|  |  |
| --- | --- |
| **mslogo-1.jpg** | PluralsightColorLogoLgTrans.png |

A Developer’s Guide to the Microsoft® .NET Access Control Service

Access control in the cloud

Keith Brown, Pluralsight

December 2008

Contents

[Abstract 4](#_Toc216770152)

[An Overview of the .NET Access Control Service 4](#_Toc216770153)

[Identity Challenges 4](#_Toc216770154)

[A Better Solution: Microsoft .NET Access Control Service 5](#_Toc216770155)

[Using a Claims-based Identity Model 5](#_Toc216770156)

[Introduction to Claims-based Identity 6](#_Toc216770157)

[Claims-based Identity Terminology 6](#_Toc216770158)

[Identity 6](#_Toc216770159)

[Claim 6](#_Toc216770160)

[Security Token 7](#_Toc216770161)

[Issuing Authority & Identity Provider 7](#_Toc216770162)

[Security Token Service (STS) 8](#_Toc216770163)

[Relying Party (RP) 8](#_Toc216770164)

[A Simple Claims-Based Scenario using SOAP 9](#_Toc216770165)

[Standards 9](#_Toc216770166)

[Browser-based Applications 10](#_Toc216770167)

[Chained Issuers 11](#_Toc216770168)

[Identity Federation across Security Realms 11](#_Toc216770169)

[Understanding the .NET Access Control Service 14](#_Toc216770170)

[Getting Started with ACS 15](#_Toc216770171)

[ACS in Action: the Calculator Sample 15](#_Toc216770172)

[Configuring the .NET Access Control Service 20](#_Toc216770173)

[Accounts and Solutions 20](#_Toc216770174)

[Solution Scopes 21](#_Toc216770175)

[Settings in a Scope 22](#_Toc216770176)

[Identity Issuers 23](#_Toc216770177)

[Claim Types 24](#_Toc216770178)

[Claims Transformation 25](#_Toc216770179)

[Implementing Role Based Access Control (RBAC) 26](#_Toc216770180)

[Passive Examples 27](#_Toc216770181)

[Integrating ACS into an ASP.NET Web Application 27](#_Toc216770182)

[The Code behind Contoso Woodworking 31](#_Toc216770183)

[Summary 36](#_Toc216770184)

[Additional Resources 36](#_Toc216770185)

[Microsoft® .NET Services Whitepaper Series 36](#_Toc216770186)

[.NET Access Control Service Resources 36](#_Toc216770187)

[About the Author 36](#_Toc216770188)

[Acknowledgements 37](#_Toc216770189)

# Abstract

The goal of this whitepaper is to show developers how to use a claims-based identity model and the Microsoft® .NET Access Control Service – part of the Microsoft® .NET Services family – to implement single sign-on, federated identity, and role based access control in Web applications and services.

Since the Geneva Framework is the best way to integrate this service into your applications, I strongly recommend reading the “[Geneva” Framework whitepaper](http://download.microsoft.com/download/7/d/0/7d0b5166-6a8a-418a-addd-95ee9b046994/GenevaFrameworkWhitepaperForDevelopers.pdf) once you’re done with this one.

# An Overview of the .NET Access Control Service

Microsoft® .NET Services[[1]](#footnote-2) are a set of highly scalable developer-oriented services running in Microsoft data centers as part of the Azure™ Services Platform. Microsoft .NET Services provides developers with common building blocks and infrastructure services for cloud-based and cloud-aware applications. Much like you rely on the .NET Framework for common building blocks when developing on-premise software, you will rely on Microsoft® .NET Services for the common building blocks in your cloud applications.

The Microsoft® .NET Access Control Service is one of the core service offerings found within Microsoft® .NET Services. Today it’s complemented by two other services: the Microsoft® .NET Service Bus and the Microsoft® .NET Workflow Service. The .NET Service Bus relies on the Access Control Service for securing access to your Azure solutions through a claims-based identity model. The .NET Workflow Service allows you to define cloud-based workflows that model service interactions through the .NET Service Bus. Together these services provide a valuable development fabric required by most cloud applications, thereby simplifying development by allowing you to focus more directly on business needs.[[2]](#footnote-3)

This whitepaper focuses specifically on how to implement a claims-based identity model using the .NET Access Control Service in order to support single sign-on, federated identity, and role based access control. I will first lay some groundwork with terms and concepts that are important for understanding claims-based identity, and with that in place, I’ll talk about ACS in depth.

## Identity Challenges

The first two questions most applications have to answer these days are related to identity: who is the user, and what is she allowed to do? This need for authentication and authorization are common across many different types of applications, from web services and browser-based applications, to rich Windows desktop applications to console command line applications. But despite the common need for these features, many applications end up with custom solutions. Most developers are not security experts, and many feel uncomfortable being given the job of authenticating, authorizing, and personalizing experiences for users. It’s not a subject that has been traditionally taught in computer science curriculum, and there’s a long history of ignoring it until late in the development lifecycle.

It’s not surprising nowadays to see a single company with tens or hundreds of applications and services, many of which have their own private silo for user identities, and most of which are hardwired to use one particular means of authentication. Developers know how tedious it is to build identity support into each application, and IT pros know how expensive it is to manage the resulting set of applications.

The rampant use of passwords has lead to widespread phishing attacks[[3]](#footnote-4). And with so many applications building in custom solutions for authentication and authorization, it’s often difficult to implement single-sign on across them, or to federate identity across security realms.

## A Better Solution: Microsoft .NET Access Control Service

The identity solution that Microsoft has been moving toward over the last few years is based on the idea of claims. A claims-based identity model allows the common features of authentication and authorization to be factored out of applications and centralized into external services written and maintained by subject matter experts in security and identity, which is beneficial to all involved.

The Microsoft® .NET Access Control Service is a cloud-based service that does exactly that. Instead of writing your own custom user account and role database, you can let ACS orchestrate the authentication and authorization of your users. ACS relies on existing user account stores such as Windows Live ID, Active Directory, as well as any other store that supports the standard federation protocols I’ll talk about shortly. This makes it natural to achieve single sign on across applications. It’s also a great way to centralize authentication and access control logic, which simplifies your applications.

## Using a Claims-based Identity Model

When you build a claims-aware application, the user presents her identity to your application as a set of *claims* (see Figure 1). One claim could be the user’s name; another might be her email address. These claims are supplied by an issuing authority that knows how to authenticate the user and where to find her attributes. The client application, which might be a browser or a rich client, transparently works with the authority to discover these claims and pass them along to your application. The end result is that your application receives all of the identity details it needs to know about the user in a set of claims. And the claims are signed to give you cryptographic assurance of their origin.



Figure 1: User Presents Claims

Under this model, single sign-on is easier to achieve, and your application is **no longer responsible** for:

* Authenticating users
* Storing user accounts and passwords
* Calling to enterprise directories to look up user identity details
* Integrating with identity systems from other platforms or companies

This model allows your application to make identity-related decisions based on claims supplied by the user. This could be anything from simple application personalization with the user’s first name, to authorizing the user to access higher valued features and resources in your application.

# Introduction to Claims-based Identity

In this section of the paper, I’m going to introduce some terminology and concepts so that you, as a developer, can get your head around this identity architecture.

## Claims-based Identity Terminology

There are several terms commonly used to describe claims-based identity including: *identity*, *claim*, *security token*, *issuing authority*, *security token service* (STS), and *relying party*. It’s important for us to share a clear understanding of these terms before diving into the details of the ACS.

### Identity

The word “identity” is a much overloaded term. So far I’ve been using it to describe the problem space that includes authentication, authorization, etc. But for the purposes of describing ACS, I will use the term identity to describe a set of attributes (well, claims as you’ll see shortly) that describe a user or some other entity in the system that you care about from a security standpoint.

### Claim

You can think of a claim as a bit of identity information such as name, email address, age, membership in the Sales role, and so on. The more claims your application receives, the more you’ll know about the user who is making the request. You may be wondering why I’m using the word “claim”, instead of the more traditional “attribute”, commonly used in the enterprise directory world. The reason has to do with the delivery method – in this model your application doesn’t look up user attributes in a directory. Instead, the user delivers claims to your application, and you’re going to examine them with a certain measure of doubt. Claims are signed by an *issuer*, and you’ll trust a set of claims only as much as you trust that issuer. Part of accepting a claim is verifying that it came from an issuer that you trust. This involves some cryptographic heavy lifting, which you could write yourself using plumbing in the .NET Framework, but as I’ll show later in this paper, WCF, the Geneva Framework, and ACS handle this transparently for you, allowing you to simply focus on which identity providers you trust.

### Security Token

The user delivers a set of claims to your application piggybacked along with her request. In a web service, these claims are carried in the security header of the SOAP envelope. In a browser-based web application, the claims arrive via an HTTP POST from the user’s browser, and may be used to establish a typical cookie-based logon session. Regardless of how they arrive, claims must be serialized somehow, and this is where security tokens come in. A *security token* is a set of claims that has been serialized into XML (most commonly as SAML[[4]](#footnote-5)) and digitally signed by the issuing authority. The signature is important – it gives you assurance that the user didn’t just make up a bunch of claims and send them to you.

### Issuing Authority & Identity Provider

An issuing authority has two main features. The first and most obvious is that it issues security tokens. The second feature, which is really important but often overlooked in the literature, is the logic that determines which claims to issue. This is typically based on the user’s identity, the application being used, and other context such as time of day. This type of logic is often referred to as *policy*[[5]](#footnote-6).

There are many issuing authorities, including Windows Live ID, Geneva Server (a product which uses Active Directory as its user store), PingFederate from Ping Identity (a product that exposes user identities from the Java world), and many more.

Some authorities, such as Windows Live ID, are entirely focused on user identity. Their job is to authenticate a user and issue a SAML token with the user’s ID and possibly other identity attributes. These types of authorities are called *identity providers* (sometimes shortened as IdP). It’s ultimately their responsibility to answer the question, “who are you?” and ensure that the user knows her password, is in possession of her smart card, knows her PIN code, has a matching retinal scan, and so on.

There are lots of ways of authenticating a user, and by allowing an identity provider to do that heavy lifting, that’s one less hard problem for your application to solve.

ACS is also an issuing authority, and while the CTP version includes a temporary feature so that it can act as an identity provider, that’s really not its job. Rather the job of ACS is to *provide a set of claims that are immediately useful to your application*. This means translating the claims that come from an identity provider such as Geneva Server or Windows Live ID into a set of roles and other claims that have meaning to your application. While your app might not care that the user’s name is Alice, it most definitely does care to know which features she is authorized to use.

A big part of ACS is its administration system. ACS has a web-based administration portal, which you can access by simply firing up your browser and signing in[[6]](#footnote-7). This is where you configure the rules that determine how ACS will issue claims for various users, and ultimately answer the question, “What can you do?” Figure 2 shows the portal.

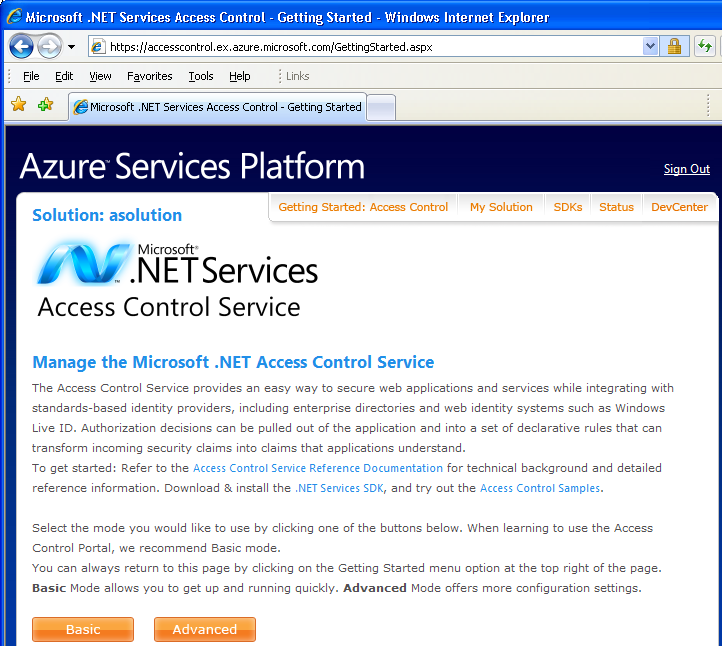


Figure 2: The ACS portal

### Security Token Service (STS)

A security token service (STS) is the plumbing that builds, signs, and issues security tokens according to the interoperable protocols that I’ll discuss in the upcoming section called Standards. You might find it interesting to note that internally, ACS uses the Geneva Framework to do this heavy lifting.

For those who want a more technical definition, an STS is technically an implementation of WS-Trust that accepts Request for Token requests (RST) and returns a response (RSTR), but there’s a tendency in the literature to refer to an issuing authority as an STS, even when WS-Trust isn’t being used (such as in passive, browser-based scenarios). I’m not going to sweat these details in this paper. If you see the term STS, just think of it as the feature of the issuing authority that is responsible for actually issuing tokens.

### Relying Party (RP)

When you build an application that relies on claims, you are technically building a *relying party*. Some synonyms that you may have heard are, *claims aware application*, or *claims-based application*. I’m mainly including this definition here because you’ll find it in the literature, but since ACS is all about helping you build applications that act as relying parties, I’ll try to avoid being overly technical and simply refer to your application as just that: your application.

## A Simple Claims-Based Scenario using SOAP

Now that you’ve learned the terminology, here’s an example of a generic claims-based system in action.



Figure 3: Basic Scenario with Web Services

Figure 3 shows a claims-aware application (which happens to be a web service) and a smart client that wants to use the application. The application exposes WSDL[[7]](#footnote-8) that describes its addresses, bindings, and contracts. The policy section in the WSDL includes a list of claims that the application needs, for example user name, email address, and role memberships. The policy also tells the smart client the address of the issuing authority’s STS, which is where the smart client must go in order to retrieve these claims.

After discovering the application’s policy (1), the client now knows where to go to authenticate. The smart client makes a WS-Trust request (2) to the STS, requesting the claims that the application needs. The job of the STS is to authenticate the user and return a security token that gives the application all of the claims it needs. The smart client then sends its request to the application (3), including the security token in the SOAP security header. The application can now use plumbing such as WCF or the Geneva Framework to verify the signature on the token. This particular application simply rejects any request unless it comes with a token signed by the issuing authority that the application trusts.

As you’ll see shortly, a similar protocol exists for browser-based applications; claims are not just for web services. What I want you to take away from this scenario is a feel for how claims flow. The basic idea is that *the client must visit an authority in order to get the claims it needs to present to an application*. In this very simple example, ACS would act as the authority.

## Standards

In order to make all of this interoperable, several WS-\* standards are used in the above scenario. WSDL can be retrieved using WS-MetadataExchange or a simple HTTP GET, and the policy inside the WSDL is structured according to the WS-Policy specification. The issuing authority’s STS exposes an endpoint that implements the WS-Trust specification[[8]](#footnote-9), which describes how to request and receive security tokens.

Another important standard is SAML, the *Security Assertion Markup Langauge*, is an industry-recognized XML vocabulary that can be used to represent claims in an interoperable way. ACS accepts SAML tokens as input from identity providers, and issues SAML tokens as output for your applications to consume.

This adherence to standards means that claims-based applications will have more and more options for pluggable identity and access control providers as time goes on.

## Browser-based Applications

Smart clients aren’t the only ones who can participate in the world of claims-based identity. Browser-based applications (also referred to as *passive clients[[9]](#footnote-10)*) can participate as well. Figure 3 shows how this works. The user points her browser at a claims-aware web application. The web application notes that the user is not yet logged in, and so redirects the browser to a web page exposed by the issuing authority that the application trusts (ACS, for example).

The issuing authority is responsible for orchestrating the authentication of the user. In some cases, the authority can do this directly (ACS supports this temporarily while in beta), or it may redirect the browser once again to a web page exposed by an identity provider such as Windows Live ID.

Once the user is authenticated, the authority figures out what claims to issue, and packages them up into a SAML token, which it signs with its private key. The SAML token is then encoded into the response with some javascript so that when the user’s browser receives the response, it automatically sends the token to the application using HTTP POST. If the application wants to establish a logon session for the user at that point, it’ll typically issue a cookie so that the user won’t keep getting bounced back to her identity provider with each request. The WS-Federation specification includes a section[[10]](#footnote-11) that describes how to do these things in an interoperable way.

Can you see the similarity to the web service example in Figure 3? Once again the client visits an issuing authority to get claims, and then sends those claims to the application.

One thing I added in this example is the potential to have multiple issuing authorities, for example, the application might first be redirected to ACS, which would then redirect to Windows Live ID. This can also happen with web services, in which case the path followed by the smart client would be dictated by the policy exposed by the application and all of the issuers along the way.



Figure 4: Basic Scenario with a Web Browser

## Chained Issuers

As I write this paper, ACS is a Community Technology Preview (CTP), and in order to simplify things, it can play the role of an identity provider: you can configure user names and passwords that ACS will use to authenticate end users. But this feature is a temporary measure, and will eventually be removed.

At some point, you’ll need to tell ACS which external identity provider your application wants to use, be it Windows Live ID, or some other provider; perhaps a Geneva Server in your enterprise. And when that happens, you’re going to end up with more than one issuer chained together to issue the claims that your application receives. In this particular example, the reason for using multiple issuing authorities is a simple separation of concerns: ACS doesn’t want to be in the business of authenticating users (any more than you do!) What ACS wants to do is be a claims-transformation engine that you can use to implement role-based access control, personalization, and so on. It will defer responsibility for authenticating users to whatever identity provider you choose.

Using multiple issuers chained together allows you to separate concerns: one can focus on authentication and the other can focus on authorization. This is a very common pattern, and is exactly what you get when you use ACS. But there are other benefits to chaining issuers together.

## Identity Federation across Security Realms

When you build claims-aware web applications and services, you decouple yourself from any one user store. All you want to know is that an authority you trust has given you the identity details you need about the user of your application. You don’t have to worry about what domain or security realm that user happens to be part of. This makes it very natural to *federate identity* across security realms.

Here’s a concrete scenario that will help get your head around this idea. Let’s say a company called Fabrikam is in the business of manufacturing bicycles, and thousands of bike shops around the world carry their bikes. Fabrikam has a website that allows their retailers to get information about bikes, make purchases, and so on.

In a traditional (non-claims based) system, when a new retailer (Bob) starts a business and wants to sell Fabrikam’s bikes, he contacts Fabrikam, signs some agreements, and tells Fabrikam about his employees: who should be allowed to use Fabrikam’s retailer website, who should be allowed to make purchases, and so on. Fabrikam then issues a user name and password for each employee at Bob’s bike shop, and configures its website to grant those users different levels of access depending on their job.

Over time, Bob might do business with lots of other bike manufacturers, each with their own proprietary mechanism for purchasing. Some use the web, and some rely on fax and phone calls. It’s easy for Bob to forget about all of these niggling details when he’s doing his best just to sell bikes every day. So when Alice joins as a new employee, it takes Bob awhile to remember that he has to call Fabrikam (and all of the other manufacturers) and let them know that Alice should be allowed to make purchases. Alice’s first few weeks on the job are a bit daunting as she learns all of the passwords she needs to know for the various systems she’ll be using, and she’ll be denied access to Fabrikam’s retailer website until Bob gets around to calling Fabrikam to add Alice as a user.

What happens when Alice’s role in Bob’s company changes, or even worse, if she leaves the company entirely? When does Fabrikam find out about this?

What we have here are two companies that have established a trust relationship, a covenant, between one another. Fabrikam relies on Bob to indicate which employees in his shop should have access to Fabrikam’s resources, and what level of access each should have. Identity federation is all about *automating this covenant*. Since Fabrikam already trusts Bob to tell the truth about his employees, it makes sense to let Bob’s system authenticate those employees and automatically give Fabrikam the details about each employee’s current role in the company.

Once Bob’s systems are responsible for authenticating his staff, Fabrikam no longer has to issue user accounts for Bob’s employees. When Alice logs into her computer at Bob’s bike shop, that login can be used to tell Fabrikam who Alice is, and what role she plays in Bob’s organization. If Alice leaves the company, all Bob has to remember to do is disable her user account, and she’ll no longer be able to use Fabrikam’s website, or any other manufacturer’s website that federates with Bob’s shop. When Alice changes jobs, and Bob adjusts her group memberships in his directory, Fabrikam discovers that change the next time Alice logs on and uses Fabrikam’s web application.

What we have now is single sign-on across organizations, and this is a *good thing*, not just for developers, but for IT pros, users, and shareholders alike.

Even within a single company, federation can be useful. If you end up with two different implementations, say using Java and Microsoft .NET technologies, as long as your applications are built to support federated identity, you have a clear path to achieve single sign-on, and all of its benefits.

Identity federation works by using chained issuers. Your application still trusts the same issuer it always did, and that issuer will continue to issue all of the tokens that the application needs. But now instead of authenticating all users directly, your issuer will be configured to accept SAML tokens from issuers at partner organizations, leaving them to authenticate users in their own realm in a way that makes sense.



Figure 5: Bob's bike shop federates with Fabrikam

In Figure 5, the client is in a different security realm (Bob’s shop), while the application is still in Fabrikam’s data center. In this case, the client (Alice, say) authenticates with Bob’s issuer (1) and gets a security token that she can send to Fabrikam. This token indicates that Alice has been authenticated by Bob’s security infrastructure, and includes claims that specify what roles she plays in Bob’s organization.

The client sends this token to Fabrikam’s issuer, where it evaluates the claims, decides that Alice should be allowed to access the application in question, and issues a second security token that contains the claims the application expects. The client sends this second token to the application (3), which now has all of the identity details it needs about Alice, and can safely allow her to access the application according to the claims it got from Fabrikam’s issuer.

Note that the relying party didn’t have to concern itself with validating a security token from Bob’s shop. Fabrikam’s issuing authority did all of that heavy lifting: making certain to issue security tokens only to employees of trusted partner companies that have previously established a relationship with Fabrikam. In this example, the application will always get tokens from its own issuer. If it sees a token from anywhere else, it will reject it outright. This keeps the application as simple as possible.

So where does ACS fit in? In Figure 5, Fabrikam could be using ACS as its issuing authority, and Bob could be using Geneva Server to authenticate its clients and issue tokens for them. You see, when you build applications using ACS, you immediately get the option to accept users from other organizations without having to change your applications! If you want to personalize your application depending on the realm the user is from, ACS can help you do this as well.



Figure 6: Cross-Platform Identity Federation

Here’s another scenario. Figure 6 shows one company that uses the Microsoft .NET Framework to build its applications. They have recently merged with a second company whose IT platform is based on Java. Because the applications were built from the beginning to rely on claims, the company was able to install a Java-based issuer and suddenly the Microsoft .NET applications became accessible to users in the Java-based directory, with no changes to the application whatsoever.

The cross-platform possibilities extend beyond this example. While ACS happens to be written using Microsoft .NET and the Geneva Framework, there’s no reason it couldn’t have been written using Java or some other technology. As long as all parties involved follow the standards for federated identity (WS-Trust and WS-Federation), each of the four components in Figure 6 could be built using entirely different technologies and platforms. Once your application is built to rely on ACS, there’s no reason you couldn’t switch if you found a competing issuer that you liked better, even if that issuer happened to be built on a non-Microsoft platform. In the world of federated identity, it’s all about interoperability, and the ACS team is dedicated to fulfilling that ideal.

# Understanding the .NET Access Control Service

The ACS has three surfaces that you can touch: a security token service, which issues security tokens, the administration portal, a web-based user interface that allows you to configure settings that impact how those tokens are constructed, and an administration API. This paper doesn’t yet cover the administration API as it is in great flux right now. But suffice it to say that anything you can do via the administration portal, you’ll be able to do via the API, which will include both REST and SOAP endpoints as well as a set of .NET classes layered on top of those endpoints.

## Getting Started with ACS

In order to get started using the ACS CTP, you must have a Windows Live ID and an invitation code. Surf to http://portal.ex.azure.microsoft.com and sign in using your Windows Live ID. Be sure to click the sign in link indicated by (1) in Figure 7. Then you can click the second sign in link indicated by (2) in order to sign in to the portal. If this is your very first time signing in, you’ll need to type in your invitation code and click the “Sign Up” button, which will create your first *solution*. As of this writing, each solution requires a unique invitation code.

*It’s interesting to note that as you sign into the Azure Services Portal, you’re actually using ACS! The portal web application relies on Windows Live ID and ACS to authenticate and authorize access.*



Figure 7: Signing in to Azure

Solution names must be unique, so you may have to experiment a few times to find a solution name that’s not yet been taken by someone else. Once you create a solution, you’ve essentially created your own private issuing authority in ACS, complete with its own private set of endpoints and corresponding URLs. The solution name becomes part of the URL as you’ll see later.

## ACS in Action: the Calculator Sample

The best way to get your head around ACS is to see it work. So in this section, I’ll walk you through one of the samples packaged with the .NET Services SDK, called the UserNamePasswordCalculatorService.

This sample uses ACS to secure a simple WCF service that offers four operations: Add, Subtract, Multiply, and Divide. The calculator service in this example doesn’t care about the user’s name, email address, or other personal information. All this service cares about is that the user making an Add request (for example) is allowed to add. I’ll show you how to configure a solution in ACS to control access to these operations. In order to get this sample working, you first need to have a “solution” in ACS. I’ll talk more about what a solution is later on, but for now think of it as your own personal security token issuer in the cloud. I will use my own personal solution as an example; its name is “asolution”.

The calculator service is an application that relies on claims issued by ACS to make security decisions. So this service, like any other claims-based application, needs to have its own X.509 certificate that ACS can use for encrypting the security tokens that the service will receive. The sample comes with a simple test certificate called localhost.cer, and that’s what I’ll use to get started.

The Utils subdirectory of this example contains the certificate, along with a batch file (installcerts.bat) that installs it into the local machine’s personal certificate store. I ran that batch file to install the localhost.pfx file on the machine where the service will run, because the service will need access to the private key in order to decrypt tokens the tokens that ACS will produce.

Next I need to tell ACS about the Calculator application, so it knows what to do when a client shows up asking for a security token for that app. The three things ACS needs to know about the application are:

1. It’s name (a URI that clients will use to identify the app)
2. It’s certificate (ACS needs the public key to encrypt tokens it issues)
3. The rules ACS should to issue claims

So I surfed to accesscontrol.windows.net and signed in, then I selected my solution (“asolution”) and chose “Basic Mode”. This allowed me to add an application into my solution, as shown in Figure 8. Note that this is where I specified the application’s name (“Application URL”) and its certificate.

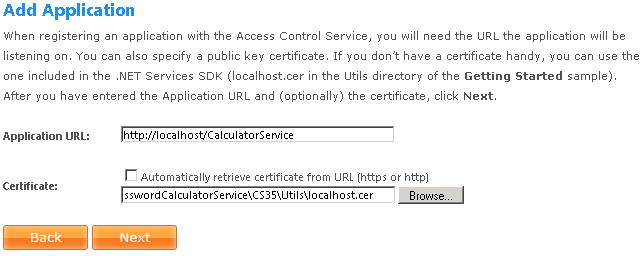


Figure 8: Adding an Application to an ACS Solution

I pressed Next to take me to the rules page, where I configured a few very simple rules for the application. The client program that comes with the Calculator service expects a user name and password: that’s how it authenticates with ACS. And as you’ll learn later in this paper, each solution in ACS comes with a password so that you can use the solution name as a user name and the password to authenticate to ACS. So when I set up my rules, I told ACS to grant permission to add, subtract, multiply, and divide only if the user provided the correct user name and password for the solution.

Figure 9 shows the resulting set of rules.

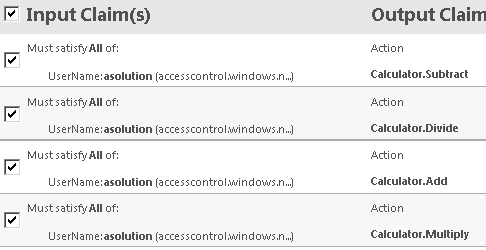


Figure 9: Rules for the Calculator Application

At its heart, ACS is a claims transformation engine, and you can see that by looking at the rules in Figure 9. The rules I specify all look for one incoming claim: a “UserName” claim with a value of “asolution” issued by ACS. The only way a user can get this claim from ACS is by supplying the solution name and its corresponding password when asking ACS for a security token. These four rules ensure that any user that proves knowledge of the password for my solution will get four “Action” claims. An “Action” claim contains a string naming the action, which can be anything: “foo”, “1234”, or “Calculator.Divide”.

So how are these claims used in the Calculator service? The service simply looks at the incoming security token presented by the client, and ensures that the corresponding action is present (see Figure 10).

public class CalculatorService : ICalculator

{

public double Add(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Add");

return n1 + n2;

}

public double Subtract(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Subtract");

return n1 - n2;

}

public double Multiply(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Multiply");

return n1 \* n2;

}

public double Divide(double n1, double n2)

{

AccessControlHelper.DemandActionClaim("Calculator.Divide");

return n1 / n2;

}

}

Figure 10: Code for the Calculator Service

I’ll show the code for the helper method, DemandActionClaim shortly, but I hope you get the idea from Figure 10 that all we’re doing is looking for an action claim with a particular string value before performing the operation. The helper method throws an “access denied” exception if it doesn’t find the specified claim. Ultimately, if you can prove you know the password for the solution, you will be allowed to add, subtract, multiply, or divide. If you don’t, you will be denied access to all of these operations.

public static void DemandActionClaim(string claimValue)

{

foreach (ClaimSet claimSet in OperationContext.Current

.ServiceSecurityContext

.AuthorizationContext

.ClaimSets)

{

foreach (Claim claim in claimSet)

{

if (AccessControlHelper.CheckClaim(claim.ClaimType,

claim.Resource.ToString(),

"http://docs.oasis-open.org/wsfed/authorization/200706/claims/action",

claimValue))

{

if (AccessControlHelper.IsIssuedByIbn(claimSet))

{

return;

}

}

}

}

throw new FaultException("Access denied.");

}

Figure 11: Code for DemandActionClaim helper method

Figure 11 shows the helper method, DemandActionClaim. You may not be aware of this, but WCF was built with claims in mind. See how the OperationContext gives access to a set of claim sets? Each claim set represents a security token. The code here loops through and finds the token (there’s only one in this case, issued by ACS). It then looks for a claim of a particular type (the hardcoded URI that ends in /action identifies an Action claim) and ensures that it has the value specified by the caller (“Calculator.Add”, for example) and that it’s issued by ACS[[11]](#footnote-12). If these requirements aren’t met, the helper method faults with “Access denied.”

Figure 12 shows a run of the client with these rules set up. I made sure I typed in the correct solution name and password so that access would be granted. If I’d typed in an incorrect password, I’d have gotten an exception complaining that I couldn’t be authenticated by ACS.

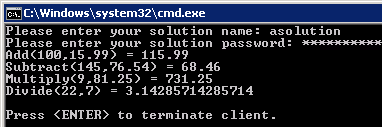


Figure 12: Running the client with all four Action claims

Once I got the solution running, I went back to the rules and disabled Calculator.Divide by simply unchecking its box (see Figure 9). Then I ran the client again. This time, it got only three of the four Action claims, and was denied access to the Divide operation (Figure 13).

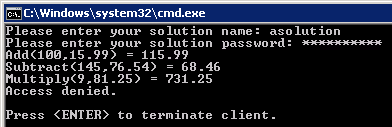


Figure 13: Running the client after disabling one of the Action claims

Next, I tried typing in another solution name and password to ACS. Figure 14 shows the result.

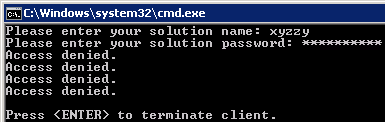


Figure 14: Running the client with a different user name

As you can see, I was able to specify the user name and password for another solution (xyzzy), but recall from Figure 9 that my rules look specifically for a UserName claim with a value of “asolution”, not “xyzzy”. So I ended up with a token with no Action claims in it, and wasn’t able to access any of the operations in the Calculator service (e.g., the DemandActionClaim authorization logic failed in this case).

Finally, I tried specifying “asolution” with an incorrect password, to simulate an attacker who might try to use the application without knowing the correct password. The result? WCF threw an exception that indicated that the caller could not be authenticated. So ACS didn’t even try to issue a token in this case, because it couldn’t verify the user name and password that I supplied.

If you examine this SDK sample further, you’ll see that it uses the claims programming model built into WCF today (System.IdentityModel). It’s also possible to pull in the Geneva Framework and use its programming model instead. The Geneva Framework is a newer approach to programming claims-based applications, which tends to be more flexible and feature rich, and more accurately represents the future of claims-based programming in Microsoft .NET going forward[[12]](#footnote-13). And since ASP.NET 2.0 doesn’t natively support claims-based programming, the Geneva Framework is your best option for using claims in that scenario, as you’ll see in my browser-based example at the end of this paper.

# Configuring the .NET Access Control Service

This section drills into the various ACS configuration options and further explains identity concepts.

## Accounts and Solutions

Each account in ACS is associated with a single Windows Live ID (WLID). Each ACS account has a set of solutions. In ACS, you can think of a solution as a virtual issuing authority with its own private STS and a set of Web pages where browsers can be redirected in order to obtain SAML tokens (see Figure 15).



Figure 15: Accounts and Solutions

A solution is also where you create rules that determine what claims will be in the SAML tokens issued by ACS. Each solution can have completely different rules.

In the CTP release, each solution includes a set of credentials including a password. You can use the solution name and password to authenticate to your private issuer and request a SAML token, but since there’s only one user name and password, this isn’t going to scale to support many users. Keep in mind that solution credentials are really only for development and testing. If you want to test with certificates or information cards, you can do that as well, as each solution currently allows you to associate personal (self-issued) information cards as well as X.509 client certificates. Later in this paper I’ll talk about how to use identity providers like Windows Live ID and Geneva Server to authenticate users.

An ACS solution contains rules that can drive token issuance for many different applications, but most applications are probably going to need different types of claims. One application might need to send email to its users, and would therefore expect an email claim to be included in the token issued by ACS, while another might not need the user’s email address, and for privacy considerations, therefore shouldn’t have access to the user’s email. For example, an expense reporting application might want to know whether the user is a ProjectManager or a ProjectMember, and would therefore need claims representing these roles. But a weather reporting application wouldn’t care about claims like that, and would instead be more interested in the user’s current geographic location.

Clearly each application needs its own claims, and therefore its own set of rules so that ACS can generate those claims. This idea is so important that WS-Trust defines the request for security token (RST) to include an AppliesTo field, which is a URI identifying the logical destination for the token. ACS makes use of this, by allowing you to assign a URI to each of your applications.

## Solution Scopes

In ACS, each application within a solution must be assigned a unique URI. This is the AppliesTo URI that I mentioned in the previous section. While technically this can be any string that follows the rules for constructing URIs[[13]](#footnote-14), generally for web applications and services, most people find it convenient to use a URL associated with the application, for example, http://www.fabrikam.com/expenseReportingService. The obvious benefit is that it becomes immediately obvious what application the name refers to, but another benefit is that it’s harder to accidentally give the same URI to two different applications.

Each ACS solution can drive several applications, so it follows that each application should have its own *scope* in the solution, with settings that make sense for that application. That’s why in Figure 15 you can see that each solution has a set of scopes. A scope is nothing more than the settings for a particular application. In order to help make things easy, as of this writing, ACS include a “basic mode” that replaces the word “scope” with “application”, and dramatically reduces the number of settings that can be configured (Figure 16). This is what I used earlier in the example with the Calculator service.

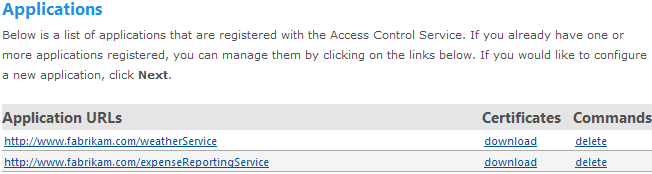


Figure 16: Scope == Application (Basic Mode)

Once you realize that a scope is really just a bundle of settings for an application, you’ll find that “advanced mode” isn’t confusing at all (Figure 17), and indeed is going to be required for doing anything interesting in ACS. So from now on, I’ll be using terminology and screenshots from advanced mode.[[14]](#footnote-15)

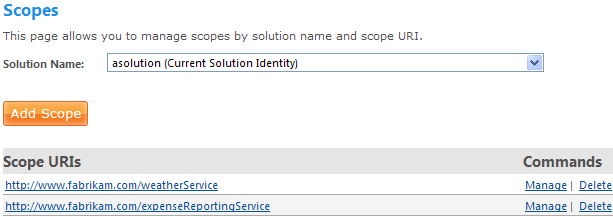


Figure 17: Scope == Application (Advanced Mode)

## Settings in a Scope

A scope is a group of settings for an application (I promise this is the last time I’ll reiterate this). These settings include the rules that determine how ACS generates claims for the application, but also include other important things such as the public key that ACS should use to encrypt the security token.



Figure 18: Settings in a Scope

I’ll cover the simpler settings here, and treat the more interesting settings in their own dedicated sections below. One of the simplest settings is *expiration*. To enhance performance and increase scalability, security tokens have an expiration date. They can be reused by the client until they expire, which reduces the number of times a client has to request a token from an issuer, reducing latency for the client and load on the issuer. By default, SAML tokens issued by ACS are valid for 8 hours from the time of issuance (or less if the client requests a shorter time to live). Each scope has settings for these maximum allowed expiration times, and you can adjust them if you like

Another important setting is that of *encryption*. ACS encrypts all of the security tokens that it issues, but clearly each token must be readable by the target application. The encryption settings in a scope allow you to upload a public key (technically an X.509 certificate, which contains a public key) to ACS. This is the public key that will be used to encrypt all tokens issued for the application that the scope represents. The application will need access to its private key in order to decrypt these tokens, and your framework of choice (WCF, Geneva Framework, etc.) will allow you to choose a private key, typically by selecting a certificate from a certificate store that includes a private key[[15]](#footnote-16).

By definition, when you create a scope within a solution, that solution owns the scope, and the person who owns the solution can always edit the scope. But if you want to grant edit permission on the scope to someone else, you can use the permissions setting on the scope to allow that. The way you do that is by specifying another solution in ACS that you want to have permission to edit the scope. This might seem a little weird at first, but remember that each solution is owned by a single Windows Live ID (WLID), so in effect, you’re giving edit permission to the solution’s WLID by doing this. The owner of that solution can now edit your scope. *Be very careful with this setting:* whomever you grant permission can edit your scope just like you can, including making changes to your rules.[[16]](#footnote-17)

## Identity Issuers

This scope setting is one of the most important to get right, because it’s how you specify which identity providers can authenticate users for your application. Each additional identity provider adds a new group of users that can potentially access your application. This is identity federation in action!

As you can see in Figure 19, ACS accepts WLID as an identity provider by default. And remember that in the CTP release, ACS has its own little identity provider based on solution credentials that you can use for testing, which accounts for the two issuers that start with accesscontrol.windows.net. Those will be going away in the future, so I’m not going to focus on them.

What I do want to focus on is Fabrikam, which is an issuer that I added myself by clicking the “Add Issuers” button. Here I was able to specify three things: a display name, URI, and a certificate. The URI identifies the issuing authority over at Fabrikam Corporation, just like each scope URI identifies an application. I got the certificate from Fabrikam and gave it to ACS on this page. This way when a user from Fabrikam shows up looking for a token for my application, ACS will know how to verify the signature on the SAML token sent from Fabrikam’s identity provider.

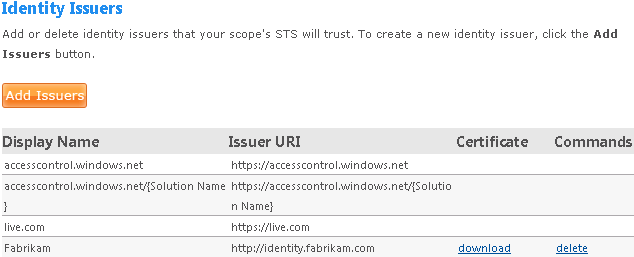


Figure 19: Identity Issuers

Both Fabrikam and live.com are trusted issuers in this scope. I want to accept SAML tokens from either one. I don’t have to worry what platform they run, or what technology they use to generate their tokens (although I think you can probably guess what live.com is using: some Microsoft .NET technology such as the Geneva Framework). As long Fabrikam’s identity provider supports WS-Trust 1.3 and generates SAML tokens that ACS can consume, it’s a perfectly good candidate for a trusted issuer. If Fabrikam happened to be using Windows with Active Directory to manage their user accounts, the obvious choice for their identity provider would be Geneva Server, for example.

The ACS team has made it clear that interoperability is an important design goal of theirs. The ACS CTP currently supports Windows Live ID as an identity provider, as well as Geneva Server, but ACS is also testing with other major vendors of federated identity products, including Tivoli, Ping Identity, IBM, etc.

## Claim Types

Each claim has a type and a value. The type of the claim is identified by a unique URI, and it’s up to you as the developer to understand what your issuer means when it issues a claim of a particular type.

Usually it’s pretty straightforward. For example, an email claim has a URI of “http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress”, and the value of the claim is simply a string with an email address. You can make claims be as complicated as you like, as long as the application consuming the claim agrees with the issuer as to what the claim means and how the value is structured, you’ll be in good shape. ACS natively supports several claim types, as shown in Figure 20.

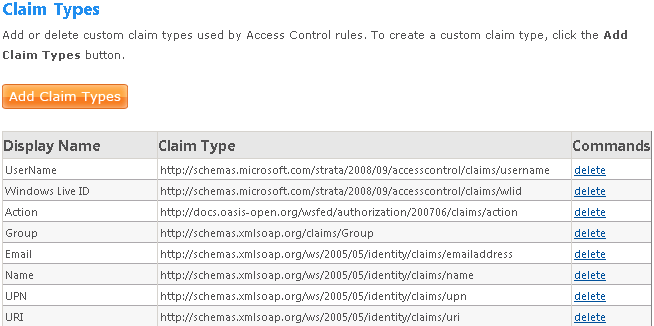


Figure 20: Claim Types in ACS

In the CTP release, a caller can get UserName claim issued by accesscontrol.windows.net by having someone authenticate using a solution name and a credential associated with the solution (a password, information card, or certificate). The UserName in this case will be the solution name. But as I mentioned before, this option won’t be around for long. Going forward it’s more likely that you’ll rely upon UserName or UPN[[17]](#footnote-18) issued from Geneva Server, or a Windows Live ID.

Email and Name are often used to personalize an application. Name is generally something that you can display to a user[[18]](#footnote-19) to indicate the identity she used to log in to your application, and Email of course has the obvious use of allowing you to communicate with the user.

You’ve already seen the Action claim – it can be used to grant access to particular operations, and its value is an arbitrary string that your application looks for. The Group claim is really powerful: it is typically used to build Role Based Access Control (RBAC) logic in conjunction with Action claims, and I’ll show an example of this later on, in the section called Implementing RBAC. Group claims are easy to generate from Geneva Server, where you can easily map a Windows group onto a Group claim.

As shown in Figure 20, you can also add your own custom claim types; all you need is a display name and a URI, and an understanding in your application of how to interpret the incoming claims.

## Claims Transformation

The Calculator example I showed earlier only scratches the surface of what you can do with ACS rules. Let’s say that your identity provider is willing to supply you with the user’s email address. You can pass this value through to your application as shown in Figure 21. Here I’ve taken an input Email claim from an issuer that I trust (fabrikam.com) and I’ve passed it through as an output claim so that my application can discover and make use of it. Note the use of \* to accept any email claim, and the checkbox, “Copy input value” to copy the claim through to the output claim set.

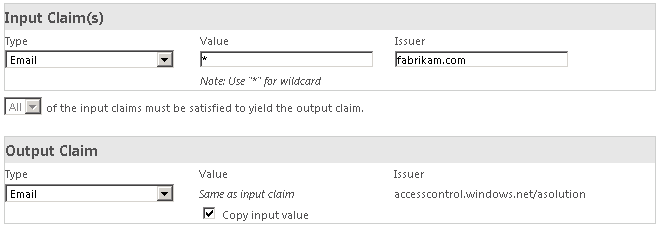


Figure 21: Passing through an Email claim

This is a simple transformation: the email claim was originally issued by Fabrikam.com, but when your application receives it, it will have been issued by ACS. More interesting transforms are also possible. For example, let’s say your application wants to know if the user is a member of a logical role called Managers. And let’s say you have several different organizations, each of which is using Geneva Server to federate identity with your application. Each of these organizations might use a different word for something that logically means, “Manager” to your application.

Figure 22 shows how to map these onto a single Group claim called “Manager”, taking into consideration the various identity providers and their particular vernacular. These are the sorts of transformations that make application developers content! Being able to simply check HttpContext.User.IsInRole(“Manager”) is a lot simpler than having to check what organization the user is from, and then checking for the “Executives” or “Managers” role.

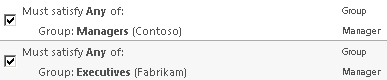


Figure 22: Using claims transformation with role names

# Implementing Role Based Access Control (RBAC)

It turns out that ACS implements forward chaining of rules so that you can transform claims in several stages, which makes it pretty easy to implement a hierarchical RBAC system. These systems typically take incoming groups of users, map them onto roles, and then map those roles onto logical operations that the users are allowed to perform. The Group and Action claims are useful for this type of thing.

In the previous example (Figure 22), I modeled a logical role called “Manager” with a Group claim. Figure 23 extends that example by looking for the “Manager” role and mapping that onto a couple of logical operations. These operations are modeled by Action claims, and my hypothetical expense reporting application will be looking for these.



Figure 23: Group to Role to Action claims transformation leads to RBAC

The result of this example is that users from Contoso may view and approve expense reports only if the identity provider at Contoso indicates that they are members of the “Managers” group. Similarly, users from Fabrikam must have the “Executives” group claim in order to view and approve expense reports. Once again, the application developer need not sweat these details: she just looks for the Action claim that corresponds to each operation to decide whether or not to grant access.

There’s no theoretical limit to the depth of role/operation hierarchy. I could add another logical role called Employee and grant it limited permission to my expense reporting application. Then, if I wanted to ensure that all members of the Manager role also get the rights afforded to any Employee, I would simply add another rule that looked for the Manager Group claim as input (issued by ACS) and output an Employee Group claim as output.

ACS will run the resulting logical graph to transitive closure, which in not so academic terms, means it’ll flatten out the resulting claims so that a Manager will have the Employee Group claim as well as the union of all Action claims that result from having either the Manager or Employee Group claims.[[19]](#footnote-20) And nothing stops me from looking for a specific Action claim and having that trigger the generation of more Action claims. ACS doesn’t care, as long the resulting graph doesn’t have any cycles.

# Passive Examples

In this section of the paper, I’ll show how you can integrate ACS into a browser-based web application using ASP.NET and the Geneva Framework.

## Integrating ACS into an ASP.NET Web Application

This is an example of a browser-based web application that uses ACS to authenticate and authorize users. The application is very simple: it’s the front end for a tool rental company called Contoso Woodworking. This company wants to rent tools like screwdrivers, planes, power saws, etc. and wants to ensure that only authorized users are allowed to rent any tools. Furthermore, they need to be certain that only users who have signed a special waiver can rent power tools. Because they want to support users ranging from the hobbyist to large corporations, they decided that ACS was a great solution.

I’ll start by showing how to use Windows Live ID to authenticate individuals such as hobbyists or users from corporations that don’t have federated identity solutions in place. Only authenticated users that have signed up to use the service can view or rent tools from Contoso Woodworking. And of those users, only ones who have signed a special waiver are allowed to view or rent power tools. But any anonymous user is allowed to browse to the home page (see Figure 24).

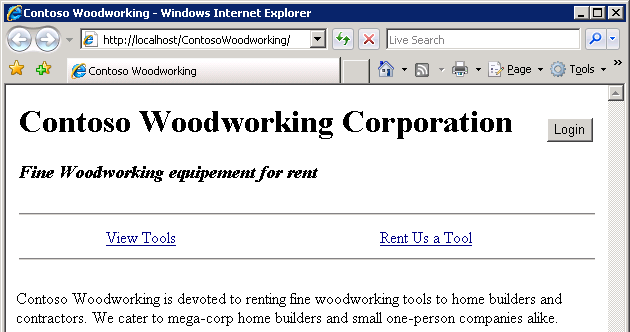


Figure 24: Home page as seen by anonymous users

Once a user browses to the home page, in order to use the service, she will need to log in by clicking the Login button. From there, she will be redirected to ACS, which will immediately redirect to WLID in order to allow the user to sign in (see Figure 25).

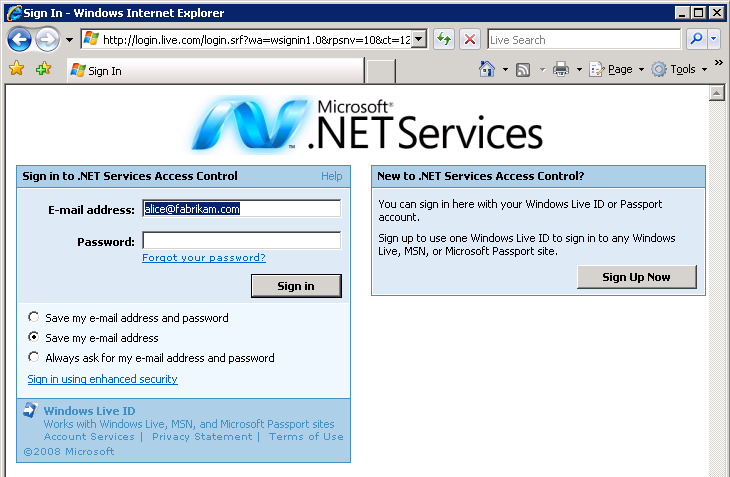


Figure 25: Logging in with Windows Live ID

Once the user (say, Alice) successfully signs in, WLID redirects the browser back to ACS, which processes the claims from WLID, runs the rules for the scope to generate claims for Alice that make sense to Contoso Woodworking, and then POSTs those claims in the form of an encrypted SAML token back to Contoso Woodworking’s home page. The effect that Alice sees is very natural: she clicks the login button and sees a WLID sign in page. Signing in takes her back to the Contoso Woodworking site, which indicates that she is logged in (the “Login” button has now become a “Logout” button) (Figure 26).

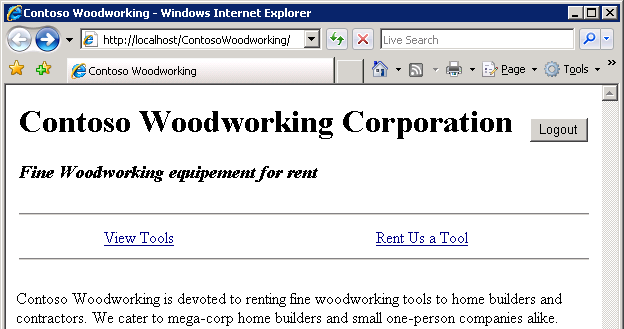


Figure 26: Home page after logging in

Since Alice has previously signed up with Contoso to use their service, she can click the View Tools link in order to see the selection of hand tools (Figure 27). But since she’s not yet signed the power tools waiver, she doesn’t see any power tools.

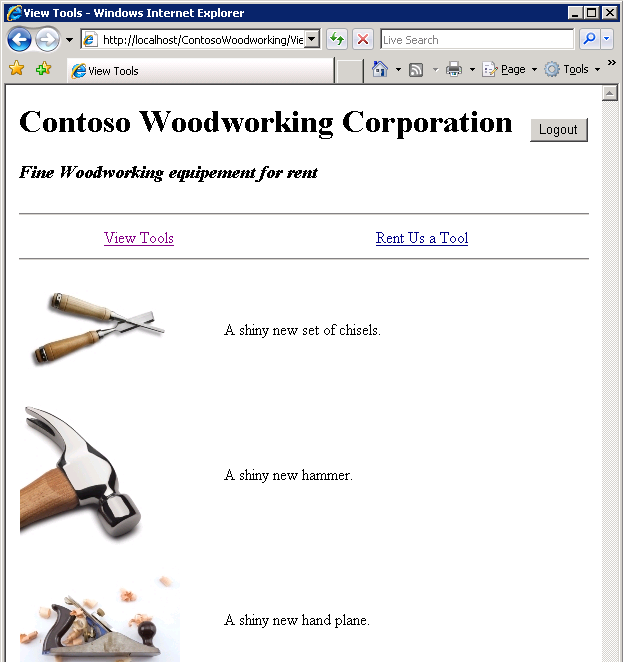


Figure 27: Normal user can only view "safe" tools

Figure 28 shows the rule that Contoso added when Alice signed up to use their service. It looks for Alice’s WLID and maps that onto the “ContosoWeb.ViewTools” Action claim, which Contoso Woodworking uses to decide whether or not to show any tools.

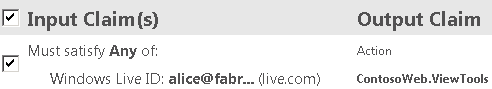


Figure 28: Configuring ACS to grant Alice permission to access normal tools

Once Alice signs the power tools waiver, Contoso will grant her permission to access the power tools for Contoso Woodworking. All they need to do is update the rules and grant Alice the “ContosoWeb.PowerTools” Action claim, as shown in Figure 29.

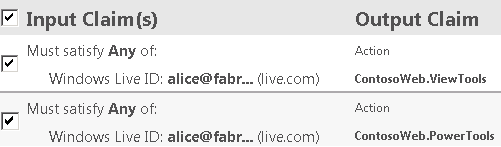


Figure 29: Adding an Action claim that allows Alice to access power tools

The next time Alice logs in, she’ll see the power tools as well (Figure 30).

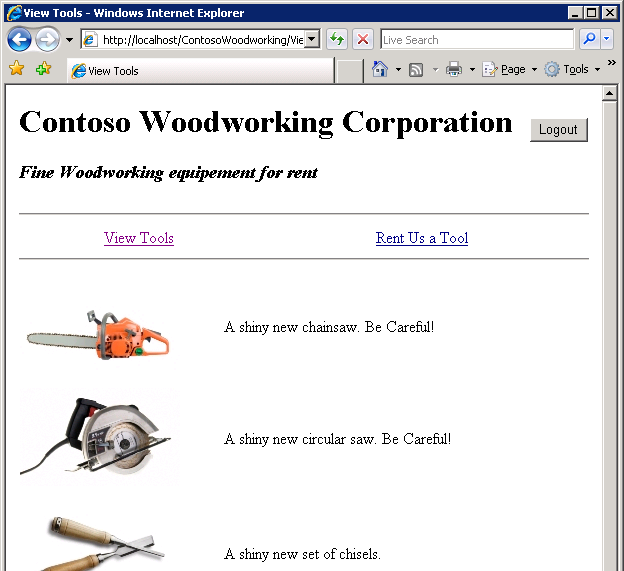


Figure 30: After signing a waiver, user can see power tools as well

This is a very simple example designed to show, in a very direct way, how you can use ACS to implement authentication and access control in a web application. This direct approach of granting Action claims to individual users isn’t the most scalable approach: it could be rather cumbersome to manage thousands of users this way. But ACS can help you scale up when you need to.

A good approach is to adopt a more hierarchical approach as I described in the section on Role Based Access Control. Once you’ve assembled your users into groups via ACS Group claims, it becomes trivial to add appropriate Action claims to each of your groups. And if you get enough rules that the ACS portal becomes cumbersome to use, you can use the administration API to manage rules programmatically.[[20]](#footnote-21)

So far, you’ve seen how to use Windows Live ID to authenticate users. This is helpful for the individual hobbyist or corporations that don’t have federated identity infrastructure in place. But imagine that Fabrikam Corp has deployed Geneva Server (or any other identity federation product) and has signed an agreement that allows certain Fabrikam employees to rent tools from Contoso Woodworking.

To implement this, Fabrikam configures its federation server to issue a Group claim for Fabrikam users that should be allowed to rent tools. I’ve called it “ToolUsers” for the sake of this example. Contoso needs to configure ACS to accept SAML tokens issued by Fabrikam’s identity provider (look back at Figure 19 to recall how this trust can be established by uploading Fabrikam’s certificate to ACS). Let’s say that Contoso has decided that it makes business sense to have Fabrikam manage the power tools waiver signing process themselves. So Fabrikam has its federation server issue a second Group claim for users who have signed the waiver. Figure 31 shows how Contoso would configure ACS to map these Group claims from Fabrikam onto the corresponding Action claims that the application understands.



Figure 31: Mapping Fabrikam's group claims to Contoso Woodworking Action claims

If Contoso decides to implement a hierarchical RBAC scheme, they might instead choose to simply map Fabrikam Group claims on to Group claims of their own, which then would map onto Action claims.

With these configuration changes in place, Fabrikam’s users will have single sign-on access to Contoso Woodworking’s website. Fabrikam users don’t need a Windows Live ID to rent tools. And did you notice that Contoso Woodworking didn’t need to issue a single user name and password to any of its users? This is a huge cost savings; managing user identities can be painful and expensive, and ACS helps you get out of that business as an application developer.

## The Code behind Contoso Woodworking

The code for Contoso Woodworking isn’t included in the .NET Services SDK samples, but you can get it from Justin Smith’s blog (<http://blogs.msdn.com/justinjsmith/>).

ASP.NET 2.0 didn’t ship with support for federated sign in, so the logical choice for adding that support is the Geneva Framework. If you look at the web.config file for the Contoso Woodworking website (Figure 32), you’ll see how the Geneva Framework is incorporated.

<configuration>

<configSections>

<section name="microsoft.identityModel"

type="...MicrosoftIdentityModelSection..."/>

</configSections>

<system.web>

<compilation>

<assemblies>

<!-- standard ASP.NET assemblies omitted for brevity -->

<add assembly="Microsoft.IdentityModel, ..."/>

</assemblies>

</compilation>

<!-- ACS will handle authentication, not ASP.NET -->

<authentication mode="None"/>

</system.web>

<system.webServer>

<modules>

<add name="WSFederationAuthenticationModule"

type="Microsoft.IdentityModel..."

preCondition="managedHandler"/>

<add name="SessionAuthenticationModule"

type="Microsoft.IdentityModel..."

preCondition="managedHandler"/>

</modules>

</system.webServer>

<microsoft.identityModel>

<serviceCertificate>

<certificateReference

findValue="CN=ContosoWoodworking"

x509FindType="FindBySubjectDistinguishedName"

storeLocation="LocalMachine"

storeName="My"/>

</serviceCertificate>

<audienceUris>

<add value="http://localhost/ContosoWoodworking/Default.aspx"/>

</audienceUris>

<federatedAuthentication enabled="true"/>

<issuerNameRegistry type="...">

<trustedIssuers>

<add thumbprint="416e6fa5d982b096931fbf42c4a3dcd608856c95"

name="http://accesscontrol.windows.net/asolution/"/>

</trustedIssuers>

</issuerNameRegistry>

</microsoft.identityModel>

</configuration>

Figure 32: Contoso Woodworking's web.config file

Here are some interesting points to note about the config file:

* <configSections> declares a section for microsoft.IdentityModel (the Geneva Framework)
* <assemblies> pulls in a reference to the Geneva Framework assembly from the GAC[[21]](#footnote-22)
* <modules> wires in the WSFederationAuthenticationModule, which enables federated authentication, as well as the SessionAuthenticationModule, which implements a cookie-based logon session so that the user isn’t constantly being asked to authenticate (both of these modules are from the Geneva Framework)
* <serviceCertificate> tells the Geneva Framework which certificate should be used to decrypt SAML tokens from ACS
* <audienceUris> tells the Geneva Framework where to POST SAML tokens
* <issuerNameRegistry> tells the Geneva Framework that it should only trust SAML tokens signed by ACS and issued specifically for the corresponding solution (my solution is called “asolution”; if you run this example yourself, you’ll replace this with your own solution name)

When an anonymous user points her browser to Contoso Woodworking (or any other website that uses ACS, for that matter), her browser is redirected to ACS, which then needs to redirect once again to the user’s identity provider. Since there can be many different identity providers (WLID is only one possibility – I also mentioned Fabrikam as an issuer as well), it can be a bit tricky to figure out which identity provider makes sense for a particular user. This process is known as *home realm discovery*, and in its current incarnation, ACS requires you to supply a hint via the standard WS-Federation query string parameter called *whr*. This parameter tells ACS which *home realm* the user belongs to.

The Geneva Framework supports the whr parameter, but it currently takes a little bit of boilerplate code to inject it, which means that I can’t use the FederatedPassiveSignIn control that’s built into Geneva Framework to redirect to ACS. This should be corrected by the time the Geneva Framework ships, (probably by adding a parameter to the control allowing you to specify the home realm, but we’ll see how that shakes out). Figure 33 shows the boilerplate code in global.asax that’s used to handle the redirect to ACS for sign in. Note that the home page (Default.aspx) is excluded from this redirect so that anonymous users can view the home page.

You may have noticed that this example hardcodes the whr parameter to “http://login.live.com” because WLID is used to authenticate all users in this sample. If you wanted to implement identity federation (for Fabrikam Corp, say), you’d need to have some way for the user to indicate whether she has Fabrikam credentials or if she uses a Windows Live ID to access the tool rental service.

One way to do this is to give the user multiple login buttons (“Login using my Windows Live ID”, and “Login using my Fabrikam ID”), but a more transparent approach is to give all of the Fabrikam users a different link to the Contoso Woodworking website, perhaps using a query string argument to specify the user’s home realm (realm=Fabrikam, for example). Each additional company that you federate with would have its links customized to indicate the name of their home realm. With this hint in place, you’ve got everything you need to redirect the user to ACS correctly and support multiple identity providers.

<%@ Application Language="C#" %>

<%@ Import Namespace="Microsoft.IdentityModel.Web" %>

<%@ Import Namespace="Microsoft.IdentityModel.Claims" %>

<%@ Import Namespace="Microsoft.IdentityModel.Protocols.WSFederation" %>

<script runat="server">

void Application\_AuthenticateRequest(Object sender, EventArgs e)

{

if (Request.Path.EndsWith("Default.aspx", StringComparison.OrdinalIgnoreCase))

return;

IClaimsIdentity identity =

HttpContext.Current.User.Identity as IClaimsIdentity;

if (identity != null) return;

WSFederationAuthenticationModule authModule =

new WSFederationAuthenticationModule();

// scope name

authModule.Realm = "http://localhost/contosowoodworking/Default.aspx";

// here's my own personal issuer URL for asoltuion

authModule.Issuer = "https://accesscontrol.windows.net/" +

"passivests/asolution/livefederation.aspx";

String uniqueId = Guid.NewGuid().ToString();

SignInRequestMessage signInMsg = authModule.CreateSignInRequest(

uniqueId, authModule.Realm, false);

// Here's the whole reason we need to do this manually

signInMsg.Parameters.Add("whr", "http://login.live.com");

// Redirect to ACS, which will in turn redirect to WLID

Response.Redirect(signInMsg.RequestUrl);

}

</script>

Figure 33: Global.asax for Contoso Woodworking website

Now that I’ve shown you how sign in works, let’s focus on where the sample makes use of the user’s login and claims to customize its look and functionality. One simple place where this is done is the Login button, which changes to “Logout” when the user is signed in. This isn’t really claims-specific, and you can see by the code in Figure 34 how the Geneva Framework makes it easy to incorporate claims into an ASP.NET application using familiar techniques such as HttpContext.User.

protected void Page\_Load(object sender, EventArgs e)

{

loginButton.Text = Page.User.Identity.IsAuthenticated ?

"Logout" : "Login";

}

Figure 34: Setting the Login button text in the Banner user control

Figure 35 shows the code that displays different sets of tools depending on the Action claims in the user’s token. Note how this code selectively binds to different sets of data depending on the claims.

protected void Page\_Load(object sender, EventArgs e)

{

if (ClaimsVerification.ViewToolsPermission)

{

if (ClaimsVerification.PowerToolsPermission)

toolList.DataSource = ToolUtility.AllTools();

else toolList.DataSource = ToolUtility.HandTools();

toolList.DataBind();

}

}

Figure 35: Selectively binding to data depending on claims

The above code uses a helper class called ClaimsVerification, which uses the Geneva Framework to search the user’s token for Action claims. One easy way to do this is by using LINQ (see Figure 36).

public static Boolean PowerToolsPermission

{

get

{

IClaimsIdentity identity =

Thread.CurrentPrincipal.Identity as IClaimsIdentity;

if (identity == null) return false;

return identity.Claims

.Where(claim => claim.ClaimType.Equals(ActionClaim) &&

claim.Value.Equals("ContosoWeb.PowerTools"))

.Count() > 0;

}

}

Figure 36: Using LINQ to search for the ContosoWeb.PowerTool Action claim

As you can see, it’s really not that hard to get started with ACS today in an ASP.NET application, and you can be sure that it’ll get easier in the future, as more tooling goes into Visual Studio to support the Geneva Framework and ACS.

# Summary

Claims-based programming is the future of identity on the Microsoft Windows platform, and ACS is a great way to get started. By adopting a claims-based approach to identity, your applications will benefit from single sign on, and your application developers will no longer have to worry about how they are going to authenticate users, which is a complicated and expensive business to be in. ACS can further simplify applications by handling much (if not all) of the authorization logic. You can get started today by designing applications that are built to accept claims instead of doing their own authentication!

# Additional Resources

We’ve provided links to several resources below that will further your education on the suite of Microsoft® .NET Services and the .NET Access Control Service in particular.

## Microsoft® .NET Services Whitepaper Series

* An Introduction to Microsoft .NET Services for Developers
  + <http://go.microsoft.com/?linkid=9638347>
* A Developer’s Guide to the Microsoft® .NET Service Bus (*this paper*)
  + <http://go.microsoft.com/?linkid=9638348>
* A Developer’s Guide to the Microsoft® .NET Access Control Service
  + <http://go.microsoft.com/?linkid=9638349>
* A Developer’s Guide to the Microsoft .NET Workflow Service
  + <http://go.microsoft.com/?linkid=9638350>

## .NET Access Control Service Resources

* Microsoft Code Name “Geneva” Framework Whitepaper for Developers
  + <http://download.microsoft.com/download/7/d/0/7d0b5166-6a8a-418a-addd-95ee9b046994/GenevaFrameworkWhitepaperForDevelopers.pdf>
* Justin Smith’s blog
  + <http://blogs.msdn.com/justinjsmith/>

# About the Author

Keith Brown is a cofounder of Pluralsight, a premier Microsoft .NET training provider offering both instructor-led and online training courses. Keith has authored books on Windows security, and spent eight years as the security columnist for MSJ and MSDN magazines. Keith has spent over a decade developing courses, speaking at conferences, and teaching developers about security. You can reach him at http://www.pluralsight.com/keith.

# Acknowledgements

This paper wouldn’t have been possible without the tremendous help from Justin Smith of Microsoft. His PDC sessions coupled with his personal guidance and sharp eye during editing drove this whitepaper. Thanks Justin!

1. Microsoft® .NET Services is the new, more appropriate name, for the original *BizTalk Services* initiative. [↑](#footnote-ref-2)
2. For more information on Microsoft® .NET Services, the .NET Access Control Service, and the .NET Workflow Service, see the accompanying papers in the Microsoft .NET Services Whitepaper Series referenced at the end of this paper. [↑](#footnote-ref-3)
3. [Phishing](http://en.wikipedia.org/wiki/Phishing) is all about convincing a user to divulge sensitive information (such as passwords). This is commonly done by sending an email that masquerades as being from a legitimate company with which the user may have an account. The email includes a link that leads to the attacker’s website, convincingly built to look like the legitimate company’s website. When the user “logs on”, her password is captured by the attacker, along with any other information the user is duped into giving away. [↑](#footnote-ref-4)
4. SAML is the Security Assertion Markup Language, an XML vocabulary well suited for representing claims. SAML tokens have been in use for years in the industry in both Microsoft .NET systems as well as Java-based systems. [↑](#footnote-ref-5)
5. Not to be confused with WS-Policy. [↑](#footnote-ref-6)
6. ACS also includes SOAP and REST endpoints for programmatic administration, as well as a set of .NET classes that simplify calling these endpoints. So you can build your own administration console if you don’t like the one ACS provides, or if you want problem domain-specific customizations. [↑](#footnote-ref-7)
7. WSDL is the Web Service Description Language. [↑](#footnote-ref-8)
8. As of this writing, the latest version of WS-Trust is 1.3, and in order to use ACS, your web service stacks in both the client and service must support this. Microsoft supports WS-Trust 1.3 in WCF version 3.5 and above, as well as the Geneva Framework. The Java world has support for this as well (the [Metro](https://metro.dev.java.net/) web service stack is one example). [↑](#footnote-ref-9)
9. Smart clients are referred to as “active” because they have plumbing (WCF, for example) that can parse policy and implement WS-Trust directly. Web browsers are referred to as “passive” because they aren’t expected to know anything about policy and WS-Trust, so browser-friendly techniques such as query strings, redirection, and javascript are used to flow claims from issuer to application, with SSL being used to mitigate attacks such as server spoofing, eavesdropping, and tampering. [↑](#footnote-ref-10)
10. Section 13, to be precise. You may have heard this referred to in the past as the *passive requestor profile*, although as of this writing, the latest version of WS-Federation undergoing standardization no longer uses this term. [↑](#footnote-ref-11)
11. “Ibn” apparently was an old internal code name for ACS, so don’t let the IsIssuedByIbn helper method confuse you. [↑](#footnote-ref-12)
12. The only reason the ACS team used the WCF programming model in the calculator sample was due to shipping considerations (they weren’t allowed to redistribute the Geneva Framework at the time the SDK shipped). [↑](#footnote-ref-13)
13. For example, *urn:foo* is a valid URI, although it isn’t terribly unique. [↑](#footnote-ref-14)
14. By the time ACS is officially released, expect this notion of “basic” and “advanced” modes to go away. There’ll only be one mode, which (in my humble opinion) will actually make the ACS portal less confusing. [↑](#footnote-ref-15)
15. Note that this private key is not uploaded to ACS – the private key, by definition, is private to your application. When you upload a certificate to ACS, you’re only sending the public key. [↑](#footnote-ref-16)
16. Note that the .NET Service Bus and .NET Workflow Service both use this technique to allow you to tweak settings in the scopes that they create. [↑](#footnote-ref-17)
17. UPN == User Principal Name, a term that comes from Active Directory and is used to identify a user in a particular security realm. UPN uses a format similar to an email address (e.g., [alice@fabrikam.com](mailto:alice@fabrikam.com)). [↑](#footnote-ref-18)
18. Although you should always be careful when using the values of claims in your programs, especially claims that might originate from your users, such as Name claims. Assume these are just another form of user input, and you should take every precaution to avoid attacks like Cross Site Scripting by properly encoding these values for output. [↑](#footnote-ref-19)
19. If you’ve used Authorization Manager (AzMan) in the past, you’ve seen this sort of hierarchy before. [↑](#footnote-ref-20)
20. I mentioned earlier that this API is undergoing dramatic changes at this time, but if you want to see some sample code for its current incarnation, check out the two samples found in the Samples\AccessControl\ExploringFeatures\AccessControlManagement subdirectory of the .NET Services SDK. [↑](#footnote-ref-21)
21. GAC == Global Assembly Cache [↑](#footnote-ref-22)