T15%3a35%3a06Z&wreply=https%3a%2f%2f127.0.0.1%3a81%2f**. The last parameter, **wreply=https%3a%2f%2f127.0.0.1%3a81%2f**, is the URL-encoded value **https://127.0.0.1:81/**.
  15. Back in your web browser click the **Logout** link in the a-Expense application. The Adatum issuer sign-out page shows that you have been signed out of the application at **https://127.0.0.1:81/** (the a-Expense application).
      1. 
  16. You have now completed this task and this exercise. In this task you have seen how, with minimum adjustments to a claims-based application, you can easily deploy it in Windows Azure simply by setting the appropriate reply address for requests sent to the identity provider or token issuer.
  17. An alternative approach is to modify the **<microsoft.identityModel>** section of the application's Web.config file to specify the appropriate URLs. This is discussed I more detail in Chapter 3 of the book "*A Guide to Claims–based Identity and Access Control, 2nd Edition*" associated with these labs (<http://claimsid.codeplex.com/>).

In the final exercise of this lab you will modify the application to use ADFS 2.0 as the identity provider and token issuer.

## Running the "End" Solution

1. If you did not complete all of the tasks in this exercise, you can run the "end" solution we provide.

To run the end solution

* 1. Start Visual Studio 2010 as an administrator.
  2. Open the solution named **Lab01.Ex04.End** from the folder **Lab01-SingleSignOn\Source\Ex04\End**.
  3. Open the **Build** menu and click **Rebuild Solution**.
  4. In Solution Explorer, check that the a-Expense Cloud project is set as the start-up project for the solution. If the **a-Expense Cloud** item is not displayed in bold text, right-click it and click **Set as StartUp Project**.
  5. Press *F5* to deploy the application to the local development fabric and start it running.
  6. Select the user **ADATUM/johndoe** and click **Continue with login**. You will see the list of expenses.

# Exercise 5: Integrating the Application with ADFS

* 1. In this final exercise of Lab 1 you will modify the a-Order and a-Expense applications to use ADFS 2.0 as the identity provider and token issuer instead of the simulated issuer you have used in previous exercises. Then you will configure ADFS for the a-Order and a-Expense applications as relying parties, and add the required rules to ADFS.
  2. This exercise is optional, and is only possible if you have access to an ADFS 2.0 instance that you can configure for this exercise. For information about installing ADFS 2.0 for these labs and obtaining the necessary information for configuring the example, see the section "*Integrating with Active Directory Federation Services*" in the *Introduction* document for these labs.

This exercise contains the following tasks:

* + [Task 1](#_Task_1:_Configure): Configure ADFS for Adatum
  + [Task 2](#_Task_2:_Configure): Configure the applications to use ADFS
  + [Task 3](#_Task_3:_Configure): Configure a-Expense as a relying party in ADFS
  + [Task 4](#_Task_4:_Configure): Configure a-Order as a relying party in ADFS
  + [Task 5](#_Task_5:_Examine): Verify the configuration

You should be able to complete this exercise in approximately 30 minutes.

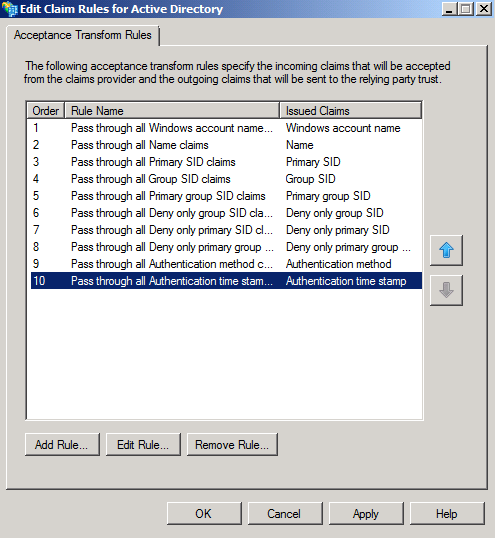
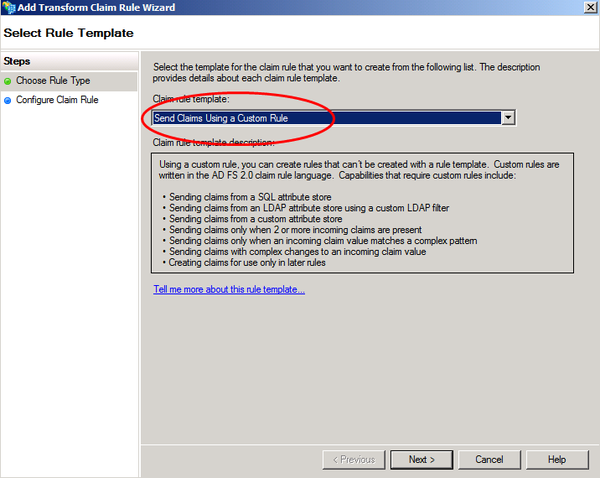
## Task 1: Configure ADFS for Adatum

* 1. In this task, you will configure ADFS to fulfill the role that the Adatum simulated issuer played in the previous lab exercises. Both the a-Order and a-Expense applications will trust the tokens that ADFS issues. To configure ADFS you must ensure that you add the John Doe user account to Active Directory, and that ADFS will pass through the claims that the two relying parties expect to see.

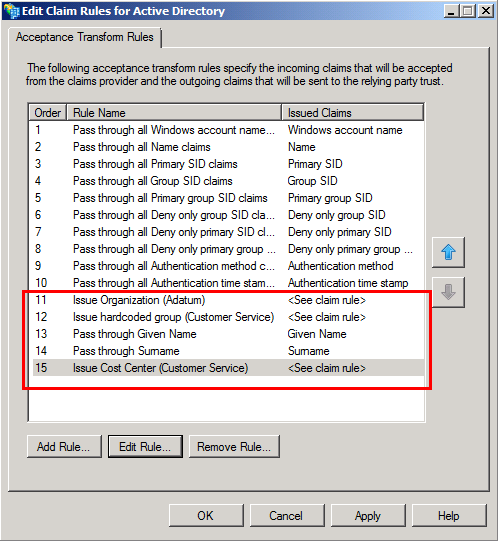
To configure ADFS for Adatum

* 1. On the machine where you have installed Active Directory for use in this Lab, start the **Active Directory Users and Computers** console.
  2. Add a new user in your domain with the attributes listed in the following table.

|  |  |
| --- | --- |
| Attribute | Value |
| Display name | johndoe |
| User logon name | johndoe |
| Password | Pa$$w0rd |
| User must change password at next logon | False |

* 1. On the machine where you have installed ADFS, start the **AD FS 2.0 Management** console. Expand the **Trust Relationships** node, click on **Claims Provider Trusts**, right-click on **Active Directory,** and click **Edit Claim Rules**. Notice that there is are default rules called **Pass through all Windows account name claims** and **Pass through all Name claims** that pass through the standard name claims automatically.
     1. 
  2. Now you can add the additional rules that you need to test the scenario. In the **Edit Claim Rules for Active Directory** dialog, click **Add Rule.** Then, in **Add Transform Claim Rule Wizard** dialog, in the **Claim rule template** dropdown list, select **Send Claims Using a Custom Rule**, and then click **Next**.
     1. 
  3. On the **Configure Rule** page of the wizard, in the **Claim rule name** textbox, type **Issue Organization (Adatum)**. In the **Custom rule** textbox, add the following rule:
     1. => issue(Type = "http://schemas.adatum.com/claims/2009/08/organization", Value = "Adatum");
  4. On the **Configure Rule** page of the wizard, click **Finish**.
  5. Using the information in the following table, repeat steps 3, 4 and 5 to add the Group rule to ADFS.

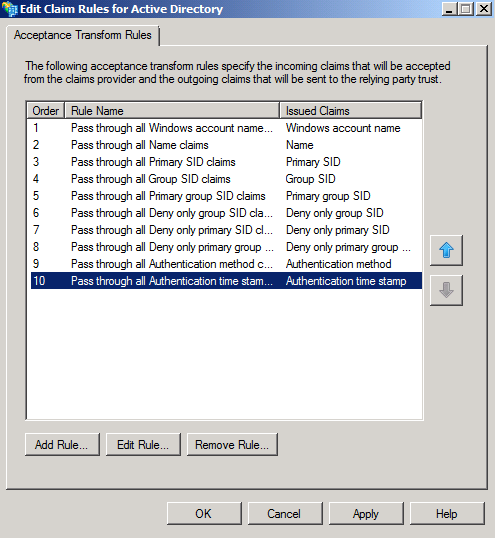
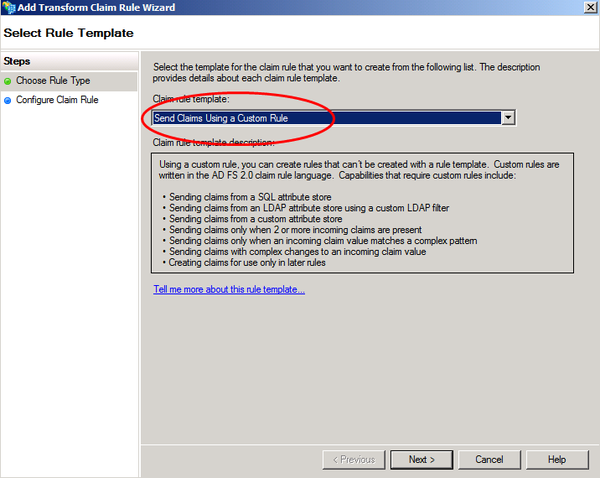
|  |  |
| --- | --- |
| Claim rule name | Custom rule |
| Issue hardcoded group (Customer Service) | => issue(Type = "http://schemas.xmlsoap.org/claims/group", Value = "Customer Service"); |
| Pass through Given Name | c:[Type == "http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname", Issuer == "AD AUTHORITY"]  => issue(store = "Active Directory", types = ("http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname"), query = ";displayName;{0}", param = c.Value); |
| Pass through Surname | c:[Type == "http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname", Issuer == "AD AUTHORITY"]  => issue(store = "Active Directory", types = ("http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname"), query = ";sn;{0}", param = c.Value); |
| Issue Cost Center (Customer Service) | => issue(Type = "http://schemas.adatum.com/claims/2009/08/costcenter", Value = "Customer Service"); |

* 1. When you have finished, the **Edit Claim Rules for Active Directory** dialog should look like the following screenshot. Click **OK** to close the dialog.
     1. 
  2. You have now completed this task to add a new test user called John Doe to Active Directory and to configure the Active Directory claims provider trust in ADFS.

## Task 2: Configure the Applications to Use ADFS

* 1. In this task, you will modify the a-Expense and a-Order applications so that they use ADFS as the identity provider and token issuer instead of the simulated issuer you have used in previous exercises.
  2. You can configure the issuer that an application (a relying party) uses in Visual Studio with the STS Wizard. You used this to configure the applications to use the simulated issuer in the first exercise of this lab. However, you can also specify the issuer by editing the values in the Web.config file manually. This is a useful technique where you do not want to rebuild the application when the issuer details change. In this exercise, you will see both approaches demonstrated.

To configure the applications to use ADFS

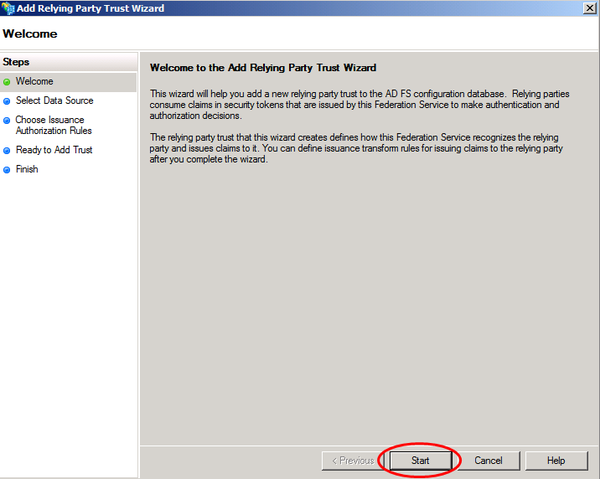
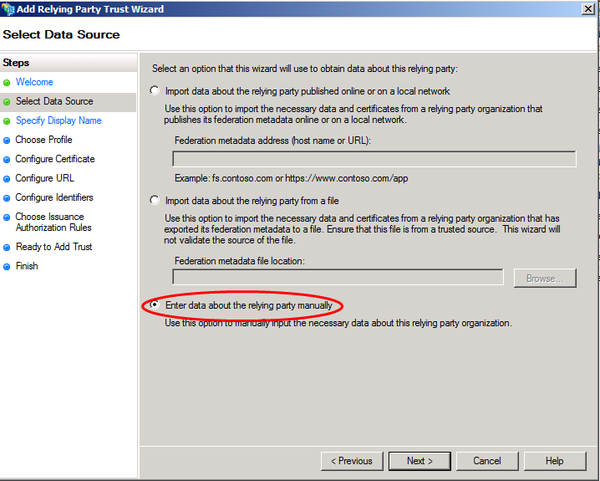
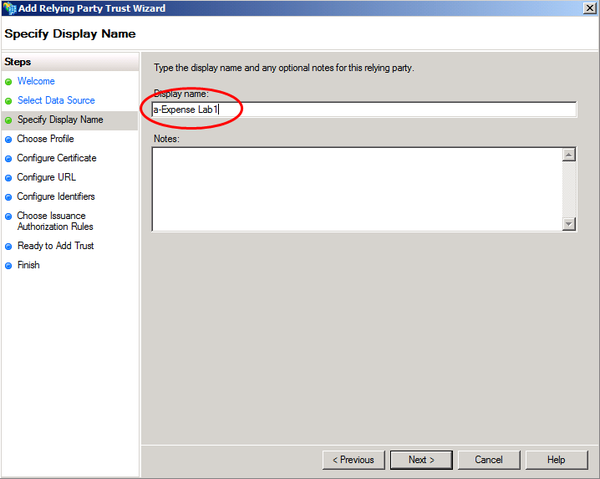
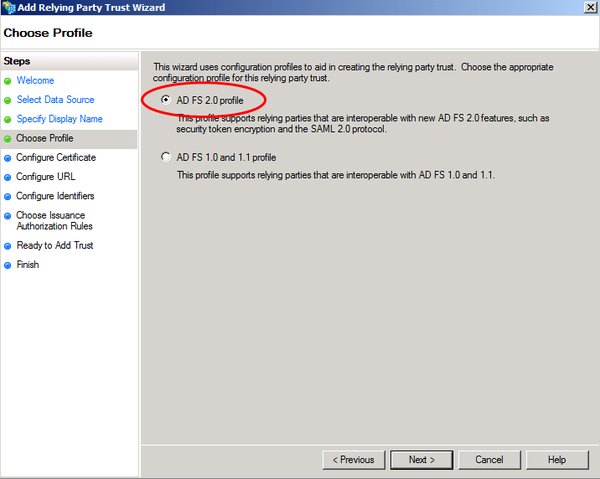
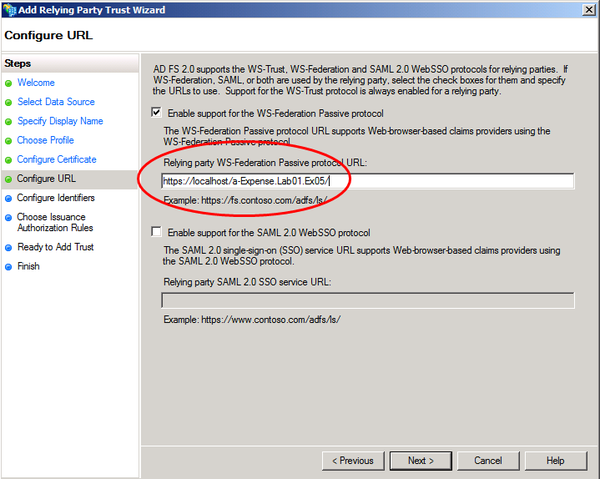
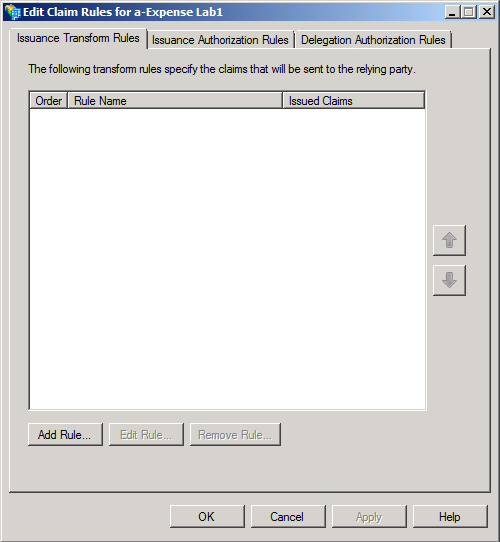
* 1. Start Visual Studio 2010 as an administrator.
  2. Open the solution named **Lab01.Ex05** from the folder **Lab01-SingleSignOn\Source\Ex05\Begin**.
  3. First, you will remove the Adatum simulated issuer from the solution. In this exercise, you are replacing this mock identity provider with ADFS. In Solution Explorer, right-click on **Adatum.SimulatedIssuer.Lab01.Ex05** and click **Remove**, and then click **OK**.
  4. Next, you will modify the a-Expense application. Right-click the **a-Expense.Lab01.Ex05** project item and click **Add STS Reference**.
  5. Click **Next** in the Welcome page.
  6. In the Security Token Service page select **Use an existing STS** and enter the URL of the federation metadata document exposed by your ADFS service. This URL is the host name of your ADFS server with the path **/FederationMetadata/2007-06/FederationMetadata.xml** appended to it.
     1. 
  7. Click the **Test location** button. A browser window opens that appears to be empty. Right-click on the empty page and click **View Source** to confirm that the metadata document was correctly downloaded.
     1. If you receive a warning: **ID1025: A certificate processed, but terminated in a root certificate which is not trusted by the trust provider**, you may not have installed your ADFS service certificate on your development machine — see the section "*Integrating with Active Directory Federation Services*" in the *Introduction* document for these labs.
  8. In the STS Wizard click **Next**. Ensure that **Disable certificate chaining validation** is selected. Click the **View Certificates** button if you want to see the token signing certificate used by the ADFS STS. Then, in the main page of the STS Wizard, click **Next**.
  9. Ensure that **No encryption** is selected in the Security token encryption page and click **Next**.
  10. You will see the list of claims offered by ADFS 2.0. These will be populated from your Active Directory when a user authenticates and placed in the security token returned from ADFS. Notice the message indicating that, by default, only the name and role claims will be requested by the application.
      1. 
  11. In the Offered claims page click **Next**, then click **Finish** in the Summary page. You should see a message confirming that the Federation Utility completed successfully. Click **OK**.
  12. In Solution Explorer, expand the **a-Expense.Lab01.Ex05** project item and double-click the **Web.config** file to open it in the Visual Studio editor.
  13. Look at the **<microsoft.identityModel>** section. You will see the URL of the ADFS issuer that the application will use to authenticate in the **<wsFederation>** element, and the list of claims offered by the issuer in the **<claimTypeRequired>** element, and the thumbprint and name identifier of your ADFS service in the **<trustedIssuers>** section (the previous issuer, "**adatum**", is still shown here but will not be used).
      1. Only the **name** and **role** claim types are enabled; the rest are commented out. If you wanted to request additional claims, you would uncomment them here. Also notice, near the top of the file, that the STS Wizard updated the value of the **FederationMetadataLocation** in the **<appSettings>** section.
  14. Find the **<serviceCertificates>** element that has been commented out by the FedUtil utility and remove the comment markers so that it is enabled. This section should look like that shown here.
      1. XML
      2. <!--Commented out by FedUtil-->
      3. **<serviceCertificate>**
      4. **<certificateReference x509FindType="FindByThumbprint"**
      5. **findValue="5a074d678466f59dbd063d1a98b1791474723365" />**
      6. **</serviceCertificate>**
  15. Now you will update the a-Order application, but this time without using the STS Wizard. In Solution Explorer, expand the **a-Order.Lab01.Ex05** project item and double-click the **Web.config** file to open it in the Visual Studio editor.
      1. You will need the URL of the ADFS service, the federation service identifier, and the thumbprint of the certificate the service uses to sign the tokens for this. You should have made a note of these values when working through the procedure described in the section "*Integrating with Active Directory Federation Services*" in the *Introduction* document for these labs. However, in this case, you can copy them from the **Web.config** file in the a-Expense application that is still open in the editor.
  16. In the **<appSettings>** section of the a-Order application's **Web.config** file, comment out the element that specifies the location of the federation metadata document as shown here.
      1. XML
      2. <appSettings>
      3. <!--
      4. <add key="FederationMetadataLocation" value="http://...FederationMetadata.xml" />
      5. -->
      6. </appSettings>
      7. If you do not use the STS Wizard, or if the ADFS service you are using does not expose metadata, this value will not be in the **Web.config** file.
  17. Edit the value of the issuer attribute of the **<wsFederation>** element that specifies the issuer the application will use, as shown in the following highlighted code. You must insert the URL of your ADFS service.
      1. XML
      2. <federatedAuthentication>
      3. <wsFederation passiveRedirectEnabled="true" issuer="**https://{issuer-url}/adfs/ls/**"
      4. realm="https://localhost/a-Order.Lab01.Ex05/" requireHttps="true" />
      5. <cookieHandler requireSsl="true" />
      6. </federatedAuthentication>
  18. Add a new thumbprint to the **<trustedIssuers>** section for your own ADFS service, as shown in the following highlighted code. The thumbprint should not include spaces (ensure that you do not include the non-visible space at the start). The name will be of the form **http://***issuer-url***/adfs/services/trust**.
      1. XML
      2. <trustedIssuers>
      3. <add thumbprint="f260042d59e14817984c6183fbc6bfc71baf5462" name="adatum" />
      4. **<add thumbprint="{your-thumbprint}" name="{your-name-identifier}" />**
      5. </trustedIssuers>
      6. Notice that the **<microsoft.identityModel>** section of this Web.config file does not include an **<applicationService>** section containing the claim definitions. In the a-Expense application the **<applicationService>** section contains a **<claimTypeRequired>** element that lists the claims that must be present in the token in order for WIF to consider it valid. You do not require a list of required claim types, but bear in mind that your code may fail if it assumes that a specific claim is available for an authenticated user when that claim is not included in the token.
  19. Close both of the Web.config files, open the **Build** menu, and click **Rebuild Solution**. Ensure that it builds without error.

You have now completed this task. You have seen how to use both the STS Wizard in Visual Studio and how to edit the values in the **Web.config** file manually when configuring the issuer that an application (a relying party) uses. In the next task, you will configure the a-Expense application as a relying party in ADFS using the management console.

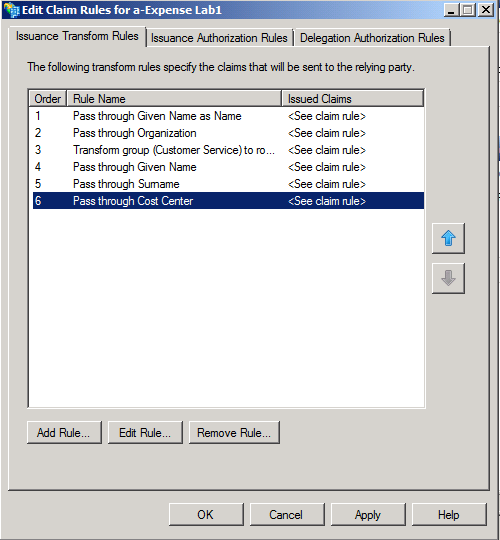
## Task 3: Configure a-Expense as a Relying Party in ADFS

* 1. In this task you will configure ADFS so that the a-Expense application is recognized as a relying party, and you will add suitable rules to ADFS for this relying party so that it can successfully authenticate users and authorize the operations these users perform in the application. You will use the ADFS management console for this task.

To configure a-Expense as a relying party in ADFS using the management console

* 1. Open the **AD FS 2.0 Management** console from your **Start** menu of the server that hosts your ADFS service. You must be logged into the server using an account that has administrative privileges.
  2. Expand the top-level **AD FS 2.0** folder, then expand **Trust Relationships**, then right-click on **Relying Party Trusts**, and click **Add Relying Party Trust**.
  3. In the **Add Relying Party Trust Wizard** click **Start**.
     1. 
  4. On the **Select Data Source** page of the wizard, select **Enter data about the relying party manually** and click **Next**.
     1. 
  5. On the **Specify Display Name** page of the wizard, type **a-Expense Lab1** in the **Display name** textbox, then click **Next**.
     1. 
  6. On the **Choose Profile** page of the wizard, ensure that the **AD FS 2.0 profile** option is selected, and then click **Next**.
     1. 
  7. On the **Configure Certificate** page of the wizard, click **Next**.
  8. On the **Configure URL** page of the wizard, ensure that **Enable support for the WS-Federation Passive protocol** is checked, in the **Relying party WS-Federation Passive protocol URL** text box type **https://localhost/a-Expense.Lab01.Ex05/**, and then click **Next**.
     1. 
  9. On the **Configure Identifiers** page of the wizard, click **Next**.
  10. On the **Choose Issuance Authorization Rules** page of the wizard, click **Next**.
  11. On the **Ready to Add Trust** page of the wizard, click **Next**.
  12. On the **Finish** page of the wizard, click **Close**. The **Edit Claim Rules for a-Expense Lab1** dialog will open ready for you to complete the next steps.
      1. 
  13. Now you can add the claim transformation rules you need to test the scenario. In the **Edit Claim Rules for a-Expense Lab1** dialog, click **Add Rule.** Then, in **Add Transform Claim Rule Wizard** dialog, in the **Claim rule template** dropdown list, select **Send Claims Using a Custom Rule**, and then click **Next**.
  14. On the **Configure Rule** page of the wizard, in the **Claim rule name** textbox, type **Pass through Given Name as Name**. In the **Custom rule** textbox, add the following rule:
      1. c:[Type == "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname"]
      2. => issue(Type = "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name", Issuer = c.Issuer, OriginalIssuer = c.OriginalIssuer, Value = c.Value, ValueType = c.ValueType);
  15. On the **Configure Rule** page of the wizard, click **Finish**.
  16. Using the information in the following table, repeat steps 2, 3, and 4 to add the remaining rules to ADFS.

|  |  |
| --- | --- |
| Claim rule name | Custom rule |
| Pass through Organization | c:[Type == "http://schemas.adatum.com/claims/2009/08/organization"] => issue (claim = c); |
| Transform group (Customer Service) to role (Order Tracker) | c:[Type == "http://schemas.xmlsoap.org/claims/group", Value =~ "^(?i)Customer\ Service$"]  => issue(Type = "http://schemas.microsoft.com/ws/2008/06/identity/claims/role", Issuer = c.Issuer, OriginalIssuer = c.OriginalIssuer, Value = "Order Tracker", ValueType = c.ValueType); |
| Pass through Given Name | c:[Type == "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname"]  => issue(claim = c); |
| Pass through Surname | c:[Type == "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname"]  => issue(claim = c); |
| Pass through Cost Center | c:[Type == "http://schemas.adatum.com/claims/2009/08/costcenter"]  => issue(claim = c); |

* 1. When you have finished, the **Edit Claim Rules for a-Expense Lab1** dialog should look like the following screenshot. Click **OK** to close the dialog.
     1. 

1. You have now completed this task. You have seen how to configure the a-Expense application as a relying party in ADFS using the management console. In the next task, you will configure the a-Order application as a relying party using PowerShell scripts.

## Task 4: Configure a-Order as a Relying Party in ADFS

* 1. In this task you will configure ADFS so that the a-Order application is recognized as a relying party, and you will add suitable rules to ADFS for this relying party so that it can successfully authenticate users and authorize the operations these users perform in the application. You will use PowerShell scripts for this task to demonstrate how you can automate the configuration of relying parties in ADFS.

To configure a-Order as a relying party in ADFS using PowerShell scripts

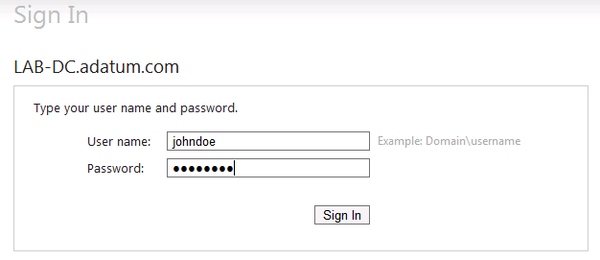
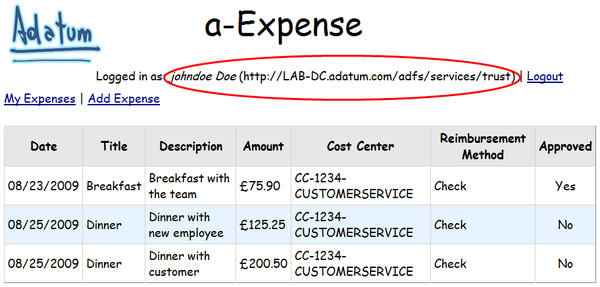
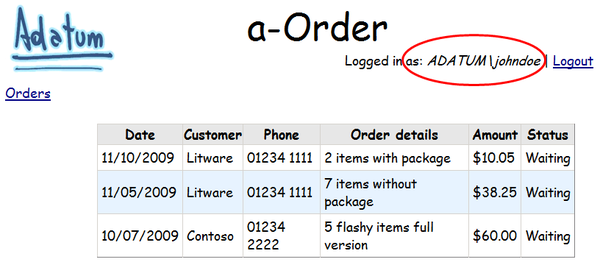
* 1. Log into the server that hosts your ADFS service using an account that has administrative privileges.
  2. Open the **PowerShell** integrated scripting environment (in Windows Server 2008 R2, this is **Windows PowerShell ISE** in the **Accessories | Windows PowerShell** folder of the **Start** menu).
  3. Change to the **Lab01-SingleSignOn\Source\Ex05\Assets** folder.
  4. Type **Add-aOrder-RelyingParty-Begin.ps1** and press **Enter** to create the relying party trust in ADFS.
  5. Type **Add-aOrder-Rules-Begin.ps1** and press **Enter** to create the rules for the **a-Order Lab 1** relying party in ADFS.

You have now completed this task. You have seen how to configure the a-Order application as a relying party in ADFS using the management console. In the next task, you will examine the flow of authentication requests between the relying party and the ADFS identity provider and token issuer.

## Task 5: Verify the Configuration

* 1. In this task, you will verify the configuration changes you have made by authenticating with ADFS and visiting the a-Order and a-Expense applications. This will illustrate the scenario running with a "real-world" issuer rather than the simulated issuer you used in previous examples.
  2. If you are running this Lab on the same machine as the one where you have installed ADFS, then you will receive a warning message "The security certificate presented by this website was issued for a different website's address" when your browser is redirected to ADFS. You should click **Continue to this website** to continue.

To verify the configuration

* 1. Continue with the **Lab01.Ex05** solution you used in the previous task.
  2. Open your browser and navigate to **https://localhost/a-Expense.Lab01.Ex05/**. ADFS will redirect your browser to the ADFS sign in page. In the **User name** field, enter **johndoe**, in the **Password** field, enter **Pa$$w0rd**, and then click **Sign In**.
     1. 
     2. If you don't see the ADFS sign in page, you should check your ADFS configuration. See the section "*Integrating with Active Directory Federation Services*" in the *Introduction* document for these labs for details.
  3. ADFS will redirect your browser back to the a-Expense web site.
     1. 
  4. In the same browser window, navigate to **https://localhost/a-Order.Lab01.Ex05/**. ADFS will use single sign-on to sign you into the a-Order application.
     1. 

You have now completed this task and this exercise. In this task, you have seen how to replace the simulated claims issuer with ADFS to implement a more "real-world" version of the single sign-on scenario.

## Running the "End" Solution

1. If you did not complete all of the tasks in this exercise, you can run the provided "end" solution.

To run the end solution

* 1. You must manually add a new user to Active Directory as described in task 1 of this exercise.
  2. You must add the Active Directory claims transformation rules to ADFS. You can do this by running the **Add-Adatum-Rules-End.ps1** PowerShell script in the **Lab01-SingleSignOn\Source\Ex05\Assets** folder.
  3. You must configure the a-Expense application to work with your ADFS installation. Use the following code snippet from the Web.config file in the a-Expense application for guidance.
     1. You will need the URL of the ADFS service, the federation service identifier, and the thumbprint of the certificate the service uses to sign the tokens for this. You should have made a note of these values when working through the procedure described in the section "*Integrating with Active Directory Federation Services*" in the *Introduction* document for these labs.
     2. XML
     3. <microsoft.identityModel>
     4. <service>
     5. <audienceUris>
     6. <add value="https://localhost/a-Expense.Lab01.Ex05.End/" />
     7. </audienceUris>
     8. <federatedAuthentication>
     9. <cookieHandler requireSsl="true" />
     10. <wsFederation passiveRedirectEnabled="true"
     11. issuer="https://**{issuer-url}**/adfs/ls/"
     12. realm="https://localhost/a-Expense.Lab01.Ex05/" requireHttps="true" />
     13. </federatedAuthentication>
     14. <issuerNameRegistry type="Microsoft.IdentityModel …
     15. <trustedIssuers>
     16. <!--Adatum's identity provider -->
     17. <add thumbprint="f260042d59e14817984c6183fbc6bfc71baf5462" name="adatum" />
     18. <!—Your ADFS details -->
     19. <add thumbprint="**{your-thumbprint}**"
     20. name="http://**{issuer-url}**/adfs/services/trust" />
     21. </trustedIssuers>
     22. </issuerNameRegistry>
     23. <serviceCertificate>
     24. <certificateReference x509FindType="FindByThumbprint" findValue="5a074d678466f59dbd063d1a98b1791474723365" />
     25. </serviceCertificate>
     26. <certificateValidation certificateValidationMode="None" />
     27. </service>
     28. </microsoft.identityModel>
  4. You must configure the a-Order application to work with your ADFS installation. Use the following code snippet from the Web.config file in the a-Order application for guidance.
     1. You will need the URL of the ADFS service, the federation service identifier, and the thumbprint of the certificate the service uses to sign the tokens for this. You should have made a note of these values when working through the procedure described in the section "*Integrating with Active Directory Federation Services*" in the *Introduction* document for these labs.
     2. XML
     3. <microsoft.identityModel>
     4. <service>
     5. <audienceUris>
     6. <add value="https://localhost/a-Order.Lab01.Ex05.End/" />
     7. </audienceUris>
     8. <federatedAuthentication>
     9. <cookieHandler requireSsl="true" />
     10. <wsFederation passiveRedirectEnabled="true"
     11. issuer="https://**{issuer-url}**/adfs/ls/"
     12. realm="https://localhost/a-Order.Lab01.Ex05/" requireHttps="true" />
     13. </federatedAuthentication>
     14. <issuerNameRegistry type="Microsoft.IdentityModel …
     15. <trustedIssuers>
     16. <!--Adatum's identity provider -->
     17. <add thumbprint="f260042d59e14817984c6183fbc6bfc71baf5462" name="adatum" />
     18. <!—Your ADFS details -->
     19. <add thumbprint="**{your-thumbprint}**"
     20. name="http://**{issuer-url}**/adfs/services/trust" />
     21. </trustedIssuers>
     22. </issuerNameRegistry>
     23. <serviceCertificate>
     24. <certificateReference x509FindType="FindByThumbprint" findValue="5a074d678466f59dbd063d1a98b1791474723365" />
     25. </serviceCertificate>
     26. <certificateValidation certificateValidationMode="None" />
     27. </service>
     28. </microsoft.identityModel>
  5. You can configure the a-Expense end solution as a relying party in ADFS by running the **Add- aExpense-RelyingParty-End.ps1** and **Add- aExpense-Rules-End.ps1** PowerShell scripts in the **Lab01-SingleSignOn\Source\Ex05\End** folder.
  6. You can configure the a-Order end solution as a relying party in ADFS by running **Add- aOrder-RelyingParty-End.ps1** and **Add- aOrder-Rules-End.ps1** PowerShell scripts in the **Lab01-SingleSignOn\Source\Ex05\End** folder.
  7. To verify the solution, you should use the following URLs to access the applications: **https://localhost/a-Expense.Lab01.Ex05.End/** and **https://localhost/a-Order.Lab01.Ex05.End/**.

In this lab, you have seen how to claims enable an ASP.NET web application and implement a single sign-on and single sign-out environment using claims. You have also seen how to migrate a claims-enabled web application to Windows Azure and how to configure ADFS as your claims issuer.