Microsoft

Programming Windows® Identity Foundation

Vittorio Bertocci

Microsoft®

Programming Windows® Identity Foundation

Vittorio Bertocci

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2010933007

Printed and bound in the United States of America.

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental Editor: Devon Musgrave
Project Editor: Rosemary Caperton
Editorial Production: Waypoint Press (www.waypointpress.com)
Technical Reviewer: Peter Kron; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X17-09958

To Iwona, moja kochanie

Contents at a Glance

Table of Contents

Foreword

A few years ago, I was sitting at a table playing a game of poker with a few colleagues from Microsoft who had all been involved at various times in the development of Web Services Enhancements for Microsoft .NET (WSE). Don Box, Mark Fussell, Kirill Gavrylyuk, and I played the hands while showman extraordinaire Doug Purdy engaged us with lively banter and more than a few questions about the product—all of this in front of the cameras at the MSDN studios.

We had each selected a person from the field to play for; someone whom we each thought had made a significant contribution to the success of WSE but hadn't been a direct member of the product team itself. If we won, then our nominee would get a prize, a token of our appreciation for the work that he or she had done. My selection was a guy called Vittorio Bertocci who was working for Microsoft in Italy at the time. I'd never met Vittorio, nor even seen a photo of him, but he was a prolific poster on our internal discussion list, clearly understood the key security concepts for the product including the WS-* protocols, and had even crafted an extension to enable Reliable Messaging despite some of the crude extensibility we had in place at the time. Vittorio was someone worth playing for but, unfortunately, I didn't win.

Time passed, the Windows Communication Foundation (WCF) superseded WSE, and I moved to become the Architect for the Identity and Access team tasked with building a Security Token Service for Windows Server. One day, out of the blue, I got an e-mail from Vittorio to say that he'd moved to Redmond to take on a Platform Evangelist role and asking if we could meet up. Of course I said yes, but what I couldn't have anticipated was that mane of jet-black hair....

Vittorio was deeply interested in the work that we were doing to enable a claims-based programming model for .NET, on top of which we planned to build the second version of our security token service. Over time, these ideas became the "Geneva" wave of products and were finally birthed as the Windows Identity Foundation and Active Directory Federation Services 2.0.

Throughout several years of product development, Vittorio became not only a remarkable spokesperson for the products but a key source of feedback on our work, both from the customers and partners that he met with and from his own direct efforts to use the product. He was instrumental in encouraging me, and the product team, to take on the last-minute task of making WIF run in Windows Azure just in time for PDC 2009 and the product release. Watching Vittorio present a session on WIF is a pleasure—his depth of knowledge and his creative presentation skills allow him to deliver the message on an increasingly important topic despite the fact that it is too frequently tainted with the dryness of the "security" label.

Within the pages of this book, you'll learn how to use the Windows Identity Foundation from someone who is not only a great teacher but is also deeply familiar with the concepts behind the technology itself and who has worked directly with the product team, and myself personally, on a very close basis over the course of the last four to five years.

Vittorio takes you through the terminology and key concepts, and explains the integration of WIF with ASP.NET, Windows Communication Foundation, and Windows Azure, culminating in a speculative look ahead at the scenarios that the product might tackle in a future release. I encourage you, the reader, to think deeply about the concepts here and how you will manage identity in the applications that you go on to build; it's a topic that is becoming increasingly important to both enterprises and the Web community.

Finally, I want to thank Vittorio for his enthusiasm, support, and tireless energy over the years. I have but one final request of him: please get a haircut.

Hervey Wilson
Architect, AppFabric Access Control Service
Microsoft, Redmond
July 2010

Acknowledgments

You create the world of the dream. We bring the subject into that dream and fill it with their subconscious.

—Cobb in "Inception", Christopher Nolan, 2010

Some time ago, a friend asked me what the point was of writing a book when I already have a well-read blog. There are many excellent answers to that question, from the extra reach that a book has to the advantages of reading without having to constantly fight the opportunity costs of not following a link. My favorite answer, however, is that whereas a blog is a one-man operation, a book is the result of the contribution of many people and its value for the reader is proportionally higher. It might be my name on the cover, but the reality is that I stand on the shoulders of many fine people, who I want to acknowledge here. I've been working with identity for the last 8 years or so, interacting with an incredible amount of people; hence, I am pretty sure I'll forget somebody. I apologize in advance.

Peter Kron is a Principal Software Developer Engineer on the WIF team, and the official technical editor of this book. Without his patience, thoroughness, and deep knowledge of WIF, this would have been a much inferior book.

Hervey Wilson is the Architect of the Access Control service. He led the Web Services Enhancements (WSE) team, and he happens to be the one who envisioned Windows Identity Foundation. I've been working with Hervey since 2002, well before I moved to Redmond. At the time, I was still using his WSE for securing solutions for Italian customers. If you believe what Malcom Gladwell says in his book *Outliers: The Story of Success* (Little, Brown and Co., 2008), that you need 10,000 hours of practice for becoming real good at something, nobody contributed more than Hervey to my professional growth in the field of Identity. I am very honored he agreed to write the foreword for this book. Thanks, man!

The **crew at Microsoft Press** has been outstanding, chopping into manageable chunks my long "Itanglish" sentences without changing the meaning and working around my abysmal delays and crazy schedule. (In the last year alone, I handed a boarding pass to smiling ladies 55 times.) Specifically, thanks go to **Ben Ryan** and **Gerry O'Brien** for having trust in me and the book, to **Devon Musgrave** for bootstrapping the project, and to **Rosemary Caperton** for running the project. **Steve Sagman** of **Waypoint Press** led a fantastic production team: **Roger LeBlanc** as Copy Editor, **Thomas Speeches** as Proofreader, and **Audrey Marr** as Illustrator. Special thanks to Audrey for working on really challenging illustrations: you can pull out the needles from my doll now!

Stuart Kwan, Group Program Manager for WIF, and **Conrad Bayer**, GM for the Identity and Access division, have been great partners and supported this project from the very start.

I did most of the writing at night, on weekends, and during vacation time, but at times the book did impact my day job. **James Conard** and **Neil Hutson**, Senior Directors in the Developer and Platform Evangelism group and my direct management chain, have been very patient and supportive of the effort.

Justine Smith and **Brjann Brekkan**, from the Business Group of the Identity and Access Division, have been incredibly helpful on activities that ultimately had an impact on the sample code discussed here.

Todd West, at the time with the WIF test team, is one of the most gifted Web services developers I've ever met. Most of the guidance regarding WIF and Windows Azure in this book and out there is the result of his work.

My good friend **Caleb Baker**, Program Manager on the WIF team, is a never-ending source of insights and useful discussions. He is also the owner of the WIF and Silverlight integration. The Silverlight code samples are all based on his work.

Together with Hervey, the **original WSE team** merged with WIF too. I had a chance to tap their brains countless times. Thanks to **Sidd Shenoy**, **Govind Ramanathan**, **Vick Mukherjee**, **HongMei Ge**, and **Keith Ballinger**.

The entire **WIF team** contributed to this book. Here I'll call a few people out to give you a feeling for the quality of their work. **Daniel Wu** was of great help on sessions; **Brent Schmaltz** was key for helping me understand the inner workings of WIF and WCF; **Vani Nori** and **Vick** devised the way of using WIF with MVC; **Junaid Tisekar** was key for starting the work with WIF and OAuth 2.0; **Shiung Yong** was instrumental in figuring out some parts of the WIF pipeline in the early days of WIF.

Many others in the identity product team contributed through the years: thanks to **Jan Alexander**, **Vijay Gajjala**, **Arun Nanda**, **Marc Goodner**, **Mike Jones**, **Craig Wittenberg**, **Don Schmidt**, **Ruchi Bhargava**, **Sesha Mani**, **Matt Steele**, and **Sam Devasahayam**.

My teammates in the **Windows Azure platform evangelism team** played a key role in keeping me on my toes, and they're simply awesome to hang out with. Thanks to **Ryan Dunn**, **David Aiken**, **Nigel Watling**, and **Zach Owen**. Please delete all the pictures you saved!

The guys at **Southworks**, the company that helped me with practically all the identity samples and labs in the last two years, are fantastic to work with. Many thanks to **Matias Woloski**, **Pablo Damiani**, **Tim Osborn**, **Johnny Halife**, and many others.

Conversations about identity with **Gianpaolo Carraro** and **Eugenio Pace** were extremely valuable, especially the ones related to the P&P guide on claims-based identity led by Eugenio.

Donovan Follette has been the ADFS evangelist for a long time, sharing with me the pains and the joys of the claims-based identity renaissance at PDC08. Even if now he is all cozy in his new Office role, I cannot forget his incredible contribution to bringing identity to the community.

Of course, we would not be even discussing this if **Kim Cameron** had not driven the conversation on the identity metasystem and claims-based identity with the entire industry. Thank you, Kim!

My wife, **Iwona Bialynicka-Birula**, deserves special thanks. She accepted and supported this crazy initiative no matter what, whether it meant skipping beach time while in Maui or coping with insurance agents and contractors after our house got flooded. Without her, not only would you not be holding this book in your hands, I don't know what I would do…. Thank you, darling. I promise: no more books for some time!

Finally, I want to thank *you*: the readers of my blog, who followed faithfully my ramblings for seven years without asking too often about the weird blog name; the participants of the WIF workshops in Belgium, UK, Germany, Singapore, Melbourne, and Redmond, who put up so nicely with my "sexy" accent; and the attendees of the many sessions I gave at events all over the world in the last five years. Without your questions, your critiques, your comments, your compliments, and your longing for understanding, I would have never found the motivation to do this and the other things I do for evangelizing identity. This book is for you.

Introduction

It has been said that every problem in Computer Science can be solved by adding a level of indirection.

You don't have to go far to find examples of successful applications of that principle. Before the introduction of the concept of driver, programs had to be rewritten every time one changed something as simple as the monitor. Before the introduction of TCP/IP, programs targeting a token ring network environment had to be rewritten if the network protocol changed. Drivers and TCP/IP helped to free application developers from the need to worry about unnecessary details, presenting them with a generic façade while leaving the nitty-gritty details to the underlying infrastructure. In addition to making the developer profession a happier one, the approach led to more robust and long-lived software for the benefit of everybody.

For various historical reasons, authentication and identity management practices never really followed the same route of monitors and network cards. Adding "authentication" to your software today still largely means messing with the code of the application itself, writing logic that takes care in detail of low level tasks such as verifying username and passwords against an account store, juggling with X509 certificates or similar. When you are spared from handling things at such low level, which usually means that you took a strong dependency on your infrastructure and your application will be unmovable without substantial rewriting: just like a program from the pre-drivers era.

As you will learn in the first chapters of this book, claims-based identity is changing all this.

Without going too much into details, claims are the means to add that extra level of indirection that eluded the identity world so far. The introduction of open protocols enjoying wide industry consensus & support, the converge toward the idea of a meta-system for identity, the success of metadata formats which can automate many tedious and error-prone tasks created the perfect storm that generated the practices collectively known as claims-based identity. Claims are paving the way for identity and access management to be pushed outside of applications and down in the infrastructure, freeing developers from the need to handle it explicitly while enhancing solutions with welcome extra advantages (such as cross-platform interoperability out of the box).

I have spent full four years working almost exclusively on claims-based architectures with customers and product teams here in Redmond; the model is sound, and it invariably delivers significant improvements against any other authentication system. However, until recently, actually implementing systems according to the model was a painful experience, since it required writing large amounts of custom code that would handle protocols, cryptography, and similar low level aspects.

This all changed when, in October 2008, Microsoft announced the "Geneva" wave of claims-aware beta products: among those there was Windows Identity Foundation, the protagonist of the book you are holding, which was finally released in November 2009.

Windows Identity Foundation (WIF) is Microsoft's stack for claims-based identity programming. It is a new foundational technology which helps .NET developers to take advantage of the claims based approach for handing authentication, authorization, customization and in general any identity-related task without the need to write any low-level code.

True to the claims-based identity promise, you can decide to use WIF to externalize all identity and access control logic from your applications: Visual Studio will make it a breeze, and you will not be required to know any detail about the underlying security protocols. If you want to take finer control of the authentication and authorization process, however, WIF offers you a powerful and flexible programming model that will give you complete access to all aspects of the identity management pipeline.

This book will show you how to use Windows Identity Foundation for handling authentication, authorization and identity-driven customization of your .NET applications.

Although the text will often be task-oriented, especially for the novice part of the book, the ultimate goal will always be to help you understanding the claims based approach and the pattern that is most appropriate for the problem at hand.

Who Is This Book For?

Part I of the book is for the ASP.NET developer who wants to take advantage of claims-based identity without having to become a security expert. Although there are no requirements about pre-existing security knowledge, you do need to have hands-on ASP.NET programming knowledge to proficiently read Part I.

In Part II I shift gear pretty dramatically, assuming that you are an experienced .NET developer who knows about ASP.NET pipeline, Forms authentication, X.509 certificates, LINQ syntax and the like. I often try to add sidebars which introduce the topic if you know little about it but you want to follow the text anyway, but reality is that without concrete, hands-on knowledge of the .NET Framework (and specifically C#) Part II could be hard to navigate. I also assume that you are motivated to invest energy on understanding the "why"s of identity and security.

Identity is an enabling technology, which is never found in isolation but always as a component and enhancement of other technologies and scenarios. This book discusses how to apply WIF with a variety of technologies and products, and of course cannot afford providing introductions for everything: in order to be able to apply the guidance in the various chapters you'll need to be proficient in the corresponding technology. The good news is that the chapters are reasonably decoupled from each other, so that you don't need

to be a WCF expert for appreciating the chapters about ASP.NET. Chapter 3 and Chapter 4 require you to be familiar with ASP.NET and its extensibility model. Chapter 5 is for experienced WCF developers. Chapter 6 requires you to be familiar with Windows Azure and its programming model. Chapter 7 sweeps on a number of different technologies, including Silverlight and ASP.NET MVC Framework, and expects you to be at ease with terminology and usage.

The bottom line is that in order to fully take advantage of the book you need to be an expert .NET and Web developer. On the other hand, the book contains a lot of architectural patterns and explanations which could easily be applied to products on other platforms: hence if you are an architect that can stomach patterns explanations intertwined with code commentary, chances are that you'll find this book a good reference on how claims-based identity solves various canonical problems in the identity and access space.

System Requirements

You'll need the following software and hardware to build and run the code samples for this book:

- Microsoft® Windows 7; Windows Server 2003 Service Pack 2; Windows Server 2008 R2; Windows Server 2008 Service Pack 2; Windows Vista
- Windows Identity Foundation 1.0 runtime
- Windows Identity Foundation SDK 4.0
- Microsoft® Internet Information Services (IIS) 7.5, 7.0 or 6.0
- Microsoft® .NET Framework 4.0
- Visual Studio 2010
- 1.6-GHz Pentium or compatible processor
- 1 GB RAM for x86
- 2 GB RAM for x64
- An additional 512 MB RAM if running in a virtual machine
- DirectX 9–capable video card that runs at 1024 × 768 or higher display resolution
- 5400-RPM hard drive (with 3 GB of available hard disk space)
- DVD-ROM drive
- Microsoft mouse or compatible pointing device
- Approximately 78 MB of available hard disk space to install the code samples

Note that the WIF runtime and the WIF SDK 3.5 are compatible with Visual Studio 2008 and the .NET Framework 3.5 SP2. The March 2010 version of the Identity Training Kit contains most of the samples of the book in a form that is compatible with VS 2008 and the .NET Framework 3.5, however please note that the code in the text refers to VS 2010 and there are small differences here and there.

Code Samples

The code samples for this book are available for download here:

http://go.microsoft.com/fwlink/?Linkid=196688

Click the download link and follow the instructions to save the code samples to your local hard drive.

The code samples used in this book are mostly from the Identity Developer Training Kit, a collection of hands-on labs, presentations, and instructional videos, which is meant to help developers learn Microsoft's identity technologies. It is a self-extracting .EXE. Every lab has its own setup, which will take care of most prerequisites for you. Please follow the instructions on the Welcome page.

Producing the Identity Developer Training Kit is one of the things I do during my day job. Whereas in the book I highlight code snippets to help you understand the technology, in the Identity Developer Training Kit documentation I give step-by-step instructions. Feel free to combine the two approaches as you ramp up your knowledge of Windows Identity Foundation.

The Identity Developer Training Kit is a living deliverable; every time there is a new version of a product I update it accordingly. However, I want to make sure that the code samples referenced in the book will not break. For that reason, I am including in the book code sample archive the current version of the training kit, June 2010, which will always be available, even if I keep updating the training kit in its original download location.

Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site at Oreilly.com.

1. Go to *http://microsoftpress.oreilly.com*.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, under the cover image, you'll see a list of links.
5. Click View/Submit Errata.

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

Part I
Windows Identity Foundation for Everybody

In this part:

Claims-based identity promotes separation of concerns at a level never achieved before in the identity management world. As a result, implementations such as Windows Identity Foundation (WIF) can provide tooling that will enable developers to add authentication capabilities to their applications without the need to become security experts.

The two chapters in this part of the book deliver on that promise: they contain indications that can be understood and applied by any ASP.NET developer, regardless of how much the developer already knows about security. If you are not a security guru, and you don't want to become one, Windows Identity Foundation allows you to tackle the most common authentication and authorization challenges without entering into the gory details of credentials and protocol mechanics. It is so simple that ideally you could even skip most of Chapter 1, "Claims-Based Identity," and go straight to the "WIF Programming Model" section. You would still be able to use WIF for securing your applications in the simplest case, although having the background we provide in Chapter 1 would help you to do so more effectively.

If you are interested in taking finer control of the identity and access management process, Part II, "Windows Identity Foundation for Identity Developers," is for you. However, I suggest that you still glance through Part I, as its characterization of claims-based identity will be required knowledge in Part II.

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.

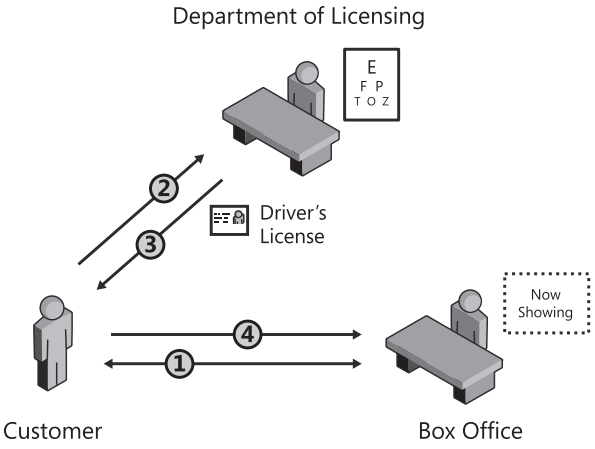


FIGURE 1-1 One identity transaction taking place in real life

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.

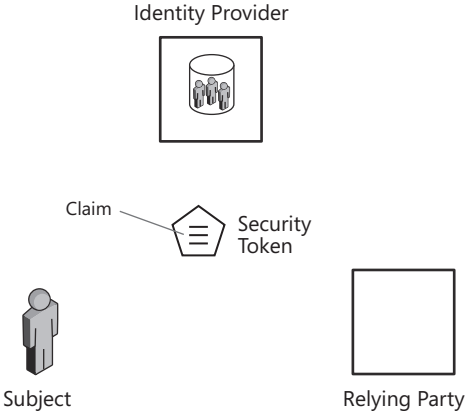


FIGURE 1-2 The main entities in claims-based identity

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.

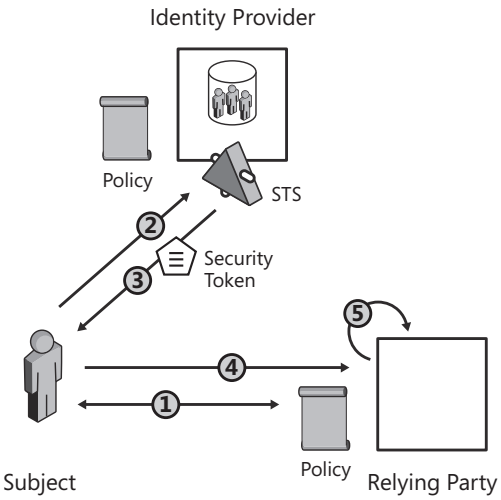


FIGURE 1-3 The flow of the canonical transaction in claims-based identity

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.

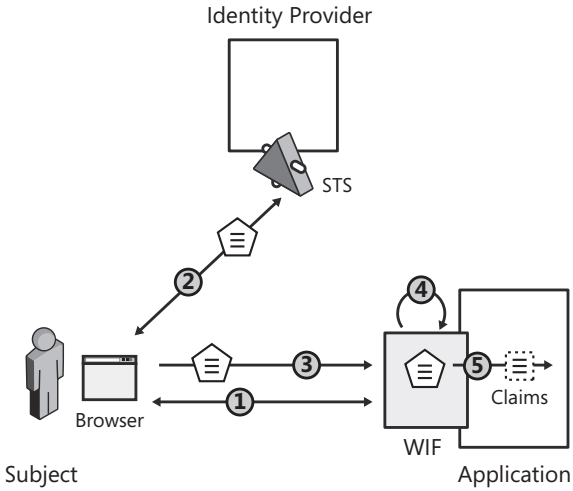


FIGURE 1-4 A simplified diagram of how Windows Identity Foundation takes care of handling authentication for an ASP.NET application

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 2

Core ASP.NET Programming

In this chapter:

By now, you know what claims-based identity is. You also have a firm grasp of how authentication is outsourced to an external entity, such as an identity provider.

That's about all the theory you need to handle the most common cases. This entire chapter is about how to use Windows Identity Foundation (WIF) for managing authentication and authorization for ASP.NET Web sites. As promised, you'll be able to perform those tasks without ever having to look and understand what's going on under the hood.

I'll start by walking you through the most basic task, configuring an ASP.NET Web site to outsource its authentication process to an external Security Token Service (STS). I'll show you that all the old authentication and authorization tricks offered by ASP.NET still apply here, and make sure you understand what configurations you need to apply for them.

After confirming that the existing code still works, we'll venture into new territories. I'll show you how to use claims for performing tasks that were difficult or awkward with traditional approaches. For example, you'll see how you can customize the appearance of ASP.NET pages according to user attributes dynamically received in the form of claims. I'll close the chapter by taking a first look at <*microsoft.identityModel*>, which is the element WIF uses for holding its configuration settings, and giving you a sense of the true power of claims by implementing a simple but effective authorization mechanism that goes beyond roles.

After reading this chapter, you'll be able to use WIF to implement the most common authentication and authorization functions for your ASP.NET Web site. If you are not an expert in identity and security, and not interested in becoming one, this chapter likely covers all you need to know for handling common authentication and authorization cases. If you want to exercise more control over the process, or if your requirements go beyond the most common cases, Part II, "Windows Identity Foundation for Identity Developers," will help you to understand WIF in more depth and show you how to address more complex scenarios.

Externalizing Authentication

In Chapter 1, "Claims-Based Identity," I walked you through a high-level overview of how WIF handles authentication, taking control of the flow before it even reaches the application code. In this section, you'll learn how to take advantage of the tooling provided by WIF for adding Claims-Based authentication to a common ASP.NET application.

I'll spend a few words describing what's inside the various WIF packages, and then I'll walk you through the wizard that performs most of the work.

WIF Basic Anatomy: What You Get Out of the Box

Although it's officially part of the foundational technologies in the Microsoft .NET Framework, WIF is not part of the .NET Framework 4.0 redistributable package. The first version of Windows Identity Foundation was released as a standalone package in November 2009, months before the release of the .NET Framework 4.0. One advantage of this is that you are not forced to migrate to 4.0 to enjoy the benefits of claims-based identity—WIF works with the .NET Framework 3.5 SP1 as well. For the purposes of this book, you'll need to work with the two different WIF packages I describe next.

WIF Runtime

The WIF 1.0 runtime is a Windows Update package (.MSU) that contains everything you need for running an application that takes advantage of WIF features. The .MSU package applies to Microsoft Windows 7, Windows Vista, Windows 2008 R2, and Windows 2008 SP2—of course, both in the x86 and x64 flavors. Windows 2003 SP2, the oldest supported platform, gets the update in classic .EXE format. To preempt a common question: Windows Identity Foundation is not supported on Windows XP.

What do you get with the runtime? The main element is the key WIF assembly, *Microsoft.IdentityModel.dll*, which will land (together with its localization info) in the Program Files\Reference Assemblies\Microsoft\Windows Identity Foundation\v3.5 folder. The runtime package installs a few other things, but for now you can safely ignore them.

You deploy the runtime on the servers that will host your claims-enabled Web sites and Web services. You normally don't need to deploy the runtime on clients unless you have rich clients taking advantage of some specific application programming interface (API) for acquiring tokens. (See Chapter 5, "WIF and WCF," for more details.)

WIF SDK

If you want to develop with WIF in Microsoft Visual Studio, you need to install the WIF Software Development Kit (SDK).

The SDK is a Windows Installer package (.MSI) for all the supported platforms, and it requires the WIF runtime as a prerequisite. There are two different versions available: the WIF SDK 3.5 and the WIF SDK 4.0. Choose WIF SDK 3.5 if your development environment is based on Visual Studio 2008. If you use Visual Studio 2010, you need to install WIF SDK 4.0 instead. Note that the two versions of the SDK will not work side by side; therefore, if you have both versions of Visual Studio installed on your system, you have to choose which one you want to do WIF development with and pick the suitable SDK version.

Besides the samples and documentation you can expect in every developer kit, the WIF SDK contains important tools to help you use WIF in your application. Among the most important are the following:

- **Visual Studio 2008 and Visual Studio 2010 Web site templates** You can use these templates for creating new claims-based Web sites, Web services, or STSes.
- **FedUtil.exe** You can use this tool to modify the *.config* file of a .NET application for externalizing WIF authentication to an STS. (I discuss this tool in more detail in the upcoming sections.)
- **Visual Studio integration features** These tools augment Visual Studio menus with WIF-specific items, provide Microsoft IntelliSense for WIF configuration elements, and so on.

All the code samples described in the book require you to have the SDK installed, so you should install it on your development machines.

Our First Example: Outsourcing Web Site Authentication to an STS

It's finally time to get down to business. You installed the WIF runtime and the WIF SDK. Now you're ready to fire up Visual Studio (either the 2008 or 2010 edition) and write some code. Let's say that you want to create a classic WebForm ASP.NET Web site. Go ahead and create the Visual Studio solution as usual, picking the classic ASP.NET Web site template.

Note I'm not starting from the WIF SDK Visual Studio templates because I don't want to give you the impression that you have to do that to take advantage of claims-based identity. In fact, you can just as easily modify an existing Web site to use WIF for authentication, and that's what I'll demonstrate in this section.

Modify the Web site as you desire, adding the UI elements and the functionalities you need. Because this is the first time you'll see WIF in action, though, you should probably keep things simple.

When you are satisfied with your UI, you can start thinking about authentication. Recall from Chapter 1 that one of the key strategies of claims-based identity is to outsource authentication to an identity provider—namely, to the provider's STS. In a real application, which identity provider you use—your own user directory, a partner's user directory, a custom user store, or anything else—is determined by the business needs addressed by your application. Here you are just experimenting; therefore, what you pick is not especially important. Furthermore, the procedure you follow in WIF is pretty much the same regardless of the IP.

WIF provides you with a straightforward method for linking your Web site to an STS. Right-click the Web site in Solution Explorer in Visual Studio: you'll notice a new menu item, Add STS Reference. If you click it, Visual Studio opens an instance of FedUtil.exe, the tool provided with the SDK, and displays a wizard to walk you through the process of hooking up to an STS.

Figure 2-1 shows the Add STS Reference menu entry in Visual Studio.

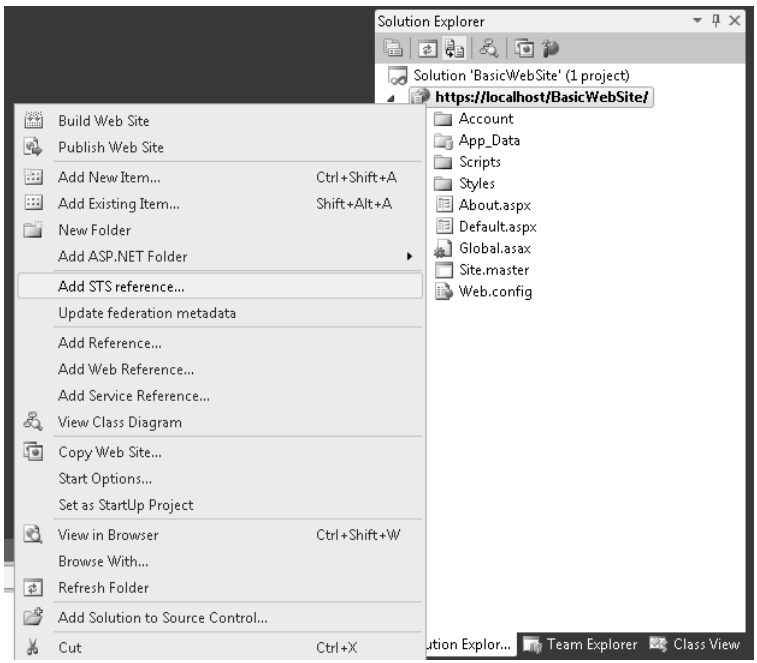


FIGURE 2-1 The Add STS Reference menu entry added to the project context menu by WIF

Using the Federation Utility Wizard

The first page of the wizard determines which application configuration file should be modified and the URI of the application itself. The tool has been invoked from the context of a Visual Studio project; therefore, both fields already exist and are populated. If you had run FedUtil.exe directly from a command line, that would not have been the case.

Figure 2-2 shows the first page of the Federation Utility Wizard.

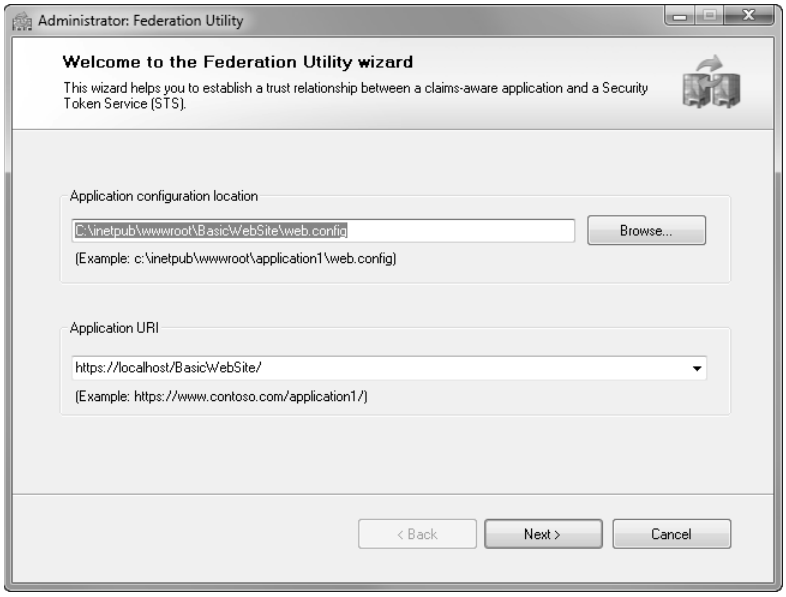


FIGURE 2-2 The first page of the Federation Utility Wizard, which gathers the location of the application configuration file and the URI of the application

The second page gathers more information about the STS to which you want to outsource authentication. Figure 2-3 shows the three STS options offered by the wizard.

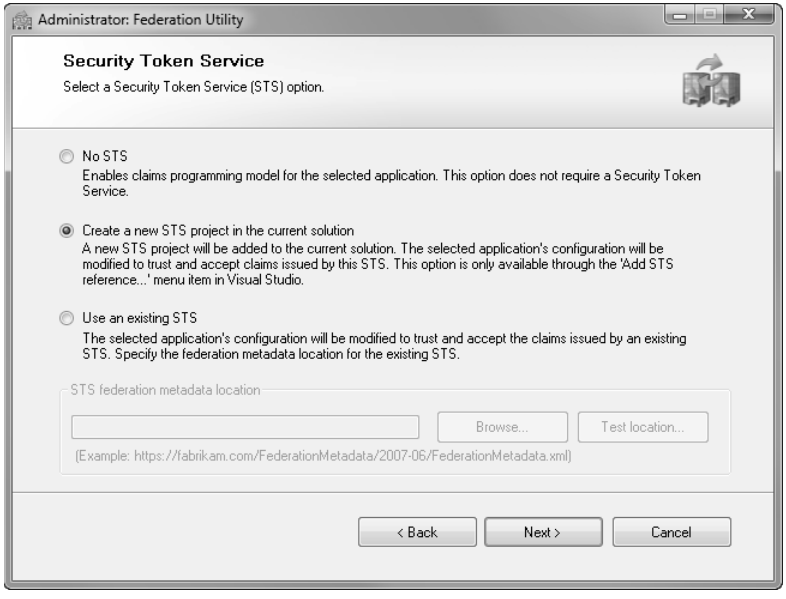


FIGURE 2-3 The Security Token Service page, which you can configure to refer to an existing STS or use to create your own test STS as a new project in your solution

The options, from the bottom up, work as follows:

- **Use An Existing STS** If you already know which STS you want to outsource authentication to, you can pick this option. If you recall the description of the canonical identity transaction in Chapter 1, you'll remember that both the relying party (RP) and the identity provider (IP) published in the form of policies their requirements and capabilities. Here, the wizard asks you to indicate the policies of the target STS, which are normally published in what is called a *federation metadata document*. WIF can use the information contained in that document to automatically configure your Web site to correctly involve the STS in an authentication transaction and consume the resulting data. Sometimes, IP owners decide not to publish federation metadata for their STS. In that case, WIF can still take advantage of those policies for externalizing authentication; however, you won't be able to use this wizard and you'll be forced to directly edit WIF's configuration.
- **Create A New STS Project In The Current Solution** When developing a Web site, you are often in the position of not knowing which identity provider the Web application will use in production. At other times, the identity provider will not be reachable from your environment, you'll not have test credentials available, and so on. For all cases in which the STS of the IP you want is not available at development time, you can choose this option to have WIF generate a test STS within the current Visual Studio solution. The test STS does not really implement authentication, although it will offer the proper placeholders if you choose to add it and it will issue two hard-coded claims (Name and Role). A test STS can be extremely handy at development time.
- **No STS** If you choose this option, WIF does not really take care of authentication for you; however, it will enable the claims-based programming model in your application. In practice, choosing this option makes *IClaimsPrincipal* and *IClaimsIdentity* instances available to your application code by injecting arbitrary values so that you can experiment with the claims object model.

Note I'm not a big fan of the "No STS" option because it can create confusion without really simplifying things.

You don't have any IP available in this phase; therefore, the best option you have is to simply simulate one by generating it. If you select the Create A New STS Project In The Current Solution option and click Next, you'll land on the summary page displayed in Figure 2-4.

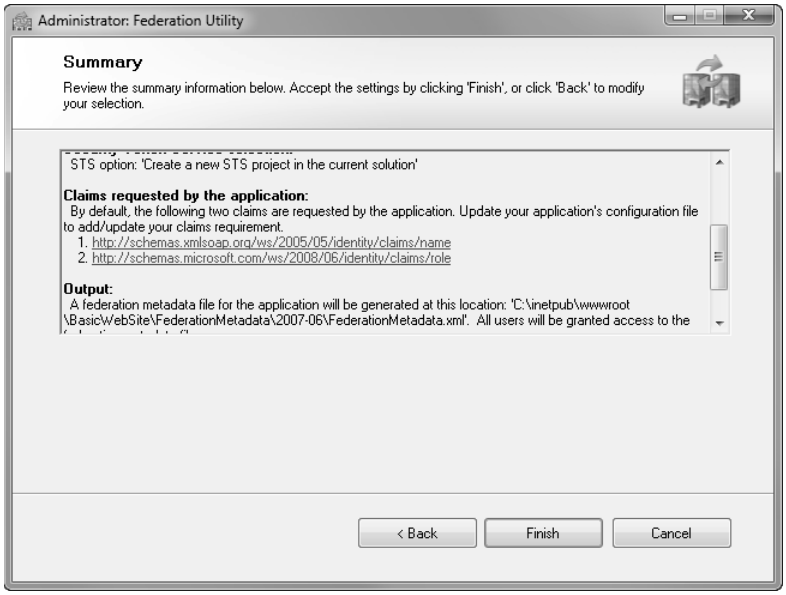


FIGURE 2-4 The final page of the Federation Utility Wizard, which summarizes the choices you made so that you can review them before applying the settings to the application configuration

The final page of the wizard briefly summarizes the options you picked. The only piece of information I want to point out is the Claims Requested By The Application section. Here, the wizard informs you that your application will be getting the claims Name and Role. As I mentioned earlier, those are the hard-coded claims that are issued by default by an automatically generated STS.

Note The wizard generates a policy document for your application, called *FederationMetadata.xml*, which contains all the security requirements that other parties need to know to enter in a relationship, including which security protocol you expect (in this case, the automatically generated one), which claims your application requires (in this case, Name and Role), and so on. This document can be used by an IP for getting your application onboard as one the IP is willing to issue a token for. The information provided allows the IP to perform the task via tooling, without forcing administrators to go through lengthy and error-prone data entry procedures. Claims-based identity is good for IT professionals, too!

After you click Finish, you'll notice that a new Web site named *<yourwebsitename>_STS* appears in your solution: that Web site is the STS that the wizard generated for you.

Figure 2-5 shows the new structure of your solution. For now, you can happily forget that the new project is there. I'll get back to it in a moment.

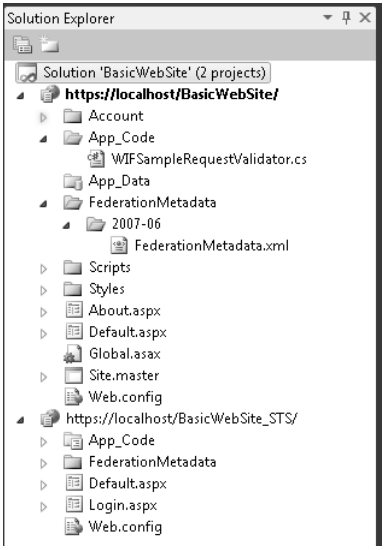


FIGURE 2-5 The new Web site that the Federation Utility Wizard added to your solution, which contains the test STS your original Web site now refers to

Another thing you might notice is that the *web.config* file of your Web site has been modified: the wizard added all the necessary elements for inserting WIF in the authentication pipeline. Later in the chapter, we'll take a quick glance at some of those elements; in Part II, I'll describe most of them in detail.

I hope you were not expecting anything fancier than the coverage in the preceding paragraphs, because you're already done setting up authentication for our sample application. Simply press F5 and see what happens.

The browser opens at your Web site address, but instead of rendering the first page it redirects you to the STS authentication page. If you try to type the direct address of any page, the result will be the same: you'll always be redirected to the STS. Figure 2-6 shows the UI of the default authentication page created within the autogenerated STS project. Note that no real authentication takes place; the STS always issues the same token regardless of the user name and password provided.

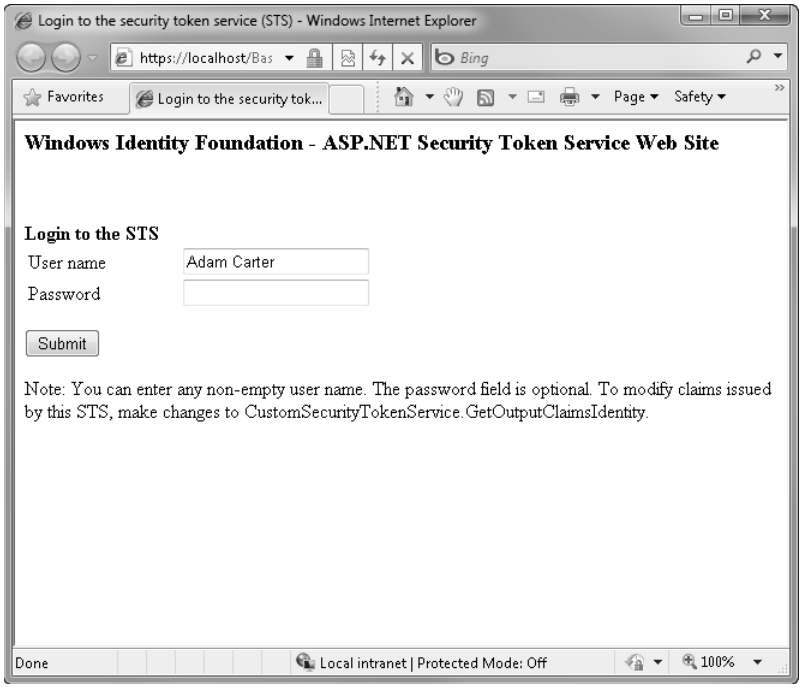


FIGURE 2-6 The authentication page of a test STS automatically generated by WIF

The automatically generated STS does not try to authenticate you. It just provides an endpoint that correctly handles claims-based identity protocols and issues test tokens for you so that you can externalize authentication from your application at development time. If you click the Submit button, the STS fakes a successful authentication: it generates a token, generates proof of successful authentication with the STS, and sends the token to your application. Here, WIF verifies that the incoming token is a valid one, that it is indeed coming from the intended identity provider, and runs through many other validation criteria. If all of those criteria are satisfied, WIF considers the incoming user authenticated. It then creates a session cookie, and from that moment on requesting a page will succeed.

Allow me to summarize: you started with one existing ASP.NET Web site; you went through a three-step, point-and-click wizard, and now your application has an authentication mechanism in place. Case closed. You didn't write a single line of code, and you didn't have to see any angle brackets. And apart from the words "claim" and "STS," you didn't have to understand any security mumbo-jumbo. It cannot get much simpler than this, can it?

The hands-on lab "Web Sites and Identity" (C:\IdentityTrainingKit2010\Labs\WebSitesAndIdentity\Source\Ex1-ClaimEnableASPNET) *exercise 1, tasks 1 and 2 walk you through the process of adding an automatically generated STS to your Web site and using it for authentication.*

Using an Actual Identity Provider

I'm sure a lot of you at this point will want to protest. The preceding solution does not perform any actual authentication; we just added an extra redirect and a button click. Did we really solve anything?

Yes, we did. One of the main reasons traditional access control technologies are difficult to use is that they force application developers to handle the mechanics of authentication—credential types, user stores, and the like—by themselves. The wizard you ran on the application broke the rules of that game by configuring your Web site to rely on the STS to take care of user authentication. At risk of being pedantic, I'll say this: this approach allows you to focus your undivided attention on your application functions rather than on the details of the authentication machinery.

What probably bothers you, if anything, is that this specific STS does not do a very good job of authenticating users. In fact, it does not even try. That's a valid concern; however, it is indeed a nonissue. Using an autogenerated STS allows you to use claims-based identity in your code while you develop your application, at a time you usually don't need to authenticate actual users. When your application is ready to be deployed, you can easily run through the same Federation Utility Wizard again and, this time, choose the Use An Existing STS option and pick the true identity provider you want to use in production. Again, this can be anything: your own directory, a partner, or anything similar. Products such as Active Directory Federation Services v2 (ADFSv2) are able to expose STS endpoints and metadata documents that can be easily consumed by the WIF tooling. Those STS endpoints do authenticate users before issuing tokens, with all the strength offered by Active Directory, but they still talk to your application with the same protocol that was used by the wizard-generated STS. Thanks to the fact that you relied on the same protocol at development time, the application code does not need to change. In fact, you don't even need to be the one who changes the STS reference from development to production: the Visual Studio wizard you explored in the earlier section is available as the standalone tool fedutil.exe, which can be used as applications are rolled into production. As a developer, you might never even need to know which STS the application will end up using.

Of course, somebody *eventually* needs to worry about making authentication happen and to make decisions about whether your application should trust an external identity provider or run its own, which protocols to use, which credential types are adequate, and so on. The main interaction between you and that person will be her giving you the URI of the metadata document of the identity provider you should use. If your job role entails only the development of the application functions, adding STS references is all you need to do.

Sometimes, you are the lucky winner of having to make both the application development and authentication decisions. That is indeed quite common, especially in smaller shops where the difference between developers and administrators is blurred. In such cases, you need

to make some decisions about authentication and protocols, which means you'll probably stick around for Part II of this book. However, even in these cases the value of separation of concerns is obvious: claims-based identity offers you an architecture in which the various elements are as autonomous as possible. Even if the responsibility of implementing the mechanics of authentication and authorization is on your shoulders, it is far better to be able to scope those down into well-known elements of the architecture, such as STSes, rather than having identity management logic shredded and spread across multiple applications of different form and function.

*The hands-on lab "Web Sites and Identity" (*C:\IdentityTrainingKit2010\Labs\WebSitesAndIdentity\Source\Ex1-ClaimEnableASPNET*) exercise 3 shows you how to use the Federation Utility Wizard for trusting an external identity provider. The STS used in the lab is an ADFSv2 instance available on the Internet, and it has been preconfigured to accept token requests for the application URI in the lab.*

Authorization and Customization

Outsourcing authentication to an external identity provider takes the credential management off your back, and that's a great improvement. However, authentication is not all you need to worry about in access management. In fact, that's usually just the beginning: the inordinate amount of attention it has enjoyed so far in literature is more due to the difficulty of implementing it and the fine details that need to be provided, rather than its relative importance in the development life cycle. The *authorization* phase typically needs to include building an awareness of the resources and actions you want to protect. Therefore, you can expect it to have more touch points with your logic. Now that your application is finally decoupled from the mechanics of authentication, which would otherwise require a disproportionate amount of attention because of its many details, the aforementioned ratio will become more evident in your development activities.

Let's take a quick moment to learn the ropes of authorization. The time you invest will pay off quickly—within a page or two, in fact—because I'll soon describe how Basic authorization is implemented with Windows Identity Foundation.

The idea behind authorization is straightforward. You have resources and actions that should be available to a restricted group of people and denied to everybody else. The job of your authorization logic is to establish that the current user belongs to either the first or second category. How do you do that? You ask questions about the user—questions such as, "Are you at least 21 years old?" or "Do you belong to the Remote Debuggers group?"—and the answers determine whether you'll grant or deny access to the requested page, Web service, document, or whatever resource you are protecting.

The traditional approaches to authorization developed many techniques, way too many for me to mention here. The two elements that are very useful for characterizing most approaches are the following:

- **Caching the answers to questions you have about users** If you own the credentials store used for authenticating the user, chances are that you own the only representation of users in the system. If you want to know something about one user, there's no other place to find the answer than in your system. (This of course excludes situations in which you leverage a directory. In that case, you don't own the credentials store.) That means you also have the responsibility of collecting information about users that will help you make authorization decisions—information such as group memberships, names, and the like. That's a handful of extra tables in your user database and more headaches for you in trying to keep the user data safe from unauthorized access. It also means extra logic for acquiring the information about the user, usually at sign up time, and devising strategies for verifying the truthfulness of the collected data in relation to the expected reliability of the application. For example, a free Web mail service can be more forgiving about forged names and addresses than software for submitting tax declarations.

  With claims-based identity, the questions can be posed directly to the identity provider at authentication time. Thus, you no longer need to cache answers about your users. You can still decide to store things about the user that are specific to your application and that you can't expect an external identity provider to know (for example, the state of the UI elements during last session), but the point is that you are no longer forced to. Figure 2-7 shows a schema that summarizes the differences between a classic approach and a claims-based approach in the sense I just described.

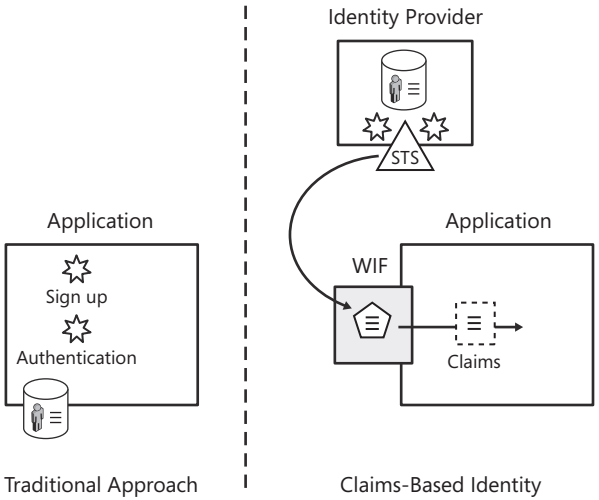


FIGURE 2-7 A comparison of the traditional approach and the claims-based identity approach

In traditional approaches to authentication, such as the one adopted by the ASP.NET membership provider, your solution maintains stores for both credentials and attributes that are associated to users. At sign-in time, the attributes of the user are fetched from the store and used for authorization. With claims-based identity, none of that is necessary: the application can receive all the information it needs in the form of claims. Therefore, you no longer need to maintain local stores and the associated logic.

- **Relying on groups and roles** This is the introductory section of the book, so I won't bother you with detailed definitions of Role-Based Access Security (RBAC), which is the most commonly encountered approach to authorization. In this approach to authorization, the category of people who have access to a resource gets explicitly defined; we call such categories *roles*. In this case, the question your authorization system must answer becomes extremely straightforward. If the users who have access to a management portal must belong to the Manager role, you ask, "Do you belong to the Manager role?". The answer gives you an unambiguous clue about whether access should be granted or denied. Of course, claims make the modeling of such questions easy. One nice property of this approach is that you are not always forced to create an explicit category every time you need to protect a resource. If your application lives in the context of one organization, often you can leverage existing user groups as access categories. (Differences between roles and groups do exist, but they are too subtle to be useful here.)

  Roles are extremely powerful, especially in business environments in which the organization and job functions map nicely to application privileges. In fact, you can expect roles to be widely used in claims-based identity. On the other hand, you often need to address scenarios in which the explicit category definition required by the roles approach is less than ideal.

  Consider the simple case in which you want to restrict access to a document only to people at least 21 years old. Sure, you could create a 21YearsAndOlder role and assign users to it accordingly. However, that forces you to constantly shuffle people into the role as they come of age. The age check is a simple case, but you can easily imagine sophisticated authorization criteria for which explicit category creation is so complicated that the solution is impossible to put in practice. Luckily, with claims you can model the question "Are you at least 21?" directly, without the need to create and maintain an artificial category, and to enforce authorization decisions accordingly.

In the remainder of this chapter, you'll discover how to use Windows Identity Foundation to handle authorization in your application and take practical advantage of the aforementioned principles.

ASP.NET Roles and Authorization Compatibility

The .NET Framework has offered role-based authorization capabilities since well before the advent of claims-based identity. The *IsInRole* method in the *IPrincipal* interface, mentioned in Chapter 1, is a good example of that. The good news is that Windows Identity Foundation provides a mechanism for interpreting certain incoming claims as roles, so if you want to keep using *IsInRole* for authorization you can.

By default, Windows Identity Foundation interprets as a role all the values of claims of type *http://schemas.microsoft.com/ws/2008/06/identity/claims/role*. That also happens to be one of the claims issued by default by the STS and is automatically generated by FedUtil.exe and the Federation Utility Wizard in Visual Studio. If you followed the procedure in the earlier section, you can verify right away that the *IsInRole* integration works.

If your scenario calls for a different claim type to be interpreted as a role, you can take advantage of specific mechanisms for telling WIF to change the claim type to role mapping. However, explaining how to do it would require you to know a bit more about how WIF configuration works. I'll return to this subject in Part II and show you how to pick an arbitrary claim type as a role.

Note WIF applies a similar mechanism for assigning the *Name* property of the current *IIdentity*. The default behavior is to pick the value of the claim type *http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name*. Again, WIF offers a way to change the default claim type for *Name* that is analogous to what you would do for changing the *Role* mapping.

ASP.NET offers many more possibilities for taking advantage of role information for authorizing access to resources. One of the most interesting is the <*authorization*> element in the <*system.web*> block, which you can use for settings such as restricting the visibility of all pages to a specific role:

```
<system.web>
   <authorization>
     <allow roles="Manager"/>
     <deny users="*"/>
   </authorization>
</system.web>
```

The preceding construct will work handsomely with WIF, including when it is enclosed in a <*location*> block for restricting its scope to a specific application page or path. If you have existing code of this form, selecting Add STS Reference on your Web site will not break it; as long as the STS provides the correct role information, it will keep working as expected.

Some authorization features you find in ASP.NET today won't work with WIF when you are developing a relying party application. For the most part, these include all the APIs

that assume you are taking on the burden of handling user authentication and role stores yourself. All the functions on the Provider tab of the ASP.NET Web Site Administration Tool are a good example. You expect to receive the identity information of the user in a security token and its claim collection; therefore, you no longer need account and role management tools. Note that the need to handle account and credential authentication does not disappear, it is merely shifted to the rightful owner. Although you might no longer be on the hook for it, the developer on point for creating an identity provider application still needs to maintain accounts and authenticate requests before issuing tokens. In that case, the tools offered by ASP.NET for handling accounts and roles can be a useful aid for authenticating token requests and providing roles as claim values.

*The hands-on lab "Web Sites and Identity" (*C:\IdentityTrainingKit2010\Labs\WebSitesAndIdentity\Source\Ex1-ClaimEnableASPNET*) exercise 1, tasks 4 and 5 demonstrate how you can take advantage of the* <authorization> *element with WIF.*

For good coverage of ASP.NET authentication and authorization features, refer to Chapter 17 of Programming Microsoft ASP.NET 3.5 *by Dino Esposito (Microsoft Press, 2008).*

Claims and Customization

Now that I reassured you that your existing ASP.NET code still works with WIF, it's time to move to the new capabilities that Windows Identity Foundation offers.

In Chapter 1, you learned about *IClaimsIdentity* and *IClaimsPrincipal*, and about how to extract from them the value or values of a certain claim. The most trivial way in which you can use that information is directly from the code-behind class of your pages. Whenever your logic needs to know something about the user, you just query *IClaimsPrincipal* for the desired value. Although that strategy can be applied in every situation, its effects can be particularly evident when used to drive the user experience. The ASP.NET *LoginName* control is a primitive example of an identity-driven user experience. (It displays on the page the *Name* of the authenticated user.) Of course, with claims you can do so much more.

Imagine that your Web application offers a function that can be performed only by a user older than 21. If you are handling authorization in the right way, any other user attempting to access that function will be denied access. Even though that guarantees the access control policies are enforced, in certain situations this might not be the preferred user experience. If the application interface is designed to be intuitive and guide the user through the desired execution flow, it could be best to hide in the first place the UI elements that trigger functions for which the current user is not authorized. That way, the user would not even be aware of the option, and this would lead to better attention management and usability.

Note As is often the case for considerations about user experience, sometimes the exact opposite approach is the best course of action. For example, the access denied might be the result of the user not having complied with some prerequisite, and getting an error message might be a way of making him aware of the mistake. Also, the user interface layer is not always able to predict whether the user will be allowed to perform a certain function because the authorization logic might live elsewhere. This can happen, for example, when calling an external Web service. This is not an exact science. However, the fact that you can't address all cases does not mean you should not act upon the cases you can address.

The code for handling that UI customization can be straightforward. Let's take a look:

```
protected void Page_Load(object sender, EventArgs e)
{
    IClaimsIdentity claimsIdentity = Thread.CurrentPrincipal.Identity as IClaimsIdentity;

    DateTime birthDay = DateTime.Parse(
        (from c in claimsIdentity.Claims
         where c.ClaimType == ClaimTypes.DateOfBirth
         select c.Value).FirstOrDefault());


    HyperLink1.Visible = (DateTime.Today >= birthDay.AddYears(21));
}
```

The first line of code retrieves *IClaimsIdentity* from the current context. The following line queries the claims collection for the birth date claim value and puts it into a *DateTime*. The rest is standard time-mangling code, which is finally used for making the *HyperLink1* element (which presumably points to the protected resource) invisible if the user is younger than 21. Like most examples I present here, this is a trivial case. You can imagine much more sophisticated criteria in which multiple claims can be used to steer, customize, and enrich the user experience provided by your application.

*The hands-on lab "Web Sites and Identity" (*C:\IdentityTrainingKit2010\Labs\WebSitesAndIdentity\Source\Ex1-ClaimEnableASPNET*) exercise 1, task 4 demonstrates how to use claim values to influence various aspects of a Web site*.

Driving the behavior of ASP.NET controls according to identity information can be easily hidden behind custom controls, so that logic like that described in this section can be implemented without having to write any code. The ClaimsDrivenModifierControl sample, available at *http://code.msdn.microsoft.com/ClaimsDrivenControl*, demonstrates how to take advantage of WIF to do exactly that.

A First Look at <*microsoft.identityModel*>

The coding techniques I described in the preceding section are perfectly adequate for user interface customization, but if you're dealing with authorization you are usually better off trying to make access decisions *before* the execution reaches your application. Even if in some scenarios the required authorization logic is deeply entrenched with the execution flow and can't be easily factored out (making the preceding techniques a good solution), in general you gain a lot by externalizing authorization. If the authorization logic lives outside of the application code, you gain in efficiency because unauthorized requests fail early in the pipeline. You are able to change access policies at deploy time and give administrators greater control and autonomy, with obvious gains in manageability. Finally, because all the updates to access control policies can be applied without touching the code or requiring recompilation, your applications are more resilient to change. I have shown how WIF can leverage the <*authorization*> element mechanism, which can indeed be used for changing access policies at deploy time. Nonetheless, after having seen what can be done with claims, you can be excused if using roles alone starts to look a bit like a blunt tool.

Windows Identity Foundation does offer the means for implementing claims-based authorization. Those means heavily rely on WIF's extensibility mechanisms rather than on tools available out of the box. As a result, I'll need to dig a bit deeper into WIF's structure before enabling you to reap the full advantages of claims-based authorization. Namely, I'll give you a cursory glance of <*microsoft.identityModel*>, the main configuration element used by Windows Identity Foundation. (In Part II, I'll cover the topic at length.) Once you have a better idea of how WIF configuration works, I'll show you how to take advantage of the extensibility model for adding claims-based authorization to your application.

WIF and the *web.config* File

At the beginning of the chapter, I mentioned that the main effect of FedUtil.exe, whether called directly or via the Federation Utility Wizard from Visual Studio, is to alter the application's *web.config* file by adding all the necessary elements for inserting WIF in the authentication pipeline. In less than 50 words, here's a description of the process: *FedUtil* sets the <*authentication*> mode to *None* and inserts its own *HttpModule*s in the application pipeline. Those modules will come up at every request and take over, enforcing whatever authentication and authorization steps WIF has been configured to perform.

The parameters influencing the behavior of the *HttpModule*s live in the <*microsoft.identityModel*> element, which was also added by *FedUtil*. Although for the most common cases you can rely on the settings established by the wizard and never even see a single angle bracket, from time to time you'll need to edit <*microsoft.identityModel*> to achieve the effect you want.

Let's take a look at a typical WIF *config* element, as added to the *web.config* file by FedUtil.exe:

```
<microsoft.identityModel>
   <service>
      <audienceUris>
         <add value="https://localhost/SimpleWebSite/"/>
      </audienceUris>
      <federatedAuthentication>
         <wsFederation passiveRedirectEnabled="true"
                       issuer="https://localhost/SimpleWebSite_STS/"
                       realm="https://localhost/SimpleWebSite/"
                       requireHttps="true"/>
         <cookieHandler requireSsl="true"/>
      </federatedAuthentication>
      <applicationService>
         <claimTypeRequired>
            <claimType type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
                       optional="true"/>
            <claimType type="http://schemas.microsoft.com/ws/2008/06/identity/claims/role"
                       optional="true"/>
         </claimTypeRequired>
      </applicationService>
      <issuerNameRegistry
               type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
       ...">
         <trustedIssuers>
            <add thumbprint="0E2A9EB75F1AFC321790407FA4B130E0E4E223E2"
                 name="https://localhost/SimpleWebSite_STS/"/>
         </trustedIssuers>
      </issuerNameRegistry>
   </service>
</microsoft.identityModel>
```

There's quite a lot of stuff going on there: it makes you grateful that the tooling takes care of it most of the time! In fact, after you know the function of the main elements, the code block starts to look much less intimidating. For example:

- *FederatedAuthentication* stores the parameters that define the authentication settings, including the protocol you want to use (*ws-federation* in this case), which identity provider is trusted to authenticate users (the STS at *https://localhost/SimpleWebSite_STS/*), whether the authentication legs of the request should mandatorily happen on HTTPS, and so on.
- *IssuerNamedRegistry* lists the X.509 certificates of all the trusted identity providers so that WIF can pick the right certificate for verifying the signature of incoming tokens and refuse all the tokens signed with keys that are not recognized.

You get the point: all those elements control a specific aspect of how WIF processes requests.

The block of code shown earlier contains a minimal set of elements, the bare minimum for outsourcing authentication to the STS available at the *https://localhost/SimpleWebSite_STS/*

address. In fact, <*microsoft.identityModel*> can contain many more elements, specifying details ranging from how tokens are deserialized to how to use incoming claims for granting or denying access to application resources. In the last section of this chapter, we take a look at the latter task using claims-based authorization. In Part II, you'll get a more comprehensive view of all other options.

Basic Claims-Based Authorization

Windows Identity Foundation does not supply a ready-made API for claims-based authorization. The approach followed by WIF is to provide you with a place in the processing pipeline where you can inject your own authorization logic, which will be executed before reaching your application's code. As you might have guessed at this point, there is a specific configuration element in <*microsoft.identityModel*> that you can use for weaving in your authorization logic. Furthermore, WIF offers some base classes you can use for giving structure to your authorization code and maximizing its potential for reuse. Even though this model requires some coding up front, it gives you complete freedom to implement whatever criteria you want while still relying on a declarative model that is configuration driven.

Let's learn how that is done by exploring one detailed example. You can expect this last section to be a bit more challenging than what you have seen so far. Consider it a stepping stone toward the more advanced content of Part II.

Let's say that you want to be able to perform the age check I described in the "Claims and Customization" section, but that instead of implementing it in the code-behind class you want to be able to assign the age check declaratively to arbitrary pages, directly in the *web.config* file.

The first step you need to do is encapsulate the age check, which is your authorization logic, in a subclass of *ClaimsAuthorizationManager*, which is a class provided by WIF in the *Microsoft.IdentityModel.Claims* namespace. Let's take a look at the base class:

```
public class ClaimsAuthorizationManager
{
    public ClaimsAuthorizationManager();
    public virtual bool CheckAccess(AuthorizationContext context);
}
```

Ignore the constructor for now. The key element here is the *CheckAccess* method. WIF invokes it toward the end of the processing pipeline, after all other verifications are done. If the method returns *true*, the execution is passed to the application code; otherwise, the caller receives an unauthorized error and the call will end. The decision depends on some internal data of our choosing. In the current example, this is the age threshold, which can be represented as a member of your *ClaimsAuthorizationManager* subclass. The other decisive factor is the content of the *context* parameter, representing the characteristics of the current call

(such as a resource being requested), claims of the current caller, and so on. Let's take a peek at the *AuthorizationContext* type definition:

```
public class AuthorizationContext
{
    private Collection<Claim> _action;
    private IClaimsPrincipal _principal;
    private Collection<Claim> _resource;
    public AuthorizationContext(IClaimsPrincipal principal,
                                Collection<Claim> resource,
                                Collection<Claim> action);
    public AuthorizationContext(IClaimsPrincipal principal,
                                string resource,
                                string action);
    public Collection<Claim> Action { get; }
    public IClaimsPrincipal Principal { get; }
    public Collection<Claim> Resource { get; }
}
```

The *Resource* collection represents the entity that is being requested. You can expect this collection to often be constituted by a single element, such as the URI of a page or Web service.

The *Action* collection represents the operation being attempted on the resource. Again, you can expect this to be a one-element collection containing something like a Web service *SOAPAction* method, an HTTP verb, and similar.

The *Principal* is the *IClaimsPrincipal* that the former stages in the WIF pipeline assigned to the caller, which can be used for querying the user claim collection.

The *CheckAccess* method will compare those parameters, representing the call data, with the access policy that you have assigned to the resource.

Now that you understand the base structure, all you need to do is create your own subclass of *ClaimsAuthorizationManager*. However, before jumping into coding the custom class, it is useful to spend a moment to understand the way in which you use the configuration elements for inserting it into the WIF pipeline.

The schema of the block <*microsoft.identityModel/service*>, which has been briefly examined in the preceding section, contains a <*claimsAuthorizationManager*> element of the following form:

```
<xs:element name="claimsAuthorizationManager">
    <xs:complexType>
      <xs:attribute name="type" type="xs:string" use="required" />
    </xs:complexType>
</xs:element>
```

The only thing that is mandatory in this schema is the *type* attribute, which refers to your *ClaimsAuthorizationManager* subclass. Assuming that you called your class *AgeThresholdClaimsAuthorizationManager*, you need to add a reference to it in the *<microsoft.identityModel/service>* element:

```
...
</applicationService>
   <claimsAuthorizationManager
            type="ClaimsBasedAuthorization.AgeThresholdClaimsAuthorizationManager" />
<federatedAuthentication>
...
```

Furthermore, you need to tell ASP.NET that from now on you are also interested in handling authorization. The Federation Utility Wizard enlists WIF to handle authentication and session management by adding two *HttpModule*s in the ASP.NET pipeline. You need to add a third one, the *ClaimsAuthorizationModule*. Below is one example of module entries from the *<system.webServer>* element. Note that depending on the IIS version and pipeline configuration, you may need to add the same modules under *<system.web/httpModules>,* instead.

```
  <system.webServer>
    <modules>
      <remove name="ScriptModule" />
...
      <add name="WSFederationAuthenticationModule" type="Microsoft.IdentityModel.Web.
WSFederationAuthenticationModule, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" preCondition="managedHandler" />
      <add name="SessionAuthenticationModule" type="Microsoft.IdentityModel.Web.
SessionAuthenticationModule, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, Publ
icKeyToken=31bf3856ad364e35" preCondition="managedHandler" />
      <add name="ClaimsAuthorizationModule" type="Microsoft.IdentityModel.Web.
ClaimsAuthorizationModule, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, Public
KeyToken=31bf3856ad364e35" preCondition="managedHandler" />
    </modules>
```

Note Chapter 3 will dig very deep in the WIF *HttpModules* topic, if you want more details please refer to it.

Something is clearly missing at this point. Not only do you want WIF to be aware of your authorization code, you also need to be able to assign your authorization policies to the resources you intend to protect. This is typically done by adding to the *<claimsAuthorizationManager>* block a list of *<policy>* elements, which represent the set of your resources and the authorization criteria you want to enforce upon access. In your specific case, you want to limit access to a page only to users who are older than a certain

threshold age. A possible form that the <*claimsAuthorizationManager*> element might assume is shown here:

```
. . .
</applicationService>
   <claimsAuthorizationManager
            type="ClaimsBasedAuthorization.AgeThresholdClaimsAuthorizationManager">
      <policy resource="/ClaimsEnableWebSiteEx01_End/SecretPage.aspx" action="GET">
         <claim claimType="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/
dateofbirth"
                minAge="21" />
      </policy>
   </claimsAuthorizationManager>
<federatedAuthentication>
...
```

Note that the <*policy*> element, its attributes, and child elements are not part of WIF. In this specific example, they represent a possible syntax you can choose for representing your authorization policies in the *web.config* file.

The *resource* attribute represents the local URI of the page you want to protect; the *action* attribute is the operation for which you want the authorization logic to be triggered. The element *claim* contains a claim type that must be present in the request when the resource is accessed—in this case, the user's birth date—and *minAge* represents the threshold below which access to the resource should be denied.

The good news is that now you have all you need for describing directly in the *config* file your authorization policies. The not-so-good news is that, considering that you can put whatever you want in them, you have to handle the deserialization of the <*policy*> elements on your own code. However, in a moment, you'll see that is not too bad.

It's finally time to take a look at the code of *AgeThresholdClaimsAuthorizationManager*:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Web.Configuration;
using System.Xml;
using Microsoft.IdentityModel.Claims;
using Microsoft.IdentityModel.Configuration;

public class AgeThresholdClaimsAuthorizationManager : ClaimsAuthorizationManager
{
    private static Dictionary<string, int> _policies = new Dictionary<string, int>();

    public AgeThresholdClaimsAuthorizationManager(object config)
    {
        XmlNodeList nodes = config as XmlNodeList;

        foreach (XmlNode node in nodes)
```

```
    {
        XmlTextReader rdr = new XmlTextReader(new StringReader(node.OuterXml));
        rdr.MoveToContent();
        string resource = rdr.GetAttribute("resource");
        rdr.Read();
        string claimType = rdr.GetAttribute("claimType");
        if (claimType.CompareTo(System.IdentityModel.Claims.ClaimTypes.DateOfBirth) != 0)
                throw new NotSupportedException("Only birthdate claims are supported");
        string minAge = rdr.GetAttribute("minAge");
        _policies[resource] = int.Parse(minAge);
        }
    }

    public override bool CheckAccess(AuthorizationContext pec)
    {
        Uri webPage = new Uri(pec.Resource.First().Value);
        if (_policies.ContainsKey(webPage.PathAndQuery))
        {
            int minAge = _policies[webPage.PathAndQuery];
 DateTime birthDay = DateTime.Parse(
     (from c in claimsIdentity.Claims
      where c.ClaimType == ClaimTypes.DateOfBirth
      select c.Value).FirstOrDefault());


 if ( DateTime.Today < birthday.AddYears( 21 ) )
        {
            return false;
        }
        }
        return true;
    }
}
```

The only private member in the class is a static collection of policies, represented as *<string, int>* couples.

The constructor gets as input the *config* element you saw earlier. The *_policies* collection is filled up with a list of resources paired with their age threshold. If a policy requires a claim type that is different from the birth date, the constructors return a *NotSupportedException*. Note that for simplicity the preceding code does not really do anything about the *Action*, but you can easily imagine how it could be handled.

The *CheckAccess* method is where the true authorization check takes place. The first line extracts the page being requested from the *AuthorizationContext.Resource* in the context passed in the parameter *pec*. If the page has been assigned a policy, the corresponding age threshold is retrieved and compared to the current user's age as indicated by the birth date claim in the *AuthorizationContext.Principal* member of the context. This is the same code described in the customization section. A user younger than *minAge* will cause *CheckAccess* to return *false*, and vice versa.

Note You should look at the code here with a critical eye. *CheckAccess* simply assumes the correct claim type is present in the context; it does not perform any error check if that assumption is incorrect. Furthermore, this implementation of *CheckAccess* grants automatic access to all the resources that are not mentioned in the policy. Your scenario might be better served by the opposite heuristic—that is, to automatically deny access unless an explicit policy entry for the resource is provided.

The code I use in this book is designed to help you understand how to use the technology; for the sake of clarity, it will often not contain checks and practices that are absolutely necessary for production systems. For an excellent reference on secure coding, refer to *Writing Secure Code: Practical Strategies and Proven Techniques for Building Secure Applications in a Networked World* by Michael Howard and David LeBlanc, (Microsoft Press 2002).

This is all you need for adding custom authorization to your application. You can do something as simple as the example here, or you can use the same model to encapsulate your existing authorization code in a reusable format. The policy schema to use is completely up to you; therefore, you can literally implement whatever format or standard you like.

Note that the technique I described in this section still qualifies as externalization of identity management code, because *ClaimsAuthorizationManager* is not proper application-specific code. You can write *ClaimsAuthorizationManager* classes once and reuse them across many applications. In fact, if you are a developer who focuses on the user experience or business processes, you might never even see the code of a *CheckAccess* class. You'd likely receive ready-to-use assemblies and policy schema reference documents directly from your colleagues who take care of security, the same people from which you'd receive the address of identity providers mentioned in the first section of the chapter. And you would just be expected to reference and use the classes provided. Life is good!

Summary

This chapter gave concrete information about using WIF for addressing common authorization and authentication challenges in application development.

At this point, the text likely has covered what you need to know about WIF for handling authentication (by leaving to others the honor of taking care of the fine details) and for taking advantage of identity information to drive the behavior of your application (by using claims directly in your code or the various *config*-based authorization mechanisms).

You saw the main tooling that WIF offers for externalizing authentication, and how to go through the Federation Utility Wizard to automatically enable and configure WIF on ASP.NET Web sites. The same wizard can be used both for generating local test STSes at development time and for creating a reference to a production identity provider. I also presented some key differences between the two situations.

This chapter devoted some time to describing authorization in general terms, refreshing the definition of role-based access control, and pointing out the main differences between traditional methods and claims-based identity. Most of the existing role-based authorization functionalities offered by the .NET Framework are preserved when using WIF—namely, you saw how *IsInRole* and the <*authorization*> element still work as expected.

As anticipated by reading Chapter 1, you saw how WIF exposes claims to the application developer via the *IClaimsPrincipal* and *IClaimsIdentity* interfaces. I showed you how to take advantage of that information for influencing the behavior of your application—for example, by deciding, based on the age of the current user, whether elements of the UI should be visible.

You had your first look at <*microsoft.identityModel*>, the main *config* element used to store WIF settings. Finally, you learned how to encapsulate your own authorization logic and extend the WIF configuration schema to enable sophisticated claims-based authorization for your application resources.

If you do not intend to become a security expert, at this point you can put down the book and start working. Enjoy your new skills and your new found freedom from the nitty-gritty details of old-fashioned authentication. I hope you had fun reading, and I wish you the best of luck in your future endeavors!

But you should keep reading if, on the other hand, any of the following applies to you:

- You are not satisfied with running a wizard and want to know what really happens under the hood.
- You want to use claims-based identity with Windows Communication Foundation (WCF).
- You need to address advanced scenarios such as delegation, flowing identity across multiple tiers, and the like.
- You run an identity provider, and you want to understand how an STS works.

If you recognize yourself in any of the preceding cases, read on. In Part II, "Windows Identity Foundation for Identity Developers," I'll dig much deeper into WIF's structure and usage. Whereas Part I was all about enabling you to add authentication and authorization capabilities to your applications without being exposed to the underlying complexity, Part II will focus on helping you to understand how WIF works and how to take advantage of its powerful features in more advanced scenarios.

Part II

Windows Identity Foundation for Identity Developers

In this part:

Part II constitutes the bulk of the book, and it's meant to provide you with a solid reference on using Windows Identity Foundation to address a wide range of scenarios. If you are serious about learning how to use WIF and take control of the identity and access aspects of your solutions, this part is for you.

Chapter 3, "WIF Processing Pipeline in ASP.NET," dissects the WIF programming model, using the ASP.NET sign-in process as an opportunity to parade most of the classes and settings that will be used later in the book for dealing with all the other scenarios. Chapter 4, "Advanced ASP.NET Programming," explores the canonical solutions to some of the classic problems in access management and federated identity for Web applications, such as home realm discovery, single sign-out, and various others. Chapter 5, "WIF and WCF," examines the relationship between WIF and WCF, showing how the two libraries augment each other and going into the details of how WIF integrates the WCF programming model and introduces some changes in its practices. Chapter 6, "WIF and Windows Azure," provides a few considerations about how to use WIF in Windows Azure. Finally, Chapter 7, "The Road Ahead," touches on some scenarios that the first version of WIF does not address out of the box and gives some indication about how to make integration happen nonetheless.

Part II goes beyond the sheer description of WIF's developer surface, providing solid architectural explanations and an industry/historical context that will empower you to truly grasp the nature of the issues and solutions. Without that deep understanding, you'd run the risk of seeing the solutions here as a list of recipes: the reality is that you'll always need to slightly tweak things to accommodate the specific needs of your own situation, and this book aims precisely at equipping you to make the right choices when the time comes.

Chapter 3

WIF Processing Pipeline in ASP.NET

In this chapter:

Welcome to Part II of *Programming Windows Identity Foundation*! Whereas in Part I, "Windows Identity Foundation for Everybody," I shielded you as much as possible from the details of how things really work and focused on showing you how to get the job done, from here on I'll assume you are willing to partake in the wonders of identity programming internals. The extra effort on your part comes with a great reward: by understanding how things work, you'll be able to exercise near-complete control of every aspect of authentication and authorization, while still conserving the good properties granted by externalizing identity management. Note that if you would rather solve common scenarios without getting too deep into Windows Identity Foundation (WIF) extensibility, feel free to skim through the chapter just to pick up the terminology and then jump directly to Chapter 4, "Advanced ASP.NET Programming."

If you plan to read the chapter in detail, let me give you a little advice about how to get the most from it.

The first section, "Using Windows Identity Foundation," discusses the use of WIF beyond the basics and spells out why you need to acquire a deeper knowledge of the underlying protocol and WIF's request processing to fully master the product.

The second section, "WS-Federation: Protocol, Tokens, Metadata," is all about describing the protocol that WIF implements out of the box for handling identity in ASP.NET applications. Apart from the occasional forward reference to WIF classes, this section could be easily found in a book about competing technologies: the ideas described are absolutely platform-independent and can be applied to any stack that implements WS-Federation. In addition to describing the protocol in itself, the text takes a deeper look at important concepts such as tokens, metadata documents, and what functions they satisfy. If you're already familiar with such concepts and the terminology—for example, if you've already worked with WS-Federation on other platforms—feel free to skip to the WIF-specific section "How WIF Implements WS-Federation."

"How WIF Implements WS-Federation" describes how WIF takes advantage of the extensibility mechanisms offered by ASP.NET for making the WS-Federation magic happen. I'll list all the main *HttpModule*s, handler classes, and relevant elements that constitute WIF's base arsenal, describing the order in which they will activate in the context of serving a request. Note that although I'll detail which module performs what function, I'll assume you already learned about what the function itself entails in the first section of this chapter. If you ever find yourself unsure about what a function such as validating a token really means, do not hesitate to backtrack to the explanation in the second section before moving forward.

"WIF Configuration and Main Classes" revisits the <*microsoft.identityModel*> section briefly mentioned at the end of Chapter 2, "Core ASP.NET Programming," explaining both its default form and its key elements. The section also provides an annotated list of the main classes you encountered in the section "How WIF Implements WS-Federation," and comments on their intended use.

After you read this chapter, you'll have a solid understanding of how WS-Federation works. Moreover, you'll learn about WIF's intended use as a programming tool. You'll be able to enumerate all the main elements WIF uses in addressing the ASP.NET scenario, describe the functions performed by every relevant class, and decide where you need to change a configuration or add custom code for influencing how identity is processed. The chapter focuses on describing how all those elements work together in the default case; subsequent chapters will provide examples of how to tweak things to adapt the flow to the needs you'll encounter in common scenarios.

Using Windows Identity Foundation

Windows Identity Foundation has been designed to integrate with ASP.NET or Windows Communication Foundation (WCF) applications, and it provides various out-of-the-box mechanisms for achieving that. Although in principle nothing prevents you from ignoring those mechanisms and using WIF classes from scratch to build your own pipeline processor, in practice it's a good idea to keep the existing integration methods as a frame of reference and obtain whatever variation you need by leveraging the powerful WIF extensibility model. In this chapter, you'll study the ASP.NET integration model; however, many concepts are directly applicable to the WCF scenario.

You normally take advantage of Windows Identity Foundation for securing applications in four main ways: using the SDK tooling, manipulating configuration elements, serving events, and subclassing items. Note that this has more to do with influencing how the authentication process takes place. The use of the claims object model takes place after this and is common to all methods.

Using the SDK Tools

The easiest way to take advantage of WIF is by using the tools offered by the SDK. Chapter 2 provided an example of the approach in the section "Our First Example: Outsourcing Web Site Authentication to an STS." Other valid examples are applications created by using the WIF SDK templates as a starting point.

As you already discovered in Part I, the tooling approach as it stands today empowers the nonexpert to take advantage of claims-based identity. The system supplies reasonable defaults for addressing the most common cases and, provided that a suitable identity provider is available, that is normally enough for getting authentication out of the way.

If your needs go beyond the simplest case, however, things can rapidly get more complex.

Manipulating the Configuration Elements

Nearly every aspect of WIF can be tweaked and influenced via configuration. In fact, most of the work required by the tooling consists of adding the appropriate elements you use to the *web.config* file. If you want to change the way in which WIF behaves, often all you need to do is add the correct details to the default configuration. Here you run into the first catch: to know what elements you need to tweak, you need to have deeper insight into how things work under the hood. For the simplest settings, that might mean having some awareness, however vague, of how the underlying protocol works. More advanced settings require you to know more intimately how WIF processes requests—namely, the classes it uses and the subfunctions they perform—so that you can competently change parameters or substitute the default elements with your own. Somewhere in the middle, you can find settings that enable you to manipulate claims before they reach the application itself. Although you can proficiently take advantage of those mechanisms without really knowing much about the protocol or the request processing details, you are required to combine configuration with the development of custom classes. (The *ClaimsAuthorizationManager* class you studied at the end of Chapter 2 is a good example of this approach.)

In other words, as soon as you want to go beyond the tooling, you need to learn more about how WIF works. You don't need to know everything, but the more you know the more in control you are.

Serving Events

Some of the classes involved in the WIF pipeline raise events at crucial moments of the request processing sequence. Providing handlers for those events can be a powerful means of addressing more advanced scenarios. Of course, you need to know what you are doing in those handlers, and that often implies understanding the protocol that WIF is implementing.

Subclassing

WIF functionality has been factored in many classes, which can be tweaked and recombined in many different ways. It is not uncommon to encounter situations in which the out-of-the-box functionality is not enough, and the solution warrants rolling out your own custom version of existing classes. WIF is designed to gracefully accommodate this situation; in fact, that is at times the standard way of providing a given functionality. (Again, consider how you are required to provide your own custom implementation of *ClaimsAuthorizationManager* if you want to implement claims-based authorization in your application).

Subclassing can be very easy or very complicated. In any case, it entails having solid knowledge of the base class role in WIF's pipeline so that the custom implementation provides all the methods and properties that the rest of the pipeline expects from it.

As you can see, if you want to move beyond the basics you need to break through the façade put in place for protecting the uninitiated and deal with some of the underlying complexity. The remainder of this chapter will help you to understand the protocol that WIF uses in the ASP.NET case and the process it follows for handling it.

WS-Federation: Protocol, Tokens, Metadata

In Chapter 2, I happily dismissed the entire process of securing a Web site with WIF as "outsource authentication to an identity provider." That works well if you are OK with all the default behaviors and you just want authentication out of your way, but it is less than satisfying if you want to adapt things and address the demands of less common scenarios. A more accurate portrayal of the truth would be that WIF offers one general-purpose infrastructure for plugging in arbitrary identity and access management protocols in front of ASP.NET applications, and that it provides on top of that infrastructure a ready-to-use implementation of the WS-Federation protocol. Most of the WIF tooling, classes, and extensibility mechanisms you use in the context of ASP.NET will somewhat be influenced by the choice of WS-Federation as the out-of-the-box protocol for Web applications. It is safe to say that to truly understand what makes WIF tick in ASP.NET you need to learn a bit about how WS-Federation works. Note, this is still a far cry from being forced to deal with authentication mechanics directly as is necessary with pre-claims approaches: it is just a matter of fleshing out some details of how the outsourcing of authentication to a Security Token Service (STS) truly takes place. You don't even need to ever read the protocol specification. For that, the information in this section will likely give you more information than you need.

In the upcoming "WS-Federation" section, I'll establish some scope: what WS-Federation is, what its relationship is with other protocols, and what aspects of it we will focus on.

In the section "The Web Browser Sign-in Flow," I'll describe the sequence of events constituting *sign-on*, the most common WS-Federation transaction you'll encounter when working with WIF.

The sections "Metadata Documents" and "A Closer Look at Security Tokens" will dig deeper into important artifacts that play a key role in WIF programming, describing their structure and the role they play in the context of identity and access management.

The good news is that, if you set aside the obvious differences in the syntax definitions,the main ideas and concepts featured by WS-Federation are very similar to what you would find in comparable protocols, such as Security Assertion Markup Language protocol (SAML-P) and WS-Trust. (I'll say more about those next.) The investment you make in this section will pay off not only in later chapters, but for all your future identity-related development and design.

WS-Federation

It is almost a law of nature: the more the IT world moves toward increasingly distributed and connected solutions, the more painful the difficulty of communicating across silos is perceived to be. In response to that and other market pressures, back in 2001 many key industry players embarked on the endeavor of developing a set of communication protocols and languages that would guarantee easy interoperability across platforms. Those protocols and languages are based on Simple Object Access Protocol (SOAP) Web services and follow a naming scheme of *WS-<function>*, where *WS* stands for "Web services" and "<function>" indicates the specific aspect of the communication the protocol takes care of (for example, security, addressing, policy, and so on). The specifications of those protocols came to be collectively known as "WS-*" (pronounced "WS-star"). The most important of the WS-* specifications became de jure industry standards with various standardization entities (OASIS, W3C, and others). As of today, practically every major platform offers WS-* programming stacks and many released products rely on WS-* for implementing key features.

The WS-* specifications tackled all major aspects of communication, dictating how to augment the bare-bones capabilities offered by SOAP with capabilities such as transport independent addressability (WS-Addressing), discoverability (WS-Discovery), and many others.

Note The rationale behind using multiple individual specifications regarding a single aspect—as opposed to a monolithic, all-encompassing, boil-the-ocean prescriptive document (such as CORBA, a popular approach in the late 90s/early 2000s)—was that with a modular set of protocols you should be able to mix and match just the features that are needed in your scenario and ignore the rest. It is interesting to observe how a decade later the REST community is pressuring WS-* to reduce complexity.

It is safe to say that security has received the most attention, being one of the first capabilities to be addressed in a specification (WS-Security) and eliciting the highest number of documents (WS-Security, WS-Trust, WS-SecurityPolicy, WS-Addressing, token profiles, and so forth) dealing with different aspects of secure messaging.

WS-Trust is one of the most interesting standard WS-* protocols covered in this book, given that it introduces the concept of STS and specifies the messages that are used for requesting, issuing, and renewing security tokens. In practice, WS-Trust gives prescriptive guidance about how to implement via SOAP messages legs 2 and 3 of Figure 1-3 in Chapter 1, "Claims-Based Identity." WIF makes extensive use of WS-Trust for dealing with Web services scenarios: I'll give more details about WS-Trust in Chapter 5, "WIF and WCF."

WS-Federation is another extremely important specification. Whereas WS-Trust describes primitive operations such as requesting and issuing tokens, regardless of the reasons for which they are requested, WS-Federation specifies a language that can be used for expressing how to address the security needs of actual, complex scenarios. Examples of those scenarios include how to give access to resources across different security realms, how to handle distributed sign-out operations, and similar. With perhaps a bit of oversimplification, you can think of WS-Federation as a list of recipes that describe how to combine WS-Trust and other WS-* specifications for tackling (more) realistic scenarios.

Web services are extremely powerful, and I have always been an enthusiastic fan of the approach. (Here's a true fact: at the moment of writing this book, the license plate on my car is "WS-STAR.") However, even I have to admit that they are not viable at all times. Not every client is able to perform the sometimes complex cryptographic operations that WS-Trust and other protocols require. In industry literature, you'll often find a client that is WS-* capable defined as *active*, whereas one that is not WS-* capable is known as *passive*. The most notable example of a passive client is the Web browser. Given the universal diffusion of Web applications, it is unthinkable to leave the browser-based scenario unaddressed: mechanisms have to be developed that allow the key capabilities defined for the Web service cases to be applied in passive client scenarios as well.

I hope you are not superstitious, because such mechanisms can indeed be found in section 13 of the WS-Federation 1.2 specification. Section 13 describes how to use the tools available in HTTP 1.1 (GET and POST verbs, redirects, cookies) for requesting and obtaining tokens, sending them to relying parties and in general handling sign-in, sign-out, and similar operations from a Web browser.

Although WS-Federation describes a wide range of scenarios and services, it is safe to say that among developers it is most well known for what it defines in section 13 and that is usually what people are referring to when they use the term. In other words, "WS-Federation" often stands for "authentication and authorization in the passive case," despite the fact that the specification itself covers so much more. This book is no exception. In fact, I am going to cover an even smaller subset of section 13: in the next section, I'll walk you through the browser sign-in procedure only, as that is pretty much all you need to know to understand how WIF handles ASP.NET requests.

Protocols Salad

The panorama of the available protocols can be very confusing, as you might have guessed by reading this section. The family of specifications I described here, WS-*, is defined by open standards that enjoy wide consensus and are implemented in multiple released products. However, they are not the only open standards in use in the market. SAML-P is also an open standard (ratified by OASIS) and enjoys wide adoption. Although in the past the choice between the two approaches was an either-or decision, today an increasing number of products implement both WS-* and SAML-P. Active Directory Federation Services 2.0 (ADFS 2.0) is one such product: it extends Active Directory with claims-based identity capabilities that can be interchangeably expressed via WS-Trust, WS-Federation, and SAML-P.

For your reference, the following table shows the list of protocols and standards that WIF uses or supports out of the box.

Protocol or Specification	Supported?
WS-Federation	Yes
WS-Trust	Yes
WS-Security	Yes
WS-Trust	Yes
WS-SecurityPolicy	Yes
SAML protocol	No
WS-Addressing	Yes

At the time of this writing, WIF supports only WS-Federation and WS-Trust out of the box, but it is perfectly plausible that future versions might add SAML-P capabilities. Although SAML-P is not supported, WIF can use SAML 1.1 and 2.0 tokens with WS-Federation and WS-Trust.

The Web Browser Sign-in Flow

In Chapter 2, in the "Externalizing Authentication" section, you saw an example of how WIF handles authentication for a Web site: the browser requests a page, WIF redirects the request to the STS authentication page, the user enters credentials, and then the user gets redirected back to the desired page and is finally granted access. From what you read in Chapter 1, you have a high-level idea of *why* that happens: the Web site needs a token from the STS for authenticating the user, and the sequence of redirects ensures that it gets one. It's now time to focus on *how* that happens: WS-Federation defines a specific HTTP syntax for driving the sign-in process, which I am going to describe in some depth.

Note The purpose of this section is not to make you a protocol wizard but to give you a frame of reference for understanding the main WIF processing pipeline in ASP.NET. I am going to omit many details that are mentioned in the specification; on the other hand, I'll fill in some details (such as session cookies) that the specification does not cover. Finally, I'll simplify the terminology to adapt it to the sample scenario. The bottom line is that I describe *one* specific scenario that happens to use WS-Federation, but the specification itself is far more general. If you are interested in understanding the full scope of WS-Federation in the Web-browser scenario, please refer to section 13 of the specification itself (available at *http://docs.oasis-open.org/wsfed/federation/v1.2/cd/ws-federation-1.2-spec-cd-01.html*).

Figure 3-1 shows an example of the WS-Federation sign-in sequence.

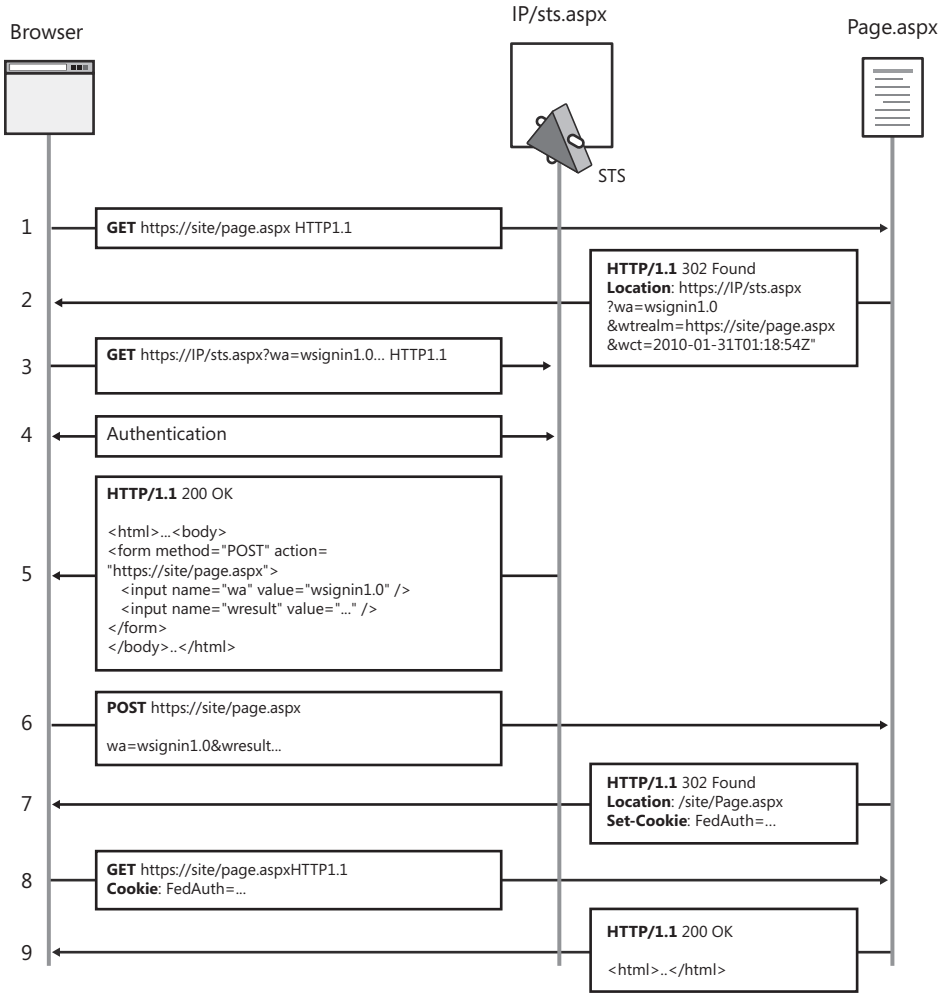


FIGURE 3-1 The Web-browser sign-in sequence according to WS-Federation

Although the desired action is the simplest of all, requesting a Web page via a GET verb, the limited set of operations supported by a Web browser and HTTP force the system to engage in a series of redirects to flow identity information among the various parties. WS-Federation defines a number of parameters with well-known semantics that drive the redirection process and help every party in the sequence determine which role they are playing and what information they should process. Once you know what every step achieves, however, you'll discover that it is much simpler than the first look at Figure 3-1 might lead you to believe.

Let's examine all the steps of the process.

1. The browser requests the page *page.aspx* from the Web site *https://site*. As you might recall from earlier chapters, in industry literature the Web site is defined by the relying party (RP) in the transaction.
2. The Web site determines that the requestor is not authenticated. The Web site trusts the identity provider (IP), which exposes a WS-Federation-compliant STS at *https://ip/sts.aspx*. Therefore, the Web site takes steps for redirecting the browser to the IP so that the user can authenticate there. It returns a 302 Found HTTP code, which indicates that the resource should be requested at another temporary address. The *Location* header contains the address of the STS at which the user should authenticate, plus various WS-Federation-specific *querystring* parameters that indicate the operation that should be performed. Next I give a quick description of the parameters used in the sample sequence. Note that various other WS-Federation parameters could have been used here for extra functionalities. I'll list some of them before the end of the section.
   - **wa** indicates the kind of operation that should be performed. In this case, the value is *wsignin1.0*, which will induce the destination of the redirect (the STS) to interpret the message as a sign-in request. The parameter *wa* must be included; otherwise, the destination would not know what to do with the message.
   - **wtrealm** represents the security realm for which a token is required—that is, the Web site itself. If you think of a token as a payment check, *wtrealm* represents the named payee. Just as a check can be cashed only by the named payee, a token should be consumed only by the Web site that required it. When requesting a token, the presence of *wtrealm* is mandatory. I'll discuss this aspect in more depth in the "A Closer Look at Security Tokens" section.

Note In this case, *wtrealm* is also used to keep track of the address to which the sequence must eventually return. Please note that this is a convention used by the samples in the WIF SDK, but it is not mandated by the specification or WIF itself. WS-Federation offers various other parameters that can be used for the purpose. You'll see later in the chapter that WIF uses *wctx* for that purpose.

- **wct** is an optional parameter that captures the time at which the Web site emits the response. It is useful for indicating to the destination a measure of the freshness of the message. If a receiver deems a message to be stale, it might assume that some form of attack is taking place and refuse to process it.

3. The browser follows the redirect indication and attempts a GET of the URL described in the earlier step—that is, the STS page followed by the mentioned *querystring* parameters.
4. The way in which the IP site processes the request is entirely dependent on how the site itself is secured. At this point, the user is not authenticated with the IP site; therefore, the attempted GET will trigger whatever authentication mechanism is in place before serving the STS page back. If, for example, *https://IP* is protected by Forms authentication, there will be a redirect step toward a login page and some back and forth as the user credentials are gathered and verified and the flow is finally redirected to the STS page. If *https://IP* is protected by Windows integrated authentication, the authentication flow will happen at the network layer instead—and so on. I am purposefully not showing a specific mechanism here to stress that the WS-Federation process is independent from the authentication protocol chosen by the STS.
5. Here I assume that the authentication process represented by step 4 concluded successfully, and that the STS page processed the request accordingly. The STS returns a 200 OK code and sends to the payload a form that will POST back to the original page (as indicated by *wtrealm*) the information requested. The form contains the following WS-Federation parameters:
   - **wa** is the same parameter described in step 2. The sign-in process is still in progress.
   - **wresult** contains the result of the authentication process: one (or more) security tokens that the STS issued if the user authenticated successfully, or a fault (in SOAP format) if something went wrong. Either way, *wresult* must be present in the message at this point of the process.
6. The browser starts to render the response, which results in a POST to the original page of the form described in the previous step.
7. Assuming that *wresult* in step 5 carried the desired token as opposed to a fault, the Web site receives the sign-in message and the token. The token undergoes various validation and authentication checks (more about those in the next section), and if everything goes well the user is authenticated. To avoid repeating the entire process for all the subsequent page requests, the Web site returns a redirect to the same page URI but sets a session cookie in the process.

8. The browser follows the redirect and attempts another GET of the page, this time sending along the session cookie.
9. The request finally succeeds. The user browser sent a valid session cookie, and the user is authenticated. Therefore, the requested resource can be returned.

There can be variations here and there, but in general the sequence just described captures well how a Web site can outsource to an external identity provider the burden of authenticating users. For the sake of completeness, next I'll introduce a few extra WS-Federation parameters in common use. Those parameters will be useful later in the text for tweaking the flow and implementing the custom behaviors that are necessary for addressing certain advanced scenarios. You can find them, together with the ones you learned about in the sequence description, in Table 3-1.

TABLE 3-1 WS-Federation Parameters You Can Use When Programming with WIF

Parameter	Description
wa	Indicates the action to be performed when processing the message. Common values are *wsignin1.0* for sign-in and *wsignout1.0* for sign-out scenarios.
wtrealm	Contains the URI of the security realm to which the requested resource belongs.
wctx	This parameter can contain any context information that the RP wants to maintain through all the redirections. Its semantics are typically known only to the RP itself—*wctx* is said to be opaque—and every other actor in the transaction will simply reattach it at every step.
wct	Reports the current time at the instant of the request creation.
wresult	Contains the outcome of the requested operation: one or more tokens, fault messages, and so on.
wreply	Indicates the URI to which responses should be directed. In the preceding sequence, *wtrealm* was used, but the two are in fact different. Whereas *wtrealm* can be used to represent the entire Web site, *wreply* can point to one specific resource (page, document, endpoint, and so forth).
wauth	Indicates the required authentication level that the STS should use for authenticating users. The specification suggests a few values representing common authentication types (SSL, SSL and strong password, smartcard, and so on), but it is really not normative on this respect.
whr	Contains the security realm of the identity provider that should be used for the transaction. In the example sequence shown, there was a direct-trust relationship between the RP and the IP; however, in many scenarios that relationship is brokered through one or more entities. I'll describe one of those scenarios and how WIF uses *whr* to address it in Chapter 4, in the section about federation.

By now, you have a good idea of how an RP Web site, regardless of the technology it uses, can get out of the user credentials authentication business by following the sequence detailed in the preceding steps. However, that does not mean the RP is entirely off the hook when it comes to validating incoming requests. You can be sure that the bad guys will still attempt forgeries, man-in-the-middle attacks, and the rest of the usual entourage of security

attacks against your RP. You might have shaken password management responsibilities off your shoulders, but you still need to be careful with how you handle requests. Simply put, the emphasis shifts from validating user credentials to validating security tokens. The good news is that this shift allows the system to take care of a lot of details on your behalf.

A Closer Look to Security Tokens

From the relying party vantage point, a security token is the only visible outcome of the user authenticating with the identity provider. A token must provide all the information that the relying party needs to know about the user, from the fact that the authentication happened at all (often the sheer existence of a token constitutes proof of that) to the required claims.

Needless to say, guaranteeing the validity of a token is of the utmost importance. Being able to forge a token, alter it, or even just steal and reuse it can lead to catastrophic consequences, given that the RP bases important authentication and authorization decisions on it. As a result, the very structure of a security token is designed to prevent tampering and abuse. Those security measures are verified via a set of common checks that every token consumer is expected to perform on incoming tokens in order to confirm that the accompanying request is legitimate.

In this section, you'll learn about those common checks in some details. You'll also have a chance to familiarize yourself with SAML tokens, by far the most common token format to date in systems based on WS-Federation and WS-Trust.

For the most common cases, you absolutely don't need to know anything about the actual token structure or the checks it mandates, because WIF takes good care of everything out of the box. The reason I touch on the topic here is that for addressing more advanced scenarios you'll occasionally need to take control of this or that part of the process. I think that understanding the big picture will pay off more than if I had just given you recipes for every case.

Verifying a Token

Here's a list of things that the RP needs to examine about a token before deciding whether it should be accepted as valid. Some of them are about token validity—that is, determining whether the token itself is well formed, regardless of the specific transaction. Others are aimed at verifying whether the token satisfies the conditions imposed by the relying party, which is the more proper authentication phase.

- **Format** The format used to issue the token must be clearly understandable and known to the RP. Once the format has been established, the token must demonstrate that it correctly implements it. If you do not know the format, you will not be able to deserialize the token and interpret what its various parts mean; therefore, making decisions based on its content impossible.

- **Integrity** The token must go from the IP to the RP without changes because any modification will alter the information that the IP stated about the subject. This condition is typically enforced by applying a digital signature to the token. Even the smallest tampering will "break" the signature, immediately informing the RP that something went wrong in transit.
- **Expiration** Tokens normally have a validity period clearly indicated. The IP does not guarantee the validity of information saved in a token outside its validity window; therefore, the RP should always verify it before trusting the content of the corresponding claims.
- **Decryption** The main way of guaranteeing end-to-end confidentiality of the content of a token is to encrypt it with the public key of the RP. Token encryption ensures that only the intended recipient will be able to see the token content, regardless of whether the communication happened on a secure channel or whether the token gets caught in some audit trail. An encrypted token must be fully decipherable by the RP.
- **Duplication** If a token is used for bootstrapping a session, as in the sign-in example shown earlier, the RP should expect to see it only once. If an RP receives the same token multiple times, it might conclude that it's a victim of a replay attack. In a replay attack, a malicious party acquires a legitimate token (for example, by eavesdropping on traffic on an unsecure channel) and sends it in an attempt to impersonate the owner of the token and start a session. One common check for preventing the situation is maintaining a cache of tokens that have already been used for successfully starting a session, and verifying that every new request is not reusing any of the known tokens.
- **Source** Normally, one RP will trust only a few well-known identity providers and accept tokens coming exclusively from them. A token might be valid per se in the sense defined so far, but if it was not issued by a trusted IP the RP has no use for it. Tokens must henceforth provide mechanisms for establishing their source of origin. The most common method is to sign the token with a private key that corresponds to a public key that the RP knows is associated with the intended STS. That way, not only is the integrity of the token guaranteed, but the token issuer is unequivocally identified. The issuer is the only entity that can apply signatures with that specific private key because it can be verified by anybody who knows the corresponding public key.
- **Audience** Usually the STS knows at the time of issuance which RP a token is being requested for. The STS embeds that information, commonly referred to as the *intended audience*, in the token itself. An RP receiving a token whose intended audience does not match the URI of the RP itself can conclude that the requestor is attempting to reuse a token that was meant for somebody else, and therefore conclude that it should block the request.

- **Claims** The claims that the STS includes in a token should match the ones requested by the RP via policy. Before making use of a set of claims, an RP should verify that all the requested claim types are present and that no extra claims (for which the issuing STS is not deemed authoritative upon) are in the token.
- **Authentication Level** In the general case, one RP outsourcing authentication to one IP is very happy to get out of the business of knowing the details of how authentication happens. However, you can encounter cases in which the RP needs assurance that the authentication operation took place using strong credentials or, in general, according to some specified authentication level. Examples of such situations are sensitive operations for which the federal government mandates specific authentication levels or the need to comply with the requirements of some process certifications. A token used in one of those scenarios must be constructed in such a way that would allow one RP to verify which authentication method was used by the IP for validating the user's credentials.

Not all of the conditions just mentioned are checked at all times, and the ability of one RP to perform those checks depends on how well the specific token format provides the necessary information. In any case, the preceding list should give you a good idea of the salient aspects of token processing. WIF has features and extensibility points for taking control of them all.

Anatomy of a Security Token

Now that you know what you need to look for in a security token, it's time to dig a bit deeper. What is a security token, anyway? The WS-* specifications define it as a collection of claims, which does not say much about the structure of the token itself. This is intentional because the idea is that all the WS-* protocols should work regardless of the token format used (up to and including token types not yet invented). I'll need to reduce the scope of this discussion a bit, so I'll focus on tokens typically used by WIF and similar systems.

Tokens are typically represented by XML fragments, digitally signed in most of the scenarios of interest here. Although they all contain claims, per the definition, they mainly differ in terms of where they come from and how they are supposed to be used.

Token Provenance A token can be requested of one STS, which will issue it on the fly. Even though no formal restrictions apply with regard to the token format used by an STS, the reality is that most of the time you'll get a token in SAML or similar formats that are good at general-purpose claims representation. Those token types are the main focus of this book. Next I'll provide more detailed information about the SAML token format. I often refer to that kind of token as *issued tokens*.

Some other tokens are representations of existing credentials or cryptographic material in token format rather than being tokens issued by one STS. I like to refer to this kind of token with the term *primitive tokens* because they are a projection of existing credential types rather than something that has been issued ad hoc in response to a custom-tailored request. Examples include X.509, Kerberos, and Username tokens. The purpose of those token types is to be able to take advantage of existing investments in authentication systems in the context of WS-* protocols. For example, if a SAML token has been signed with a key corresponding to an X.509 certificate, a requestor could send to the RP both the SAML token itself and the bits of the X.509 certificate itself. In that case, the RP can use the incoming certificate bits to create an X.509 token and verify that the signature of the SAML token is valid. This relieves the RP from needing to maintain a local copy of the certificate itself. This exact case is very common when working with WIF.

Another important difference between the two token types lies in the verification criteria that the type mandates. Whereas an issued token should be processed according to the token verification considerations described in the earlier section, a primitive token will be verified and used according to the specific credential type it projects.

Holder of Key vs. Bearer Tokens In addition to claims, a token can contain additional data, such as cryptographic keys. In that case, a client that is capable of performing cryptographic operations can actually use the token for securing the messages that are sent to the RP. Also, the token is useful both for conveying claims about the user and for providing security to the message itself. Such tokens are known as *holder of key tokens*. I'll come back to the topic in Chapter 5.

A Web browser cannot do much with a token, apart from attaching it to a request. An RP must be content with the fact that a token has been presented and hope that the requestor obtained it in legitimate ways. Such tokens are known as *bearer tokens* because they are meant to be used like cash: whomever is the bearer gets to use it.

An Example: a SAML Token As mentioned, SAML tokens are the most common tokens in claims-based identity solutions. Let's take a good look at one SAML token that could have been produced in the WS-Federation sequence I described earlier.

Note At the cost of sounding pedantic: you don't need to know any of the details I present next, and you'll likely never need to see a raw SAML token again in your career. The purpose of looking at the XML of a real token is to give more concreteness to the discussion and provide one example of how a token format supports the validity checks I described earlier. If you don't like angle brackets, feel free to skip ahead to the next section.

```
<saml:Assertion
  MajorVersion="1"
  MinorVersion="1"
  AssertionID="_1f62c6b1-a22d-4d9c-adc5-e5fc6c2ce7cb"
  Issuer="IdentityProviderSts"
  IssueInstant="2010-02-07T01:51:52.770Z">                          Description
    <saml:Conditions
      NotBefore="2010-02-07T01:51:52.766Z"
      NotOnOrAfter="2010-02-07T02:51:52.766Z">
        <saml:AudienceRestrictionCondition>
            <saml:Audience>https://site/page.aspx</saml:Audience>
        </saml:AudienceRestrictionCondition>
    </saml:Conditions>

    <saml:AttributeStatement>
        <saml:Subject>
            <saml:SubjectConfirmation>
                <saml:ConfirmationMethod>
                urn:oasis:names:tc:SAML:1.0:cm:bearer
                </saml:ConfirmationMethod>
            </saml:SubjectConfirmation>
        </saml:Subject>
        <saml:Attribute AttributeName="Group"
          AttributeNamespace="http://schemas.xmlsoap.org/claims">
            <saml:AttributeValue>Domain Users</saml:AttributeValue>
        </saml:Attribute>
        <saml:Attribute AttributeName="emailaddress"                Claims
          AttributeNamespace=
           "http://schemas.xmlsoap.org/ws/2005/05/identity/claims">
            <saml:AttributeValue>john@adatum .com</saml:AttributeValue>
        </saml:Attribute>
    </saml:AttributeStatement>

    <ds:Signature>
        <ds:SignedInfo>                                             Signature
         ...
        </ds:SignedInfo>
        <ds:SignatureValue>GG3no/osxH ... B4wY=</ds:SignatureValue>
        <KeyInfo>
            <X509Data>
                <X509Certificate>MIICHjCC ... +DvWJI1NjO+</X509Certificate>
            </X509Data>
        </KeyInfo>
    </ds:Signature>
</saml:Assertion>
```

Back in 2003, I used to handle tokens at the XML level. I can't tell you how relieved I am now that WIF takes care of the details and I rarely have to resort to the raw token code. Anyway, once you understand how to break the token into its subelements, it is actually quite easy. I divided the text into three parts, which I'll informally call the token descriptor section, the claims section, and the signature section. If you really can't stomach angle brackets, Figure 3-2 provides a simplified representation of the sample token and its three parts.

The token descriptor contains the root element <*saml:Assertion*> and the element <*saml:Conditions*>. The <*saml:Assertion*> element includes information about the format itself, including information about the freshness of the token, the issuer name, and the unique identifier of the token. The identifier will come in handy for the signature (which I'll say more about next) and can be used for detecting duplicates. The <*saml:Conditions*> element defines the validity window and establishes the intended audience for the current token.

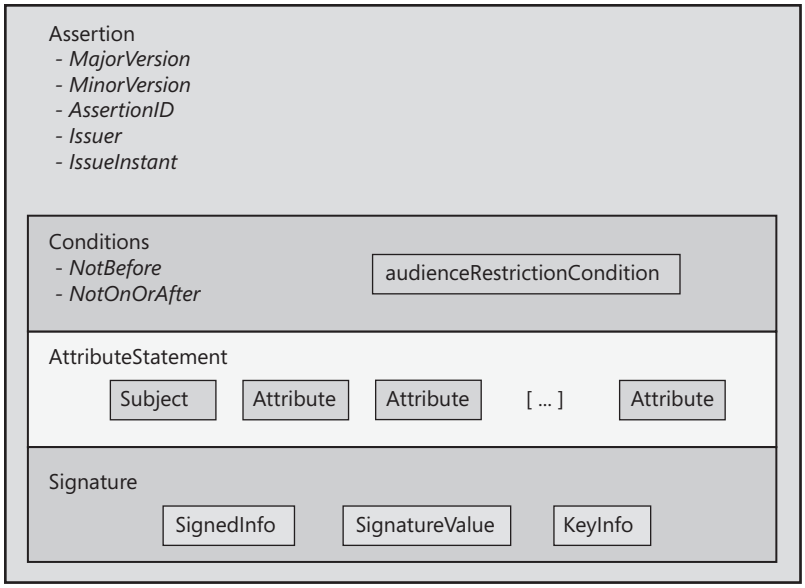


FIGURE 3-2 A simplified representation of the sample SAML token

The actual claims—in this case, *Group* and *emailaddress*—are all expressed via <*saml:Attribute*> elements. The <*saml:SubjectConfirmation*> element tells you what you already know: this is a bearer token as mandated by the passive scenario you are implementing.

The signature contains some details on the algorithm used and a reference that indicates that the signature entails the entire token (which is not shown in the preceding simplified text), plus the base64-encoded bits of the X.509 certificate that should be used for checking the signature itself.

WIF automatically performs the checks I mentioned earlier on SAML tokens and various other common token types. If you want to modify the routine check to accommodate custom logic, or if you want to support new token types, you'll have to take advantage of the many configuration tweaks and extensibility points.

Examining Tokens with WIF in ASP.NET Applications

WIF hides the format details of the specific token type received behind the *IClaimsPrincipal* interface, which is a good thing because it does not burden you with minutiae you seldom need and it does not tempt you to create artificial dependencies on things that might later change and break your application. However, at times you do want to look at the actual token details—for example, when you are troubleshooting an error situation. The object model allows you to do so. However, you can obtain

the same effect without writing code or attaching a debugger by taking advantage of *SecurityTokenVisualizerControl*, a sample ASP.NET control that can easily visualize all the incoming token details on the page itself. Just download the sample code from *http://code.msdn.microsoft.com/ClaimsDrivenControl*, and then drag the control onto the page. You'll have a collapsible element that can be consulted at runtime for checking the main token elements, the raw XML code (which is what I used for obtaining the token code for this chapter), and even the signing certificate used by the IP. Figure 3-3 shows an example of the control in action.

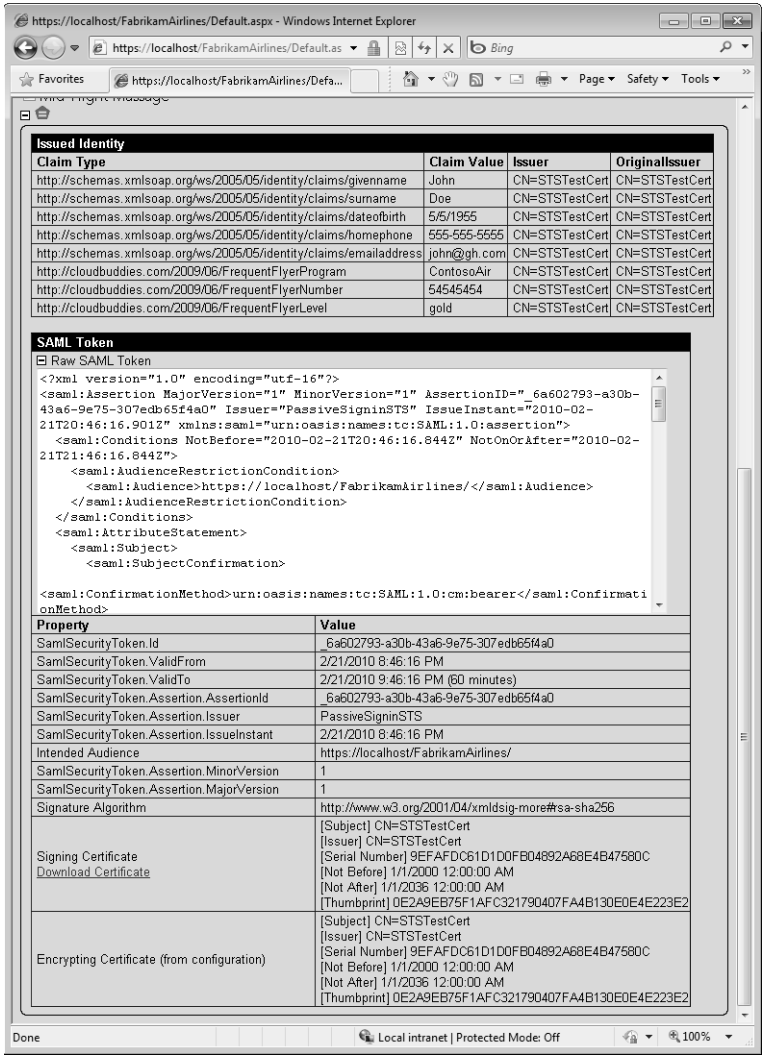


FIGURE 3-3 *SecurityTokenVisualizerControl* in action

SAML: Protocol or Token Format?

Earlier I said that WIF does not support SAML-P, but in this section I gutted a SAML token in front of you and suggested that it is the most common token handled by WIF. Both assertions are right. The possible confusion stems from the fact that the SAML specification describes both a token format and a protocol for requesting and using SAML tokens. Very often in industry literature you'll find articles mentioning SAML without specifying further, and you'll need to decide from the context if it means the token format or the protocol. Although WIF circulates SAML *tokens* as the most common currency, it does so in WS-Trust and WS-Federation transactions and (as of today) cannot use SAML-P. This is a common point of confusion when developing on the Microsoft stack. The Web Services Enhancements toolkit could process SAML tokens back in 2004, and WCF has offered SAML token support since its first version, but the first Microsoft product to announce support for SAML-P was ADFS 2.0 in November 2008.

Metadata Documents

Before I leave the WS-Federation topic behind and we move back to the implementation, I want to spend a few lines on one unsung hero of identity systems: metadata documents. This section is not strictly necessary for understanding how the WIF pipeline works, so if you are in a hurry feel free to skip ahead.

One of the most useful practices that service orientation brought to the mainstream is the idea of attaching machine-readable documents to every service that establish intended usage up front. From the description of the service operations signature itself (provided via WSDL documents) to the accurate specification of security requirements via WS-SecurityPolicy, this practice made it possible for a new breed of programming tools to arise. The Add Web Reference menu entry in Microsoft Visual Studio and the subsequent Add Service Reference menu are good examples. Anybody who tried to work with Web services before those tools became available can confirm that the quality of life for developers has improved significantly since then.

WS-Federation is no exception to the trend. The specification describes a metadata document format that can be used to describe an entity so that potential requestors can learn about its role, recognized claims, cryptographic material, available endpoints, and the like. Windows Identity Foundation takes full advantage of metadata, using it for establishing trust relationships (as you saw in Chapter 1 when I described the Add STS Reference Wizard) and for automatically exposing metadata documents for the applications being developed. The metadata document exposed by one application can be consumed by the administrator of one identity provider, who can use tooling (such as the administrative console offered

by ADFS 2.0) for getting the application onboard with the recognized relying parties, configuring which certificate should be used for encrypting tokens for that application, and so on.

Next you can find two examples of metadata documents that are compatible with the WS-Federation sequence illustrated in this chapter. The first one describes the RP; the second one describes the IP.

As is the case every time the discussion gets to under-the-hood topics, you should know that the documents described next are shown for reference purposes only: the metadata documents are meant to be machine-consumed and you'll likely never see one directly. It's just nice to see what's inside them at least once in your career.

```
<?xml version="1.0" encoding="utf-8"?>
<EntityDescriptor
  ID="_409b6137-ebf1-4e47-b411-91c3aa02935e"
  entityID="https://site/"
  xmlns="urn:oasis:names:tc:SAML:2.0:metadata">

    <RoleDescriptor xsi:type="fed:ApplicationServiceType"                    I am an RP
      xmlns:fed="http://docs.oasis-open.org/wsfed/federation/200706"
      protocolSupportEnumeration="http://docs.oasis-open.org/wsfed/federation/200706"
      mlns:xsi="http://www.w3.org/2001/XMLSchema-instance">                  I speak WS-Federation
      <fed:TargetScopes>
          <EndpointReference xmlns="http://www.w3.org/2005/08/addressing">
              <Address>https://site/</Address>
          </EndpointReference>                                             This is my address
      </fed:TargetScopes>

      <fed:ClaimTypesRequested>
          <auth:ClaimType
            Uri="http://schemas.xmlsoap.org/claims/Group"                  These are the claims I need
            Optional="true"/>
          <auth:ClaimType
            Uri="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress"
            Optional="true"/>
      </fed:ClaimTypesRequested>

      <fed:PassiveRequestorEndpoint>                                       This is the entry point for signing in
          <EndpointReference
             xmlns="http://www.w3.org/2005/08/addressing">
            <Address>https://site/</Address>
          </EndpointReference>
      </fed:PassiveRequestorEndpoint>
    </RoleDescriptor>
</EntityDescriptor>
```

The preceding document describes a relying party, as established by the *type* attribute value *fed:ApplicationServiceType* in the *RoleDescriptor* element. The element *protocolSupportEnumeration* indicates which protocol is supported (in this case, WS-Federation). The *TargetScopes* element contains the audience values for the site. The *ClaimsTypesRequested* element is the list of claims that the RP requests. The *PassiveRequestorEndpoint* element provides the address that can be used for starting the sign-in process or any other supported WS-Federation sequence.

```
<?xml version="1.0" encoding="utf-8"?>
<EntityDescriptor
ID="_d4e7c157-b6d4-49cb-b9cf-dc908e3a2f47"
entityID=https://IP/sts/
xmlns="urn:oasis:names:tc:SAML:2.0:metadata">
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      ...
    </ds:SignedInfo>
    <ds:SignatureValue>MLiY5mSCN...omnQU=</ds:SignatureValue>
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <X509Data>
        <X509Certificate> MIIB9DCC...D22T36Ld7tM6</X509Certificate>
      </X509Data>
    </KeyInfo>
  </ds:Signature>

  <RoleDescriptor xsi:type="fed:SecurityTokenServiceType"
    protocolSupportEnumeration=http://docs.oasis-open.org/wsfed/federation/200706
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:fed="http://docs.oasis-open.org/wsfed/federation/200706">
    <KeyDescriptor use="signing">
      <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <X509Data>
          <X509Certificate>MIIB9DCC...D22T36Ld7tM6</X509Certificate>
        </X509Data>
      </KeyInfo>
    </KeyDescriptor>
    <ContactPerson contactType="administrative">
      <GivenName>contactName</GivenName>
    </ContactPerson>

    <fed:ClaimTypesOffered>
      <auth:ClaimType
       Uri="http://schemas.xmlsoap.org/claims/Group"
       Optional="true">
        <auth:DisplayName>Group</auth:DisplayName>
        <auth:Description>A group to which the subject belongs.</auth:Description>
      </auth:ClaimType>
      <auth:ClaimType
         Uri=" http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress"
         Optional="true">
        <auth:DisplayName>email</auth:DisplayName>
        <auth:Description>The email of the subject.</auth:Description>
      </auth:ClaimType>
    </fed:ClaimTypesOffered>

    <fed:SecurityTokenServiceEndpoint>
      <EndpointReference xmlns="http://www.w3.org/2005/08/addressing">
        <Address>https://IP/sts/</Address>
      </EndpointReference>
    </fed:SecurityTokenServiceEndpoint>
    <fed:PassiveRequestorEndpoint>
      <EndpointReference xmlns="http://www.w3.org/2005/08/addressing">
        <Address>https://IP/sts/</Address>
      </EndpointReference>
    </fed:PassiveRequestorEndpoint>
  </RoleDescriptor>
</EntityDescriptor>
```

Signature protecting the metadata document

I am an STS

I speak WS-Federation

This is the certificate you should use to check the signatures of the tokens I issue

These are the claims I can issue

This is the endpoint on which the STS listens

This is the entry point of the STS application

The preceding document refers to the identity provider and is a bit more complex. The first thing you might notice is that it is signed. This is very important because if it is possible to modify the metadata document, it would be easy to mislead RPs into trusting malicious doppelgangers of the intended IP.

This time the *RoleDescriptor* element reports a *SecurityTokenServiceType* type, making clear that the entity offers one STS. It also contains a reference to the certificate associated to the signing key so that the certificate bits can be acquired directly by the RP at trust establishment time.

The *ClaimTypesOffered* element is the IP counterpart of *ClaimTypesRequested*, which you encountered earlier. It lists the claim types that can be obtained when getting a token from this IP. Note the human-readable components associated with every *ClaimType*, which can be used by the metadata reading tools for helping the user to make informed choices.

In addition to *PassiveRequestorEndpoint*, the metadata document for the IP contains an explicit indication of which endpoint should be used as the STS (via the *SecurityTokenServiceEndpoint* element).

The metadata documents can contain much more information, such as the kind of tokens accepted or offered, but at this point you get the idea. All that information can be harvested by WIF for automating various configuration tasks. In Chapter 4, I'll get back to the topic as I discuss scenarios in which it will be necessary to take control of metadata creation, but even in that case you won't have to work at the angle brackets level: Windows Identity Foundation wraps most metadata concepts in handy APIs.

Now that you know how things should work, it's time to take a detailed look at how WIF makes all of the above happen and how you can exercise control over every aspect of the process.

How WIF Implements WS-Federation

Windows Identity Foundation implements WS-Federation by taking advantage of the ASP.NET extensibility model. As I briefly mentioned in Chapters 1 and 2, WIF provides a set of *HttpModule*s that intercept the request and process it according to WS-Federation, producing sequences such as the one you read about in the earlier section. Those modules are typically added in the *web.config* file of the Web site, although it is also possible to add them programmatically. Here I briefly list the key modules so that you can start familiarizing yourself with their names and functions before we get into more complex scenarios. I'll get back to talking about each of them and provide an in-depth description before the end of the chapter.

- ***WSFederationAuthenticationModule*** Provides the bulk of the functionality. Often abbreviated as *WSFAM*, this module takes care of redirecting unauthenticated requests to identity providers and processing incoming tokens at sign-in time.
- ***SessionAuthenticationModule* (SAM)** Takes care of session management. After the token has been accepted and a session has been established, all subsequent requests mainly bypass the WSFAM and are processed by the SAM until the session expires, an explicit sign-out occurs, or in general the conditions change.

- ***ClaimsAuthorizationModule* (CAM)** Is responsible for enforcing authorization policies on every request. You already encountered CAM at the end of Chapter 2.

In turn, those modules take advantage of specific WIF classes that take care of finer-grained functionality, such as validating tokens, ensuring that a token has been issued by a trusted IP or implementing custom authorization logic (as you saw in Chapter 2 when describing the *ClaimsAuthorizationManager* class).

Customizing WIF's behavior to address your own scenario can entail tweaking the configuration settings for those modules, handling some events, substituting some classes used in the pipeline with your own implementation, or a mix of all those.

To be in complete control of the process, you need to understand how things flow in default conditions, which classes are activated and in what order, and how to use configuration settings and APIs for influencing it. The remainder of the chapter is dedicated to that. Don't worry if at the end of the section you don't remember everything: this section is meant to provide you with a frame of reference that you can come back to when you deal with more complex scenarios later.

ASP.NET *HttpModules*

To follow the explanations in this section, you should be familiar with how ASP.NET serves incoming HTTP requests. And to that end, I highly recommend that you refer to *Programming ASP.NET 3.5* by Dino Esposito (Microsoft Press, 2008).

Here's a super-short explanation of how it works, just for the sake of getting things going. Every request meant to be processed by ASP.NET goes through the *aspnet_isapi.dll* ISAPI extension (in IIS 6.0 or IIS 7.0 classic mode) or is handled directly by the managed pipeline in IIS 7.*. First ASP.NET creates an instance of *HttpRuntime*, which in turn creates an *HttpContext* for keeping track of how the request-serving process is progressing. The runtime then creates (or gets from a pool) one instance of *HttpApplication* for performing the actual request processing. The *HttpApplication* object raises various events that correspond to key stages of the request processing and application life cycle: *Application_Start*, *BeginRequest*, *AuthenticateRequest*, *EndRequest*, and many others. All those events can be handled by classes implementing *IHttpModule*.

ASP.NET provides a set of modules out of the box that perform common functions, such as performing Forms authentication, handling session state, and so on. The *HttpApplication* object maintains a pipeline of *HttpModule* objects, which will be invoked according to a fixed sequence whenever the events they subscribed to are fired. If you want to inject your own logic in the request-processing sequence, you can register your own handlers for those events (via *global.asax*) or write your own *HttpModule*s and add them to the *HttpApplication* pipeline. That last strategy is exactly the one followed by WIF to implement WS-Federation for ASP.NET applications.

The WIF Sign-in Flow

Figure 3-1 showed a swim lane diagram of all the messages exchanged among all the players: the browser, RP (page.aspx), and IP (sts.aspx). Here I'll revisit that diagram exclusively from the point of view of *page.aspx*, demonstrating how the various WIF components come together to produce the observed behavior. This gives me a chance to introduce all the main classes WIF uses for processing requests and touch upon their functions.

Important I cannot stress enough that in normal conditions you do not need to be exposed to most of the upcoming details. I am going into detail because I want to equip you with a thorough understanding of how WIF works so that you can take full control of the process. Nothing prevents you from following recipe-like instructions for dealing with common scenarios without really understanding how things work: you can still get the job done, although it will be harder for you to troubleshoot if something goes wrong. If you prefer a lighter-touch, task-oriented approach, feel free to skip directly to Chapter 4 and subsequent chapters.

Figure 3-4 shows a view of the *HttpModules* pipeline, displaying only the modules of interest for the sign-in sequence.

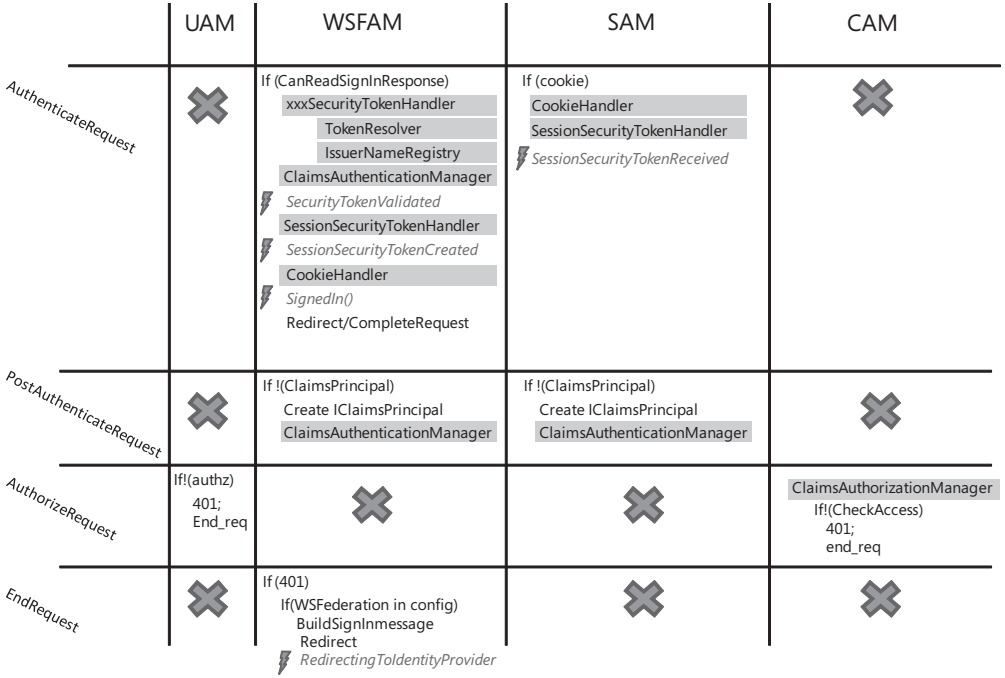


FIGURE 3-4 A matrix representation of the modules that come into play during sign-in and the pseudocode of their handler implementations

Every column in Figure 3-4 represents a module, with a left-to-right order of activation. UAM represents ASP.NET's *UrlAuthorizationModule*. Every row represents an *HttpApplication* event implemented by the module. An "X" symbol indicates that the module does not provide an implementation (or it is not interesting in this context); otherwise, the cell contains pseudo-code giving you a rough idea of how a module handles a given event. Gray rectangles represent classes that are used for performing a function at a given step, and the standard symbol for event (a lightning bolt) is shown whenever the process raises an event. You read the table from left to right and from top to bottom. ASP.NET serves *AuthenticateRequest* by calling in order all the implementations of all the modules. It then moves to *PostAuthenticateRequest*, repeats the sequence, and so on. The main exception of the flow is in the case in which one handler alters the flow itself—for example, by forcibly ending one request, which has the effect of serving *EndRequest* right away. Given certain input, you can use this diagram for predicting how WIF will react to it, which is exactly what I am going to do in the following sections.

Here we assume the following authentication settings:

```
<authentication mode="None"/>
<authorization>
    <deny users="?"/>
</authorization>
```

We are not using any built-in ASP.NET authentication mechanism. The preceding *<authorization>* element establishes that users must be known to be able to access resources.

To have the modules in the configuration depicted by Figure 3-2 in an IIS 7-based system, you need the following configuration lines:

```
<system.webServer>
    <validation validateIntegratedModeConfiguration="false"/>
    <modules>
        <add name="WSFederationAuthenticationModule"
         type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule,
               Microsoft.IdentityModel, Version=3.5.0.0,
               Culture=neutral, PublicKeyToken=31bf3856ad364e35"
         preCondition="managedHandler"/>
    <add name="SessionAuthenticationModule"
         type="Microsoft.IdentityModel.Web.SessionAuthenticationModule,
               Microsoft.IdentityModel,Version=3.5.0.0,
               Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
    <add name="ClaimsAuthorizationModule"
         type="Microsoft.IdentityModel.Web. ClaimsAuthorizationModule,
               Microsoft.IdentityModel,Version=3.5.0.0,
               Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
. . .
```

Many other settings should be added to the *web.config* file—namely, the entire <*microsoft.identityModel*> element you saw at the end of Chapter 2. I'll get back to those in the "A Second Look at <*microsoft.identityModel*>" section. By then, you will have had a chance to examine how the protocol work is subdivided across WIF classes.

Before the Redirect to the IP

The process starts when the Web site receives a GET request for *page.aspx* from one unauthenticated user.

Let's see how the sequence goes: remember to follow from left to right and from top to bottom:

- AuthenticateRequest
  - UAM: Not implemented.
  - WSFAM: WSFAM is interested only in processing sign-in messages from an STS. This is not the case. It is a simple GET, so nothing happens.
  - SAM: The SAM engages when it can find a session cookie in place, which is not the case (yet). Nothing happens.
  - CAM: Not implemented.
- PostAuthenticateRequest
  - UAM: Not implemented.
  - WSFAM: The presence of WIF in the pipeline should guarantee that the application will always have a valid *IClaimsPrincipal*; so at this point the WSFAM will take whatever kind of *IPrincipal* is available in the current context and transform it into an *IClaimsPrincipal*. Different *IPrincipal* types (anonymous, Windows, and so forth) and the presence of a client certificate in the current context all influence the values in the resulting *IClaimsPrincipal*. Before copying it into the current thread, WIF authenticates the claims in the *IClaimsPrincipal* by processing it with the *ClaimsAuthenticationManager* class. (I'll say more about that class later.)
  - SAM: The SAM serves the *PostAuthenticateRequest* event in the same way as WSFAM. Thanks to WSFAM, the current *IPrincipal* is already an *IClaimsPrincipal*, so nothing happens.
  - CAM: Not implemented.

- AuthorizeRequest
  - UAM: The current user is not authenticated, and only authenticated users can access the Web site resources. GET cannot be allowed to succeed. The UAM returns a 401 message and sends the execution straight to *EndRequest*.
- EndRequest
  - UAM: Not implemented.
  - WSFAM: The request is terminating with a 401 message, which indicates that it might be a situation in which the user needs to be redirected to one IP and be given a chance to authenticate. WSFAM inspects its configuration (in <*microsoft.identityModel*>) to see whether the WS-Federation, redirect-based authentication is enabled. (Non-redirect methods exist, and I'll touch upon them later.) If it is, it gathers all the various WS-Federation parameters (usually *wa*, *wtrealm*, *wct*, and *wctx*, and the latter is used to store the return address) and sends back a redirect toward the address of the IP found in the configuration.

That's it! The user is now off to connect with a trusted identity provider and authenticate using whatever protocols or credential types that the STS deems appropriate. You are done with the request processing until a sign-in message gets back.

The preceding sequence highlighted a couple of interesting facts. If WIF is in the pipeline, you are guaranteed to get an *IClaimsPrincipal* in your application, and you can use a class (*ClaimsAuthenticationManager*) for processing the *IClaimsPrincipal* before it reaches the application code.

Processing the Sign-in Request

Let's assume that the user successfully authenticated with the STS and is coming back with a token. According to the diagram in Figure 3-1, the Web site receives a POST containing *wa* and *wresult* with the token in it. Given that WSFAM included *wctx* in the redirect just shown, you expect to find it in the new request as well.

This sequence is probably the most important of all the ones you have seen so far. I'll introduce many new classes. For now, I suggest you pay attention mainly to their function and placement in the pipeline. There will be time later to explore how to use them as extensibility points. Figure 3-5 offers a view of how WSFAM processes *AuthenticateRequest* when the input is a sign-in message, revealing in greater detail the sequence with which the various classes are invoked.

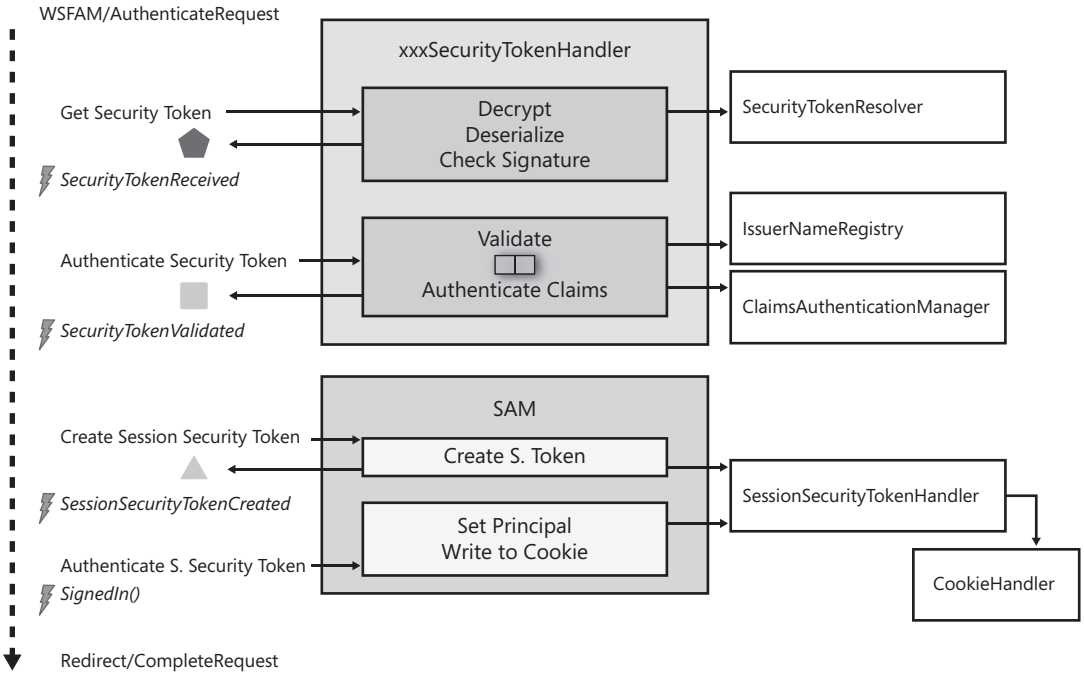


FIGURE 3-5 WSFAM processing of *AuthenticateRequest* when the input is a sign-in message containing a token

For the sake of brevity, this time I'll omit the handlers for which no implementation was provided.

- AuthenticateRequest
  - WSFAM: The incoming message is a sign-in request. Here's the processing sequence:
    - The token is extracted.
    - The *SecurityTokenReceived* event is raised.
    - Pass the raw token bits to the appropriate *SecurityTokenHandler* class for validation. WIF provides out of the box a number of classes deriving from *SecurityTokenHandler*, one for every supported token format. In this case, *Saml11SecurityTokenHandler* would be used. The *SecurityTokenHandler* takes care of validating the token. It deserializes the token according to its format, decrypts it, checks its signature, verifies its intended audience, checks for duplicates, verifies that it has been issued by a trusted IP, and makes sure that it is not expired. The validation process makes use of two helper classes: *TokenResolvers* for retrieving the cryptographic keys for decrypting the token and checking the issuer's signature, and

IssuerNameRegistry for verifying that the issuer is indeed trusted. If successful, the validation process returns a *ClaimsIdentityCollection* which, you guessed it, is a collection of *IClaimsIdentity* containing the claims from the incoming token.

- The *ClaimsIdentityCollection* instance is passed to another class, *ClaimsAuthenticationManager*, for evaluating whether the claims collections should be filtered, enriched with further values, or both. The default behavior simply returns the collection unchanged in an *IClaimsPrincipal*, but you can provide your own *ClaimsAuthenticationManager* implementation if you have logic you want to execute at this stage.
- The *SecurityTokenValidated* event is raised.
- WSFAM invokes a method on the SAM, *CreateSessionSecurityToken*, which will use the *SessionSecurityTokenHandler* class for creating a session token (*SessionSecurityToken*) that mirrors the received token. (If you want to customize the layout of the session token, *SessionSecurityTokenHandler* is what you want to subclass.)
- The *SessionSecurityTokenCreated* event is raised.
- The *Thread.CurrentPrincipal* and *HttpContext.Current.User* properties are assigned with an *IClaimsPrincipal* built from the *SessionSecurityToken*.
- If the application is configured to take advantage of cookie-based sessions, a *CookieHandler* (another customizable class) is used to persist in a cookie the *SessionSecurityToken*.
- The *SignedIn* event is raised.
- The execution goes straight to *EndRequest* with a redirect to *page.aspx*.

As you can see, behind a simple redirect you can find quite a lot of moving parts! A monolithic approach would have been easier to describe, but it would have been much harder to customize. The good news is that the modular design of this object model allows you to isolate the aspect you want to change and just focus on that, taking advantage of the defaults for everything else.

All the classes and events mentioned in this section will be discussed in detail later in the chapter, and they will be called out in upcoming chapters every time a scenario requires you to modify any of them.

Accessing a Page During a Valid Session

At this point, the bulk of the work is done. The user presented a valid token and is now considered authenticated, so the GET being performed as a result of the redirect (and all subsequent requests for the duration of the session) should entail significantly less work.

Let's use once more the handy diagram in Figure 3-2 to see how the GET of *page.aspx* will be processed differently than the last time.

- AuthenticateRequest
  - WSFAM: This is not a sign-in message, so WSFAM is not interested in processing it.
  - SAM: The SAM comes into play when a session cookie exists, and that's exactly our case. Let's see how the sequence unfolds:
    - The SAM attempts to read a cookie via the *CookieHandler* class. If its effort is successful, it will use the configured *SessionSecurityTokenHandler* for rehydrating its content in memory.
    - Raise the *SessionSecurityTokenReceived* event.
    - The *Thread.CurrentPrincipa*l and *HttpContext.Current.User* properties are assigned with an *IClaimsPrincipal* built from the *SessionSecurityToken*.

In normal conditions, the execution moves forward; if the session expired, some handling steps take place.

- PostAuthenticateRequest
  - WSFAM: *CurrentPrincipal* is already an *IClaimsPrincipal*. Do nothing.
  - SAM: *CurrentPrincipal* is already an *IClaimsPrincipal*. Do nothing.
- AuthorizeRequest
  - UAM: The user is marked as authenticated, which is all that is asked for allowing access to resources. Execution goes forward.
  - CAM: The CAM retrieves the configured *ClaimsAuthorizationManager* class and uses it for calling *CheckAccess* on the current *IClaimsPrincipal*. If *CheckAccess* returns *false*, the CAM sets 401 as the response code and sends execution straight to *EndRequest*; otherwise, the request goes further down the pipeline.

The preceding sequence introduced you to practically all the classes and extensibility points that WIF offers you for customizing the way in which passive scenarios are served. Later in the chapter, I'll provide a quick reference to all those extensibility points, highlighting their intended usage and notable aspects. In subsequent chapters, I'll demonstrate how those extensibility mechanisms are leveraged for solving classic authentication problems in a variety of situations.

Redirect-Based Protection vs. Login Page

This section explored how WIF implements WS-Federation authentication via redirection initiated via *HttpModules*. When the application is correctly configured,

any attempt to access a resource triggers the sign-in process before the user has any possibility of seeing the application UI.

WIF offers an alternative method that does not require the same blanket protection mechanism. An ASP.NET control, named *FederationPassiveSignIn* control, can be dragged onto a page of the application and used as an explicit entry point for the sign-in process. *FederationPassiveSignIn* instantiates WSFAM explicitly instead of hooking it up in the *HttpModule*s pipeline. This allows you to render some UI and engage with unauthenticated users instead of forcing an immediate redirect to the STS. In the typical scenario, you configure the application to use some standard authentication techniques (such as Forms authentication) and include the *FederationPassiveSignIn* control on the *login.aspx* page. The properties of *FederationPassiveSignIn* mirror various settings you would normally find in the *<microsoft.identityModel> config* element. This allows you, for example, to easily handle alternative settings by acting programmatically on the control properties, or by simply having more than one instance. Figure 3-6 shows the *FederationPassiveSignIn* control in one of the sample pages of the WIF SDK.

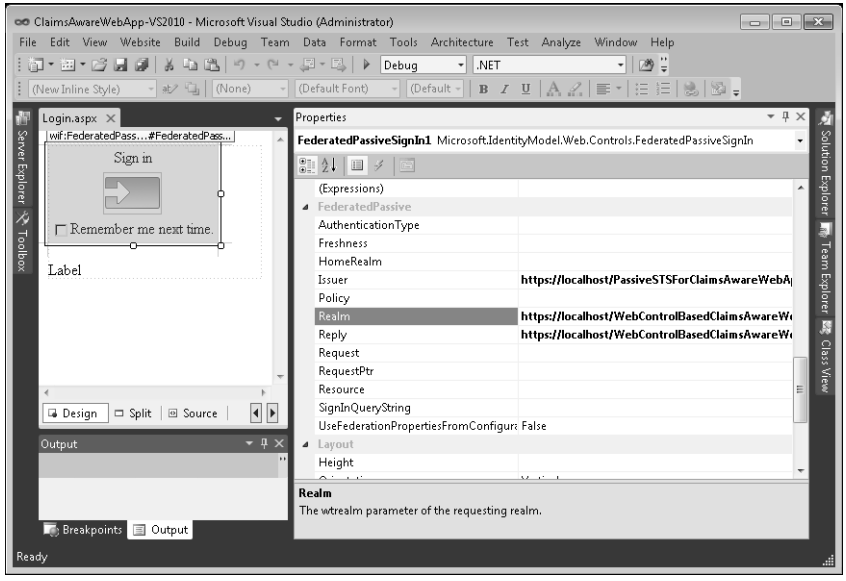


FIGURE 3-6 A *FederationPassiveSignIn* control on one page and some of its notable properties

Here I won't provide many details on how to use *FederationPassiveSignIn*. The concepts you need to understand to use it are similar to what holds true for the redirect case, which has been explored in depth. In Chapter 4, I'll occasionally mention the use of the control whenever its use will be more convenient for addressing specific scenarios than the passive redirect approach.

WIF Configuration and Main Classes

Now that you've had a chance to see in greater detail how WIF implements the sign-in sequence, you are in a better position to understand which configuration settings should be tweaked and which classes should be manipulated in order to bend the default behavior to your needs. In this last section, I'll give a quick list of the main *config* options and notable classes. Starting with the next chapter, we'll explore how to apply specific settings for addressing concrete scenarios you might encounter in your development activities.

A Second Look at *<microsoft.identityModel>*

It is time to revisit WIF's configuration element and learn about its various elements and their functions.

The Default Configuration as Generated by FedUtil.exe

The sequence examined in the earlier section was driven by the default configuration, as generated by the Add STS Reference Wizard or by the explicit execution of *FedUtil.exe* against an RP's *web.config* file. (From now on, I'll just refer to *FedUtil.exe*, given that the effects are completely equivalent.) Note that the existing *config* file is preserved in a backup copy (following a naming schema of the form *web.config.backup.1*).

Next you can see a typical *web.config* file, where most of the code has been eliminated to focus your attention on the elements inserted by *FedUtil.exe*. Because the file can be long, I'll break it up into pieces and comment as we encounter interesting elements.

```
<?xml version="1.0"?>
<configuration>
    <configSections>
        [..]
        <section name="microsoft.identityModel"
          type="Microsoft.IdentityModel.Configuration.MicrosoftIdentityModelSection,
          Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31bf3856ad364e35"/>
    </configSections>
```

The first sign of the presence of WIF in the application is the registration of its configuration section, under the moniker *microsoft.identitytModel*:

```
    <appSettings>
        <add key="FederationMetadataLocation"
         value="C:\inetpub\wwwroot\BasicWebSite_STS\FederationMetadata\2007-
         06\FederationMetadata.xml"/>
    </appSettings>
[..]
```

The *appSetting* collection gets a new entry that indicates the location of the metadata document of the IP STS to which authentication is being outsourced. Note that this entry is used by the WIF tooling but it is not really used by the framework itself during the pplication execution.

```
<location path="FederationMetadata">
    <system.web>
        <authorization>
            <allow users="*"/>
        </authorization>
    </system.web>
</location>
```

FedUtil.exe generates metadata for the current application. The <*location*> element shown in the preceding code has the function of making the metadata documents accessible to unauthenticated users, excluding it from the blanket protection that guards every other resource in the Web site. This makes it possible for the IP itself to consume the metadata without having to authenticate and automatically provision the RP among the recognized ones:

```
<system.web>
    <authorization>
        <deny users="?"/>
    </authorization>
    <authentication mode="None"/>
    [..]
```

You already encountered this setting in the sign-in sequence explanation:

```
        <assemblies>
            [..]
            <add assembly="Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
                           PublicKeyToken=31BF3856AD364E35"/>
        </assemblies>
    [..]
```

The wizard takes care of adding a reference to the WIF assembly, of course:

```
    <!--Commented out by FedUtil-->
    <!--<authentication mode="Windows" />-->
    <!--
        [..]
```

Any existing authentication settings are commented out:

```
        <httpModules>
           [..]
            <add name="WSFederationAuthenticationModule"
                 type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule,
                       Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
                       PublicKeyToken=31bf3856ad364e35"/>
            <add name="SessionAuthenticationModule"
                       type="Microsoft.IdentityModel.Web.SessionAuthenticationModule,
                             Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
                             PublicKeyToken=31bf3856ad364e35"/>
        </httpModules>
    </system.web>
    [..]
```

Applications hosted on Microsoft Internet Information Services (IIS) versions older than 7 refer to the <*httpModules*> element of <*system.web*> for populating the *HttpModules* pipeline. Here *FedUtil.exe* added *WSFAM* and *SAM*.

```
    <system.webServer>
        [..]
        <modules>
            [..]
            <add name="WSFederationAuthenticationModule"
                 type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule,
                 Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
                 PublicKeyToken=31bf3856ad364e35" preCondition="managedHandler"/>
            <add name="SessionAuthenticationModule"
                 type="Microsoft.IdentityModel.Web.SessionAuthenticationModule,
                       Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
                       PublicKeyToken=31bf3856ad364e35" preCondition="managedHandler"/>
        </modules>
        [..]
    </system.webServer>
```

Applications hosted on IIS7 use the <*modules*> element in <*webServer*> for adding *HttpModules*, so the preceding lines are added as well:

```
[..]
    <microsoft.identityModel>
      <service>
```

The preceding code segment is where the proper WIF configuration begins. A <*service*> element contains a group of settings. Alternative configurations can be provided by adding multiple named <*service*> elements in <*microsoft.identityModel*>:

```
        <audienceUris>
          <add value="https://site/"/>
        </audienceUris>
```

The <*audienceURI*> element lists all the intended audience URIs that are considered valid for tokens received by the current Web site. Note that the comparisons on the audience URIs are case sensitive, something to keep in mind if you want to avoid nasty errors. The <*audienceURI*> value will be picked up and verified by the current *SecurityTokenHandler*:

```
<federatedAuthentication>
  <wsFederation passiveRedirectEnabled="true"
                issuer= https://IP/STS/
                realm="https://site/"
                requireHttps="true"/>
  <cookieHandler requireSsl="true"/>
</federatedAuthentication>
```

The <*federatedAuthentication*> element drives the behavior of both WSFAM and the SAM.

The <*wsFederation*> element controls WSFAM. Its attributes stand for specific WSFAM switches or straight WS-Federation parameter values. As you can imagine, many settings can be applied in this element. Here is a brief explanation of the default values shown:

- *passiveRedirectEnabled= true* indicates that the WSFAM is supposed to react to 401 return codes by redirecting to the configured STS, as we observed earlier.
- *requireHttps=true* establishes that communications with the STS must take place on a Secure Sockets Layer (SSL) protected channel.
- *Issuer* contains the address of the intended STS to which unauthenticated requests will be redirected.
- *Realm* is the value that will be assigned to *wtrealm*.

The <*cookieHandler*> element drives the SAM. In this case, it expresses the requirement to make the session cookie available only to Https requests. Note that *requireHttps* and *requireSsl* are both referring to the same thing, the use of https.

```
<applicationService>
  <claimTypeRequired>
   <claimType type=" http://schemas.xmlsoap.org/claims/Group"
              optional="true"/>
   <claimType type="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress"
              optional="true"/>
  </claimTypeRequired>
</applicationService>
```

The <*applicationService*> element lists the claims that the application requires. It is used mainly for metadata document generation.

```
        <issuerNameRegistry
          type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
                Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
                PublicKeyToken=31bf3856ad364e35">
           <trustedIssuers>
             <add thumbprint="0E2A9EB75F1AFC321790407FA4B130E0E4E223E2"
                  name="https://IP/STS/"/>
           </trustedIssuers>
        </issuerNameRegistry>
      </service>
    </microsoft.identityModel>
</configuration>
```

The last element in the default <*microsoft.identityModel*> is <*issuerNameRegistry*>. It specifies which class should be used for maintaining a list of trusted issuers and, in particular, references their certificates so that the signature of an incoming token can always be verified as having been generated by a trusted IP. In the default case, WIF uses a class available out of the box, *ConfigurationBasedIssuerNameRegistry*, which associates certificate thumbprints to assigned issuer names. This mechanism makes it possible to associate a simple issuer name string property to claims, deferring the association to the corresponding identity provider here in the configuration.

Beyond the Default Configuration: Quick Reference

The configuration just described gets the job done in basic cases. By featuring sensible default choices, WIF protects you from the underlying complexities and keeps <*microsoft.identityModel*> simple by hiding all the options that would otherwise crowd it.

Earlier in the book, I said that WIF allows you to do practically everything from the configuration. In concrete terms, that means that <*microsoft.identityModel/Service*> offers a list of subelements that can control (or even substitute tout court) the classes listed in the sign-in sequence and the way in which they operate. Next I'll list some of the most common settings you might need. Unless otherwise noted, all elements are direct child elements of <*microsoft.identityModel/Service*>. The list is not meant to be a complete reference. I'll dig deeper into individual settings when I discuss the classes themselves. For more information on anything else, refer to the WIF SDK documentation. Figure 3-7 shows a simplified schema of the element <*microsoft.identityModel/Service*> structure.

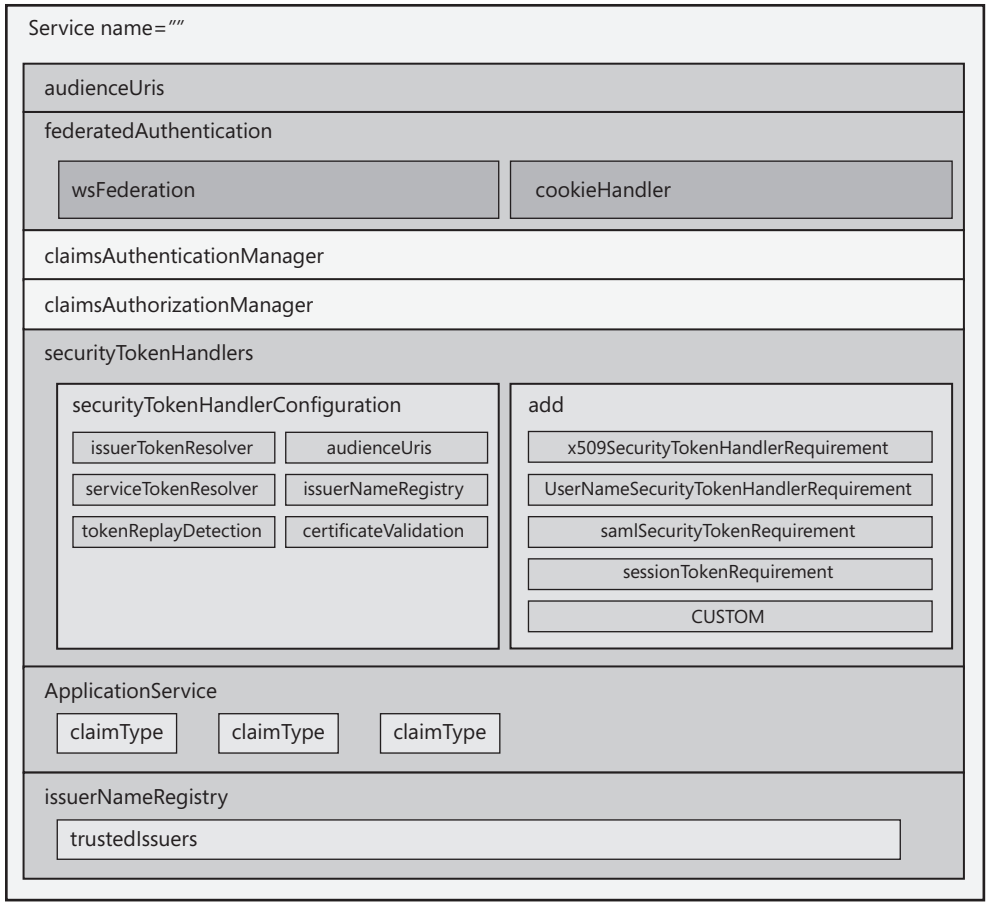


FIGURE 3-7 A schema of the main subelements of <*microsoft.identityModel/Service*>

The following sections describe the common settings:

<ClaimsAuthenticationManager> This element offers the possibility of specifying a custom subclass of *ClaimsAuthenticationManager* to be used for authenticating incoming claims.

<ClaimsAuthorizationManager> You encountered <*ClaimsAuthorizationManager*> in Chapter 2. Its function is similar to <*ClaimsAuthenticationManager*> in that it allows you to register your own implementation of *ClaimsAuthorizationManager*.

<federatedAuthentication> This element does not have attributes.

<wsFederation> This subelement was described in the section "The Default Configuration as Generated by FedUtil.exe". The main tasks you can perform with this subelement, in addition to the ones already mentioned, are related to sign-out flows and specifying extra WS-Federation parameter values as attributes. Because the attribute names do not correspond to the WS-Federation parameter names, I've provided Table 3-2 as a short conversion table of the main parameters.

TABLE 3-2 WS-Federation Parameters You Can Use When Programming with WIF

Configuration Attribute	WS-Federation Parameter
authenticationType	*wauth*
freshness	*wfresh*
homeRealm	*whr*
policy	*wp*
realm	*wtrealm*
reply	*wreply*
request	*wreq*
requestPtr	*wreqptr*
resource	*wres*

<cookieHandler> This subelement allows for a great deal of customization of the session cookie itself and its handling. I'll say more about session customization later in this chapter and in Chapter 4.

<maximumClockSkew> This element specifies the maximum difference deemed acceptable when evaluating expirations.

<serviceCertificate> If the application expects encrypted tokens, <*serviceCertificate*> specifies the decryption certificate. The syntax it uses is analogous to similar elements in WCF. *x509FindType* determines the criteria that must be used to identify the certificate, and *storeLocation* is the store to inspect.

<***applicationService***> As described earlier, this element lists the claim types required by the application.

<issuerNameRegistry> The obvious ways of extending the default behavior of <*issuerNameRegistry*>, described earlier, is to provide your own implementation of the *IssuerNameRegistry* class. You might have various reasons for doing that, from the most trivial (for example, you might prefer to select certificates by subject string rather than thumbprint if the certificate is already in a local store) to the most sophisticated issuer selection criteria.

Token Validation Settings The <*service*> element can contain some subelements that will directly influence the way in which WIF validates incoming tokens. You already encountered <*audienceUri*> and <*issuerNameRegistry*>. The <*issuerTokenResolver*> element allows you to define a custom class for retrieving the cryptographic material that should be used for checking the signature of incoming tokens; <*serviceTokenResolver*> offers a similar functionality, allowing you to configure your own class for retrieving the key that should be used for decrypting incoming tokens. Finally, <*certificateValidation*> gives you control over the way in which certificates are validated. In addition to defining the usual validation modes (*PeerTrust*, *ChainTrust*, *None*, and so forth), <*certificateValidation*> can even handle custom logic, by specifying a custom *X509CertificateValidator* class via the <*certificateValidator*> subelement.

<securityTokenHandlers> One thing was clear from the analysis of how WIF handles the sign-in sequence: most of the heavy lifting is done by the *SecurityTokenHandler* class. <*securityTokenHandlers*> is the element that gives you control over the collection of the various *SecurityTokenHandler*s offered by default in WIF, and it gives you a chance to register your own.

The first child element of <*securityTokenHandlers*> is <*securityTokenHandlerConfiguration*>. It contains the same elements described earlier in the "Token Validation Settings" bullet point, and their effects are completely equivalent. The WIF SDK suggests that <*securityTokenHandlerConfiguration*> should be preferred, although FedUtil.exe does not follow that advice in the configuration it generates. <*securityTokenHandlerConfiguration*> contains an extra <*tokenReplayDetection*> element, which rules over the cache used for detecting attempts to improperly reuse tokens and offers a hook for providing your own cache if you choose to.

The other child elements of <*securityTokenHandlers*> are elements that can add or remove handlers from the current collection, in line with the ASP.NET convention for handling configuration collections. The <*clear*> element gets rid of all the handlers in the collection, whereas <*remove*> can be used for removing a given handler by specifying its type.

The default token handlers collection includes types predefined in WIF—namely, *Saml11SecurityTokenHandler*, *Saml2SecurityTokenHandler*, *X509SecurityTokenHandler*, *KerberosSecurityTokenHandler*, *WindowsUserNameSecurityTokenHandler*, *RsaSecurityTokenHandler*, and *EncryptedSecurityTokenHandler*. The first three, plus *MembershipUserNameSecurityTokenHandler* and *SessionSecurityTokenHandler*, have their own configuration settings: you can see the corresponding elements listed in Figure 3-7. If you want to configure a given token handler type, you can use the <*add*> element to add that token handler type with the desired settings. If one handler of the same type is already present in the collection, the new options will overwrite the existing ones. The same procedure applies for your custom handler types. Here I won't go into the details of every handler setting. In Chapters 4 and 5, I'll give more details for some selected types. For everything else, refer to the WIF SDK documentation for more information.

Notable Classes

The detailed description of how WIF processes an ASP.NET request introduced you to most of the classes you need to understand in order to control WIF's behavior. This section contains a brief list for your reference, summarizing some of the things mentioned earlier and giving some indication about intended usage. This list is by no mean complete or exhaustive, and it is designed to summarize the most salient point about those classes. Refer to the WIF SDK documentation for more detailed information. Again, you'll have a chance to see in greater detail in upcoming chapters those classes that are instrumental in addressing key scenarios.

WSFAM

WSFederationAuthenticationModule is the *HttpModule* implementing the WS-Federation protocol. It can be used as *HttpModule* in the ASP.NET pipeline, in which case its behavior can be driven by the <*federatedAuthentication/wsFederation*> element. Furthermore, its events can be handled via *global.asax*. Table 3-3 lists the events WSFAM raises and some possible uses.

TABLE 3-3 WSFAM Events

Event	Uses
AuthorizationFailed	You can handle this event for fine-tuning the reporting, both for end user messages and instrumentation, in case of failures in the authentication process.
RedirectingToIdentityProvider	This event is the ideal event to use if you need to programmatically modify the sign-in message to the STS. You often need to do this if you need to inject WS-Federation parameters that are not known at design or that must be dynamically selected, because <*wsFederation*> admits only static values.
SecurityTokenReceived	After the token has been received, some validation already has taken place. That means you can augment it without invalidating it. Note that if you want to work on claims, the preferred place to do so is in the *ClaimsAuthenticationManager* class, which is discussed in the upcoming "*ClaimsAuthenticationManager*" section.
SecurityTokenValidated	*SecurityTokenValidated* signals the success of all the validation checks performed by the *SecurityTokenHandler*. If you need to complete further verification—such as when the incoming token not only is valid but is also associated with a user actually provisioned for your applications—this event provides a good place to do so.
ServiceConfigurationCreated	This is your chance to augment the *config*. For example, you can handle this event for the rare options that WIF cannot set via configuration. This event fires only once when the application starts. The handler for *ServiceConfigurationCreated* should be assigned in *Application_Start*.

Event	Uses
SessionSecurityTokenCreated	Here you can manipulate the session token—for example, by changing its expiration value. Note that in general the customizations of session tokens are applied via specialized *SecurityTokenHandler*s.
SignedIn	Any preparation work you want to perform once you know that the user successfully established a session, but before hitting the actual application code, can take place in this handler.

WSFAM can also be instantiated directly, as in the *FederationPassiveSignIn* control. In that case, whomever instantiates it must work directly with properties and events. Again, in the *FederationPassiveSignIn* control case, this is done by the control itself, which exposes properties and events accordingly.

WSFAM uses, directly or indirectly, nearly all the customizable classes I've described so far. WSFAM is usually not subclassed; however, you might want to write your own equivalent module to implement protocols other than WS-Federation.

SAM

The *SessionAuthenticationModule* takes care of creating and verifying sessions. It is driven by the *config* element *<federatedAuthentication/cookieHandler>*. SAM works directly with *CookieHandler* and *SessionSecurityTokenHandler* classes, and its *config* can drive those directly. SAM raises two events, *SessionSecurityTokenCreated* and *SessionSecurityTokenReceived*, as part of normal session life cycle (similar to WSFAM). It also raises various sign-out events (*SigningOut, SignedOut, SignOutError*), which you can use for performing any cleanup your application requires at session's end.

SAM can be subclassed—for example, to implement custom behaviors such as sliding sessions (sessions that can time out if the user is not active for a given time interval, regardless of the token expiration time). In that case, you need to make sure that your implementation provides the methods that the WSFAM expects in the session token creation phase, during sign-in request processing. Some examples of those are *ContainsSessionTokenCookie*, *CreateSessionSecurityToken*, and *AuthenticateSessionSecurityToken*. Often, a good strategy for those is to call the base class implementation before adding your logic (if necessary).

An interesting property of the SAM is *IsSessionMode*. When set to *true*, *IsSessionMode* has the effect of storing the bulk of the session on a server-side token cache instead of writing everything in the cookie. The cookie itself will just contain a small context identifier, which will be used for retrieving the session on the server. Unfortunately, in this version of the

product there is no way to set *IsSessionMode* from the configuration file. You can set it via a property of the *PassiveSignInControl*, or in the *global.asax* file as follows.

```
void WSFederationAuthenticationModule_SessionSecurityTokenCreated(object sender,
     Microsoft.IdentityModel.Web.SessionSecurityTokenCreatedEventArgs e )
{
    FederatedAuthentication.SessionAuthenticationModule.IsSessionMode = true;
}
```

CAM

The *ClaimsAuthorizationModule* is little more than an instancing mechanism for your *ClaimsAuthorizationManager* class of choice, as explained in Chapter 2. The only thing that can be specified in <*claimsAuthorizationManager*> is its associated configuration element. It raises no events of its own.

ClaimsAuthenticationManager

The *ClaimsAuthenticationManager* class provides you with a hook for inserting claims-processing logic before your application code is invoked. It is used both by WSFAM (in *AuthenticateRequest* and *PostAuthenticateRequest*) and SAM (in *PostAuthenticateRequest*), so if your application has an *IClaimsPrincipal* in the *Thread.CurrentPrincipal* property, you can be certain that it has been generated by a *ClaimsAuthenticationManager*.

The default *ClaimsAuthenticationManager* does not perform any processing; it simply packages the incoming claims (received in the form of *ClaimsIdentityCollection*) and packages them in an *IClaimsPrincipal*. If you want to create your own implementation, all you need to do is derive from *ClaimsAuthenticationManager* and override the method *Authenticate*. Referring the class from the <*claimsAuthenticationManager*> tells WIF to use your implementation instead of the default one.

You might want to roll your own *ClaimsAuthenticationManager* for various reasons, from the need to filter the set of claims received (you might want to weed out all the claims for which you don't consider the originating IP to be authoritative) to the creation of custom *IClaimsPrincipal*s featuring specialized object models.

ClaimsAuthorizationManager

ClaimsAuthorizationManager was discussed in the section "Basic Claims-Based Authorization" in Chapter 2. More examples of *ClaimsAuthorizationManager* in action will be given in the upcoming chapters.

SecurityTokenHandler

SecurityTokenHandler is an abstract class for serializing and deserializing tokens. Token handling is pivotal to almost everything else in WIF. Many standard concrete implementations of *SecurityTokenHandler* are provided out of the box. As discussed in the section "A Second Look at <*microsoft.identityModel*>," *Saml11SecurityTokenHandler*, *Saml2SecurityTokenHandler*, *X509SecurityTokenHandler*, and *MembershipUserNameSecurityTokenHandler* have their own specialized subconfiguration elements. If the settings that can be changed via *config* do not cover the functionality you need in your scenario, you can create your own token handler class. If you plan to process a token type for which WIF already provides a handler, deriving from the appropriate base class will save you a lot of work; if instead you need to handle an entirely new token type, you can derive directly from the base abstract class.

To write a new handler, you must implement the *TokenType* property and the *GetTokenTypeIdentifiers* method to define what tokens the handler will be used for. Obviously, if you are deriving from a concrete handler here, you should call the base class implementation.

Next you must override the *CanWriteToken* and *CanValidateToken* properties to expose the functionality of the handler. That basically corresponds to declaring to the pipeline that the handler is willing and able to take care of the given token type. Note that those methods (plus *CreateSecurityTokenReference*) will throw a *NotImplemented* exception in the base class.

Similar methods are *CanReadToken*, which examines the raw bits of the token and checks if the type is the expected one, and *CanReadKeyIdentifier/CanWriteKeyIdentifier*, which I'll talk about more in Chapter 5. The actual token processing takes place in *CreateToken, WriteToken, ReadToken*, and *ValidateToken*. The WIF SDK has examples of all those.

Of course, nothing prevents you from extending standard token handlers by adding custom fields or processing. Some token handlers also have protected methods you can override to add custom functionality—for example, *Saml11TokenHandler* and *Saml2TokenHandler* have many *Read/Write* methods for assertions, attributes, and conditions that can be customized.

SessionSecurityTokenHandler

*SessionSecurityToken*s are used to preserve an authenticated session, typically by encoding information in a cookie submitted by the client. *SessionSecurityToken*s determine the layout of the session cookie—namely, which information is stored and which data is discarded. The handler has various interesting properties, such as *TokenLifeTime* (the lifetime of the session, independent of the token establishing the session), *Transforms* (an extensible mechanism for defining encryption, compression, and other operations on the cookie), and *TokenCache* (the cache used to reduce the overhead of applying transforms). You create your own custom *SessionSecurityTokenHandler* if you need to influence the layout of the session in ways that are not achievable through sheer configuration.

CookieHandler

CookieHandler abstracts the packaging of bytes into a cookie or cookies, independently of the actual *SessionSecurityToken* format. You might want to roll your own in situations in which you want to take control of the packaging process—for example, in the case in which knowledge of the likely content of the claims allows you to devise a more efficient compression algorithm.

TokenResolvers

When keys are specified in tokens, either for encryption (*ServiceTokenResolver*) or signing (*IssuerTokenResolver*), this class will map the specification to a *SecurityKey*.

IssuerNameRegistry

IssuerNameRegistry provides a mapping of *SecurityTokens* (usually certificates) to a string that can be included in claims. A valid mapping must exist, or the request will fail. You can have many reasons for rolling your own *IssuerNameRegistry* instead of relying on the default *ConfigurationBasedIssuerNameRegistry*, the most common one being the need to create a façade for a store of issuer certificates. Implementing your own *IssuerNameRegistry* is not hard. All you need to do is override *GetIssuerName*.

Summary

This chapter examined in depth how WIF addresses authentication in the ASP.NET case. It introduced many new concepts that will come in handy in upcoming chapters that talk about WCF.

You learned that in addition to using SDK tooling three more ways of programming with WIF exist: acting on the configuration, handling events, and subclassing.

You read about WS-Federation, the standard used by WIF for securing browser-based applications, and you discovered how it takes advantage of HTTP for distinguishing identity information between subject, IP, and RP.

You examined in depth how WIF processes a WS-Federation request and learned how various classes collaborate in sequence to achieve the desired result.

Finally, you gained insight into how WIF configuration elements can be used to steer request processing, and you reviewed the key classes that play a role in the process.

You now have a much deeper understanding of the mechanisms underlying claims-based identity and how WIF implements the approach. In the next chapter, I'll present various use cases in which WIF's configuration and extensibility points are used for solving identity and access challenges.

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.

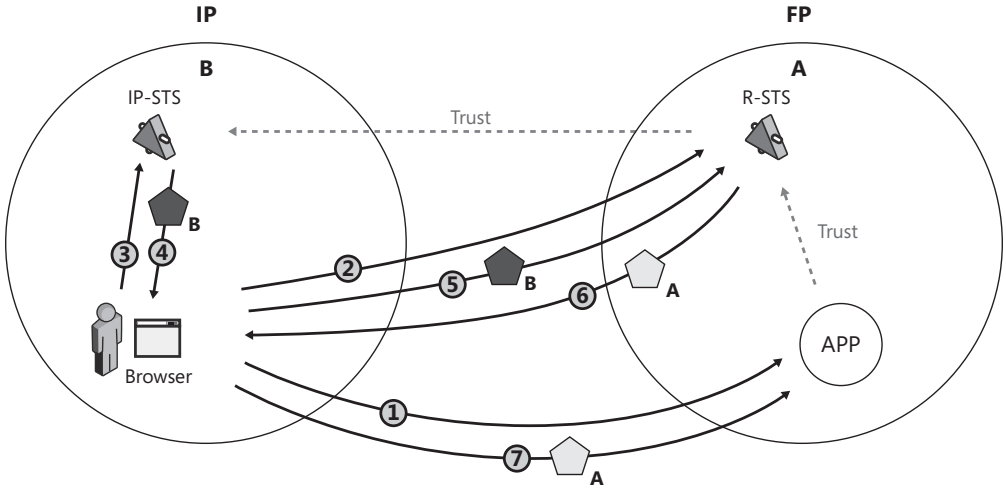


FIGURE 4-1 The authentication flow in a federation relationship between two organizations

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.

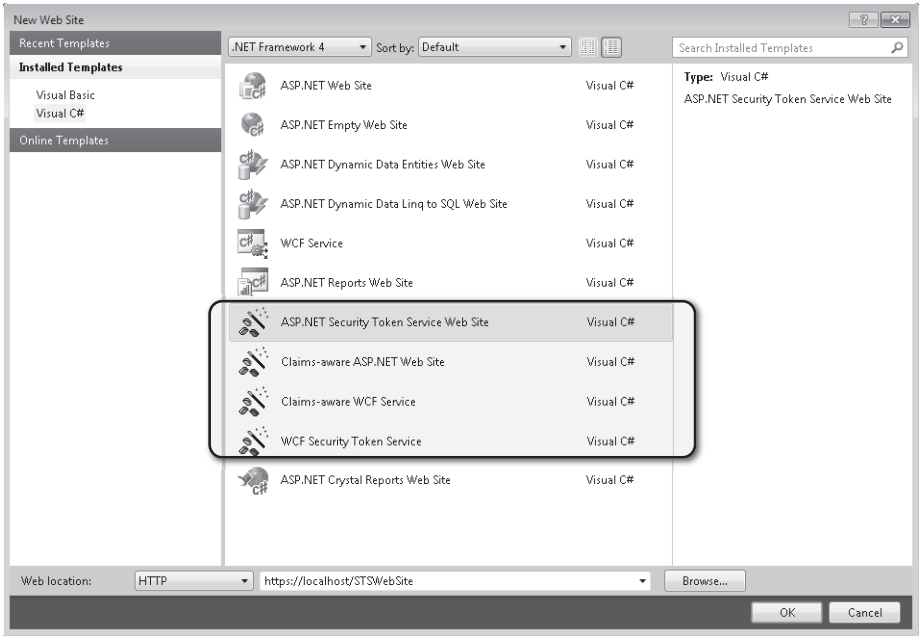


FIGURE 4-2 The templates installed by WIF SDK 4.0, with the template used for creating an ASP.NET STS highlighted

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.

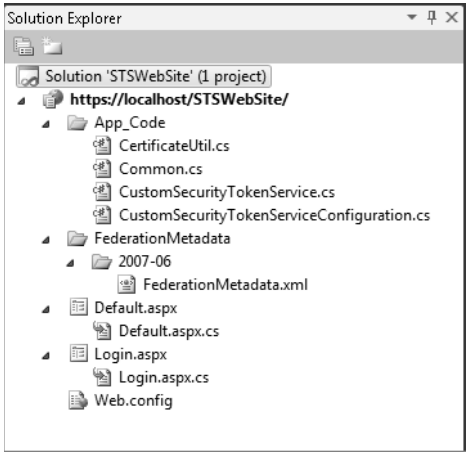


FIGURE 4-3 The ASP.NET STS project structure

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*.

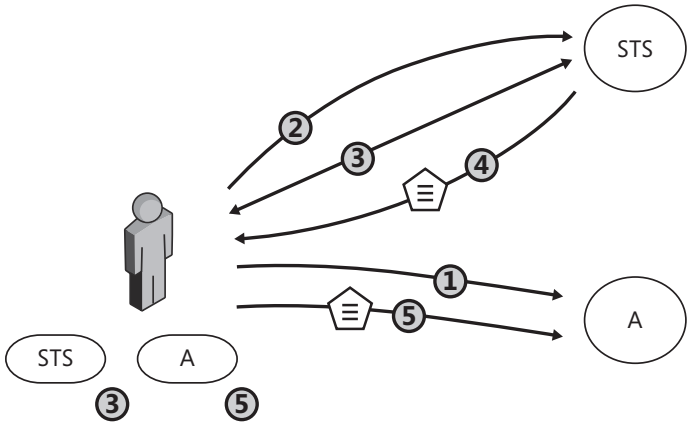


FIGURE 4-4 The user signs in the RP named A, and in so doing it receives session cookies both from the STS and A

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.

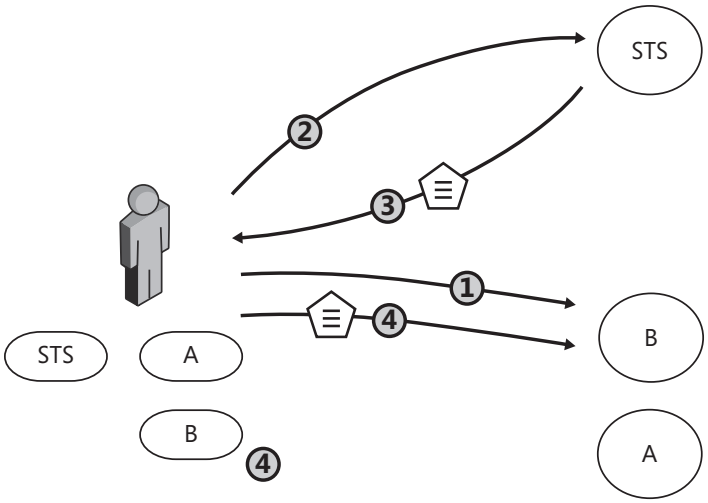


FIGURE 4-5 The user signs in to the RP named B, and the existing session with the STS allows the user to sign in without being prompted for the STS credentials

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.

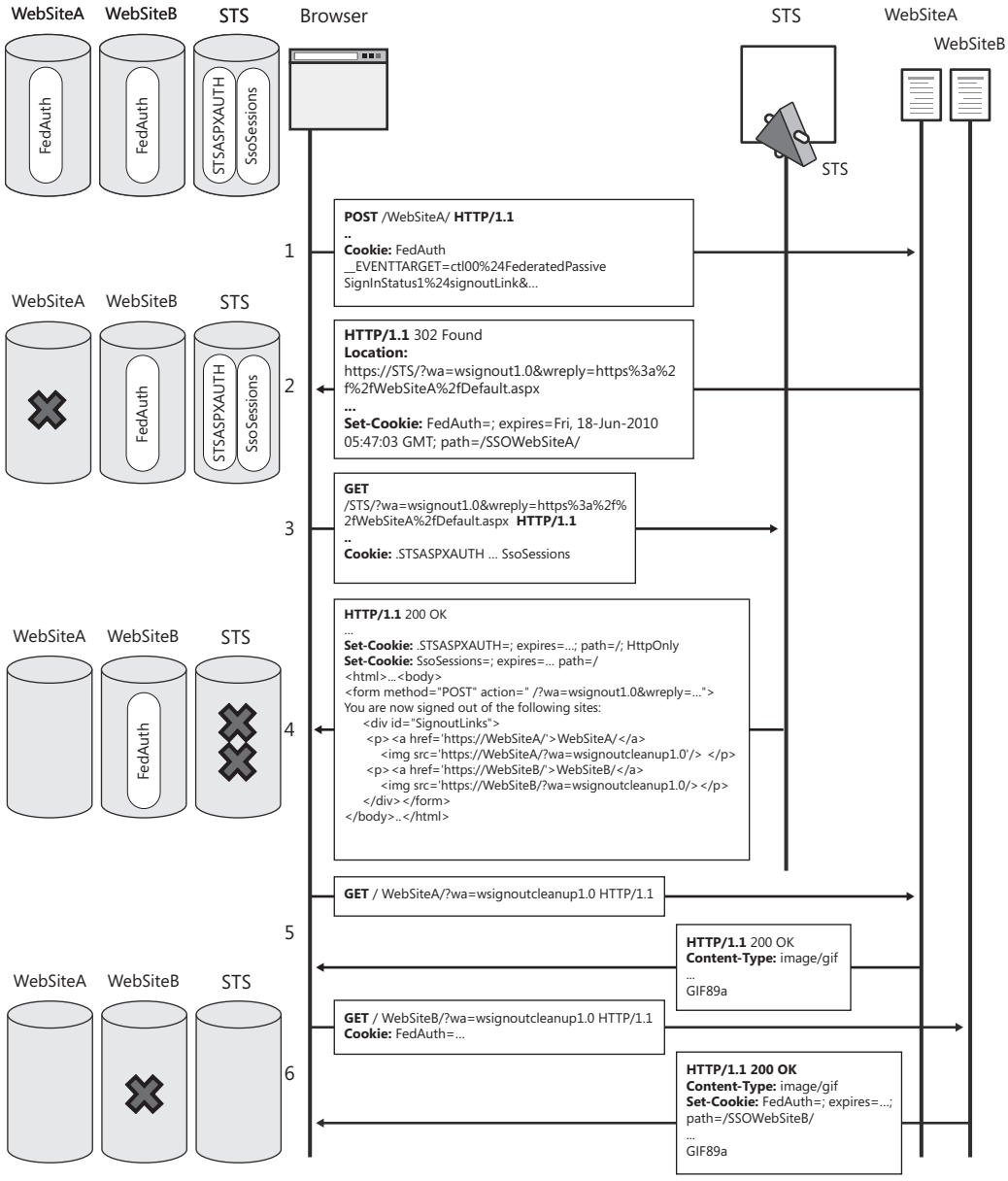


FIGURE 4-6 A Single Sign-out process taking place as described in WS-Federation

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.

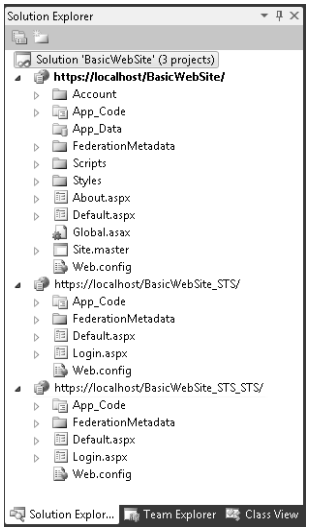


FIGURE 4-7 *BasicWebSite* trusts *BasicWebSite_STS*, which in turn trusts *BasicWebSite_STS_STS*

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.

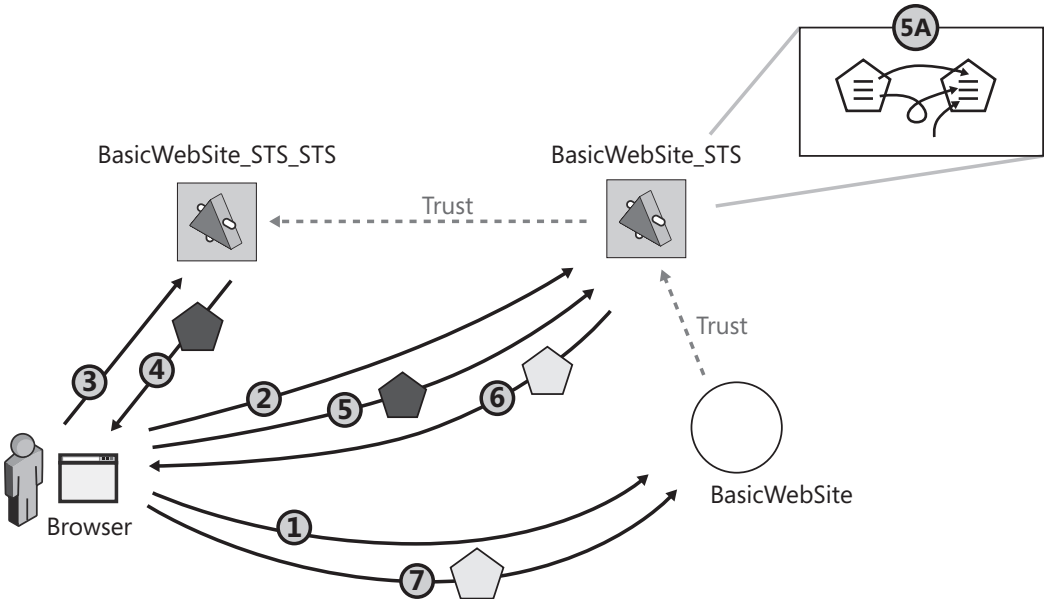


FIGURE 4-8 The authentication flow linking *BasicWebSite*, *BasicWebSite _STS*, and *BasicWebSite_STS_STS*

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.)

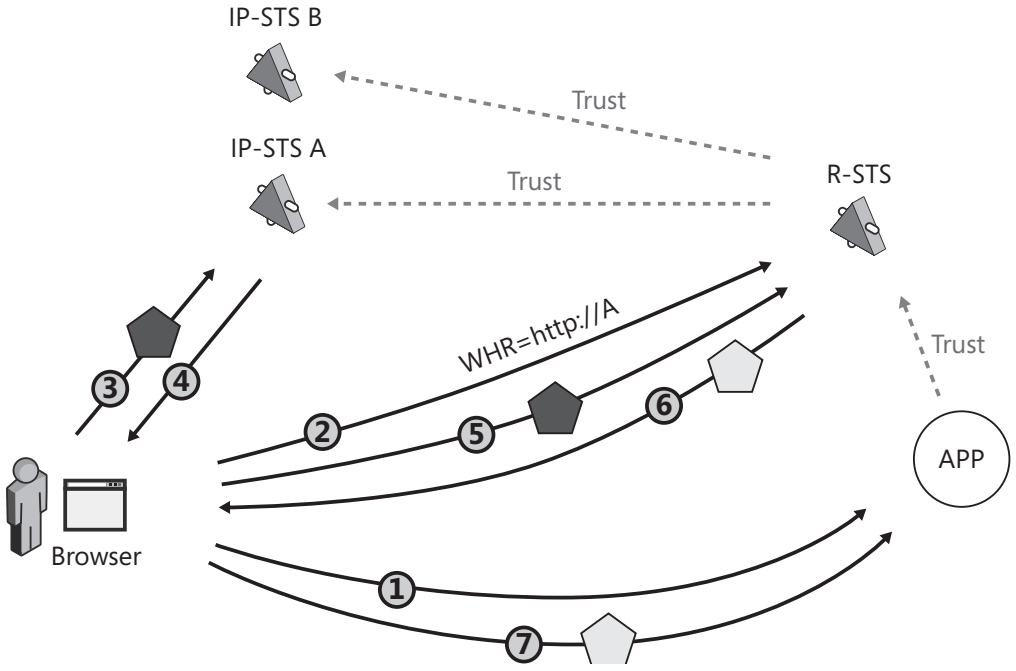


FIGURE 4-9 The Home Realm Discovery problem

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.

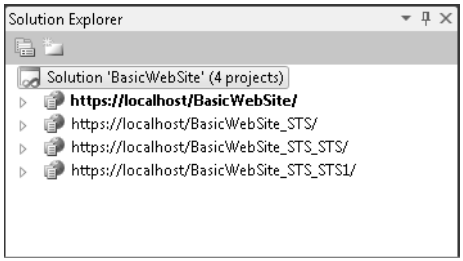


FIGURE 4-10 The sample solution showing how to handle HRD

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.

Chapter 5

WIF and WCF

In this chapter:

Windows Identity Foundation (WIF) is the ideal complement to Windows Communication Foundation (WCF) for handling authentication and authorization for services. Although you still use WCF for defining bindings and handling service details, the WIF programming model offers a much easier and more natural way of defining security requirements, working with claims in the service body, and authorizing requests before they reach their destination.

This chapter requires you to already know your way through WCF, which is a prerequisite for understanding how WIF enhances and at times supersedes existing WCF practices.

The first section, "The Basics," covers the core integration points between WIF and WCF. After describing in depth the differences between the active and passive cases, I'll show you how you can configure WCF to use WIF in its pipeline. You'll discover that most of the things you learned in the former chapters still apply here, from how to access claims from *Thread.CurrentPrincipal* to how to enforce your authorization logic via *ClaimsAuthenticationManager*. After showing you an example of a custom *SecurityTokenHandler* applied to a WCF scenario, I'll give you an in-depth tour of how WIF leverages the WCF extensibility model for wedging itself into the request processing. Those notions will be useful for WCF experts, as they clearly mark what parts of the WCF programming model can still be used and what others should be avoided in favor of the new model introduced by WIF.

The second section, "Client-Side Features," describes some of the main new features that WIF enables client-side. I'll explore the trusted subsystem pattern, highlight its shortcoming, and show how WS-Trust offers a good solution for delegation scenarios. The pattern will be used as a backdrop for the introduction of *CreateChannelActingAs*, *WSTrustChannel*, and *CreateChannelWithIssuedToken*, which is a new API that confers to you more control of how tokens are requested and used in the context of service-based solutions.

After you read this chapter, you'll be able to use WIF to help secure WCF services. You'll clearly understand the differences between the active and passive cases, both in terms of potential and requirements. You'll know how to apply to WCF scenarios the WIF

programming skills you acquired in the former chapters. You'll understand how delegation works in WS-Trust, and you'll be able to implement solutions that take advantage of delegation and impersonation. Finally, you'll learn how to take control of the token-issuance process from the client side and enable scenarios such as token caching.

The Basics

As stated many times already, the beauty of the claims-based approach to identity is that it can be used to give high-level yet accurate systems descriptions, irrespective of their implementation details. The good properties you learned about related to the IP-FP-RP pattern apply to Web services just as they did for Web applications; the design skills you acquired will come in handy in this chapter as well. Windows Identity Foundation takes advantage of this in its programming model—the claims object model is precisely the same both in ASP.NET and WCF, and the two pipelines use the same main components (token handlers, *ClaimsAuthenticationManager*, *ClaimsAuthorizationManager*, and so on) in more or less the same order.

However, there *are* practical differences between Web services and Web applications, differences that surface in the hosting model, the associated programming model, and the extra properties of solutions. Although the application developer can, in theory, remain isolated from the details and just deal with the claims model, in practice somebody eventually has to deal with bindings, behaviors, handlers for message-based activation, and all the things that make WCF development fun for true believers.

In this section about the basics, I'll equip you with the knowledge you need to apply what you have learned so far to WCF applications. That means acquiring some terminology, understanding the potential of using claims with Web services, and familiarizing yourself with the way in which WIF interlocks with the WCF object model.

Here I'm assuming you're already confortable with using WCF bindings and behaviors, although I don't assume that you understand their inner workings. If that is not the case, I have to be blunt: this section is not for you yet. If your boss told you to go and learn about Web services security, I suggest you spend some time with a good reference, such as *Programming WCF Services*, by Juval Lowy, and then come back to this chapter.

Passive vs. Active

You already encountered mentions of active and passive systems in Chapter 3, "WIF Processing Pipeline in ASP.NET," in the WS-Federation and token sections. Here I'll define things in a much more precise fashion. Web services and claims-based identity are all about the active case, although you might still develop with WIF and WCF without having a clear understanding of what actually happens. From the sheer fact you are reading this book,

I assume you want to know why we do things the way we do. Besides, this will give me a chance to introduce some much needed terminology.

Everything I've shown you so far pertains to the passive case—that is, scenarios in which the user interacts with your solutions via a browser. The browser itself is the passive element (the technical term is *passive client*) of the system. And passive it is, indeed. A browser is nothing but a rendering and execution engine for code residing on remote machines. If you recall the WS-Federation sequence you studied earlier, you have the perfect example of that. Apart from the GET kick-starting the process, everything that followed was simply the browser executing the commands received in the form of HTTP + HTML from the relying party (RP) and the identity provider (IP). The browser just bounced around without memory of what happened in the former steps or a drive of its own.

The traditional idea of a "browser" also has another limitation—its inability to perform cryptographic operations other than transport-level Secure Sockets Layer (SSL) activities. This limitation has led the industry to implement claims-based security in the passive case through the use of bearer tokens. As stated earlier, a bearer token is a token that can be used simply by attaching it to a request: the receiver does not ask more than that in order to associate the token to the requestor. The technical term for this process is a *confirmation method*—the "bearer token" is the confirmation method the relying party uses for deciding if it should believe that the token being presented really represents the current caller. (Note that this is independent from the fact that the token is valid or the user is authenticated or authorized. For example, the fact that I believe the driver's license is indeed yours is independent from the fact that the age it states makes you eligible to buy alcohol.)

The Browser Is Passive? Really?

The notion of a "dumb browser" is increasingly falling out of fashion. On one side, the current browser capabilities have proven to be sufficient for sustaining the remarkable Web applications in daily use today—JavaScript can work wonders, and it can do digital signatures as well. On the other side, there is the upcoming HTML5, which at the time of this writing promises to bring radical improvements in what can be done with sheer HTML. To make the mix even more interesting, the browser is routinely used as the preferred execution environment for rich stacks such as Microsoft Silverlight and Adobe Flash.

The strategies discussed in this book for coping with the passive case were born out of necessity, given the limitations of the browsers at the time. Today they are no longer necessary; however, they have intrinsic merits. For example, thanks to their use of core browser capabilities, they promptly interoperate across different browsers and technology stacks. Perhaps more importantly, they are widely adopted in enterprise solutions. That said, this space is in constant evolution: don't be surprised if, as the Web as a whole evolves, things change toward convergence once more.

Let's move the focus to rich clients. Applications such as Microsoft Outlook, Microsoft Excel, and TurboTax follow their own execution flow, as decided by their developers—they are *active clients*, which choose when and how to invoke remote services instead of being entirely driven by those. There's more. Whereas in the passive model authentication is mostly about starting a session with a Web site, an active client might decide to call a service so infrequently that it would not warrant the creation of a session. For example, a Twitter client would probably want to establish a session, but Microsoft Word, which infrequently calls a thesaurus or translation service, would not need to. Furthermore, an active client can entertain multiple concurrent sessions (or occasional calls) with many services from different security domains. Imagine a travel application getting flight timetables from various airlines, the weather from a different service, the highway traffic from another, the currency exchange rates from yet another, and so on.

This is a very different world from the passive cases one. Here every Web service has its own message format, and even within the boundaries of the standards (WS-Security, WS-Trust) the security requirements can vary greatly from instance to instance. Luckily, Web services have metadata of their own, describing message contracts and security policies in detail. This arrangement allows clients to build service proxies at design time, which in turn allows developers to program against the service without worrying about message serialization or the usual crypto.

Note Here I am talking in general terms. I am sure you already mapped those concepts with their counterparts in WCF speak—that is, the tool *svcutil.exe* and its corresponding Add Service Reference entry in the Microsoft Visual Studio menus.

That means that with Web services redirections are no longer necessary (although they're still technically possible in some sense) to obtain tokens from one Security Token Service (STS). When the execution of a .NET client reaches the line that invokes a Web service method, WCF already knows what it takes to secure the call. WCF will take care of everything—including contacting IPs for obtaining the necessary tokens— before engaging with the service itself.

Figure 5-1 shows a simplified sequence of how a rich client can invoke a Web service using a token obtained by one IP via WS-Trust.

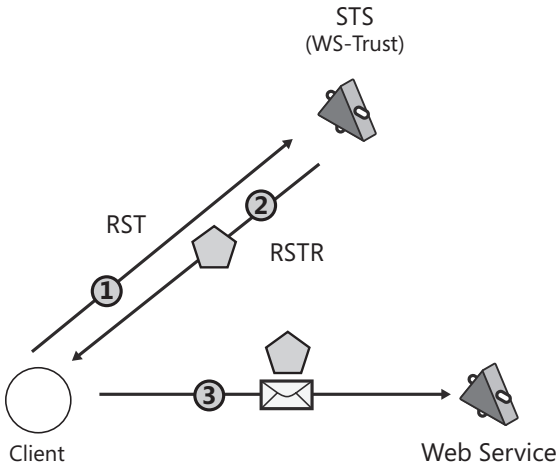


FIGURE 5-1 The basic WS-Trust flow when invoking Web services

Note It is interesting to observe how the higher capabilities of the parties at play result in a much simpler flow than the one observed with WS-Federation.

Here (and everywhere else from now on), I assume that the Web service security policy requires the caller to authenticate by presenting a token from a given IP or Federation Provider (FP). The process illustrated by Figure 5-1 can be detailed as follows:

1. The client reached a line of code that requires a call to the Web service. The library used for service calls—in our case, WCF—establishes that to invoke the service, it is necessary to obtain a token from a given STS. An STS is in itself a Web service, which understands a standard set of messages described by the WS-Trust specification. In this case, the method called is *Issue*, and the message it requires is known as a *Request for Security Token* (*RST*). The RST contains all the information the STS needs for issuing the token as required by the RP. (I'll discuss RST in depth later in the chapter.) The requests to the STS can be secured using any method admitted by WS-Security, which typically reflects the way in which the corresponding IP authenticates its users. Kerberos, user name and password, X.509, and even SAML tokens are all examples of classic credential types used for securing RSTs.
2. The STS successfully authenticates the call and proceeds to issue a token as requested by the RST. The token is sent back to the requestor as part of a *Request for Security Token Response* (*RSTR*) message.

3. The client uses the token for securing the message that constitutes the call to the service requested by the code. The primary effect of the call should be to obtain the result expected from the method, although the call could also be used for establishing a session that allows subsequent calls to take place without further involvement of the STS.

The details behind the phrase "the client uses the token for securing the message" constitute the main difference between passive and active cases. Whereas in the passive case the message flow relies on the transport (read: HTTPS) to provide confidentiality and protection, active clients have the further option of handling security at the message level. This basically means that WS-Security offers mechanisms that can sign and encrypt the message itself, in part or entirely, so that confidentiality and integrity can be guaranteed regardless of the transport used. This holds even for data-at-rest cases, when the message is saved as part of audit trails. The literature on the subject refers to this style as *message-based security*—that is the quintessential defining feature for active clients, because traditional passive clients simply do not have the necessary capabilities.

Message vs. Transport Security

The fact that a rich client can use message-based security does not necessarily imply that it will, or even that it always should. Message security comes with a price: the complexity of the libraries it requires. Not every platform or environment has the necessary capabilities—Silverlight and mobile devices being two good examples of that. WCF admits a mixed mode in which message-based and transport-based security are combined, which is often a good compromise. Pure transport security is a perfectly acceptable option for all cases in which sophisticated guarantees such as end-to-end security or nonrepudiation are not necessary. However, it would be awkward for me to include those considerations in every explanation; hence, from now on I'll identify the active case with the message-based security case. As you read through the rest of the chapter, just keep in mind that things are more generic than that.

Message security has various interesting properties, such as end-to-end security. A message encrypted for a given ultimate recipient can travel through multiple intermediaries, such as message routers, without disclosing anything in transit. That is not the case with SSL forwarders, where security is guaranteed point-to-point from the client to the forwarder only. Another property is nonrepudiation applied to single messages. If the message to a purchasing service is signed with the private key corresponding to the certificate of the client, the details of the transaction cannot be tampered with and the buyer cannot pretend at a later time that she didn't intend to buy what the message in the merchant's record states.

The encryption for end-to-end security and the signature for nonrepudiation operations imply that in the active case tokens are not just *attached* to a message—they are *used* as the

cryptographic material for actually modifying the message being sent. Whereas a passive relying party is content with the requestor simply including the token in the call, an active RP expects the requestor to use the token to recognize its association. In SAML terms, we say that in the active case the confirmation method is *holder-of-key* (as opposed to the passive case's bearer token). Holder-of-key tokens are especially useful against man-in-the-middle attacks, because simply stealing the bits of a token in transit does not grant the ability to use the key or keys it carries.

This has implications for the software used for enabling claims-based identity in Web services scenarios. An STS must issue tokens that contain not only claims, but keys as well; a client needs to be able to use those keys for modifying the messages; and an RP needs to be able to verify the use of the keys in the incoming tokens on the messages. WCF and WIF do a superb job of keeping the details of this hidden, but occasionally the abstraction leaks. If you plan to do a lot of work with Web services, being aware of the issues mentioned in the preceding discussion can occasionally save the day.

Proof Keys and a Deep Dive into the Active Case Flow

This section already delivered more theory than many readers would like, which is why I am enclosing the following in a sidebar. A sidebar makes it easier for you to skip the content if you have had enough of the topic and you want to get straight to the code.

I find it very useful to have a clear mental picture of what happens in the active case flow, especially in terms of which keys are used and where. Once you understand that, certain things about the solutions architecture become absolutely intuitive—you don't need to waste any more cycles trying to understand which certificates should be distributed where, why the customer should not worry about man-in-the-middle attacks, and so on. In this sidebar, I'll revisit the flow shown in Figure 5-1, this time factoring in the merits of the token structure and use. Many developers (or even architects) live full, happy lives without knowing things at this level of detail—feel free to skip this in its entirety if you feel you don't need the extra insight this would provide you. Figure 5-2 constitutes the basis for the following discussion. Note that not all the cryptographic operations actually taking place in the real scenario are depicted. The figures show only the ones that are useful for understanding this specific flow.

Note You'll often hear or read people using the expression "cracking a token" for indicating the act of decrypting a token. I hate that expression because "cracking" implies that you can somehow force the token open, although you are in fact diligently using the appropriate key for decrypting it. Unless some big advancement in quantum computing occurred, or something went very wrong in key distribution, there is no such a thing as "token cracking." Now you know how to spoil my mood if you meet me at a party.

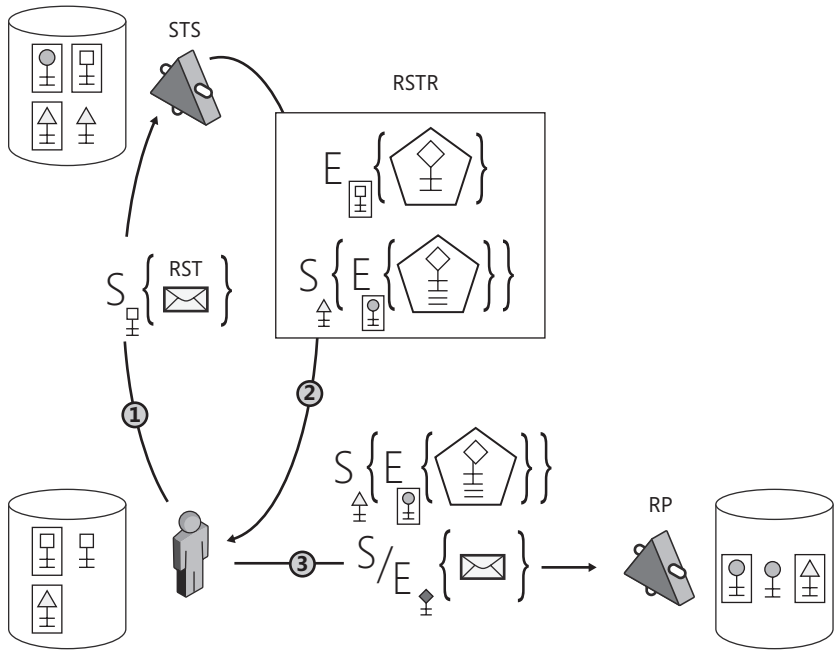


FIGURE 5-2 The WS-Trust flow highlighting the use of keys

The actors are the usual: the user, RP, and IP. However, this time I added some extra information—namely, the certificate and private key store of every role. As I walk you through the sequence, the criteria used to deploy certain keys on given machines will become clear. Keys can be public (represented in Figure 5-2 as the items enclosed in a rectangle, which include the X.509 certificate that contains the public key bits), or they can be private. Different keys can be identified by the shape of their bow: because a private key can be installed only at its legitimate owner's machine, in this case the STS has the triangle key, the RP has the round key, and the user has the square one.

Note In this sequence, I am assuming that the user authenticates with the STS using an X.509 certificate, as is the case when using smartcards. As strange as it might sound, it makes things easier to explain than if the user had used a user name and password.

Let's examine every step.

1. The user agent (the rich client application) sends an RST message to the STS that contains the details of the token required for invoking RP. As usual, the STS can mandate whatever authentication mechanism it prefers—in this case, I chose to have the user sign something with his private key. Note that the call would likely have some extra security measure, such as sending the request over SSL or encrypting the message for the STS. That's the reason the client certificate store also has a copy of the STS certificate, the triangle key.

2. The STS authenticates the request. To that purpose, it needs to know the public key of the user to verify the request signature and determine whether it comes from a well-known source. (Technically, that's not always the case. Sometimes the certification authority is enough, and the certificate might arrive with the request, but let's keep things simple.)

   After successful user authentication has taken place, the STS verifies that the intended RP is recognized. (That means the RP certificate, the round key, should be installed in the STS certificate store; however, remember the caveats for nonauditing STS that were discussed in Chapter 4, "Advanced ASP.Net Programming"). If it is, the STS crafts a token with the requested claims, in the required format. The STS also generates a new key, for simplicity let's say symmetric, which ends up inside the token as well.

   The new key, represented with a diamond bow, is needed for supporting the holder-of-key confirmation method. The STS then goes ahead and signs the bits with its own private triangle key and encrypts the token with the RP's round public key. The encryption makes the token opaque to everybody but the RP, including to the requestor itself. That is actually a problem—the diamond key is now inaccessible, buried inside a token encrypted for someone else, but the client needs to demonstrate the ability of using the key in order to comply with the confirmation method. That's where the famous *proof token* comes in. In addition to the requested token, the STS adds to the RSTR an extra token, called a proof token because it is what enables the client to demonstrate *proof of possession* of the diamond key. The proof token contains the same diamond key carried in the main token, but it is encrypted for the client with its square key. This enables the client to use the diamond key in the next step.

3. The user agent crafts the message for the service method it intends to call, and then it proceeds to include in the WS-Security header the token obtained from the STS and somehow use the diamond key from the proof token for doing something on the message itself. It could be pretty much anything, from signing the message body and the WS-Addressing headers to initiating the creation of a security context token, which will derive a series of keys. The details do not matter here. What's important is that the client does something that can be done only by using the diamond key so that the RP can verify it is in the requestor's possession. That's exactly what the RP does—upon receiving the token, the RP verifies it is signed from the intended STS, that it is encrypted for itself (along with similar noncrypto checks such as verifying the *AudienceRestriction*, as discussed in Chapter 3), and that the diamond key found within the token has been used on the message in a way that unequivocally proves the caller had access to it.

That's it—it's not the easiest bit of the book, but I am sure that now that you went through it you are glad you did it. As mentioned, reality is a bit more complicated than what I portrayed, but this discussion should be enough for you to understand the essence of what a proof key is and why it is useful. Now you see how a man-in-the-middle attack in the user-to-RP leg has very few chances of succeeding.

Canonical Scenario

After getting through that preamble, you probably expect that coding with WCF and WIF is rocket surgery. At least for the basic scenario, that is absolutely not the case. In fact, the flow is remarkably similar to what happens in the ASP.NET case. Let's walk through it—it won't take long.

The Service

Create a new Web site in Visual Studio 2010, and pick the WCF Service template. You can choose an http address this time—https is not required.

Note I'll use Web sites rather than self-hosted services because it's just handier to have the service always listening, especially when you work with metadata. If you got this far in the book, it is reasonable to assume you do have IIS installed! In any case, the self-hosted case is pretty much the same, plus the usual *ServiceHost* juggling.

Right-click on the new Web site in Solution Explorer. You'll find the familiar Add STS Reference entry. That leads you through the same wizard you encountered in the ASP.NET scenario, with only an extra screen that helps you determine which service you are working with in case the project includes multiple endpoints. If you elect to create a new test STS on the spot, the wizard experience is exactly the same as its passive counterpart. Figure 5-3 shows the status of the solution after having added the STS.

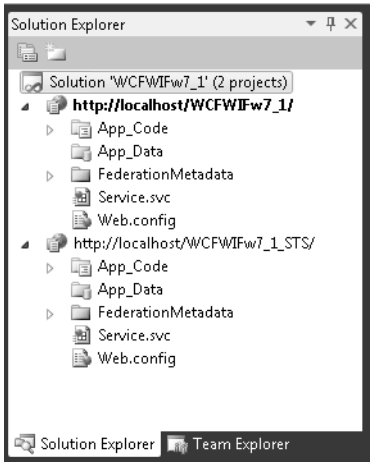


FIGURE 5-3 The solution after having added a test STS for the WCF service

As usual, apart from the automatic metadata generation, the effect of the wizard is to modify the RP's config file to outsource authentication to the STS. Let's take a look at the relevant parts of the new Web.config of the service, from the bottom up. Here's the freshly added *<microsoft.identityModel>* section:

```
<microsoft.identityModel>
  <service>
    <audienceUris>
      <add value="http://localhost/WCFWIFw7_1/Service.svc"/>
    </audienceUris>
    <issuerNameRegistry
        type="Microsoft.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
              Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
              PublicKeyToken=31bf3856ad364e35">
      <trustedIssuers>
        <add thumbprint="9DABA69D4CE3DD9BFFC5CFB53D60E3E9035C74A9"
             name="http://localhost/WCFWIFw7_1_STS/Service.svc"/>
      </trustedIssuers>
    </issuerNameRegistry>
  </service>
</microsoft.identityModel>
```

You recognize all the elements in there, right? The *<audienceUri>* is the service itself, and *<issuerNameRegistry>* keeps track of the certificate necessary for checking the IP signature. In the passive case, *<microsoft.identityModel>* contained a lot of extra information—namely, all the things that were necessary for getting the WS-Federation flow going: address of the issuer, requirements such as the use of SSL, parameters such as *whr*, and so on.

Note Remember, the STS address in the *<trustedIssuers>* is not really interpreted by WIF as an address. It is the mnemonic name that indicates the issuer in the code—for example, in the *Issuer* or *OriginalIssuer* properties of the *Claim* class. That's just the default name that WIF picked for the IP.

The reason is that in the services world WIF does not drive the protocol—that is the prerogative of WCF. It is the WCF configuration that determines which IP should be used, how to authenticate RST messages, what kind of token should be requested, and how to use that token with the RP. Let's take a look at the WCF configuration that is generated by default by WIF:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <federatedServiceHostConfiguration/>
        <!-- To avoid disclosing metadata information, set the value below to false
             and remove the metadata endpoint above before deployment -->
        <serviceMetadata httpGetEnabled="true"/>
        <!-- To receive exception details in faults for debugging purposes,
             set the value below to true.
             Set to false before deployment to avoid disclosing exception information -->
        <serviceDebug includeExceptionDetailInFaults="false"/>
        <serviceCredentials>
          <!--Certificate added by FedUtil.
              Subject='CN=DefaultApplicationCertificate',
              Issuer='CN=DefaultApplicationCertificate'.-->
          <serviceCertificate findValue="760D78BE577699D3B6F6BCC7ABD25B13BCAFEF23"
                              storeLocation="LocalMachine"
                              storeName="My"
                              x509FindType="FindByThumbprint"/>
        </serviceCredentials>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <serviceHostingEnvironment multipleSiteBindingsEnabled="true"/>
  <extensions>
    <behaviorExtensions>
      <add name="federatedServiceHostConfiguration"
   type="Microsoft.IdentityModel.Configuration.ConfigureServiceHostBehaviorExtensionElement,
       Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
       PublicKeyToken=31bf3856ad364e35"/>
    </behaviorExtensions>
  </extensions>
  <protocolMapping>
    <add scheme="http" binding="ws2007FederationHttpBinding"/>
  </protocolMapping>
  <bindings>
    <ws2007FederationHttpBinding>
      <binding>
        <security mode="Message">
```

```
          <message>
            <issuerMetadata address="http://localhost/WCFWIFw7_1_STS/Service.svc/mex"/>
            <claimTypeRequirements>
              <!--Following are the claims offered by STS
                 'http://localhost/WCFWIFw7_1_STS/Service.svc'.
                  Add or uncomment claims that you require by your application
                  and then update the federation metadata of this application.-->
              <add claimType="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
                   isOptional="true"/>
              <add claimType="http://schemas.microsoft.com/ws/2008/06/identity/claims/role"
                   isOptional="true"/>
            </claimTypeRequirements>
          </message>
        </security>
      </binding>
    </ws2007FederationHttpBinding>
  </bindings>
</system.serviceModel>
```

That looks like quite a mouthful, but if you are used to WCF that's actually a pretty concise *<system.serviceModel>*.

The first interesting thing to notice is the *<federatedServiceHostConfiguration/>* behavior and its configuration:

```
<behaviorExtensions>
    <add name="federatedServiceHostConfiguration"
type="Microsoft.IdentityModel.Configuration.ConfigureServiceHostBehaviorExtensionElement,
      Microsoft.IdentityModel,
      Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
</behaviorExtensions>
```

ConfigureServiceHostBehaviorExtensionElement, besides being the longest configuration element name I've ever seen, is the mechanism that WIF uses for wedging itself into the WCF request processing pipeline. I'll go into more details about how it accomplishes that in the next section.

The rest of the *<behavior>* element is fairly standard: turning on metadata publication—turning off exception details, and indicating the service certificate. The *<serviceHostingEnvironment>* element (new in .NET 4) enables the service to listen on multiple addresses at the same time, which is not especially useful in our scenario.

The *<protocolMapping>* element is another improvement introduced by WCF 4 that can associate one of the default WCF bindings to a protocol schema—in this case, HTTP. Thanks to this new functionality there is no need to explicitly define an endpoint in the config because the desired binding will be blanket-applied to all HTTP endpoints. The binding of choice is *ws2007FederationHttpBinding*, which expects the caller's credentials in the form of an issued token—in this case, using message-based security.

The <*message*> element gives some important requirements about the token. The first one is the metadata address of the token issuer, followed by the list of claim types that should be included in the incoming token.

The WIF STS Template for WCF

That's pretty much it for what concerns the RP. Let's now focus on the active STS generated by WIF. The STS project is remarkably similar to the one generated by the passive template; in fact, the content of the *App_Code* folder is pretty much the same. Of course, there are no ASPX pages here. There is a single *Service.svc* file containing the following line:

```
<%@ ServiceHost Language="C#" Debug="true"
    Factory="Microsoft.IdentityModel.Protocols.WSTrust.WSTrustServiceHostFactory"
    Service="CustomSecurityTokenServiceConfiguration"  %>
```

The code here feeds the custom STS config class, *CustomSecurityTokenServiceConfiguration*, to *WSTrustServiceHostFactory*, a service factory that is offered by WIF for initializing WS-Trust services and integrating in them the logic defined in the *SecurityTokenService* subclass. Once again, the interesting parts are in the web.config and precisely in the <*system.serviceModel/*> element.

```
  <system.serviceModel>
  <services>
    <service name="Microsoft.IdentityModel.Protocols.WSTrust.WSTrustServiceContract"
             behaviorConfiguration="ServiceBehavior">
    <endpoint address="IWSTrust13" binding="ws2007HttpBinding"
              contract="Microsoft.IdentityModel.Protocols.WSTrust.IWSTrust13SyncContract"
              bindingConfiguration="ws2007HttpBindingConfiguration"/>
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost/WCFWIFw7_1_STS/Service.svc" />
      </baseAddresses>
    </host>
    <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
    </service>
  </services>
  <bindings>
    <ws2007HttpBinding>
    <binding name="ws2007HttpBindingConfiguration">
      <security mode="Message">
      <message establishSecurityContext="false" />
      </security>
    </binding>
    </ws2007HttpBinding>
  </bindings>
```

```
<behaviors>
  <serviceBehaviors>
  <behavior name="ServiceBehavior">
    <!-- To avoid disclosing metadata information, set the value below to false
         and remove the metadata endpoint above before deployment -->
    <serviceMetadata httpGetEnabled="true" />
    <!-- To receive exception details in faults for debugging purposes,
        set the value below to true.
        Set to false before deployment to avoid disclosing exception information -->
    <serviceDebug includeExceptionDetailInFaults="false" />
  </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
```

The service implementation comes from *WSTrustServiceContract*, a class that implements various flavors of WS-Trust (1.3 and 2005.) The use of the *IWSTrust13SyncContract* selects the WS-Trust 1.3 implementation.

The binding itself is straightforward. The use of *ws2007HttpBinding* means that the authentication with the STS will take place via NTLM or Kerberos. For a demo that's absolutely great, there will be no prompt for obtaining a token from the STS.

Invoking the Service

It's time to give our service a spin. In the first release of the .NET Framework 3.5, WCF introduced a great tool for testing services without having to write a client for them. The name of the tool is *WcfTestClient.exe*, and it usually can be found in *%ProgramFiles%\Microsoft Visual Studio 9.0\Common7\IDE*. It's easy to use—you just enter the service address via File/Add Service and the tool will generate a client on the fly, giving you the chance to specify some input value and call the service. All you need to do is double-click the name of the method you want to call and enter the desired values.

Note *WcfTestClient.exe* does not support any kind of credential gathering; hence, it works only with services secured via NLTM, Kerberos, or issued tokens. In our scenario, that works. However, as soon as I move beyond that later in the chapter, I'll have to create clients in order to test the services.

Figure 5-4 shows *WcfTestClient* used for invoking the sample service.

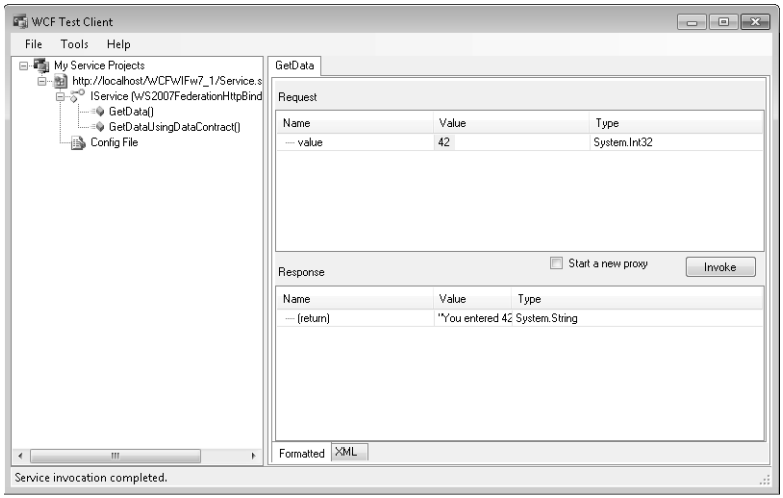


FIGURE 5-4 *WcfTestClient* used for invoking the sample service

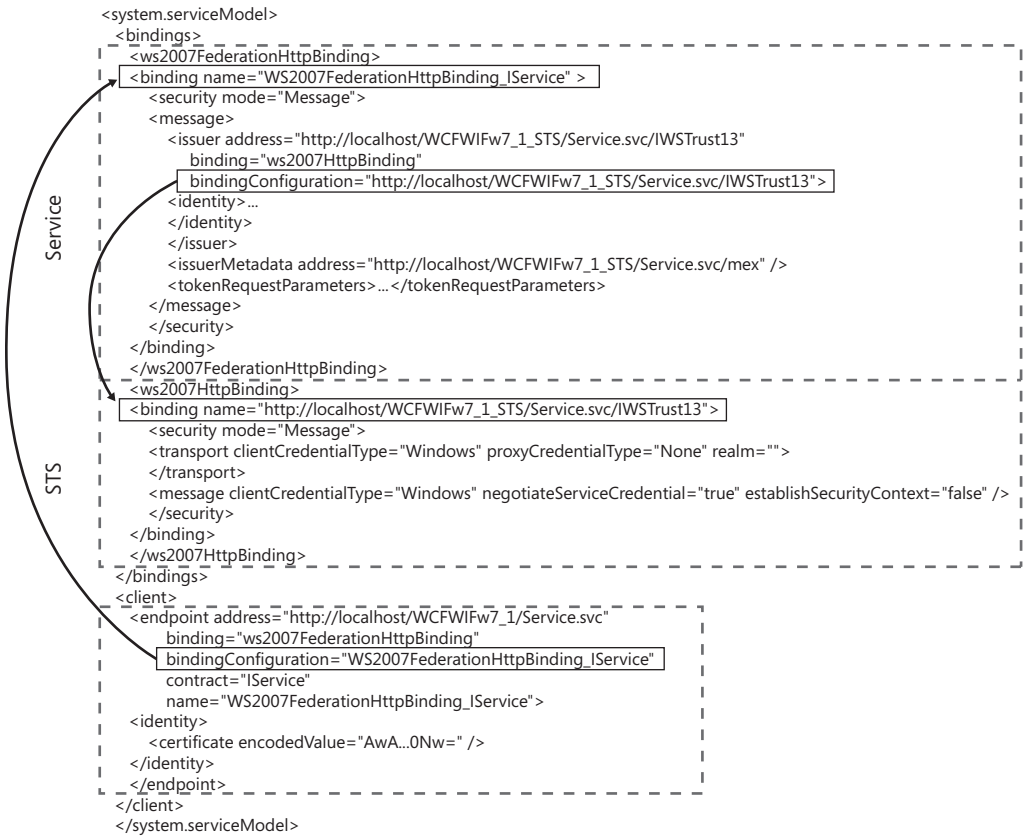
If you try to call one method—for example, *GetData*,—you'll get the result without any indication that an STS was involved in the process. Again, that is because the WIF STS template for the active case uses *ws2007HttpBinding* and, hence, Windows authentication.

WIF Is (Usually) Not Needed on the Client

The thing that might surprise you a bit is that *WcfTestClient* has absolutely no knowledge of WIF! *WcfTestClient* has shipped with Visual Studio since the 2008 version, and even the version in Visual Studio 2010 has no references whatsoever to the WIF assemblies. If you think about it, that's exactly as it should be. When you create a client for a Web service, you should not even need to know on which platform the service runs. As long as everybody sticks to the standards, a WCF client is able to transparently invoke a Java service. In the same way, the fact that the STS is implemented using a WIF class and the service uses WIF for validating the incoming token does not really have any influence on the client. A client built on the .NET Framework 3.5 SP1 has all it needs for calling a WIF-secured WCF service. That's good news if you are still targeting Windows XP machines—WIF is not supported on XP, but you don't need it anyway for consuming WIF services.

Important There are features introduced by WIF that can make life easier when developing clients. If you elect to use those features, of course you'll create a dependency between your client and WIF and you'll need to rely on its presence on the target machines accordingly. I'll go through those features later in the chapter.

Let's take a look at the WCF configuration needed on the client for calling the service with a token from the STS template. Another handy feature of *WcfTestClient* is that it offers the possibility of examining the config file it generated for the client from the service metadata. The one generated for the sample service is shown here, edited for clarity:



Examining the file from the bottom up, the first element you encounter is the <*client*>. The binding for the service is *ws2007FederationHttpBinding*, which corresponds to the settings on the service side. On the service side, the binding just deferred to the STS metadata, without mandating any specific kind of authentication for requesting tokens. The client is on point to actually send the RST message; hence, there is a need to specify further how to engage with the STS.

Jumping up to the <*ws2007FederationHttpBinding*> element, you can observe that it contains a full-fledged <*issuer*> element with all the information necessary to invoke the STS as a service: the service endpoint, which bindings should be used, and the binding configuration. The <*tokenRequestParameter*>, edited out in the preceding code for clarity, contains

indications for the STS about the requested token: a list of the required claims, the type and size of the key, which algorithms should be used for signing and encrypting the token, and so on.

The binding indicated for invoking the STS, as you already know, is *ws2007HttpBinding*. *WcfTestClient* uses *SvcUtil* for generating the client configuration. *SvcUtil* starts by analyzing the service policies and then crawls via the pointer to the STS metadata location to resolve the STS binding and embed it in the client configuration as appropriate.

Using Claims with a WCF Service

Did you dismiss all the rambling about WIF having a consistent model across ASP.NET and WCF solutions as pure marketing? Here's your chance to experience firsthand that it is true.

Accessing claims from within the service code is absolutely equivalent to what you have done with ASP.NET. The following code changes the default WCF service implementation (the *Service.cs* file in the *App_Code* folder of the service project) to use some claims in the method results:

```
using Microsoft.IdentityModel.Claims;
using System.Threading;

// NOTE: You can use the "Rename" command on the "Refactor" menu
// to change the class name "Service" in code, svc and config file together.
public class Service : IService
{
  public string GetData(int value)
  {
    IClaimsIdentity ici = Thread.CurrentPrincipal.Identity as IClaimsIdentity;
    string role = (from c in ici.Claims
                   where c.ClaimType == ClaimTypes.Role
                   select c.Value).FirstOrDefault();

    return string.Format("You entered {0}, dear {1}, {2}," value, ici.Name, role);
  }
```

If you run *WcfTestClient* again, you'll see that the claim values are used in the return string as expected.

All the obvious extensibility points in the claims processing pipeline are available in WCF as well. For example, here you can see a trivial *ClaimsAuthorizationManager* implementation:

```
public class MyClaimsAuthorizationManager: ClaimsAuthorizationManager
{
  public override bool CheckAccess(AuthorizationContext context)
  {
    IClaimsIdentity ici = context.Principal.Identity as IClaimsIdentity;
    string nm = (from c in ici.Claims
                 where c.ClaimType == ClaimTypes.Name
                 select c.Value).FirstOrDefault();
    return (nm.ToLower() == "earnest");
  }
}
```

The policy here is to allow access only to the users whose name is "Earnest". Wedging this access check in front of the service is done in exactly the same way as you did it in ASP.NET—by adding a <*ClaimsAuthorizationManager*> entry in the <*microsoft.identityModel/service*> element as shown below. Of course, here there's no need to add the *ClaimsAuthorizationModule*, but you can chalk it down as one difference that is rooted in the different hosting models.

```
<microsoft.identityModel>
    <service>
      <audienceUris>
        <add value="http://localhost/WCFWIFw7_1/Service.svc"/>
      </audienceUris>
      <claimsAuthorizationManager type="MyClaimsAuthorizationManager"/>
```

I could go on and on, but you get the idea—using WIF for customization and authorization in WCF means applying many of the techniques you have already learned. This holds true as long as authentication is outsourced to an STS. You can also use WIF with WCF for handling raw credentials directly at the service. In that case, you need to resort to crafting the appropriate WCF bindings and customizing token handlers. That holds true, naturally, also in the case in which you are responsible for writing an STS.

Custom TokenHandlers

As you have seen, the active STS template uses Windows authentication for releasing tokens. That's handy for quick checks and demos, but it is not especially useful if you want to test your solution on the Internet, across different machines, while simulating users,

and so on. For that, you need something less tied to the infrastructure—for example, user name and password or X.509 certificates. Showing how to craft WCF bindings and custom *SecurityTokenHandler*s is beyond the scope of this book, but I feel that it is useful to discuss the topic a bit.

Let's say you want to have an STS that authenticates users using a user name and password. The first thing you need to do is write an appropriate binding. You can start by adding another STS in the solution you created earlier, using the WCF STS template. Let's use an SSL channel for sending the RST. All you need to do on the STS web.config is change the protocol schema of *baseAddress* to **https** and change the *ws2007HttpBinding* in the following:

```
<ws2007HttpBinding>
<binding name="ws2007HttpBindingConfiguration">
  <security mode="TransportWithMessageCredential">
  <message clientCredentialType="UserName"/>
  </security>
</binding>
</ws2007HttpBinding>
```

Note Of course, your IIS must be configured to have an HTTPS binding for this to work.

Here you see the wisdom of separating the STS-proper functions from the authentication mechanism: you don't have to change anything in the STS class to accommodate this possibility. (Granted, the STS template does not do much to begin with.)

You do have to change the way in which you handle the credentials verification, from Windows to arbitrary user name and password. Whereas in classic WCF you would use some special API tied to the kind of credentials, in this case you would probably throw in a *UsernamePasswordValidator*. WIF offers a consistent approach: you modify the *SecurityTokenHandler* corresponding to the credential types you expect.

As a WCF expert, you are probably frowning at the thought. After all, specializing the way in which a token is handled in WCF means dealing with a rather Byzantine mechanism that requires you to subclass *WSSecurityTokenSerializer*, *SecurityTokenProvider*, *SecurityTokenAuthenticator*, *xxxCredentialsSecurityTokenManager*, and a few others.

WIF changes all this. All the functionalities implemented by those classes are now subsumed in a single *xxxSecurityTokenHandler*. This has obvious advantages when you need to do small modifications to the default behavior. More often than not, you just need to override a method to obtain the effect you want. Next I'll show you a trivial example of how

you can weave arbitrary user name and password verification logic via your own version of *UserNameSecurityTokenHandler*: It is really easy:

```
using System;
using System.IdentityModel.Tokens;
using Microsoft.IdentityModel.Claims;
using Microsoft.IdentityModel.Protocols.WSIdentity;
using Microsoft.IdentityModel.Tokens;

class CustomUserNameSecurityTokenHandler : UserNameSecurityTokenHandler
{
  public override bool CanValidateToken
  {
    get
    {
      return true;
    }
  }

  public override ClaimsIdentityCollection ValidateToken(SecurityToken token)
  {
    UserNameSecurityToken usernameToken = token as UserNameSecurityToken;
    if (usernameToken == null)
    {
      throw new ArgumentException("usernameToken",
                              "The security token is not a valid username security token.");
    }

    string username = usernameToken.UserName;
    string password = usernameToken.Password;

    if (("paul" == username && "p@sswOrd" == password) ||
        ("john" == username && "p@sswOrd" == password))
    {
      IClaimsIdentity identity = new ClaimsIdentity();
      identity.Claims.Add(new Claim(WSIdentityConstants.ClaimTypes.Name, username));

      return new ClaimsIdentityCollection(new IClaimsIdentity[] { identity });
    }

    throw new InvalidOperationException("The username/password is incorrect");
  }
}
```

Here I just override *ValidateToken* because I am happy with how the base implementation deals with everything else, but nothing would prevent me from doing a more complete overhaul. At the end of Chapter 3, I gave a list of the methods you most often need to override in your own *SecurityTokenHandler* implementations for achieving specific results.

The mechanism you use for substituting (or integrating) token handlers in the default collection is configuration, which is precisely the same mechanism you use in ASP.NET:

```
<microsoft.identityModel>
<service>
  <securityTokenHandlers>
  <remove type="Microsoft.IdentityModel.Tokens.WindowsUserNameSecurityTokenHandler,
               Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
               PublicKeyToken=31BF3856AD364E35"/>
  <add type="CustomUserNameSecurityTokenHandler, App_Code"/>
  </securityTokenHandlers>
</service>
</microsoft.identityModel>
```

Consuming the service is straightforward; however, this time you do need to create a client.

You can simply create a console application in the same solution, and add a service reference via the usual Visual Studio tools. I'll spare you the details of the resulting configuration; however, the main differences from the one seen in the former example are the presence of HTTPS in all references to the STS endpoint (make sure they're there, the tool does not always do it) and the binding, of course:

```
<issuer address=https://localhost/UPSTSService/Service.svc/IWSTrust13
        binding="ws2007HttpBinding"
        bindingConfiguration="https://localhost/UPSTSService/Service.svc/IWSTrust13" >
  <identity>
    <dns value="CN=localhost"/>
  </identity>
</issuer>
```

Invoking the service is just a matter of feeding in the correct credentials before performing the call. In the following example, I turn off the certificate validation because I am using a self-signed certificate; that is something that you would not do on a production system, of course:

```
class Program
{
  static void Main(string[] args)
  {
    ServiceReference1.ServiceClient sc = new ServiceReference1.ServiceClient();
    sc.ClientCredentials.UserName.UserName = "john";
    sc.ClientCredentials.UserName.Password = "p@ssw0rd";
    sc.ClientCredentials.ServiceCertificate.Authentication.CertificateValidationMode =
                                                  X509CertificateValidationMode.None;
    Console.WriteLine("calling the service");
    Console.ReadLine();
    try
    {
      Console.WriteLine("the service says {0}", sc.GetData(42));
    }
```

```
        catch (Exception ee)
        {
          Console.WriteLine("the service says {0}", ee.ToString());
        }
        finally
        {
          Console.ReadLine();
        }
      }
    }
```

Note If you still have the *ClaimsAuthorizationManager* from the earlier sample, this time you'll get an "access denied" message as the user presents a different *Name* claim. That underscores, once again, how easy it is to switch from one IP to another without even touching the RP application.

The hands-on lab "Web Services and Identity" (c:\IdentityTrainingKit2010\Labs\WebServicesAndIdentity) exercise 1, demonstrates how to implement the Username and Password authentication scenario using WIF within WCF: The only difference is that here authentication was brokered by an STS, whereas in the exercise it is the service itself that directly verifies the incoming credentials. The object model is, however, exactly the same. If you want to experiment with the custom X509SecurityTokenHandler*, you can get inspiration from the FabrikamShipping sample (http://code.msdn.microsoft.com/FabrikamShipping), and precisely from its ActAs STS (more details later in the chapter).*

Object Model and Activation

This section is for WCF experts. If you are already familiar with the WCF security model, here I'll help you cope with the differences that the WIF model introduces.

Note Even if you are an expert, it is easy to get lost in all those passages. I'll readily admit that it happens to me as well from time to time. One trick that works really well for me is activating the WCF and WIF tracing and then observing the sequence of events in detail. All you need to do to turn on WIF tracing is follow the same procedure you would do with WCF, using the source *Microsoft.IdentityModel*. Note that tracking is really useful in passive scenarios as well!

To make a long story short: WIF uses the publicly available WCF extensibility points for wedging itself into the request processing pipeline. Namely, it provides its own *ServiceAuthorizationManager*, an *IServiceBehavior* implementation, and its own *SecurityTokenManager* class. "Wedging itself" basically means that the actual token processing is performed by the WIF *SecurityTokenHandler* classes, that the rest of the WIF pipeline (*ClaimsAuthorizationManager*, *ClaimsAuthenticationManager*) is executed as expected, and

that the service ends up with an *IClaimsPrincipal* in the *Thread.CurrentPrincipal* property. Next I'll give a quick description of how that happens, in the hopes that it will help you to overcome the temptation to touch things at the underlying WCF level instead of sticking with the WIF programming model.

In the last section, you saw that a way of adding WIF to the WCF pipeline is leveraging the *ConfigureServiceHostBehaviorExtensionElement* element, which in turn hooks the behavior extension *ConfigureServiceHostServiceBehavior*. The main thing the behavior extension does, besides hooking the *IServiceBehavior* implementation *FederatedServiceCredentials*, is call on the current *ServiceHost* the static method *FederatedServiceCredentials.ConfigureServiceHost*. That is also the method you call for configuring the service to use WIF if you have direct visibility of the *ServiceHost*—for example, when writing a service factory or for self-hosted services.

FederatedServiceCredentials instantiates WIF's *SecurityTokenManager*, *FederatedSecurityTokenManager*, as its token manager class. This allows WIF to take complete control over the way in which the token is processed. Whenever the flow requires one of the WCF subclasses used for processing tokens, such as serializers or authenticators, *FederatedSecurityTokenManager* instantiates wrapper classes that will delegate the operations to the *SecurityTokenHandler* configured in WIF.

Note It is somewhat ironic (but perfectly valid from an engineering standpoint) that on many occasions the *SecurityTokenHandler* handlers offered by WIF out of the box will end up calling some of the existing WCF classes, closing the circle.

Remember that the *SecurityTokenHandler* classes used here are precisely the same ones described in the sign-in flow in Chapter 3. The subject confirmation code might activate execution paths that are unused in the passive case; however, in terms of the pipeline, you can expect the same stages and events to take place in the active case as well. For example, the *SecurityTokenHandler* is what takes care of invoking your *ClaimsAuthenticationManager* class if you configured one and assigning the *IClaimsPrincipal* to the current principal.

FederatedServiceCredentials.ConfigureServiceHost assigns the WIF *ServiceAuthorizationManager* class, *IdentityModelServiceAuthorizationManager*. That is the class responsible for executing *ClaimsAuthorizationManager*.

Figure 5-5 summarizes the WIF-WCF pipeline integration.

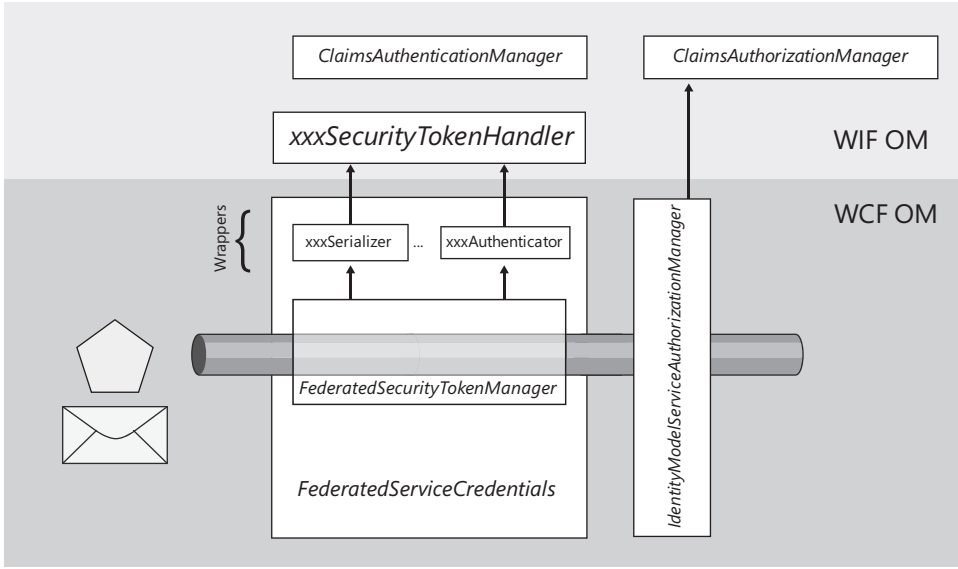


FIGURE 5-5 How WIF integrates in the WCF pipeline by leveraging standard WCF extensibility points

Don't worry if you didn't follow everything. As I mentioned at the beginning, this section is for experts only. In any case, in the following subsection I'll outline some of the practical consequences of this model regarding the way you should change your use of the WCF API.

Things You Do Differently in WCF When You Use WIF

WCF supported claims from its very first version. Developers used to find claims information (the famous *ClaimSet*) in the thread local *ServiceSecurityContext.Current.Claims*. Although it was a good first attempt, the WCF claim model had very limited integration with how the rest of the .NET platform handles identity. As a result, when you use WIF with WCF you should get your caller's claim information from *IClaimsPrincipal* in *Thread.CurrentPrincipal* as usual, and avoid using the old WCF claims object model. To make the decision easier for you, WIF actually empties *ServiceSecurityContext.Current.Claim*. When WIF is in the pipeline, that property is not available. The same goes for *OperationContext*.

Custom implementations of *ServiceAuthorizationManager*, a common means of modifying how WCF handles requests, would collide with the fact that WIF uses its own *IdentityModelServiceAuthorizationManager* for integrating with WCF. There are ways of working around this, but the safest strategy is to avoid using a custom *ServiceAuthorizationManager*. The WIF model offers plenty of extensibility options anyway.

One general principle is this: if it's in WIF and in WCF, you should usually go with WIF. Good examples of this are authorization (use *ClaimsAuthorizationManager*, and don't override *GetAuthorizationPolicies*) and server certificates (if you define it both in WIF and WCF, the WIF one will be used). Also, watch out for WIF requirements that are stricter than their WCF counterparts—for example, a service expecting X.509 or SAML tokens should have an entry for the issuer or issuers in <*issuerNameRegistry*>.

Client-Side Features

In the first half of the chapter, I argued that you don't need to deploy the WIF runtime assemblies on the client, because WCF offers all you need from .NET 3.5 SP2 on. After all, good service orientation requires that the service implementation be able to change from Java to a WIF-protected service and vice versa with the client being none the wiser, as long as the service contract and policies are preserved.

That is all true, and I absolutely stand behind it. However, WIF introduces a few new APIs that can make the life of developers easier when consuming services with WCF. The extra power and the new scenarios those new APIs unlock make it well worth redistributing WIF on the client, as well as wherever possible.

Delegation and Trusted Subsystems

Let's talk about trusted subsystems. An easy (though not entirely accurate) way of depicting a trusted subsystem is to think about what happens in certain theme parks or tourist facilities. You pay a fee at the entrance and are given in return something very visible, like a fluorescent wristband, that identifies you as a paying customer. From that moment on, you can do all the rides, eat from all the buffets, and drink as many cocktails as you want. The theme park is a *trusted subsystem*—once you are authenticated at its boundaries, your right to access resources inside is no longer in question.

That might work for theme parks, but for many applications that approach spells trouble. Consider the following scenario. An x-ray and MRI center serves all the hospitals of a certain region. The x-ray center offers a Web application that federates with all the hospitals so that every doctor can enjoy single sign-on when using the application for accessing the records of their patients. That's fine and dandy for establishing a session with the application UI—good old WS-Federation. The application code-behind logic retrieves the actual records by invoking a back-end service. How does the back end authenticate requests from the Web application?

Very often, the Web site will use its own application identity—common examples are the identity of the account with which its AppPool runs, or the X.509 certificate used for exposing the Web site on SSL. In any case, using the application identity makes this a trusted

subsystem. The application serves all possible doctors; hence, it needs the rights to retrieve the record of all possible patients. Sure, the service method might require the doctor's name among the input parameters, but that is no real protection against subversion. Imagine that a doctor wants to stalk the patient of a colleague and wants to know where she lives. If the Web application has some vulnerability—for example, some naïve use of SQL or query string parameters— the "bad" doctor might *within his own session* inject the name of his colleague in a call and obtain the information he is not supposed to access. Figure 5-6 shows the architecture of the application described here.

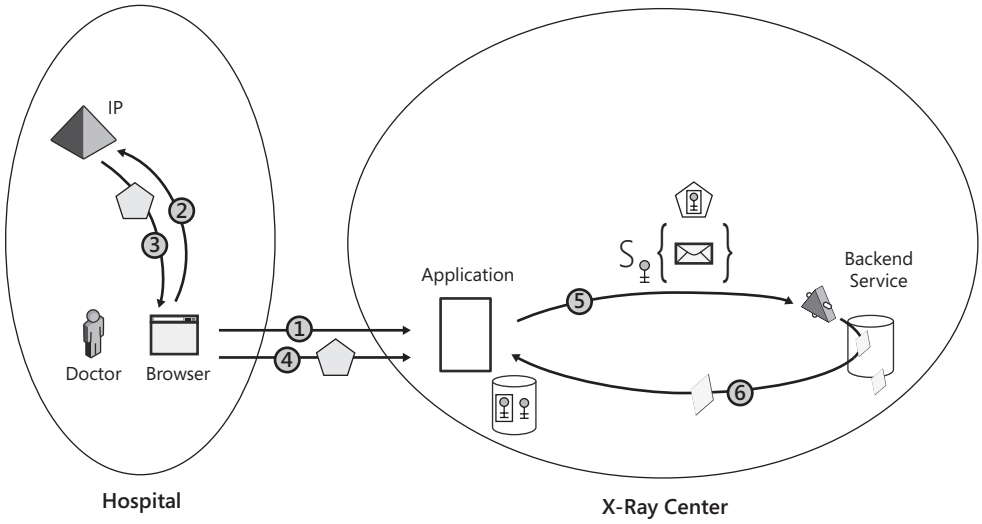


FIGURE 5-6 A trusted subsystem

Steps 1 through 4 show the usual WS-Federation sign-in dance. In step 5, the application invokes the back-end service, securing the call with its own X.509 certificate. The service verifies the credentials and, in step 6, sends back any patient record requested, without enforcing any restrictions.

The root of the issue here is obvious. The way in which the back-end service is accessed is independent from the actual user who started the session with the front end; hence, it cannot reflect the actual privileges the call should carry. The issue is well known in IT, and traditional solutions usually involve some infrastructural element. A classic example is Kerberos constrained delegation. Besides being expensive to set up and maintain, those solutions are not always practical. In the example just shown, the doctors are coming in from federated partners and they do not really have accounts at the x-ray center, making any Kerberos-based solution more complicated.

Version 1.4 of WS-Trust offers the tools for solving the conundrum. With WS-Trust 1.4, a requestor is able to specify in the RST to an STS that the requested token should contain information about a given identity, and that the requestor intends to use the issued token to

act as this identity. In practical terms, the Web application in the x-ray center example could reach out for one STS and ask a token declaring its intent to act as the current user. The STS would then be able to embed in the resulting token the actual permissions of the user, and the Web application could employ that token for calling the back-end service with just enough rights to access the doctor's own patients and no more than that. The solution is no longer a trusted subsystem: the rights of the actual user are flowing all the way to the back end. Neat!

Let's get a bit deeper into the mechanics of how this works. The application secures the RST with its own credentials, but it includes (in the special element *<wsse:ActAs>*) the token the user presented so that it can bootstrap the session with the Web application. The STS still authenticates the request according to the application's credentials, but it is now able to consider for its claims transformations the claims from the bootstrap token, too.

Bootstrap Tokens

The term *bootstrap token* is not a figure of speech—it's the technical term that refers to the token used by the client to authenticate with the RP and start the session. By default, the token is processed in the sign-in phase, its claims content and other data are harvested for the session, and the token itself is discarded. However, there are situations, such as the one described earlier, in which preserving the token itself comes in handy. WIF offers a configuration mechanism for saving the entire bootstrap token in the session, along with the claims it produced. The mechanism is the Boolean flag *saveBootstrapTokens*, which can be set either as an attribute of *<service>* (as shown below) or as an attribute of *<sessionTokenRequirements>* for the *SessionSecurityTokenHandler*:

```
<microsoft.identityModel>
    <service saveBootstrapTokens="true">
...
```

The saved bootstrap token can be accessed by the application code from *IClaimsIdentity*, by the appropriately named *BootstrapToken* property. *BootstrapToken* is of type *SecurityToken*, the base class of all the out-of-the-box token types in WCF.

Saving bootstrap tokens should not be taken lightly and should be done only when needed. In addition to the session-cookie-size issues it entails, which could be solved via the *IsSessionMode* trick introduced at the end of Chapter 3, there are serious security considerations to be addressed. For example, what if the bootstrap token is a *UserName* token, which includes the password of the user? That's clearly something you don't want to send around or save in places where it could be picked up by the wrong process. In this case, WIF deletes the password from the token in *BootstrapToken*, but if you dig into the WCF object model you can still retrieve the full token and do damage. The bottom line is this: use the feature with proper care.

Figure 5-7 schematizes how the x-ray center application handles access by taking advantage of the ActAs approach.

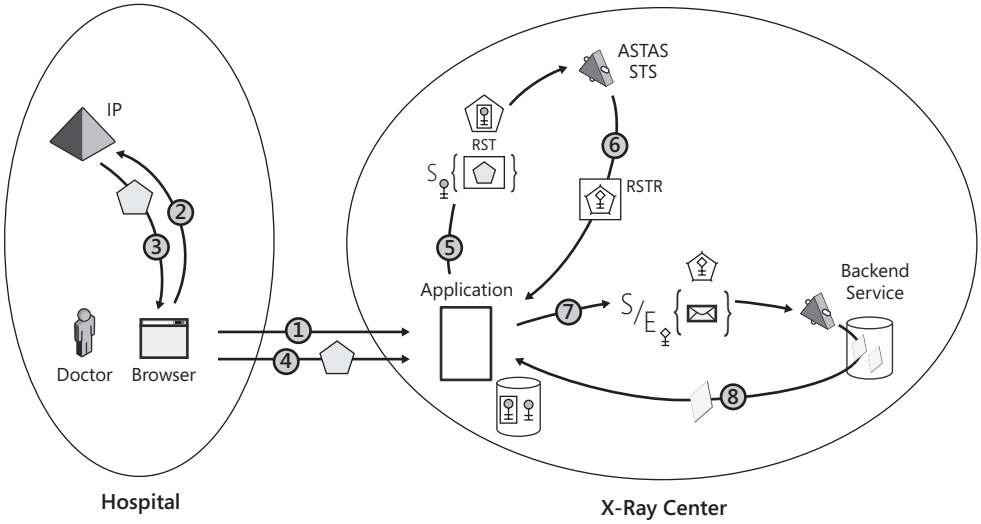


FIGURE 5-7 ActAs at work

As in the trusted subsystem version, the first four steps represent the usual WS-Federation sign-in: the doctor comes in with a token issued by the IP of the hospital he is employed with.

In step 5, the application sends an RST to an internal STS; the RST contains in the *<wsse:ActAs>* element the bootstrap token from the hospital IP, describing the current user, and it is secured by using the application's own certificate.

Note In this case, I introduced in the scenario a new STS, internal to the x-ray center. Here it makes sense as a policy evaluation point and as an authority trusted by the back end, given the nature of the data. In theory, nothing prevents you from having the original IP playing the ActAs STS role as well. It's just a matter of exposing the appropriate WS-Trust endpoint. In this case, it would be a bit awkward, as it means exposing the endpoint to the public Internet. (The IP for the WS-Federation parts does not have to.)

In step 6, the STS authenticates the call, analyzes the ActAs token, and looks up the patients whose records the user is allowed to examine. The STS packages those in the output token, wraps it in the RSTR message, and sends it back.

The application code-behind logic can now use the newly issued token as shown in step 7. The back-end service ensures that the requested records are among the ones the current user should have access to. If that is the case, it sends the data back in step 8.

As you can imagine, this approach has great potential for solving problems beyond the trusted subsystem pattern. ActAs unlocks all kinds of delegation scenarios, without suffering from the limitations of classic approaches (think double-hop problem with Windows authentication) and with great tools that can facilitate the process. (Think of how you can use the *Issuer* and *OriginalIssuer* claim properties for keeping track of the various delegation steps.)

OnBehalfOf

ActAs is often mentioned together with *OnBehalfOf*, but the latter is not usually given as much attention as the former. This section is no exception to the trend.

<wst:OnBehalfOf> is another element in the RST message introduced by WS-Trust 1.4, and like ActAs it is meant to contain a reference to an identity that is different from the token requestor. The key difference here is that whereas ActAs is meant for *delegation*, OnBehalfOf is meant for *impersonation*. The difference is not that subtle: with ActAs the STS issues a token containing details both about the requestor and the original user, whereas with OnBehalfOf the STS ignores the requestor and issues a token only about the original user. A service invoked via an ActAs token can reveal something about the delegation chain; a service called via an OnBehalfOf token has no clue that the caller is not actually the original user. In the following text, I'll mainly focus on ActAs. The syntax with which WIF implements OnBehalfOf is nearly the same as the one used for ActAs; therefore, you should be able to apply what you'll learn to both scenarios.

Note In the WIF documentation, one description of ActAs goes as follows: "we perform a delegated call to the backend service on behalf of the original caller." As with every loaded term, things can get confusing when it's not clear if the term is used in its technical meaning or for its colloquial English use. When I refer to the WS-Trust OnBehalfOf concept, I'll always use the term without blanks between the words.

WIF offers great support for ActAs and OnBehalfOf. In the next couple of subsections, I'll explore the relevant APIs in detail.

Note Although in general WIF supports the 1.3 and 2005 versions of WS-Trust, it also supports the ActAs and OnBehalfOf specific features despite of the fact that they are defined in WS-Trust 1.4.

CreateChannelActingAs and *CreateChannelOnBehalfOf*

How do you implement delegation as described here in practice?

On the service side, you already have all the necessities, because the scenario calls for a service that trusts an STS, regardless of how the STS itself issues the required token. I won't go into the details there.

The issue is on the client. You need to inject in the RST an *<wsse:ActAs>* element containing a token that refers to the identity you want to delegate from, but WCF does not offer an obvious way of doing that.

WIF provides an elegant solution to the problem: it defines an extension method for the *ChannelFactory* class, *CreateChannelActingAs*, which allows you to feed in the desired ActAs token as a parameter directly at channel creation time. *CreateChannelOnBehalfOf* does exactly the same for eliciting OnBehalfOf requests. Its use is straightforward. Take a look at the following code of one hypothetical Web page performing a service call protected by an ActAs token:

```
SecurityToken bootstrapToken =
 ((IClaimsPrincipal)Thread.CurrentPrincipal).Identities[0].BootstrapToken;

string tmpResult = "Call failed";

ChannelFactory<IXRayServiceChannel> factory = new
          ChannelFactory<IXRayServiceChannel>("CustomBinding_IXRayService");
factory.Credentials.ServiceCertificate.SetDefaultCertificate("CN=localhost",
                                  StoreLocation.LocalMachine, StoreName.My);
factory.ConfigureChannelFactory();

IXRayServiceChannel channel;
channel = factory.CreateChannelActingAs<IXRayServiceChannel>(bootstrapToken);

try
{
  tmpResult = string.Format(CultureInfo.InvariantCulture,"XRay Records: {0}",
                            channel.XRayRecords().ToString());
  channel.Close();
}
catch (SecurityAccessDeniedException)
{
  channel.Abort();
  tmpResult = "Access is denied";
}
```

The first line retrieves the bootstrap token from the current session, and of course, it assumes you opted for saving it.

The next three lines prepare the channel with the proper binding settings, referring to a custom setting *CustomBinding_IXRayService* in the configuration.

The next line finally creates the channel with *CreateChannelActingAs*, injecting in the bootstrap token as the ActAs token. The rest is standard WCF. The channel can now be used for invoking the service methods. The first call will trigger one RST to the STS containing the suitable ActAs element and will store the resulting issued token for subsequent uses.

The hands-on lab "Web Sites and Identity" (c:\IdentityTrainingKit2010\Labs\WebSitesAndIdentity) exercise 4 demonstrates how to implement the ActAs scenario described here. The STS point of view will be discussed next. If you are interested in the service, feel free to explore the lab solution. As mentioned, ActAs requires nothing special of the service and the binding used has nothing remarkable per se. If you want to experiment with a services-only scenario, you can try exercise 3 of the lab "Web Sites and Identity."

Using Delegation

The *Claims* collection of an *IClaimsIdentity* resulting from the deserialization of one token issued via ActAs is about the original user—that is, the identity specified in the ActAs element in the request to the STS. If the receiving service just worked with *Claims* it might never even realize that the call was performed by some entity other than the original caller. (With OnBehalfOf, that's exactly what happens.) However, *IClaimsIdentity* contains a property I didn't mention yet, a property called *Actor*. That property is an *IClaimsIdentity* itself, and it represents the identity of the actual caller. I am sure you see the recursive potential here, which can be leveraged for tracing a call all the way to its origins. If the call in the step before was itself a delegated call, the *Actor* property of the current *IClaimsPrincipal* will itself have a non-null *Actor* property, indicating the former delegation level, and so on. In that case, the first caller would be at the bottom of the *Actor* chain. Figure 5-8 shows a watch in Visual Studio of the *IClaimsIdentity* received by the service in the earlier example.

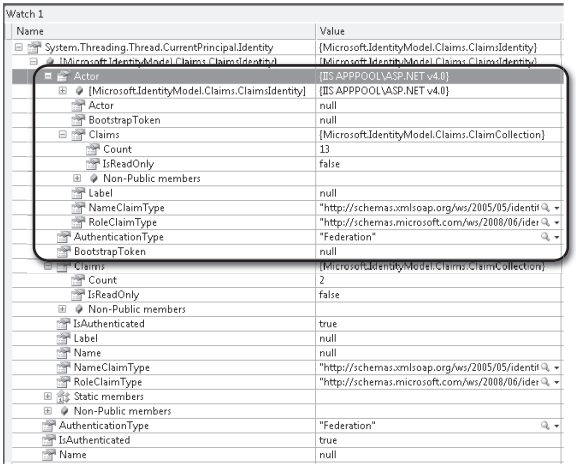


FIGURE 5-8 An *IClaimsIdentity* containing evidence that the current caller (the *Actor*) is invoking the service on behalf of another user, to which the current *Claims* collection refers (containing the usual two hard-coded claims produced by the STS template)

ActAs on the STS

ADFS 2.0 offers full support for *<wsse:ActAs>* and *<wst:OnBehalfOf>* elements in RST messages. As such, chances are that you'll never have to worry about what it takes for one STS to process delegation requests. However, you might always end up having to cobble up some test endpoint, and the ActAs support on the STS has great educational value in terms of consolidating your understanding of how WIF works. Hence, I'll spend a few lines on the topic.

Put yourself in the shoes of the STS developer. You are now required to process RSTs that contain not one, but two, entirely different tokens. One is used for security in the RST itself—business as usual. The other represents the original client's identity. To be processed, the ActAs token must go through much of the same pipeline of the token used for securing the call, except for the confirmation parts of course. That means that WIF must be configured with the proper *SecurityTokenHandler*, verify if the issuer of the ActAs token is known via its own entry in the *issuerNameRegistry*, and so on. It's not that hard, but it must be done. Let's take a look at the code of one *ServiceFactory* used for setting up an ActAs STS:

```
public class ActAsSecurityTokenServiceFactory : WSTrustServiceHostFactory
{
  public ActAsSecurityTokenServiceFactory()
  {
  }

  public override ServiceHostBase CreateServiceHost(string constructorString, Uri[]
baseAddresses)
  {
    SecurityTokenServiceConfiguration config = new SecurityTokenServiceConfiguration("STS");

    Uri baseUri = baseAddresses.FirstOrDefault(a => a.Scheme == "http");
    if (baseUri == null)
      throw new InvalidOperationException("The STS should be hosted under http");


    config.TrustEndpoints.Add(new
           ServiceHostEndpointConfiguration(typeof(IWSTrust13SyncContract),
           GetWindowsCredentialsBinding(), baseUri.AbsoluteUri));


    config.SecurityTokenService = typeof(CustomSecurityTokenService);


    SecurityTokenHandlerCollection actAsHandlers = new SecurityTokenHandlerCollection();
    Saml11SecurityTokenHandler actAsTokenHandler = new Saml11SecurityTokenHandler();
    actAsHandlers.Add(actAsTokenHandler);
    actAsHandlers.Configuration.AudienceRestriction.AudienceMode = AudienceUriMode.Never;

    actAsHandlers.Configuration.IssuerNameRegistry = new ActAsIssuerNameRegistry();

    config.SecurityTokenHandlerCollectionManager[
      SecurityTokenHandlerCollectionManager.Usage.ActAs] = actAsHandlers;
```

```
    WSTrustServiceHost host = new WSTrustServiceHost(config, baseAddresses);
    return host;
  }

  static Binding GetWindowsCredentialsBinding()
  {
    WS2007HttpBinding binding = new WS2007HttpBinding();
    binding.Security.Message.ClientCredentialType = MessageCredentialType.Windows;
    binding.Security.Message.EstablishSecurityContext = false;
    binding.Security.Mode = SecurityMode.Message;
    return binding;
  }
}
```

WSTrustHostFactory is provided by WIF, and it contains all the necessary logic to set up the token issuance pipeline. Its function is to create a *WSTrustServiceHost*, a specialized *ServiceHost* for hosting STS endpoints.

The first five lines of *CreateServiceHost* mostly do plumbing, associating binding, addresses, and the *SecurityTokenService* type to the current instance.

The following three lines create a new *SecurityTokenHandlerCollection*. In this case, the expected type for the ActAs token is SAML1.1. The right handler is added to the collection.

The *AudienceUri* is set to ignore the *AudienceRestriction* because it is by design that the token will have been issued for somebody else (namely, the front end). A dedicated *IssuerNameRegistry* is assigned to the collection, which can (and often is) different from the class used for the token that secures the RST.

Finally, the new handlers collection is assigned to the current *SecurityTokenHandler* collection with the indication that its intended usage is for ActAs tokens. The last two lines simply start the host.

Without the extra lines above, WIF would not know what to do with the ActAs token and would throw an exception. Note that those settings can also be added via configuration.

Once WIF authenticated the RST and correctly deserialized it, you need to access the claims from the ActAs token if you are to include claims in the output token on the basis of that. Consider the code of the *GetOutputClaimsIdentity* implementation that's shown next, taken from the WIF SDK:

```
protected override IClaimsIdentity GetOutputClaimsIdentity( IClaimsPrincipal principal,
                                          RequestSecurityToken request, Scope scope )
    {
      // Create new identity and copy content of the caller's identity into it
      // (including the existing delegate chain)
      IClaimsIdentity callerIdentity = (IClaimsIdentity)principal.Identity;
      IClaimsIdentity outputIdentity = callerIdentity.Copy();
```

```
    // There may be many GroupSid claims which we ignore to reduce the token size
    // Just select the PrimarySid and Name claims for the purpose of this sample
    Claim[] claims = (from c in outputIdentity.Claims
                      where c.ClaimType == ClaimTypes.PrimarySid ||
                            c.ClaimType == ClaimTypes.Name
                      select c ).ToArray<Claim>();

    outputIdentity.Claims.Clear();
    outputIdentity.Claims.AddRange( claims );

    // If there is an ActAs token in the RST, return a copy of it as the top-most identity
    // and put the caller's identity into the Actor property of this identity.
    if ( request.ActAs != null )
    {
      IClaimsIdentity actAsSubject = request.ActAs.GetSubject()[0];
      IClaimsIdentity actAsIdentity = actAsSubject.Copy();

      // Find the last actor in the actAs identity
      IClaimsIdentity lastActor = actAsIdentity;
      while ( lastActor.Actor != null )
      {
        lastActor = lastActor.Actor;
      }

      // Set the caller's identity as the last actor in the delegation chain
      lastActor.Actor = outputIdentity;

      // Return the actAsIdentity instead of the caller's identity in this case
      outputIdentity = actAsIdentity;
    }

    return outputIdentity;
  }
```

The code does what you would expect, given what you have learned earlier about *IClaimsIdentity* and *Actor*. It puts the ActAs identity as the topmost one, and it appends the current caller at the bottom of the *Actor*s chain.

Taking Control of Token Requests

The general attitude of WCF toward WS-Trust is to take care of the RST/RSTR deep in the binding logic. As a developer, you have the responsibility to set the correct bindings and addresses and feed in the client credentials if required. From that point on, WCF takes over, performing the RST/RSTR exchange and using the resulting token with the service in what looks like a single operation from the developer's point of view. *CreateChannelActingAs* does not really change this. It simply offers you a chance to add an extra parameter in the RST, but the token request process remains as opaque as in pure WCF. This approach keeps things simple for the developer, but it has a few drawbacks. The main one is related to performance.

Imagine you are performing ActAs service calls from the code-behind logic of a Web site. At the first call from the channel, the ActAs token is acquired and stored in the channel itself. As long as the channel instance remains in memory, the same ActAs token will be reused for all subsequent service calls—as long as it is valid, and if the binding calls for it. However, the channel might not stay in memory for very long. IIS recycles from time to time, and it is the nature of certain Web pages to be sparsely accessed. When a channel instance gets disposed of, the ActAs token disappears with it. The next channel instance will have to reach again for the STS, re-obtain the token, and restart the game from there.

The way around the issue is to take explicit control of the step in which the STS is invoked and the requested token is returned. After the token itself surfaces at the API level, it becomes possible to do interesting things (such as caching it). Naturally, there is a complementary aspect that must be considered: if the token is obtained explicitly, there must also be a way of explicitly feeding it into service calls.

Windows Identity Foundation offers tools for addressing both ends of the problem: a new channel dedicated to explicit client-to-STS communications, and a *ChannelFactory* extension method for explicitly providing the authentication token to be used in a service call. The next couple of sections will show you how to take advantage of those new capabilities.

WSTrustChannel and *WSTrustChannelFactory*

Every WS-Trust STS is basically an instance of the same service contract (or of a few service contracts, if you take into account variations due to specification versions). This is what makes possible to ship in WIF a stock class that covers the basics and only requires few modifications to create a fully functional issuer.

Earlier, I argued that it would be handy to be able to deal with the STS directly from the client. In API terms, that pretty much amounts to providing a stock proxy for a given contract that happens to implement an STS. Given that WIF extends the WCF programming model, the best way of providing that is by offering a special channel and associated channel factory: *WSTrustChannel* and *WSTrustChannelFactory*, respectively.

WSTrustChannel protects the developer from needing to work at the angle brackets level, wrapping all the intelligence necessary for crafting messages according to WS-Trust and deserializing the corresponding response messages into objects that can be easily incorporated in the WIF programming model. You get a few parameters in, and you get a complete *GenericXmlSecurityToken* in return. All the various WS-Trust messages are abstracted by the method calls *Issue*, *Validate*, *Renew*, and *Cancel*, both in their synchronous and asynchronous versions.

Note Although *WSTrustChannel* can emit all the WS-Trust messages listed; the *SecurityTokenService* out-of-the-box implementation will honor only *Issue*.

The most important method in *WSTrustChannel* is undoubtedly *Issue*. Although it is overloaded, I'll focus just on the following form:

```
public virtual SecurityToken Issue(RequestSecurityToken rst,
                                   out RequestSecurityTokenResponse rstr)
```

You provide a *RequestSecurityToken*, and you get back the issued token and the entire *RequestSecurtityTokenResponse* in case you need to take a peek at it as well. The key to the expressive power of this approach to issuance is in *RequestSecurityToken*; by giving you complete control over what goes in the RST, you can easily re-create any scenario, including the ones related to the delegation discussed earlier in the chapter. Let's take a look at one code example of using *WSTrustChannel* for obtaining an ActAs token:

```
WS2007HttpBinding binding = new WS2007HttpBinding();
binding.Security.Mode = SecurityMode.Message;
binding.Security.Message.ClientCredentialType = MessageCredentialType.Certificate;
SecurityTokenHandlerCollectionManager securityTokenManager = new
             SecurityTokenHandlerCollectionManager(string.Empty);
securityTokenManager["ActAs"] =
  SecurityTokenHandlerCollection.CreateDefaultSecurityTokenHandlerCollection();
securityTokenManager[string.Empty] =
  SecurityTokenHandlerCollection.CreateDefaultSecurityTokenHandlerCollection();

X509Certificate2 clientCertificate = CertificateUtility.GetCertificate(
  StoreName.My,
  StoreLocation.LocalMachine,
  WebConfigurationManager.AppSettings["FabrikamShippingCertificateSubjectName"]);
X509Certificate2 endpointIdentityCertificate = CertificateUtility.GetCertificate(
  StoreName.TrustedPeople,
  StoreLocation.LocalMachine,
  WebConfigurationManager.AppSettings["EndpointIdentityCertificateSubjectName"]);

var trustChannelFactory = new WSTrustChannelFactory(
  binding,
  new EndpointAddress(new Uri(WebConfigurationManager.AppSettings["ActAsStsUrl"])));

trustChannelFactory.TrustVersion = TrustVersion.WSTrust13;
trustChannelFactory.SecurityTokenHandlerCollectionManager = securityTokenManager;
trustChannelFactory.Credentials.ClientCertificate.Certificate = clientCertificate;
trustChannelFactory.Credentials.ServiceCertificate.Authentication.CertificateValidationMode
                  = X509CertificateValidationMode.PeerOrChainTrust;
trustChannelFactory.Credentials.ServiceCertificate.Authentication.RevocationMode
                  = X509RevocationMode.NoCheck;

try
{
  RequestSecurityToken rst = new
    RequestSecurityToken(WSTrust13Constants.RequestTypes.Issue);

  rst.ActAs = new SecurityTokenElement(callerToken);
  rst.AppliesTo = new EndpointAddress(
              new Uri(WebConfigurationManager.AppSettings["AppliesToUrl"]),
              new X509CertificateEndpointIdentity(endpointIdentityCertificate));
```

```
  WSTrustChannel channel = (WSTrustChannel)trustChannelFactory.CreateChannel();
  RequestSecurityTokenResponse rstr = null;
  SecurityToken delegatedToken = channel.Issue(rst, out rstr);
  return delegatedToken;
}
finally
{
  trustChannelFactory.Close();
}
```

This code is taken from the FabrikamShipping sample, available at http://code.msdn.microsoft.com/FabrikamShipping.

Well, clearly we are not in Kansas anymore. At this point of the book, I expect you can handle code fragments like the one shown here. In fact, it's not that hard—let's decompose it in its main functional blocks.

The first three lines create the binding for the call. After all, an STS is just like any other service in this respect. Here I chose to use message security and client certificates as the credential type.

The following three lines are meant to prepare the token handler's collection for dealing with the ActAs token, similar to what you have seen in the "ActAs on the STS" section.

The two lines after that retrieve the certificate for client authentication and for the RP I'll ask the token for.

The following line is finally about the new API. Here I create the blueprint for the channel, providing the binding and the URI of the intended STS.

The following five lines are about setting up the channel properly: picking the right protocol version (1.3 as opposed to 2005), assigning the token handler collections just defined, providing the client credentials in the form of certificates, and determining the certificate validation options.

The try-catch block is where the interesting part takes place. You can see how an RST is constructed from the ground up, requiring just three lines despite the complexity of the delegation scenario. The first one creates the general message structure, selecting the right version of the standard; the second provides the *AppliesTo* parameter, the intended recipient of the requested token; the third provides the ActAs token, which in this case was acquired in some earlier steps not shown in the code. The last line is the equivalent of passing the ActAs token in *CreateChannelActingAs*. Here you are simply bolting it explicitly in the right place in the RST.

After that, it is just a matter of creating a *WSTrustChannel* from the appropriately configured *WSTrustChannelFactory* and calling *Issue* to retrieve the desired token.

RequestSecurityToken and *RequestSecurityTokenResponse*

I hate to use a cliché here, but what I said earlier about using *WSTrustChannel* does not even begin to scratch the surface of what can be done when you control the details of the *RequestSecurityToken*. To be fair to WCF, it was already possible to influence the content of the RST by using the <*issuedTokenParameters*> element in <*customBinding/security*>, but *RequestSecurityToken* blows that out of the water in terms of usability and expressive power. You can use it for sending extra information to the STS via *AdditionalContext*, you can specify the exact list of claims you need per each request in its *Claims* property, you can control everything of the crypto that should be applied to the requested token, and so on. You can even put your hands on the proof token if you are curious after the log explanation at the beginning of the chapter. I encourage you to explore the *RequestSecurityToken* properties, but be warned: not every STS implementation will understand or honor all flags.

RequestSecurityTokenResponse will likely come in handy less often, because the main prize you want from the STS is the token, and it is returned directly as the result of *Issue*. In any case, there will be times when you'll want to inspect the RST for extra parameters you might want to get from the STS, or there might be things that are inside the token but are inaccessible to the client because of the encryption for the intended RP. One example of an extra parameter is the *DisplayToken*, a structure that makes visible to the client the values (or a subset of the values) in the claims returned in the token. An example of token details you might find useful is *AppliesTo*, which is something that can help you route a message or reuse a cached token with the correct intended destination, even when the *AudienceRestriction* in the token itself is sealed.

CreateChannelWithIssuedToken

Great! You have the issued token available to your code. Now what? WIF provides another *ChannelFactory* extension method, which allows you to specify a token already in your possession as the credential to be used for invoking a service. This holds for tokens just obtained via *WSTrustChannel*, tokens rehydrated from a cache, or self-generated tokens.

Note Invoking one STS is not the only way of obtaining a token. You can autogenerate tokens using the *xxxSecurityTokenHandler.CreateToken* method. That's usually not a good idea because you often end up with messy key distribution issues, but it is technically possible.

The method in question is *CreateChannelWithIssuedToken*. Its use could not be more intuitive:

```
IXRayServiceChannel channel;
channel = factory.CreateChannelWithIssuedToken<IXRayServiceChannel>(issuedToken);
tmpResult = string.Format(CultureInfo.InvariantCulture,"XRay Records: {0},"
                          channel.XRayRecords().ToString());
```

I intentionally adapted the x-ray example to emphasize the differences from the *CreateChannelActingAs* case. Here you supply the token that will be used for securing the call, whereas in the *CreateChannelActingAs* case that token was silently obtained via an RST with the bootstrap token. With *CreateChannelWithIssuedToken*, you created the RST with the bootstrap token and obtained the issued token yourself, via *WSTrustChannel.Issue*. As a result, you have much more freedom in the way you save and reuse tokens.

Summary

WIF and WCF form a powerful combination. WIF brings simplicity and integration to many things that were hard to do with WCF alone, and it does that while maintaining consistency with the programming model it offers for ASP.NET.

The chapter started with some theory on message-based security. After that, I explored the canonical scenario consisting of one RP, one IP, and one client—all implemented via the path of least resistance, which consists of the WCF Web site template, the WIF active STS template, and the WCF built-in test client, respectively. After verifying that there was nothing surprising about the use of WIF in the solution, I took the chance to point out some of the characteristics of *ws2007HttpFederationBinding* and how they relate to the claims-based identity roles. I spent a few words on how WIF's *SecurityTokenHandlers* improve on their old WCF counterparts, and I explored a concrete example of a custom token handler. That provided a great segue to a detailed analysis of the WCF-WIF integration and of the WCF development area, which should be avoided when using WIF.

The second part of the chapter explored the trusted subsystem pattern and how proper delegation can improve on it. The theory was followed by concrete indications of how to use WIF on the client, on the service side, and even at the STS to take advantage of delegation and impersonation according to the WS-Trust model. Finally, I described how WIF offers an API for explicitly requesting tokens from one STS and one API for feeding authentication tokens directly in a WCF channel.

If you are a WCF developer, you now know how to integrate WIF in your work. If you are a WCF expert, you now have a precise idea to what extent you should still use the WCF object model and where WIF takes over. If you are neither, reading the chapter was probably a difficult experience. At least you now have a deep understanding of what makes the active case different from the passive one, and why a proof token is such a great remedy for man-in-the-middle attacks.

Chapter 6

WIF and Windows Azure

In this chapter:

The cloud trend is perhaps one of the most important and transformational trends of the past few decades. It has the potential to change the way we do computing as an industry well after the hype cycle has run its course. In *The Big Switch: Rewiring the World*, (W.W. Norton, 2008), Nicholas Carr compares this trend to the switch that happened in the last century when businesses moved from producing electricity with their own private power plants to buying it from the grid. I believe it's a perfect metaphor for what's happening in the IT industry. The idea of using computing and storage power as a utility, offered as a pay-per-use service that relieves businesses and individuals from the capital expenses of having their own data center, is powerful because it appeals both to technical people (because it makes your life easier) and business people (because it reduces costs).

Windows Azure and its associated platform is the main offering from Microsoft in this space. In this chapter, I assume that you already know how to develop with Windows Azure. However, to make a long story short, it is a platform for running Windows applications and storing their data on Microsoft data centers (the Microsoft cloud) instead of on your own servers.

Claims-based identity is uniquely suitable for handling access management for cloud applications, to the point that it is common to hear people say that claims-based identity works in exactly the same way on-premises and in the cloud. Applications that outsource identity to external entities already have earned their autonomy from the infrastructure; hence, they can easily be hosted anywhere without loss of functionality or without restructuring being required. The open standards used in claims-based identity further lower the requirement bar. Finally, IP-STS and R-STS are powerful tools for modeling relationships and handling access, regardless of physical network boundaries.

If claims-based identity works in the cloud just like on-premises, why am I devoting an entire chapter to describing how Windows Identity Foundation (WIF) works on Windows Azure? The point is that, although the principles (and the code) remain almost exactly the same, a few infrastructural differences force you to pay attention to things you take for granted when you have full control of your environment (as is usually the case in on-premises scenarios). Furthermore, the types of applications that really shine on Windows Azure are the ones that can easily scale out to accommodate varying needs—that means dealing with massively distributed architectures, which warrant their own identity considerations whether they target Windows Azure or other environments. Think of this in the following terms: Imagine that you have to teach a friend how to play some card game. You would probably focus on the main rules of the game, such as how many cards you keep in your hand and the meaning of every figure, rather than details such as how to prevent the card deck from being scattered by the wind if you are playing on the beach. At this point, you already know how to play cards, so this chapter teaches you how to put a big stone on your deck so that you also can enjoy the game outdoors.

The section "The Basics" covers essential programming hygiene when developing WIF applications that target Windows Azure.

"Web Roles" presents a few things that deserve your attention when using WIF for securing Web applications hosted in Windows Azure: handling sessions in a load-balanced environment, dealing with Windows Azure certificate management tools, and coping with the multistaged development environment.

'WCF Roles" covers some challenges you need to be aware of when hosting your services in Windows Azure. Again, sessions are a key part of the discussion—this time adapted to the Windows Communication Foundation (WCF) object model,—in addition to service metadata generation and performing diagnostics in the cloud.

"WIF and ACS" mentions the relationship between WIF and the Windows Azure AppFabric Access Control Service, both in its version 1 and in light of projected plans for version 2.

"Custom STS in the Cloud" points out some considerations you should keep in mind before writing a Security Token Service (STS) meant to be hosted in Windows Azure.

After you read this chapter, you'll be able to adapt what you've learned about using WIF for securing applications to solutions that are hosted, entirely or in part, on Windows Azure.

The Basics

This section covers a few essential considerations about using WIF in Windows Azure. You should feel free to use WIF with a Windows Azure project exactly in the way you learned so far, including the use of tooling and wizards. A few points that warrant your attention when

setting things up are listed in this section. I'll get more into the specifics of each project type in the following sections. As mentioned, I assume you're already familiar with Windows Azure development, and I'll spend a few words introducing concepts only when they constitute a touch point with WIF or when I need to make sure we are on the same page about something.

Warning One key difference between services and boxed software is that rolling out new features and versions is much easier in the former than the latter. I fully expect Windows Azure to change, even significantly, over the shelf life of this book. For that reason, *I will not provide screen shots of current tools or portals here*, which would be more confusing than enlightening as soon as a new version alters the way they look. Instead, I'll try to focus on the fundamentals, which have a lower probability of changing (or will likely do so at a slower pace). If trees need to give their lives for this book to be printed, let's make sure that their sacrifice does not end in obsolescence too soon.

Packages and Config Files

Windows Azure comes with a set of tools and Microsoft Visual Studio templates that help you build applications just the way you are used to and test everything without leaving the boundaries of your own development machine on a local simulation environment (informally called *DevFabric*). Once you are satisfied with the results, the same Visual Studio tools can package up your application in a form that allows you to upload everything to Windows Azure and run your code in the cloud. The package is a file with the CSPKG extension and is, in fact, just a ZIP archive of all the files constituting your project. The package is accompanied by a CSCFG file, which describes the services you want to run and some important settings, such as how many instances should be spawned, which certificates should be used, and so on.

Here comes the first point you need to pay attention to. The CSPKG file contains *all* the files in your project. If you change anything in any file, you need to redeploy the package. That can be quite time-consuming and complicate matters when there's a version of the project already running. When I say "any file," that includes the config file of your project, which is no longer a viable option for tweaking the behavior of your application after deployment. Any changes to the config will require you to resend an entire package. This is a bit of a bummer for the wealth of config-based settings that WIF offers. One mitigation strategy can be storing multiple <*identityModel/service*> sections in the web.config file and using something external to the package itself (such as variables kept in Windows Azure storage, for example) to drive from code the selection of the appropriate config settings. I'll show an example of this strategy in action later in the chapter.

The WIF Runtime Assembly and Windows Azure

Once it is in the cloud, your code runs within *Roles*, Windows Azure's unit of execution, on a standardized environment based on .NET 3.5 SP2 or .NET 4.0 (at the time of this writing). As you have known since the first chapter of the book, the WIF runtime assemblies aren't part of the standard .NET Framework distribution package and are available as a standalone download. That means WIF will not be available in the cloud execution environment unless you include it yourself in your project's CSPKG. That is very easy to do: your Visual Studio project will have a reference to WIF, and you just need to go into the properties of the reference and set *Copy Local* to *true*. That will cause the assembly to be copied in the local folder, and that's enough for the Windows Azure tools for Visual Studio to pick it up and include it in the package.

One thing you need to make sure of is that you flag your project to be executed in Full Trust mode. In addition to the heavy usage of advanced crypto by the WIF assembly, executing in Full Trust mode is necessary for allowing an assembly outside of the Windows Azure global assembly cache (GAC) to run. The Visual Studio tools offer a UI for flipping the full trust flag, which is available via the properties of the role entry under the cloud project.

Important The fact that you upload the WIF assembly yourself means that the Windows Azure environment isn't really aware that the assembly is there. As a result, WIF will not partake of the advantages of the automatic update management feature of the Windows Azure execution environment. If there is an updated version of the WIF runtime, it is your responsibility to update your application accordingly.

Windows Azure and X.509 Certificates

Windows Azure executes your roles on virtual machine (VM) instances; however, it goes a long way to prevent you from taking a dependency on anything specific to the underlying machine. Registry, file system, and X.509 certificate stores are examples of things you should not try to access directly, because that leads to taking dependencies on the state of a specific instance. That prevents your application from enjoying the scale-out capabilities that make using a cloud platform so compelling.

Windows Azure has its own strategy for handling certificates. You can use the Windows Azure portal or the management API for uploading the bits of the certificates you need to use in your projects.

Note For detailed instructions on how to upload certificates via a portal or the management API, see the Windows Azure online documentation. As mentioned at the beginning of the section, those practices can change so fast that giving detailed instructions here would be pointless.

Windows Azure and the .PFX File Format

As of today, Windows Azure accepts certificates only in .PFX format. The PFX (Personal Information Exchange) format has been traditionally used for packaging and transferring certificates together with their corresponding private keys, thanks to its password-protected encryption. Standalone certificates are usually distributed in .CER or .DER file formats; however, Windows Azure won't understand those and will expect a PFX file in any case. This is one of those things I would not be surprised to see changing before the book goes into print. Just in case it doesn't, here's how you can produce a PFX file containing a certificate without its private key:

```
X509Certificate myCert = new X509Certificate(@"C:\myCert.cer",);
byte[] certData = myCert.Export(X509ContentType.Pkcs12);
System.IO.File.WriteAllBytes(@"C:\myCert.pfx", certData);
```

It's that easy. Isn't .NET a great platform?

After the certificate bits have been uploaded, you can specify for every role which ones should be available and in which certificate stores—the fabric itself will take care of deploying the certificates on the fly at the VM instantiation time. To be more precise, in the CSCFG file you associate a friendly name to the thumbprint of a certificate you already uploaded in Windows Azure, as shown here:

```
<Certificates>
  <Certificate name="myCert" thumbprint="991D46673C0A06C6A519614BE46742C78E872298"
               thumbprintAlgorithm="sha1" />
</Certificates>
```

Cloud projects also feature another "meta"-file, with a CSDEF extension, which contains other settings such as which endpoints the roles should expose. One of the settings they offer is defining the location of the certificates in the VM instance stores, as shown in the following snippet. Note that the certificate name must match across CSCFG and CSDEF.

```
<Certificate name="myCert" storeLocation="CurrentUser" storeName="<enter a value>" />
```

If you think for a moment about this, it makes perfect sense. Developers do not normally deploy and maintain certificates on data centers even according to classic on-premises development practices. Rather, they usually work with the ones put in there by the system administrators. (Because in smaller companies people often wear more than one hat, the developer and sysadmin could be the same person.)

Checking Token Signatures Without Storing the STS Certificate

It is at times possible to verify the signature of an incoming token without needing to explicitly maintain in the certificate store the X.509 of the trusted identity providers. This is a trick that works in on-premises scenarios as well, but it's especially popular for cloud scenarios. It goes as follows.

If you go back to Chapter 3, "WIF Processing Pipeline in ASP.NET," specifically to Figure 3-2 and the listing of the XML of an incoming SAML token, you can see that the element <ds:Signature/KeyInfo> contains the element <X509Data/X509Certificate>. That element is the base64 bits dump of the certificate used by the STS for signing the token. If you save that Base64 string in a file and give it a .CER extension, you can see the certificate properties just by double-clicking it in Windows Explorer.

If you get all the bits of the certificate together with the token, you don't need anything but the token itself for verifying the integrity of the signature. Verifying the integrity is just half of the equation, naturally—you also want to make sure that the token is coming from the expected STS. However, that part is already covered by the *issuerNameRegistry* element. As long as the thumbprint of the certificate embedded in the incoming *<ds:Signature>* corresponds to one of the thumbprints listed in the *<trustedIssuer>* elements, you are fine. The only thing you need to do to make this check work is turn off the *certificateValidationMode* for the service by setting it to *None*. If you do not, WIF will try to resolve the thumbprint to one certificate in the local stores and, failing to find one, would throw an exception. PKI fans will probably frown at this perspective, but here the check occurs more at the trusted IP level than the associated certificates.

The catch with this approach is that the STS is not forced to include the certificate bits in the token signature, and it can be perfectly compliant with the standards by not doing so. As a rule of thumb, Microsoft stacks usually do include the entire certificate; however, you should verify this on a case-by-case basis and, when in doubt, fall back to explicitly storing the STS signing certificate.

Web Roles

One of the main role types available on Windows Azure is the Web role, which is just Windows Azure speak for indicating an application that serves Web pages. For all practical purposes, in the development phase it refers to precisely the Web sites or Web applications you are used to working with—they just happen to be listed as candidates to become Web roles in a cloud project that lives in the same solution and have further descriptions in the CSCFG and CSDEF files described earlier. Another difference between roles and

their on-premises counterparts is the presence of the file *WebRole.cs*, which contains some handlers that trigger at specific moments of the role instance life cycle.

There are, however, a few things you might want to pay special attention to when using WIF and passive federation to handle access to Web applications meant to be hosted in Windows Azure.

Sessions

Web roles are designed to be executed in a network load balanced (NLB) environment. As a result, the default DPAPI-based mechanism provided by WIF out of the box for handling sessions is not suitable. I already covered in Chapter 4, "Advanced ASP.NET Programming" (in the "Sessions and Network Load Balancers" section) how to handle browser sessions for NLB scenarios; the same guidance applies here.

Controlling how the cookie is encrypted is not enough to manage sessions in Windows Azure if you are taking advantage of the *IsSessionMode* settings as described in Chapter 3. In that case, the cookie is just a reference to state that you are actually keeping on the server side—and, as you know, in Windows Azure "state" is actually a four-letter word. If you want to be able to handle sessions by reference in Windows Azure, you need to take explicit control of the way WIF handles the server-side token cache.

Out of the box, WIF uses *MruSecurityTokenCache*, an implementation of the abstract class *SecurityTokenCache*, which keeps session tokens in an in-memory database, inaccessible from any machine other than the one running it.

Note In the first prereleases, the *SecurityTokenCache* was mainly used for avoiding decrypting the incoming cookie at every request. *IsSessionMode* was introduced late in the cycle, in the release candidate that came out just weeks before RTM. If the cache gets flushed—for example, if the process gets recycled—it is not a big deal. You just decrypt the cookie again, and that's it. Of course, the music is different if you have *IsSessionMode* set to *true* and the token cache is the only place where you keep the session information. If the cache is deleted, at the next request WIF will not be able to resolve the reference and will not be able to retrieve the session content.

Nothing prevents you from writing your own *SecurityTokenCache* class and saving the tokens in durable storage. *SessionSecurityTokenCookieSerializer* is your friend here because it can produce a nice *byte[]* dump of the session token for you to store without complications. You can use various keying strategies for your cache. WIF uses a *SecurityTokenCacheKey* class, based on the cookie path, a context ID (associated with the reference in the cookie if *IsSessionMode* is on), a unique key generated at construction time, and a Boolean indicating whether the *IsSessionMode* flag is active. If those work for you, you can reuse the same keys.

Your custom cache needs to be associated with your *SessionSecurityTokenHandler*. If you are using the out-of-the-box handler, you can associate your custom cache type in its *<sessionTokenRequirement>* element—namely, via the *securityTokenCacheType* attribute. If you are writing your own handler, make sure that the *Owner* property of the custom cache instance gets assigned to the instance of your custom *SessionSecurityTokenHandler*.

It does not make a lot of sense to give you a full implementation of a Windows Azure–ready token cache here because it would be an exercise in Windows Azure table-storage API and caching strategies. However, I also don't want this to become a Fermat's margin: if by book publication time there isn't a public sample showing a *SecurityTokenCache* working on Windows Azure, I'll write the sample myself and reference it from the book's Web site.

If you are not familiar with Fermat's Last Theorem, you can get a primer on it at the following site: http://en.wikipedia.org/wiki/Fermat's Last_Theorem.

Endpoint Identity and Trust Management

In the default case, WIF assumes that the endpoint identity of a relying party is well known and stable, as reflected in various configuration elements. The *realm* and *AudienceUris* are typically captured in config at trust establishment time, and they're used by WIF to drive redirects and validations at runtime. Unfortunately, that doesn't work especially well under Windows Azure or any staged environment that influences the URIs on which services listen.

Windows Azure offers three distinct environments, all with different URI-generation rules:

- The already mentioned *DevFabric*, which is a simulation environment running on the local development machine where every application listens on *127.0.0.1:<portnumber>* regardless of the project name or settings.
- The *Staging Environment*, which is a cloud hosting area that listens on addresses of the form *<GUID>.cloudapp.net*. The GUID is assigned at deployment time, and you don't get to know it until the deployment takes place. That means you can never reflect it in the web.config, because that would require you to alter the config and redeploy, which in turn would result in a new GUID being assigned to you.
- The *Production Environment*, which is a cloud hosting area that listens on addresses of the form *<project name>.cloudapp.net*. In many cases, the actual application will use DNS mapping for presenting itself on an arbitrary vanity domain to the end user.

The same application will listen on a completely different address at every environment jump, sometimes without any chance to reflect it accordingly in the web.config. This behavior leads to at least two problems:

- **Failure to recognize the relying party (RP)** Because the identity of the RP is typically given by its network-addressable URI and associated certificate, changing any of the two would make the RP not recognized by the STS.

- **Redirecting to the wrong address** As briefly discussed in Chapter 4, the default STS template uses the incoming *AppliesTo* value as the address the *wresult* should be dispatched to. That clearly does not fly in this case.

There is no clear-cut solution to those problems that will satisfy all criteria. On one side, there is the need to keep things as secure as possible. That means maintaining strict controls at the STS and reprovisioning the same RP at every environment change, if it comes to that. On the other end of the spectrum, there is the need to seamlessly integrate those changes in the development process and keep things smooth. ADFS 2.0 does not compromise on security, and it sticks with the first approach—you have to reprovision the RP every time.

In fact, WS-Federation does offer separate mechanisms for handling audience verification (via *AppliesTo/wstream*) and redirection (via *wreply*). Because of that, you can assign your application a logical name (for example, a URN name rather than a network-addressable one) and use it for validation regardless of the physical address you are listening to. Also, you can dynamically assign *wreply* at every sign-in request for communicating to the STS the address at which you expect the *wresult* at that given moment. The big drawback of this approach (and the reason why ADFS 2.0 does not consent to it) is that it exposes you to redirect attacks, in which a malicious intermediary can just append to the query string a "*?wreply=http://myevilplace.com*" and hijack your freshly issued token even before you have a chance to use it. There are countermeasures you can put in place for limiting those risks—for example, you can explicitly validate the *wreply* value on the STS side before sending back the *wresult*.

Note A common misconception is that HTTPS will protect the body of HTTP requests but will do nothing to protect the URI itself. In fact, the query string *is* as encrypted as everything else—the only thing that remains visible to the occasional man in the middle is the hostname.

In case you want to try this approach, I've provided the following code snippet. Because it is customary for any case in which you want to communicate something to the STS, handling *RedirectingToIdentityProvider* in the *global.asax* is the way to go:

```
void WSFederationAuthenticationModule_RedirectingToIdentityProvider
     (object sender, RedirectingToIdentityProviderEventArgs e)
{
  HttpRequest request = HttpContext.Current.Request;
  Uri requestUrl = request.Url;
  StringBuilder wreply = new StringBuilder();

  wreply.Append(requestUrl.Scheme);   // e.g. "http" or "https"
  wreply.Append("://");
  wreply.Append(request.Headers["Host"] ?? requestUrl.Authority);
  wreply.Append(request.ApplicationPath);

  if (!request.ApplicationPath.EndsWith("/"))
    wreply.Append("/");
  e.SignInRequestMessage.Reply = wreply.ToString();
}
```

The gist of what the method does is in its last line, when a dynamic value of *wreply* gets assigned to the *Reply* property of the *SignInRequestMessage*. What is all the code above it for, then? Why fiddle with the *Host* header instead of simply assigning *HttpContext.Current.Request.Url* as *wreply*? There's an interesting story behind this.

In Windows Azure, every endpoint is always fronted by a load balancer, actual or virtual. If you use the traditional means of obtaining the address of the application, you usually get a URI that includes the internal port number that maps to the load-balancer node. That port number makes sense only behind the Windows Azure load balancer—it is not addressable from the outside. That means if you use it for the *wreply*, the redirect will fail. Using the host header, on the other hand, guarantees that the address works as expected (or that the application code will be running).

As mentioned, ADFS 2.0 does not work with *wreply*. If you want to use it from your custom STS, I recommend that you do so with caution and thoroughly validate it before accepting it. When you are confident it is a legitimate value, you can simply comment the line with the default *ReplyToAddress* line and modify as follows the redirect line in *GetScope* from your custom *ServiceTokenService* class:

```
//scope.ReplyToAddress = scope.AppliesToAddress;
scope.ReplyToAddress = String.IsNullOrEmpty(request.ReplyTo) ?
                       scope.AppliesToAddress : request.ReplyTo;
```

Multiple Config Settings for the Passive Case

WIF does provide a mechanism for storing in a single config multiple groups of settings in named <*service name*="*X*"> sections, which can be easily switched by picking the right one in some event handler according to some external variable. However, the config values assigned to *HttpModules* (such as the content of <*wsFederation*>, which is our main concern for the passive case scenario) are hard to control in that way. This is because properties associated with modules are loaded at module instantiation time, which is pretty hard to control. One possible strategy for regaining control of that is deriving your own class from WSFAM and overriding *InitializePropertiesFromConfiguration*, where you can embed your own logic that looks up the environment in which the role is running and picks the right config values accordingly.

That's it. Those remedies take more time to explain than to put in action. Again, there's nothing specific to Windows Azure here—those are all things you'd have to deal with in NLB or staged deployment environments.

The hands-on lab WindowsAzureAndPassiveFederation (c:\IdentityTrainingKit2010\Labs\WindowsAzureAndPassiveFederation) walks you through all the steps just described.

WCF Roles

Another important role flavor offered by Windows Azure is the WCF role. That is a classic svc-based WCF project that just happens to be listed as a Windows Azure role in a cloud project template, much like what you saw for the Web role. The WCF case shares various attention points with the Web role in terms of how to accommodate WIF.

All the code described in the upcoming sections is available in the hands-on lab "Web Services and Identity in the Cloud," (c:\IdentityTrainingKit2010\Labs\WebServicesAndIdentityInTheCloud).

Service Metadata

Creating a client for a service in WCF is easy, thanks to svcutil.exe. You point it to the service metadata, and moments later you have a channel that serializes parameters in the right format and applies policies (including security) as requested. On the service side, the metadata generation takes place automatically—you just need to enable exposing the associated endpoint for metadata retrieval. However, in Windows Azure the WCF metadata generation out of the box suffers from the same issues described in the preceding section when establishing the address of the application: the generated metadata ends up including load-balancer port numbers in the endpoint addresses, which leads to the generation of faulty proxies.

Note This has absolutely nothing to do with WIF. It is mentioned here because it just happens to be something blocking when working with WCF and Windows Azure.

There is a hotfix (available at *http://code.msdn.microsoft.com/KB981002*) that solves the issue; however, it is opt-in: installing the hotfix is not enough to change the default WCF behavior, as other clients might be leveraging the out-of-the-box behavior and any change would break them. You just need to add the following in your service behavior:

```
<behavior name="RelyingParty.WeatherServiceBehavior">
  ...
  <serviceCredentials>
    ...
  </serviceCredentials>
  <useRequestHeadersForMetadataAddress>
    <defaultPorts>
      <add scheme="http" port="8000" />
      <add scheme="https" port="8443" />
    </defaultPorts>
  </useRequestHeadersForMetadataAddress>
</behavior>
```

Sessions

Services in a load-balanced environment will experience the same session issues described for Web applications. With WCF alone, the only solution available today is turning on affinity and establishing sticky sessions. With WIF, you can pull out the same trick for services as the one described in Chapter 4 for Web applications, with the obvious differences for accommodating the different object model. As a reminder, the solution entails encrypting the session token using the service certificate (in a Web application, it's the SSL certificate) to sign and encrypt a cookie that the client includes in the call every time. The task of setting up WCF support for sessions in an NLB environment can be divided into three separate steps:

- Configuring the service binding to put it in cookie mode
- Creating a suitable *SessionSecurityTokenHandler*
- Making sure that the new token handler is in the service pipeline

Getting WCF in cookie mode is not especially intuitive, but it's pretty easy: it boils down to setting the *requireSecurityContextCancellation* property of the <*security*> element to *false* (don't ask). That can be achieved only by defining a custom binding. If you used the Federation Utility Wizard for establishing a trust relationship with one STS, you'll have to substitute the automatically generated *ws2007FederationHttpBinding* with something like the following:

```
<customBinding>
  <binding name="RelyingParty.IWeatherService">
    <security authenticationMode="SecureConversation"
        messageSecurityVersion=
       "WSSecurity11WSTrust13WSSecureConversation13WSSecurityPolicy12BasicSecurityProfile10"
        requireSecurityContextCancellation="false">
   <secureConversationBootstrap authenticationMode="IssuedTokenOverTransport"
       messageSecurityVersion=
      "WSSecurity11WSTrust13WSSecureConversation13WSSecurityPolicy12BasicSecurityProfile10">
      <issuedTokenParameters>
      <additionalRequestParameters>
        <AppliesTo xmlns="http://schemas.xmlsoap.org/ws/2004/09/policy">
        <EndpointReference xmlns="http://www.w3.org/2005/08/addressing">
          <Address>https://identitytk_cloud_rp.cloudapp.net/</Address>
        </EndpointReference>
        </AppliesTo>
      </additionalRequestParameters>
      <claimTypeRequirements>
        <add claimType=http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
              isOptional="true" />
        <add claimType=http://schemas.microsoft.com/ws/2008/06/identity/claims/role
             isOptional="true" />
      </claimTypeRequirements>
      <issuerMetadata address="https://localhost/LocalSTSEx01_End/Service.svc/mex" />
      </issuedTokenParameters>
    </secureConversationBootstrap>
    </security>
```

```
    <httpsTransport />
  </binding>
</customBinding>
```

The preceding code comes from the "Web Services and Identity in the Cloud" lab (c:\IdentityTrainingKit2010\Labs\WebServicesAndIdentityInTheCloud). The binding just shown uses mixed-mode security: the message travels through an HTTPS pipe, and the context token is used for signing some WS-Addressing header.

The second step entails creating a custom *SessionSecurityTokenHandler*. Although that's the exact same base class I demonstrated in Chapter 4 for NLB sessions, WCF requires some stricter checks here and there. Let's take a look at one example:

```
internal class RsaSessionSecurityTokenHandler : SessionSecurityTokenHandler
{
  public RsaSessionSecurityTokenHandler(X509Certificate2 certificate)
  {
    List<CookieTransform> transforms = new List<CookieTransform>();
    transforms.Add(new DeflateCookieTransform());
    transforms.Add(new RsaEncryptionCookieTransform(certificate));
    transforms.Add(new RsaSignatureCookieTransform(certificate));

    this.SetTransforms(transforms);
  }
```

Whereas in the passive case I simply changed the list of transforms on the fly, here I establish the desired operations in the constructor. The operations are the same as in the passive case, based on the service certificate.

Another thing that falls on the plate of the *SessionSecurityTokenHandler* is establishing if the session is valid. In a load-balanced environment, that entails addressing the possibility that a session has been created for another node, a fact that does not jeopardize the validity of the session in itself. Take a look at the following code:

```
public override ClaimsIdentityCollection ValidateToken
                (SessionSecurityToken token, string endpointId)
{
  if (token == null)
    throw new ArgumentNullException("token");
  if (String.IsNullOrEmpty(endpointId))
    throw new ArgumentException("endpointId");
  Uri listenerEndpointId;
  bool listenerHasUri =
       Uri.TryCreate(endpointId, UriKind.Absolute, out listenerEndpointId);
  Uri tokenEndpointId;
  bool tokenHasUri =
       Uri.TryCreate(token.EndpointId, UriKind.Absolute, out tokenEndpointId);
  if (listenerHasUri && tokenHasUri)
  {
    if (listenerEndpointId.Scheme != tokenEndpointId.Scheme ||
        listenerEndpointId.DnsSafeHost != tokenEndpointId.DnsSafeHost ||
```

```
        listenerEndpointId.AbsolutePath != tokenEndpointId.AbsolutePath)
    {
      throw new SecurityTokenValidationException(
        String.Format("The incoming token for '{0}' is not scoped to the endpoint '{1}'.",
         tokenEndpointId, listenerEndpointId));
    }
  }
  // In all other cases, fall back to string comparison
  else if
    (String.Equals(endpointId, token.EndpointId, StringComparison.Ordinal) == false)
  {
    throw new SecurityTokenValidationException(
        String.Format("The incoming token for '{0}' is not scoped to the endpoint '{1}'.",
        token.EndpointId, endpointId));
  }
  return this.ValidateToken(token);
}
```

The preceding code strips away the port numbers from the URIs comparison so that two URIs are considered equivalent modulo port numbers.

Note That takes care of the WIF validations, but you still need to handle at the WCF level the fact that the external caller is referring to one address while the service is listening on another (the same plus port number). To prevent WCF from throwing an exception, you need to decorate the service class with *[ServiceBehavior(AddressFilterMode = AddressFilterMode.Any)]*.

The last step is making sure that your custom *SessionSecurityTokenHandler* gets in the pipeline instead of the default one. In the passive case, you had the global.asax available for doing that. As you saw in Chapter 5, "WIF and WCF," in WCF things are more complicated. There are various approaches you could follow: the most straightforward one is probably creating a custom *IServiceBehavior*, which takes on the responsibility of adding WIF in the WCF pipeline and, in doing so, ensures that your custom *SessionSecurityTokenHandler* ends up in the right place. Here's a simple example:

```
class RsaSessionServiceBehavior : IServiceBehavior
    {
        public void AddBindingParameters(ServiceDescription serviceDescription,
                    ServiceHostBase serviceHostBase,
                    System.Collections.ObjectModel.Collection<ServiceEndpoint> endpoints,
                    BindingParameterCollection bindingParameters)
        {
        }

        public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
                    ServiceHostBase serviceHostBase)
        {
        }
```

```
        public void Validate(ServiceDescription serviceDescription,
                             ServiceHostBase serviceHostBase)
        {
            FederatedServiceCredentials.ConfigureServiceHost(serviceHostBase,
                        RoleEnvironment.GetConfigurationSettingValue("Deployment"));

            FederatedServiceCredentials credentials =
                  serviceHostBase.Description.Behaviors.Find<FederatedServiceCredentials>();
            X509Certificate2 certificate =
                  serviceHostBase.Credentials.ServiceCertificate.Certificate;
            credentials.SecurityTokenHandlers.AddOrReplace(
                  new RsaSessionSecurityTokenHandler(certificate));
        }
    }
```

The *Validate* method is where the interesting things take place. As you might recall, *FederatedServiceCredentials.ConfigureServiceHost* is what adds WIF in the WCF pipeline.

Note This is an example of the selective use of named *<microsoft.identityModel/Service>* sections. *RoleEnvironment.GetConfigurationSettingValue("Deployment")* can be used for retrieving a specific string from the Windows Azure storage that represents a name of a <service> section. For example, you can have a *<service name="development">* that refers to a development STS and a *<service name="production">* section containing all the production settings for the application. Simply changing the value of the *"Deployment"* variable from "development" to "production" would have the effect of selecting the appropriate group of settings at role startup time.

The last line ensures that your *RsaSessionSecurityTokenHandler* is the one that will deal with sessions.

Now that you have a behavior, you need a way to assign it to the service. You can already do that programmatically, but it's always nice to be able to do it via config. To that purpose, you can easily define a *BehaviorExtensionElement* for it:

```
public class RsaSessionServiceBehaviorExtension : BehaviorExtensionElement
{
    public override Type BehaviorType
    {
        get { return typeof(RsaSessionServiceBehavior); }
    }
    protected override object CreateBehavior()
    {
        return new RsaSessionServiceBehavior();
    }
}
```

You just need to assign the extension to the service behavior on the service's web.config and that's it.

Before leaving the topic, I think it's useful to look at the effects of the binding on the message itself. Here's a message dump as received from the RP, edited for clarity:

```
<s:Envelope [...]>
  <s:Header>
    <a:Action s:mustUnderstand="1">
     http://tempuri.org/IWeatherService/GetThreeDaysForecast
    </a:Action>
    <a:MessageID>[...]</a:MessageID>
    <a:ReplyTo> [...]</a:ReplyTo>
    <a:To s:mustUnderstand="1">https://127.0.0.1:8443/WeatherService.svc</a:To>
    <o:Security s:mustUnderstand="1"  [...]>
      <u:Timestamp u:Id="_0">
        <u:Created>2010-03-24T06:51:59.587Z</u:Created>
        <u:Expires>2010-03-24T06:56:59.587Z</u:Expires>
      </u:Timestamp>

      <SecurityContextToken u:Id="uuid-d625e518-fc1b-4c99-b618-d75620c90d7a-4"  [...]>
        <Identifier>urn:uuid:4e101b81-a537-46e8-b5d8-a670b8826396</Identifier>
        <Cookie xmlns="http://schemas.microsoft.com/ws/2006/05/security">
           gAAAA [...]t57cJSZ9YpXjpQaZ6jVjw==
        </Cookie>
      </SecurityContextToken>

      <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
          [...]
          <Reference URI="#_0">
              [...]
          </Reference>
        </SignedInfo>
        <SignatureValue>Y+kbleV/WtbePZnzakGEggFpAos=</SignatureValue>
        <KeyInfo>
          <o:SecurityTokenReference>
            <o:Reference URI="#uuid-d625e518-fc1b-4c99-b618-d75620c90d7a-4"></o:Reference>
          </o:SecurityTokenReference>
        </KeyInfo>
      </Signature>
    </o:Security>
  </s:Header>

  <s:Body>
    <GetThreeDaysForecast xmlns="http://tempuri.org/">
      <zipCode>98052</zipCode>
    </GetThreeDaysForecast>
  </s:Body>
</s:Envelope>
```

Observe the <*SecurityContextToken*> element: the bits are in the <*cookie*> element, protected via the *RsaxxxCookieTransform* operations established by the *RsaSessionSecurityTokenHandler*. The *u:Id* of the context token is the same as the

SecurityTokenReference element in the signature, showing that the context token is used for protecting the integrity of the <*Timestamp*> addressing element (as shown by the correspondence between the timestamp's *u:Id* and the <*Reference*> element in <*Signature/SignedInfo*>).

The WCF session is secured in the NLB environment without the need to rely on affinity and sticky sessions: check.

Tracing and Diagnostics

Note Once again, the issue discussed here is by no mean specific to WIF. At the same time, the solution proposed can be applied also in situations in which WIF plays no part, as in WCF-only services.

Have you ever enjoyed the pleasure of having your code compile and run as intended at the first attempt? Me too, but not too often, I have to admit. The reality is that you'll always need some means to troubleshoot, especially when you are working with many moving parts, as is WCF programming's prerogative. Just in case.

At the time of this writing, troubleshooting code execution in the cloud has its own challenges. For example, you cannot simply set a breakpoint in a Windows Azure role running in staging and expect that Visual Studio will break on it. Luckily, both WIF and WCF provide a comprehensive tracing solution, right? That's true, but tracing in the cloud is not as straightforward as one might think. You see, the traces themselves are in some sense state information. Windows Azure abhors statefulness because it hampers the scale-out capabilities of a solution. Saving a trace on the file system of a VM instance makes that instance somehow special, which is not good for an operating system that wants to abstract away differences. The solution for all this is very simple and obvious for seasoned Windows Azure developers: just save the trace in the Windows Azure storage, where it will be maintained regardless of what fate the originating VM instance will meet.

Note There is a little detail to consider—that is, using Windows Azure storage costs money. However, it is very cheap, and you would not keep the tracing on at all times, anyway, given the significant performance hit it entails.

The main asset you need for making this approach work is a trace listener that will dump traces in the Windows Azure storage. Once you have that and have performed some basic initialization at the beginning of the role life cycle, you can hook it up using the standard

config mechanisms. Here's the code for a possible implementation of such a trace listener as we coded it in the Identity Developer Training Kit:

```
public class AzureLocalStorageTraceListener : XmlWriterTraceListener
{
  public AzureLocalStorageTraceListener()
    : base(Path.Combine(AzureLocalStorageTraceListener.GetLogDirectory().Path,
          "role.svclog"))
    {
    }
  public static DirectoryConfiguration GetLogDirectory()
  {
    DirectoryConfiguration directory = new DirectoryConfiguration();
    directory.Container = "svclog";
    directory.DirectoryQuotaInMB = 100;
    directory.Path = RoleEnvironment.GetLocalResource("Logs").RootPath;
    return directory;
  }
}
```

Shockingly plain, right? This is just an *XmlWriterTraceListener* that receives a specific file path at construction time. The actual magic happens in WebRole.cs, the code file meant to contain all the role life-cycle logic. Windows Azure has its own diagnostic service, which I'll snap to for the WIF and WCF traces:

```
public class WebRole : RoleEntryPoint
{
  public override bool OnStart()
  {
    DiagnosticMonitorConfiguration diagnosticConfig =
        DiagnosticMonitor.GetDefaultInitialConfiguration();
      diagnosticConfig.DiagnosticInfrastructureLogs.ScheduledTransferPeriod =
        TimeSpan.FromMinutes(1);
      diagnosticConfig.Directories.DataSources.Add(
        AzureLocalStorageTraceListener.GetLogDirectory());
    diagnosticConfig.Directories.ScheduledTransferPeriod = TimeSpan.FromMinutes(1);
    DiagnosticMonitor.Start("DiagnosticsConnectionString", diagnosticConfig);
    RoleEnvironment.Changing += RoleEnvironmentChanging;
    return base.OnStart();
  }
```

I don't want to go into the details of the Windows Azure programming model because that's beyond the scope for this book. Here I'll just say that the code just shown starts the Windows Azure diagnostic service as the role itself starts, and that it initializes it with the path established by the custom trace listener. The code also establishes a one-minute refresh interval for transferring traces to the storage. The Windows Azure tools for Visual Studio offer mechanisms for creating local storage entries; alternatively, you can use the Windows Azure management APIs for the same purpose.

After all the aforementioned machinery is in place, you just go ahead with the config settings as usual, including the usual WCF-only settings in *<system.serviceModel>*:

```
<configuration>
  <system.diagnostics>
    <sharedListeners>
      <add name="AzureLocalStorage"
       type="RelyingParty.AzureLocalStorageTraceListener, RelyingParty" />
    </sharedListeners>
    <sources>
    <source name="Microsoft.IdentityModel" switchValue="Verbose">
      <listeners>
        <add name="AzureLocalStorage" />
      </listeners>
    </source>
    <source name="System.ServiceModel" switchValue="Verbose, ActivityTracing">
      <listeners>
        <add name="AzureLocalStorage" />
      </listeners>
    </source>
    <source name="System.ServiceModel.MessageLogging" switchValue="Verbose">
      <listeners>
        <add name="AzureLocalStorage" />
      </listeners>
    </source>
  </sources>
  <trace autoflush="true">
    <listeners>
    <add type="Microsoft.WindowsAzure.Diagnostics.DiagnosticMonitorTraceListener,
               Microsoft.WindowsAzure.Diagnostics, Version=1.0.0.0, Culture=neutral,
               PublicKeyToken=31bf3856ad364e35" name="AzureDiagnostics">
      <filter type="" />
    </add>
    </listeners>
  </trace>
  </system.diagnostics>
[...]
<system.serviceModel>
  <diagnostics>
    <messageLogging maxMessagesToLog="3000" logEntireMessage="true"
                    logMessagesAtServiceLevel="true" logMalformedMessages="true"
                    logMessagesAtTransportLevel="true" />
  </diagnostics>
  <services>
    ...
  </services>
<system.serviceModel>
```

You can retrieve the traces using the standard Windows Azure storage API. You'll find a series of .SVCLOG files, which can be opened by the *SvcTraceViewer.exe* utility just like any other. In fact, that's what I used for extracting the message dump I've added at the end of the preceding "Sessions" section.

WIF and ACS

The Windows Azure platform offers one service that is specifically focused on identity: the AppFabric Access Control Service (ACS). You can think of the ACS as one hosted R-STS: by subscribing to the service, you get your very own Federation Provider (FP) that you can use for managing your trust relationships without having to keep anything on-premises. Furthermore, you can specify the rules that will be used for transforming the claims from incoming tokens in the tokens issued by the ACS, allowing you to externalize even more of your authentication and authorization logic.

At the time of this writing, the version of ACS available in production is exclusively focused on REST Web services. It implements the OAuth WRAP protocol and exclusively issues Simple Web Tokens (SWTs). In the next chapter, I'll give more details about that protocol and token format. For the time being, it should suffice to say that WIF and ACS implement different standards and do not talk to each other out of the box (apart from one specific case, which I'll describe in the next chapter as well).

The next version of the service, which at publication time should be in preview, promises to play a different kind of music. Along with the REST endpoints, ACS will also offer more traditional protocol heads. I would love to be able to say more here, but it's never a good idea to write about prerelease software in a book about a released product. In any case, it should be of comfort to both me and my readers that as soon as ACS features WS-Federation and WS-Trust, integrating with WIF will be a routine matter, just like it is today with ADFS 2.0, and you'll able to do so just by applying what you've learned so far.

The idea of outsourcing the FP functions is very, very powerful. Apart from the convenience of not having to maintain your own endpoints, a service can offer things that are difficult to obtain with boxed software. For example, it is reasonable to expect that such a service will keep updating the identity providers it can work with, and with it the specific protocols they require. For a service, making available a new protocol head means simply flipping the publication switch of the associated endpoint; for an on-premises software installation, it can become a distribution nightmare. Furthermore, an R-STS supporting many identity providers and protocols can act as a protocol transition STS. That means the R-STS can accept identities transmitted using various protocols but normalize them for your application by standardizing on the protocol and token type of your choosing. That means, in theory, you could accept users coming from Live ID, Facebook, and Yahoo but still code your application to handle authentication via WS-Federation, regardless of whether that protocol is supported by those IPs. An R-STS in the sky, like the ACS, can really simplify things on many occasions.

Custom STS in the Cloud

By now, you know my position about using a custom STS: it should be done only for test purposes, or only when every other option has been considered and discarded. This is especially true in the cloud, where demand can be brutal. If you need an STS hosted in Windows Azure, the ACS should be the first thing that comes to mind. However, there can be borderline situations in which the out-of-the-box offering just does not cut it, and the only possibility of having things your way is by writing an STS and hosting it in Windows Azure. For example, at the time of this writing the ACS is meant to be an FP, but there are no plans for it to become an IP. If an IP in the cloud is what you need, you might have to write your own.

Writing an STS to be hosted in Windows Azure is not that different from what you'd do when targeting an on-premises deployment. The main things that require some workaround are the metadata generation and the RP allow list. Although those are both things you should worry about on-premises as well when writing your own STS, there are some Windows Azure–specific measures you should keep in mind when targeting the cloud environment. I'll provide a few words about those measures here.

Dynamic Metadata Generation

You saw how the changes in URI across DevFabric, staging, and production can affect the way in which WS-Federation sign-in takes place and RP provisioning at the STS is managed. You also saw how the ports of the Windows Azure balancer can interfere with the WCF metadata generation. Now, imagine the effect those things will have on the metadata document of one STS. Accuracy in the metadata description of the STS is of the utmost importance, both for making things work as expected and for trust reasons. (You really don't want to redirect your users to a fraudulent IP.) Editing the metadata document is out of the question—as you saw in Chapter 3, the STS metadata document needs to be signed. In addition, having to do that at every deployment would get old very, very fast.

The natural solution to the problem is to dynamically generate the federation metadata document so that you can include in it the necessary code for sensing changes in the environment and reflecting those in the endpoint URIs. A welcome bonus of this strategy is that you can finally change the set of claims you offer at will and still have them reflected in your metadata document.

How would you do that? Any dynamic Web content generation will do. For this example, I chose to expose a WCF REST service. Let's use the simplifying assumption that you are exposing a passive STS.

Every good service design starts with the contract:

```
[ServiceContract]
interface IFederationMetadata
{
    [OperationContract]
    [WebGet(UriTemplate = "2007-06/FederationMetadata.xml")]
    XElement FederationMetadata();
}
```

The idea is to hide the details of the implementation so that consumers of the metadata will have no clue that you are using a WCF service here (and won't be tempted to take dependencies on something that can change without notice).

The service in itself is simple, in principle:

```
[ServiceBehavior(AddressFilterMode = AddressFilterMode.Any)]
class FederationMetadataService : IFederationMetadata
{
    XElement IFederationMetadata.FederationMetadata()
    {
        return StsConfiguration.Current.GetFederationMetadata();
    }
}
```

Naturally, you need to decouple the address in the request from the actual service address validation, for the reasons seen so far. Here the implementation assumes that the *SecurityTokenServiceConfiguration* class contains both the class *FederationMetadataService* and a method that will do the actual metadata generation. This makes sense because lots of key information (such as the signing key) are kept in there. Let's take a look at a possible implementation of *GetFederationMetadata*:

```
XElement _federationMetadata;

public XElement GetFederationMetadata()
{
  if (_federationMetadata != null)
  {
    return _federationMetadata;
  }

// hostname
  string host = WebOperationContext.Current.IncomingRequest.Headers["Host"];
  EndpointAddress realm = new EndpointAddress(String.Format("https://{0}", host));
  EndpointAddress passiveEndpoint = new
      EndpointAddress(String.Format("https://{0}/Sts.aspx", host));
// metadata document
  EntityDescriptor entity = new EntityDescriptor(new EntityId(realm.Uri.AbsoluteUri));
  SecurityTokenServiceDescriptor sts = new SecurityTokenServiceDescriptor();
  entity.RoleDescriptors.Add(sts);
```

```
// signing key
  KeyDescriptor signingKey = new
      KeyDescriptor(this.SigningCredentials.SigningKeyIdentifier);
  signingKey.Use = KeyType.Signing;
  sts.Keys.Add(signingKey);
// claim types
  sts.ClaimTypesOffered.Add(new DisplayClaim(
        ClaimTypes.Email, "email address", "The subject's email address."));
// passive federation endpoint
  sts.PassiveRequestorEndpoints.Add(passiveEndpoint);
// supported protocols
  sts.ProtocolsSupported.Add(new Uri(WSFederationConstants.Namespace));
// add passive STS endpoint
  sts.SecurityTokenServiceEndpoints.Add(passiveEndpoint);
// metadata signing
  entity.SigningCredentials = this.SigningCredentials;
// serialize
  MetadataSerializer serializer = new MetadataSerializer();
  MemoryStream stream = new MemoryStream();
  serializer.WriteMetadata(stream, entity);
  stream.Flush();
  stream.Seek(0, SeekOrigin.Begin);
  XmlReader xmlReader = XmlTextReader.Create(stream);
  _federationMetadata = XElement.Load(xmlReader);
  return _federationMetadata;
  }
```

The code snippet is pretty long; hence, I added some comments. However, it works exactly as you'd expect. Let's go through it in detail.

The STS keeps the federation metadata document cached in one variable. The first time somebody requests it, the method runs in its entirety and saves the result in *_federationMetadata*. The next time there is a request for the document, the content of the variable is returned. (Nothing in the lifetime of the instance will invalidate the metadata document, at least in the general case.)

Note Services are inherently multi-threaded, hence caution is advised. As usual, this is just a sample. Please make sure that in your own code you enforce synchronization as required.

If *_federationMetadata* is null, you've got work to do. The code derives the actual address of the STS application in Windows Azure, using the HOST header as demonstrated earlier in the chapter. That will be the STS *realm*. Here I assume that the STS code is in the code-behind class of the page STS.aspx and I've built the endpoint address (*passiveEndpoint*) accordingly.

From now on, you've got to remember what was shown in Chapter 3, in the "Metadata Documents" section—namely, the annotated metadata document of the example STS. The code here follows exactly the structure described there. *EntityDescriptor*

and its *RoleDescriptors*, *KeyDescriptor*, and *SecurityTokenDescriptor* and its properties *Keys*, *ClaimTypesOffered*, *PassiveRequestorEndpoints*, *ProtocolsSupported*, *SecurityTokenServiceEndpoints*, and *SigningCredentials* are just some of WIF's elements used for describing the metadata documents section. There's not much to explain here—once you pick the right namespace (hint: it's *Microsoft.IdentityModel.Protocols.WSFederation.Metadata*), IntelliSense will bring you anywhere you want to go.

MetadataSerializer is the class that does the heavy lifting—namely, writing every element as appropriate and signing the document with the credentials provided in *EntityDescriptor.SigningCredentials* (which, in our example, are the same as the STS signing credentials, which is a common practice).

Now that the metadata generation logic is in place and wrapped in a service, it's time to expose the service itself in *FederationMetadataService.svc*:

```
<%@ ServiceHost Language="C#" Debug="true" Service="Sts.FederationMetadataService" %>
```

The service is supposed to be consumed via GET, just like a static metadata file would be. Here's the necessary binding:

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
    <behavior name="webHttp">
      <webHttp />
    </behavior>
    </endpointBehaviors>
    <serviceBehaviors>
    <behavior name="Sts.FederationMetadataBehavior">
      <serviceDebug includeExceptionDetailInFaults="true" />
    </behavior>
    </serviceBehaviors>
  </behaviors>
  <bindings>
    <webHttpBinding>
    <binding name="webHttpsBinding">
      <security mode="Transport" />
    </binding>
    </webHttpBinding>
  </bindings>
  <services>
    <service behaviorConfiguration="Sts. FederationMetadataBehavior"
             name="Sts.FederationMetadataService">
    <endpoint address="FederationMetadata"
              behaviorConfiguration="webHttp"
              binding="webHttpBinding"
              bindingConfiguration="webHttpsBinding"
              contract="Sts.IFederationMetadata" />
    </service>
  </services>
</system.serviceModel>
```

The icing on the cake? It's a URL rewrite rule that will feed requests for the federation metadata document to the service:

```
<system.webServer>
  <rewrite>
    <rules>
    <rule name="FederationMetadata" stopProcessing="true">
      <match url="^FederationMetadata/(.*)$"/>
      <action type="Rewrite" url="FederationMetadata.svc/FederationMetadata/{R:1}" />
    </rule>
    </rules>
  </rewrite>
</system.webServer>
```

At this point, you just need to make sure that your Web role is exposing a suitable HTTPS endpoint (you should always expose STS metadata on HTTPS), and you are done. From now on, the metadata document will always be correct regardless of where your STS is running. That assumes, of course, that you are taking care of handling certificates across environments.

Adding an Active STS Endpoint

To my knowledge, at the time of this writing there are no examples of STS solutions exposing both a passive endpoint and an active one. There is no technical reason that prevents this from happening. It's just that the metadata document gets a bit more complicated than usual, and because it is usually static it probably is not very conducive to doing that. Well, here we've got that part covered by adding dynamic metadata generation; hence, I'll build on that and show you how to add an active STS endpoint to the passive STS described so far. I am doing this in a sidebar because that's a bit of a stretch, and I want to make sure you go through this only if you are strongly motivated to do so.

Note Of course you can easily extrapolate from this the case in which you have only the active endpoint exposed in Windows Azure, or even for the on-premises case. After all you've been through in this book, I am sure it would not be too hard.

The main change in respect to the passive-only case is that you need to add a service endpoint to listen to WS-Trust RSTs. In fact, you already have an .SVC in the project: it's the metadata generation endpoint, *FederationMetadataService.svc*. It makes sense to consolidate and make one single SVC, which will expose both the metadata endpoint

and the STS itself. Let's throw away *FederationMetadataService* and create the following mix instead:

```
[ServiceBehavior(AddressFilterMode = AddressFilterMode.Any,
                 InstanceContextMode = InstanceContextMode.Single,
                 ConcurrencyMode = ConcurrencyMode.Multiple)]
    class AzureWSTrustServiceContract : WSTrustServiceContract, IFederationMetadata
    {
        public AzureWSTrustServiceContract()
            : base(StsConfiguration.Current)
        {
            // no op
        }

        XElement IFederationMetadata.FederationMetadata()
        {
            return StsConfiguration.Current.GetFederationMetadata();
        }
    }
}
```

AzureWSTrustServiceContract merges the usual STS responsibilities with our WS-Federation metadata functionality.

The next step is creating a factory that will instantiate it:

```
class AzureStsHostFactory : ServiceHostFactory
{
  public override ServiceHostBase CreateServiceHost(string constructorString,
                                                     Uri[] baseAddresses)
  {
    WSTrustServiceHost serviceHost = new WSTrustServiceHost(
            new AzureWSTrustServiceContract(), baseAddresses);
    return serviceHost;
  }
}
```

The factory ends up in STS.svc, which substitutes FederationMetadataService.svc.

```
<%@ ServiceHost Language="C#" Debug="true" Factory="Sts.AzureStsHostFactory" %>
```

Now comes the fun part, the configuration. Here's an example of what the most salient parts might look like:

```
<microsoft.identityModel>
  <service>
    <securityTokenHandlers>
     <remove type="Microsoft.IdentityModel.Tokens.WindowsUserNameSecurityTokenHandler"
/>
     <add type="Microsoft.IdentityModel.Tokens.MembershipUserNameSecurityTokenHandler"
/>
    </securityTokenHandlers>
```

```
    <serviceCertificate>
      <certificateReference findValue="CN=myCert.cloudapp.net"
                            storeLocation="LocalMachine" storeName="My" />
    </serviceCertificate>
  </service>
</microsoft.identityModel>
```

The active STS is using the same credentials as the passive one. Here I assume they are the user name and password from some membership provider.

The WCF configuration is the most interesting:

```
<behavior name="Sts.StsBehavior">
  <serviceDebug includeExceptionDetailInFaults="true" />
  <serviceMetadata httpsGetEnabled="true" />
  <useRequestHeadersForMetadataAddress />
</behavior>
```

The new behavior does two important things: it enables GET on HTTPS, as required by the metadata, and it opts in to the metadata-generation mechanism based on the HOST header, as seen earlier in the "WCF" section of this chapter.

Important The metadata in this case is the service metadata, WSDL and WS-PolicyAttachment, not the WS-Federation metadata!

```
<ws2007HttpBinding>
  <binding name="userNameOverTransport">
    <security mode="TransportWithMessageCredential">
      <message clientCredentialType="UserName" establishSecurityContext="false" />
    </security>
  </binding>
</ws2007HttpBinding>
```

The STS binding is just a plain user name over transport:

```
<service behaviorConfiguration="Sts.StsBehavior"
         name="Sts.AzureWSTrustServiceContract">
  <endpoint address="FederationMetadata"
            behaviorConfiguration="webHttp"
            binding="webHttpBinding"
            bindingConfiguration="webHttpsBinding"
            contract="Sts.IFederationMetadata" />
  <endpoint address="mex"
            binding="mexHttpsBinding"
            contract="IMetadataExchange" />
  <endpoint address="UserNameOverTransport"
            binding="ws2007HttpBinding"
            bindingConfiguration="userNameOverTransport"
            contract=
              "Microsoft.IdentityModel.Protocols.WSTrust.IWSTrust13SyncContract" />
</service>
```

The service element nicely summarizes what's offered here: the first endpoint is the WS-Federation metadata endpoint, as defined for the passive case; the metadata endpoint offers the Web services metadata of the STS; and the last one is the issuance endpoint itself.

```
<system.webServer>
    <rewrite>
      <rules>
        <rule name="FederationMetadata" stopProcessing="true">
          <match url="^FederationMetadata/(.*)$"/>
          <action type="Rewrite" url="Sts.svc/FederationMetadata/{R:1}" />
        </rule>
      </rules>
    </rewrite>
</system.webServer>
```

The URL rewrite rule must be adjusted according to the new endpoint, of course.

The last thing you need to do is actually update the metadata document with the information about the active endpoint. Here's the code that you can add just below the passive endpoint definition in *GetFederationMetadata*:

```
// add active STS endpoints
foreach (ServiceEndpoint endpoint in OperationContext.Current.Host.Description.
Endpoints)
{
  if (endpoint.Contract.ContractType.IsAssignableFrom(typeof(IWSTrust13AsyncContract))
||
      endpoint.Contract.ContractType.IsAssignableFrom(typeof(IWSTrust13SyncContract))
||
 endpoint.Contract.ContractType.IsAssignableFrom(typeof(IWSTrustFeb2005AsyncContract))
||
    endpoint.Contract.ContractType.IsAssignableFrom(typeof(IWSTrustFeb2005SyncContra
ct)))
  {
    UriBuilder endpointUri = new UriBuilder(endpoint.Address.Uri);
    endpointUri.Host = realm.Uri.Host;
    endpointUri.Port = realm.Uri.Port;
    EndpointAddressBuilder endpointAddress = new
       EndpointAddressBuilder(endpoint.Address);
    endpointAddress.Uri = endpointUri.Uri;
    sts.SecurityTokenServiceEndpoints.Add(endpointAddress.ToEndpointAddress());
  }
}
```

The preceding code contains a really neat trick: it inspects the service endpoints in the current host and, whenever it finds one that implements WS-Trust using the well-known corresponding WCF contract types, it adds the corresponding endpoint as a *SecurityServiceEndpoint*. This is a much cleaner solution than the hard-coded "STS.aspx" path fragment used in the passive solution. On the other hand, in the passive solution we didn't have the luxury of having well-defined contracts.

OK, this time I won't say that this was easy. But it is certainly not rocket science. In any case, please remember this: writing your own STS can be tough. Whenever you have the chance to use a packaged product or a service, I strongly encourage you to do so rather than writing your own.

RP Management

When you take on the responsibility of writing one STS, maintaining the allow list of RPs falls onto your plate as well. This is true in the cloud as much as it is on-premises. The STS template does not help you much for this—it just reminds you of the need to validate the intended recipient by showing you a hard-coded check in *GetScope*.

Why do I touch on that problem here? Well, maintaining an allow list means maintaining state. As such, by now you know that in Windows Azure that warrants special attention. In fact, as long as you don't keep data exclusively on one instance, you can do pretty much anything you like. For example, keeping the list of RPs (with addresses, certificates, claims required, and so on) in Windows Azure table storage and looking that up in your implementation of *GetScope* seems like a very good idea.

Summary

This chapter examined various ways you can take advantage of WIF for securing the Windows Azure role. In the process of reading it, you explored many advanced ways of using WIF that are not necessarily exclusive to the cloud: NLB sessions both in the passive and active cases, conditional configuration, multistaged environments, tracing and diagnostics, dynamic WS-Federation metadata generation, WCF cookie mode, and so on. Other aspects, such as certificate management and HOST header use for reconstructing the application's address, are inherent in the use of Windows Azure.

Using Windows Identity Foundation with Windows Azure is straightforward, and it's a powerful enabler for reusing existing on-premises identities in the cloud or for opening up your solutions to federated partners. Being able to take advantage of WIF in Windows Azure might very well yield the best return on investment for having read this book.

Chapter 7

The Road Ahead

In this chapter:

This final chapter of the book touches on some important scenarios and technologies you should probably know about, but that are not addressed out of the box by the first version of Windows Identity Foundation (WIF). In some cases, making WIF work with a given technology will require just few lines of code. In these instances, I'll walk you through it. At other times, integration would require so much code that it would push describing a complete solution beyond the scope of this book; in those cases, I'll just frame the problem space and refer you to external resources when available.

I want to be absolutely clear on one thing: the fact that I mention a technology or scenario here does not constitute a promise that it will be tackled by the next version of Windows Identity Foundation. The fact that I judged a scenario important enough to deserve being mentioned here means that I heard many requests for it, which makes it a likely candidate for a new product feature, but nothing is really certain. I sure hope that as many of the scenarios mentioned here as possible will eventually migrate to the earlier chapters in future editions of this book! Another thing to consider is that, thanks to the extremely flexible extensibility model of WIF, any Identity developer can come out with sample code that addresses the scenarios listed here. Given how fast things change on the Internet nowadays, I would not be surprised if something unaddressed at the time of this writing will be covered by the same code sample or even by some product after this book is published. I'll make sure to keep you up to date on my blog at *http://www.cloudidentity.net*.

In the last section, I'll spend a few closing words and help you decide where to go from here.

New Scenarios and Technologies

While reading the book, you had many occasions to observe the flexibility of WIF in addressing many different scenarios, both in the Web and services worlds—and I didn't really show everything that can be done. For example, I spent very little time on synergies with ADFS 2.0, and no time at all on how WIF powers SharePoint 2010 Identity features and its effects on SharePoint use and extensibility.

However, there are also important technologies and scenarios that version 1 of WIF does not address. Some of those scenarios are so important that you will not be able to ignore them: in this case, knowing some integration tactics can save the day. In the following sections, I'll discuss how to integrate WIF with the ASP.NET MVC framework, with Silverlight applications, with the SAML-P, and with REST relying parties and issuers.

Note that nowadays most of those scenarios require you to *extend* WIF, rather than just *use* it. This is a fundamental difference from the rest of the book, where I've shown you how to use WIF to solve challenges in your solutions. Writing support for a popular protocol not natively implemented by the product is basically writing code that lives at the platform level; that is, sheer plumbing. A good criterion for deciding if something belongs to the platform is establishing whether the same code has to be written over and over again, by partners and competitors alike. If it does, that's not very strategic code: it does not give you an advantage beyond first-mover, and the investment is lost once a new version of the platform (or a third-party add-on) appears on the market.

To be clear, I'm not saying you should not extend WIF. By all means, WIF has been made to be extensible. What I am saying is that not everybody will be in a position to write their own SAML-P implementation, but many more would *use* such an implementation once it became available, and that is why in this chapter I am more interested in helping you understand how a given scenario works rather than giving you a toy implementation you'd toss away as soon as a serious alternative became available.

ASP.NET MVC

The ASP.NET MVC framework has been a phenomenal success, in no small part thanks to its test-friendly approach to Web development, its nimble bandwidth requirement, and the expressive power of the model-view-controller (MVC) pattern. However, its peculiar way of driving requests makes it harder to integrate it with WIF. Mind you, it is possible to protect an MVC application with WIF—it's not even especially hard. It just doesn't happen out of the box; instead, it requires a few lines of custom code and some care in the configuration on your part. Giving you a complete introduction to ASP.NET MVC is beyond the scope of this book. Here I am assuming you already know what ASP.NET MVC is about, and I'll just point out the few things that are relevant to our discussion about WIF.

Most of the MVC flow is in the hands of one *HttpModule* (*UrlRoutingModule*) and one *HttpHandler* (*MvcHttpHandler*), which leverage well-known conventions to dispatch execution to the appropriate controller classes and action methods according to a fixed set of URL routes. The bottom line of that complicated sentence is that MVC works on a very different basis than the hit-and-redirect regimen you got used to in the chapters about WIF and ASP.NET.

The brute-force solution for using WIF and MVC together in the same solution is making sure they never meet. In theory, you could just have WSFAM do its redirects before the MVC engine has a chance to kick in. Once there's a session in place, the FAM can just move aside and hand over control of the URL structure and execution to ASP.NET MVC. That is, of course, a non-solution because there's exactly zero integration with the MVC way of expressing authentication requirements.

Another solution that is popular at the time of this writing suggests implementing a custom *FilterAttribute*, which encapsulates the logic for triggering the call to the STS and funnels the resulting *wssignin* message to a fixed *Action*. That *Action* processes the sign-in message and, finally, redirects to the URL that originated the authentication. Although it's better than the brute-force method, this approach still falls short of integrating with the MVC way of handling things.

Let's take a look at a better solution. ASP.NET MVC offers a nice attribute, *AuthorizeAttribute*, that can be used to restrict access to action methods or even to entire controllers. If one unauthorized user attempts access to an action method decorated with *[Authorize]*, ASP.NET MVC will promptly return a 401 error message. If there is something configured to catch it, such as Forms authentication, this will kick-start the user authentication process. Another nice feature of *AuthorizeFilter* is that it allows you to further restrict access to the action method by requiring specific roles or even users, as shown in the following ASP.NET MVC documentation excerpt:

```
[Authorize(Roles = "Admin, Super User")]
public ActionResult AdministratorsOnly()
{
  return View();
}

[Authorize(Users = "Betty, Johnny")]
public ActionResult SpecificUserOnly()
{
  return View();
}
```

That's the way MVC developers are used to handling authentication; therefore, that's the programming model you need to plug into. The cleanest way of doing it is by precisely following the MVC approach: if you want to control a flow, you need a controller.

The ASP.NET MVC project template features an *AccountController* class, whose action methods are cleverly activated by the login URL stored in the Forms authentication settings in config, as shown here:

```
<authentication mode="Forms">
    <forms loginUrl="~/Account/LogOn" timeout="2880" />
  </authentication>
```

The default *AccountController* wraps functions from the ASP.NET membership provider. All you need to do is change it to drive the WS-Federation login and logout process instead. It's not too hard. The trickiest part is ensuring that you bounce the requests to and from the STS in a way that they are entirely handled by the account controller.

The first thing you need to do, as usual, is add WIF in the *web.config*. There are just two differences from the usual case. First, you want to turn off automatic redirections and let MVC lead the dance. You do that with the *passiveRedirectEnabled="false"* flag in *<federatedAuthentication/wsFederation>*. Second, you want to make sure you add the *HttpModules* as follows:

```
 <httpModules>
    <add name="WSFederationAuthenticationModule"
         type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule,
               Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
               PublicKeyToken=31bf3856ad364e35"/>
    <add name="SessionAuthenticationModule"
         type="Microsoft.IdentityModel.Web.SessionAuthenticationModule,
               Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
               PublicKeyToken=31bf3856ad364e35"/>
    <add name="ScriptModule"
         type="System.Web.Handlers.ScriptModule,
               System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
               PublicKeyToken=31BF3856AD364E35"/>
    <add name="UrlRoutingModule"
         type="System.Web.Routing.UrlRoutingModule,
               System.Web.Routing, Version=3.5.0.0, Culture=neutral,
               PublicKeyToken=31BF3856AD364E35"/>
 </httpModules>
</system.web>
```

That pretty much sets the stage. The next step is dissecting what needs to happen in the account controller to obtain the desired effect.

Let's start from the case in which somebody clicks a sign-in button—that is, you have an unauthenticated user attempting a GET of *Account/LogOn*. Figure 7-1 shows the high-level flow I want to establish between controllers and the STS for addressing this case.

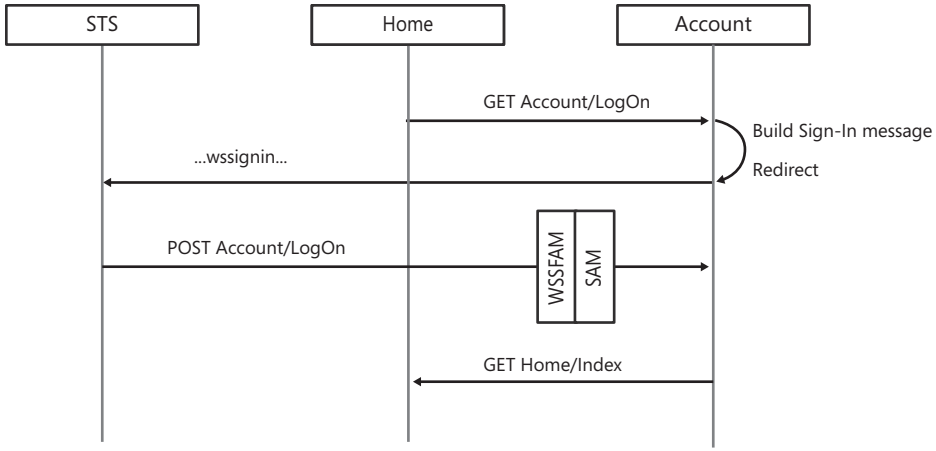


FIGURE 7-1 Signing in to an MVC application by clicking the login button

The unauthenticated GET triggers the Forms authentication redirect, which in turn calls the *LogOn* action of the account controller. In this case, there's no URL to return to after the authentication takes place: keep that in mind, it will come in useful later.

The *LogOn* action method should craft the *wssignin1.0* message according to the settings in config and ensure that the post will come back to the same controller.

The sign-in message travels to the STS; assuming the user successfully authenticated, the controller is once again invoked, this time via POST verb. The WSFAM and SAM in the pipeline take care of processing the *wresult* as usual so that the *LogOn* action method body finds the user authenticated. That was the intended result, so *LogOn* will just redirect back to the home page.

Now that's pretty neat! The other case that must be handled is the one in which one unauthenticated user invokes an action method protected via *[Authorize],* let's say when attempting a GET of *Secret/Index*. The flow is similar to the one just mentioned, with the extra complication that after the authentication phase the flow must go back to the requested view.

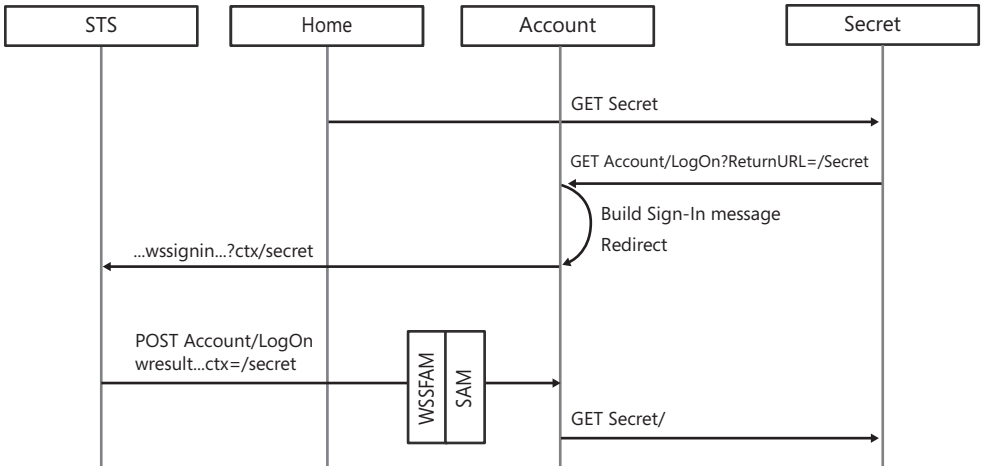


FIGURE 7-2 The authentication flow as triggered by *[Authorize]*

The flow is almost the same as earlier. The key difference is that the *LogOn* action is now invoked with a non-empty *ReturnUrl* parameter. *LogOn* takes advantage of w*ctx*, the WS-Federation parameter used for maintaining some context across multiple messages, for keeping track of that *ReturnUrl*. After the authentication successfully takes place, the URL can be retrieved and used to redirect back.

What else is missing? The logout. In fact, the logout is so simple that it does not even need a diagram. There is no automated way of doing logout if you exclude the cleanup messages (which are automatically handled by the WSFAM anyway). You can just define an explicit action method, say *LogOut*, and use it for signing out from Forms authentication, invoking *WSFAM.SignOut* and propagating the sign-out message to the STS.

Time to see some code! Here are the action methods:

```
[HandleError]
public class AccountController : Controller
{

  public ActionResult LogOn( string returnUrl )
  {
    return LogOnCommon( returnUrl );
  }

  [ValidateInput( false )]
  [AcceptVerbs( HttpVerbs.Post )]
  public ActionResult LogOn()
  {
    return LogOnCommon( null );
  }
```

```
    public ActionResult LogOff()
    {
      WSFederationAuthenticationModule fam = FederatedAuthentication.
WSFederationAuthenticationModule;

      try
      {
        FormsAuthentication.SignOut();
      }
      finally
      {
        fam.SignOut( true );
      }

      SignOutRequestMessage signOutRequest = new SignOutRequestMessage( new Uri( fam.Issuer
), fam.Realm );
      return Redirect( signOutRequest.WriteQueryString() );
    }
```

LogOff is exactly as described above.

LogOn is overloaded. The first method is the entry point, triggered by the Forms authentication redirection; the second processes the POST with the *wresult* from the STS. They both take advantage of a common helper method, *LogOnCommon*, shown next:

```
  private ActionResult LogOnCommon( string returnUrl )
    {
      //
      // If the request is unauthenticated, Redirect to the STS with a protocol request.
      //
      if ( !Request.IsAuthenticated )
      {
        string federatedSignInRedirectUrl = GetFederatedSignInRedirectUrl( returnUrl );
        return Redirect( federatedSignInRedirectUrl );
      }

      //
      // Request is already authenticated.
      // Redirect to the URL the user was trying to access before being authenticated.
      //

      string effectiveReturnUrl = returnUrl;

      //
      // If no return URL was specified, try to get it from the Request context.
      //
      if ( String.IsNullOrEmpty( effectiveReturnUrl ) )
      {
        effectiveReturnUrl = GetContextFromRequest();
      }

      //
      // If there is a return URL, Redirect to it. Otherwise, Redirect to Home.
```

```
      //
      if ( !String.IsNullOrEmpty( effectiveReturnUrl ) )
      {
        return Redirect( effectiveReturnUrl );
      }
      else
      {
        return RedirectToAction( "Index", "Home" );
      }
    }
```

The flow is straightforward. If the user is not authenticated, the method creates a sign-in message from the config (via the helper method *GetFederatedSignInRedirectUrl*) and redirects accordingly.

If the user is already authenticated, the method extracts the intended return URL from the method parameter or from the WS-Federation context (via the helper method *GetContextFromRequest*), or it defaults to Home; then it redirects to it. In the following code block, you can find the helper methods for completeness:

```
  private string GetContextFromRequest()
    {
      Uri requestBaseUrl = WSFederationMessage.GetBaseUrl( Request.Url );
      WSFederationMessage message = WSFederationMessage.CreateFromNameValueCollection(
requestBaseUrl, Request.Form );
      return ( message != null ? message.Context : String.Empty );
    }

    private string GetFederatedSignInRedirectUrl( string returnUrl )
    {
      WSFederationAuthenticationModule fam = FederatedAuthentication.
WSFederationAuthenticationModule;
      SignInRequestMessage signInRequest = new SignInRequestMessage( new Uri( fam.Issuer ),
fam.Realm, fam.Reply )
      {
        AuthenticationType = fam.AuthenticationType,
        Context = returnUrl,
        Freshness = fam.Freshness,
        HomeRealm = fam.HomeRealm
      };
      return signInRequest.WriteQueryString();
    }
```

Finally, there are the elements that trigger the authentication. The log-in and log-out UI is driven by a simple control, which is autogenerated by the MVC template:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<%
    if (Request.IsAuthenticated) {
%>
        Welcome <b><%= Html.Encode(Page.User.Identity.Name) %></b>!
        [ <%= Html.ActionLink("Log Off", "LogOff", "Account") %> ]
<%
    }
```

```
    else {
%>
        [ <%= Html.ActionLink("Log On", "LogOn", "Account") %> ]
<%
    }
%>
```

Here's the controller and action method used for testing the *ReturnUrl* case:

```
public class SecretController : Controller
  {
    public SecretController()
      : this( null )
    {
    }

    [Authorize( Roles = "Manager" )]
    public ActionResult Index()
    {
      IClaimsPrincipal icp = User as IClaimsPrincipal;
      IClaimsIdentity ici = icp.Identity as IClaimsIdentity;
      Claim roleClaim = ici.Claims.First( c => c.ClaimType == ClaimTypes.Role );

      ViewData["role"] = roleClaim.Value;
      return View();
    }

  }
```

Just by looking at how the action method is constructed, you would not be able to tell if the application is configured to use the membership provider or an STS via WS-Federation. The only things in this sample giving away that we are playing with claims-based identity is the use of *IClaimsPrincipal*, which is intended to demonstrate that everything works as expected.

That's it! Naturally, this is far from being supported out of the box. Future versions of the products may or may not offer smoother ways of working together. However, in my opinion, the level of WIF-ASP.NET MVC integration you can achieve already today is more than enough for successfully tackling real-world scenarios.

Silverlight

I don't want to talk about ASP.NET MVC and Silverlight applications as a popularity contest. Let's just say that in the last three years I have been asked over and over again how to use claims-based identity in Silverlight, even before we started openly discussing Zermatt (the very first public codename for WIF).

Let's just get the unpleasant news out of the way. At the time of this writing, there is no WIF assembly in Silverlight 4.0 or any native claims-support capabilities. It is technically possible to add WIF-like support to Silverlight. The Identity Developer Training Kit, which I've

indicated as the main code reference throughout the entire book, has step-by-step guidance on how to do that in three different scenarios. However, as of today, that comes at the price of a large amount of custom code, which has to cover everything from the *IClaimsIdentity* equivalent classes to logic for wedging STS calls within the normal Silverlight application life cycle.

See the lab "Silverlight and Identity" at c:\IdentityTrainingKit2010\Labs\SilverlightAndIdentity.

To complicate things further, the architecture of Silverlight applications does not exactly fall into one of the active and passive categories defined earlier in the book; instead, it behaves a bit like an active client and a bit like a passive Web site. The icing on the cake is that Silverlight applications come in many different flavors, and one approach might be good for one flavor but fail in another, such as in-browser, rich Internet applications (RIA), out-of-browser, Microsoft Windows Phone 7, and so on.

The custom code used by the hands-on lab for adding WIF-like capabilities to the Silverlight application is far too complex to go into details with it here. Furthermore, it will certainly change going forward, so I would do you a disservice by being too prescriptive about it in this book. Feel free to experiment with that code—I know of customers who built their solutions and products with WIF and Silverlight, getting inspiration from it, but don't be surprised if it changes in the future. What I can do to help, though, is spend a bit more time on the peculiarities of the Silverlight application and the general approach the lab uses for addressing them. The code might change, but the patterns will be much more long-lived. In fact, the issues I explore here go beyond the specific Silverlight case and can also impact traditional rich clients (WPF, Windows Forms, or any other technology capable of invoking services). The discussion will be all at the architectural level; feel free to skip to the next section if theory is not your thing.

For Whom the Token Tolls

A typical in-browser Silverlight application, as I assume you know, is mainly constituted by one .XAP file residing on a Web site. When a browser with the appropriate plug-in lands on the right page, the XAP file is downloaded, unpacked, and fed to the Silverlight plug-in, which starts rendering the UI, calling services, and doing all the things that Silverlight applications like to do. The peculiar thing you should notice is that the Silverlight code is executed on the client's machine because it is at the very heart of the matter. Figure 7-3 provides a schematic representation of our Silverlight discussion.

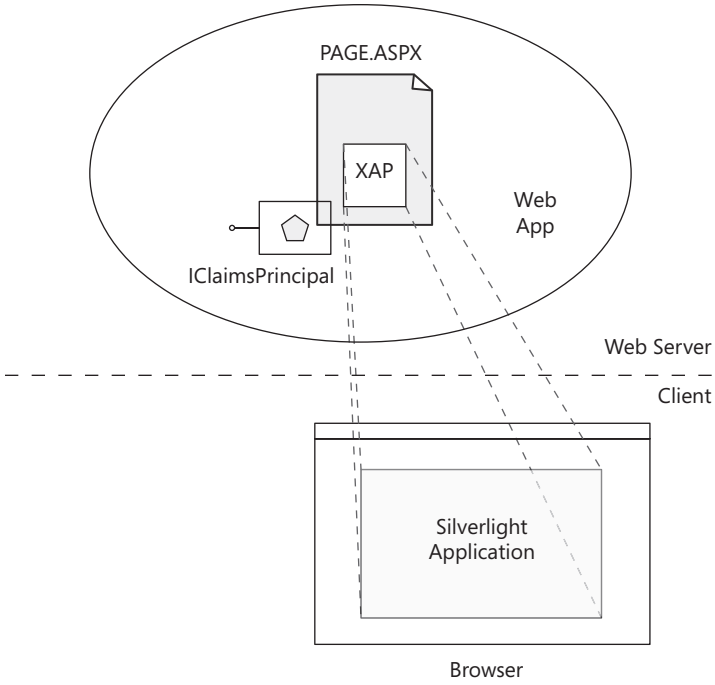


FIGURE 7-3 Anatomy of an in-browser Silverlight application

Now, imagine that the Web site hosting the Silverlight application is protected by WIF via classic passive federation. When the browser hits *Page.aspx*, the federation dance starts, and when the browser finally manages to successfully download and execute the .XAP, there is a nice session in place—that means a cookie on the client and availability of *IClaimsPrincipal* on the server. Are you starting to see the issue here? The Silverlight code does not actually have any access to the user claims: those are sealed in the cookie, encrypted for the server. The silver lining (no pun intended) is that any Web service hosted on the same Web site (no matter how simple) will be able to access the claims, thanks to the *SessionAuthenticationModule* and the session cookie. Because it is common practice to offer such services from the Web site to the Silverlight application, being able to use the standard *ClaimsAuthorizationManager* and similar equipment comes in handy.

Silverlight, as mentioned, admits many other application types. Out-of-browser applications download and execute XAP files without the need for a browser, which makes redirection-based authentication schemas pretty complicated or downright impossible. A Windows Phone application is a XAP file that resides directly on the phone's file system and behaves in all practical means as a rich client. Figure 7-4 summarizes the WS-Trust flow that such an application has to follow for invoking a service with a token from a given identity provider (IP).

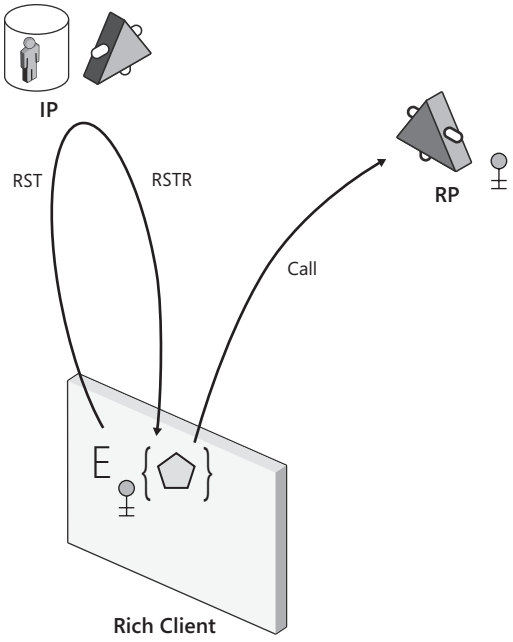


FIGURE 7-4 The token flow from the IP to the RP: in the general case, the token will be opaque to the rich client

Note Silverlight does not have the rich message-based security bindings that WCF offers on full-blown .NET desktop applications; however, with some effort it is possible to put together calls in a mixed mode and even support symmetric cryptography. Again, see the WIF and Silverlight labs at *c:\Identity\TrainingKit2010\Labs\SilverlightAndIdentity*.

Figure 7-4 stresses what you already know. Even in the case in which the client itself requests and obtains a security token, as opposed to delegating that task to browser redirects, the token content is still off limits. If the token is encrypted—and in general, I must assume it is—it is encrypted for its destination (the service) and is therefore opaque for the Silverlight code.

Note The service can still consume the token and relative claims, of course.

It turns out that developers, especially the ones who fancy user experience and such, don't care that much about application architecture and why they can't get the user info they need for driving their application's behavior. Because so much effort has been poured into the claims model for exposing user information, it's only natural to try making claims available on Silverlight applications and rich clients' code. As of today, there are a couple of approaches you can pursue. The first one is shown in Figure 7-5.

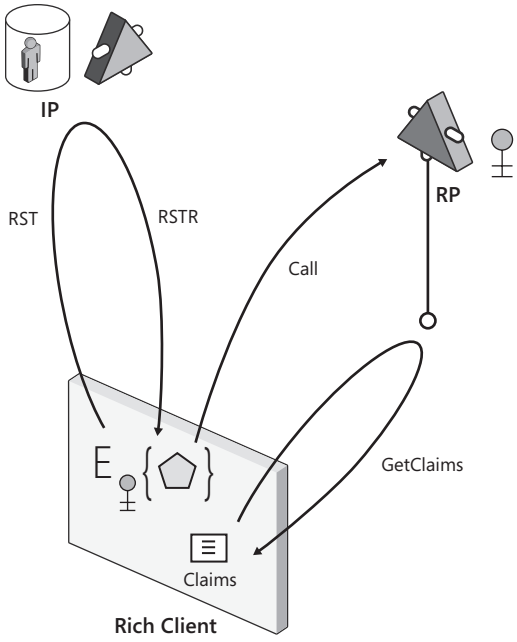


FIGURE 7-5 Making claims available on the client application by exposing a dedicated service on the relying party (RP)

This approach is somewhat similar to what RIA services do for supporting *IsInRole* in the Silverlight code. There is an extra service exposed on the RP side—a well-known service that the Silverlight application can call more or less transparently and that spills the beans about information that is normally available only on the service side, such as user name, roles, and so on. In the "Silverlight plus WIF" labs, the approach is the same: there is a service that provides the necessary information for rehydrating a copy of *IClaimsPrincipal* on the client. A lot of the code in the labs is devoted to declaratively specifying which identity provider and how to call it. There is also boilerplate code that wires up handlers for the acquisition and rehydration of *IClaimsPrincipal*, so that the assignment of *IClaimsPrincipal* in the current context can take place as transparently as possible. This is today's favorite approach for using WIF with Silverlight.

Here I enter in the realm of opinions, so take the following with a grain of salt. The approach just described derives from the nature of Silverlight applications, which can usually count on a "base" from where the main XAP is served and that hosts the services that often constitute some of the application's business logic. However, the approach is not easily generalized to the case in which the services used by the application are from third-party providers. The approach requires that those providers should all be convinced to expose a service not regulated by any of today's standards, which would necessarily have equal or lower security requirements than the existing API surface, and so on.

An alternative solution is to rely on the identity provider itself to make claims available to the client application. In fact, WIF already provides a mechanism for an IP to make claim values visible to a user agent. The mechanism is the *DisplayToken*, an artifact used by Windows CardSpace for showing to the user the claim values so that she can consent to that information being shared with the RP. It's that user consent about sharing the claims that determines whether the transaction will be aborted. Figure 7-6 demonstrates how the approach works.

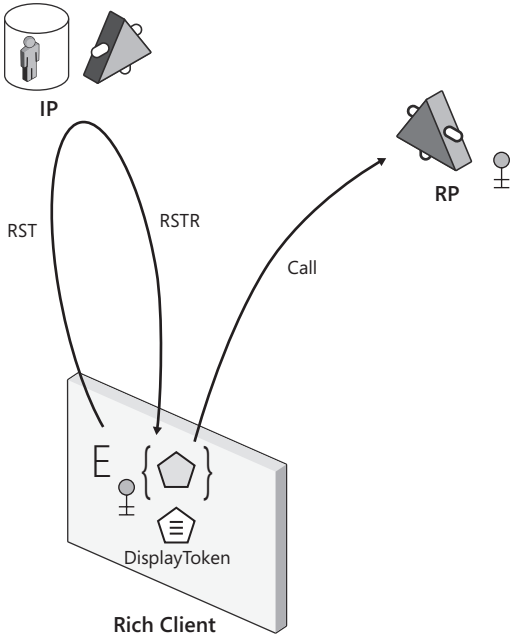


FIGURE 7-6 Making claims available on the client application by issuing a *DisplayToken* along with the requested token

Just like a proof token helps a client to use a key otherwise inaccessibly locked inside the token, the *DisplayToken* helps the client to use claim values buried behind the token's encryption. The format of the *DisplayToken* and the way in which it can be requested from one STS is described by OASIS Identity Metasystem Interoperability Technical Committee, a multivendor group. Again, WIF already has the necessary code for requesting and issuing *DisplayToken*s.

Note The IP has discretionary power over the fidelity with which claims are represented in the *DisplayToken*. There is no guarantee the claims in *DisplayToken* will be exactly the same as the ones in the actual token. On the other hand, that is also true if it is the RP that exposes a *GetClaims* service. That's why all the claims obtained in any way other than deserializing a signed token should have their *Issuer* property clearly advertise their lower trustworthiness.

As an architect, I find this solution appealing because it seems more natural (it asks issuers to issue, instead of overloading the RP role) and less tiresome (there are far fewer IPs than RPs, which makes the convincing part easier). I had one customer that hacked a solution based on this approach in less than one hour.

Is the WIF-Silverlight integration going to be easier in the future? Is it going to be based on the first solution, the second one, or a third one? It is hard to say. In any case, now you have a general understanding of what challenges the integration entails. That should enable you to navigate the solutions proposed in the Silverlight and WIF hands-on labs and tweak them to fit your scenario.

For a deeper discussion on the problem of making claims available to client applications, you can read the blog post at http://blogs.msdn.com/b/vbertocci/archive/2010/05/02/claims-on-the-client.aspx. If you are interested in knowing more about Windows CardSpace, you can refer to Understanding Windows CardSpace*, by Vittorio Bertocci, Caleb Baker, and Garrett Serack (Addison Wesley, 2007).*

SAML Protocol

The SAML 2.0 protocol is a well-established component of many sign-on solutions and products. SAML 2.0 is very similar to WS-Federation in terms of expressive power—so similar that certain artifacts are in fact overlapping (WS-Federation and SAML 2.0 metadata can coexist in the same file and be consumed by both without conflicts) and protocol transition solutions employing both are increasingly common. The main differences between the two—besides the obvious syntax and terminology dissimilarities—lie in the pluggability of multiple token types in WS-Federation and the use of profiles that SAML 2.0 employs for keeping interoperability requirements under control.

ADFS 2.0 supports the SAML 2.0 protocol natively; at the time of this writing, the current version of WIF does not. There is no technical reason that prevents WIF from handling the SAML 2.0 protocol—after all, ADFS 2.0 itself is built using WIF—that's just the way it is out of the box.

Will SAML 2.0 protocol support be added to future versions of WIF? As you know, I cannot say anything about it here. What I can say is that it is a popular feature request among customers and partners, which makes it a good candidate.

Note At the time of this writing, there is at least one commercial implementation of SAML 2.0 that extends WIF to support the SPLite profile.

In Chapter 3, "WIF Processing Pipeline in ASP.NET," you learned everything about how WIF processes WS-Federation sign-in flows and how the WSFAM and SAM modules collaborate with many other classes to move things forward. The WS-Federation-specific parts are, in fact, just in the WSFAM. If you wrote a "SAMLPAM" that handles SAML-P's authentication syntactic sugar, you'd be able to leverage all the other existing classes for managing SAML 2.0 token handling, session management, authentication and authorization checks, and so on. I am by no means suggesting that writing such a module would be easy, just that it's certainly within the realm of the possible.

It is perfectly feasible to use the WIF extensibility model for implementing SAMLP, and I would not be surprised if sample code demonstrating this in practice would emerge.

Web Identities and REST

If you speak about identity and software with nondevelopers (hint: it's not a good pickup line), chances are that their first thought will be about what they use for signing in to their favorite social-networking site, Web e-mail, or instant messaging service. At the time of this writing, the popularity of services such as Live Messenger, Facebook, Twitter, Google Gmail, and the like keeps growing. The more time users spend online and the more ubiquitous the chances of connecting are, the more interesting the data produced and associated to an account is. And the more useful services a user subscribes to, the more he wants the data attached to one service account to be available in another as well (and the less he wants to sign up and create new passwords). All those trends, combined with an increasing push toward Web programmability, created the perfect storm for nudging the Web community into devising interoperable ways of signing in across multiple Web sites, extracting data and, of course, handling the consent and authorization to do so.

The two most promising protocols in the Web space are *OpenID* and *OAuth 2.0*. Technically, they can be both handled by WIF, but (as you can guess, given their presence in this section) neither is implemented out of the box. In the following subsections, I'll spend a few words about these protocols and their relationship to WIF. I'll also provide you with a bit of history when appropriate—Web protocols move *fast* and without knowing the difference between OAuth 1.0 and 2.0, you might have a hard time interpreting the results of your searches if you decide to dig deeper into the subject.

Important Before enthusiasts about Web protocols attack me about this or that omission, be warned: this discussion is just meant to provide a basic context for the reader who knows absolutely nothing about them. The discussion here isn't meant to be exhaustive.

WS-* vs. Web Protocols

The protocols you have seen in the book have more than enough expressive power to solve the challenges I mention; however, that power often comes at a price that the Web community is unwilling to pay. WS-Trust, WS-Federation, and the SAML protocol all emerged first and foremost as ways of connecting businesses, federated partnerships, or both via secure communications. Rather than have exact requirements, they make implicit assumptions about the infrastructure they'll operate on—things like some PKI, some governance for managing relationships and metadata, clearly defined organization and security boundaries, and availability of programming stacks with advanced capabilities (such as WIF itself, Sun Metro and similar). Those requirements are put to good use, offering advanced security capabilities (end-to-end security, nonrepudiation, and so on) by incredibly simple tooling and programming models. (The Federation Utility Wizard is a glaring example of that.) Imagine all this as a high-tech tank that requires special fuel, but does incredible things and that you drive with a simple joystick.

On the Web, things are different. Users are only loosely associated with the services they use (which they access from an ever-growing variety of platforms and devices), there is no governance to speak of, and endpoints can appear and disappear overnight. Developers are perfectly comfortable coping with wild variability in conditions, are willing (and expect) to write code at the protocol level for integrating things often never meant to be integrated, and embrace dealing with thrift requirements if that means enabling a wider user population. On top of all this, transactions on the Web often do not warrant the same assurance levels as their business counterparts (think about a user returning to a blog to leave a comment with a unique nickname that no other commenter can use, versus sending the message that triggers the payment of all the monthly salaries of a large company).

Note The imbalance I've just described is rapidly changing as the sheer amount of personal information we make available online creates threats to our privacy and reputation in meatspace if misused. That creates pressure to add more and more business-level protections and mechanisms even in Web applications, which is precisely what is happening. With great power comes great responsibility.

I'll return to my earlier car analogy: the Web goes into places where the special fuel is not available and the thick-plated armor of the tank is not needed. It's better to have a car without antilock brakes and air conditioning, but that under the hood has a simple engine that can be maintained by anybody who does not fear getting his hands dirty.

When I witness rants and flames online about which system is better in absolute terms (WS-* and SAML versus Web protocols), I can't help but cringe. Every tool has its place. You would not go to school in an armored tank, but you would not deliver large quantities of cash riding your bike in a bad neighborhood, either.

OpenID

OpenID is many different things, but at the time of this writing it is mainly used as an authentication protocol for Web sites. It has been widely adopted and, despite the protocol specification constantly evolving, the level of interoperability between existing implementations (available on many different platforms) is surprisingly good.

The protocol is straightforward. Here's a simplified description of the main authentication flow:

1. The user lands on the RP, a Web site that supports OpenID. Depending on the Web site owner's design choices, the user can either be required to type in her own OpenID moniker or pick her OpenID provider (referred to as *OP*, which is an IP that understands the OpenID protocol) from a list of common ones. An OpenID moniker, or User-Supplied Identifier, is a URL typically including part of the DNS name of the OP. Note that the URL can also be an e-mail address.
2. The RP discovers the OP, which mainly means finding out the exact address and protocol version the OP supports. After that, there's typically a phase in which a shared secret is negotiated between the RP and the IP.
3. The RP redirects the user to the OP by crafting an Authentication request, the OpenID equivalent of a wsignin1.0 message in WS-Federation.
4. The OP does whatever it deems necessary to authenticate the user. If everything goes as expected, it sends the user back to the RP with a positive assertion. (You got it—it's roughly a *wresult*.)
5. The RP verifies the incoming positive assertion with some standard checks (counterparts of *AudienceRestriction*, issuer, signature verification, replay detection, and so on).

I'm sure you didn't miss the striking similarities with WS-Federation. OpenID is more relaxed about the cryptography—for example, OPs are not required to have their own signing certificate, which has implications about token validation and issuer verification—but ultimately it's a redirect protocol. OpenID also must deal with some of the challenges you have seen in WS-Federation, which are amplified by the scale of the Web and the looser associations of users to Web sites. For example, the OpenID issues with home-realm discovery are so notorious that they earned a nickname for themselves: the *NASCAR problem*, because the amount of OP logos often listed by the RPs resembles the stickers covering every inch of racing cars. The community is well conscious of those issues, and the protocol is in constant evolution (which also makes it something of a moving target). For example, at the time of this writing there is a proposal to integrate information cards with the OpenID flow to solve the NASCAR problem and mitigate the phishing-prone aspects of the OpenID protocol.

Implementing OpenID in WIF requires approximately the same approach suggested for SAML-P—that is, writing your own "OpenIDAM" (or waiting for somebody else to write one), which replaces the WSFAM for OpenID Web sites. Some things are dramatically simpler than

the SAML-P case (like the protocol in itself); others require a bit more work to be handled by WIF's object model. (Whereas SAML-P circulates a SAML 2.0 token, OpenID assertions are not even based on XML.) Even more so than with the SAML protocol, I would not be surprised to see a sample implementation of OpenID in WIF emerge at some point in the community.

If you want to know more about OpenID, the best resource is definitely the Web site of the OpenID foundation at http://OpenId.net.

OAuth 2.0

Note This section is going to be history-intensive, but I really need to go through that or it will be hard for you to put in context all the literature available on OAuth. Searching for OAuth on the Internet will return a potpourri of documents coming from different epochs, but only the latest are actionable.

The OAuth protocol appeared on the scene in late 2007 and rapidly became a strong reference in the Web identity landscape. At the time, the idea of a Web API was in full bloom—Flickr is the one I readily remember, but there were many others. The success of initiatives like the Facebook applications platform provided strong momentum across the board, surfacing the effect of API usage to end users and sealing the deal. And sure enough, any kind of API fronting personal data has to cope with identity and access sooner than later—the Web APIs were no exception.

Imagine the following situation. A user takes advantage of a given Web application, named *A*, for storing his pictures online. At a certain point, the user decides to print some of them by using another Web application, *B*. How would you design such a system? Re-uploading the pictures in B is not a very nice experience, and at times it might be downright unfeasible. One brute-force approach, which is fortunately in apparent decline nowadays, is for B to prompt the user for his A credentials; that allows B to (literally) impersonate the user at A and use whatever API A exposes (or, if everything fails, screen-scraping) to acquire the photos to be printed. If after having read this entire book this approach does not make you cringe, I failed as an author. B can store the user credentials for A and misuse them at a later date, or B can use unsecure stores and expose them to theft and many other unpleasantries. Worst of all, if this becomes a widespread practice, we'd have to train our users to disclose their passwords to third parties, which paves the way to a phishing Wild West. OAuth was born precisely to avoid that problem, and the example I gave appears in almost every OAuth explanation.

I won't go into the details of the first release of OAuth because it is being superseded by 2.0. Let's just say that it introduced to the Web the idea of using an authorization token, something you are already familiar with. In the example I gave, the user would use his own credentials at A for obtaining a token that B can use for retrieving the photos the user wants to print.

Note If that reminds you of an STS exchange, you are on the right route. However, one should not push the analogy too far because the relationships here are different. For example, instead of establishing trust between B and A, here B would be given by A an API key that grants B the right to invoke the methods A exposes. Similar in theory, very different in practice.

OAuth 1.0 was (and is) very popular, especially with important players. I've been involved in the creation of the REST API platform of a large social network, which was based on OAuth 1.0; and at the time of this writing, Twitter and various other Web applications still use it (although Twitter declared their intention to move to 2.0).

Mass adoption of OAuth 1.0 was hampered by the complexity of the cryptography it required and because the protocol was not good at scaling to the needs of large providers. Furthermore, people tried to stretch its use beyond the Web-based application to the resource flow it was centered on—exposing the limitations of the approach—while at the same time, other players were moving forward with more comprehensive solutions (think Facebook Connect).

In November 2009, Microsoft, Google, and Yahoo! presented WRAP (Web Resource Authorization Protocol), a new protocol that had similar expressive power as OAuth 1.0 but did not suffer from some of its shortcomings. The community liked it so much that it was voted to be considered an OAuth profile and dubbed OAuth WRAP. Some implementations of the protocol emerged—notably, the first version of the Access Control service from the Windows Azure AppFabric was entirely based on it.

In April 2010, the first draft of OAuth 2.0 was published. The new version of the protocol is now backward compatible with OAuth 1.0 and is largely based on OAuth WRAP. Facebook and Twitter already offer implementations based on the current draft.

Now that you have some background, let me give you an overview of the protocol itself.

A Primer on the OAuth 2.0 Protocol In this case, the game is also largely a matter of moving tokens around. These are not SAML tokens (or at least in the most common cases); they are much simpler, REST-friendly artifacts, but tokens nonetheless. The ultimate purpose of most transactions is to obtain an access token, which is what OAuth 2.0 uses for authorizing access to protected resources. All communications are expected to take place in Secure Sockets Layer (SSL), which eases the burden of baking security mechanisms into the protocol itself.

OAuth 2.0 defines various roles that can be used for describing multiple authorization transactions:

- **Protected Resource** This is the reason for the whole brouhaha: the resource you need to access. In the photo printing example I mentioned earlier, the photos to be printed would be the protected resources.
- **Client** This is the entity that wants to access the resource. In the earlier example, that would be the printing service.

- **Resource Server** This is the server that authorizes requests for protected resources; it's the entity that processes access tokens (see below). It's the photo-storing Web site in the example.
- **Authorization Server** This is the server that can issue authorization tokens. It can be the same as the resource server; in the photo printing example, it is. The authorization service is supposed to have both token endpoints (programmatic endpoints used for sending access token requests) and end user endpoints (pages that can gather the user's consent to allow the app to access the protected resource and according to what terms).

OAuth 2.0 defines many flows, called *profiles*, each representing a different topology. There is still the Web-based application one, but now there are also profiles that detail how to use OAuth with a rich client, with a browser, and many others. Here I am going to spend few words just on the Web server profile. Take a look at the diagram in Figure 7-7.

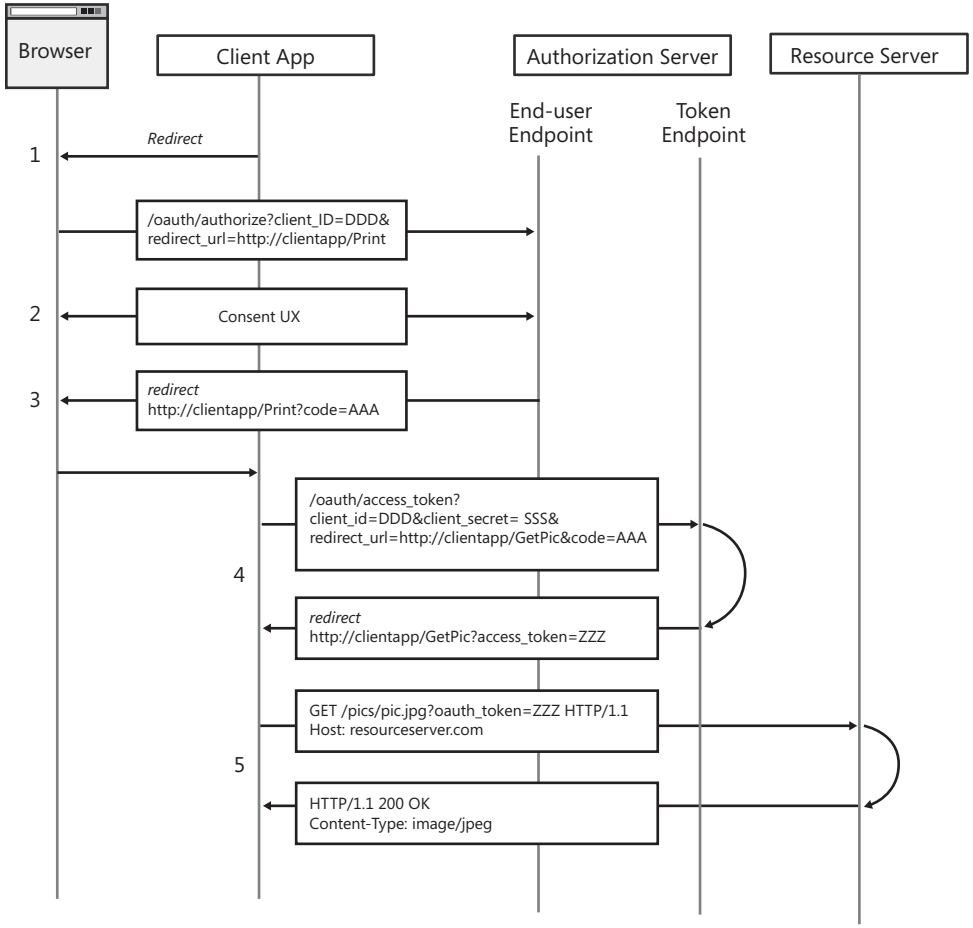


FIGURE 7-7 The OAuth 2.0 access flow for the generic Web Server profile

Here's a description of the flow shown in Figure 7-7:

1. The client application needs to access a protected resource hosted elsewhere. It starts the authorization flow by redirecting the end user's agent (the browser) to the /oauth/authorize endpoint of the appropriate authorization server. It includes in the request its own *client_id*. Often that represents the API key received from the service provider. The request also includes the URL to which the flow should be redirected once an access code has been obtained.
2. The authorization server takes whatever steps it deems necessary for establishing whether the user who owns the resource consents to the use that the client application wants to make of the resource. This can include steps such as authenticating the user and showing a consent UI about the resource and the action requested. If you ever installed an application in Facebook, you have a good idea of what goes on in this step from the user perspective. (Note that in the Facebook case you are usually already authenticated and the consent prompt takes place in the context of a session.)
3. Assuming that everything went as expected, the authorization server sends to the redirect URL the access code that can be used to obtain the access token needed for getting hold of the resource.
4. The client application goes ahead with the request for an access token, including its own credentials (*client_id* and *client_secret*, the "API key") and the return URL *redirect_url* in the message. A successful request will result in a redirect on *redirect_url*, including the bits of the access token in *oauth_token*.
5. The client application can finally use the access token to reach the resource (in the photo printing example, those would be the images to be printed) by including the token in the Authorization HTTP header, in the HTML body or, as shown in Figure 7-7, in the *oauth_token* query string parameter. If the resource server (which can and very often is the same entity as the authorization server) successfully validates the token, access to the resource is granted.

That is fairly simple, especially after having read an entire book on identity-oriented transactions—you just need to wrap your head around the terminology. There is so much more than this in OAuth 2.0, but you can easily explore it on your own—the OAuth 2.0 specification is very readable.

Just like WS-*, OAuth 2.0 does not mandate a specific token format to use with the aforementioned flow. However, OAuth WRAP did suggest a specific token format: the Simple Web Token (SWT). Despite there being no formal commitment to it, the format is in use. (The Windows Azure AppFabric Access Control Service takes advantage of it.) SWT defines many of the concepts you encountered for SAML (issuer, expiration, intended audience, and of course claims, and so on) in a much nimbler package, which needs no angle brackets and uses simplified HMAC-based signatures.

Note The OAuth 2.0 specification takes a liberal approach to the use of the term "token," whereas in this book a token has always been a signed artifact normally containing claims. In this section, the definition is weakened accordingly to accommodate the less specific meaning used in OAuth 2.0.

Integrating WS-Trust and OAuth 2.0 OAuth 2.0 admits hybrid flows in which REST resources can be secured by users who present SAML tokens (or any other token in use in an arbitrary trust framework) as proof of authentication. In fact, the authorization code returned in step 3 of Figure 7-7 is just a special case of the more generic concept of *access grant*, which is anything that authorizes the client to obtain an access token from the authorization server. To be more formal, an access grant can be either of the following:

- **An authorization code** You saw this in action earlier.
- **A user name and password** This options is for cases in which the resource owner and authorization server are collapsed in a single entity.
- **An assertion** This is pretty much anything that comes from any trust framework and that the authorization server is willing to validate. You can bet that the assertion will be a SAML token very, very often.

That capability—already present in OAuth WRAP—has been leveraged by the first version of the AppFabric Access Control Service for enabling domain users (or anybody who can obtain a SAML token) to use ADFS 2.0 to access REST services secured via OAuth. Figure 7-8 summarizes the flow using a generic WS-Trust endpoint.

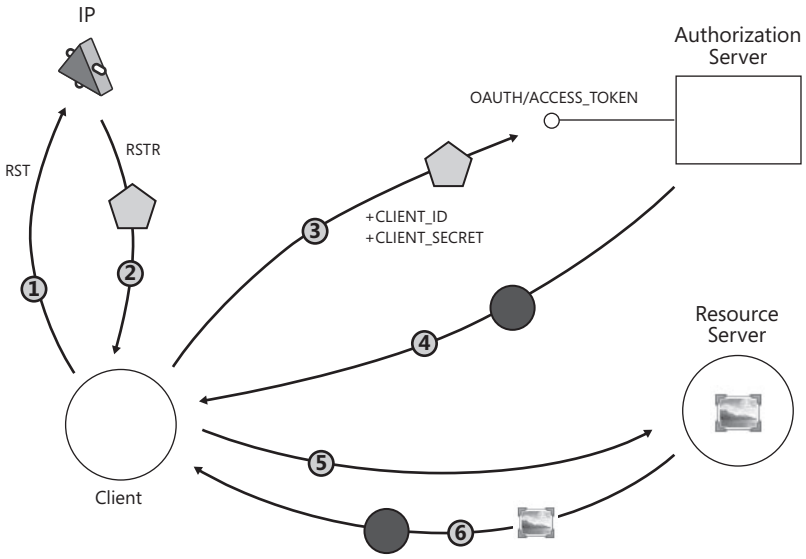


FIGURE 7-8 Leveraging the assertion access grant mode in OAuth 2.0 for bridging different trust frameworks

Here is an explanation of the steps shown in Figure 7-8:

1. The client (which, I remind you, is a Web application in the profile I am discussing) sends a Request for Security Token (RST) to one WS-Trust STS for one SAML token. The SAML token is scoped for the authorization server and will have subject confirmation method *bearer-token*, because it is meant to be used in an OAuth 2.0 request, which would not know how to handle *holder-of-key*.
2. Assuming that the authentication is successful, the STS sends back the required SAML token.
3. The client sends a request for an access token to the authorization server. The client includes in the request the *client_id*, the *client_secret*, and the raw SAML token received in step 2.
4. The authorization server validates the incoming credentials and, if satisfied by the results, issues an access token for the requested resource.
5. The client uses the access token so obtained for requesting a resource from the resource server.
6. The resource authorizes the request and sends back the desired resource.

Before the existence of the *TrustChannel*, this could have been hard to do, but with WIF it's a breeze. The advantage of a similar approach is that you can instantly enable all the users from a "traditional" identity provider (think all the Active Directory users in a network where ADFS 2.0 is available) to be authorized when accessing REST resources on the Web.

OAuth 2.0 and WIF Like in the SAML-P case, I am not going to describe a complete implementation of OAuth 2.0 for WIF. The purpose of this section is to help you get to know OAuth 2.0, give you a feeling of what it can be useful for, and help you find your way through all the documentation about it on the Internet. In any case, OAuth 2.0 is significantly simpler than SAML-P, and I am positive that a sample will appear soon, perhaps even before this book goes into print.

Just in case you can't wait, though, the actual question I want to make sure we can answer is this: "If you were the one writing the sample, what would you need to provide?"

If you came this far in the book, you can probably answer this yourself and take this chance to assess how much you have learned about WIF use and its structure. Put down the book for a moment and try to design a strategy for extending WIF with OAuth capabilities. Then come back to the text and see if your solution matches the guidelines I give next.

Let's start from the server side. Figure 7-7 admittedly gives a partial view of what the full protocol requires; however, it clearly shows that there are a lot of redirect responses. Furthermore, given the context, one can expect that those transactions will involve generic Web resources rather than exclusive services. Finally, there are cases in which the authorization server and resource server can be the same entity, so the solution must easily allow

for that. One element that allows you to model all those requirements is the *HttpModule*: by writing your own *HttpModule*, you'd be able to keep those functions outside of the application logic. Of course, this would have to play nice with other modules it might end up coexisting with on the same Web site, especially WSFAM and the Forms authentication module. Another thing that would be nice is to make sure that as much WIF pipeline as possible is preserved in your module—for example, calling *ClaimsAuthenticationManager* and *ClaimsAuthorizationManager* would be your responsibility.

The authorization server is required to validate different access grant types. What is WIF's mechanism for coping with that? That's right, *SecurityTokenHandler*s. If you build things right, assessing authorization codes and SAML assertions require the same code. Of course, that means you have to write custom token handlers for the token types used here (for example, the SWT), but the simplicity of the formats should make that a walk in the park. You might need to deal with state (especially if for you access tokens are just identifiers), but that should not be too hard.

Trust management occupies an important portion of WIF's handling of WS-Federation and WS-Trust. What would be the counterparts here? The authorization server needs to keep track of its clients and client secrets. The resource server needs to know which key should be used for checking the signature (if any) on the access token. If the access grant is a SAML token, its signature must be verified by the authorization server. And so on, and so forth.

The client has its own needs, too. The query string needs to be populated or parsed according to the leg of the protocol you are in. Messages need to be crafted. You need to decide if you want to hide the acquisition of the access token (à la *ws2007FederationHttpBinding*) or make it explicit (à la *WSTrustChannel* and *CreateChannelWithIssuedToken*). Refresh tokens, an aspect of the OAuth 2.0 specification I didn't discuss, make things even more interesting.

Let me restate my point: you almost certainly don't need to write your own plumbing for integrating WIF with OAuth, because I am positive that an example will come out sooner or later. Nor do you need to understand the code of that example in detail in order to use it—much like you don't need to see the source code of WSFAM for being able to take advantage of it. This is the reason I used this last section about figuring out a solution just for didactic purposes.

Conclusion

This chapter concludes the journey through WIF and claims-based identity that this book offered, and it shows that there is so much more to learn than the few things I managed to cover here.

Windows Identity Foundation is the first developer stack that makes it really easy to take advantage of claims-based identity. Part I of the book showed that WIF allows strict

separation of concerns and empowers nonexperts to make their application safer without having to learn much more than they already know; Part II dug deep into WIF's programming model, identity architecture, and industry standards. Although the syntax might change through the years as the products get updated, I am confident that the insights on the problem space and the solution patterns based on claims will prove handy in your career for many years to come.

If you want to stay up to date on the topic, there are many online resources you can take advantage of:

- The WIF product home page on *http://www.microsoft.com/wif*
- My own blog at *http://www.cloudidentity.net*
- The Identity Developer Training Kit at *http://go.microsoft.com/fwlink/?LinkId=148795*
- The WIF team blog at *http://blogs.msdn.com/card*
- The IdElement Show on Channel9: *http://channel9.msdn.com/shows/Identity/*

The claims revolution has barely started, and it's already changing the way in which the industry does identity on-premises and in the cloud. I hope this book inspired you to be a part of it!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

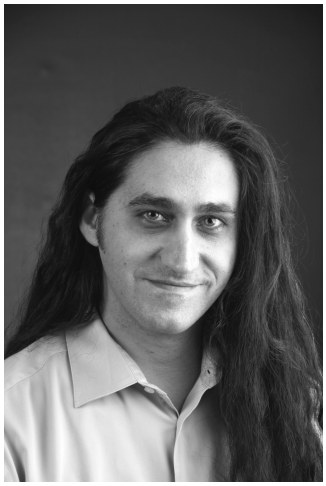
U

V

W

X

Vittorio Bertocci



Vittorio Bertocci is a Senior Architect Evangelist in Developer and Platform Evangelism (DPE) and a key member of the extended engineering team that produces Microsoft's claims-based platform components (for example, Windows Identity Foundation and ADFS 2.0). He is responsible for identity evangelism for the .NET developer community and drove initiatives such as the Identity Developer Training Kit (*http://go.microsoft.com/fwlink/?LinkId=148795*) and the IdElement show (*http://channel9.msdn.com/shows/identity/*) on Channel 9.

Vittorio holds a master degree in Computer Science, and he began his career doing research on computational geometry and scientific visualization. In 2001, he Joined Microsoft Italy, where he immediately focused on the .NET platform and the nascent field of Web services security, becoming a reference at the national and European level.

In 2005, Vittorio moved to Redmond, where he helped to launch the .NET Framework 3.5 by working with Fortune 100 and Global 100 companies on cutting-edge SOA projects based on WCF, WF, and CardSpace. He became more and more focused on identity themes, eventually undertaking his current mission of evangelizing claims-based identity into mainstream use.

In the last five years, this mission has led him to speak about identity in 23 countries and 4 continents. Vittorio is a regular speaker at conferences such as Microsoft PDC, TechEd USA, TechEd Europe, TechEd Australia, TechEd New Zealand, TechEd Japan, TechDays Belux, Gartner Summit, European Identity Conference, IDWorld, OreDev, NDC, IASA, Basta and many others.

Vittorio is a published author, both in the academic and industry worlds, and has written many articles and papers. He is co-author of *A Guide to Claims-Based Identity and Access Control* (Microsoft Press, 2010) and *Understanding Windows CardSpace* (Addison-Wesley, 2008). He is a prominent authority/blogger on identity, Windows Azure, .NET development, and related topics, and he shares his thoughts at www.CloudIdentity.net.

Vittorio lives in the lush, green city of Redmond with his wife, Iwona. He doesn't mind the gray skies too much, but every time he has half a chance he flies to some beach place, be it Hawaii or Camogli, his home town in Italy.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press

Stay in touch!

To subscribe to the *Microsoft Press® Book Connection Newsletter*—for news on upcoming books, events, and special offers—please visit:

microsoft.com/learning/books/newsletter