David Chappell
& Associates

CLAIMS-BASED IDENTITY FOR WINDOWS

TECHNOLOGIES AND SCENARIOS

DAVID CHAPPELL

FEBRUARY 2011

SPONSORED BY MICROSOFT CORPORATION

CONTENTS

Understanding Claims-Based Identity 3

The Problem: Working with Identity in Applications 3

The Solution: Claims-Based Identity 4

Claims, Tokens, and STSs 4

Identity Providers and Identity Libraries 5

Using Multiple Identity Providers 7

Federation Providers 8

Implementing Claims-Based Identity: Microsoft Technologies 10

Windows Live ID 11

Active Directory Federation Services 2.0 11

Windows Azure AppFabric Access Control 12

Windows Identity Foundation 13

Using Claims-Based Identity: Scenarios 13

On-Premises Scenarios 14

Accessing an Enterprise Application 14

Accessing an Enterprise Application via the Internet 15

Providing Single Sign-On to an Enterprise Application in Another Organization 16

Cloud Scenarios 19

Providing Single Sign-On to an Enterprise Application in the Cloud 19

Providing Single Sign-On to a SaaS Application 20

Allowing Logins with Facebook, Google, and Other Cloud Identity Providers 22

Conclusion 24

About the Author 24

UNDERSTANDING CLAIMS-BASED IDENTITY

For people who create applications, working with identity traditionally hasn't been much fun. First, a developer needs to decide which identity technology is right for a particular application. If the application will be accessed in different ways, such as within an organization, across different organizations, and via the public Internet, one identity technology might not be enough—the application might need to support multiple options. The developer also needs to figure out how to find and keep track of identity information for each of the application's users. The application will get some of what it needs directly from those users, but it might also need to look up other information in a directory service or someplace else. IT administrators must also be involved to configure this software correctly. Add the cloud to the mix, and things get even more complicated.

This is all more complex than it needs to be. Why not create a single interoperable approach to identity that works in pretty much every situation, both on-premises and in the cloud? And rather than making applications hunt for identity information, why not make sure that this single approach lets users supply each application with the identity information it requires?

Claims-based identity achieves these goals. It provides a common way for applications to acquire the identity information they need about users inside their organization, in other organizations, and on the Internet. It also provides a consistent approach for applications running on-premises or in the cloud.

Taking advantage of claims-based identity requires developers to understand how and why to create claims-based applications. It also requires infrastructure software that applications can rely on. This overview describes the basics of claims-based identity, then looks at how a group of Microsoft technologies help make this world a reality. Those technologies are Active Directory Federation Services (AD FS) 2.0, the Windows Azure AppFabric Access Control service (ACS), and Windows Identity Foundation (WIF).

THE PROBLEM: WORKING WITH IDENTITY IN APPLICATIONS

Sometimes, working with identity is simple. Think of a Windows application that doesn't need to know much about its users, for example, and that will be accessed only by people within a single organization. This application can just rely on Windows Integrated Authentication (WIA), which uses Kerberos under the covers. Kerberos is implemented as part of Active Directory Domain Services (AD DS, originally known as just "Active Directory"), and it provides a way to authenticate users and convey basic information about them. Or suppose you're creating an application that will be accessed solely by Internet users. Again, the common approach to handling identity is straightforward: just require each user to supply a username and password.

Yet the requirements for modern applications are rarely this simple. What if you need more information about each user than is provided by either Kerberos or a simple username and password? Your application will now need to acquire this information from some other source, such as AD DS, or keep track of the information itself. Or suppose the application must be accessed both by employees inside the organization and by customers via the Internet—what now? Should the application support both Kerberos and username/password-based logins? And what about the case where you'd like to let users from a business partner access this application without requiring a separate login? This kind of access can't be accomplished very well with either Kerberos or username/password logins—more is required.

The right solution is to have one approach to identity that works in all of these scenarios. To be effective, this single approach must be based on widely recognized industry standards that interoperate across both platform and organizational boundaries. But standards alone aren't enough. The solution also needs to be widely implemented in products from multiple vendors and be simple for developers to use. This unified, broadly supported approach is exactly what claims-based identity is meant to provide.

THE SOLUTION: CLAIMS-BASED IDENTITY

Claims-based identity is a straightforward idea, founded on a small number of concepts: claims, tokens, identity providers, and a few more. This section describes the basics of this technology, starting with a look at these fundamental notions.

Before launching into this description, however, there's an important point to make. While this paper focuses on the mechanics, using the technology described here can require more, such as business agreements between different organizations. Addressing the technical challenges is essential, but they're not always the whole story.

Claims, Tokens, and STSs

What is an identity? In the real world, the question is hard to answer—the discussion quickly veers into the metaphysical. In the digital world, however, the answer is simple: A digital identity is a set of information about somebody or something. While all kinds of entities can have digital identities, including computers and applications, we're most often concerned with identifying people. Accordingly, this overview will always refer to things with identities as "users".

When a digital identity is transferred across a network, it's just a bunch of bytes. It's common to refer to a set of bytes containing identity information as a *security token* or just a *token*. In a claims-based world, a token contains one or more *claims*, each of which carries some piece of information about the user it identifies. Figure 1 shows how this looks.

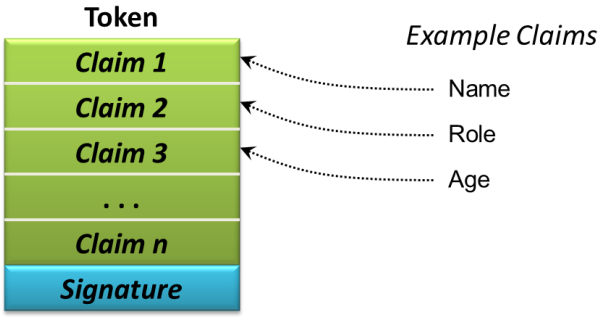

Figure 1: A token contains claims about a user along with a digital signature that can be used to verify its issuer.

Claims can represent pretty much anything about a user. In this example, for instance, the first three claims in the token contain the user's name, an identifier for a role she belongs to, and her age. Other tokens can contain other claims, depending on what's required. A claim might also indicate the user's right to do something, such as access a file, or restrict some right, such as setting an employee's

purchasing limit. And while it's common today to use tokens defined with the XML-based Security Assertion Markup Language (SAML), this isn't required. Web applications might use a simpler approach called Simple Web Token (SWT), for example.

To verify its source and to guard against unauthorized changes, a token's issuer digitally signs each token when it's created. As Figure 1 shows, the resulting digital signature is carried as part of the token.

But who issues tokens? In a claims-based world, tokens are created by software known as a *security token service (STS)*. Figure 2 illustrates the process.

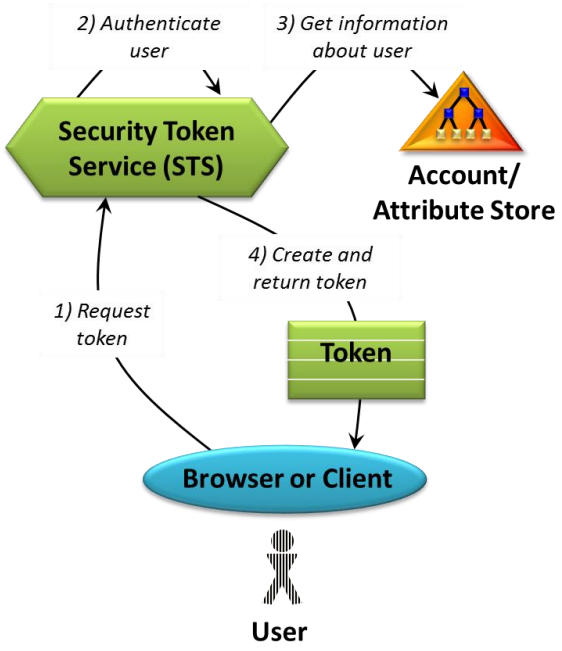

Figure 2: A user acquires a token containing some set of claims from an STS.

In a typical scenario, an application working on behalf of a user, such as a Web browser or some other client, asks an STS for a token containing claims for this user (step 1). Various protocols can be used to make this request, but however it's done, the STS authenticates the user in some way, such as by validating her Kerberos ticket or checking her password (step 2). This lets the STS be certain that the user is who she claims to be.

The request sent to an STS typically contains a URI identifying the application this user wishes to access. The STS then looks up information about both the user and the application in a database (step 3). As the figure shows, this database maintains account information and other attributes about users and applications. Once the STS has found what it needs, it generates the token and returns it to the requester (step 4).

Identity Providers and Identity Libraries

Claims, tokens, and STSs are the foundation of claims-based identity. They're all just means to an end, however. The real goal is to help a user present her digital identity to an application, then let the application use this information to make decisions. Figure 3 shows a simple picture of how this happens.

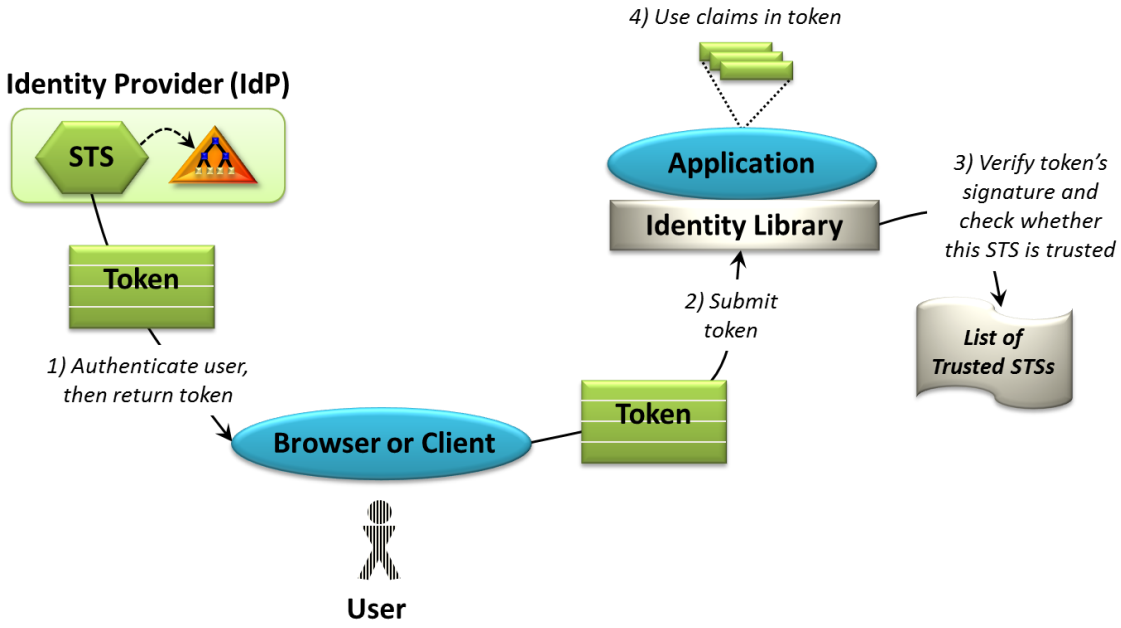

Figure 3: A browser or other client can acquire a token from an STS, then present this token and the claims it contains to an application.

As the figure shows, a Web browser or other client acting on behalf of a user gets a token for a particular application from an STS (step 1). Once it has this token, the browser or client sends it to the application (step 2), which is configured with a list of one or more trusted STSs. To process the token, the application depends on an *identity library*, a reusable set of code for working with tokens and the protocols that convey them. This library verifies the token's signature, which lets the application know which STS issued the token, then checks whether this STS is on the trusted list (step 3). If the application does trust the STS that issued this token, it accepts the token's claims as correct and uses them to decide what the user is allowed to do or in other ways (step 4).

If the token contains the user's role, for example, the application can assume that the user really has the rights and permissions associated with that role. Since the user was required to authenticate herself to get this token, the application doesn't need to authenticate her again. (In fact, because it relies on the claims in the token, an application is sometimes referred to as a *relying party*.)

Notice an important difference between what's happening here and the way that applications frequently handle identity: Rather than requiring the application itself to authenticate the user, claims-based identity relies on the STS to do this. This gets developers out of the business of authenticating users, something that definitely counts as progress. All an application needs to do is determine that the token a user presents was created by an STS this application trusts. How the user proved its identity to this STS—with a password, a digital signature, or something else—isn't the application's problem. This lets the application be deployed unchanged in different contexts, a significant improvement over the usual situation today.

Although it's not shown in the figure, there's an essential first step before any of this can happen: An administrator must configure the STS to issue the right claims for this user and this application. Without this, the STS likely can't create a token containing the claims that the application needs. While doing this might seem like a burden, the reality is that this information must also be configured in the non-claims-

based world. The big difference is that now the claims are all in one place, accessible through the STS, rather than spread across different systems.

Figure 3 also illustrates another important concept, which is that an STS can be owned by some *identity provider (IdP)*. Sometimes called an *issuer*, the identity provider is what stands behind the truth of the claims in the tokens this STS creates. In fact, this is why the contents of a token are called "claims": They're statements that this identity provider claims are true. An application that receives this token can decide whether it trusts this identity provider and the claims it makes about this user.

Identity providers come in many forms. If you use a token issued by an STS on your company's network, for example, the identity provider is your company. If you use a token issued by the STS provided by a service on the Internet, such as Windows Live ID, Facebook, or Google, this service is acting as the identity provider. But whoever the identity provider is, being able to acquire and use a token containing claims is useful.

To see why, think about the pre-claims world we (mostly) live in today. In this environment, an application typically gets only simple identity information from a user, such as her login name. All of the other information it needs about that user must be acquired from somewhere else. The application might need to access a local directory service, for instance, or maintain its own application-specific database. With claims-based identity, however, an application can specify exactly what claims it needs and which identity providers it trusts, then expect each user to present those claims in a token issued by one of those providers. A claims-aware application is still free to create its own user database, of course, but the need to do this shrinks. Instead, each request can potentially contain everything the application needs to know about this user.

Using Multiple Identity Providers

In many cases, a user has only one identity provider—and thus one STS—to choose. When you access an application inside your organization, for example, the application might only accept tokens issued by your company's STS. In some situations, though, an application might accept tokens issued by multiple identity providers—it might trust several different STSs.

For example, think about an application on the public Internet that wishes to let its users log in using a Facebook identity, a Google identity, or a Windows Live ID identity. Since hundreds of millions of people have accounts with these services, why not accept them? Or suppose the application wishes to accept identities directly from multiple instances of Active Directory—what then? Both of these situations require the application to trust multiple STSs at multiple identity providers, then to let the user choose which one he wants to use. Figure 4 shows how this looks.

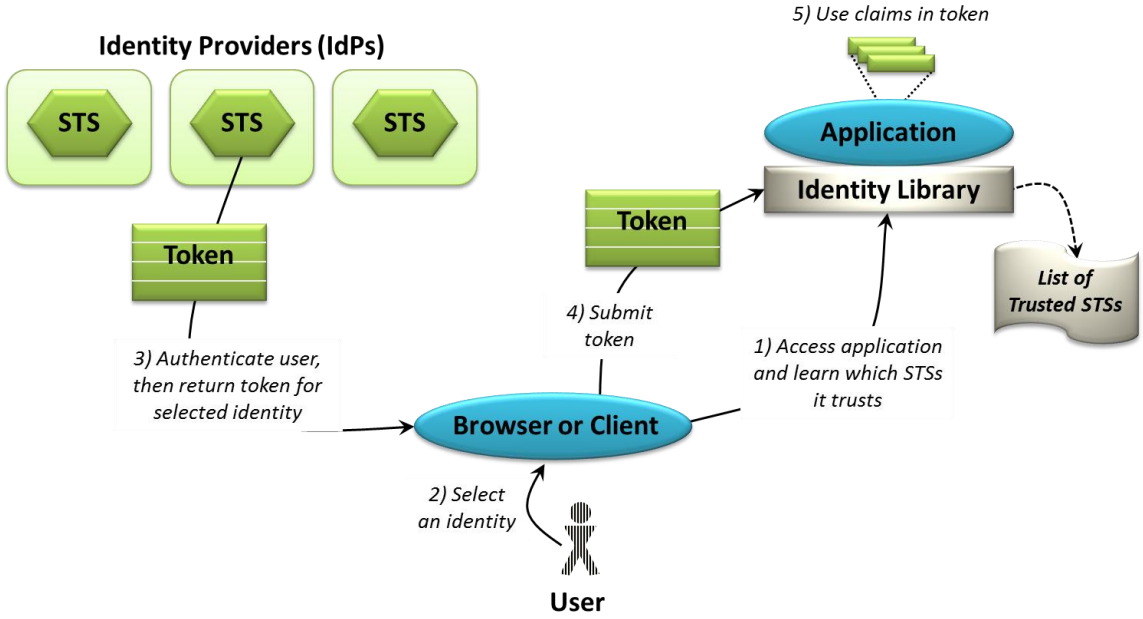

Figure 4: If an application accepts identities from multiple identity providers, the user can select which one to use.

In this situation, the user accesses the application and learns which STSs it trusts (step 1). For example, the application might provide a login screen that lets the user choose to use his Facebook identity, his Google identity, or his Windows Live ID identity. The user then chooses one, and his browser or other client software contacts that identity provider. This provider's STS authenticates the user, perhaps by requiring him to enter a username and password, then returns a token for this identity (step 3). As before, this token is then sent to the application (step 4), which validates it as usual and uses the claims it contains (step 4).

For some applications, accepting tokens from multiple identity providers, especially public providers, makes no sense. Letting employees use their Facebook identity to log into a critical business application inside your company probably isn't a good idea. But there are plenty of situations where this can be useful. Think of an enterprise application that must be accessible to employees, partners, and customers, for example, or a consumer application on the public Internet that wishes to make login as painless as possible. Addressing this requirement is an important part of modern identity technology.

Federation Providers

In a claims-based world, a user always initially gets her identity from an STS owned by some identity provider. But suppose the application she wants to access doesn't trust this STS—what then? One possibility is that there's just no way for her to access this application. There's also another option, however: even though the application she wants to access doesn't trust her STS, it might trust another STS that in turn trusts her STS. This approach, called *identity federation*, is both common and useful. With federation, an identity provider offers an STS as usual, but another STS is also offered by a *federation provider (FP)*. The federation provider STS is then configured by an administrator to trust the identity

provider STS. Figure 5 shows how a user can provide identity information to an application when federation is used.

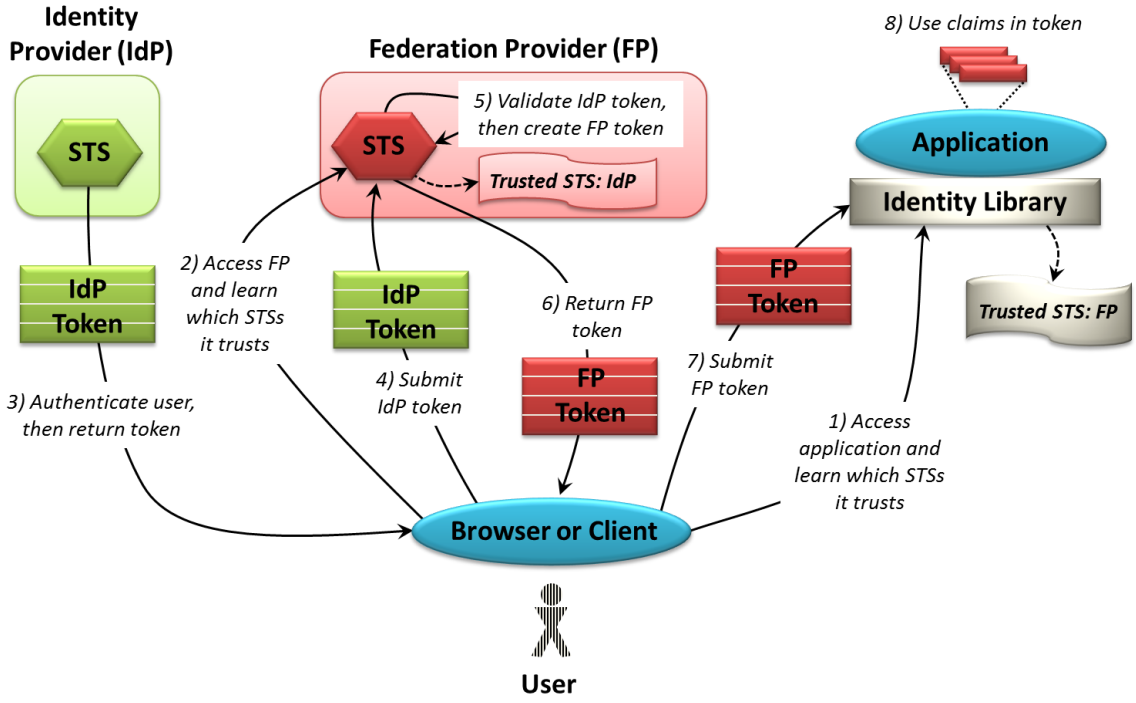

Figure 5: An STS can act as a federation provider, accepting one token and producing another.

As always, the process begins when the user accesses an application from a browser or another client, learning which STSs that application trusts (step 1). Here, the application trusts only the federation provider STS. The user's browser or client software then contacts that federation provider, learning which STSs it trusts (step 2). In this example, the federation provider STS is configured to trust the identity provider's STS, and so the user's browser or client software contacts this STS. As usual, the user is authenticated in some way, then gets back an IdP token created by this STS (step 3).

This token can't be used to access the application, however, since that application doesn't trust the STS that issued it. Fortunately, this token can be used to acquire a token that the application will accept. To do this, the browser or client software sends the IdP token to the federation provider (step 4). The federation provider validates this token, ensuring that it came from an STS it trusts. Once it determines this, it creates a new token for this user (step 5), then returns this FP token (step 6). The user's software submits this token to the application (step 7), which verifies that the FP token was issued by an STS that it trusts. The application then uses the claims in the token as usual (step 8).

From the user's point of view, all of these exchanges are invisible. She accesses the application without explicitly logging into it, that is, she gets what's known as *single sign-on*. The mechanics are a little more complex, but the core idea underlying identity federation is straightforward. It is that not only applications can trust STSs; one STS can trust another STS as well.

Step 5 in the figure is worth examining in more detail. As just described, the federation provider receives a token issued by another STS, then generates a new token for the user. But exactly what claims does this

new token contain? The answer depends on what the federation provider's STS is configured to do. In the simplest case, it might copy every claim from the IdP token directly into the FP token unchanged. In a more realistic scenario, the federation provider STS performs *claims transformation*, emitting a token that doesn't contain the exact set of claims that it received from the identity provider.

For example, suppose the IdP token contains a claim indicating that this user is a member of the role "Administrator", expressing that claim as a character string containing this English word. It's possible that the application understands the administrator role, but expects the claim to be expressed as a numeric code or in Chinese or in some other way. The federation provider can perform this translation, inserting a claim in the correct format in the FP token it generates.

Claims transformation can do other things as well. When it creates the FP token, for example, the federation provider might omit claims from the IdP token that aren't meaningful to this application. Or it might add claims to the FP token that aren't present in the IdP token, such as an indication of the IdP that issued the original token. Claims transformation is a powerful idea, and it can be used in a variety of ways.

IMPLEMENTING CLAIMS-BASED IDENTITY: MICROSOFT TECHNOLOGIES

Implementing claims-based identity requires several things. Identity provider STSs must be available to issue tokens to users. Because identity federation is common, federation provider STSs are also essential. And finally, developers will need to build *claims-aware* applications that know how to receive tokens and use the claims they contain. Rather than having every developer write this code from scratch, it makes sense to provide a standard identity library that any application can use.

The rise of cloud computing adds another requirement. All three of these things—an identity provider STS, a federation provider STS, and an identity library—should be available both on premises (e.g., running in an organization's data center) and in the cloud. Without this, some important scenarios are hard to address.

The Microsoft platform for claims-based identity targets all of these options. Figure 6 summarizes the technologies it includes today.

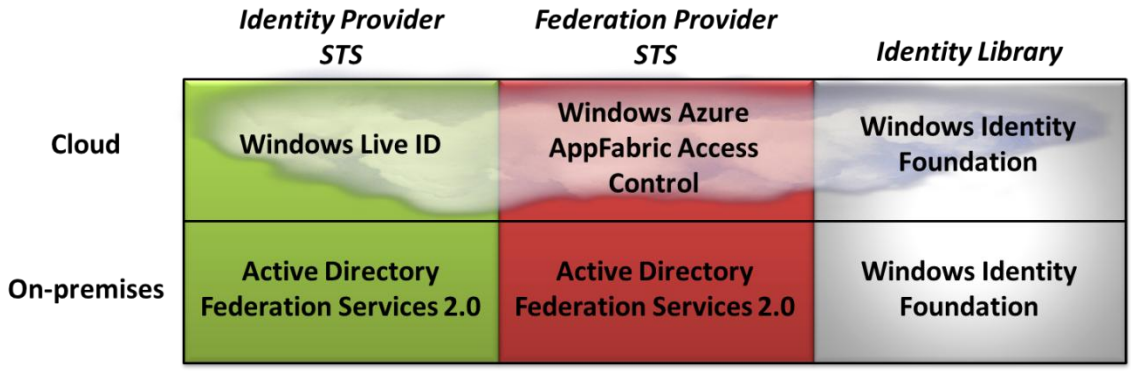

Figure 6: Microsoft provides cloud and on-premises technologies for an identity provider STS, a federation provider STS, and an identity library.

In the cloud, Microsoft provides Windows Live ID as an identity provider STS, while Windows Azure AppFabric Access Control provides a federation provider STS. For an identity library, applications can use Windows Identity Foundation (WIF, commonly pronounced "Dub-I-F").

For on-premises use, Microsoft makes available Active Directory Federation Services (AD FS) 2.0. As Figure 6 shows, this technology can be used as both an identity provider STS and a federation provider STS. And for an identity library, applications once again can use Windows Identity Foundation.

It's important to note that while this overview focuses on Microsoft technologies, claims-based identity is a multi-vendor effort. Given this, there are alternative technologies from other vendors for all of the boxes in this figure. And because interactions among the parties are based on industry standards, the offerings from Microsoft and other vendors can be combined in various ways. For example, using the Microsoft identity platform doesn't require relying on Windows Live ID for a cloud identity provider STS—other providers, such as those from Google and Facebook, can also be used. Similarly, AD FS 2.0 isn't the only option for on-premises STSs; WIF can work with tokens created by products from IBM and other vendors.

The focus here is on the Microsoft technologies, however, and the best way to understand how they fit together is to walk through scenarios showing how they're used. Before doing this, we first need to look at the basics of each one.

WINDOWS LIVE ID

Windows Live ID implements an identity provider STS in the cloud. Today, the most popular applications that accept tokens issued by this STS are Microsoft offerings such as Hotmail. Any application can choose to accept these tokens, however—it's not usable only by Microsoft itself.

The token provided by Windows Live ID contains a very simple set of claims, primarily just a globally unique identifier. The structure of this identifier is opaque, which means that an application can't derive any meaning from the identifier's content or structure. An application can use this value to recognize individual users, however. For example, a Web site that accepts Windows Live ID logins might ask each user for information such as his name and shipping address, then store this data along with the user's Windows Live ID identifier. The next time the user logs in, the application can use this identifier to look up his information.

ACTIVE DIRECTORY FEDERATION SERVICES 2.0

Having an identity provider STS in the cloud is useful. But for enterprises, having one on-premises is much more important. Business applications inside your organization probably don't let employees log in with a Windows Live ID, a Facebook identity, or any other token issued by an identity provider STS in the cloud. Instead, they require a token issued by an STS that they control.

Active Directory Federation Services 2.0 can fill this role. As its name suggests, AD FS 2.0 is the follow-on to the original Active Directory Federation Services technology. Don't be misled by the word "federation" in the technology's name, however. In fact, AD FS 2.0 can act as either an identity provider STS or a federation provider STS. The same AD FS 2.0 instance can even act in both roles simultaneously. And since it's part of Active Directory, AD FS 2.0 is available to current users at no extra cost. This makes claims-

based identity immediately accessible to the large number of organizations that use Active Directory today.

AD FS 2.0 contains several advances over its predecessor. It supports both browsers and other clients, for example, such as those built using Windows Communication Foundation (WCF)[1]. Also unlike the first AD FS release, AD FS 2.0 supports the SAML 2.0 protocol as well as WS-Federation and WS-Trust, letting it work in a broader range of environments.

The AD FS 2.0 STS can be used entirely inside an organization, exposed on the Internet, or both. The claims it supplies can come from Active Directory Domain Services, of course, but this isn't the only choice. AD FS 2.0 also supports using SQL Server as an *attribute store*, that is, a source for claims, and developers are free to create custom attribute stores as well. To help manage an organization's attribute stores, including Active Directory and others, Microsoft provides Forefront Identity Manager (FIM). This technology offers a way to synchronize information across different attribute stores, along with an identity management portal with pre-defined workflows for password resets, group management, and more.

Yet it's worth reiterating that claims-based identity doesn't require using AD FS 2.0. Any STS from any vendor (or even a custom-built STS) that supports standard protocols and token formats can be used. Still, one of Microsoft's primary goals in providing AD FS 2.0 is to make widely available a fully-featured STS built on Active Directory. Ubiquitous STSs are fundamental to making the benefits of claims-based identity real.

WINDOWS AZURE APPFABRIC ACCESS CONTROL

Identity federation is useful in many situations. For applications running on premises, such as a business application within an enterprise, using a federation provider STS that's also on premises is often the right choice. But for an application running in the cloud, using a federation provider STS that runs in the cloud is likely to be better. The Windows Azure AppFabric Access Control service fills that role.

ACS is most commonly applied today in two scenarios:

- Letting an application running in the cloud accept tokens issued by multiple on-premises identity provider STSs, such as AD FS 2.0. This can give on-premises users in various organizations single sign-on to the cloud application. This is especially useful for independent software vendors (ISVs) who wish to allow easy access to a Software as a Service (SaaS) application by customers in many different enterprises.

- Letting an application running in the cloud accept identities issued by multiple cloud identity provider STSs. ACS has built-in support for handling the protocols and token formats used by Windows Live ID, Google, Facebook, Yahoo, and OpenID. An application that trusts the ACS federation provider STS can choose to accept identities from any of these identity providers while still being shielded from the idiosyncratic details of each one.

[1] In the jargon of identity, AD FS 2.0 supports both *active* and *passive* clients, while the first release of AD FS supported only *passive* clients.

ACS also provides another important function: built-in support for claims transformation. As described earlier, a federation provider STS commonly emits a token whose claims differ from the IdP token it received. To help do this more intelligently, ACS includes a rules engine for defining these transformations.

ACS lets clients request tokens using various protocols, including WS-Federation, WS-Trust, OpenID 2.0, and OAuth 2.0. It can accept and issue tokens in various formats as well, including SAML 1.1, SAML 2.0, and Simple Web Token (SWT). The technology also allows delegated authorization using OAuth 2.0, which provides a controlled way for an application to act on behalf of a user.

WINDOWS IDENTITY FOUNDATION

Whether it's offered by an identity provider or a federation provider, an STS creates tokens containing claims. Yet those tokens are useless unless applications are able to accept and use them. The goal of WIF is to make this easier by helping developers create claims-aware Windows applications.

WIF is a set of .NET Framework classes that implement essential identity functions, such as receiving a token, verifying its signature, and accessing the claims it contains. It supports tokens created using either the SAML 1.1 or SAML 2.0 formats, so it can accept tokens issued by AD FS 2.0, ACS, or STSs from other vendors. WIF supports various standard protocols as well, including WS-Federation and WS-Trust, and it's extensible, allowing other technologies to be added. For example, Microsoft provides a sample WIF extension that implements the OAuth 2.0 protocol with SWT tokens.

Each claim is extracted into an instance of a WIF-defined Claim class, providing a consistent way for developers to work with a token's information. This class's properties include things such as:

- ClaimType, indicating what kind of claim this is. Does the claim contain a user's name, for example, or a role, or something else? Claim types are identified by strings, which are just URIs.
- Value, containing the actual content of the claim, such as the user's name.
- Issuer, which specifies the identity provider STS this claim came from. In other words, this is the entity asserting that this claim is true.

Microsoft itself is using WIF to add support for claims-based identity to its own products, including SharePoint 2010 and others.

Along with helping developers create claims-aware applications, WIF also provides support for creating a custom STS. Even though a primary goal of AD FS 2.0 is to reduce the need to hand roll your own STS, there are situations where building an STS can make sense. One important example of using WIF in this way already exists: AD FS 2.0 itself is built on WIF.

USING CLAIMS-BASED IDENTITY: SCENARIOS

Getting your mind around claims-based identity requires understanding the basics of this technology. Still, the best way to get a feel for the approach is to walk through examples of how it's applied. Accordingly, this section looks at a number of different ways that claims can be used both on premises and in the cloud, each illustrated using the Microsoft technologies just described.

ON-PREMISES SCENARIOS

Claims-based identity was first used to address problems within and between enterprises. This is still where the technology is most widely used today, and so it makes sense to look at on-premises scenarios first. This section walks through three examples:

- Accessing an enterprise application, where the application, the identity provider STS, and all of the users are within the same organization.
- Accessing an enterprise application via the Internet, which extends the first scenario to include remote Internet access by users outside the organization.
- Using identity federation between enterprises, where a user in one organization accesses an application in another organization.

Accessing an Enterprise Application

Most enterprises act as an identity provider today, and nearly every enterprise application must deal with identity. AD FS 2.0 and WIF can provide the foundation for using claims-based identity with on-premises applications, those running inside an organization. Figure 7 shows how this looks.

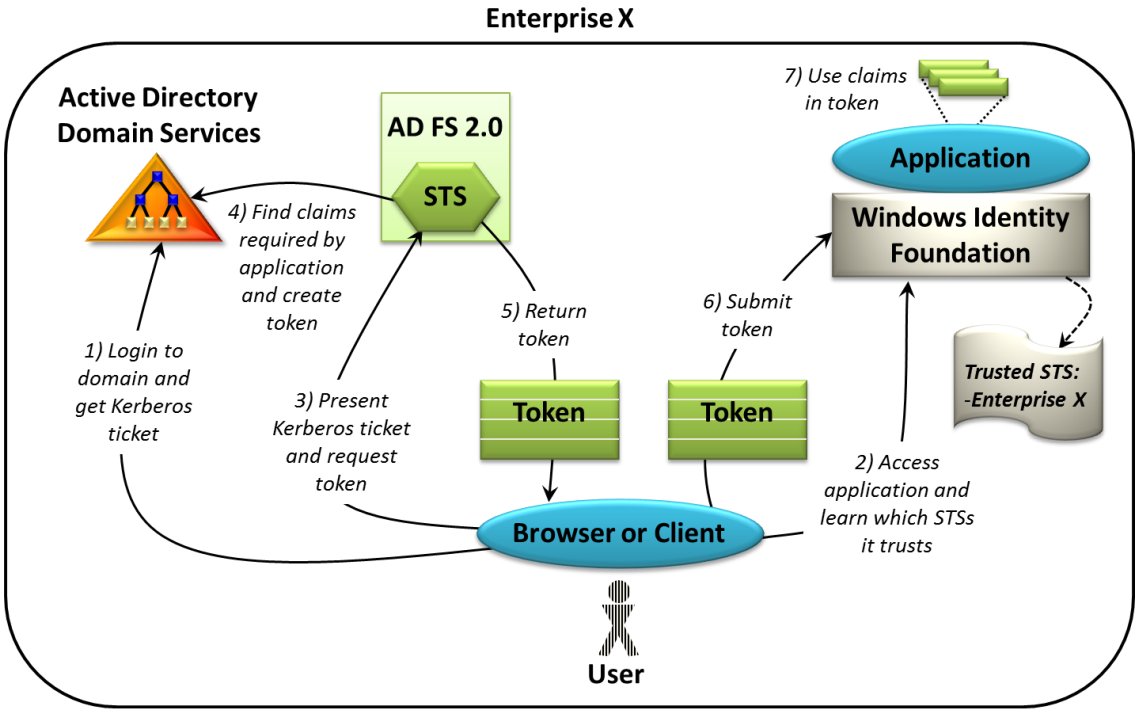

Figure 7: An enterprise can use AD FS 2.0 and WIF to support claims-based identity for its internal applications.

In this example, a user logs in using AD DS, getting an initial Kerberos ticket (step 1). The user then accesses a claims-aware application built using WIF, learning which STSs it trusts (step 2). This application only trusts the STS within its own enterprise, and so the user's browser or client requests a token from

that STS, supplying a Kerberos ticket to authenticate the user (step 3). AD FS 2.0, acting as an identity provider STS, verifies the ticket, then looks in AD DS for the information it needs to create the requested token (step 4). Exactly what claims appear in this token depends on both the user requesting it and the application that user is accessing—each application can indicate which claims it needs. Once the token has been created, the AD FS 2.0 STS sends it back to the user's browser or client (step 5), which sends it on to the application (step 6). The application then uses WIF to verify the token's signature and make its claims available for use (step 7).

One big plus of a claims-based approach is worth re-emphasizing here: Rather than having to go look for the information it needs about a user, the application can instead get everything handed to it in the token. If the application needs, say, the user's job title, it can specify this in its list of required claims. When the STS creates a token for the application, it finds the user's job title in AD DS and inserts it as a claim that the application can use. Without this, the application developer must write his own code to dig this information out of AD DS. Claims-based identity makes the developer's life significantly easier.

Accessing an Enterprise Application via the Internet

Suppose this organization wishes to make this on-premises application accessible to remote employees via the Internet. Rather than modifying the application to accept username/password logins, a traditional solution, the same claims-based approach can be used—the application remains unchanged. Figure 8 shows how this looks.

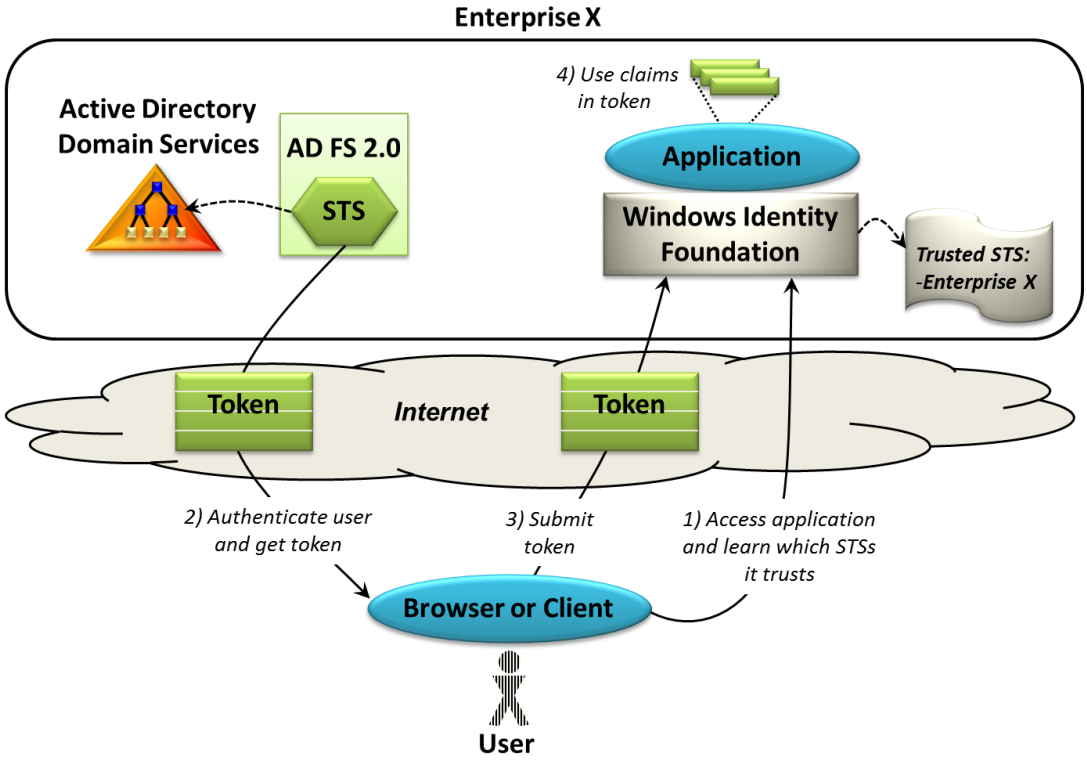

Figure 8: An enterprise can use the AD FS 2.0 STS to create tokens for users on the Internet.

Here, the user is on another computer outside the enterprise. As before, she accesses the application and learns which STSs it trusts (step 1). Once again, the application trusts only the STS in its own enterprise, and so the user's browser or client requests a token from that STS. Implemented using AD FS 2.0, that STS authenticates the user and sends back the token[2] (step 2). The user's browser or client then submits this token to the application (step 3). The application relies on WIF to verify the token, then uses the claims it contains (step 4).

Rather than requiring the application to implement a different way of handling identity for Internet access, a claims-based approach lets it handle this situation just like the inside-the-enterprise case. Still, some extra complexity creeps in. When the user requests a token in step 2, for example, how does she authenticate herself to the STS? Kerberos tickets work just fine for users inside the enterprise, as shown earlier in Figure 7, but they don't work as well for Internet users. Instead, the user might provide a username and password in step 2 to authenticate her request, an option that Microsoft supports in AD FS 2.0. Since the users in this scenario are employees, they already have accounts in AD DS, and so they can log in with no trouble.

But what if the users aren't employees? Suppose the application needs to be exposed via the Internet to customers as well. Can this approach still work? The answer, unsurprisingly, is yes. Although it's not an especially common option, information about external users can also be stored in AD DS, letting it be accessed by AD FS 2.0.

But wait a minute: If Internet users still need usernames and passwords, how is the claims-based approach making things better? One important improvement is that it frees each application from the need to store sensitive password information, moving that responsibility instead to the much smaller number of STSs. Also, users no longer have a password for each application. Instead, they'll (at most) have one for each STS they use. While claims-based identity doesn't eliminate the need for usernames and passwords, it does help make the world simpler and more secure.

Providing Single Sign-On to an Enterprise Application in Another Organization

Another common identity challenge is letting users in one organization access an application running in some other organization. For example, suppose your company wishes to make an internal SharePoint site accessible to employees at a partner firm. One way to do this is to give each of these external users their own account in your company. While this approach works, it's unappealing. Those users won't like having a separate login, and your firm's administrators won't like having to administer accounts for people outside your company. Doing this also creates security risks—how can your administrators know when an external user has left his company and so should no longer have an account?

A better solution is to let the external users access your application using their own identities. This approach requires no separate logins and no new accounts. What it does require, however, is creating a relationship between your firm and its partner. Doing this will likely require some kind of business

[2] To avoid opening the enterprise's firewall, it's possible to run an instance of AD FS 2.0 in a proxy role inside this organization's DMZ. This proxy then forwards requests to the full AD FS 2.0 service running inside the firewall.

agreement between the two organizations, a topic that's beyond the scope of this discussion. It also, of course, requires putting in place the right technology.

One approach is to configure an application running in one organization to trust the identity provider STS in another. Figure 9 shows how this looks.

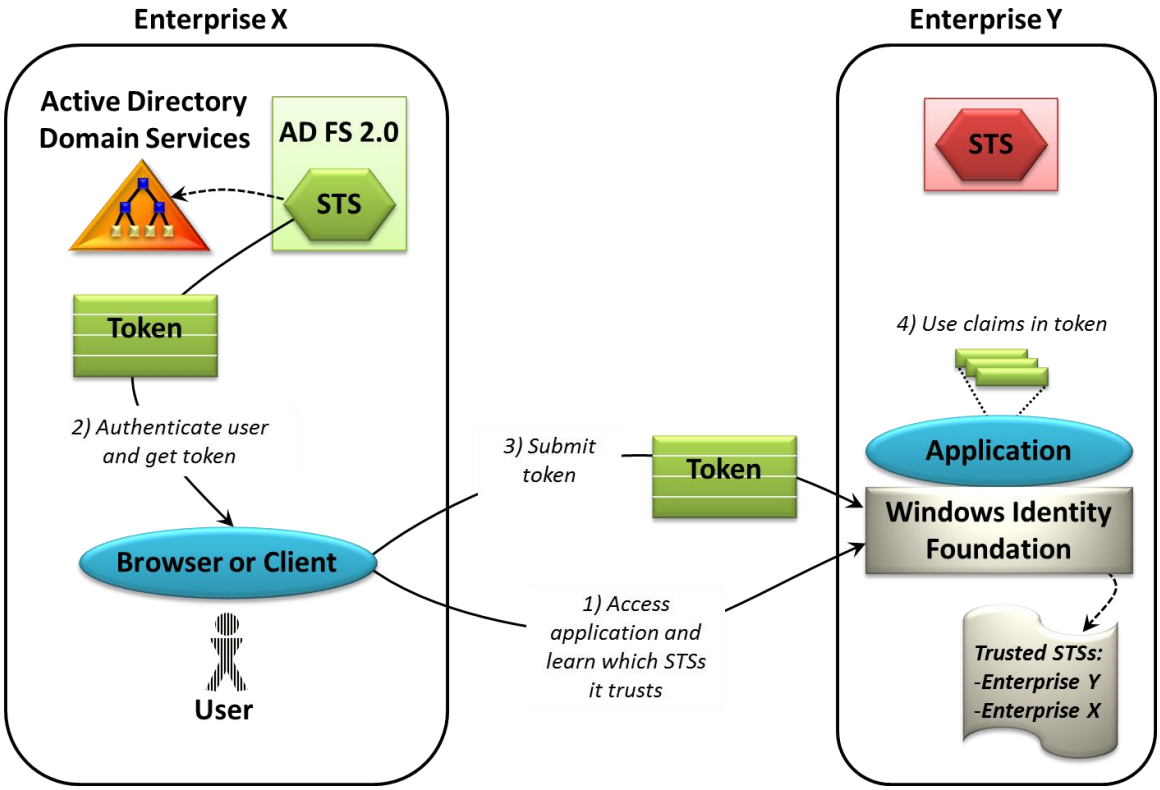

Figure 9: If the application trusts the STS in the client's enterprise, it can accept and use a token issued by that STS.

In this scenario, a user in enterprise X accesses an application in enterprise Y and learns which STSs it trusts (step 1). Here, that application is configured to trust both its own STS (the one in enterprise Y) and the identity provider STS in enterprise X. All that's required is for the user in enterprise X to get a token from its own STS (step 2), then submit this token to the application (step 3). The application uses WIF to verify the token and extract its claims, then uses those claims in any way it likes (step 4).

This solution is simple to understand, but it's not without problems. Suppose this application has users in several different enterprises, for instance. With the approach shown here, the application would need to be configured to trust the identity provider STS in each one, an unappealing prospect. A better solution is to use federated identity, introducing a federation provider STS into the mix. Doing this means that the application only needs to trust its own STS, making life significantly simpler for the people who build and administer this application. Figure 10 shows this more likely situation.

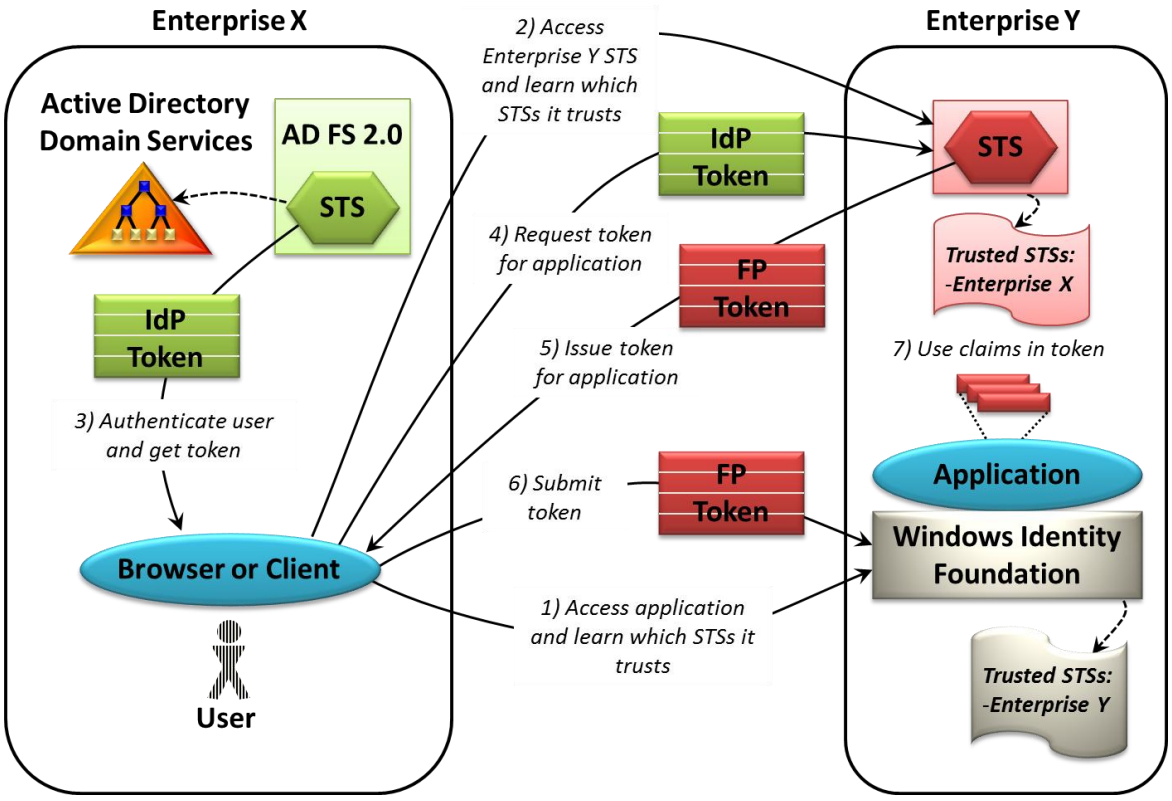

Figure 10: If the application trusts only the STS in its own enterprise, the client must get a token from that STS to access the application.

This scenario starts in the same way: The user in enterprise X accesses an application in enterprise Y and learns which STSs it trusts (step 1). This time, however, that application is configured to trust only its own STS, the one in enterprise Y. Once it determines this, the browser or client on the user's system contacts the STS in enterprise Y to learn which STSs it trusts (step 2). Here, that STS is acting in the role of federation provider, and so it's configured to trust the STS in enterprise X. Accordingly, the user's client or browser requests a token from the enterprise X STS (step 3), which is acting in the role of identity provider.

The application won't accept this IdP token, however—it was issued by an STS that the application doesn't trust. Instead, the user's browser or client submits the IdP token to the federation provider STS in enterprise Y (step 4). Because this STS is configured to trust the STS in enterprise X, the enterprise Y STS can verify the token it receives from enterprise X, then issue an FP token that allows this user to access the application (step 5). The user presents this token to the application (step 6), and the application uses WIF to verify the token and extract its claims. The application can now use these claims as usual (step 7).

Notice that the federation provider STS in enterprise Y is acting as a claims transformer, accepting a token issued by STS X, then creating its own token. The contents of the token STS Y creates might well be different from those in the token it receives from STS X—it's free to add, remove, or modify the claims. In fact, AD FS 2.0 provides a fairly sophisticated way for an administrator to define rules for these transformations. It's even possible to prohibit issuing tokens entirely when necessary.

Notice also how convenient it is for the application to get the information it needs about a user directly in the token. When both user and application are in the same organization, the application might be able to access, say, AD DS directly to get information such as the user's job title. When they're in different organizations, as in the scenario shown here, the application almost certainly won't be allowed to do this. Getting everything it needs handed to it in the user's token is very nice indeed.

One final issue worth discussing is this: In step 2 above, the federation provider STS directs the user's browser or client to an identity provider STS. But if the federation provider STS trusts multiple identity providers, as it probably does, how does it know which one it should send this user to? This problem is known as *home realm discovery*, and there are several possible solutions. One option, which AD FS 2.0 supports, is to present the user with a list of possibilities, letting her choose the one she recognizes as her own organization. Another option is to have the user provide her email address, then let the federation provider STS infer her identity provider from the address's domain name. However it's done, using identity federation typically requires solving this problem in some way.

CLOUD SCENARIOS

On-premises scenarios are important, but more and more applications run in the cloud. Using claims-based identity can simplify life for developers and users here, too. Once again, it's useful to look at several possibilities:

- Accessing a cloud-based application, where the user and the identity provider STS are inside an enterprise, but the application is running in the cloud.
- Using cloud-based identity federation, which extends the first scenario by adding a federation provider STS that runs in the cloud.
- Using multiple identity providers with cloud-based federation, where an application running in the cloud uses a cloud-based federation provider STS to accept identities issued by multiple identity provider STSs.

Providing Single Sign-On to an Enterprise Application in the Cloud

Suppose a developer wants to build an application on a cloud platform, such as Windows Azure or Amazon Web Services, then make it available to users in her enterprise. Or suppose an enterprise decides to move an existing on-premises application to a cloud platform. How can users in the enterprise access this application without having a separate login with its own username and password?

One approach is to configure the application to trust the enterprise's identity provider STS. Figure 11 shows how this might look.

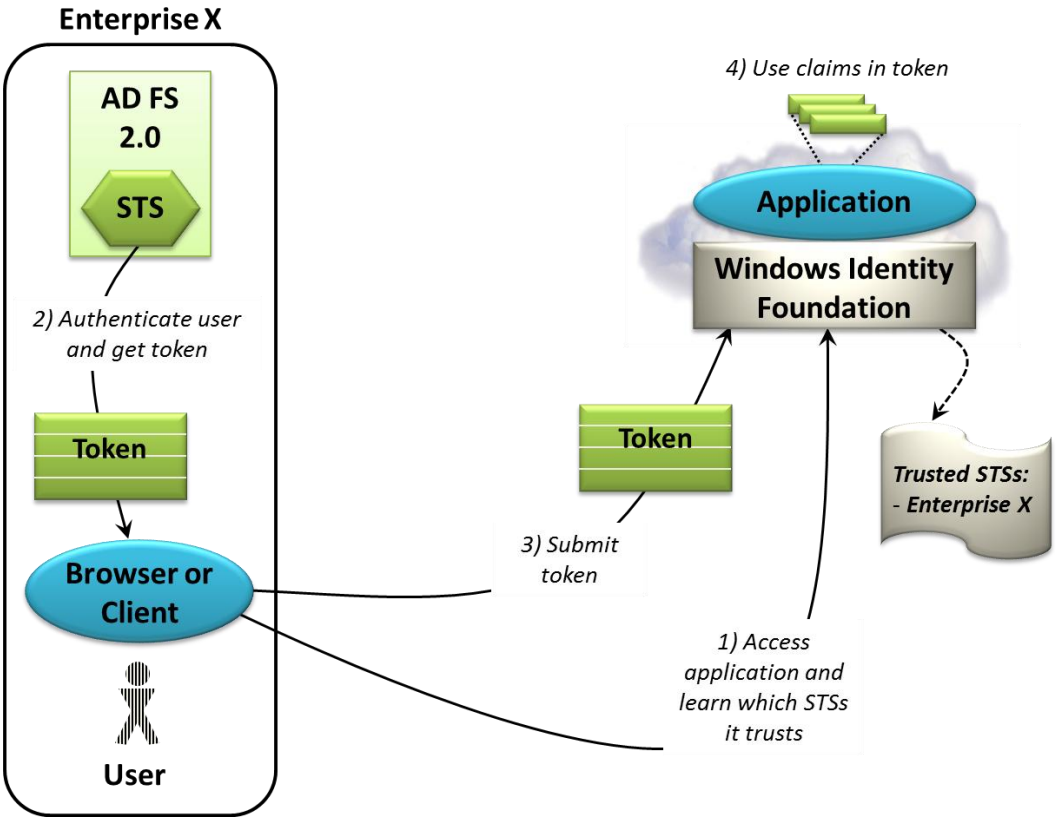

Figure 11: An enterprise can use AD FS 2.0 and WIF to provide single sign-on to an application running in the cloud.

This scenario isn't really new; in fact, it's very similar to the first scenario shown back in Figure 7. As in that example, the user accesses the application and learns which STS it trusts (step 1), then gets a token from that STS (step 2). Although it's not shown in the figure, the user probably authenticates himself to the STS using a Kerberos ticket, just as in Figure 7. Once the user has a token from this identity provider STS, his browser or client submits it to the application (step 3). The application uses WIF to validate and process the token, then uses the claims as usual (step 4).

From the user's point of view, this looks just like accessing an on-premises application. And since the application is configured to trust the identity provider STS in enterprise X, there's no need for identity federation. This simple scenario can work well for cloud applications used by a single enterprise.

Providing Single Sign-On to a SaaS Application

In the identity federation scenario shown earlier in Figure 10, the federation provider STS ran inside the same organization as the application being accessed. This isn't required, however. As long as the application trusts the federation provider STS, that STS can run anywhere—even in the cloud.

Windows Azure AppFabric Access Control is exactly that: a federation provider STS that runs in the cloud. While it can be used by applications running either on-premises or in the cloud, the most interesting ACS

scenarios today are for cloud-based applications. For example, suppose a SaaS application wishes to provide single sign-on to users in many organizations, each with its own identity provider STS. While it would be possible to configure the application to trust the STS in each organization, doing this is problematic. A better solution is to use ACS. Figure 12 shows how the interaction works with one of those organizations.

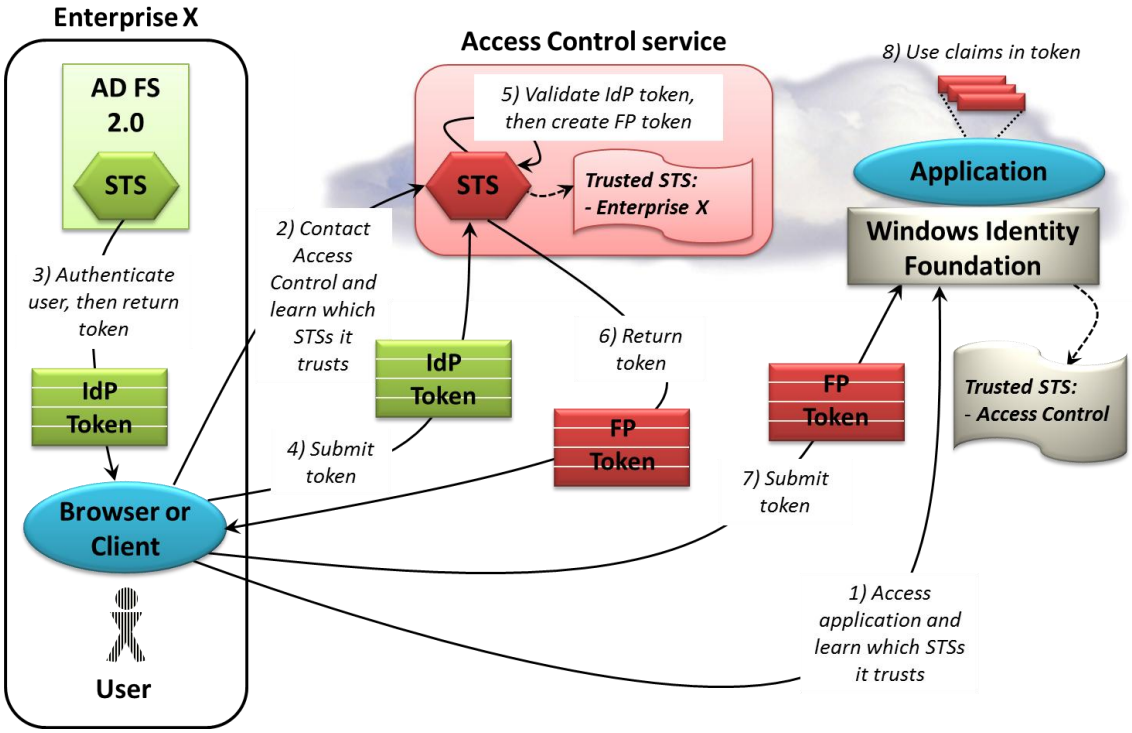

Figure 12: A cloud application can use the Access Control service as a federation provider STS, making it easier to offer single sign-on to users in multiple enterprises.

Although the figure looks a bit different, this is really just another example of federation like those shown earlier. Here's what happens, step by step:

- As before, the user accesses the application and learns that it trusts only the federation provider STS provided by ACS (step 1).
- The user's browser or client contacts that STS to learn which STSs it trusts, a list that includes the STS in enterprise X (step 2).
- The browser or client gets a token from this identity provider STS, implemented here with AD FS 2.0 (step 3).
- The browser or client sends the token to the ACS STS (step 4).
- The ACS STS validates this IdP token, then creates an FP token using the ACS rules engine (step 5). This new token may well contain some claims that are different from those in the original AD FS token; as usual, the federation provider STS can perform claims transformation.
- The token created by ACS is then returned to the browser or client (step 6).

- The browser or client sends the token to the application (step 7).
- The application uses WIF to validate the FP token, then uses the claims it contains (step 8).

Notice how using ACS as a federation provider STS simplifies life for the cloud application. Rather than trusting the identity provider STS in each of its customers, it can instead trust only the ACS STS. Because ACS is explicitly designed to handle this kind of federation scenario, configuring it to trust various identity provider STSs is straightforward. Using this cloud-based federation provider STS also lets the application rely on the rules engine ACS provides, giving it one place to define any necessary claims transformations. Rather than dealing with diverse tokens issued by diverse identity provider STSs, the application always gets a token in the same format containing an expected set of claims.

Allowing Logins with Facebook, Google, and Other Cloud Identity Providers

In the example just described, the identity provider STSs were assumed to be inside enterprises, and so they were provided by technologies such as AD FS 2.0. For a SaaS vendor providing a multi-tenant business application, this is a common situation. But think about a cloud application that's used mostly by consumers rather than business users. Consumers don't get their identities from AD FS 2.0—pretty much nobody runs Active Directory in their home. Instead, people rely on cloud identity providers such as Google, Facebook, Windows Live ID, and Yahoo.

Accepting these identities can make life simpler for a cloud application's users. For the creators of the application, however, doing this introduces some complexities. Because these identity providers use different protocols and token formats—they don't all conform to a single standard—implementing support for them is non-trivial. And without a federation provider STS, the cloud application must be configured to trust all of them.

The Access Control service addresses both problems. It's designed to maintain trust relationships with multiple identity providers, as already described, and it also has built-in support for the protocols and token formats used by today's most popular cloud identity providers. Because of this, using ACS can make it simpler for cloud applications to accept diverse identities. Figure 13 shows how this looks.

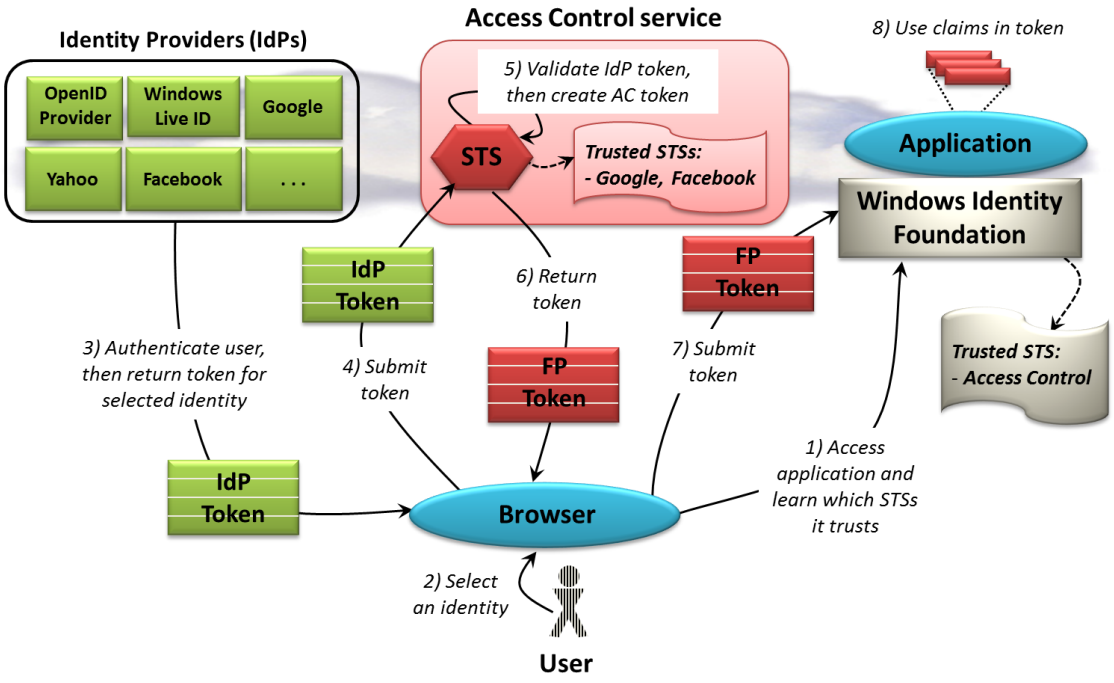

Figure 13: The Access Control service makes it easier for an application to accept identities from multiple cloud identity providers.

Once again, this is really just a federation scenario like those already described, and so it starts with the user accessing the application and learning which STSs it trusts (step 1). Here, the only directly trusted STS is provided by ACS, as the figure shows. But suppose this application will actually accept identities from, say, Google and Facebook—how does it convey this to the user? One popular option is to display icons on the login page presented to the user for each identity provider that this application trusts, then let the user select the one it wishes to use, which is what's done here (step 2). Once this choice has been made, the user's browser contacts the chosen identity provider STS. This STS authenticates the user, probably by making him enter a username and password, then returns a token for this identity (step 3).

The application doesn't directly trust the identity provider STS, however, so the user's browser must send this newly received IdP token to ACS (step 4). An administrator has configured the ACS STS to indicate that this application trusts both Google and Facebook, and so once it has verified that the incoming token is from one of these identity providers, the ACS STS creates a new token (step 5). This FP token is returned to the user's browser (step 6), then forwarded to the application (step 7). As usual, the application relies on WIF to validate the FP token, then uses the claims it contains (step 8).

As the figure shows, ACS has built-in support for Windows Live ID, Google, Facebook, Yahoo, and any identity provider that uses OpenID 2.0. As the figure also shows, however, this scenario only supports Web browsers—these cloud identity providers don't currently allow any other option. Another challenge with cloud identity providers is that they typically provide only very simple claims. Recall, for instance, that Windows Live ID sends just an opaque unique identifier. An application is free to request other information from the user, of course, then store that information itself, but ACS doesn't provide a service to do this.

CONCLUSION

Without claims-based identity, application developers are faced with a diverse set of scenarios, each with its own identity solution. Using claims-based identity collapses all of these down to just one problem: How does an application get information about a user from a trusted source? As this overview has shown, claims-based identity provides a consistent answer across a wide range of scenarios, both on premises and in the cloud. And even though the technologies described in this overview are from Microsoft, there are no proprietary links between them; all communication is based on industry standards.

Changing how people and applications work with identity is not a small thing. The whole world needn't change at once, however. Claims-based applications can be built and deployed alongside existing applications, which makes migration to this much-improved approach simpler. And with the advent of AD FS 2.0, Windows Azure AppFabric Access Control, and Windows Identity Foundation, all of the pieces required to use claims-based identity on Windows are here.

Claims-based identity is far from a Microsoft-only initiative—many vendors are involved. Still, making these components widely available for Windows is bound to make the approach more popular. For anyone who cares about improving the way we use identity in the digital world, this is certainly a step forward.

ABOUT THE AUTHOR

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technology.