Microsoft

Programming Windows® Identity Foundation

Vittorio Bertocci

Sample Chapters

To learn more about this book visit http://go.microsoft.com/fwlink/?Linkid=196688.

Table of Contents

Chapter 1

Claims-Based Identity

In this chapter:

Microsoft Windows Identity Foundation (WIF) enables you to apply the principles of claims-based identity when securing your Microsoft .NET application. Claims-based identity is so important that I want to make sure you understand it well before I formally introduce Windows Identity Foundation.

Claims-based identity is a natural way of dealing with identity and access control. However, the old ways of doing this are well established, so before delving into the new approach, it's useful to describe and challenge the classic assumptions about authentication and authorization. Once you have a clear understanding of some of the issues with traditional approaches, I'll introduce the basic principles of claims-based identity—I'll say enough to enable you to proficiently use Windows Identity Foundation for the most common scenarios. This chapter contains some simplifications that will get you going without overloading you with information. For a more thorough coverage of the subject, refer to Part II, "Windows Identity Foundation for Identity Developers."

Finally, we'll take our initial look at how WIF implements the mechanisms of claims-based identity and how you, the developer, can access the main elements exposed by its object model.

After reading this chapter, you'll be able to describe how claims-based identity works and how to take advantage of it in solutions to common problems. Furthermore, you'll be able to define Windows Identity Foundation and recognize its main elements.

What Is Claims-Based Identity?

Note If you already know about claims, feel free to skip ahead to the "WIF Programming Model" section. If you are in a big hurry, I offer you the following summary of this section before you skip to the next section: Claims-based identity allows you to outsource identity and access management to external entities.

The problem of recognizing people and granting access rights to them is one of the oldest in the history of computer science, and it has its roots in identity and access problems we all experience every day as we go through our lives.

Although we can classify almost all the solutions to the problem in relatively few categories, an incredible number of solutions tailored specifically to solve this or that problem exists. From the innumerable ways of handling user names and passwords to the most exotic hardware-based cryptography solutions, the panorama of identity and access methods creates a sequence of systems that are almost never compatible, each with different advantages, disadvantages, tradeoffs, and so on.

From the developer perspective, this status quo is bad news: this diversity forces you to continually relearn how to do the same thing with different APIs, exposes you to details of the security mechanisms that you'd rather not be responsible for, and subjects you to software that is brittle and difficult to maintain.

What you need is a way to secure your applications without having to work directly at the security mechanism level: an abstraction layer, which would allow you to express your security requirements (the "what") without getting caught in the specifics of how to make that happen (the "how"). If your specialty is designing user experiences for Microsoft ASP.NET, you should be allowed to focus your effort on that aspect of the solution and not be forced to become an expert in security (beyond the basic, secure-coding best practices, of course—all developers need to know those).

If you need a good reference on secure coding best practices, I highly recommend Writing Secure Code, Second Edition*, by Michael Howard and David LeBlanc (Microsoft Press, 2002).*

What we collectively call "claims-based identity" provides that layer of abstraction and helps you avoid the shortcomings of traditional solutions. Claims-based identity makes it possible to have technologies such as Windows Identity Foundation, which enables you to secure systems without being required to understand the fine details of the security mechanisms involved.

Traditional Approaches to Authentication

Before we go any further, let me be absolutely clear on a key point: this book does not suggest that traditional approaches to authentication and authorization are not secure or somehow bad *per se*. In fact, they usually do very well in solving the problem they have been designed to tackle. The issues arise when you have to deal with changes or you need different systems to work together. Because a single system can't solve all problems, you are often forced to re-perform the same task with different APIs to accommodate even small changes in your requirements.

It's beyond the scope of this book to give an exhaustive list of authentication systems and their characteristics; fortunately, that won't be necessary for making our point. In this section I'll briefly examine the built-in mechanisms offered by the .NET Framework and provide some examples of how they might not always offer a complete solution.

IPrincipal and *IIdentity*

Managing identity and access requires you to acquire information about the current user so that you can make informed decisions about the user's identity claims and what actions by the user should be allowed or denied.

In a .NET application the user in the current context is represented by an *IIdentity*, a simple interface that provides basic information about the user and how the user was authenticated:

```
public interface IIdentity
{
    // Properties
    string AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string Name { get; }
}
```

IIdentity lives inside *IPrincipal*, another interface that contains more information about the user (such as whether he belongs to a certain security group) that can be used in authorization decisions:

```
public interface IPrinicipal
{
    // Methods
    bool IsInRole(string role);
    // Properties
    IIdentity Identity { get; }
}
```

You can always reach the current *IPrincipal* in the code of your .NET application: in ASP.NET, you will find it in *HttpContext.Current.User*, and in general, you'll find it in *Thread.CurrentPrincipal*.

IPrincipal and *IIdentity*, as they exist out of the box, do provide some good decoupling from how the authentication actually happened. They do not force you to deal with the details of how the system came to know how the information about the user was acquired. If your users are allowed to perform a certain action only if they are administrators, you can write *Thread.CurrentPrincipal.IsInRole("Administrators")* without having to change your code according to the authentication method. The framework uses different extensions of *IPrincipal*—*WindowsPrincipal*, *GenericPrincipal*, or your own custom class—to accommodate the specific mechanism, and you can always cast from *IPrincipal* to one of those

classes if you need to access the extra functionalities they provide. However, in general, using *IPrincipal* directly makes your code more resilient to changes.

Unfortunately, the preceding discussion is just a tiny part of what you need to know about .NET security if you want to implement a real system.

Populating *IPrincipal*

Most of the information you need to know about the user is in *IPrincipal*, but how do you get that information in there? The values in *IPrincipal* are the result of a successful authentication: before being able to take advantage of the approach, you have to worry about making the authentication step happen. That is where things might start getting confusing if you don't want to invest a lot in security know-how.

When I joined Microsoft in 2001, my background was mainly in scientific visualization and with Silicon Graphics; I knew nothing about Microsoft technologies. One of the first projects I worked on was a line-of-business application for a customer's intranet. Today I can say I've had my fair share of experience with .NET and authentication, but I can still recall the confusion I experienced back then. Let's take a look at some concrete examples of using *IPrincipal*.

Up until the release of Microsoft Visual Studio 2008, if you created a Web site from the template, the default authentication mode was Windows. That means that the application expects Internet Information Services (IIS) to take care of authenticating the user. However, if you inspect the *IPrincipal* in such an application you will find it largely empty. This is because the Web application has anonymous authentication enabled in IIS by default, so no attempt to authenticate the user is made. This is the first breach in the abstraction: you have to leave your development environment, go to the IIS console, disable anonymous authentication, and explicitly enable Windows authentication. (You could do this directly by modifying the *web.config* file of the application in Microsoft Visual Studio, but going through IIS is still the most common approach in my experience.)

After you adjust the IIS authentication types, you're good to go, at least as long as you remain within the boundaries of the intranet. If you are developing on your domain-joined laptop and you decide to burn some midnight oil at home working on your application, don't be surprised if your calls to *IsInRole* now fail. Without the network infrastructure readily available, the names of the groups to which the user belongs cannot be resolved. As you can imagine, the same thing happens if the application is moved to a hoster, to the cloud, or in general away from your company's network environment.

In fact, you'll encounter precious few cases in which you enjoy the luxury of having authentication taken care of by the infrastructure. If the users you want to authenticate live outside of your directory, you are normally forced to take the matter into your own hands and use authentication APIs. That usually means configuring your ASP.NET application to use

Forms authentication, perhaps creating and populating a users and roles store according to the schema imposed by *sqlMembershipProvider*, implementing your own *MembershipProvider* if your scenario cannot fit what is available out of the box, and so on.

There's more: not everything can be solved by providing a custom user store. Often, your users are already provisioned in an existing store but that store is not under your direct control. (Think about employees of business partners, suppliers, and customers.) Store duplication is sometimes an option, but it normally brings more problems than the ones it solves. ASP.NET provides mechanisms for extending Forms authentication to those cases, but they require you to learn even more security and, above all, they are not guaranteed to work with other platforms.

If you've dealt with security issues in the past, you can certainly relate to what I've just described. If you haven't, don't worry if you didn't understand everything in the last couple of paragraphs. You can still understand that you need to learn a lot to add authentication capabilities to your application, despite ASP.NET providing you with helper classes, tooling, and models. If you're not interested in becoming a security expert, you would probably rather spend your time and energy on something else.

Here's one last note before moving on. When using Forms authentication, you do need to write extra code for taking care of authentication, but in the end you can still use the *IPrincipal* abstraction. (The user's information is copied from a *FormsIdentity* object into a *GenericPrincipal*.) This might induce you to think that all you need is better tooling to handle authentication and that the abstraction is already the right one. You're on the right track, but this is not the case if you stick with the current idea of authentication. Imagine a case in which you want authentication to happen using radically different credentials, such as a client Secure Sockets Layer (SSL) certificate, but those credentials do not map to existing Windows users. In the traditional case, you have to directly inspect the request for the incoming X.509 certificate and learn new concepts (subject, thumbprint, and so on) to perform the same task you already know how to do with other APIs.

The problem here is not with how ASP.NET handles authentication: it is systemic, and you'd have the same issues with any other general-purpose technology. By the way, if you consider how to handle identity and access with Microsoft Windows Communication Foundation (WCF), you have to learn yet another model, one that is largely incompatible with what we have seen so far and with its own range of APIs and exceptions.

When you can rely on infrastructure, like in the Windows Authentication example, you do fine: most details are handled by Windows, and all that's left for you is deciding what to do with the user information. When you can't rely on the infrastructure, as in the generic case, you can observe a consistent issue across all cases: you are burdened with the responsibility of driving the mechanics of authentication, and that often means dealing with complex issues. As I've already stressed, the gamut of all authentication options is wide, diverse, and

constantly evolving. Tooling can help you only so far, and it is doomed to be obsolete as soon as a new authentication scheme emerges.

What should developers do? Are we doomed to operate in an infinite arms race between authentication systems and the APIs supporting them?

Decoupling Applications from the Mechanics of Identity and Access

Once upon a time, developers were forced to handle hardware components directly in their applications. If you wanted to print a line, you needed to know how to make that happen with the specific hardware of the printer model in use in the environment of your customer.

Those days are fortunately long gone. Today's software takes advantage of the available hardware via *device drivers*. A device driver is a program that acts as an intermediary between a given device and the software that wants to use it. All drivers have one *logical layer*, which exposes a generic representation of the device and the functionalities that are common to the device class and reveals no details about the specific hardware of a given device. The logical layer is the layer with which the higher level software interacts—for example, "print this string." The driver contains a *physical layer* too, which is tailored to the specific hardware of a given device. The physical layer takes care of translating the high-level commands from the logical layer to the hardware-specific instructions required by the exact device model being used—for example, "put this byte array in that register," "add the following delimiter," "push the following instructions in the stack," and so forth.

If you want to print from your .NET application, you just call some method on *PrintDocument*, which will eventually take advantage of the local drivers and make that happen for you. Who cares about which printer model will actually be available at run time?

Doesn't this scenario sound awfully familiar? Managing hardware directly from applications is similar to the problem of dealing with authentication and authorization from applications' code: there are too many (difficult!) details to handle, and results are too inflexible and vulnerable to changes. The hardware problem was solved by the introduction of device drivers; there is reason to believe that a similar approach can solve the access management problem, too.

Although an operating system provides an environment conducive to the creation of a thriving driver ecosystem, the identity and access problem space presents its own challenges—for example, authentication technologies and protocols belong to many different owners, the ways in which resources and services are accessed is constantly changing and is fragmented in many different segments, different uses imply dramatically different usability and security requirements, users and data are often sealed in inaccessible silos, and

so on. The chances of a level of indirection spontaneously emerging from that chaos are practically zero.

With the inflationary growth of distributed systems and online businesses, in the last few years the increasing need for interoperable protocols that could tear down the walls between silos became clear. The big players in the IT industry got together and agreed on a set of common protocols that would support interoperable communications across different platforms. Some examples of those protocols are SOAP, WS-Security, WS-Trust, WS-Federation, Security Assertion Markup Language (SAML), and in more recent times, OpenID, OAuth, and other open protocols. Don't worry if you don't recognize some or any of those names. What is important here is that the emergence of common protocols, combined with the extra attention that the security aspects commanded in their redaction, finally created the conditions for introducing the missing logical layer in identity and access management. It is that extra layer that will make it possible to isolate applications and their developers from the gory details of authentication and authorization mechanics. In this part, I am not going to go into the details of what those protocols are or how they work; instead, I will concentrate on the scenarios that they enable and how to take advantage of them.

Now that you've gained some perspective on why today's approaches are less than ideal, it is time to focus on how you can move beyond them.

Authentication and Authorization in Real Life

Imagining what should be in the logical layer of a printer driver is easy. After all, you have a good idea of what a printer is supposed to do and how you'd like to take advantage of it in your code. Now that you know it is possible to create a logical layer for identity, do you know what it should look like? Which kind of API should you offer to developers?

We have been handling low-level details for so long that it may be hard to see the bigger picture. A useful exercise is to step back and spend a moment analyzing how identity is actually used for authorization in the real world, and see if what you learn can be of help in designing your new identity layer. Let's look at an easy example.

Imagine you are going to a movie theater to see a documentary film. Consider the following facts:

1. The documentary contains scenes that are not suitable for a young and impressionable audience; therefore, the clerk at the box office asks you for a picture ID so that he can verify whether you are old enough to watch the film. You reach for your wallet and extract your driver's license, and in so doing you realize that it is expired.
2. Resigned to missing the first show, you walk to a nearby office of the Department of Licensing (DOL). At the DOL, you hand over your old driver's license and ask to get a new one.

3. The clerk takes a good look at you to see whether you look like the photo on record. Perhaps he asks you to read a few letters from an eye test chart. When he's satisfied that you are who you claim to be, he hands you your new driver's license.
4. You go back to the movie theater and present your new driver's license to the clerk. The clerk, now satisfied that you are old enough to watch the movie, issues you a ticket for the next show.

Figure 1-1 shows a diagram of the transaction just described.

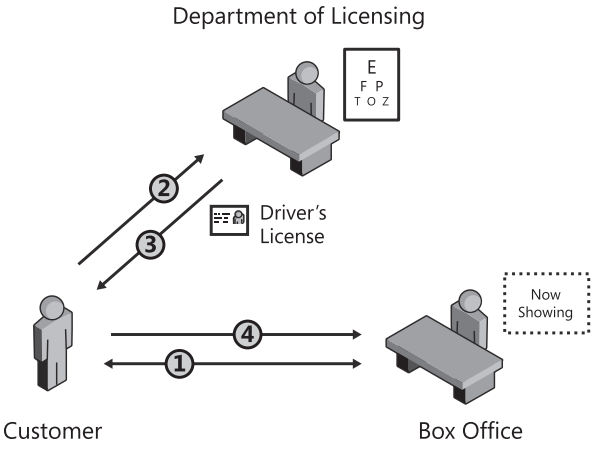

FIGURE 1-1 One identity transaction taking place in real life

This is certainly not rocket science. We go through similar interactions all the time, from when we board a plane to when we deal with our insurance companies. Yet, the story contains precious clues about how we can add our missing identity layer.

Let's consider things from the perspective of the box-office clerk. The clerk regulates access to the movie, actually authorizing (or blocking) viewers from acquiring a ticket. The question that the clerk needs to answer is, "Is this person older than X?" Here comes the interesting part: the box-office clerk does not verify your age *directly*. How could he? Instead, he relies on the verification that somebody else already did. In this case, the DOL certified your birth date in its driver's license document. The box-office clerk trusts the DOL to tell the truth about your age. The DOL is a recognized government institution, and it has a solid business need to know a person's correct age because it is relevant to that person's ability to drive. The outcome of the interaction would be different if you presented the box-office clerk a sticky note on which you scribbled your age. In such a transaction, you are not a trustworthy source. (Unless the clerk knows you personally, he must assume bias on your part—that is, you could lie in order to get into the movie theater.)

Note that in this scenario you presented a driver's license as proof of age, but from the clerk's point of view not much would have changed if you had used your passport or any other document *as long as the institution issuing it is known and trusted by the box office clerk*.

One last thought before drawing our parallel to software: the box-office clerk does not know which procedure the DOL clerk followed for issuing you a driver's license, how the DOL verified your identity, which things he verified, and how he verified them. He does not need to know these things because once he decides he trusts the DOL to certify age correctly, he'll believe in whatever birth date appears on a valid driver's license with the picture of the bearer.

Let's summarize our observations in this scenario:

- The box-office clerk does not verify the customer's age directly, but relies on a trusted party (the DOL) to do so and finds the result in a document (the driver's license).
- The box-office clerk is not tied to a particular document format or source. As long as the issuer is trusted and the format is recognized, the clerk will accept the document.
- The box-office clerk does not know or care about the details of how the customer has been identified by the document issuer.

This sounds quite efficient. In fact, similar transactions have been successfully taking place for the last few thousand years of civilization. It's high time that we learn how to take advantage of such transactions in our software solutions as well.

Claims-Based Identity: A Logical Layer for Identity

The transaction described in the preceding section, including the various roles that the actors played in it, can be generalized in one of the most universal patterns in identity and access and forms the basis of claims-based identity. The pattern does not impose any specific technology, although it does assume the presence of certain capabilities, and it contains all the indications you need for defining your logical identity layer.

Let's try to extract from the story a generic pattern describing a generic authentication and authorization system. Pay close attention for the next few paragraphs. Once you understand this pattern, it is yours forever. It will provide you with the key for dealing with most of the scenarios you encounter in implementing identity-based transactions.

Entities Figure 1-2 shows the main entities that play a role in most identity-based transactions.

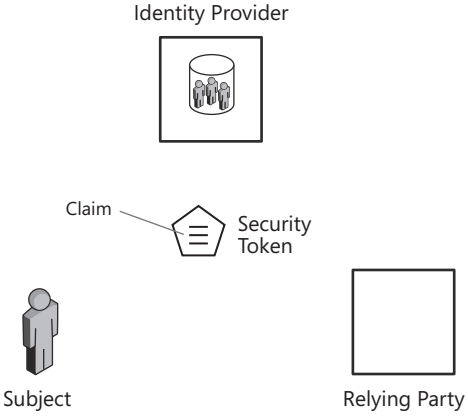

FIGURE 1-2 The main entities in claims-based identity

Let's say that our system includes a user, which in literature is often referred to as a *subject*, and the application the user wants to access. In our earlier example, the subject was the moviegoer; in the general case, a subject can be pretty much anything that needs to be identified, from an actual user to the application identities of unattended processes.

The application can be a Web site, a Web service, or in general any software that has a need to authenticate and authorize users. In identity jargon, it is called a *relying party*, often abbreviated as *RP*. In our earlier example, the RP is the combination of the box-office clerk and movie theater.

The system might include one or more *identity providers* (IPs). An IP is an entity that knows about subjects. It knows how to authenticate them, like the DOL in the example knew how to compare the customer's face to its picture archives; it knows facts about the customer, like the DOL knows about the birth date of every licensed driver in its region. An identity provider is an abstract role, but it requires concrete components: directories, user repositories, and authentication systems are all examples of parts often used by an identity provider to perform its function.

We assume that a subject has standard ways of authenticating with an IP and receiving in return the necessary user information (like the birth date in the example) for a specific identity transaction. We call that user information *claims*.

The magical word "claim" finally comes out. *A claim is a statement about a subject made by an entity*. The statement can be literally anything that can be associated with a subject, from attributes such as birth date to the fact that the subject belongs to a certain security group. A claim is distinct from a simple attribute by the fact that a claim is always associated with the entity that issued it. This is an important distinction: it provides you with a criterion for deciding if you want to believe that the assertion applies to the subject. Recall the example of the birth date printed on the driver's license versus a birth date scribbled on a sticky note: the clerk believes the former but not the latter because of the entities backing the assertion.

Claims travel across the nodes of distributed systems in *security tokens*, which are XML or binary fragments constructed according to some security standard. Tokens are digitally signed, which means that they cannot be tampered with and that they can always be traced back to the IP that issued them (which provides a nice mechanism for associating token content with its issuer, as required by the definition of claims).

Flow Claims are the currency of identity systems: they are what describe the subject in the current context, what the IP produces, and what the RP consumes. Here's how the transaction unfolds.

Well before your transaction starts, the RP publishes a document, often called a *policy*, in which it advertises its security requirements: things such as which security protocols the RP understands and similar information. This is analogous to the box office hanging up a sign that says, "Be ready to show your driver's license or your passport to the clerk." The most important part of the RP policy is the list of the identity providers it trusts. This is equivalent to another sign at the box office specifying, "Drivers' licenses from U.S. states only; passports from Schengen Treaty countries only."

Again, before the transaction starts, the IP publishes an analogous policy document that advertises its own security requirements. This document provides instructions on how to ask the IP to issue a security token. In literature, you will often find that IPs offer their token issuance services via a special flavor of Web services, called STS (Security Token Service). You'll read more (MUCH more) about STS throughout the book.

Figure 1-3 summarizes the steps of the canonical identity transaction.

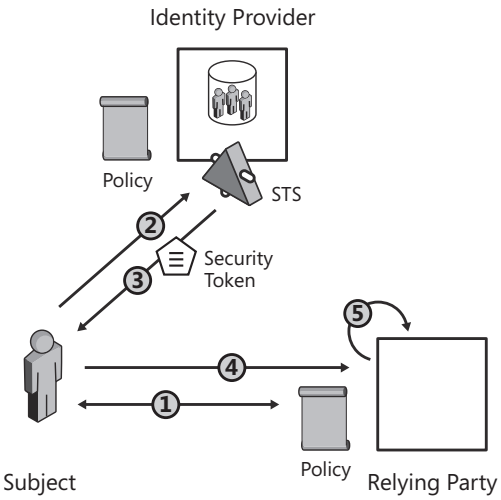

FIGURE 1-3 The flow of the canonical transaction in claims-based identity

Here's a description of that flow:

1. The subject wants to access the RP application. It does that via an agent of some sort (a browser, a rich client, and so on). The subject begins by reading the RP policy. In so doing, it learns which identity providers the RP trusts, which kind of claims are required, and which security protocols should be used.
2. The subject chooses one of the IPs that the RP trust and inspects its policy to find out which security protocol is required. Then it sends a request to the IP to issue a token that matches the RP requirements. This process is the equivalent of going to the DOL and asking for a document containing a birth date. In so doing, the subject is required to provide some credentials in order to be recognized by the IP. The details of the protocol used are described in the IP policy.
3. The IP processes the request; if it finds the request to be satisfactory, it retrieves the values of the requested claims, sending them back to the subject in the form of a security token.
4. The subject receives the security token from the IP and sends it together with his first request to the RP application.
5. The RP application examines the incoming token and verifies that it matches all the requirements (coming from one trusted IP, in the expected format, not having been tampered with, containing the right set of claims, and so on). If everything looks as expected, the RP grants access to the subject.

This sequence of steps could describe a user buying something online and presenting to the Web merchant a credit score from a financial institution; it could describe the user of a Windows Presentation Foundation (WPF) application accessing a Web service on the local intranet by presenting a group membership claim issued from the domain controller; it could describe pretty much any identity transaction if you assign the subject, RP, and IP roles in the right way.

The abstraction layer we were searching for The pattern we've been discussing describes a generic identity transaction. Without going into detail about the actual protocols and technologies involved, we can say that it just makes assumptions about what capabilities those technologies should have, such as the capability of exposing policies.

The model is profoundly different from what we have observed in classic approaches: whereas a traditional application takes care of authentication more or less directly, here the RP outsources it entirely to a third party, the identity provider. The details of how authentication happens are no longer a concern of the application developer; all you need to do is configure your application to redirect users to the intended identity providers and be able to process the security tokens they issue. Although you can use many different protocols for obtaining and using a security token, the abstract idea of claims and security tokens is

nonspecific enough to allow you to create a generic programming model for representing users and the outcome of authentication operations without exceptions.

Those changes in perspective finally eliminate the systemic flaw that prevented us from eradicating from the application code the explicit handling of identity without relying on demanding infrastructure. All that's left to do is for platform and developer tools providers to take advantage of the claims-based identity model in their products.

Note The model is extremely expressive. In fact, you can easily use it for representing traditional scenarios too. If the IP and the RP are the same entity, you are back to the case in which the application itself takes care of handling authentication. The important difference in the implementation is that both code and architecture will show that this is just a special case of a more generic scenario. Therefore, the decoupling will be respected and changes will be accommodated gracefully.

WIF Programming Model

Microsoft has been among the most enthusiastic promoters of the claims-based identity model. It should come as no surprise that it has also been one of the first to integrate it in its product offerings. For example, Active Directory Federation Services 2 (ADFS2) is a Windows Server role that, among other things, enables your Active Directory instance to act as an identity provider and issue claims for your user accounts.

Windows Identity Foundation (WIF) is a set of classes and tools, an extension to the .NET Framework, that enables you to use claims-based identity when developing ASP.NET or WCF applications. It is seamlessly integrated with the core .NET Framework classes and in Visual Studio so that you can keep using the tools and techniques you are familiar with for developing your applications, while reaping the advantages of the new model when it comes to identity.

In this section, I will introduce the basics of Windows Identity Foundation: how it exposes claims-based identity principles to developers, some fundamental considerations about its structure, and the essential programming surface every developer should be aware of.

An API for Claims-Based Identity

In the previous section, you learned about claims-based identity. If you had to expose it as a programming model so that an application developer could take advantage of it, what requirements would you follow? Here is my wish list:

- Make claims available to the developer in a clear, consistent, and protocol-independent fashion.
- Take care of all (or nearly all) authentication, authorization, and protocol handling *outside* of the code of the application, away from the eyes of the developer.
- Minimize the need to change the code when changes at deployment time occur. Drive as much of the application's behavior as possible via configuration.
- Provide a way to easily configure applications to rely on external identity providers for authentication.
- Provide a way for applications to easily advertise their requirements via policy.
- Organize everything in a pluggable architecture that can support multiple protocols and isolate the developer from the details of the deployment (on premises and cloud, ASP.NET and WCF, and so on).
- Respect as much as possible existing code and practices, maximizing the amount of old code that will still work in the new model while offering incremental advantages with the new APIs.

As you'll see time and time again throughout the book, WIF satisfies all these criteria.

WIF's Essential Behavior

Earlier in the text, I wrote that Part I of the book will show you how to take advantage of WIF in your applications without the need to become a security expert, and I intend to keep that promise. Here I'll start with a simplified description of how WIF works, covering the essential points for allowing you to use the product. Part I will be about ASP.NET applications, and I'll stick with discussing scenarios that can be tackled by using WIF tooling alone. I'll omit the details that have no immediate use. You can refer to Part II of the book if you want to know the whole story.

WIF allows you to externalize authentication and authorization by configuring your application to rely on an identity provider to perform some or all those functions for you. How does it do that in practice?

Figure 1-4 shows a simplified diagram of how WIF handles authentication in the ASP.NET case.

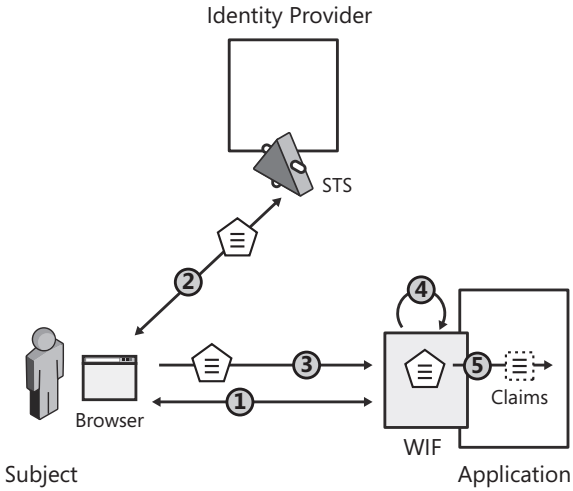

FIGURE 1-4 A simplified diagram of how Windows Identity Foundation takes care of handling authentication for an ASP.NET application

The idea is extremely simple and closely mimics the canonical claims-based identity pattern:

1. WIF sits in front of your application in the ASP.NET pipeline. When an unauthenticated user requests a page, it redirects the browser to the identity provider pages.
2. Here the IP authenticates the user in whatever way it chooses (perhaps by showing a page with user name and password, using Kerberos, or in some other way). Then it manufactures a token with the required claims and sends it back.
3. The browser posts the token it got from the IP to the application, where WIF again intercepts the request.
4. If the token satisfies the requirements of the application (that is, it comes from the right IP, contains the right claims, and so on), the user is considered authenticated. WIF then drops a cookie, and a session is established.
5. The claims in the incoming token are made available to the application code, and the control is passed to the application.

As long as the session cookie is valid, the subsequent requests won't need to go through the same flow because the user will be considered to be authenticated.

You are not supposed to know it yet, but the preceding flow unfolds according to the WS-Federation protocol specification: most of the magic is done by two HTTP modules: *WSFederationAuthenticationModule* (WSFAM) and *SessionAuthenticationModule*.

The whole trick of using WIF in your application boils down to the following tasks:

1. Configure the application so that the WIF HTTP modules sit in the ASP.NET pipeline in front of it.
2. Configure the WIF modules so that they refer to the intended IPs, use the right protocols, protect the planned resources of the application, and in general enforce all the desired application policies.
3. Access claim values from the application code whenever there is a need in the application logic to make a decision driven by user identity attributes.

The good news is that in many cases steps 1 and 2 can be performed via Visual Studio tooling. There is a handy wizard that walks you through the process of choosing an identity provider, offers you various options, and informs you about the kind of claims you can get about the user from the specific IP you are referring to. The wizard translates all the preferences you expressed via point and click in the *web.config* settings. The next time you press F5, your application will already apply the new authentication strategy. Congratulations, your application is now claims-aware.

The good news keep coming; performing step 3 is simple and perfectly in line with what .NET developers are already accustomed to doing when handling user attributes.

IClaimsIdentity and *IClaimsPrincipal*

Remember *IIdentity* and *IPrincipal* as a means of decoupling the application code from the authentication method? It worked pretty well until we found an authentication style (client certificates) that broke the model. Now that authentication is no longer a concern of the application, we can confidently revisit the approach and apply it for exposing new information (claims) by leveraging a familiar model.

WIF provides two extensions to *IIdentity* and *IPrincipal, IClaimsIdentity* and *IClaimsPrincipal*, respectively—which are used to make the claims processed in the WIF pipeline available to the application code. The instances live in the usual *HttpContext.Current.User* property in ASP.NET applications. You can use them as is with the usual *IIdentity* and *IPrincipal* programming model, or you can cast them to the correct interface and take advantage of the new functionalities.

Let's take a quick look at the members of the new interfaces. Note that the list for now is by no means exhaustive and highlights only properties that will be useful in basic scenarios.

IClaimsPrincipal is defined as follows:

```
public interface IClaimsPrincipal : IPrincipal
{
    // ...

    // Properties
    ClaimsIdentityCollection Identities { get; }
}
```

Because *IClaimsPrincipal* is an extension of *IPrincipal*, all the usual functionalities (such as *IsInRole*) are supported. As you'll see in Chapter 2, "Core ASP.NET Programming," this useful property extends to other ASP.NET features that take advantage of *IPrincipal* roles—for example, access conditions expressed via the <*authorization*> element still work.

The only noteworthy news is the *Identities* collection, which is in fact a list of *IClaimsIdentity*. Let's take a look at the definition of *IClaimsIdentity*:

```
public interface IClaimsIdentity : IIdentity
{
    // ...
    ClaimCollection Claims { get; }
 }
```

Here I stripped out most of the *IClaimsIdentity* members (because I'll have a chance to introduce them all as you proceed though the book), but I left in the most important one, the list of claims associated with the current user. What does a *Claim* look like?

```
public class Claim
{
    // ...
    // Properties
    public virtual string ClaimType { get; }
    public virtual string Issuer { get; }
    public virtual IClaimsIdentity Subject { get; }
    public virtual string Value { get; }
}
```

Once again, many members have been stripped out for the sake of clarity. The properties shown are self-explanatory:

- **ClaimType** Represents the type of the claim: birth date, role, and group membership are all good examples. WIF comes with a number of constants representing names of claim types in common use; however, you can easily define your own types if you need to. The typical claim type is represented with a URI.
- **Value** Specifies, as you can imagine, the value of the claim. It is always a string, although it can represent a value of a different CLR type. (Birth date is a good example.)

- **Issuer** Indicates the name of the IP that issued the current claim.
- **Subject** Points to the *IClaimsIdentity* to which the current *Claim* belongs, which is a representation of the identity of the subject to which the claim refers to.

If you understand what a claim is, and if you have any type of identity card in your wallet, the properties just described are intuitive and easy to use. Let's look at one easy example.

Suppose that you are working on one application that has been configured with WIF to use claims-based identity. Let's say that authentication takes place at the very beginning of the session, so that during the execution you can always assume the user is authenticated. At a certain point in your code, you need to send an e-mail notification to your user. Therefore, you need to retrieve her e-mail address. Here there's how you do it with WIF:

```
IClaimsIdentity identity = Thread.CurrentPrincipal.Identity as IClaimsIdentity;
string Email = (from c in identity.Claims
                where c.ClaimType == System.IdentityModel.Claims.ClaimTypes.Email
                select c.Value).SingleOrDefault();
```

The first line retrieves the current *IClaimsIdentity* from the current principal of the thread, exactly as it would if you wanted to work with the classic .NET *IIdentity*—the only difference is the downcast to *IClaimsPrincipal*.

The second line uses LINQ for retrieving the e-mail address from the current claim collection. The query is very intuitive: you search for all the claims whose type corresponds to the well-known *Email* claim type, and you return the value of the first occurrence you find. For the e-mail case, it is reasonable to expect that there will be only one occurrence in the collection, However, this is not true in the general case. Just think of how many group claims would be generated for any given Windows user; thus, the standard way of retrieving a claims value must take into account that there might be multiple claims of the same type in the current *IClaimsIdentity*.

Nothing in the code shown indicates which protocol or credential types have been used for authenticating the user. That means you are free to make any changes in the way in which users authenticate, without having to change anything in your code. Relying on one IP for handling user authentication and using open protocols delivers true separation of concerns; therefore, making those changes is also very easy.

Relying on claims for getting information about the user mitigates the need for maintaining attribute stores, where the data can become stale or be compromised. As you can observe, the code shown in this section does not contain any call to a local database that could be broken by routine changes or that could become a problem if the application is moved to an external host that cannot access local resources. In the age of the cloud, the importance of being able to move applications around cannot be overestimated.

Finally, the two lines of code shown earlier will work with any kind of .NET program, ASP.NET or WCF. The way in which WIF snaps to the two different hosting models and pipelines is different. I will describe how it does this in detail in Part II; however, from the perspective of the application developer, nothing changes. The tooling operates its magic for configuring the application to externalize authentication. All you need to know is how to mine the results with a consistent API without worrying about underlying protocols, hosting model, or location.

It would appear that adding one extra layer of indirection worked. We finally found an API that can secure your applications without forcing you to take care of the details.

Summary

Traditional approaches to adding identity and access management functionality to applications all have the same issues: they require the developer to take matters into his own hands, calling for specialized security knowledge, or they heavily rely on the features of the underlying infrastructure. This situation has led to a proliferation of APIs and techniques, forcing developers to continually re-learn how to perform the same task with different APIs. The resulting software is brittle, difficult to maintain, and resistant to change. In this chapter, I gave some concrete examples of how this systemic flaw in the approach to adding identity and access management affects development, even development in .NET.

Claims-based identity is an approach that changes the way we think about authentication and authorization, adding a logical representation of identity transactions and identifying the roles that every entity plays. By adding that further level of indirection, claims-based identity created the basis for the decoupling of the programming model and the details of deploy-time systems. In the chapter, I described the basics of claims-based identity and you learned how it can be used to model a wide variety of scenarios.

Windows Identity Foundation is one set of .NET classes and tools that helps developers to secure applications by following the principles of claims-based identity. This chapter introduced the essential programming surface exposed by WIF, and it demonstrated how WIF does not suffer from the issues I mentioned for traditional approaches.

In the next chapter, I will show how to take advantage of WIF for performing authentication, authorization and identity-driven customization in a variety of common Web scenarios.

Chapter 4

Advanced ASP.NET Programming

In this chapter:

Now that most technicalities are out of the way, we can focus on intended usage of the product for addressing a wider range of scenarios.

This chapter resumes the architectural considerations that drove Part I of the book, "Windows Identity Foundation for Everybody," by tackling more complex situations. I'll assume you are now familiar with the flow described in Chapter 3, "WIF Processing Pipeline in ASP.NET." I'll give you concrete indications about how to customize the default behavior of Windows Identity Foundation (WIF) to obtain the desired effect for every given scenario.

Using claims-based identity in your application is, for the most part, the art of choosing who to outsource authentication to and providing just the right amount of information for influencing the process. This chapter will not exhaust all the possible ways you can customize WIF—far from it. However, it will equip you with the principles you need to confidently explore new scenarios on your own.

The first section, "More About Externalizing Authentication," takes a deeper look at the entities to which you can outsource authentication for your application. I'll go beyond the simplifications offered so far, introducing the idea of multiple provider types. A lot of the discussion will be at the architectural level, helping you with the design choices in your solutions. However, hardcore coders should not fear! The section also dives deep into the Security Token Service (STS) project template that comes with the WIF SDK. Although in real scenarios you'll rarely need to create a custom STS, given that more often than not you'll rely on off-the-shelf products such as Active Directory Federation Services 2.0 (ADFS 2.0), you'll find it useful to see a concrete example of how the architectural considerations mentioned are reflected in code.

The "Single Sign-on, Single Sign-out, and Sessions" section explores techniques that reduce the need for users to explicitly enter their credentials when visiting affiliated Web sites and

shows how to clean up multiple sessions at once. One specific case, sessions with sliding validity, is the occasion for a deeper look at how WIF handles sessions.

The "Federation" section dissects the pattern that is most widely used for handling access across multiple organizations. I'll cover more in depth the use of STSes for processing claims, and we'll tackle the problem of deciding who should authenticate the user when there are many identity providers (IPs) to choose from (something known as the *home realm discovery problem*). The solutions to those problems can be easily generalized to any situation in which the relying party (RP)—which was discussed in Chapter 3—needs to communicate options to the IP. I'll demonstrate that with another example: the explicit request for a certain authentication level.

The "Claims Processing at the RP" section closes the chapter by describing how to use Windows Identity Foundation for preprocessing the claims received from the identity provider. I'll briefly revisit the claims-based authorization flow—introduced in minimal terms in Chapter 2, "Core ASP.NET programming." Then I'll show you how to filter and enrich the *IClaimsPrincipal* before the application code gains access to it.

After you read this chapter, you'll be able to make informed decisions about the identity management architecture of your solutions. You'll know what it takes to implement such decisions in ASP.NET. You'll have concrete experience using the WIF extensibility model for solving a range of classic identity management scenarios. That experience will help you to devise your own WIF-based solutions. Once again, I'll give you practical code indications about the ASP.NET case, but the general principles introduced here can be applied more broadly, often to the WCF services case and even on non-Microsoft platforms.

More About Externalizing Authentication

Until now, I have described situations in which the application relies on only one external entity—what I defined as the *identity provider*, or IP. Although this is an accurate representation of a particular common scenario, the general case can be a bit more complicated. Not only might you have to accept identities from multiple identity providers, identity providers are not the only entities you can outsource authentication to!

So far, the role played by the entity within a transaction (the identity provider) has been conflated with the instrument used to perform the function (the STS). The purpose of this section is to help you better understand the separation between the two by providing more details about the nature of the identity provider, introducing a new role known as the *federation provider*, and studying how those high-level functions reflect on the implementation of the associated STS.

Identity Providers

Being an identity provider is a role, a job if you will. You know from Chapter 1, "Claims-Based Identity," that an IP "knows about subjects." In fact, all the thinking behind the idea of IP is just good service orientation applied to identity.

The standard example of a concrete identity provider is one built on top of a directory, just as ADFS 2.0 is built on top of Active Directory. In this scenario, there's an entity that is capable of authenticating users and making assertions about them, and all you are doing is making that capability reusable to a wider audience by slapping a standard façade (the STS) in front of it. The use of standards when exposing the STS is simply a way of maximizing the audience and increasing reusability. Here's an example: Although a SharePoint instance on an intranet can take advantage of Active Directory authentication capabilities directly via Kerberos, that is not the case for a SharePoint instance living outside the corporate boundaries and hosted by a different company. Exposing the authentication capabilities of Active Directory via ADFS 2.0 makes it possible to reuse identities with the SharePoint instance in the second scenario, removing the platform and location constraints. WIF is just machinery that enables your application to take advantage of the same mechanism. It is worthwhile to point out that SharePoint 2010 is, in fact, based on WIF.

Another advantage of wrapping the actual authentication behind a standard interface is that you are now isolated from its implementation details. The IP could be a façade for a directory, a membership provider–based site, or an entirely custom solution on an arbitrary platform; as long as its STS exposes the authentication functionality through standards, applications can use it without ties or dependencies outside of the established contract. Who cares if the connection string to the membership database changes, or even if there is a membership database in the first place? All you need to know is the address of the STS metadata.

Those characteristics of the IP role tell you quite a lot about what to expect regarding the structure of the STS exposed by one IP.

Note In literature, you'll often find that one STS used by one IP can be defined as an "IP-STS." In a short, you'll see how this can sometimes be useful for disambiguating the function the STS offers.

In the WS-Federation Sign-in flow, described in Chapter 3, you saw that the details of how the STS authenticates the request for security tokens is a private matter between the STS and the user. Now you know that such a system has to be something that allows the STS to look up user information from some store—so that it can be extracted and packaged in the form of claims. Notable examples are the ones in which the STS leverages the same authentication methods of the resource it is wrapping. If the IP is a façade for Active Directory and the user

is on the intranet, the STS might very well be hosted on one ASPX page that is configured in Internet Information Services (IIS) to leverage Windows native authentication. If the source is a membership database, the STS site will be protected via a membership provider, and so on. The claim value's retrieval logic in the STS will use whatever moniker the authentication scheme offers for looking up claim values, but the authentication will often be performed by the infrastructure hosting the STS rather than the STS code itself.

Nothing prevents one IP from exposing more than one STS endpoint to accommodate multiple consumption models. For example, the same IP might be listening for Kerberos authenticated requests from the intranet and X.509 secured calls on an endpoint available on the Internet; the IP might expose further endpoints, both for browser-based requestors via WS-Federation and SAMLP or for active requestors via WS-Trust; and so on. This process offers another insight into how one IP is structured: authentication and claims issuance logic should communicate but remain separate so that multiple STS endpoints scenarios are handled with little or no duplication. As you'll see later in the section, the WIF STS programming model is consistent with that consideration.

An IP will actively manage the list of the RPs it is willing to issue a token for. This is not only a matter of ensuring that claims are transmitted exclusively to intended recipients, but also a practical necessity. Especially in the passive case, in which token requests are usually simple, the IP decides what list of claims will be included in a token according to the RP the token is being issued for. ("Passive case" is mainly another way to say that you use a browser. You'll know everything about it after reading Chapter 5, "WIF and WCF.") Such a list is established when the RP is provisioned in the IP's allow list. Just like WIF enables one application to establish a trust relationship with an IP by consuming its metadata via the Federation Utility Wizard, IP software such as ADFS 2.0 includes wizards that can consume the application metadata and automatically provision the RP entry in its allow list.

Note In computer science as in other disciplines, an allow list is a list of entities that are approved to do something or to be recipients of some action. For example, if your company network has an allow list of Web sites, that means you can browse only on those sites and no other. Conversely, having a blacklist of Web sites means that you can browse everywhere but on those. An IP normally maintains an allow list of RPs it is willing to issue a token for: any request for a recipient not in the allow list is refused. The ADFS 2.0 UI describes that as *Relying Party Trust*. I am not very fond of that use of "trust," which in this context has a special meaning (believing that the claims issued by a given IP about a subject are true), but your mileage may vary.

The IP also keeps track of the certificate associated with the RP, both for ensuring that the RP has a strong endpoint identity (exposed via HTTPS) and for encrypting the token with the correct key if confidentiality is required.

Nonauditing STS

There are situations, especially in the area of e-government, in which the user would like to keep private the identity of the RP he is using. For example, a citizen might want to use a token issued by a government IP proving his age, but at the same time he would like to maintain his privacy about what kind of sites (for example, liquor merchants) he is using the token for.

Technically, the scenario is possible, although setting up such functionality would introduce some limitations. For example, not knowing the identity of the RP, the IP would not know the associated X.509 certificate and that would make it impossible to encrypt the issued token. Also, some protocols handle the scenario better than others. Although the WS-Federation specification allows for specifying which claims should be included in the requested token, most implementations expect the list of claims required by one RP to be established a priori, which is of course of no help if the identity of the RP is not known. Things can be a little easier with WS-Trust, as you'll see in the next chapter.

In the business world, the most common scenario requires the IP to have a preexisting relationship with the RP before issuing tokens for it; therefore, off-the-shelf products such as ADFS 2.0 normally mandate it.

The scenario described so far—one application outsourcing authentication to one identity provider—is common, and none of the further details about IPs I gave here invalidate it. However, sometimes the planets do not align the way you'd like, and for some reason simple direct outsourcing to one IP does not solve the problem.

Federation Providers

Let's consider for a moment the matter of handling multiple identity providers. Imagine being a developer for a financial institution. Let's say you are writing a corporate banking application, which allows companies to handle the salary payment process for their workforce. This is clearly one case in which you need to trust multiple identity providers—namely, all the companies who access your financial institution for managing payments.

From what you have seen so far, you know only one way of handling the situation: adding multiple *FederatedPassiveSignIn* controls to your application entry page, each of them pointing to a different identity provider. Although the approach works, it can hardly be called a full externalization of identity management because provisioning and deprovisioning identity providers forces you to change the application code. Things get worse when you have one entire portfolio of applications to make available to a list of multiple identity providers—having to reapply the trick mentioned previously for every application rapidly

becomes unsustainable as the number of apps and IPs goes up. This clearly indicates the need to factor out IP relationship management from the application responsibilities.

Another common issue you might encounter has to do with the ability of your application to understand claims as issued by one identity provider. Here is why:

- Sometimes you might have simple format issues. For example, the users you are interested in might come from another country and their IP might use claim URIs containing locale-specific terms your application does not understand. (An English application might need to know the name of the current user and expect it in an *http://claims/name* format, while an Italian IP might send the desired information in the *http://claims/nome* claim format.)
- Sometimes the information will need some processing before being fed to your application. For example, an IP might offer a birth date claim, but your application might be forbidden from receiving personally identifiable information (PII). All you require here is a simple Boolean value indicating if the user is below or above a certain threshold age. Although the information is clearly available to the IP, it might not be offered as a claim.
- Finally, you might need to integrate the claims received from the IP with further information that the IP does not know. For example, you might be an online book shop accepting users from a partner IP. The IP can provide you with name and shipping address claims, but it cannot provide you with the last 10 books the user bought from your store. That is data that belongs to you, and you have the responsibility of making it available in the form of claims if you want to offer to your developers a consistent way of consuming identity information.

What is needed here is a means of doing some preprocessing—some kind of intermediary that can massage the claims and make them more digestible for the application.

The standard solution to these issues is the introduction of a new role in identity transactions, which goes by the name of Federation Provider (FP).

A Federation Provider is a claims transformer; it is an entity that accepts tokens in input—kind of like an RP does—and issues tokens that are (usually) the result of some kind of processing of the input claims. An FP offers its token manipulation capabilities exactly like an IP, by exposing STS endpoints. The main difference is that, whereas one IP usually expects requests for security tokens secured by user credentials that will be used for looking up claims, the FP expects requests to be secured with an issued token that will be used as input for the claims transformation process. In the IP case, the issued token contains the claims describing the authenticated user; in the FP case, the issued token is the result of the processing applied to the token received in the request. Given the fact that an FP exposes one STS, applications can use it for externalizing authentication in exactly the same way as you have seen they do with IPs. WIF's Federation Utility Wizard does not distinguish between IPs and FPs—all it needs is an STS and its metadata.

The reason that it's known as the *Federation* Provider is that enabling federation is the primary purpose that led to the emergence of this role. In a nutshell, here's how that works. Imagine company A is a manufacturer that has a number of line-of-business (LOB) applications for its own employees, including applications for supply management, inventory, and other usual stuff. Company B is a retailer that sells the products manufactured by A. To improve the efficiency of their collaboration, A and B decide to enter into a federation agreement: certain B employees will have access to certain A applications. Instead of having every A application add the B identity provider and having the B IP provision every application as a recognized RP, A exposes a Federation Provider.

The B IP will provision the A FP just like any other RP, associating to the relationship the list of claims that B decides to share with A about its users. All of the A applications that need to be accessible will enter into a trust relationship with the A FP, outsourcing their authentication management to its STS. Figure 4-1 shows the trust relationships and the sign-in flow.

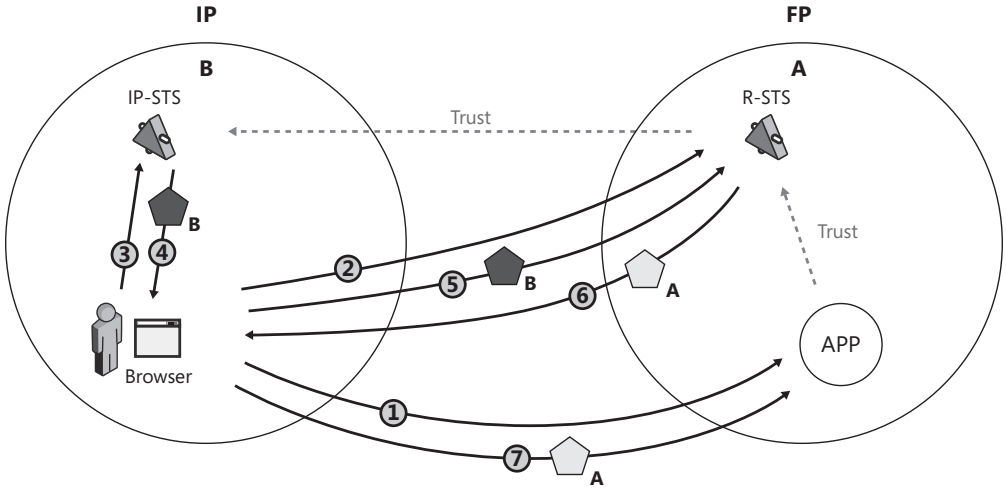

FIGURE 4-1 The authentication flow in a federation relationship between two organizations

The flow goes as follows:

1. One employee of B navigates to one application in A.
2. The user is not authenticated because the application will accept only users presenting tokens issued by the A FP. The application redirects the user to the A FP.
3. Again, the user is not authenticated. The A FP will accept only users presenting tokens issued by the B IP. The application redirects the user to the B IP.
4. The user lands on the B IP, where authentication will take place according to the modes decided by B. The user gets a token from the B IP.
5. The user gets back to the A FP and presents the token from the B IP.

6. The A FP processes the token according to the application's needs—some claims might be reissued verbatim as they were received from B; others might be somehow processed; still others might be produced and added anew. The A FP packages the results of the processing in the form of claims and issues the new token to the user.
7. The user gets back to the application and presents the token from the A FP; the application authenticates the call by examining the token from A FP.

The main advantage of using an FP in a federation scenario is obvious: you now have a single place where you can manage your relationship, defining its terms (such as which claims you should receive). The applications are decoupled from those details. Because the FP knows about both the incoming claims (because it is on point for handling the relationships) and the claims needed by the application (because it is part of the organization, it knows about which claim types are available and their semantics), applications can effectively trust it to handle authentication on their behalf even if the actual user credentials verification takes place elsewhere. The process can be iterated. For example, you can have an FP trusting another FP, which in turn trusts an IP, although that does not happen too often in practice.

The WIF STS Template

Outsourcing authentication to one external STS makes life much easier for the application developer, at the price of relinquishing control of a key system function to the STS itself. Although relinquishing control of the mechanics of authentication is sweet, as I've been pointing out through the entire book, the STS you choose better be good, or else. Here's what I mean by "good" in this case:

- **An STS must be secure** A compromised STS is an absolute catastrophe because it can abuse your application's trust by misrepresenting the user privileges.
- **An STS must be available** If the STS endpoint is down, as a consequence of peak traffic or any other reason, your application is unreachable: no token, no party.
- **An STS must be high-performing** Every time a user begins a session with your application, the STS comes into play. Bad performance is extremely visible, can become a source of frustration for users, and even pile up to compromise the system's availability.
- **An STS must be manageable** If you own the STS, whether it used as an IP or FP, you'll need to manage many aspects of its activities and life cycle, such as the logic used for retrieving claim values, provisioning of recognized RPs, establishment of trust relationships with the IP of federated partners, management of signing and encryption keys, auditing of the issuing activities, and management of multiple endpoints for different credential types and protocols. The list goes on and on.

In other words, running an STS is serious business: don't let anybody convince you otherwise. An endpoint that understands WS-Federation, WS-Trust, or SAMLP requests and can issue a token accordingly technically fits the definition of "STS," but protocol capabilities alone can't help with any of the requirements just mentioned.

This is why in the vast majority of real-world scenarios it is wise to rely on off-the-shelf STS products, such as ADFS 2.0. Those products host STS endpoints and advanced management features that simplify both small and large maintenance operations that running an IP or an FP (or both) entails. Let's take ADFS 2.0 as an example: ADFS 2.0 is a true Windows server role—tried, stressed, and tested just like any other Windows server feature.

The Windows Identity Foundation SDK makes the generation of an STS deceivingly simple by offering Microsoft Visual Studio templates for both ASP.NET Web sites and WCF services projects that implement a bare-bones STS endpoint (for WS-Federation and WS-Trust, respectively). The Generate New STS option in the Add STS Reference Wizard just instantiates one of those templates in the current solution. Those test STSes are an incredibly useful tool for testing applications, thanks to the near absence of infrastructure requirements (ADFS 2.0 requires a working Active Directory instance, SQL Server, Windows Server 2008 R2, and so on) and instantaneous creation. As somebody who had to write STSes from scratch with WCF in the past (a long and messy business), I am delighted by how easy it is to generate a test STS with WIF. For the same reason, such test STSes are consistently used in WIF samples and courseware. This book is no exception.

Why do I say "deceivingly simple"? Because of all the requirements I listed earlier. WIF can certainly be used to build an enterprise-class STS—it has been used for building ADFS 2.0 itself. However, between the STS template offered by the WIF SDK and ADFS 2.0, there are many, many man-years of design, enormous amounts of development and testing, tons of assumptions and default choices, brutal fuzzing, relentless stressing, and so on. The fact that the STS template gives you back a token does not mean it can be used as is in a real-life system. People regularly underestimate the effort required for building a viable STS, an error of judgment that can result in serious issues. That is why I always discourage the creation of custom STSes unless it's absolutely necessary, and there's not a lot of detailed guidance on that.

Now that I've got the disclaimer out of the way: this chapter will use a lot of custom STSes. Taking a peek inside an STS is a powerful educational tool that can help you understand scenarios end to end. Being able to put together test STSes can help you simulate complex setups before committing resources to them. Finally, you'll likely encounter situations in which setting up a custom STS is the way to go—for example, if your user credentials are not stored in Active Directory. The guidance here is absolutely not enough for handling the task—that would involve teaching how to build secure, scalable, manageable, and performing services, which is well beyond the scope of this text—but it can be a starting point for understanding the token issuance model offered by WIF.

The rest of the section describes the STS template for ASP.NET offered by WIF SDK 4.0. As you read through this section, I suggest you go back to the simple example you created in Chapter 2 and put breakpoints on the parts of the STS project being discussed. Every time something is not too clear, try a test run in the debugger to get a better sense of what's going on.

Structure of the STS ASP.NET Project Template

The ASP.NET Security Token Service Web Site template, as WIF SDK 4.0 names it, can be found in the C# Web sites templates list in Visual Studio. As mentioned, this is also the template that is used by the Add STS Reference Wizard for generating an STS project within an existing solution. Figure 4-2 shows the list of templates installed by the WIF SDK 4.0.

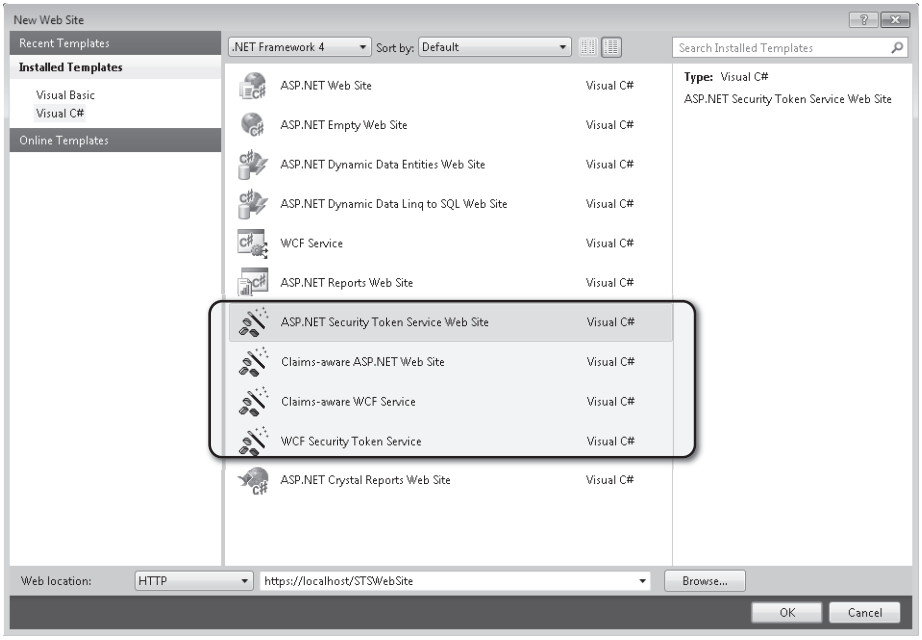

FIGURE 4-2 The templates installed by WIF SDK 4.0, with the template used for creating an ASP.NET STS highlighted

The STS Web site is typically created on the local IIS. Although it is possible to use the plain HTTP binding, in general the STS Web site will be created on an HTTPS endpoint.

Note Using HTTP in this case is normally a really bad idea. Even if you encrypt the tokens you issue, and even if the RP can take steps for mitigating the risk of accepting stolen tokens, the reality is that using plain HTTP on browser-based scenarios makes you vulnerable to man-in-the-middle and other attacks. In Chapter 5, you'll have a chance to dig deeper into the topic.

IIS vs. Visual Studio Built-in Web Server

Visual Studio allows you to develop Web sites without requiring the presence of IIS on your development machine. Visual Studio offers a built-in Web server, called the ASP.NET Development Server, which can be used to render pages directly from the file system.

Although you can get WIF to work on Web sites running on the ASP.NET Development Server, there are limitations (for example, the built-in Web server does not support HTTPS) and complications (for example, the dynamically assigned ports change the site URIs and thus force changes in the configuration). Because of this, it's just simpler to use IIS.

Similar considerations led me to use Web site projects rather than Web application ones. Web application development starts on the file system and requires extra steps for hosting (and debugging) the application in IIS. Furthermore, at the time of this writing, Fedutil.exe is not a big friend of the dynamic ports system featured by ASP.NET Development Server. The Add STS Reference Wizard will not always work as expected when launched on a Web application project.

Figure 4-3 shows the structure of the STS project.

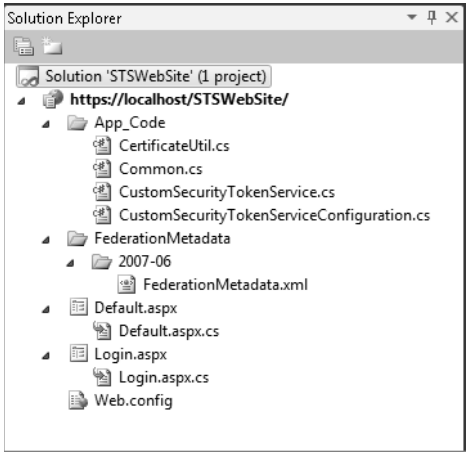

FIGURE 4-3 The ASP.NET STS project structure

That is the structure of a minimal Web site protected via Forms authentication, containing the classic *Login.aspx* and *Default.aspx* pages. The *web.config* file is minimal, containing practically nothing specific to WIF apart from the reference to its assembly and a few values in the <*appSettings*>. The Web site is configured to use Forms Authentication. As you saw in the first example in Chapter 2, *Login.aspx* does not actually verify any credentials and represents

just a pro-forma authentication page: the page will just create the authentication cookie and start a session regardless of the credentials entered in the UI.

The hands-on lab Web Sites and Identity (C:\IdentityTrainingKit2010\Labs\WebSitesAndIdentity\Source\Ex1-ClaimEnableASPNET) exercise 2, shows how to use an existing Membership store for authenticating calls to the STS, and how to source claim values from a Role provider.

All this emphasizes what I mentioned earlier about the separation between the STS functions and the authentication mechanism: here Forms authentication is the method of choice, but it is independent from what WIF does for implementing the token-issuing functionality. The authentication system could be easily substituted with Windows integrated authentication or whatever else, as long as it takes care of authenticating the user before giving access to *Default.aspx*.

Note An obvious observation is that the STS template generates an IP-STS, something that authenticates users and issues tokens describing them. It is not hard to transform it into an R-STS: you can just run the Add STS Reference Wizard on the STS project itself, and that will be enough for excluding the current Forms authentication settings and externalize authentication to the second STS of your choosing. However, that would change only the way authentication is handled, not the way claims are generated: an R-STS transforms incoming claims, but the default template implementation does not do that. At the end of the section, I'll discuss what you need to change for modifying the claim issuance criteria as well.

The *Default.aspx* page represents the STS endpoint, and it takes care of instantiating and executing the token-issuing logic in the context of an ASP.NET request. The page itself does not contain much. What we are interested in is the *Page_PreRender* handler in *Default.aspx.cs*:

```
public partial class _Default : Page
{  /// <summary>
  /// Performs WS-Federation Passive Protocol processing.
  /// </summary>
  protected void Page_PreRender( object sender, EventArgs e )
  {
    string action = Request.QueryString[WSFederationConstants.Parameters.Action];

    try
    {
      if ( action == WSFederationConstants.Actions.SignIn )
      {
        // Process signin request.
        SignInRequestMessage requestMessage =
          (SignInRequestMessage)WSFederationMessage.CreateFromUri( Request.Url );
        if ( User != null && User.Identity != null && User.Identity.IsAuthenticated )
        {
```

```
            SecurityTokenService sts =
            new CustomSecurityTokenService( CustomSecurityTokenServiceConfiguration.Current );
            SignInResponseMessage responseMessage =
               FederatedPassiveSecurityTokenServiceOperations.ProcessSignInRequest
                (requestMessage, User, sts );
             FederatedPassiveSecurityTokenServiceOperations.ProcessSignInResponse
               (responseMessage, Response );
          }
          else
          {
            throw new UnauthorizedAccessException();
          }
        }
        else if ( action == WSFederationConstants.Actions.SignOut )
        {
          // Ignore the rest for now
          // ...

    }
}
```

This code is the STS counterpart of the WS-Federation processing logic that WIF provides for RPs, as studied in Chapter 3. Whereas the RP generates the request for a security token and validates it, the STS listens to those requests and issues tokens according to the WS-Federation protocol. Here's a quick explanation of what the method does:

- The handler inspects the request *QueryString* for the WS-Federation action parameter, *wa*. Let's focus on the case in which *wa* is present and has the value *wsignin1.0*, which indicates a request for a token. (We'll explore the sign-out case later in the chapter.)
- The code creates a new *SignInRequestMessage* from the request—that is, a name-value collection that surfaces the various WS-Federation parameters as properties.
- Do you have a non-empty *IPrincipal*? Is the current user authenticated? If it isn't, an *UnauthorizedAccessException* is thrown and the user is redirected to the login page. If it is, the following must take place:
  - Get an instance of *SecurityTokenService* by retrieving an instance of a subclass, *CustomSecurityTokenService*. This class contains the core STS logic, as you'll see in a moment.
  - The new STS instance, along with the incoming *SignInRequestMessage* and the user's *IPrincipal*, is fed to *FederatedPassiveSecurityTokenServiceOperations.ProcessSignInRequest*, where it will be used for issuing the token and producing a suitable *SignInResponseMessage*.
  - Finally, *FederatedPassiveSecurityTokenServiceOperations.ProcessSignInResponse* writes the *SignInResponseMessage* in the response stream, which will be eventually forwarded to the RP and processed as you saw in Chapter 3.

There are a lot of classes with long names, but in the end the code shown earlier just feeds the authenticated user and the request to a custom *SecurityTokenService* class and sends back the result. The STS project features an *App_Code* folder, which contains all the classes the STS needs, including the *CustomSecurityTokenService* class; all you need to do is take a look at what happens there.

The Redirect Exception in the STS Template in Visual Studio 2010

At the time of this writing, the ASP.NET STS template exhibits a small issue with Visual Studio 2010. At the end of the *Page_PreRender* method, there is a catch clause that handles generic *Exception*s and re-throws them after having added a message. Unfortunately, the code described earlier contains at least a redirect, which throws an exception. Normally, you would not see it, but the re-throw makes Visual Studio stop at the unhandled exception. There are various workarounds for this issue. You could catch *ThreadAbortException* and ignore it. You could just press F5 again, and the application will move forward without issues. You could comment that line in the template. You could start without debugging. I do not suggest disabling the Visual Studio default behavior of stopping at unhandled exceptions unless you know very well what you are doing.

STS Classes and Methods in *App_Code*

The *Common.cs* file is not very interesting; it's just a bunch of constants. *CertificateUtil.cs* is not that remarkable either; it's a helper class for retrieving X.509 certificates from the Windows stores, although there is an interesting piece of trivia for it. WIF uses that code, instead of the classic *X509Certificate2Collection.Find* because the latter does not call *Reset* on the certificates it opened.

CustomSecurityTokenServiceConfiguration, as the name implies, takes care of storing some key configuration settings for the STS: the name, the certificate that should be used for signing tokens, serializers for the various protocols, and so on. The most important setting it stores is the type of the custom *SecurityTokenService* itself.

Finally, we get to the very heart of the STS: the class in *CustomSecurityToken.cs*. The code generated by the template has the purpose of doing the bare minimum for obtaining a working STS; hence, I won't analyze it too closely here, except for pointing out some notable behavior. Rather, I'll use it as a base for telling you about the more general model that you have to follow when developing a custom STS in WIF. Note that the considerations about *SecurityTokenService* apply both to ASP.NET and WCF STSes.

SecurityTokenService In WIF, a custom STS is always a subclass of *SecurityTokenService*, and the ASP.NET template is no exception. The claims-issuance process is represented by a series

of *SecurityTokenService* methods, which are invoked following a precise syntax that leads the form request validation to emit the token bits. Complete coverage of that sequence is beyond the scope of this book; however, here I'll list the main methods you should know about:

- ***ValidateRequest*** This method takes in a *RequestSecurityToken* and verifies that it is in a request that can be handled by the current implementation. For example, it checks that the required token type is known. *SecurityTokenService* provides an implementation of *ValidateRequest*. You should override it only if you are adding or subtracting from the default STS capabilities. There are also few things taking place in *GetScope* that could perhaps be done in *ValidateRequest*. I'll point those out as we encounter them.
- ***GetScope*** *GetScope* is an abstract method in *SecurityTokenService* that must be overridden in any concrete implementation. It takes as input the *IClaimsPrincipal* of the caller and the current *RequestSecurityToken*.

  The purpose of *GetScope* is to validate and establish some key parameters that will influence the token-issuance process. Those parameters are saved in one instance of *Scope*, which is returned by *GetScope* and will cascade through all the subsequent methods in the token-issuance sequence. Here are the main questions that *GetScope* answers:

  - **Which certificate should be used for signing the issued token?** Although a signing certificate has already been identified in the configuration class, *GetScope* should confirm that certificate (as done by the template implementation) or override it with custom criteria—for example, if something in the request influences which certificate should be used.
  - **Is the intended token destination a recognized RP?** As discussed earlier, normally an STS issues tokens only to the RP URIs that have been explicitly provisioned. If the incoming *wtrealm* (available in *RequestSecurityToken* via the property *AppliesTo*) does not correspond to a known RP, an *InvalidRequestException* should be thrown.

Note The template implementation of *GetScope* performs the check against a hard-coded list. One could argue that a validation check would belong to the *ValidateRequest* method, but the item about encryption that follows shows how *GetScope* would need to query an RP settings database anyway.

  If the *AppliesTo* value is valid, it is fed into the *Scope* object. It will be needed for the *AudienceRestriction* element of the issued token, which in turn will be validated by WIF against the <*audienceURI*> config element on the RP.

  - **Should the issued token be encrypted?** If yes, with which certificate? The STS configuration should specify whether the token should be encrypted. If it should

be, the same store that was used for establishing whether the RP was valid should also carry information about which encryption certificate should be used. The template uses a value from config.

- **To which address should the token be returned?** The template assumes that *wtrealm*—that is, the *AppliesTo* value—is both the identifier of the RP and its network-addressable URI. As a result, *GetScope* assigns the value of *AppliesTo* to the *ReplyToAddress* property of the *Scope* object.

Important Although in many cases it is true that *AppliesTo* contains the network addressable endpoint of one RP, that does not always hold. Sometimes *wtrealm* will be a logical identifier for the application rather than a network address, and the actual address to which the token should be returned will be different. A way of handling this is by sending the actual address in the request via the *wreply* parameter, and then assigning it to *Scope.ReplyToAddress* (from *RequestSecurityToken.ReplyTo*). *ReplyTo* addresses should always be thoroughly validated because supporting *wreply* opens your STS up to redirect attacks.

Note ADFS 2.0 does not handle *wreply*.

When the *Scope* is ready, a number of lower level token-issuance preparation steps take place. You can influence those if you want to, but I won't go into further details here. After those steps are completed, it is finally time to work with claims.

- ***GetOutputClaimsIdentity*** This method takes as input the *IClaimsPrincipal* of the caller, the *RequestSecurityToken*, and the *Scope*. It returns an *IClaimsIdentity*, which contains the claims that should be issued in the token for the caller. Note that at this point the *IClaimsPrincipal* of the caller is a representation of the *IPrincipal* obtained from the STS caller via Forms authentication. This should not be confused with the output *IClaimsPrincipal* created by the STS, which will be available at the RP after successful sign-in.

  This is perhaps the least realistic of the implementations in the STS template. It returns two hard-coded claims, Name and Role, regardless of the targeted RP or the caller (the only concession being the value of the Name claim, extracted from the incoming principal):

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

```
    ClaimsIdentity outputIdentity = new ClaimsIdentity();

    // Issue custom claims.
    // TODO: Change the claims below to issue custom claims required by your
application.
    // Update the application's configuration file too to reflect new claims
requirement.

    outputIdentity.Claims.Add( new Claim( System.IdentityModel.Claims.ClaimTypes.Name,
principal.Identity.Name ) );
    outputIdentity.Claims.Add( new Claim( ClaimTypes.Role, "Manager" ) );

    return outputIdentity;
  }
```

In a more realistic setting, your *GetOutputClaimsIdentity* implementation would need to make some decisions about the outgoing *IClaimsIdentity*. These are the questions it will need to answer:

- **Given the current request, which claim types should be included?** The list of claims that should be issued is often established per RP, at provisioning time. That is especially common for WS-Federation scenarios, and some products will go as far as implementing that tactic for the WS-Trust case as well.

Note ADFS 2.0 uses that approach in every case. The list of claims to issue is always established on the basis of the RP for which the token is being issued.

  Chances are that the list of claims to use will be available in the same store you used in *GetScope* for retrieving the RP URI and encryption certificate.

  WS-Trust (and WS-Federation, via *wreq* or *wreqptr* parameters) supports requesting a specific list of claims for every request. Although that requires more work, which probably includes checking on an RP-bound list if the required claims are allowed for that given RP, there are many advantages to the approach. Apart from minimal disclosure and privacy considerations, possibly a bit out of scope here, one obvious advantage is that this can help keep the token size under control. A token representing a Windows identity can have *many* group claims. If for a given transaction the group claim is not required, being able to exclude it can dramatically shrink the resulting token.

  If you want to support requests that specify the required claims, you'll find that list in the *RequestSecurityToken.Claims* collection.

- **Given the current principal, which claim values should be assigned?** Together with the request authentication method, this is the question that determines whether your STS is an IP-STS or an R-STS.

One IP-STS uses some claims of the incoming *IClaimsPrincipal* for looking up the caller in one or more attribute stores, from where the STS will retrieve the values to assign to the established claim types. That's the direct descendent of using a user name for looking up attributes in a profile store; in fact, it can take place in exactly the same way if you have a user name claim. Of course, you are not limited to it—you can use any claim you like.

One R-STS processes the claims in the incoming *IClaimsPrincipal* in arbitrary ways, storing the results in other claims in the outgoing *IClaimsIdentity*. Note that the STS can also just copy some claims from the incoming token to the outgoing one without modification, and it can even add new claims in the same way the IP-STS does. I'll show some examples of this later, during the federation and home-realm discovery discussions.

ADFS 2.0 offers a management UI, where administrators can specify how to source or transform claims. The mappings can be specified via a simple UI or via a SQL-like language that is especially well suited for claims issuance. In your own STS, you can embed the corresponding code directly in *GetOutputClaimsIdentity*, or you can develop a mechanism for driving its behavior from outside.

Metadata

You know about metadata from Chapter 3. If you need to change something in the metadata document of one RP, you can simply edit it. Perhaps that's not the greatest fun you'll have, but it is feasible.

Doing the same for one STS is out of the question because an STS metadata document must always be signed. The WIF SDK has one example showing how to use the WIF API for generating a metadata document programmatically. It's not rocket science, just a lot of serialization. Generating the document has the advantage of keeping it automatically updated if you play your cards well and read things from the config. It also has another advantage of granting you better control of complicated situations, such as cases in which on the same Web site you expose both WS-Federation and WS-Trust endpoints.

Any dynamic content generation mechanism will do. My favorite is exposing a WCF service and hiding the .svc extension with some IIS URL rewriting.

Single Sign-on, Single Sign-out, and Sessions

In this section, I'll formalize some of the session-related concepts I've been hinting at so far. Namely, I'll help you explore how WIF can reduce the number of times a user is prompted for credentials when browsing Web sites that are somehow related to each other. I'll show you how you can sign out a user from multiple Web sites at once, making sure no dangling

sessions are still open. Finally, I'll share a few tricks you can use for tweaking the way in which WIF handles sessions.

Single Sign-on

In Chapter 3, I illustrated the dance that WS-Federation prescribes for signing in a relying party and how the WIF object model implements that. Let's move the scenario a little further by supposing that you want to model the case in which the user visits more than one RP application.

If the RPs have absolutely nothing in common, there is not much to be said: every RP session will have its own independent story. But what happens if, for example, two RPs trust the same STS? Things get more interesting. Figure 4-4 briefly revisits the sign-in sequence, showing the user signing in the first RP application, named *A*.

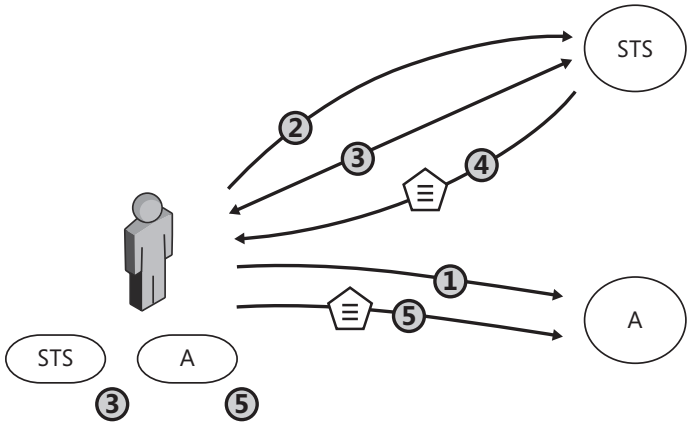

FIGURE 4-4 The user signs in the RP named A, and in so doing it receives session cookies both from the STS and A

By now, you know the drill:

1. The user sends a GET for a page on A.
2. The user is redirected to the STS.
3. The user is authenticated by whatever system the STS chooses and obtains a session cookie.
4. The user gets back a token.
5. The user sends the token to A and gets back a session cookie.

Here step 3 is especially interesting: In Figure 4-4, I assumed the authentication method picked by the STS involves the creation of a session with the STS site itself. That's a reasonable assumption because that's precisely the case with common authentication methods

such as Kerberos (which leverages the session that the user created from her workstation at login time) or Forms authentication (which drops a session cookie, just like the WIF STS template does). If that is the case, at the end of the sign-in sequence the user's machine will have two cookies: one representing the session with A, created by WIF, and one representing the session with the STS. Starting from that situation, let's now look at Figure 4-5 to see what happens when the user signs in with *B*, another RP, that trusts the same STS.

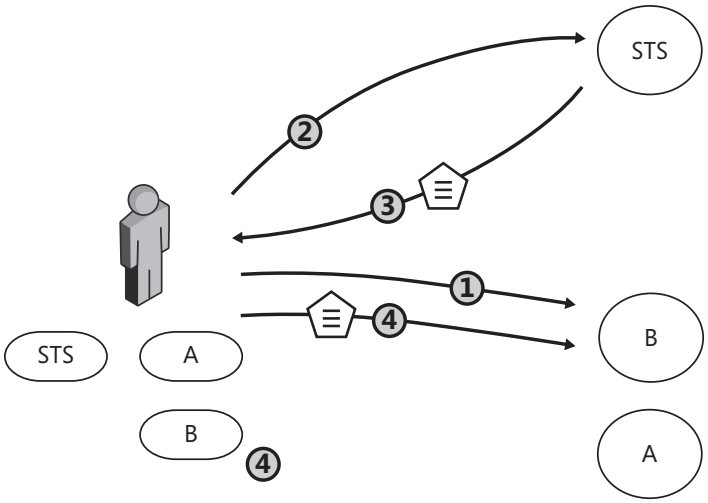

FIGURE 4-5 The user signs in to the RP named B, and the existing session with the STS allows the user to sign in without being prompted for the STS credentials

The flow starts as usual, the user requests a page from B (step 1, as shown in Figure 4-5) and gets redirected to the STS to obtain a token (step 2). However, this time the user is already authenticated with the STS site because there is an active session represented by the STS cookie. This means the request for the STS page—say, *Default.aspx* if you are in the WIF STS template case—leads straight to execution of the *SecurityTokenService* issuing sequence without showing to the user any UI for credential gathering. The token is issued silently (step 3) and forwarded to B (step 4) according to the usual sequence. From the moment the user clicks on the link to B and the browser displays the requested page from B, only some flickering of the address bar in the browser will give away the fact that some authentication took place under the hood. That's pretty much what Single Sign-on (SSO) means: the user went through the experience of signing in only once, and from that moment on the system is able to gain access to further RPs without prompting the user for credentials again.

SSO is an all-time favorite for end users. Using a single set of credentials for different Web sites without being reproached for it? Typing stuff only once? Count me in! This is also something that greatly pleases system administrators, because reducing the number of credentials to manage eases the administrative burden, lowers the probability that users will reuse the same password in different Web sites, and so on.

Note By now, you can certainly see the fundamental difference between authenticating with an STS only once, and silently obtaining tokens for multiple Web sites after that single credential gathering moment and reusing the same credentials across multiple Web sites (each handling their own authentication). Whereas the first approach minimizes the chances of passwords being stolen, the second maximizes it.

You'll find that although most uninitiated people will not understand most of the stuff I covered in this book, everybody will have a clear, intuitive understanding and appreciation of SSO. Perhaps not surprisingly, SSO became the Holy Grail of the industry long before the emergence of claims-based identity, and as of today a lot of people think that the ultimate goal of identity management should be universal SSO.

The good news? As long as the STS creates a session in its authentication method, having SSO across Web site RPs protected via WIF is something that works right out of the box. There's no arcane WS-Federation trick here, just good old cookies and a bit of trust management.

The hands-on lab ASP.NET Membership Provider and Federation (c:\IdentityTrainingKit2010\Labs\MembershipAndFederation) demonstrates how you can easily obtain SSO across Web sites using WIF. In fact, it shows how it is enough to add a page to an existing Web site, without modifying anything else, to add IP capabilities to it. The scenario in the lab modifies a Web site secured via the Membership provider, but this pattern can be applied to any *authentication system.*

Single Sign-out

In one of those rare instances in which building is easier than destroying, you are about to discover that Single Sign-out is somewhat harder to implement than Single Sign-on.

Single Sign-out, or SSOut, takes place when the termination of one session with a specific RP triggers the cleanup of state and other sessions across the same über session. In other words, signing out from one Web site cascades through all the Web sites that were part of the SSO club and signs out from them as well.

Note The basic idea of SSOut is readily understood and can be easily experienced even outside federated scenarios: the sign-out option of Live ID, which (at the time of this writing) throws you out at once from all the Web sites accepting Live ID you've been signing in to, is a good example of that. However, in literature "Single Sign-out" is almost always used as a synonym of "federated sign-out" and is expected to behave as specified by WS-Federation or SAMLP.

The mechanics of SSOut are not very straightforward, especially because the outcome of the entire process relies on all the entities involved receiving messages and complying. Both of those things are hard to enforce without reliable messaging or transactions; hence, the entire thing ends up being a "make your best effort" attempt. This state of affairs was well known to the authors of the WS-Federation specification, who were not especially prescriptive in describing the messages and mechanisms used for implementing SSOut. WIF does support SSOut out of the box for RPs, but the STS template is not especially thorough in implementing all its details. In this section, I'll clue you in to the things you need to add for achieving more complete support.

Signing Out from One RP

Before getting into the details of how to handle signing out from multiple Web sites, let's see what it takes to sign out from just one.

What keeps a user session alive, apart from the sheer Forms authentication machinery? First of all, it's the existence (and validity) of the session cookie generated at sign-on time. The default name used by WIF for that cookie is *FedAuth*, with an additional *FedAuth1…FedAuthn* if the size of the *SessionSecurityToken* requires multiple cookies. You can easily take care of that yourself—it's just a matter of calling *FormsAuthentication.SignOut* and deleting the session cookie (by hand or via *SessionAuthenticationModule.DeleteSessionTokenCookie*).

Second, it's the session with the STS. If you delete the session with the RP but the user still has a valid session with the STS, she will still have access to the RP. The first unauthenticated GET elicits the usual redirect to the STS, and a valid session means that the user will be issued a new token without even being prompted for credentials.

The RP cannot directly change the STS session. In fact, it is not even supposed to know how that session (if any) is implemented to begin with! Luckily, WS-Federation defines a way for the RP to ask the STS to sign out the current principal. It will be up to the STS to decide what specific steps that entails in the context of its own implementation.

The mechanism that WS-Federation uses for signing out is straightforward: you are supposed to do a GET of the STS endpoint page with the parameter *wa=wsignout1.0* and a *wreply* indicating where you want the browser to be redirected after the sign out is done. Once again, this is something you could do yourself; but why bother, when there is something that can take care of both the RP session cleanup and sending the sign-out message to the STS? That something is *FederatedPassiveSignInStatus*, an ASP.NET control that comes with WIF.

FederatedPassiveSignInStatus, as the name implies, can be used for easily displaying on your Web site the current state of the session. Drag it on any page, and its appearance will change according to whether you have a valid session in place. If you do, by default the control appears as a hyperlink with the text "Sign Out." Clicking that link results in the current RP session being cleaned up. If the control property *SignOutAction* is set to *FederatedSignOut*,

the control takes care of sending the *wsignout1.0* message to the STS indicated in the *SessionSecurityToken*. Handy, isn't it? That's my favorite way of implementing sign out with WIF—it's easy and painless.

Warning *FederatedPassiveSignInStatus* has a property, *SignOutPageUrl*, that indicates the page the browser should return to after the sign-out is done. In practice, it's the *wreply* in the *wsignout1.0* message. If you leave the property blank, WIF sets *wreply* to your *wtrealm* and appends "*login.aspx*" to it. Chances are that your Web site does not contain a login page because you are using an STS. If that's the case, you might get an error at the next successful authentication. The bottom line is this: make sure you add a meaningful value to *SignOutPageUrl*.

The WIF STS Template and *wsignout1.0*

In the description of the WIF STS template, I purposefully omitted the code that takes care of signing out. Now that you know what an STS is supposed to do in response to a *wsignout1.0* message, I can get back to it and complete the description of the template. The following code shows the missing branch:

```
else if ( action == WSFederationConstants.Actions.SignOut )
      {
        // Process signout request.
        SignOutRequestMessage requestMessage =
         (SignOutRequestMessage)WSFederationMessage.CreateFromUri( Request.Url );
        FederatedPassiveSecurityTokenServiceOperations.ProcessSignOutRequest(
          requestMessage, User, requestMessage.Reply, Response );
      }
```

SignOutRequestMessage is analogous to *SignInRequestMessage*, in that it's just a dictionary of *querystring* values. *FederatedPassiveSecurityTokenServiceOperations*. *ProcessSignOutRequest* is not all that glamorous either, I'm afraid. It just signs out from the Form authentication session, deletes the WIF session token (if there is any—the STS template does not include *SessionAuthenticationManager* by default) and redirects to the address indicated by *wreply*.

Signing Out from Multiple RPs

From the perspective of the RP from which the user is signing out, cleaning up its own session and sending *wsignout1.0* to the STS is all that is needed for closing the games. If there are other RPs with which the user still entertains an active session, it is responsibility of the STS to propagate the sign-out to them as well.

All that is left to do is for the other RPs to get rid of their sessions. Note that the STS already eliminated its own session with the user; hence, there is no risk of silent re-issuing after the other RPs do their cleanup.

Once again, WS-Federation provides a mechanism for that. I won't go into the details here—it suffices to say that one way of requesting a cleanup to one RP is simply by doing a GET request on the RP and including in the query string the action *wa=wsignoutcleanup1.0*. You could specify an address via *wreply* to return to after the cleanup is done, but things can get problematic here. What if you have three RPs that need to clean up their sessions? If you are relying on the browser to perform the necessary GETs, you'd have to chain the requests. In addition to being complicated, this is a very brittle approach because something going wrong with one RP would jeopardize the chance of sending cleanup requests to all the subsequent RPs in the list. The STS can avoid using the browser and send the GET requests directly, but again, this is not very straightforward. For those reasons and others, the presence of a *wreply* is optional in *wsignoutcleanup1.0* messages; it is acceptable to return something from the RP that somehow indicates the outcome of the operation. There's more: the cleanup operation is required to be idempotent—that is, you should be able to call the same operation multiple times without affecting the outcome or raising errors. This allows you to retry the operation if you think something went wrong, without worrying about creating error situations.

Now for some good news: RPs secured via WIF handle *wsignoutcleanup1.0* messages out of the box. The WSFAM looks out for those messages in its *AuthenticateRequest* handler. If the incoming message has a *wsignoutcleanup1.0* action, WSFAM promptly deletes the session cookie and drops the corresponding token from the cache.

What sets apart the cleanup from all other actions I've described so far is that it might not end with a redirect. If the message contains a *wreply*, WSFAM dutifully returns a 302 message to the indicated location; if it doesn't, it will return an image or .gif of a green check mark.

Returning the bits of one image upon successful cleanup is part of a clever strategy for working around the "chaining of sign-out redirects" problem described earlier. After the STS successfully clears its own session, it can return a page containing an <*img*> element for each RP whose session is up for cleanup. If the *src* value of the <*img*> elements is of the form *https://RPAddress/Default.aspx?wa=wsignoutcleanup1.0*, just rendering the list of images in the browser sends as many cleanup messages to the RPs in the list. Every successful cleanup sends back the image of the green check box, which the STS page can use for confirming that the sign-out actually took place for a given RP. Failure to render the image might be an indication that something went wrong with the cleanup operations.

All of the preceding activity relies on the fact that the STS will keep track of the RPs for which it issued a token in the context of one federated session. At sign-out time, the STS needs to remember the address of all RPs in order to generate the correct cleanup URIs for the *src* of the images collection in the sign-out page. The STS can use whatever state-preserving mechanism its owner sees fit. In my samples, I usually keep the list of RP URIs in a protected cookie because it requires zero state-management code on the server.

Did you get lost in all the back and forth required by the SSOut process? Let's take a look at one example. Figure 4-6 illustrates the Single Sign-out message flow across two Web sites and a common STS, together with what happens to the client's cookie collection as the sequence progresses.

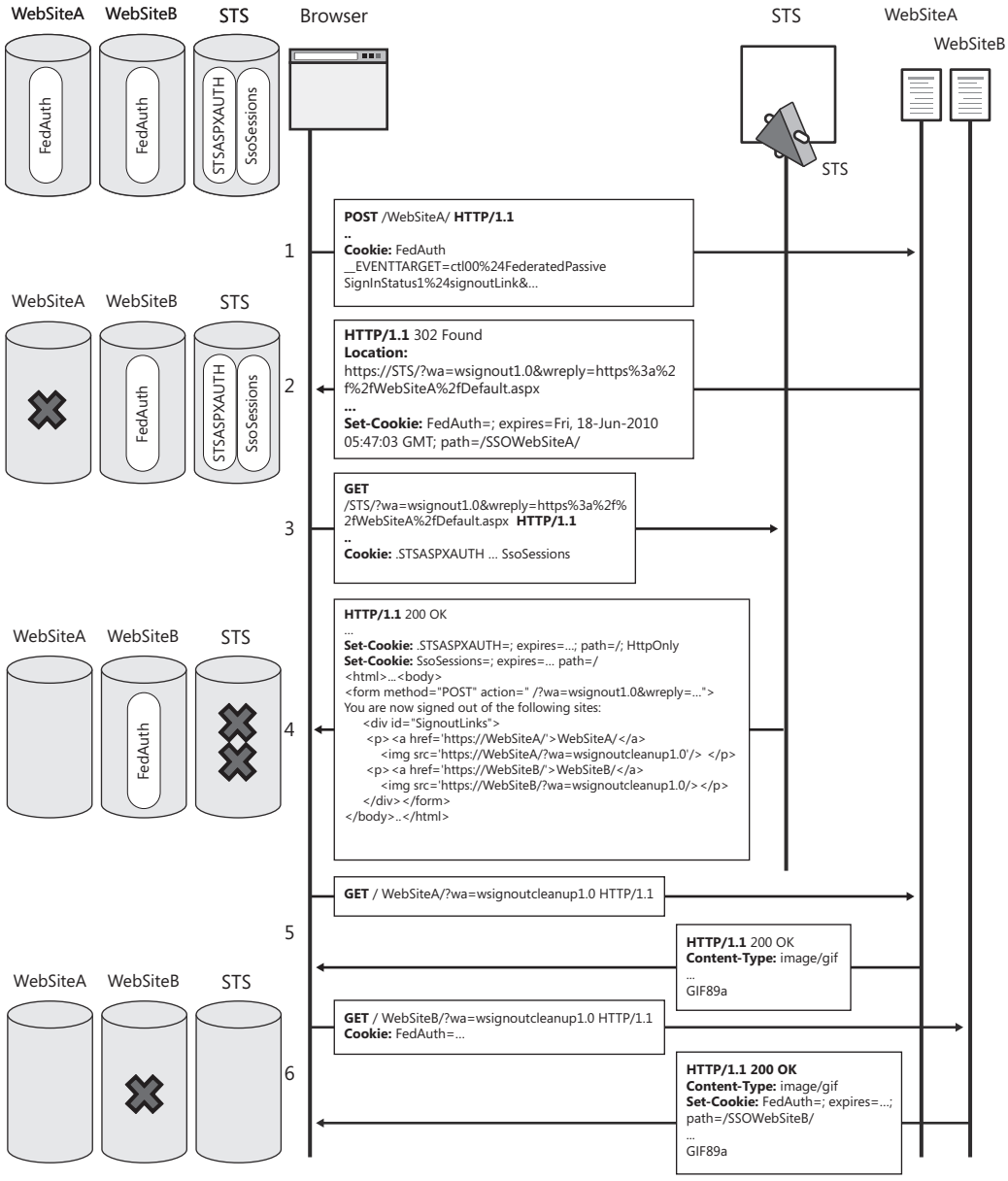

FIGURE 4-6 A Single Sign-out process taking place as described in WS-Federation

Let's examine every step. In the beginning, the user is signed in to WebSiteA and WebSiteB via tokens obtained from STS, and his browser is currently on WebSiteA. His cookie collection contains a *FedAuth* session cookie for each RP and one Forms authentication cookie (*STSASPXAUTH*) with STS. It also has an *SsoSessions* cookie with STS, which contains the list of RPs for which the STS issued a token in the context of its *STSASPXAUTH* session. Here's how the process unfolds:

1. The user clicks on a *FederatedSignInStatus* control instance on WebSiteA, triggering a POST in the authenticated session described by WebSiteA's *FedAuth* cookie. The *SignOutAction* property of the control is set to *FederatedPassiveSignOut*.
2. WebSiteA receives the request for signing out. As a result, it destroys its own session (by cleaning *FedAuth* from the WebSiteA cookie collection on the client) and redirects the browser to send a sign-out message to the STS that originated the current session.
3. The browser follows the redirect, sending to the STS the sign-out message, along with the session cookie STSASPXAUTH and the cookie containing the list of RPs with whom the user might still entertain active sessions.
4. The STS reacts by cleaning up all its cookies and sends back a page that contains images whose *src* URIs are in fact cleanup messages for all the RPs listed in the *SsoSessions* cookie—that is, WebSiteA and WebSiteB.
5. The browser renders the first image, pointing to WebSiteA. Hence, it sends a GET for its source, which in fact delivers a cleanup message. WebSiteA already cleaned up its session because it was the originator of the Single Sign-out sequence. If the STS had known this, it could have avoided adding WebSiteA to the list of cleanup RPs; however, nothing bad happens, thanks to the idempotency requirements of *wssignoutcleanup1.0* messages. WebSiteA simply returns the bits of the GIF indicating that cleanup successfully took place.
6. The browser renders the image, pointing to WebSiteB. WebSiteB receives the cleanup message and reacts by deleting its own *FedAuth* cookie and returning the bits of the GIF of the check mark as expected. At this point, all the sessions have been cleaned up: the Single Sign-out concluded successfully, and the user can see on the STS page the list of Web sites he has been signed out from.

Once you get the hang of it, it's really not that hard. One of the things I like best about this approach is that it allows you to herd the behavior of multiple Web sites without knowing any detail. Some sites could be hosted on your intranet, others could be hosted in the cloud, or sites could be running on different stacks and operating systems, but as long as they all speak via WS-Federation and share a common, trusted ground, the right thing just happens.

The WIF STS Template and Single Sign-out

As you saw earlier, the STS template handles *wssignout1.0* messages. However, it does not propagate them via *wssignoutcleanup1.0* to the other RPs in the session, nor does it contain any mechanism for keeping track of the RPs in the current session at issuance time. The sample discussed here offers such a mechanism in the *SingleSignOnManager* class. It is a façade for a collection of RP URIs saved in a cookie, which gets updated with the RP address every time the STS issues a token (in *GetOutputClaimsIdentity*) and that can be looked up when it's time to send cleanup messages. That is just one example—you can use any equivalent mechanism. Once you have that capability, enhancing the STS template code to support SSOut is easy. Consider the following modified version of the sign-out branch in the *Default.asp.cs* code:

```
else if ( action == WSFederationConstants.Actions.SignOut )
      {
        // Process signout request.
        SignOutRequestMessage requestMessage =
          (SignOutRequestMessage)WSFederationMessage.CreateFromUri( Request.Url );

        FederatedPassiveSecurityTokenServiceOperations.ProcessSignOutRequest(
           requestMessage, User, /*requestMessage.Reply*/ null, Response );
        // new
        string[] signedInUrls = SingleSignOnManager.SignOut();
        lblSignoutText.Visible = true;
        foreach (string url in signedInUrls)
        {
          SignoutLinks.Controls.Add(
            new LiteralControl(String.Format(
             "<p><a href='{0}'>{0}</a> <img src='{0}?wa=wsignoutcleanup1.0'
              title='Signout request: {0}?wa=wsignoutcleanup1.0'/></p>," url)));
        }
      }
```

The changes are straightforward. The call to *ProcessSignOutRequest* does not redirect to *wreply*, because after it cleaned up its own session there's still work to do that would not be done if it redirected as in the default case. After cleaning its own session, the STS prepares the UI for the sign-out by turning on the visibility of a sign-out message (here, in a label). The call to *SingleSignOutManager* returns the list of all the RPs whose session should be cleaned up. The *foreach* that appears below that uses that list for generating and appending to the page as many images as needed, which will dispatch the cleanup message once they are rendered.

More About Sessions

I briefly touched on the topic of sessions at the end of Chapter 3, where I showed you how you can keep the size of the session cookie independent from the dimension of its originating token by saving a reference to session state stored on the server side. The WIF programming model goes well beyond that, granting you complete control over how sessions are handled. Here I'd like to explore with you two notable examples of that principle in action: sliding sessions and network load-balancer-friendly sessions.

Sliding Sessions

By default, WIF creates *SessionSecurityTokens* whose validity is based on the validity of the incoming token. You can overrule that behavior without writing any code, by adding to the *<microsoft.identityModel>* element in the *web.config* file something like the following:

```
<securityTokenHandlers>
    <add type="Microsoft.IdentityModel.Tokens.SessionSecurityTokenHandler,
              Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
              PublicKeyToken=31bf3856ad364e35">
      <sessionTokenRequirement lifetime="0:02" />
    </add>
</securityTokenHandlers>
```

Note The *lifetime* property can restrict only the validity expressed by the token to begin with. In the preceding code snippet, I set the lifetime to 2 minutes, but if the incoming security token was valid for just 1 minute, the session token would have 1 minute of validity. If you want to increase the validity beyond what the initial token specified, you need to do so in code (by subclassing *SessionSecurityTokenHandler* or by handling *SessionSecurityTokenReceived*).

Now, let's say that you want to implement a more sophisticated behavior. For example, you want to keep the session alive indefinitely as long as the user is actively working with the pages. However, you want to terminate the session if you do not detect user activity in the past 2 minutes, regardless of the fact that the initial token would still be valid. This is a common requirement for Web sites that reveal personally identifiable information (PII) or give control to banking operations. Those are cases in which you want to ensure that the user is actually in front of the machine and the pages are not abandoned to the mercy (or mercenary instincts) of bystanders.

In Chapter 3, I hinted at this scenario, suggesting that it could be solved by subclassing the *SessionAuthenticationModule*. That is the right strategy if you expect to reuse this functionality over and over again across multiple applications, given that it neatly packages it in a class you can include in your code base. In fact, SharePoint 2010 offers sliding sessions and implements those precisely in that way. If, instead, this is an improvement you need to apply

only occasionally, or you own just one application, you can obtain the same effect simply by handling the *SessionSecurityTokenReceived* event. Take a look at the following code:

```
<%@ Application Language="C#" %>
<%@ Import Namespace="Microsoft.IdentityModel.Web" %>
<%@ Import Namespace="Microsoft.IdentityModel.Tokens" %>

<script runat="server">

  void SessionAuthenticationModule_SessionSecurityTokenReceived
       (object sender, SessionSecurityTokenReceivedEventArgs e)
  {
    DateTime now = DateTime.UtcNow;
    DateTime validFrom = e.SessionToken.ValidFrom;
    DateTime validTo = e.SessionToken.ValidTo;
    double halfSpan = (validTo - validFrom).TotalMinutes / 2;
    if ( validFrom.AddMinutes( halfSpan ) < now && now < validTo )
    {
      SessionAuthenticationModule sam = sender as SessionAuthenticationModule;
      e.SessionToken = sam.CreateSessionSecurityToken(e.SessionToken.ClaimsPrincipal,
e.SessionToken.Context,
         now, now.AddMinutes(2), e.SessionToken.IsPersistent);
      e.ReissueCookie = true;
    }
  }
  //...
```

As you certainly guessed, this is a fragment of the *global.asax* file of the RP application. *SessionSecurityTokenReceived* gets called as soon as the session cookie is deserialized (or resolved from the cache if you are in session mode). Here you verify whether you are within the second half of the validity window of the session token. If you are, you extend the validity to another 2 minutes, starting now. That change takes place on the in-memory instance of the *SessionSecurityToken*. Setting *ReissueToken* to true instructs the *SessionAuthenticationModule* to persist the new settings in the cookie after the execution leaves *SessionSecurityTokenReceived*. Let's say that the token is valid between 10:00 a.m. and 10:02 a.m. If the current time falls between 10:01 a.m. and 10:02 a.m.—say, 10:01:15—the code sets the new validity boundaries to go from 10:01:15 to 10:03:15 and saves those in the session cookie.

Note This is the same heuristic that FormsAuthentication uses for sliding expiration. Why renew the session only during the second half of the validity interval? Well, writing the cookie is not for free. This is just a heuristic for reducing the times at which the session gets refreshed, but you can certainly choose to apply different strategies.

If the current time is outside the validity interval, this implementation of *SessionSecurityTokenReceived* will have no effect. The *SessionAuthenticationModule* will take care of handling the expired session right after. Note that an expired session does not elicit any explicit sign-out process. If you recall the discussion about SSO and SSOut just a few

pages earlier, you'll realize that if the STS session outlives the RP session the user will just silently re-obtain the authentication token and renew the session without even realizing anything happened.

Sessions and Network Load Balancers

By default, session cookies written by WIF are protected via DPAPI, taking advantage of the RP's machine key. Such cookies are completely opaque to the client and anybody else who does not have access to that specific machine key.

This works well when all the requests in the context of a user session are aimed at the same machine. But what happens when the RP is hosted on multiple machines—for example, in a load-balanced environment? A session cookie might be created on one machine and sent to a different machine at the next postback. Unless the two machines share the same machine key and use it for encrypting the cookie instead of taking advantage of the DPAPI Encryption key, a cookie originated from machine A will be unreadable from machine B.

There are various solutions to the situation. One obvious one is using sticky sessions—that is, guaranteeing that a session beginning with machine A keeps referring to A for all subsequent requests. I am not a big fan of that solution because it dampens the advantages of using a load-balanced environment. Furthermore, you might not always have a say in the matter—for example, if you are hosting your applications on a third-party infrastructure (such as Windows Azure), your control of the environment will be limited.

Another solution is to synchronize the machine keys of every machine and use those for encrypting cookies. I like this better than using sticky sessions, but there is an approach I like even better. More often than not, your RP application will use Secure Sockets Layer (SSL), which means you need to make the certificate and corresponding private key available on every node. It makes perfect sense to use the same cryptographic material for securing the cookie in a load-balancer-friendly way.

WIF makes the process of applying the aforementioned strategy in ASP.NET applications trivial. The following code illustrates how it can be done:

```
public class Global : System.Web.HttpApplication
{
  //...
    void OnServiceConfigurationCreated(object sender, ServiceConfigurationCreatedEventArgs
e)
    {
      //
      // Use the <serviceCertificate> to protect the cookies that are
      // sent to the client.
      //
      List<CookieTransform> sessionTransforms =
        new List<CookieTransform>(new CookieTransform[] {
          new DeflateCookieTransform(),
```

```
            new RsaEncryptionCookieTransform(e.ServiceConfiguration.ServiceCertificate),
            new RsaSignatureCookieTransform(e.ServiceConfiguration.ServiceCertificate)  });
        SessionSecurityTokenHandler sessionHandler = new
SessionSecurityTokenHandler(sessionTransforms.AsReadOnly());

    e.ServiceConfiguration.SecurityTokenHandlers.AddOrReplace(sessionHandler);
    }


    protected void Application_Start(object sender, EventArgs e)
    {
      FederatedAuthentication.ServiceConfigurationCreated += OnServiceConfigurationCreated;
    }
```

Instead of using the usual inline approach, this time I am showing you the code-behind file *global.asax.cs*. *OnServiceConfigurationCreated* is—Surprise! Surprise!—a handler for the *ServiceConfigurationCreated* event and fires just after WIF reads the configuration. If you make changes here, you have the guarantee that they will already be applied from the request coming in.

Note Contrary to what various samples out there would lead you to believe, *OnServiceConfigurationCreated* is pretty much the only WIF event handler that should be associated to its event in *Application_Start*. This has to do with the way (and the number of times) ASP.NET invokes the handlers though the application lifetime.

The code is self-explanatory. It creates a new list of *CookieTransform* transformations, which takes care of cookie compression, encryption, and signature. The last two take advantage of the *RsaxxxxCookieTransform*, taking in input the certificate defined for the RP in the *web.config* file.

Note Why do you sign the cookie? Wouldn't it be enough to encrypt it? If you use the RP certificate, encryption would not be enough. Remember, the RP certificate is a *public* key. If you just encrypt it, a crafty client can just discard the session cookie, create a new one with super-privileges in the claims, and encrypt it with the RP certificate. The RP would not be able to tell the difference. Adding the signature successfully prevents this attack because it requires a private key, which is not available to the client or anybody else but the RP itself.

The new transformations list is assigned to a new *SessionSecurityTokenHandler* instance, which is then used for overriding the existing session handler. From this point on, all session cookies will be handled using the new strategy. That's it! As long as you remember to add an entry for the service certificate in the RP configuration, you've got network load balancing (NLB)–friendly sessions without having to resort to compromises such as sticky sessions.

Federation

At the beginning of the chapter, I introduced the Federation Provider and discussed some of the advantages that the IP-FP-RP pattern offers. The temptation to expand the architectural considerations about this important pattern is strong; however, here I want to keep the focus on WIF and give you a concrete coding example. There are many good high-level introductions to the topic you can refer to.

For a good introduction to the subject, refer to A Guide to Claims-Based Identity and Access Control *by Dominick Baier, Vittorio Bertocci, Keith Brown, Matias Woloski, and Eugenio Pace (Microsoft Press, 2010).*

WIF does not really care if the STS used by the RP is an IP-STS or an R-STS. Both types look the same in their metadata description and, despite the differences in the sequence that ultimately lead to that, they both issue a token as requested. It helps to see this in action in a concrete example.

Note As usual, in a realistic scenario you can expect the R-STS to be provided by one ADFS 2.0 instance playing the FP role. Once again, for educational purposes, I'll take advantage of custom STSes here.

Do you recall the first example we explored in Chapter 2? It was a classic RP-IP scenario, but it is very easy to transform it into a toy federation sample. Just right-click on the *BasicWebSite_STS* project in Solution Explorer, select the Add STS Reference entry, and use the wizard for creating yet another new STS project in the current solution.

Note The Add STS Reference Wizard adds an <*httpModules*> element in the <*system.web*> section of *BasicWebSite_STS* config, which does not play well with the IIS integrated pipeline. You might have to comment out that <*httpModules*> entry.

Figure 4-7 shows the new solution layout.

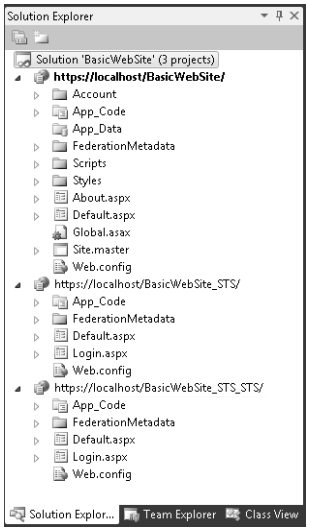

FIGURE 4-7 *BasicWebSite* trusts *BasicWebSite_STS*, which in turn trusts *BasicWebSite_STS_STS*

Nothing changed for the RP, *BasicWebSite*, which is still outsourcing authentication to *BasicWebSite_STS*. *BasicWebSite_STS* was an IP-STS when we started, because it was an unmodified instance of the WIF STS template. After the wizard configured it to outsource authentication to *BasicWebSite_STS_STS*, however, *BasicWebSite_STS* became an R-STS; therefore, its *login.aspx* page will not be used anymore. If you run the solution you'll observe the browser being redirected from *BasicWebSite* to *BasicWebSite_STS*, which will redirect right away to *BasicWebSite_STS_STS*, which will finally show its own *login.aspx* page. After you click Submit on the login form, the flow will go through the chain in the opposite order: *BasicWebSite_STS_STS* will issue a token that will be used for signing in *BasicWebSite_STS*, which in turn will issue a new token that will be used for signing in *BasicWebSite*. Figure 4-8 summarizes the sign-in flow.

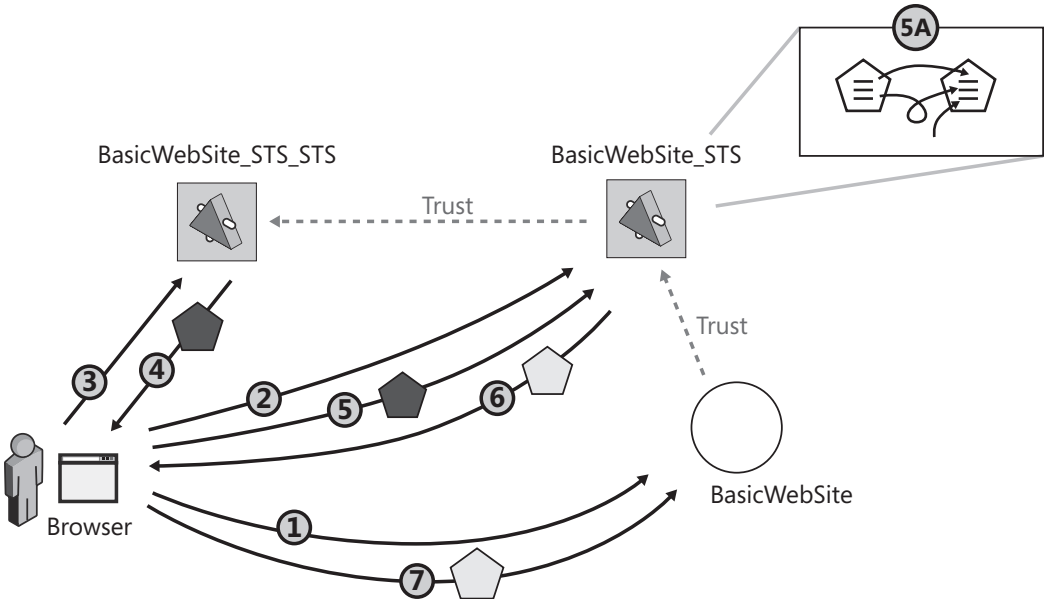

FIGURE 4-8 The authentication flow linking *BasicWebSite*, *BasicWebSite _STS*, and *BasicWebSite_STS_STS*

1. The user requests a page from BasicWebSite.
2. Because the user is not authenticated, he is redirected to BasicWebSite_STS for authentication.
3. BasicWebSite_STS itself outsources authentication to BasicWebSite_STS_STS; hence, it redirects the request accordingly
4. Once the user successfully authenticates with BasicWebSite_STS_STS, he gets back a token.
5. The user gets redirected back to BasicWebSite_STS, which validates the token from BasicWebSite_STS_STS and considers the user authenticated thanks to it.
6. BasicWebSite_STS issues a token to the user, as requested.
7. The user gets back to BasicWebSite with the token obtained from BasicWebSite_STS as required, and the authenticated session starts.

Convoluted? A bit, perhaps. On the upside, *BasicWebSite* is now completely isolated from the actual identity provider—changes in the IP will not affect the RP. If you have multiple RPs, you can now have them all trust the same R-STS, which will take care of enforcing any changes in the relationship with the IP (or IPs, as I'll show in a moment) without requiring any ad-hoc intervention on the RP code or configuration itself. Pretty handy!

Transforming Claims

The example in the preceding section modified the authentication flow to conform to the federation pattern, but it didn't really change the way in which *BasicWebSite_STS* processes claims. With its hard-coded claims entries, the default WIF STS template behavior mimics that of an IP-STS; whereas in its new FP role, *BasicWebSite_STS* is expected to process the incoming claims (in this case, from *BasicWebSite_STS_STS*). If you want to change *BasicWebSite_STS* into a proper R-STS, you need to modify the *GetOutputClaimsIdentity* method of the *CustomSecurityTokenService* class.

As you already know, in *GetOutputClaimsIdentity* the incoming claims are available in the *IClaimsPrincipal principal* parameter. You can pretty much do anything you want with the incoming claims, but I find it useful to classify the possible actions into three (non-exhaustive) categories: pass-through, modification, and injection of new claims. They are represented in step 5a of Figure 4-8. Here is a simple example of a *GetOutputClaimsIdentity* implementation that features all three methods:

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

    ClaimsIdentity outputIdentity = new ClaimsIdentity();

    IClaimsIdentity incomingIdentity = (IClaimsIdentity)principal.Identity;

    // Pass-through
    Claim nname = (from c in incomingIdentity.Claims
             where c.ClaimType == ClaimTypes.Name
             select c).Single();
    Claim nnnm = new Claim(ClaimTypes.Name, nname.Value, ClaimValueTypes.String, nname.
OriginalIssuer);
    outputIdentity.Claims.Add(nnnm);

    // Modified
    string rrole = (from c in incomingIdentity.Claims
            where c.ClaimType == ClaimTypes.Role
            select c.Value).Single();
    outputIdentity.Claims.Add(new Claim(ClaimTypes.Role, "Transformed " + rrole));

    // New
    outputIdentity.Claims.Add(new Claim("http://maseghepensu.it/hairlength",
                                        "a value", ClaimValueTypes.Double));

    return outputIdentity;
  }
```

Before going into the details of how the various transformations work, it is finally time to take a deeper look at that *Claim* class we've been using without giving it too much thought so far. Here are the various properties of the class and some methods of interest:

```
public class Claim
{
  // Methods

  public virtual Claim Copy();
  public virtual void SetSubject(IClaimsIdentity subject);
  // Properties

  public virtual string ClaimType { get; }
  public virtual string Issuer { get; }
  public virtual string OriginalIssuer { get; }
  public virtual IDictionary<string, string> Properties { get; }
  public virtual IClaimsIdentity Subject { get; }
  public virtual string Value { get; }
  public virtual string ValueType { get; }
}
```

One thing that immediately grabs your attention is that all properties of *Claim* are read-only: after the class has been created, the values cannot be changed. The only exception is the subject to which the *Claim* instance is referring to: *SetSubject* will change the value of the *Subject* property to a new *IClaimsIdentity*.

You are already familiar with *Value* and *ClaimType* because I've been using those throughout the entire book. *ValueType* is more interesting. It allows you to specify a type for the claim value, which the claim consumer can use to deserialize the claim in a common language runtime (CLR) type (or whatever type system your programming stack requires if you are not in .NET) other than the default string. That is a key enabler for applying complex logic to claims. Without knowing that *DateOfBirth* should be deserialized in a *DateTime*, you'll find it difficult to verify whether it is below or above a given threshold. Note that the *ValueType* is just one indication: the *Value* returned by the claim is always a string regardless of the *ValueType*. You'll have to call the appropriate *Parse* method (or similar) yourself.

The *Properties* dictionary is used for carrying extra information about the claim itself when the protocol requires it. For example, in SAML2 you might have properties such as *SamlAttributeDisplayName* assigned to a claim.

Note The WIF token handlers will *not* serialize the properties. If you want them to travel, you'll have to take care of that yourself.

The *Issuer* property is a string representing the token issuer from which the claim has been extracted. The string itself comes from the mapping that *IssuerNameRegistry* makes between the certificate used for signing the token and the friendly name assigned to the associated issuer. The *OriginalIssuer* property records the first issuer that produced this claim in the federation chain. I've included more details about this in the "Pass-Through Claims" section.

Claim Types and Value Constants

WIF offers two collections of string constants that gather most of the known claim type URIs. One is *Microsoft.IdentityModel.Protocols.WSIdentity.WSIdentityConstants.ClaimTypes* (which is almost the same as the WCF collection *System.IdentityModel.Claims.ClaimTypes*); the other is *Microsoft.IdentityModel.Claims.ClaimTypes* (which is a superset of the first one). For your reference, the content of *Microsoft.IdentityModel.Claims.ClaimTypes* is listed next. Note that some popular claim types (such as *Group*) are kept in the *Prip* subtype and are often overlooked. *Prip* stands for WS-Federation Passive Requestor Interoperability Profile, which is a specific subset of WS-Federation used during early multivendor interoperability tests.

```
public static class ClaimTypes
{
    // Fields
    public const string Actor =
      "http://schemas.xmlsoap.org/ws/2009/09/identity/claims/actor";
    public const string Anonymous =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/anonymous";
    public const string Authentication =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/authentication";
    public const string AuthenticationInstant =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationinstant";
    public const string AuthenticationMethod =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationmethod";
    public const string AuthorizationDecision =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/authorizationdecision";
    public const string ClaimType2005Namespace =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims";
    public const string ClaimType2009Namespace =
      "http://schemas.xmlsoap.org/ws/2009/09/identity/claims";
    public const string ClaimTypeNamespace =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims";
    public const string CookiePath =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/cookiepath";
    public const string Country =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/country";
    public const string DateOfBirth =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/dateofbirth";
    public const string DenyOnlyPrimaryGroupSid =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/
denyonlyprimarygroupsid";
    public const string DenyOnlyPrimarySid =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/denyonlyprimarysid";
    public const string DenyOnlySid =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/denyonlysid";
    public const string Dns =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/dns";
    public const string Dsa =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/dsa";
```

```
    public const string Email =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress";
    public const string Expiration =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/expiration";
    public const string Expired =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/expired";
    public const string Gender =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/gender";
    public const string GivenName =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname";
    public const string GroupSid =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/groupsid";
    public const string Hash =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/hash";
    public const string HomePhone =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/homephone";
    public const string IsPersistent =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/ispersistent";
    public const string Locality =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/locality";
    public const string MobilePhone =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/mobilephone";
    public const string Name =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name";
    public const string NameIdentifier =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier";
    public const string OtherPhone =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/otherphone";
    public const string PostalCode =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/postalcode";
    public const string PPID =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/
privatepersonalidentifier";
    public const string PrimaryGroupSid =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/primarygroupsid";
    public const string PrimarySid =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/primarysid";
    public const string Role =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/role";
    public const string Rsa =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/rsa";
    public const string SerialNumber =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/serialnumber";
    public const string Sid =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/sid";
    public const string Spn =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/spn";
    public const string StateOrProvince =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/stateorprovince";
    public const string StreetAddress =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/streetaddress";
    public const string Surname =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname";
```

```
    public const string System =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/system";
    public const string Thumbprint =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/thumbprint";
    public const string Upn =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn";
    public const string Uri =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/uri";
    public const string UserData =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/userdata";
    public const string Version =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/version";
    public const string Webpage =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/webpage";
    public const string WindowsAccountName =
      "http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname";
    public const string X500DistinguishedName =
      "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/x500distinguishedname";

    // Nested Types
    public static class Prip
    {
        // Fields
        public const string ClaimTypeNamespace = "http://schemas.xmlsoap.org/claims";
        public const string CommonName = "http://schemas.xmlsoap.org/claims/
CommonName";
        public const string Email = "http://schemas.xmlsoap.org/claims/EmailAddress";
        public const string Group = "http://schemas.xmlsoap.org/claims/Group";
        public const string Upn = "http://schemas.xmlsoap.org/claims/UPN";
    }
}
```

You can, of course, create your own claim types. However, I suggest that before doing so you take a look at the Information Card Foundation Web site, which (among other things) gathers all the known and emergent claim types from the community. The direct address is *http://informationcard.net/resources/claim-catalog*.

WIF also offers various constants representing common types of claim values:

```
public static class ClaimValueTypes
{
    // Fields
    public const string Base64Binary = "http://www.w3.org/2001/XMLSchema#base64Binary";
    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 DaytimeDuration = "http://www.w3.org/TR/2002/WD-xquery-
operators-20020816#dayTimeDuration";
    public const string Double = "http://www.w3.org/2001/XMLSchema#double";
    public const string DsaKeyValue = "http://www.w3.org/2000/09/xmldsig#DSAKeyValue";
    public const string HexBinary = "http://www.w3.org/2001/XMLSchema#hexBinary";
    public const string Integer = "http://www.w3.org/2001/XMLSchema#integer";
```

```
    public const string KeyInfo = "http://www.w3.org/2000/09/xmldsig#KeyInfo";
    public const string Rfc822Name = "urn:oasis:names:tc:xacml:1.0:data-
type:rfc822Name";
    public const string RsaKeyValue = "http://www.w3.org/2000/09/xmldsig#RSAKeyValue";
    public const string String = "http://www.w3.org/2001/XMLSchema#string";
    public const string Time = "http://www.w3.org/2001/XMLSchema#time";
    public const string X500Name = "urn:oasis:names:tc:xacml:1.0:data-type:x500Name";
    private const string Xacml10Namespace = "urn:oasis:names:tc:xacml:1.0";
    private const string XmlSchemaNamespace = "http://www.w3.org/2001/XMLSchema";
    private const string XmlSignatureConstantsNamespace =
     "http://www.w3.org/2000/09/xmldsig#";
    private const string XQueryOperatorsNameSpace =
     "http://www.w3.org/TR/2002/WD-xquery-operators-20020816";
    public const string YearMonthDuration =
     "http://www.w3.org/TR/2002/WD-xquery-operators-20020816#yearMonthDuration";
}
```

The types are represented according to W3C and OASIS type URIs, but the mapping to CLR types is obvious most of the time.

Now that you understand a bit better how the *Claim* class works, let's resume the discussion about the claim transformations.

Pass-Through Claims

One of the most common transformations you'll want to apply to your claims is…no transformation at all. Sometimes the IP directly issues the claims the RP needs; hence, you have to make sure that those claims are reissued as-is by the R-STS.

Although the claim type and value come straight from the incoming values, the fact that the new claim is issued in a token signed by the R-STS makes the R-STS itself the asserting party and shadows the original issuer. The R-STS might even be accepting tokens from multiple issuers, which would complicate things further. There could be situations in which knowing the actual origin of the claim could change the way in which the information it carries is processed; therefore, it is important to somehow let the RP know which IP issued the claim in the first place. This is done by setting the *OriginalIssuer* property of the outgoing claim to the *OriginalIssuer* carried by the claim you are re-issuing. Here are the relevant lines from the *GetOutputClaimsIdentity* implementation shown earlier:

```
// Pass-through
    Claim nname = (from c in incomingIdentity.Claims
             where c.ClaimType == ClaimTypes.Name
             select c).Single();
    Claim nnnm = new Claim(ClaimTypes.Name, nname.Value, ClaimValueTypes.String, "," nname.
OriginalIssuer);
    outputIdentity.Claims.Add(nnnm);
```

In this example, the claim to be reissued is the *Name* claim. The code retrieves it from the incoming principal, and then it just creates a new claim that copies everything from the original except for the issuer. (Here the issuer parameter is left empty because it is going to be overridden with the current R-STS, anyway.) That snippet is designed to surface to you the use of *OriginalIssuer*, but in fact you can use a more compact form using *Copy* as shown here:

```
// Pass-through
    Claim nname = (from c in incomingIdentity.Claims
             where c.ClaimType == ClaimTypes.Name
             select c).Single();
    Claim nnnm = nname.Copy();
    outputIdentity.Claims.Add(nnnm);
```

Modifying Claims and Injecting New Claims

The distinction between modifying claims and injecting new claims is a bit philosophical, because from the code perspective the two transformations are the same.

Modifying a claim means producing a new claim by processing or combining the value of one or more incoming claims, according to arbitrary logic. An excellent example of that is given by the ADFS 2.0 claims-transformation language, which allows administrators to specify transformations without writing any explicit code. Of course, in *GetOutputClaimsIdentity* you can literally write whatever logic you want.

Injecting new claims usually entails looking up new information about the incoming subject—information that was not available to the IP but that the RP needs. A classic example is the buyer's profile: imagine that the user is one employee, the IP is the user's employer, and the RP is some kind of online shop. The R-STS might maintain information such as the last 10 items the user bought, data that the employer does not keep track of and that should be injected by the resource organization—for example, in the R-STS. The challenge here can be choosing which incoming claims should be used for uniquely identifying the current user and looking up his data in the R-STS profile store. Whereas the IP has one strong incentive to have such a unique identifier—because that is usually needed in order to apply the mechanics of the authentication method of choice—the R-STS does not have a similar requirement per se. The claims chosen should be unique, at least in the context of the current R-STS, and stable enough to be reusable across multiple transactions. The e-mail claim is a good example, but of course it's not a perfect one because e-mail addresses do change from time to time—think of the situation where interns become full-time employees and similar events.

Home Realm Discovery

One of the great advantages of federation is the possibility of handling multiple identity providers without having to change anything in the RP itself. The Federation Providers can

take care of all the trust relationships. Extending the audience of the application without paying any complexity price is great; however, the sheer possibility of using more than one IP does introduce a new problem: when an unauthenticated user shows up, which IP should she ultimately authenticate with? In the trivial federation case examined so far, the one with one FP and one IP, the answer is obvious: the redirect chain crawls all the way to the IP and back. When you have more than one IP, however, how does the R-STS decide if the redirect should go to IP A or IP B?

The problem of deciding which IP should authenticate the user is well known in literature, and it goes under the name of Home Realm Discovery (HRD). The HRD problem has many solutions, although as of today they are mostly ad hoc and what works in one given scenario might not be suitable for another. For example, one classic solution (offered out of the box by ADFS 2.0) asks the R-STS to show a Web page in which the user can pick his own realm among the list of all trusted IPs. This is often a good solution, but there are situations in which it is not advisable to reveal the list of all trusted IPs. Furthermore, sometimes asking the user to make a choice is inconvenient or unacceptable, in which case the IP selection should be done silently according to some criteria.

WS-Federation provides a parameter that can be useful in handling HRD: *whr*. It is meant to carry the address (or the *urn:* identifier) of the home realm. An R-STS receiving a *wsignin1.0* message that includes *whr* will consider *whr* content to be the IP-STS of the requestor and will drive the sequence accordingly. (See Figure 4-9.)

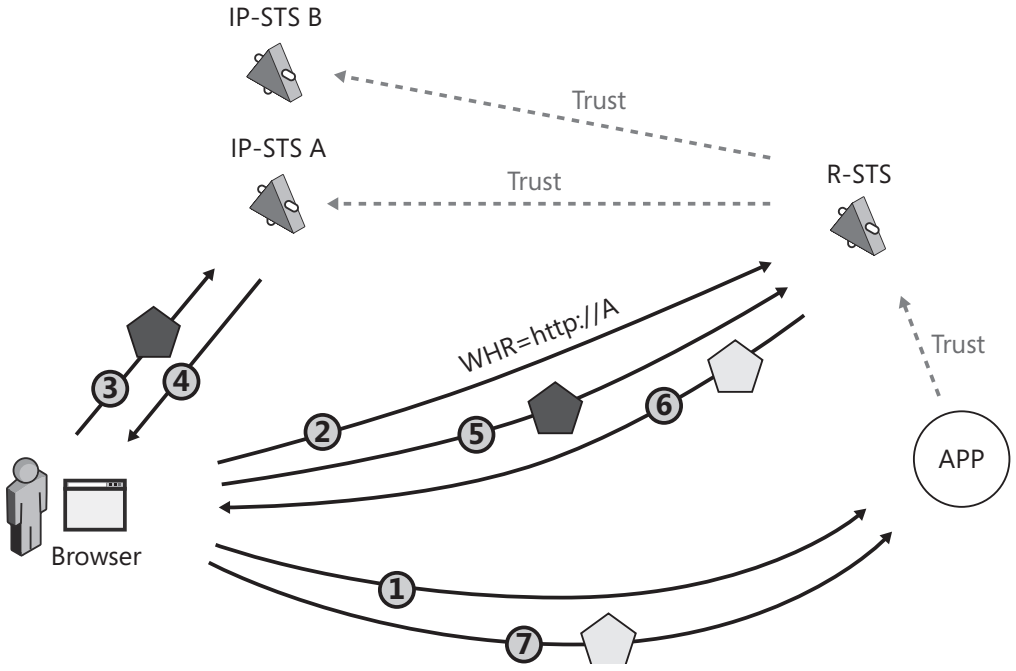

FIGURE 4-9 The Home Realm Discovery problem

1. The user requests a page from App.
2. Because the user is not authenticated; instead, he is redirected to R-STS for authentication. The sign-in message includes a new parameter, *whr*, which indicates A as the home realm for the request.
3. R-STS redirects the request to A.
4. Once the user successfully authenticates with A, he gets back a token.
5. The user gets redirected back to R-STS, which validates the token from A and considers the user authenticated thanks to it.
6. R-STS issues a token to the user, as requested.
7. The user gets back to App with the token obtained from R-STS as required, and the authenticated session starts.

Who injects the *whr* value in the authentication flow? There are at least two possibilities:

- **The requestor** You can imagine a scenario in which the administrator of the organization of IP A gives to all users a link to the RP that already contains the *whr* parameter preselecting IP A. That is a handy technique, which eliminated the HRD problem at its root. Unfortunately, this is not guaranteed to work: this system requires the RP to understand (or at least preserve in the redirect to the R-STS) the *whr* parameter, but WS-Federation does not mandate this to the RP. In fact, RPs implemented via WIF do not support this behavior out of the box (although it's not especially hard to add it).
- **The RP** The RP itself could inject *whr* in the message to the R-STS. Imagine the case in which the RP is one specific instance of a multitenant application. In that case, the *whr* might be one of the parameters that personalize the instance for a given tenant. WIF supports this specific setup on the RP, by allowing you to specify the attribute *homeRealm* in the <*federatedAuthentication/wsFederation*> element of the WIF configuration. The value of *homeRealm* will be sent via *whr* to the R-STS. However, the WIF STS template project knows nothing about *whr* and will just ignore it. Once again, it is not hard to add some handling logic.

The R-STS is the recipient of *whr*. If the execution reaches the FP without having added a *whr,* it is up to the R-STS to make a decision on the basis of anything else that is available in the specific situation and can help decide which IP should be chosen.

Let's once again set up a hypothetical solution in Visual Studio so that you can gain hands-on experience with the flow the scenario entails.

If you still have the solution we used for showing how federation works, right-click on *BasicWebSite_STS*, and again use the Add STS Reference Wizard to outsource its authentication to a new STS. Visual Studio will call the new STS *BasicWebSite_STS_STS1*. The current situation is described in Figure 4-10.

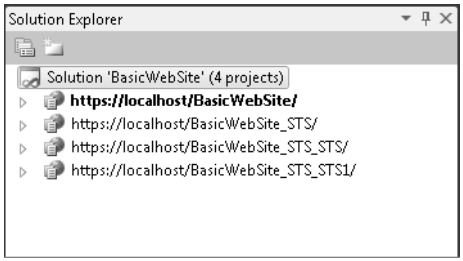

FIGURE 4-10 The sample solution showing how to handle HRD

BasicWebSite trusts *BasicWebSite_STS*, the R-STS of the scenario. *BasicWebSite_STS* now trusts *BasicWebSite_STS_STS1* because with the latest add STS reference, its former trust relationship with *BasicWebSite_STS_STS* has been overridden. The goal here is to establish a mechanism that allows the flow to switch between the two IPs in the scenario (*BasicWebSite_STS_STS* and *BasicWebSite_STS_STS1*) dynamically.

Note With all those STSes looking alike, things might become hard to follow. A good trick for always knowing what is going on is assigning different colors to the background of the *login.aspx* pages of the various STS projects.

The easiest thing to accomplish in the scenario is enabling the RP *BasicWebSite* to express a preference for one IP via *whr*. As mentioned earlier, this can be done easily via configuration:

```
<federatedAuthentication>
        <wsFederation passiveRedirectEnabled="true"
                      issuer="https://localhost/BasicWebSite_STS/"
                      realm="https://localhost/BasicWebSite/"
                      homeRealm="https://localhost/BasicWebSite_STS_STS/"
                      requireHttps="true" />
        <cookieHandler requireSsl="true" />
      </federatedAuthentication>
```

The value of *homeRealm* establishes that *BasicWebSite_STS_STS* should be used for authentication, which is contrary to what the WIF configuration of *BasicWebSite_STS* currently says. That way, it will be obvious whether the system successfully overrides the static settings.

Note As is usually the case for the parameters in *<wsFederation>*, you can do something to the same effect by using the *PassiveFederationSignInControl* and its properties. From now on, I'll omit this note, assuming that in similar situations you'll know that the control alternative is available.

The next step is making the WIF STS template understand *whr*. It is actually simple—it is mainly a matter of intercepting the redirect to the IP and forcing it to go whenever the *whr*

decides. Add to the *BasicWebSite_STS* project a *global.asax* file. Here you can handle the *WSFAM RedirectingToIdentityProvider* event as follows:

```
<%@ Application Language="C#" %>
<%@ Import Namespace="Microsoft.IdentityModel.Web" %>

<script runat="server">
    void WSFederationAuthenticationModule_RedirectingToIdentityProvider
     (object sender, RedirectingToIdentityProviderEventArgs e)
    {
        string a = HttpContext.Current.Request.QueryString["whr"];
        if (a != null)
        {
            e.SignInRequestMessage.BaseUri = new Uri(a);
        }
    }
```

The code could not be easier. It verifies whether there is a *whr* parameter in the query string, and if it there is one, it assigns it to the *BaseUri* in the *SignInRequestMessage,* overwriting whatever value the *BasicWebSite_STS* configuration had put in there. As soon as the handler returns, the WSFAM will redirect the sign-in message to the *whr*—in this case, *BasicWebSite_STS_STS*. And that is exactly as you wanted it.

Note The code here assumes that *whr* carries a network-addressable URI, but per the WS-Federation specification this might not be the case. If the URI is an urn identifier, *BasicWebSite_STS* should look up the actual address in some mapping store.

Having to specify the home realm in the RP configuration might be too static a behavior for many occasions. Fortunately, the *RedirectingToIdentityProvider* event can be easily handled on the RP as well, implementing any dynamic behavior. For example, you can think of maintaining a table of IP ranges where requests might come from, and map them to the corresponding IP addresses. For the sake of simplicity, here I'll show you how to implement the approach when it is the requestor that sends the *whr* up front in its first request to the RP.

If you add a *global.asax* file to *BasicWebSite*, almost exactly the same code as shown earlier will give you the desired effect:

```
<%@ Application Language="C#" %>
<%@ Import Namespace="Microsoft.IdentityModel.Web" %>

<script runat="server">
    void WSFederationAuthenticationModule_RedirectingToIdentityProvider
            (object sender, RedirectingToIdentityProviderEventArgs e)
    {
        string a = HttpContext.Current.Request.QueryString["whr"];
        if (a != null)
        {
            e.SignInRequestMessage.HomeRealm = a;
        }
    }
```

The code here intercepts the execution right before sending back the redirect to the R-STS, and if the original request contained *whr* it ensures that it will be propagated to the R-STS as well. That means you can delete the *homeRealm* attribute in the *BasicWebSite* config, because now you have the ability to express *whr* directly at request time.

Important Keep in mind that all the samples here aim to help you understand the problem, but they do not constitute complete solutions. Handling HRD in practice is not just a matter of complying with the protocol. Instead, it presents various challenges with manageability and maintenance aspects that are beyond the scope of this book and are best addressed by using packaged server-grade products such as ADFS 2.0.

Step-up Authentication, Multiple Credential Types, and Similar Scenarios

The trick of using *RedirectingToIdentityProvider* for steering the request to the STS has many applications that go beyond the HRD problem examined earlier.

One eminent example of this shows up every time the RP needs to communicate some kind of preference about the authentication process the IP should use when issuing tokens to users. It's great that claims-based identity decouples the RP from the authentication responsibilities, but there are situations in which the value of the operation imposes certain guarantees about the strength of the authentication. Imagine a banking Web site or a medical records Web site that gives access to certain operations only if the user is authenticated with a high-assurance method such as X.509 certificates or similar.

As you've grown to expect, WS-Federation has a parameter for that: *wauth*. It is supposed to be attached to *wsignin1.0* messages to communicate to the STS the authentication method preference. Usually, the STS uses that for performing internal redirects to one endpoint that is secured with the corresponding authentication technique, or something to that effect (for example, wiring custom *HttpHandlers* or similar low-level tricks).

Important I won't go into the details here of how an STS should handle *wauth*, mainly because it would do so by leveraging the authentication infrastructures rather than WIF APIs. The main thing to remember on the STS side is that a token will advertise the authentication method that led to its own issuance by the presence of the claim of type *ClaimTypes.Authentication*.

Each RP has its own criteria for assigning a value to *wauth*. Sometimes it is a blanket property for the entire Web site—in which case, it is expressed directly in *<wsFederation>* in the *authenticationType* attribute. At other times, the user is given the chance of selecting (directly or indirectly) from among multiple credential types. In yet another situation, there might be logic that silently establishes whether the current authentication level is enough for accessing the requested resource, or whether the system should step up to a higher level of assurance

and re-authenticate the user accordingly. The last two cases call for a dynamic assignment of *wauth*, which is when reusing what you learned about *whr* and *RedirectingToIdentityProvider* comes in handy for *wauth* too.

Authentication Methods

WIF offers handy constants representing common authentication methods. Once again, they are grouped in multiple collections: *Microsoft.IdentityModel.Claims.AuthenticationMethods* and *Microsoft.IdentityModel.Tokens.Saml11.Saml11Constants+AuthenticationMethods* (shown next). The SDK samples use the first one, whereas the second one is used when communicating with ADFS (though in that case, it boils down to *Password*, *TlsClientString*, and *WindowsString*). In fact, the values in the following AuthenticationMethods are only used in the on-the-wire format specified by SAML. In the general case you won't need them.

```
public static class AuthenticationMethods
{
    // Fields
    public const string HardwareTokenString = "URI:urn:oasis:names:tc:SAML:1.0:am:
HardwareToken";
    public const string KerberosString = "urn:ietf:rfc:1510";
    public const string PasswordString = "urn:oasis:names:tc:SAML:1.0:am:password";
    public const string PgpString = "urn:oasis:names:tc:SAML:1.0:am:PGP";
    public const string SecureRemotePasswordString = "urn:ietf:rfc:2945";
    public const string SignatureString = "urn:ietf:rfc:3075";
    public const string SpkiString = "urn:oasis:names:tc:SAML:1.0:am:SPKI";
    public const string TlsClientString = "urn:ietf:rfc:2246";
    public const string UnspecifiedString = "urn:oasis:names:tc:SAML:1.0:am:
unspecified";
    public const string WindowsString = "urn:federation:authentication:windows";
    public const string X509String = "urn:oasis:names:tc:SAML:1.0:am:X509-PKI";
    public const string XkmsString = "urn:oasis:names:tc:SAML:1.0:am:XKMS";
}
```

The WS-Federation specification lists yet a different set of *wst:AuthenticationType* values, but to be fair it explicitly states that those types are optional.

Claims Processing at the RP

In this final section of the chapter, I cover some of the things you can do with claims at the last minute, when they are already in the RP pipeline and are about to hit the application code.

There is not a whole lot of coding required, especially considering that I already covered *ClaimsAuthorizationManager* in detail in Chapter 2. This section attempts to give you an idea of the intended usage of those extension points and inspire you to take advantage of them in your scenarios.

Authorization

Claims authorization is a fascinating subject that probably deserves an entire book of its own. One thing that puts off the various Role-Based Access Control (RBAC) aficionados is that there is so much freedom and so many ways of doing things. For example, take the coarse form of authorization that can be implemented by simply refusing to issue a token. You can set up rules at the IP that prevent from obtaining a token all the users that are already known not to be authorized to access the application they are asking for. That is feasible for all the situations in which the IP knows enough to make a decision—for example, in cases like Customer Relationship Management (CRM) online, in which users need to be explicitly invited before having access, even when there's a federation in place.

Another obvious place for enforcing authorization is in the R-STS, which might deny tokens on the basis of some cross-organizational considerations. For example, the R-STS used by one independent software vendor (ISV) for managing access to its application portfolio might keep track of how many concurrent users are currently holding active sessions and refuse to issue a new token if that would exceed the number of licenses bought by the IP organization.

The enforcement point that is the closest to traditional authorization systems is the RP itself, which is where *ClaimsAuthorizationManager* is positioned. There are intrinsic advantages to enforcing authorization here. The resources are well known. For example, if the RP is a document management system, the life cycle of documents themselves is under the control of the RP, which can easily manage permissions as well; whereas others (such as the R-STS, or worse still, the IP) would need to be synchronized. Another advantage is the availability of the call itself, although that's easier to see with Web services than with Web sites. If you want to authorize the user to make a purchase according to a spending-limit claim, you need both the claim value and the amount of the proposed purchase: one STS would only see the claim value, as the body of a call plays no part in RST/RSTR exchanges.

The absolute flexibility offered by *ClaimsAuthorizationManager* is both its greatest strength and biggest weakness. Claims-based authorization is really powerful, but at the time of this writing there are no out-of-the-box implementations of *ClaimsAuthorizationManager* or tools and official policy formats for it. You can do everything with it, but you are required to write your own code.

Authentication and Claims Processing

Sometimes it just makes sense to do some claims processing at the RP side. Perhaps you need to make available to the application code information about the user that is known to the RP but not to the R-STS, such as in the case of a user profile specific to the application. Or maybe there are claims you need to see only once, at the beginning of the session, but that you prefer not to make available to the application code.

For doing any of these things, WIF offers you a specific hook in the RP pipeline, which you can leverage by providing your own claims-manipulation logic wrapped in a custom *ClaimsAuthenticationManager* class. *ClaimsAuthenticationManager* works a lot like *ClaimsAuthorizationManager*: you provide your logic by overriding one method (here it's *Authenticate*), and you add your class in the pipeline by adding in the WIF config the element *<claimsAuthenticationManager type="CustomClaimsAuthnMgr"/>*.

In your implementation of *Authenticate*, you can do whatever you want with the principal, including deleting claims, adding claims, or even using a custom *IClaimsPrincipal* implementation. Here is a super-simple example of *ClaimsAuthenticationManager*:

```
public class CustomClaimsAuthnMgr: ClaimsAuthenticationManager
{
  public override IClaimsPrincipal Authenticate(string resourceName, IClaimsPrincipal
incomingPrincipal)
  {
 //If the identity is not authenticated yet, keep this principal and let it redirect to the
STS
    if (!incomingPrincipal.Identity.IsAuthenticated)
    {
      return incomingPrincipal;
    }
    ((IClaimsIdentity)incomingPrincipal.Identity).Claims.Add(
      new Claim(ClaimTypes.Country,"Saturn,"ClaimValueTypes.String,"LOCAL AUTHORITY"));
    return incomingPrincipal;
  }
}
```

In this case, the code simply adds an extra claim to the principal. Note that the issuer is assigned to "LOCAL AUTHORITY." You can use pretty much anything you want here, but you should really avoid using an existing issuer identifier because it is equivalent to pretending to be a legitimate issuer.

Summary

Wow, that was an intense chapter! I hope you had as much fun reading it as I had writing it.

This chapter took a much more concrete approach to WIF programming, leveraging the programming model knowledge you acquired in Chapter 3 to tackle many important problems and scenarios you might encounter when securing ASP.NET applications.

You learned about the distinction between identity providers and Federation Providers, acquiring familiarity with the WIF STS template in the process.

You finally saw applied in practice the sign-in flow studied in Chapter 3, applying it to the case of multiple Web sites and discovering how the underlying structure makes SSO possible. You had a chance to learn how Single Sign-out works, and how to use WIF for implementing

it in a few lines of code. We explored one case of exotic session management, in which the validity is driven by user activity rather than fixed expiration times.

The classic federation case and home realm discovery are now very concrete scenarios for you, and you know what it takes for dealing with them in various situations. In the process of learning this, you also gained familiarity with WIF's object model for claims.

Finally, you had a chance to tie up a few loose ends regarding the use of *ClaimsAuthenticationManager* and *ClaimsAuthorizationManager* for processing claims once they have already reached the RP.

If you develop for the ASP.NET platform, this chapter should have equipped you with all the knowledge you need for tackling the most common problems and then some. For anything not explicitly covered here, you should now be able to investigate and solve issues on your own.

In the next chapter, I'll turn to Web services and explore how WIF and WCF can work together to create safer applications while delivering a killer development experience.