

# Microsoft Code Name "Geneva" Framework Whitepaper for Developers

Keith Brown

Pluralsight, LLC

Sesha Mani

Microsoft Corporation

## Legal Information

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, CardSpace, Windows, Windows Server, are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

## About this Paper

The goal of this whitepaper is to help developers get started building claims-aware applications using Microsoft© Code Name "Geneva" Framework. In this paper I introduce concepts and terminology to help developers understand the benefits and concepts behind the claims-based model of identity. My target audience does not consist of security experts, rather those familiar with ASP.NET or Windows Communication Foundation (WCF) programming, and who are building web applications or services that care about authentication and authorization. As such, my focus will be on building relying parties using Geneva Framework. I will talk about issuance and security token services (STS) and will provide an example of an STS built using Geneva Framework. However, that is not the focus of this paper.

## Identity Challenges

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 these features being ignored until late in the software development lifecycle.

It's not surprising nowadays to see a single company with tens or hundreds of web 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.

One very useful step toward solving the problem has been to centralize user accounts into an *enterprise directory*. Commonly it's the IT pro that knows the most effective and efficient way to query the directory, but today the task is typically left up to the developer. And in the face of mergers, acquisitions, and partnerships, the developer might be faced with accessing more than one directory, using more than one API.

In the Microsoft .NET Framework, there are lots of different ways of building identity support into an application, and each communication framework treats identity differently, with different object models, different storage models, and so on. Even in ASP.NET, developers can get confused about where they should look for identity: should they look at the `HttpContext.User` property? What about `Thread.CurrentPrincipal`?

The rampant use of passwords has led to a cottage industry for phishers<sup>1</sup>. And with so many applications doing their own thing, it's difficult for a company to upgrade to stronger authentication techniques.

---

<sup>1</sup> [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

## A Better Solution

One step toward solving these problems is to stop building custom identity plumbing and user account databases into every new application that comes along. But even developers who rely on a central enterprise directory still feel the pain of mergers, acquisitions, and external partnerships, and may even be blamed for poor performance that is actually due to another application bogging down the directory with inefficient queries. The *claims-based* solution described in this paper avoids asking developers to connect to any particular enterprise directory in order to look up identity details for users. Instead, the user's request arrives with all of the identity details the application needs to do its job. By the time the user arrives with these *claims*, the user has already been authenticated, and the application can go about its business without worrying about managing or finding user accounts.

Factoring authentication out of applications leads to many benefits for developers, IT pros, and users. Simply put, there are less user accounts for everyone to manage, and the resulting centralization of authentication makes it easier to upgrade to stronger authentication methods as they evolve, and even *federate* identity with other platforms and organizations.

This paper will help you, as a developer, to understand the claims-based identity model and take advantage of it using Geneva Framework, the new framework from Microsoft that is focused on identity.

## What is Geneva Framework?

Geneva Framework is a set of .NET Framework classes; it is a framework for implementing claims-based identity in your applications. By using it, you'll more easily reap the benefits of claims-based systems described in this paper. Geneva Framework can be used in any web application or web service that uses the .NET Framework version 3.5.

Geneva Framework is just one part of Microsoft's Geneva software family that implements the shared industry vision for an interoperable Identity Metasystem. Geneva comprises three components: "Geneva" Server, Windows CardSpace "Geneva", and "Geneva" Framework. Together, these three components form the core of Microsoft's new claims based access platform. You can refer to the [Geneva website](#) for more information about the server and CardSpace components. The white paper "[Introducing "Geneva"](#)" provides an overview on the full set of Geneva technologies. As of this writing, beta releases of all the three products are available for download.

## Claims-based identity model

When you build claims-aware applications, 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. The idea here is that an external identity system is configured to give your application everything it needs to

---

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.

know about the user with each request she makes, along with cryptographic assurance that the identity data you receive comes from a trusted source.



Figure 1: User Presents Claims

Under this model, single sign-on is much 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

Under this model, your application makes 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.

## It's not Just About Federation

The claims-based model of identity has been incubating for awhile now inside Microsoft. The original reason for proposing this model was to enable federation between organizations, but over time it's become apparent that **claims aren't just for federation**. But some of these terms still linger on. For example, when you use Geneva Framework in your ASP.NET application, one way to perform claims processing is to enable an Geneva Framework component called the *WS-Federation Authentication Module*. Don't let the word "federation" throw you off. There are clear benefits to building applications that outsource authentication and authorization. Any company that has, or plans to have in the future, more than one web application or web service, can benefit by starting with a claims-based model for identity.

## 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 new architecture for identity. Let's start with some terminology.

### Identity

The word "identity" is a very 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 the programming model in Geneva Framework, 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 your user. You may be wondering why I'm using the word "claim", instead of the more traditional "attributes", 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. Each claim is made by an *issuer*, and you'll trust the claim only as much as you trust the issuer. You'll trust a claim made by your company's domain controller more than you would if it were made by the user herself! As you'll see shortly, the Claim class in Geneva Framework has an Issuer property that allows you to find out who issued the claim.

## 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 later be cached in a cookie if a session is desired. Regardless of how they arrive, they must be serialized somehow, and this is where security tokens come in. A *security token* is a serialized set of claims that is 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. In low security situations where cryptography isn't necessary or desired, you can use unsigned tokens, but that's not a scenario I'm going to focus on in this paper.

One of the core features in Geneva Framework is the ability to create and read security tokens. Geneva Framework and the underlying plumbing in the .NET Framework handles all the cryptographic heavy lifting, and presents your application with a set of claims that you can read.

## Issuing Authority

There are lots of different types of issuing authorities, from domain controllers that issue Kerberos tickets, to certificate authorities that issue X.509 certificates, but the specific type of authority I'll be talking about in this paper issues security tokens that contain claims. The issuing authority I'm speaking of is a web application or web service that knows how to issue security tokens. It must have enough knowledge to be able to issue the proper claims for the target relying party given the user making the request, and may be responsible for interacting with user stores to look up claims and authenticate the users themselves.

Whatever issuing authority you choose to buy or build, it will play a central role in your identity solution. When you factor authentication out of your application by relying on claims, you're ultimately just passing responsibility to that authority and asking it to authenticate users on your behalf.

## 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. There's a lot of work that goes into implementing these protocols, but Geneva Framework does all of this heavy lifting for you, making it feasible for someone who isn't an expert in the protocols to get an STS up and running with very little effort.

Geneva Server, one of the products featured in Geneva technologies, is a security token service that you can use instead of building your own STS. You might be wondering what Geneva Server uses for implementing the protocols and building a security token. Yes, you guessed it right, it uses Geneva Framework for all of this heavy lifting.

If you want to build your own STS, Geneva Framework offers all the necessary APIs to make your job easy. It's up to you to figure out how to implement the logic, or rules that drive it (often referred to as *security policy*).

## Relying Party (RP)

When you build an application that relies on claims, you are building a *relying party*. Some synonyms that you may have heard are, *claims aware application*, or *claims-based application*. Web applications and web services can both be built this way, as you'll see later in this paper.

## Basic Scenario

Now that you've learned some basic terminology, here's an example of a claims-based system in action.

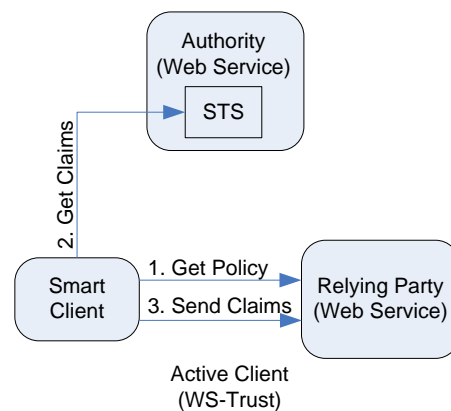


Figure 2: Basic Scenario with Web Services

Figure 2 shows a claims-aware web service (the relying party) and a smart client that wants to use that service. The relying party exposes policy that describes its addresses, bindings, and contracts. But the policy also includes a list of claims that the relying party needs, for example user name, email address, and role memberships. The policy also tells the smart client the address of the STS (another web service in the system) where it should retrieve these claims. After retrieving this policy (1), the client now knows where to go to authenticate: the STS. The smart client makes a web service request (2) to the STS, requesting the claims that the relying party asked for via its policy. The job of the STS is to

authenticate the user and return a security token that gives the relying party all of the claims it needs. The smart client then makes its request to the relying party(3), sending the security token along in the security SOAP header. The relying party now receives claims with each request, and simply rejects any requests that don't include a security token from the issuing authority that it trusts.

## Standards

In order to make all of this interoperable, several WS-\* standards are used in the above scenario. Policy is retrieved using HTTP GET and the policy itself is structured according to the WS-Policy specification. The STS exposes endpoints that implement the WS-Trust specification, which describes how to request and receive security tokens. Most STSs today issue SAML tokens (*Security Assertion Markup Language*). SAML is an industry-recognized XML vocabulary that can be used to represent claims in an interoperable way. This adherence to standards means that you can purchase an STS instead of building it yourself. Or, if you end up in a sticky multi-platform situation, this allows you to communicate with an STS on an entirely different platform and achieve single sign-on across all of your applications, regardless of platform. Identity federation also becomes an option, as I'll explain shortly.

## 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*<sup>2</sup>) can participate as well. Figure 3 shows how this works. The user points her browser at a claims-aware web application (relying party). The web application redirects the browser to the STS so the user can be authenticated. The STS in Figure 3 is wrapped by a simple web application that reads the incoming request, authenticates the user via standard HTTP mechanisms, and then creates a SAML token and emits a bit of JavaScript that causes the browser to initiate an HTTP POST that sends the SAML token back to the relying party. The SAML token in the POST body contains the claims that the relying party requested. At this point it is common for the relying party to package the claims into a cookie so that the user doesn't have to be redirected for each request. The WS-Federation specification includes a section<sup>3</sup> that describes how to do these things in an interoperable way.

---

<sup>2</sup> 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 can't typically be modified to do these things directly, so cookies, redirection, and JavaScript are used to mimic the WS-Trust protocol in a browser-friendly way.

<sup>3</sup> 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.



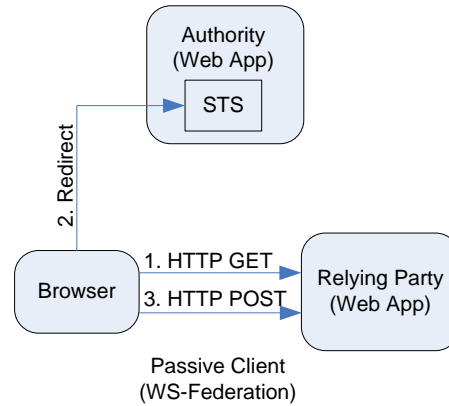


Figure 3: Basic Scenario with a Web Browser

## Identity Federation

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 who is using your application. You don't have to worry about what domain or security realm that user happens to be part of. This makes it a lot easier to *federate identity* with other platforms or organizations.

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.

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 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 ends up doing business with lots of other bike manufacturers, each of which has 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 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 is responsible for authenticating his own 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. 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 Java-based and Microsoft .NET-connected, as long as your applications are built to support federated identity, you have a clear path to achieve single sign-on, and all of the benefits it provides.

Identity federation works by introducing a second issuer. Your applications still trust the same STS they used to, and it will continue to issue all of the tokens that your application needs. But now instead of directly authenticating all users directly, your STS will be configured to accept SAML tokens from partner organizations, leaving it to them to authenticate users in their own realm in a way that makes sense.

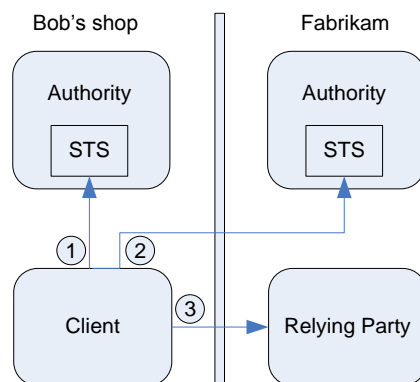


Figure 4: Bob's bike shop federates with Fabrikam

In Figure 4, the client is in a different security realm over in Bob's bike shop, while the relying party is still in Fabrikam's data center. In this case, the client (Alice, say) authenticates with Bob's STS (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 STS, where it evaluates the claims, decides

whether Alice should be allowed to access the relying party in question, and issues a second security token that contains the claims the relying party expects. The client sends this second token to the relying party(3), which now discovers Alice as a new user, and allows her to access the application according to the claims issued by Fabrikam's STS.

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

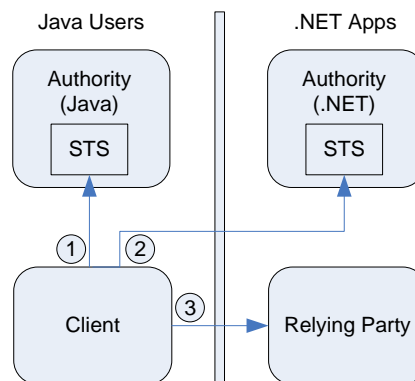


Figure 5: Cross-Platform Identity Federation

Figure 5 shows a company that uses .NET Framework and Geneva Framework to build its applications. They have recently merged with another company whose IT platform is based on Java. Because the Microsoft .NET-connected applications are already claims-aware, the company was able to install an STS built on Java technology and suddenly the Microsoft .NET-connected applications became accessible to users in the Java-based directory, with no changes to application code or even application configuration.

## Information Cards and the Identity Selector

I don't have room in this paper to motivate and explain the ideas behind information cards and identity selectors like Windows CardSpace™, but you can read more about these topics here: <http://msdn.microsoft.com/en-us/library/aa480189.aspx>.

Suffice it to say that an identity selector provides a few additional features that are important in many scenarios:

- Helps users manage multiple identities for the Web
- Helps users select an appropriate identity for a given relying party
- Protects user privacy
- Gives consumers a non-phishable credential

An identity selector can be very helpful in federation scenarios. Consider Fabrikam's STS in Figure 4. Fabrikam has many partners, not just Bob's bike shop. If you asked Fabrikam for its policy, it would

supply a long list of trusted issuers. And imagine if the trust chain was longer, with three or four STSs involved: if you start at the relying party and work backward, you find a whole tree of possible paths from a leaf STS to the relying party. When the identity selector pops up in this case, the only cards that will be lit up are identities that represent leaves on that tree. So when the user selects a particular card, the identity selector knows exactly which path of trust to follow in order to get the required security token.

Geneva Framework includes an ASP.NET control, the InformationCard control, that makes it easy for you to pop up the user's identity selector. The Geneva Framework includes a few samples focused on this control to get you started.

Now that I've introduced some terminology and concepts behind the claims-based identity model, it's time to look at the programming model of Geneva Framework.

## Programming Claims: Geneva Framework

Claims-based identity has been evolving within the Microsoft .NET Framework during the last few years. Active Directory Federation Services (ADFS) was released with Microsoft Windows Server® 2003 R2, and included its own claims-based programming model. Soon afterward, the .NET Framework version 3.0 shipped with a little assembly called System.IdentityModel.dll, which included classes like Claim and ClaimSet, and WCF exposed an AuthorizationContext that allowed you to access these in a web service. Another pillar of this new framework was CardSpace, and some sample code was released that helped to decrypt and parse SAML tokens obtained by dereferencing an information card. And while WCF already has all the plumbing you need to build an STS from scratch, many of the classes you'd need to use are marked *internal*, making the task rather challenging for anyone outside of the WCF team. Suffice it to say that in the .NET Framework 3.0 timeframe, the developer story around claims wasn't very appealing.

Geneva Framework solves this problem. It was designed to unify and simplify claims-based applications. It builds on top of WCF's plumbing to implement WS-Trust and comes with an HttpModule called the *WS-Federation Authentication Module* (FAM) that make it trivial to implement WS-Federation in a browser-based application by simply tweaking your web.config file a bit.

## The Geneva Framework Object Model for Claims

When you build a relying party with Geneva Framework, you're shielded from all of the cryptographic heavy lifting that Geneva Framework (and its underlying WCF plumbing) does for you. It decrypts the security token passed from the client, validates its signature, validates any proof keys<sup>4</sup>, shreds the token into a set of claims, and presents them to you via an easy-to-consume object model.

Geneva Framework represents a claim with the Claim class, whose key properties are shown in Figure 6.

---

<sup>4</sup> A proof key provides assurance that the token wasn't stolen and used by someone other than the subject who requested it. To learn more about proof keys, see the discussion of Kerberos in [The Developer's Guide to Identity](#).

```

public class Claim {
    // some members omitted for brevity
    public virtual string ClaimType { get; }
    public virtual string Value { get; }
    public virtual string ValueType { get; }
    public virtual IDictionary<string, string> Properties { get; }
    public virtual string Issuer { get; }
    public virtual string OriginalIssuer { get; }
    public virtual IClaimsIdentity Subject { get; }
}

```

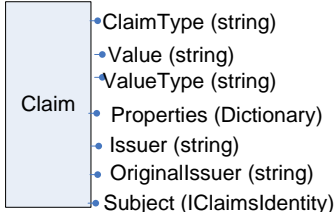


Figure 6: Claim

Claim.ClaimType is a string (typically a URI) that tells you what the claim means. For example, a claim with a ClaimType of "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname" represents a user's first name. This claim type was defined by Microsoft<sup>5</sup> for use with CardSpace. A ClaimType of "http://myclaimtype/role" might be your own simple representation of a role. The point here is that you don't have to wait around for some standards body to define a claim type that you need – as long as you and your issuer agree on what a particular claim means, you can call it anything you want!

Once you know the type of the claim, you can read its value from Claim.Value. In order to reduce dependencies and simplify administration, Geneva Framework represents the value of a claim with a string instead of anything more complicated (such as an object reference that could point to any CLR type). So an integer value of 42 would be represented as "42". An email address is very naturally represented in a string. Anything more complicated and it is recommended to use standard XML schema types to serialize the value into a string. This is where Claim.ValueType comes in; it helps you figure out how to deserialize the value of the claim by telling you the format of the value. The Microsoft.IdentityModel.ClaimValueTypes class (Figure 7) includes a number of helpful value types that can be used to represent claims, and of course you can define your own if you build your own issuer.

```

public static class ClaimValueTypes {
    // I have omitted some elements for brevity
    public const string Boolean = "http://www.w3.org/2001/XMLSchema#boolean";
    public const string Date = "http://www.w3.org/2001/XMLSchema#date";
    public const string Datetime = "http://www.w3.org/2001/XMLSchema#dateTime";
    public const string Double = "http://www.w3.org/2001/XMLSchema#double";
    public const string Email = "http://www.w3.org/2001/XMLSchema#Email";
    public const string Integer = "http://www.w3.org/2001/XMLSchema#integer";
    public const string String = "http://www.w3.org/2001/XMLSchema#string";
    public const string Uri = "http://www.w3.org/2001/XMLSchema#anyURI";
    public const string UpnName = "http://www.w3.org/2001/XMLSchema#UpnName";
}

```

Figure 7: ClaimValueTypes

Claims are supposed to be about the subject, typically a human who is using your application. But sometimes you want more information about the claim itself. Take an email claim as an example. Maybe along with the email address, you want the issuer to tell you when the email address last changed. This is the reason for the Claim.Properties collection. Geneva Framework allows an STS to add metadata

<sup>5</sup> This and several other fundamental claim types are documented in the [Identity Selector Interop Profile specification](#).

about a claim to this collection, and it'll be sent along with the claim so that your application can make use of it.

One claim that might seem a bit like metadata is the authentication method. Did the user present a password? An information card? A smart card? These sorts of questions often come up in high security scenarios, and some applications restrict features or resources based on the strength of the technique used to authenticate the user. Geneva Framework represents this information as a claim, as you'll see in the Step-up Authentication section below.

The Claim class includes a property called Issuer. This is a simple string that gives your application a name for the issuer of the claim. In federation scenarios, a chain of two or more issuers are involved (as shown earlier in Figure 4). In this case, Claim.Issuer names the last issuer in the chain (Fabrikam in Figure 4), while Claim.OriginalIssuer names the first issuer in the chain (Bob's shop in the same figure). You can use both of these tidbits to personalize or authorize access. For example, in Figure 4, Fabrikam might want to have a special discount page that only users from Bob's Bike Shop are allowed to use.

But many (probably most) applications won't care precisely who issued the claims in the user's identity; all these applications need to worry about is that the issuer is one it should trust. In order to answer this question of trust, Geneva Framework includes an abstract class called IssuerNameRegistry. There are a couple of built-in implementations of this abstraction that allow you to specify trusted issuers via configuration, or you can derive from this class and implement GetIssuerName yourself. This method takes a security token as input and returns a name as output. This allows you to pick an issuer naming scheme that makes sense for your application, and it also allows you to validate and determine if a security token comes from an issuer that you trust. If you don't like a particular security token for any reason whatsoever, you simply throw an exception from GetIssuerName in order to indicate to Geneva Framework that the request should be denied.

## Introducing IClaimsIdentity

Remember `IIdentity` from the .NET Framework? It is a very simple interface that allows you to discover the user's name. Since an issuer has the ability to tell you much more than just a name, Geneva Framework defines a new interface that extends `IIdentity`. It's called, aptly enough, `IClaimsIdentity`. In Geneva Framework, when you look at a user's identity, you can get her name the same way you always have, but you can also look at `IClaimsIdentity.Claims` to get more bits of the user's identity, like her email address.

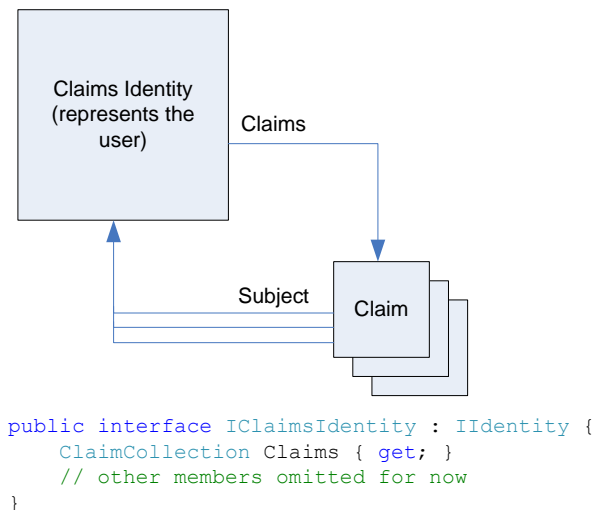


Figure 8: Getting at Claims via IClaimsIdentity

Figure 8 shows how claims are exposed from `IClaimsIdentity` via the `Claims` property (I will show the full definition of this interface a bit later in Figure 12). Keep in mind that the user whose identity you are examining is called the *subject*. Note how you can enumerate the list of claims in the identity via the `Claims` collection, and you can get back to the subject's `IClaimsIdentity` via the `Subject` property of any of those claims.

## IClaimsPrincipal

Remember how Geneva Framework extended the existing `IIdentity` interface with `IClaimsIdentity`? Well, Geneva Framework also extends `IPrincipal` with `IClaimsPrincipal` (see Figure 9).

```

public interface IClaimsPrincipal : IPrincipal {
    ClaimsIdentityCollection Identities { get; }
}

```

Figure 9: IClaimsPrincipal

`IClaimsPrincipal` exposes a collection of identities, each of which implements `IClaimsIdentity`. In the common case, there will be a single issuer and a single token, and the `Identities` collection will only have one element. In this mainstream case, you can use `IPrincipal.Identity` to get at the identity as usual. But it's possible in advanced scenarios for a relying party to ask (via policy) for more than one security token,

potentially from different issuers, in which case having access to a collection of identities becomes important.

In some scenarios it's useful to write your own class that implements `IPrincipal` and `IClaimsPrincipal`. You can do this simply by deriving from `ClaimsAuthenticationManager` and implementing the `Authenticate` method. The `Authenticate` method has access to the `IClaimsPrincipal` originally created by the framework, and you can make necessary transformations to the existing claims or completely replace the existing `IClaimsPrincipal` with your own implementation. Geneva Framework will then use your `IClaimsPrincipal` and make it available via `Thread.CurrentPrincipal` and other means that an ASP.NET developer would use to obtain the caller's `IPrincipal`, such as `HttpContext.User`.

## ClaimsPrincipal

```
public class ClaimsPrincipal : IClaimsPrincipal {
    // from IPrincipal
    public IIdentity Identity { get; }
    public bool IsInRole(string role);

    // from IClaimsPrincipal
    ClaimsIdentityCollection Identities { get; }
}
```

Figure 10: ClaimsPrincipal

The `ClaimsPrincipal` class (Figure 10) is the default implementation of `IClaimsPrincipal`, and while it implements the `Identities` property of `IClaimsPrincipal`, it also implements the more familiar `IsInRole` method and `Identity` property from `IPrincipal`. And since a typical relying party will receive a single `IClaimsIdentity` in the `Identities` collection, `ClaimsPrincipal.Identity` simply returns the first element in the `Identities` collection, as shown in Figure 11.

```
public IIdentity Identity {
    get {
        if (this._identities.Count > 0)
            return this._identities[0];
        else return null;
    }
}
```

Figure 11: ClaimsIdentity.Identity property implementation

## IClaimsIdentity Defined

So if all it has to work with is a set of arbitrary claims, how does Geneva Framework implement `IPrincipal.IsInRole` or `IIdentity.Name`, which are commonly used by application developers in existing web applications and services? One possible approach would have been to predefine a `ClaimType` for roles and a `ClaimType` for names, and force everyone to use them, but Geneva Framework is more flexible than that. In your system, the type of claim you pick to represent a role or name might be very different than what another company would choose. The Geneva Framework solution becomes clear when you look at the full definition of `IClaimsIdentity` (Figure 12). Geneva Framework decided to leave



this up to your application and easily configure `NameClaimType` and `RoleClaimTypes` to indicate which claims represent the user's name and her roles in the `web.config` under `Microsoft.IdentityModel` section.

```
public interface IClaimsIdentity : IIdentity {
    ClaimCollection Claims { get; }
    string NameClaimType { get; set; }
    ICollection<string> RoleClaimTypes { get; }
    string Label { get; set; }
    IClaimsIdentity Delegate { get; set; }
}
```

Figure 12: `IClaimsIdentity`

This has an exciting implication for developers: any code you have that already relies on `IPrincipal` and `IIdentity` doesn't need to change. If you're using the `PrincipalPermission` attribute to control access to a web service method, you can continue to use it as long as your issuer specifies the same set of roles that your service was expecting. If you're using ASP.NET's `LoginView` control, it'll also continue to work because it's based on `IPrincipal.IsInRole()`.

If there are no `NameClaimType` or `RoleClaimTypes` configured in the application then by default Geneva Framework uses "<http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name>" for `NameClaimType` and "<http://schemas.microsoft.com/ws/2008/06/identity/claims/role>" for `RoleClaimTypes`. Note that while it's possible that the issuer may set `NameClaimType` and `RoleClaimTypes` when it creates an `IClaimsIdentity` these properties are not serialized as part of the security token and so they don't flow between the STS and the relying party. It is uncommon to have STS decide which roles that the application need to use.

In looking at Figure 12, you might be wondering about the `Label` and `Delegate` properties. `Label` is a string that can be used to distinguish one identity from another in the more complicated case where a relying party receives multiple security tokens (as I mentioned earlier, this is an uncommon case, so the `Label` property won't be used by the vast majority of developers). Note that while it's possible that this may change in the future, in this version of Geneva Framework, `Label` is not serialized as part of the security token and so it doesn't flow between the STS and the relying party.

The `Delegate` property is a more interesting and advanced topic. It helps support the delegation of credentials in multi-tier systems, where a middle tier makes requests to a back end system while "acting as" the client. I'll cover this optional feature in more detail later in this paper.

## How to get at Identity

So far you've learned how Geneva Framework represents claims, subjects, and issuers, and how it extends the traditional `IPrincipal` and `IIdentity` interfaces to add support for claims-based identity. But where do you get at these interfaces? There are lots of places where `IPrincipal` and `IIdentity` are already exposed in the .NET Framework, and with Geneva Framework you should continue to use them: for example, `Thread.CurrentPrincipal` from the .NET Framework, or `HttpContext.User` from ASP.NET. In a relying party, Geneva Framework sets up all of these properties so that you can access `IClaimsPrincipal`

and `IClaimsIdentity` from any of these familiar places. Figure 13 shows a couple of different ways to get at the user's identity in a typical web application.

```
protected void Page_Load(object sender, EventArgs e) {
    IClaimsPrincipal p =
    (IClaimsPrincipal)Thread.CurrentPrincipal;
    IClaimsIdentity ci = p.Identities[0];
    DisplayClaims(ci);
}

protected void Page_Load(object sender, EventArgs e) {
    IClaimsIdentity ci = (IClaimsIdentity)User.Identity;
    DisplayClaims(ci);
}
```

Figure 13: Accessing `IClaimsIdentity`

## Programming with Claims: a Practical Example

The Geneva Framework object model for claims may seem a bit complicated at first glance, with subjects, issuers, claim types and values, but in practice it's very easy to use. Figure 14 shows a typical example from a claims-aware ASP.NET web application. This example sends a personalized email to the user when she clicks a button.

```
protected void SendLetter_Click(object sender, EventArgs e)
{
    IClaimsIdentity id =
        ((IClaimsPrincipal)Thread.CurrentPrincipal).Identities[0];

    // you can use a simple foreach loop to find a claim...
    string usersEmail = null;
    foreach (Claim c in id.Claims) {
        if (c.ClaimType == System.IdentityModel.Claims.ClaimTypes.Email) {
            usersEmail = c.Value;
            break;
        }
    }

    // you can also use LINQ to find a claim
    string usersFirstName = (from c in id.Claims
        where c.ClaimType == System.IdentityModel.Claims.ClaimTypes.GivenName
        select c).First().Value;

    StringBuilder body = new StringBuilder();
    body.AppendFormat("Dear {0},", usersFirstName);
    body.AppendLine();
    body.AppendLine("Thank you for shopping with us!");

    new SmtplibClient().Send(new MailMessage(
        "admin@fabrikam.com",
        usersEmail,
        "Message from Fabrikam",
        body.ToString()));
}
```

Figure 14: Sending a Personalized Email

In this example, the code uses `Thread.CurrentPrincipal` to access the user's identity. Then it loops through all of the claims for the user via `IClaimsIdentity.Claims`, looking for the ones it needs right now: first name and email address. It then uses those claims to send a personalized email message to the

user. The example also shows two ways of finding claims. The code finds the email claim by with a foreach loop, and uses a LINQ expression to find the first name claim. Both techniques work just fine.

## Configuring Geneva Framework

The code in Figure 14 made a lot of assumptions. It assumed the caller was authenticated and that her first name and email address had been sent as claims. The reason this program can make these assumptions is because it has a web.config file that wires up the WS-Federation Authentication Module (FAM) from Geneva Framework and configures it with the address of an STS that can authenticate the user and supply these types of claims. Figure 15 shows the relevant parts of web.config for the personalized email example.

```
<configuration>
  <configSections>
    <section name="microsoft.identityModel"
      type="Microsoft.IdentityModel.Configuration.MicrosoftIdentityModelSection, Microsoft.IdentityModel,
      Version=0.5.1.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
  </configSections>
  <system.web>
    <compilation debug="true">
      <assemblies>
        <add assembly="Microsoft.IdentityModel, Version=..." />
      </assemblies>
    </compilation>
    <authentication mode="None"/>
    <authorization>
      <deny users="?" />
    </authorization>
    <httpModules>
      <add name="WSFederationAuthenticationModule"
        type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule..." />
      <add name="SessionAuthenticationModule"
        type="Microsoft.IdentityModel.Web.SessionAuthenticationModule..." />
    </httpModules>
  </system.web>

  <microsoft.identityModel>
    <issuerNameRegistry type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry">
      <trustedIssuers>
        <add thumbprint="..." name="...CN=SampleSTS..." />
      </trustedIssuers>
    </issuerNameRegistry>
    <audienceUris>
      <add value="https://localhost/MyWebApp" />
    </audienceUris>
    <federatedAuthentication enabled="true">
      <wsFederation passiveRedirectEnabled="true"
        issuer="https://localhost/STS/"
        realm="https://localhost/MyWebApp" />
    </federatedAuthentication>
    <serviceCertificate>
      <certificateReference x509FindType='FindBySubjectName' findValue='localhost' storeLocation='LocalMachine'
        storeName='My' />
    </serviceCertificate>
  </microsoft.identityModel>
</configuration>
```

Figure 15: Typical Geneva Framework Configuration for a Relying Party

There are two things going on here in the <system.web> section. The config first references the Geneva Framework assembly, Microsoft.IdentityModel.dll. Then it wires up the FAM, which is an HttpModule that plugs into ASP.NET pipeline so that it can listen for the AuthenticateRequest event.

SessionAuthenticationModule enables sessions by issuing cookies. I'll explain in more detail how the FAM fits into the ASP.NET pipeline later in this paper.

The <microsoft.identityModel> section is new to Geneva Framework. Application can configure a list of trusted issuers in <issuerNameRegistry> element by choosing the issuerNameRegistry type as ConfigurationBasedIssuerNameRegistry. The <audienceUris> section is the place to list the target URIs to which the FAM should expect security tokens to be delivered. If the STS posts a security token to a URI not in this list then FAM throws an exception. Also you should note that FAM performs a case sensitive URI comparison between the incoming URI and the list from <audienceUris> section, so it is important to consider case when you configure this setting.

The <federatedAuthentication> section is where you configure the FAM. Setting enabled="true" tells the FAM to work its magic in the AuthenticateRequest event and convert incoming security tokens into an IClaimsPrincipal. The <wsFederation> section with passiveRedirectEnabled set to "true" tells the FAM that when a user points her browser at the application, the FAM should automatically redirect the browser to a particular STS (the *issuer* attribute indicates the URL for the STS) where the user will be authenticated and receive a security token. When the user's browser is redirected, the value of the *realm* attribute will be included in the request to the STS, telling it which application is in use. The <serviceCertificate> section is where you would specify the application certificate that FAM should use to decrypt incoming security tokens.

## Understanding the WS-Federation Authentication Module (FAM)

The FAM is an HttpModule that is specifically designed to make it easy to build federated claims-aware web applications using ASP.NET 2.0. There are two options available to build federated claims-aware web applications. One option is to use the FAM and SessionAuthenticationModule and provide passive redirect based protection; another option is to provide a login page that uses the FederatedPassiveSignIn control, which is an ASP.NET control offered by Geneva Framework. I'll cover the control aspects in a separate section; I'll focus on the FAM in this section.

As the name implies the FAM is capable of handling the WS-Federation protocol; while SessionAuthenticationModule is specifically designed to be protocol-agnostic and handle session cookies. Both of these modules are a required bit of plumbing that must be configured, as I showed earlier in Figure 15, in your federated claims-aware web application. You may wonder why these are two separate modules. The motivation for this split is to provide the flexibility for configuring additional authentication modules in conjunction with the protocol-agnostic SessionAuthenticationModule.

In essence the FAM extracts the claims from the security token issued by an STS and makes them easily accessible to the application. However, it is possible that some applications may not have an STS and just want to convert Windows authentication information into an IClaimsPrincipal that is usable by the claims-aware aspects of the application. Geneva Framework provides an authentication module for this scenario too; ClaimsPrincipalHttpModule is the class that addresses this scenario. In this scenario the default claims generated are based on the Windows identity and includes user name, group SIDs, and other authentication information. This module eliminates the application's need to get a security token

from an STS and provides a way to always have claims-based principal available in an ASP.NET application.

## FAM Events

As is typical with HttpModules in ASP.NET, the FAM fires off several events that allow you to customize its default processing. Keep in mind that all of the work done by the FAM occurs during the ASP.NET AuthenticateRequest event. All of the events fired by the FAM will occur during this part of the ASP.NET request pipeline.

### ConfigurationLoading/Loaded

ASP.NET programmers are already familiar with the Application\_Start event, which fires during the very first request that comes through the pipeline once an application starts up. The ConfigurationLoading and ConfigurationLoaded events are similar to this: the first time the FAM processes an AuthenticateRequest event, it lazily loads its settings from web.config and gives you a chance to dynamically change that configuration by handling these events. There are certain aspects of the FAM that are not available through configuration, and ConfigurationLoaded is a useful event to handle in case you want to programmatically configure the FAM beyond what's in web.config.

### SecurityTokenReceived

This is a very useful event that fires with every message that contains a WS-Federation passive sign-in response, right before the FAM does its work. One important thing you can do here is to reject the token before the FAM processes it. You have access to the security token received and you can implement your own custom validation code that decides whether the token is rejected or not.

If you want to leave some pages that get WS-Federation passive messages open to anonymous access then you can set the FAM's Cancel property to "true" which makes the FAM *not* do anything and you can configure the authorization policy in the ASP.NET pipeline to allow anonymous access to those pages.

### SecurityTokenValidated

This event is raised after the security token is validated by the FAM and an IClaimsPrincipal is created with the claims extracted from the token. Note that the IClaimsPrincipal that is available as a property of the event arguments has already passed through any configured claims authentication manager, which has done the necessary transformations. One of the simplest things you can do here is to audit this event for successful token validations that occur in the application.

### SessionSecurityTokenCreated

Early in this paper I showed how browser-based applications receive claims from an STS via an HTTP POST. At this stage of processing, the FAM has received the security token, validated and parsed it into a set of claims, created a session security token with those claims in it, and is ready to issue a cookie to cache that session security token for a little while (so that the user isn't constantly getting redirected back to the STS). Geneva Framework calls the contents of this cookie a "session security token", and has a class called SessionSecurityToken to represent it.

By handling this event you'll be informed whenever fresh claims have been delivered from an STS and are about to be cached for a configured time in a cookie. If you plan on storing claim values in an application data store, this would be a great time to do that. See the 'Caching Claim Values Over Time' section for an example. Also, by handling this event you have the opportunity to modify the session security token per your needs. The FAM takes any modifications made to the session security token and then writes out a cookie with the modified session security token in it. For example if you want the cookie to be issued for single use only instead of persisting for the session, you can set the `WriteSessionCookie` property of `SessionSecurityTokenCreatedEventArgs` to "false".

### SignedIn

This event fires right after the `IClaimsPrincipal` is set in the appropriate places (`Thread.CurrentPrincipal` and `HttpContext.Current.User`). As you can imagine this event doesn't have any event arguments and its main purpose is to allow applications to audit the user signed-in event.

### SignInError

At this stage of processing the FAM has received the token from an STS, extracted claims from it, and created a session security token, but encounters an error when it tries to write out a cookie with the session security token. This event is also triggered when errors occur in token validation process.

You can use this event to handle the exceptions thrown by the FAM and convert them to user-friendly messages per your needs. Also, during application development this event is very helpful in debugging the cause of the exceptions thrown.

### SigningOut

At this stage of processing, the user has indicated her desire to sign out of your application. This doesn't always happen: some users simply stop making requests to your application without explicitly signing out. But when this event does fire, you can use it as a way to more aggressively release resources that you may have been holding for the user. If you have a good reason to do so, you can choose to cancel the `SignOut` operation via your event handler and keep the user signed in.

### SignedOut

This event fires when the user or application has triggered sign-out process and the FAM has deleted the cookie corresponding to the session. As you can imagine this event doesn't have any event arguments and its main purpose is to allow applications to audit the event that the user has signed out.

## SessionAuthenticationModule Events

### SessionSecurityTokenEventReceived

Once a session security token has been issued in the form of a cookie, during all subsequent requests (until the session security token expires) the `SessionAuthenticationModule` will simply read the contents of this cookie instead of redirecting the browser back to the STS. By default session security tokens are set to expire after 10 minutes; however, at runtime, applications typically extend the lifetime as needed.

This event is triggered after the session security token is received and processed by SessionAuthenticationModule. One possible use of this event is to allow applications to check for expiration of the session security token and extend the lifetime as needed. You can find an example of this in the Geneva Framework samples collection: **Samples\Extensibility\Claims Aware AJAX Application**, in the AjaxRP (relying party) project.

### ConfigurationLoading/Loaded

Similar to the FAM events described above, the SessionAuthenticationModule fires the ConfigurationLoading and ConfigurationLoaded events. The details I've covered on these events for the FAM are true for the SessionAuthenticationModule as well. You can find an example of this in the Geneva Framework samples collection: **Samples\End-to-end Scenario\Identity Delegation**, in the WFE (web front-end) project.

## Ideas for Hooking FAM and SessionAuthenticationModule Events

Here are some practical examples of how you can hook FAM events to achieve various goals.

### Substitute your own Principal

Claims are very simple to read – just enumerate a collection and look for the claim type you want. But if you've got a lot of code that depends on claims, this can get tedious. Recall the code I wrote that used claims to send a personalized email message to the user (Figure 14). Wouldn't it be easier instead to write code like that shown in Figure 16?

```
protected void SendLetter_Click(object sender, EventArgs e) {
    MyPrincipal customPrincipal = (MyPrincipal)Thread.CurrentPrincipal;

    StringBuilder body = new StringBuilder();
    body.AppendFormat("Dear {0},", customPrincipal.FirstName);
    body.AppendLine();
    body.AppendLine("Thank you for shopping with us!");

    new SmtplibClient().Send(new MailMessage(
        "admin@fabrikam.com",
        customPrincipal.Email,
        "Message from Fabrikam",
        body.ToString()));
}
```

Figure 16: Simplified Personalized Email Example

In this new version, you don't have to enumerate claims to find the user's first name and email address. A custom principal class already has that sorted out, and the code can simply use its properties to access the claims. This is even more useful if you have claims that are more complicated than strings. Because Geneva Framework passes all claims as strings, you may need to deserialize the value of a claim into an instance of a class. Using a custom principal to expose this is a great way to centralize this code. Figure 17 shows the implementation of the custom principal used in the code above.

```

public class MyPrincipal : ClaimsPrincipal, IMyClaims {
    public MyPrincipal(IClaimsPrincipal claimsPrincipal)
        : base(claimsPrincipal)
    {}
    public string FirstName { get; set; }
    public string Email { get; set; }
}

```

Figure 17: Custom Principal Definition

With this class definition, all you need to do now is ensure that the FAM uses an instance of `MyPrincipal` instead of the default `ClaimsPrincipal` when it sets up the user's identity. The `SecurityTokenValidated` event in the FAM is the place to do this, as shown in Figure 18.

```

void WSFederationAuthenticationModule_SecurityTokenValidated(object sender,
    SecurityTokenValidatedEventArgs args) {
    MyPrincipal customPrincipal = new MyPrincipal(args.ClaimsPrincipal);
    foreach (Claim c in args.ClaimsPrincipal.Identities[0].Claims) {
        if (c.ClaimType == System.IdentityModel.Claims.ClaimTypes.GivenName)
            customPrincipal.FirstName = c.Value;
        if (c.ClaimType == System.IdentityModel.Claims.ClaimTypes.Email)
            customPrincipal.Email = c.Value;
    }
    args.ClaimsPrincipal = customPrincipal;
}

```

Figure 18: Injecting a Custom Principal

You can also achieve the same customization by creating a custom claims authentication manager and overriding the `Authenticate` method.

## Caching Claim Values over Time

One of the first things you'll notice when you build a claims-aware web application is that you no longer need to provision user accounts. This is one of the responsibilities that you can shrug off when you rely on an external party (your issuing authority) to authenticate your users. But even claims-aware applications may need to store per-user data such as preferences. And sometimes it makes sense to store claims for later use. For example, what if you wanted to send an email blast out to every user of your application?

The solution to this problem is simple: just keep a record of the claims you think you might want to remember during times when the user may not be logged on. The user's name and contact information are prime candidates for a profile. Just keep in mind that as soon as you write that data, it's by definition stale, and will only become more so as time goes by, so you should update the user's profile every time you think you might have fresh information from your issuer. It's also wise to keep a timestamp of the last time the information was updated, so you can track just how stale the information is.

The `SessionSecurityTokenCreated` event fired by the FAM is a great place to freshen the user's profile, because it only fires when the FAM parses actual claims posted from the STS. Figure 19 shows an example of how this could be done in `global.asax`:



```
void WSFederationAuthenticationModule_SessionSecurityTokenCreated(object sender,
SessionSecurityTokenCreatedEventArgs args)
{
    string usersFirstName = null;
    string usersEmailAddress = null;
    foreach (Claim c in args.ClaimsPrincipal.Identities[0].Claims)
    {
        if (c.ClaimType == System.IdentityModel.Claims.ClaimTypes.GivenName)
            usersFirstName = c.Value;
        if (c.ClaimType == System.IdentityModel.Claims.ClaimTypes.Email)
            usersEmailAddress = c.Value;
    }
    DateTime lastUpdated = DateTime.Now;

    // the definition of this method is up to you
    UpdateUserProfile(usersFirstName, usersEmailAddress, lastUpdated);
}

void UpdateUserProfile(string firstName, string email, DateTime lastUpdated)
{
    // update whatever data store you're using to store profile details
}
```

Figure 19: Tracking Claims in a User Profile

## Controls

Geneva Framework includes a couple of helpful ASP.NET controls that help an end user get involved with authentication in passive scenarios. For example, you can have different instances of these controls that redirect the user to different issuers, depending on your needs, as I describe in the section on authentication assurance. All of these controls derive from a base control that supplies common properties such as whether you want a session to result, or just a single blast of claims. Typically these controls are embedded to login pages; and there are lots of references on how to configure a login page using Forms Authentication in ASP.Net.

Once the user signs in using either of these controls, you can discover the resulting claims using the same mechanisms I've discussed throughout this paper.

## InformationCard

This control generates the required HTML to trigger a browser to pop up an identity selector, and makes it easy for a web page to obtain claims from both personal as well as managed information cards.

The properties of this control allow you to specify the issuer you trust, the type of token you require, and the claim types that you require.

```

<%@ Register Assembly="Microsoft.IdentityModel, ..."
    Namespace="Microsoft.IdentityModel.Web.Controls"
    TagPrefix="idfx" %>
<idfx:InformationCard
    ID="InformationCard1"
    runat="server"
    DisplayRememberMe="False"
    SignInMode="Session"
    SignInText="Sign in to Fabrikam using an Information Card"
    TitleText="Click here"
    Issuer="http://localhost/ManagedCardSTS/windows.svc"
    IssuerPolicy="https://localhost/ManagedCardSTS/windows.svc/mex"
    RequiredClaims="[space delimited list of claim types go here]"
    OnSignInError="OnSignInError"
    DisplayType="CardTile"
    RequireUserInteraction="True"
/>

```

Figure 20: The InformationCard Control

Figure 24 shows an example of this control on a web page, along with the ASP.NET Register directive you'll need in order to use it. For a sample application that uses this control refer to the **Geneva Framework** samples collection and **Samples\Getting Started\Simple Web Application with Information Card Signin**.

One property that I want to highlight in this InformationCard control; that is "DisplayType" property, which can be set to either "CardTile" or "None". CardTile display type is a new feature added in [Windows CardSpace "Geneva"](#). If you are familiar with current version of Windows CardSpace shipping in .Net Framework 3.0 you would have noticed the CardSpace UI that pops up when you click on an Information Card control. This CardTile feature is to make the login experience user friendly and show the card image on the web page or application itself. The sample mentioned above illustrates this; but note that you would need to install the Windows CardSpace Geneva before trying out the sample.

## FederatedPassiveSignIn

This login control is similar to the InformationCard control in that it initiates a login when the user clicks it, but instead of popping an identity selector, this simply redirects the user's browser to a passive STS. This login control effectively works the same as the FAM but the functionality is happening in the scope of login page instead being in the ASP.Net pipeline. This control can be useful if you want to allow various levels of authentication for a given page. For example, the AuthAssurance example (**Samples\End-to-end Scenario\Authentication Assurance**) uses this control to prompt the user to sign in with a certificate.

```

<%@ Register Assembly="Microsoft.IdentityModel, ..."
    Namespace="Microsoft.IdentityModel.Web.Controls"
    TagPrefix="idfx" %>

<idfx:FederatedPassiveSignIn
    runat="Server"
    TitleText="Sign In to Access More Features"
    SignInText="Sign In"
    Issuer="https://localhost/MyPassiveSTS/Login.aspx"
    Realm="https://localhost/MyRelyingParty/"
    OnSignInError="MyErrorHandler"
    VisibleWhenSignedIn="false"
    AutoSignIn="false"
    DisplayRememberMe="false"/>

```

Figure 21: The FederatedPassiveSignIn Control

Figure 25 shows an example of this control on a web page, along with the ASP.NET Register directive you'll need in order to use it. Look at **Samples\End-to-end Scenario\Authentication Assurance** for an example of how this control can be used.

## Using Geneva Framework in Web Services

So far all of my examples have been browser-based web applications. Now I'd like to show how you can use Geneva Framework to build and consume claims-aware services.

### Writing a Claims-Aware Service

A claims-aware web service expects to receive a security token (SAML by default) in the security header of the SOAP envelope. This security token identifies the user (subject) to the web service (relying party). But the service usually also proves its identity to the user, and in claims-based systems this is done by configuring the service with a certificate.

WCF already supports building web services that accept tokens issued from an STS. Your service just needs to indicate to WCF the address of the STS's metadata. In order to do this, today you need to create a WSFederationHttpBinding or a custom WCF binding, which is easy to do both in code and config.

The example I'll be walking through here is from the Geneva Framework samples collection: **Samples\End-to-end Scenario\Identity Delegation**. The relying party in this example is in the **Service2** project, and all of the code for this web service can be found in Service2.cs. If you examine the Main() method, you'll see how to construct the required binding in code (you can also do this in config if you prefer). The bit that you'll customize for your own web service is the IssuedSecurityTokenParameters class, which specifies the type of token being requested (SAML 1.1 in this example), and the addresses of the metadata and WS-Trust endpoints for the issuer's STS, which in this example is also built using Geneva Framework in the **STS** project.

There's nothing special about the ServiceHost configuration other than the explicit shutting off of a couple of certificate validation features that WCF normally runs, in order to support the test certificate

that ships with the samples. *This is not appropriate in production code*, where you normally want to validate certificate chains and check for certificate revocation.

So far, nothing about this web service is Geneva Framework specific. The line of code that calls `FederatedServiceCredentials.ConfigureServiceHost()` is where Geneva Framework comes in. This call gives Geneva Framework a chance to install Geneva Framework specific behaviors that hook Geneva Framework into WCF's claims-processing pipeline. This means that from inside your service method (`ClaimsAwareWebService.ComputeResponse` in this example), you'll be able to reach up into Geneva Framework via `Thread.CurrentPrincipal` and use the simplified Geneva Framework object model to read the claims sent with the request. The code for reading claims in a web service looks exactly the same as it does in my earlier browser-based examples. Once you've got the claims, it's the exact same programming model as in the ASP.NET environment: `IClaimsPrincipal`, `IClaimsIdentity`, and `Claim`. One point to note is that once Geneva Framework is enabled for a WCF service `ServiceSecurityContext.Current` can no longer be used in that service.

### Calling a Claims-Aware Service

WCF already has everything a client needs to call a claims-based service, so you don't need Geneva Framework on the client side at all! All the client needs is the .NET Framework 3.0, so when you're developing a smart client, you can work the way you are used to with WCF, which is typically to point Visual Studio or `svcutil.exe` at the metadata address for the service you want to call, and generate a proxy that you can use to make those calls. The binding that the service configured will be exposed via WSDL, and that will give the client all the information it needs, including the type of token required by the relying party and the address of the STS from which it should be obtained. Keep in mind that the binding in this config file wasn't written by hand: it was generated by `svcutil.exe` along with the proxy.

If you examine the code in `Client.cs`, you'll see that there's nothing interesting going on here; the smart client developer is blissfully unaware of the way identity is being managed on the back end.

### Using Geneva Framework to Build a Security Token Service

This paper is primarily aimed at developers building relying parties, but I would be remiss if I didn't at least briefly discuss how an STS can be built using Geneva Framework.

#### Issuing Authority versus STS

Before I show you how to implement an STS using Geneva Framework, here's an important caveat: in any non-trivial claims-based system, the STS is a very small part of an issuing authority. The STS is the component that accepts incoming requests, validates, decrypts, and shreds incoming security tokens into claims, and does the opposite for outgoing security tokens. Geneva Framework takes care of all of that heavy lifting. But what Geneva Framework does not do is provide a framework for managing or administering policy, which you can think of as the logic, or the rules, behind the STS.

Here's an example of some of the questions that an issuer's policy could answer:

- What applications am I providing security tokens for?

- What claims do those applications care about?
- How should I authenticate users?
- Do different apps have different authentication requirements?
- What partners am I federating with?

So if you're trying to figure out whether you should build your own STS or use one that was built by someone else (like the ["Geneva" Server](#), for example), most people would be much better off buying one.

If, on the other hand, your company wants to do something that existing products don't support, you can use Geneva Framework to get started building your own authority.

### What an Issuer has to Work With

A request for a security token includes details about what claims the relying party needs. This might come in the form of a WS-Policy document from the relying party that explicitly lists which claim types it needs. On the other hand, it might simply be a string that identifies a particular relying party, in which case it's the issuer's job to know what claims that particular application needs.

Recall from Figure 2 that the request originates from the user (the subject), and it's the issuer's job to either authenticate that user, or bounce her to some other STS to be authenticated. In the latter case, the issuer will receive a set of claims about the subject from the upstream issuer.

So ultimately the issuer has a couple of very important categories of information to work with: information about the subject, and information about the relying party.

The issuer must take this information, along with any contextual data it may need (time of day, etc.) and either reject the request if it can't satisfy the relying party's needs, or issue a security token containing claims that will make the relying party happy.

Where does the issuer look to get claims? If the user is part of the issuer's enterprise, the user's record in the enterprise directory would be a natural source for claims. Sometimes user data is stored in other, less obvious places such as SQL databases, so the issuer might need to look in multiple places to get the claims that the relying party needs. But centralizing this logic for looking up claims is very useful: as your organization adds more and more applications, they all benefit by not having to figure out how to query the various user stores themselves. Issuers are typically configured by IT pros, and they usually know best where to get any given attribute for a user, since they manage identity on a daily basis.

### Geneva Framework STS Architecture

Geneva Framework includes a base class, `SecurityTokenService`, which you derive from to create a custom STS. There are two key methods that you must override in your derived class, which are shown in Figure 20.

```
protected abstract Scope GetScope(IClaimsPrincipal subject,
                                   RequestSecurityToken request)

protected abstract IClaimsIdentity GetOutputClaimsIdentity( IClaimsPrincipal subject,
                                                            RequestSecurityToken request,
                                                            Scope scope)
```

Figure 22: Methods to Override on SecurityTokenService

The first method, `GetScope`, gives you the opportunity to normalize the relying party's address and choose signing and encryption keys (security tokens are typically encrypted so that only the relying party can read them, and signed by the issuing authority). If you're accessing a database during any of this work, you can also use that database round-trip to prefetch data that `GetOutputClaimsIdentity` will need, in the interest of efficiency.

The second method, `GetOutputClaimsIdentity`, lets you define the claims that will be folded into an `IClaimsIdentity` that will be serialized into a security token. `GetOutputClaimsIdentity` is the method that should fire off all of the logic in your issuer, evaluating the claims for the subject, information about the relying party, and ultimately result in a set of claims exposed from an `IClaimsIdentity` implementation.

By overriding these two methods, you've effectively wired the STS up to your policy for claims issuance. Now all you have to do is actually issue those claims, and how you do that depends on whether you are exposing your STS for active or passive clients.

### WS-Trust (for Active Clients) Example

The example I'll use for this discussion is from the Geneva Framework samples collection: **Samples\Getting Started\Simple Claims Aware Web Service**. There are three projects in this solution: a WCF client, a WCF service for the relying party, and a WCF service for the STS. The STS project is called `SimpleActiveSTS-VS2008`, and that's what I'll be focusing on here.

Start by opening the `MySecurityTokenService.cs` file and note how the sample derives a class from `SecurityTokenService` and overrides the two methods I discussed earlier. In `GetScope`, you'll see a signing certificate for the service being specified as part of the scope. This is the certificate of the issuer, and its private key will be used to sign any security tokens issued by this STS. You'll also notice that a certificate is being specified to use for encrypting the token so the relying party can read it. This is the certificate of the relying party, and its public key will be used to encrypt the tokens issued by this STS to the relying party.

This is one reason why this example is called "Simple". The STS is hardcoded to issue tokens for one relying party only: the web service in the `ClaimsAwareWebService-VS2008` project. A real STS would need to support multiple relying parties to be useful, and thus would need to keep a list of certificates and provide some way of managing that list.

Now have a look at the `GetOutputClaimsIdentity` override. The body of this method pulls out the subject's `IClaimsIdentity` that resulted from the STS authenticating the user (using Windows integrated authentication, which is the default client authentication method in WCF). You can see that it creates a new `IClaimsIdentity` and simply copies the name claim it gets by authenticating the user with Windows

integrated authentication, then adds one new claim, one that represents the user's age. Note the ClaimType (<http://GenevaSamples/2008/05/AgeClaim>) used here is a URI defined by the issuer; this is a simple example of creating a custom claim that allows you to communicate arbitrary identity details to relying parties, and as long as the issuer and relying party agree on what the claim means, it's all good!

In practice, it would make more sense to look up the user's date of birth in a user store, but this example is focused on how you use Geneva Framework to issue claims. If you build your own issuer using an example like this, where you get the claims is entirely up to you.

To see how the issuer exposes its WS-Trust endpoint to the world, open SimpleActiveSTS.cs and have a look at the Main() method. You'll notice down towards the bottom of the code that ServiceHost is not used; instead, the sample uses a derived host that Geneva Framework supplies called WSTrustServiceHost. This class simplifies configuring WCF to expose a WS-Trust endpoint. Note that in configuring the host, you specify the security token service configuration instance and the WS-Trust endpoint you want to expose, although you don't have to actually implement it yourself. The security token service configuration has the property SecurityTokenService which is set to the type of the custom class that you derived from SecurityTokenService (MySecurityTokenService in this example).

When you run this sample, if you watch closely, you'll see that the client first makes a call to the STS, (it prints out the claims it's issuing to its console window). Then, a moment later, you'll see that the relying party receives the claims and prints them out into its console window.

## WS-Federation (for Passive Clients) Example

The example I'll use for this discussion is from the Geneva Framework samples collection: **Samples\End-to-end Scenario\Federation For Web Apps**. For this discussion let us focus on the FPSTS project. Unlike the active example, there is no client project because there is no smart client; this is a browser-based example.

When you are building a passive STS for browsers, the way you expose your STS to the world is by handling HTTP requests from a browser, and in this example, a web page called Default.aspx is used. The ASPX page itself isn't very interesting, but if you look at the code behind file, Default.aspx.cs, you'll see some boilerplate code that processes WS-Federation requests. This particular example handles the request in Page\_PreRender, and if you look inside that method, you'll see the request being read and deserialized into a WSFederationMessage object, which is a helper class provided by Geneva Framework. Note that you don't actually need to use an ASPX page to process the request – if you know how to build an ASP.NET HttpHandler, it would work just as well.

The ProcessSignInRequest helper method is the most interesting section of code in this example. It creates an instance of custom security token service (which looks exactly like it did in the Active example) and calls its Issue method (Figure 21), passing in the caller's identity and the details of the request. Note the use of WSFederationSerializer, which supplies the impedance mismatch between the active and passive scenarios. It is used to deserialize the incoming request into a RequestSecurityToken that can be consumed by the STS, and also to serialize the resulting RequestSecurityTokenResponse

from the STS into a response message that includes JavaScript to auto-post the response back to the relying party.

```
public virtual RequestSecurityTokenResponse Issue(IClaimsPrincipal principal,
        RequestSecurityToken request)
```

Figure 23: SecurityTokenService.Issue

## Delegation and ActAs

When you build a multi-tier system, typically with a web front end and a collection of web services and other resources on the back end, you have a tough choice to make. Should the web front end use its own identity to access those back end resources, or should *delegate* the user's credentials to make those requests? If you choose the first option, you end up with what is known as a *trusted subsystem model*. If you choose the second, you'll need some way to delegate the client's identity to the back end. There are performance and security tradeoffs for both of these choices, with the trusted subsystem model generally favoring performance over security and the delegation model generally favoring security over performance. Sometimes it makes sense to have a mixture of these models where high volume, low value transactions are handled using the trusted subsystem model, but low-volume, high value transactions requiring the original caller's credentials. Regardless of how you design your system, it's important to note that the claims-based model of identity and the implementation in Geneva Framework supports delegation of credentials.

The web front end, once it receives a client's token, can make calls using its own identity to back end claims-aware web services. No special code is required to do this. But if the web front end wishes to delegate the client's credentials, it needs to retrieve the bootstrap token and send it along with its request for a security token for the back end web service.

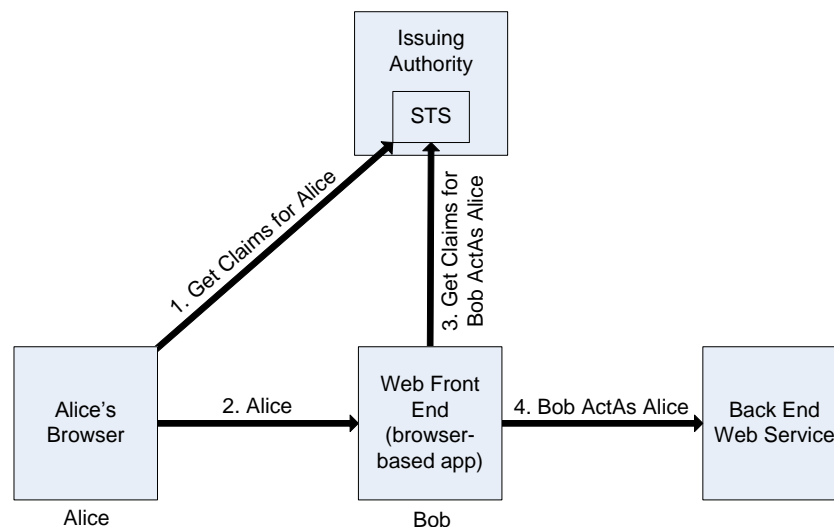


Figure 24: ActAs Scenario



Figure 22 shows a typical ActAs scenario. Alice has pointed her browser at a web application that, as part of its implementation, makes use of a back end web service. Alice's browser goes through the passive redirection handshake just like normal in order to present a security token to the web front end. This is where things get interesting: the web front end which, for the sake of this discussion, runs under an identity called Bob, takes Alice's token and submits it as an "ActAs" parameter in his request to get a security token for the back end web service. The issuing authority notes that Bob wants to make requests to the back end using Alice's credentials, and so crafts an IClaimsIdentity for Alice and an IClaimsIdentity for Bob, and links them together via the Delegate property, as shown in Figure 23. These identities are serialized into a security token for the back end, where Geneva Framework rehydrates this same structure so that the back end can see that this is Alice making the request (but technically, Bob is delegating her credentials). The back end can then perform appropriate access control, typically granting access based on Alice's level of permission. The back end can also audit the request, typically noting the fact that Bob delegated Alice's credentials to make the request. This is richer than the current model of delegation in Kerberos on the Windows platform today, where the back end has no programmatic way to discover that Alice's credentials were delegated by some middle tier component.

In the claims-based model, the back end can see that Alice went to the web front end (Bob) and that Bob delegated her credentials to get to the back end. If the back end were to receive a token for Alice without Bob as a delegate, it would know that Alice was accessing the back end directly, and could take appropriate action (deny the request, perhaps).

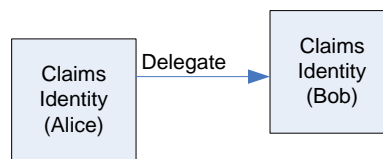


Figure 25: Geneva Framework Programming Model for Delegation

Consider the information the authority gets in this scenario. The authority knows which target relying party is the target of the request (the back end web service). It knows who is making the request (Bob) and knows that Bob wants to act on Alice's behalf. The authority may decide not to issue a security token in this case if Alice is a sensitive user such as an administrator with very high privilege. Or it may issue a token with a restricted set of claims to limit what Bob can do while using Alice's credentials. Or it may issue an entirely different set of claims based on what the back end needs. The authority might decide to deny direct requests from Alice to talk to the back end, if that is desirable. The only limitation is the policy supported by the STS that you buy. Of course, if you implement your own STS, you'll only be limited by your imagination.

The Geneva Framework samples collection includes an example of ActAs (**Samples\End-to-end Scenario\Identity Delegation**), and I'd like to point out some of the more interesting bits. I'll start with a look at the web front end (the WFE project), which uses ActAs to delegate the client's identity to a back end web service. In `global.asax.cs`, you'll see an example of handling the `ConfigurationLoaded` event fired by `SessionAuthenticationModule` to wire up a channel factory with custom binding configured to `Service2`.

In this example, the web front end handles the passive redirect to the STS and makes use of bootstrap tokens collection preserved by Geneva Framework; you can see this code in `default.aspx.cs` in the `Page_Load` method. Note how the Geneva Framework is exposing the bootstrap tokens collection through session security token's property. This capability removes the burden of caching bootstrap tokens in your application.

When the web front end wishes to make a call to the back end web service (the Service2 project), it pulls the user's bootstrap token out of session security token and prepares a WCF channel for Service2. You can see this code in `Page_Load`. It then uses an Geneva Framework supplied helper class called `FederatedClientCredentials` to configure its `ChannelFactory` to Service2; then creates a channel using `CreateChannelActingAs` method to perform a web service request using two sets of credentials (one user acting as another). The user's security token is specified as the `ActAs` credentials, and the channel is used to make a call to Service2's `ComputeResponse` operation. On the relevant note it is helpful to point out that Geneva Framework offers APIs to easily add the capability of sending `WSTrust` messages from a WCF client to `WSTrust`-based token services; `WSTrustClient` class empowers you to add this capability.

However, before the channel can make the request to Service2, it needs to get an appropriate token from the STS that Service2 trusts. If you look in the `STSBackend` project, you'll find the guts of this STS, and you'll see how this request gets processed. In `MySecurityTokenService.cs`, have a look at the `GetOutputClaimsIdentity` override. Note how the `request` argument includes an `ActAs` property that allows the issuer to see the `IClaimsIdentity` of the original client (via the `Subject` property). This sample issuer is very simple. It creates a new `ClaimsIdentity` by copying the `IClaimsIdentity` of the `ActAs` user (this is the identity from the original user's security token that was passed via `ActAs`), appending a second identity for the caller (the web front end) to the end of the delegate chain.

Note that this is a very simple example that demonstrates how `ActAs` works. In the real world, the issuer would likely perform at least checks similar to the checks that domain controllers in Windows deal with Kerberos delegation. Typical policy queries might include the following:

- Is the caller trusted to delegate credentials?
- For which users?
- To which relying parties?

The resulting SAML token that is issued will include the entire chain of identities. In this case there will be only two: the original client, and the web front end. You can see this deserialized by Geneva Framework in the back end web service, Service2. Open `Service2.cs` and note that the code prints out the caller's claims, (which in this case will be the user sitting behind the browser). It also traverses the delegate chain via `IClaimsIdentity.Delegate`, printing out the claims for each subject that delegated the user's credentials.

## Authentication Assurance

Sometimes, in order to find a balance between security and usability, it's good to have different levels of authentication. For example, for high volume, low value transactions, you might allow the user to authenticate with a user name and password via a web form. And for convenience, you might create a session via a cookie so that the user doesn't have to log in every time she submits a request.

But for those infrequent, high value transactions, you may want stronger authentication. Maybe there's a particular web page that exposes a sensitive feature, and your security policy absolutely requires the user to prove her identity using multi-factor authentication (such as a smart card with a PIN) before granting access. What you need here is *authentication assurance* – when the user arrives at the sensitive web page, you need to have assurance from the authenticating authority that the user has indeed proved possession of her smart card, and knowledge of her PIN code.

The STS can inform the relying party of the type of authentication that it used by injecting a special authentication method claim. The sample I'm going to discuss uses this technique, and can be found in the Geneva Framework samples collection: **Samples\End-to-end Scenario\Authentication Assurance**.

There are two issuers in this example: AuthPassiveSTSWindows, and AuthPassiveSTSCert. The first uses Windows integrated authentication, and the second requires the client to present a certificate, which is a stronger but more cumbersome form of authentication. Each issuer adds an Authentication claim into the list of claims for the user, indicating the form of authentication used. You can see this in the GetOutputClaimsIdentity method found in the App\_Code\CustomSecurityTokenService.cs file for each of these projects.

The relying party in this example is a browser-based application (called AuthAssuranceRP) that exposes a low value page (LowValueResourcePage.aspx) and a high value page (HighValueResourcePage.aspx). The low value page simply checks to see if the user is authenticated, and if not, redirects to default.aspx, on which is an instance of the FederatedPassiveSignIn control. This control presents the user with a link she can click in order to initiate the WS-Federation passive redirect to AuthPassiveSTSWindows, which uses Windows authentication to authenticate the user quickly and without much hassle.

Regardless of whether the user is authenticated or not, when she visits HighValueResourcePage.aspx, the code checks not only whether the user is authenticated, but if she also has an authentication method claim with the expected claim value of "CertOrSmartCard" and is only issued by AuthPassiveSTSCert STS, which requires the user to authenticate with a certificate (or smart card, if you have that infrastructure). So instead of redirecting the user to default.aspx, the high value page redirects to a separate sign-in page specifically for high-assurance logins. This is easy to implement; if you look at HighAssuranceSignInPage.aspx, you'll see another instance of the FederatedPassiveSignIn control that redirects to the AuthPassiveSTSCert STS instead.

In order to handle the scenario where the user logs in once using Windows authentication, and then logs in a second time using a cert, one would imagine that the sample has a bit of plumbing to do. But Geneva Framework handles this scenario seamlessly; the FAM simply refreshes its session security token

with the latest claims received. This means that when the user initially logged in with Windows authentication, the FAM issues a session security token with the claims issued in that case; when the user performs steps up authentication with a certificate, the FAM sees that the user is already authenticated and a new token has been issued, so it refreshes the session security token with the new collection of claims. If an application wants the second set of claims to be appended to the initial set it needs to preserve the initial set of claims and then redirect the user to the second issuer and use a custom claims authentication manager to append the new claims to the initial claims. In essence Geneva Framework offers more flexibility so that you can achieve these advanced scenarios in your claims-aware applications.

## Overview of Geneva Claims to Windows Token Service (GTS)

When you introduce claims-based model together with federated authentication into your existing multi-tiered system, typically with a web front end and a group of web services and other resources on the back end, it is likely that you would follow an incremental approach in migrating the back end services to claims-based model. For an interim migration period you would need to create the compatibility between services that are claims-aware and federated, and services that are not claims-aware. Typically, these claims-unaware services require Windows identity to authenticate the requests. On a related note, the back end services might depend on other software products that require a Windows identity and don't recognize the security tokens issued by an STS. For example, imagine a back end service that communicates with a SQL-based data store that requires a Windows identity for authentication.

In summary, we need a solution that acts as a bridge between the services that are federated and services that are not federated; a solution that provides a way to get a Windows identity from the security tokens issued by an STS. The claims-aware services can then use that solution to get the Windows identity of the user from the STS issued security tokens and then calls into the back end services that are not claims-aware. Geneva Framework offers such a solution: the "Geneva Claims To Windows Token Service" or in short form "GTS".

GTS is a Windows service that offers APIs to get a Windows identity by passing in the value of UPN claim. The most helpful API is `UpnLogon`, which takes a UPN claim value as a string and returns a Windows identity for that user. There are some additional steps to make this service work in the cases where you want to use the Windows identity returned from this service to access resources in remote servers (in other words, flowing identity off of the box); in that case, you would need to configure your Active Directory to allow constrained delegation. Also note that GTS needs the callers identities explicitly listed in its configuration file, `wtshost.exe.config`, which is located in the Geneva Framework install folder, in the "allowedCallers" property; it doesn't accept requests from all authenticated users in the system unless it is configured to do so.

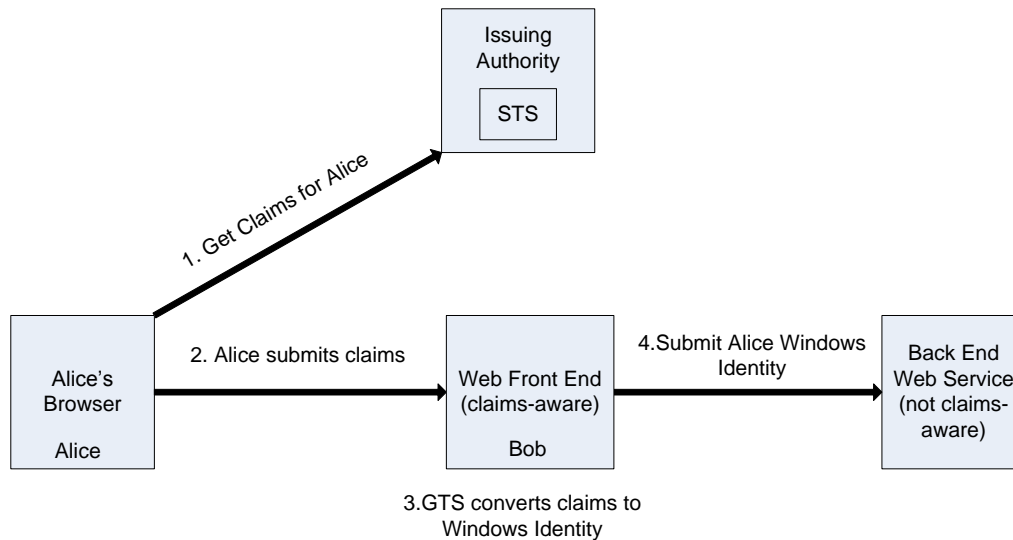


Figure 264: Geneva Claims To NT Token Service (GTS)

Figure 24 illustrates a typical GTS usage scenario. Alice has pointed her browser to a web application, which is a claims-aware application and, as part of its implementation, makes use of a back end service that is not claims aware. The normal passive redirect handshake happens in order to get a security token from STS and present it to the web front end. Now the interesting part starts: the web front end extracts claims from the security token, and notes that the content that Alice trying to access needs to be obtained from the back end web service, which is not a claims-aware service and requires the Windows identity of Alice. For the sake of this discussion, let's assume that web front end runs under an identity called Bob and is configured as the only allowed caller to the GTS. The web front end extracts the UPN claim value from the token and makes a request to GTS. Since Bob is configured in the allowed callers list, the GTS processes the request and returns the Windows identity of Alice. The web front end impersonates Alice and then calls to the back end service and gets access to the resources. The important point to note here is that the back end doesn't have to know anything about the claims model for this scenario to work.

It is possible that in some scenarios the web front end might request resources from the back end service more frequently and it would be more efficient to always have access to the Windows identity of Alice instead of calling to the GTS for each individual request. To address this scenario, Geneva Framework introduces a 'mapToWindows' configuration setting that can be set in the <Microsoft.IdentityModel> section of the application's configuration file. When the mapToWindows property is set to 'true', Geneva Framework always creates an instance of WindowsClaimsIdentity, which includes both the claims identity and Windows identity aspects of Alice, instead of a ClaimsIdentity upon successful token authentication. This means that the Windows identity of Alice is always available to the web front end.

The samples collection in Geneva Framework has an illustration (Samples\Extensibility\Convert Claims to NT Token) on how to configure the GTS and call its UPNLogon interface. Refer to the sample's readme file to get the background information about the projects included in the sample and to run the sample.

## Provisioning STSes in an relying party application using FedUtil

At the core of a federated solution lies the trust establishment between the relying party applications and STSes. Applications need a way to understand the characteristics of STSes before they can choose to accept claims issued by them. This is where federation metadata specifications play a vital role in defining the common characteristics of the STSes so that any relying party application can use a common mechanism to retrieve those characteristics. The most interesting STS characteristics from relying party applications stand point is the list of claims offered by an STS and the STS token signing key. RP applications typically require specific claims for their application logic and must examine the list of claims offered by an STS before establishing trust with it. On the other hand STSes would need to understand the characteristics of applications before they can establish trust with them. Wouldn't it be very helpful to have a tool that is capable of reading the STS metadata and automatically configuring relying party applications to trust these STSes, as well as a tool that produces an application's metadata that can be consumed by an STS. Geneva Framework offers a tool called FedUtil, which stands for 'Federation Utility for Easy STS Provisioning', to address these needs. This tool comes handy when you want to configure your federated claims-aware application to trust an STS and to indicate the application's required claims. It also helps in publishing a metadata about that application so that the STS can obtain the relevant information.

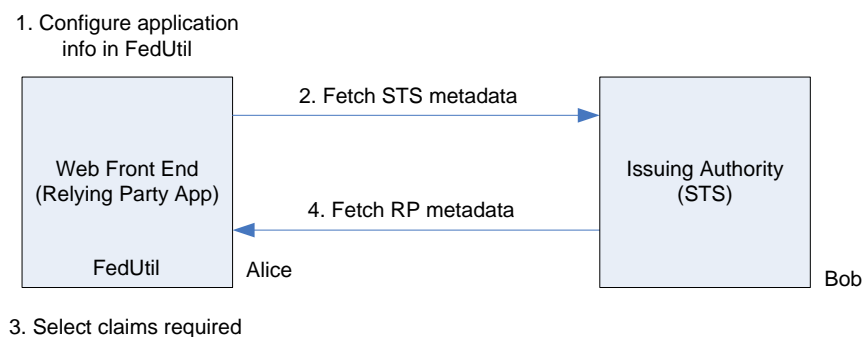


Figure 25: FedUtil tool

Figure 25 illustrates a typical flow involved in provisioning an STS to an application. Assume a scenario where FedUtil is installed in the web front end and an STS needs to be provisioned to the relying party, a typical ASP.NET application. Alice is the site administrator of the web front end and Bob is the STS administrator. Alice and Bob have agreed to provision the STS to the application and vice versa; Bob shares the metadata endpoint of the STS with Alice (through phone or email or other means) and asks for the application's metadata URL from Alice.

Alice runs FedUtil and specifies the location of the configuration file of the application (its web.config file) and the certificate to be used for encrypting purposes (1). FedUtil configures the application with the necessary Geneva Framework settings and enables the FAM and SessionAuthenticationModule. Alice then proceeds and specifies the metadata URL of the STS (2) that Bob provided earlier; FedUtil fetches the STS characteristics and shows the list of claims offered. Alice selects the claims that are required by the application (3). FedUtil creates a metadata document, which is an XML file, and places it

in the application folder itself. These three steps comprise the provisioning of STS in an application. Alice notifies Bob of the metadata URL of the application; Bob follows the STS administrator's guide and runs the STS trust establishment wizard, specifies the application's metadata URL and makes sure that the STS is configured to emit the claims that Alice's application requires. This lets the STS establish the trust with RP. This concludes the trust establishment on both the RP and STS sides.

Note that this is a very simple example that demonstrates how FedUtil works. In the real world, the issuer would require more than just metadata, such as out-of-band business agreements and policy settings. Typical policy queries might include the following:

- Is the relying party trusted to delegate credentials?
- Does the relying party specify user accounts?

## Summary

As an application developer, by building claims-aware web applications and services, you'll spend less time worrying about where to find identity attributes for users and have more time to focus on building a great application that solves real business problems. By relying on claims, you'll be able to personalize your applications more effectively, and implement important security features such as authorization and auditing, without baking one particular authentication method into your application, or writing queries against a corporate directory. By centralizing identity management in this fashion, the IT professionals can build the most efficient possible queries against their directories and give your application the identity details that it needs about users. And becoming claims-aware means you'll be in much better shape when you're asked to implement single sign-on and perhaps even identity federation.

Geneva Framework is a framework for building claims-aware web applications and services, and even issuing authorities, should you need to roll your own. Or better yet, get a pre-built authority like the Geneva Server.

Federated claims-based identity is the wave of the future. Get on board with Geneva Framework today!