

# Modern Authentication with Azure Active Directory for Web Applications

Foreword by Mark E. Russinovich  
*Chief Technology Officer, Microsoft Azure*



Vittorio Bertocci

# Modern Authentication with Azure Active Directory for Web Applications

Vittorio Bertocci

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

Copyright © 2016 by Vittorio Bertocci. All rights reserved.

No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2014954517  
ISBN: 978-0-7356-9694-5

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at [mspinput@microsoft.com](mailto:mspinput@microsoft.com). Please tell us what you think of this book at <http://aka.ms/tellpress>.

This book is provided "as-is" and expresses the author's views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at [www.microsoft.com](http://www.microsoft.com) on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

**Acquisitions and Developmental Editor:** Devon Musgrave  
**Project Editor:** John Pierce  
**Editorial Production:** Rob Nance, John Pierce, and Carrie Wicks  
**Copyeditor:** John Pierce  
**Indexer:** Christina Yeager, Emerald Editorial Services  
**Cover:** Twist Creative • Seattle and Joel Panchot

*Ai miei carissimi fratelli e sorelle: Mauro, Franco, Marino, Cristina, Ulderico, Maria, Laura, Guido e Mira—per avermi fatto vedere il mondo attraverso altre nove paia d'occhi.*



# Contents

|   |             |
|---|-------------|
| <i>Foreword</i> .....   | <i>xi</i>   |
| <i>Introduction</i> .....   | <i>xiii</i> |
| <b>Chapter 1 Your first Active Directory app</b>                            | <b>1</b>    |
| The sample application .....  | 1           |
| Prerequisites .....   | 1           |
| Microsoft Azure subscription .....  | 2           |
| Visual Studio 2015 .....  | 2           |
| Creating the application .....  | 3           |
| Running the application .....   | 4           |
| <i>ClaimsPrincipal</i> : How .NET represents the caller .....               | 7           |
| Summary .....   | 10          |
| <b>Chapter 2 Identity protocols and application types</b>                   | <b>11</b>   |
| Pre-claims authentication techniques .....                                  | 12          |
| Passwords, profile stores, and individual applications .....                | 12          |
| Domains, integrated authentication, and applications on an intranet .....   | 14          |
| Claims-based identity .....   | 17          |
| Identity providers: DCs for the Internet .....                              | 17          |
| Tokens .....  | 18          |
| Trust and claims .....  | 20          |
| Claims-oriented protocols .....   | 20          |
| Round-trip web apps, first-generation protocols .....                       | 23          |
| The problem of cross-domain single sign-on .....                            | 23          |
| SAML .....  | 25          |
| WS-Federation .....   | 27          |
| Modern apps, modern protocols .....   | 31          |
| The rise of the programmable web and the problem of access delegation ..... | 32          |

|                  |  |           |
|------------------|--|-----------|
|                  | OAuth2 and web applications . . . . .  | 33        |
|                  | Layering web sign-in on OAuth. . . . .   | 37        |
|                  | OpenID Connect. . . . .  | 39        |
|                  | More API consumption scenarios . . . . .   | 44        |
|                  | Single-page applications . . . . .   | 45        |
|                  | Leveraging web investments in native clients. . . . .                              | 47        |
|                  | Summary. . . . .   | 49        |
| <b>Chapter 3</b> | <b>Introducing Azure Active Directory and Active Directory Federation Services</b> | <b>51</b> |
|                  | Active Directory Federation Services. . . . .                                      | 52        |
|                  | ADFS and development . . . . .   | 53        |
|                  | Getting ADFS . . . . .   | 54        |
|                  | Protocols support. . . . .   | 55        |
|                  | Azure Active Directory: Identity as a service . . . . .                            | 56        |
|                  | Azure AD and development. . . . .  | 60        |
|                  | Getting Azure Active Directory. . . . .  | 61        |
|                  | Azure AD for developers: Components . . . . .                                      | 63        |
|                  | Notable nondeveloper features . . . . .  | 65        |
|                  | Summary. . . . .   | 67        |
| <b>Chapter 4</b> | <b>Introducing the identity developer libraries</b>                                | <b>69</b> |
|                  | Token requestors and resource protectors . . . . .                                 | 69        |
|                  | Token requestors . . . . .   | 70        |
|                  | Resource protectors. . . . .   | 73        |
|                  | Hybrids. . . . .   | 74        |
|                  | The Azure AD libraries landscape . . . . .   | 75        |
|                  | Token requestors . . . . .   | 76        |
|                  | Resource protectors. . . . .   | 81        |
|                  | Hybrids. . . . .   | 85        |
|                  | Visual Studio integration . . . . .  | 85        |
|                  | AD integration features in Visual Studio 2013 . . . . .                            | 86        |
|                  | AD integration features in Visual Studio 2015 . . . . .                            | 86        |
|                  | Summary. . . . .   | 87        |

|                  |  |            |
|------------------|--|------------|
| <b>Chapter 5</b> | <b>Getting started with web sign-on and Active Directory</b>     | <b>89</b>  |
|                  | The web app you build in this chapter                            | 89         |
|                  | Prerequisites  | 90         |
|                  | Steps  | 90         |
|                  | The starting project   | 90         |
|                  | NuGet packages references  | 92         |
|                  | Registering the app in Azure AD                                  | 93         |
|                  | OpenID Connect initialization code                               | 95         |
|                  | Host the OWIN pipeline   | 95         |
|                  | Initialize the cookie and OpenID Connect middlewares             | 96         |
|                  | [Authorize], claims, and first run                               | 97         |
|                  | Adding a trigger for authentication                              | 97         |
|                  | Showing some claims  | 98         |
|                  | Running the app  | 99         |
|                  | Quick recap  | 99         |
|                  | Sign-in and sign-out   | 99         |
|                  | Sign-in logic  | 100        |
|                  | Sign-out logic   | 101        |
|                  | The sign-in and sign-out UI                                      | 102        |
|                  | Running the app  | 103        |
|                  | Using ADFS as an identity provider                               | 103        |
|                  | Summary  | 105        |
| <b>Chapter 6</b> | <b>OpenID Connect and Azure AD web sign-on</b>                   | <b>107</b> |
|                  | The protocol and its specifications                              | 107        |
|                  | OpenID Connect Core 1.0  | 108        |
|                  | OpenID Connect Discovery   | 108        |
|                  | OAuth 2.0 Multiple Response Type, OAuth2 Form Post Response Mode | 109        |
|                  | OpenID Connection Session Management                             | 109        |
|                  | Other OpenID Connect specifications                              | 109        |
|                  | Supporting specifications  | 109        |



|   |            |
|---|------------|
| OpenID Connect exchanges signing in with Azure AD . . . . .                     | 110        |
| Capturing a trace . . . . .   | 110        |
| Authentication request . . . . .  | 113        |
| Discovery . . . . .   | 119        |
| Authentication . . . . .  | 122        |
| Response . . . . .  | 123        |
| Sign-in sequence diagram . . . . .  | 126        |
| The ID token and the JWT format . . . . .                                       | 127        |
| OpenID Connect exchanges for signing out from the app<br>and Azure AD . . . . . | 134        |
| Summary . . . . .   | 136        |
| <b>Chapter 7 The OWIN OpenID Connect middleware</b>                             | <b>137</b> |
| OWIN and Katana . . . . .   | 137        |
| What is OWIN? . . . . .   | 137        |
| Katana . . . . .  | 139        |
| OpenID Connect middleware . . . . .   | 155        |
| <i>OpenIdConnectAuthenticationOptions</i> . . . . .                             | 155        |
| Notifications . . . . .   | 159        |
| <i>TokenValidationParameters</i> . . . . .                                      | 167        |
| Valid values . . . . .  | 168        |
| Validation flags . . . . .  | 169        |
| Validators . . . . .  | 169        |
| Miscellany . . . . .  | 170        |
| More on sessions . . . . .  | 171        |
| Summary . . . . .   | 172        |
| <b>Chapter 8 Azure Active Directory application model</b>                       | <b>173</b> |
| The building blocks: <i>Application</i> and <i>ServicePrincipal</i> . . . . .   | 174        |
| The <i>Application</i> . . . . .  | 177        |
| The <i>ServicePrincipal</i> object . . . . .                                    | 187        |
| Consent and delegated permissions . . . . .                                     | 189        |
| Application created by a nonadmin user . . . . .                                | 189        |
| Interlude: Delegated permissions to access the directory . . . . .              | 192        |

|   |     |
|---|-----|
| Application requesting admin-level permissions . . . . .      | 197 |
| Admin consent . . . . .                                       | 200 |
| Application created by an admin user . . . . .                | 204 |
| Multitenancy . . . . .  | 205 |
| App user assignment, app permissions, and app roles . . . . . | 211 |
| App user assignment . . . . .                                 | 211 |
| App roles . . . . .   | 213 |
| Application permissions . . . . .                             | 216 |
| Groups . . . . .  | 219 |
| Summary . . . . .   | 221 |

**Chapter 9 Consuming and exposing a web API protected by Azure Active Directory 223**

|  |     |
|--|-----|
| Consuming a web API from a web application . . . . .   | 223 |
| Redeeming an authorization code in the OpenID Connect hybrid flow . . . . .                          | 224 |
| Using the access token for invoking a web API . . . . .  | 232 |
| Other ways of getting access tokens . . . . .  | 251 |
| Exposing a protected web API . . . . .   | 253 |
| Setting up a web API project . . . . .   | 253 |
| Handling web API calls . . . . .   | 258 |
| Exposing both a web UX and a web API from the same Visual Studio project . . . . .                   | 265 |
| A web API calling another API: Flowing the identity of the caller and using “on behalf of” . . . . . | 266 |
| Protecting a web API with ADFS “3” . . . . .   | 271 |
| Summary . . . . .  | 272 |

**Chapter 10 Active Directory Federation Services in Windows Server 2016 Technical Preview 3 273**

|  |     |
|--|-----|
| Setup (for developers) . . . . .                   | 273 |
| The new management UX . . . . .                    | 274 |
| Web sign-on with OpenID Connect and ADFS . . . . . | 276 |
| OpenID Connect middleware and ADFS . . . . .       | 276 |

|  |         |
|--|---------|
| Setting up a web app in ADFS .....   | 277     |
| Testing the web sign-on feature .....  | 280     |
| Protecting a web API with ADFS and invoking it from<br>a web app .....       | 281     |
| Setting up a web API in ADFS .....   | 281     |
| Code for obtaining an access token from ADFS and invoking<br>a web API ..... | 285     |
| Testing the web API invocation feature .....                                 | 288     |
| Additional settings .....  | 289     |
| Summary .....  | 292     |
| <br><i>Appendix: Further reading</i>   | <br>293 |
| <br><i>Index</i>   | <br>295 |

# Foreword

The purpose of an application is to take input from users or other applications and produce output that will be consumed by those same users or applications or by other ones. That's true of a website that gains input from a click on a link and sends back the content of the requested page as output; a middle tier that processes database requests queued from a front end, executing them by sending input to a database; or a cloud service that gets input from a mobile application to look up nearby friends. Given this, a fundamental question faced in the design of every application is, Who is sending the input and should the application process it to produce the resulting output? Put another way: every application must decide on an identity system that represents users and other applications, a means by which to validate an application's or user's claimed identity, and a way to determine what outputs the user or application is allowed to produce.

These decisions will determine how easily users and applications can interact with an application, what functionality they can take advantage of to secure and manage their identities and credentials, and how much work the application developer must do to enable these capabilities, which are known as authentication and authorization. The ideal answers make it possible for users and applications to use their preferred identities, whether from Facebook, Gmail, or their enterprise; for the application to easily configure the access rights for authorized users; and for the application to rely on other services as much as possible to do the heavy lifting. Identity and access control, while key to an application's utility, are not the core value an application delivers, so developers shouldn't spend any more time on this area than they have to. Why create a database of users and worry about which algorithm to use to encrypt the users' passwords if you can take advantage of a service that's built for doing just that, with industry-leading security and management?

Microsoft Azure Active Directory (Azure AD) is arguably the heart of Microsoft's cloud platform. All Microsoft cloud services, including Microsoft Azure, Microsoft Xbox Live, and Microsoft Office 365, use Azure AD as their identity provider. And because Azure AD is a public cloud service, application developers can also take advantage of its capabilities. If an application relies on Azure AD as its identity provider, it can rely on Azure AD APIs to provision users, rely on Azure AD to manage their passwords, and even give users the ability to use multifactor authentication (MFA) to securely authenticate to the application. For application developers wanting to integrate with businesses, including the many that are already using Azure AD, Azure AD has the most flexible and comprehensive support of any service for integrating Active Directory and LDAP identities. Fueled by enterprise adoption of Office 365, Azure AD is already a

connection point for hundreds of millions of business and organizational identities, and it's growing fast.

Using Azure AD for the most common scenarios is easy, thanks to the open source developer libraries, tooling, and guidance available on Microsoft Azure's GitHub organization. Going beyond the basics, however, requires a good understanding of modern authentication flows—specifically OAuth2 and OpenID Connect—and concepts such as a relying party and tokens, federation, role-based access control, a provisioned application, and service principles. If you're new to these protocols and terms, the learning curve can seem daunting. Even if you're not, knowing the most efficient way to use Azure AD and its unique capabilities is important, and it's worthwhile understanding what's available to you.

There's no better book than *Modern Authentication with Azure Active Directory for Web Applications* to help you make your application take full advantage of Azure AD. I've known Vittorio Bertocci since I started in Azure five years ago, and I've watched his always popular and highly rated Microsoft TechEd, Build, and Microsoft Ignite conference presentations to catch up with the latest developments in Azure AD. He's a master educator and one of Microsoft's foremost experts on identity and access control.

This book will guide you through the essentials of authentication protocols, decipher the disparate terminology applied to the subject, tell you how to get started with Azure AD, and then present concrete examples of applications that use Azure AD for their authentication and authorization, including how they work in hybrid scenarios with Active Directory Federation Services (ADFS). With the information and insights Vittorio shares, you'll be able to efficiently create modern cloud applications that give users and administrators the flexibility and security of Microsoft's cloud and the convenience of using their preferred identities.

Mark Russinovich  
Chief Technology Officer, Microsoft Azure

# Introduction

It's never a good idea to use the word "modern" in the title of a book.

Growing up, one of the centerpieces of my family's bookshelf was a 15-tomes-strong encyclopedia titled *Nuovissima Enciclopedia* (Very new encyclopedia), and I always had a hard time reconciling the title with the fact that it was 10 years older than me.

I guarantee that the content in this book will get old faster than those old volumes—cloud and development technologies evolve at a crazy pace—and yet I could not resist referring to the main subject of the book as "modern authentication."

The practices and technologies used to take care of authentication in business solutions have changed radically nearly overnight, by a perfect storm of companies moving their assets to the cloud, software vendors starting to sell their products via subscriptions, the explosive growth of social networks with the nascent awareness of consumers of their own digital identity, ubiquitous APIs offering programmatic access to everything, and the astonishing adoption rate of Internet-connected smartphones.

"Modern authentication" is a catch-all term meant to capture how today's practices address challenges differently from their recent ancestors: JSON instead of XML, REST instead of SOAP, user consent and individual freedom alongside traditional admin-only processes, an emphasis on APIs and delegated access, explicit representation of clients, and so on. And if it is true that those practices will eventually stop appearing to be new—they are already mainstream at this point—the break with traditional approaches is so significant that I feel it's important to signal it with a strong title, even if your kids make fun of it a few years from now.

As the landscape evolves, Active Directory evolves with it. When Microsoft itself introduced one of the most important SaaS products on the planet, Office 365, it felt firsthand how cloud-based workloads call for new ways of managing user access and application portfolios. To confront that challenge Microsoft developed Azure Active Directory (Azure AD), a reimagined Active Directory that takes advantage of all the new protocols, artifacts, and practices that I've grouped under the modern authentication umbrella. Once it was clear that Azure AD was a Good Thing, it went on to become the main authentication service for all of Microsoft's cloud services, including Intune, Power BI, and Azure itself. But the real *raison d'être* of this book is that Microsoft opened Azure AD to every developer and organization so that it could be used for obtaining tokens to invoke Microsoft APIs and to handle authentication for your own web applications and web APIs.

*Modern Authentication with Azure Active Directory for Web Applications* is an in-depth exploration of modern authentication protocols and techniques used to implement sign-on for web applications and to protect web API calls. Although the protocols and pattern descriptions are applicable to any platform, my focus is on how Azure AD, the latest version of Active Directory Federation Services (ADFS), and the OpenID Connect and OAuth2 components in ASP.NET implement those approaches to handle authentication in real applications.

The text is meant to help you achieve expert-level understanding of the protocols and technologies involved in implementing modern authentication for a web app. Substantial space is reserved for architectural pattern descriptions, protocol considerations, and other abstract concerns that are necessary for correctly contextualizing the more hands-on advice that follows.

Most of the practical content in this book is about cloud and hybrid scenarios addressed via Azure AD. At the time of writing, the version of ADFS supporting modern authentication for web apps is still in technical preview; however, on-premises-only scenarios are covered whenever the relevant features are already available in the preview.

## Who should read this book

---

I wrote this book to fill a void of expert-level content for modern authentication, Azure AD, and ADFS. Microsoft offers great online quick starts, samples, and reference documentation—check out <http://aka.ms/aaddev>—that are perfect for helping you fulfil the most common tasks as easily as possible. That content covers many scenarios and addresses the needs of the vast majority of developers, who can be extremely successful with their apps without ever knowing what actually goes on the wire, or why. I like to think of that level of operation as the automatic mode for handheld and smartphone cameras—their defaults work great for nearly everybody, nearly all the time. But what happens if you want to take a picture of a lunar eclipse or any other challenging subject? That's when the point-and-click facade is no longer sufficient and knowing about aperture and exposure times becomes important. You can think of this book as a handbook for when you want to switch from automatic to manual settings. Doing so is useful for developers who work on solutions for which authentication requirements depart from the norm and for the devops who run such solutions.

Developers who worked with Windows Identity Foundation will find the text useful for transferring their skills to the new platform, and they'll pick up some new tricks along the way. The coverage of how the OWIN middleware works is deeper than

anything I've found on the Internet at this time: if you are interested in an in-depth case study of ASP.NET's Katana libraries, you'll find one here.

This book also comes in handy for security experts coming from a classic background and looking to understand modern protocols and approaches to authentication—the principles and protocols I describe can be applied well beyond Active Directory and ASP.NET. Security architects considering Azure AD for their solutions can use this book to understand how Azure AD operates. Protocol experts who want to understand how Azure AD and ADFS use OpenID Connect and OAuth2 will find plenty to mull over as well.

## Assumptions

This book is for senior professionals well versed in development, distributed architectures, and web-based solutions. You need to be familiar with HTTP trappings and have at least a basic understanding of networking concepts. All sample code is presented in C#, and all walk-throughs are based on Visual Studio. Azure AD and ADFS can be made use of from any programming stack and operating system; however, if you don't understand C# syntax and basic constructs (LINQ, etc.), it will be difficult for you to apply the coding advice in this book to your platform of choice. For good background, I'd recommend John Sharp's *Microsoft Visual C# Step by Step, Eighth Edition* (Microsoft Press, 2015).

Above all, this book assumes that you are strongly motivated to become an expert in modern authentication techniques and Azure AD development. The text does not take any shortcuts: you should not expect a light read; most chapters require significant focus and time investment.

## This book might not be for you if...

---

This book might not be for you if you just want to learn how to use Azure AD or ADFS for common development tasks. You don't have to buy a book for that: the documentation and the samples available at <http://aka.ms/aaddev> will get you up and running in no time, thanks to crisp step-by-step instructions. If there are tasks you'd like to see covered by the Azure AD docs, please use the feedback tools provided at that address: the Azure AD team is always looking for feedback for improving its documentation.

This book is also not especially good as a lookup reference. The text covers a lot of ground, including information that isn't included in the documentation at this



time, but the information is unveiled progressively, building on the reader's growing understanding of the topic. The book is optimized as a long lesson, not for looking things up.

Finally, this book won't be of much help if you are developing mobile, native, and rich-client applications. I originally intended to cover those types of applications, too, but the size of the book would have nearly doubled, so I had to cut them from this edition.

## Organization of this book

---

This book is meant to be read cover to cover. That's not what most people like to do, I know: bite-size and independent modules is the way to go today. I believe there are media more conducive to that approach, like video courses or the online documentation at <http://aka.ms/aaddev>. I chose to write a book because to achieve my goal—helping you understand modern authentication principles and how to take advantage of them with Azure AD—I cannot feed you only factlets and recipes. I have to present you with a significant amount of information, highlight relationships and implications for you, and then often ask you to tuck that knowledge away for a chapter or two before you actually end up using it. That's where I believe a book can still deliver value: by giving me the chance to hold your attention for a significant amount of time, I can afford a depth and breadth that I cannot achieve in a blog post. (By the way, did I mention that I do blog a lot as well? See [www.cloudidentity.com](http://www.cloudidentity.com) and [www.twitter.com/vibronet](http://www.twitter.com/vibronet).)

If this book has a natural fault line in its organization, it lies between the first four chapters and the last six. The first group provides context, and the later chapters dive deeply into the protocols, code, libraries, and features of Active Directory. Here's a quick description of each chapter's focus:

- Chapter 1, "Your first Active Directory app," is a soft introduction to the topic, giving you a brief glimpse of what you can achieve with Azure AD. It mostly provides instructions on how to use Visual Studio tools to create a web app that's integrated with Azure AD. Instant gratification.
- Chapter 2, "Identity protocols and application types," is a detailed history of identity protocols. It introduces terminology, topologies, and relationships between standards and helps you understand how modern authentication came to be and why identity is managed the way it is today.

- Chapter 3, “Introducing Azure Active Directory and Active Directory Federation Services,” presents basic concepts, terminology, and a list of developer-relevant features for Azure AD and ADFS. The hands-on chapters (Chapters 6-10) provide detailed descriptions of the features of both services that come into play in the scenarios of interest for the book.
- Chapter 4, “Introducing the identity developer libraries,” covers basic concepts, terminology, and the features of the Active Directory Authentication Library (ADAL) and ASP.NET OWIN middleware.
- Chapter 5, “Getting started with web sign-on and Active Directory,” provides a walk-through of how to create from scratch a web app that can sign in with Azure AD. Starting with the vanilla MVC templates, you learn about the NuGets packages you need to add, what app provisioning steps you need to follow in the Azure portal, and what code you need to write to perform key authentication tasks.
- Chapter 6, “OpenID Connect and Azure AD web sign-on,” provides a very detailed description of OpenID Connect and related standards, grounded on network traces of the actual traffic generated by the sample app. This is a very practical way of understanding the underlying protocol and why it operates the way it does. The descriptions of the constellation of ancillary specifications for OpenID Connect and OAuth2 will help you to navigate this rather crowded space, even if you are not planning to use Azure AD at the moment.
- Chapter 7, “The OWIN OpenID Connect middleware,” is a detailed analysis of how the authentication pipeline in ASP.NET works—with an emphasis on the OpenID Connect middleware, its extensibility points, and what scenarios these are meant to address.
- Chapter 8, “Azure Active Directory application model,” is a deep dive into the Azure AD application model: how Azure AD represents apps and handles consent, and how it deals with app provisioning, multitenancy, app roles, groups, app permissions, and the like.
- Chapter 9, “Consuming and exposing a web API protected by Azure Active Directory,” does for web APIs what Chapters 6 and 7 do for web apps—it explains the protocol flows used by web apps for gaining access to a protected API and describes how to use ADAL and the OAuth2 middleware for securely invoking and protecting a web API. This chapter also briefly introduces the Directory Graph API and discusses advanced scenarios such as exposing and securing both the user experience and an API from the same web project.

- Chapter 10, “Active Directory Federation Services in Windows Server 2016 Technical Preview 3,” discusses the new modern authentication features in ADFS, showing how to adapt web sign-on, web API invocation, and code protection covered in the earlier chapters to on-premises-only scenarios.
- The appendix, “Further reading,” provides you with pointers to online content describing ancillary topics and offerings that are still too new to be fully fleshed out in the book but are interesting and relevant to the subject of modern authentication.

## Finding your best starting point in this book

As I mentioned, every chapter in this book builds on the knowledge you acquire in the preceding ones. That makes choosing an arbitrary starting point a tricky exercise. I recommend that you look over the description of the book’s chapters in the previous section and decide whether you feel comfortable enough on the matter to choose a specific starting point.

## System requirements

---

You will need the following software if you want to follow the code walk-throughs in this book:

- Any Windows version that can run Visual Studio 2015 or later.
- Visual Studio 2015, any edition (technically, apart from Chapter 1, Visual Studio 2015 isn’t a hard requirement; Visual Studio 2013 will work with just a few adjustments).
- A Microsoft Azure subscription and access to the Azure portal.
- Telerik Fiddler v4 (<http://www.telerik.com/fiddler>).
- Internet connection to reach Azure AD during authentication operations and provisioning tasks.

In addition, Chapter 10 requires you to have access to an ADFS instance using Windows Server 2016 Technical Preview 3. Its system requirements can be found at <https://technet.microsoft.com/en-us/library/mt126134.aspx>. For the book, I hosted my own instance in a Hyper-V virtual machine, running on a laptop with Windows 10.

## Downloads: Code samples

---

This book contains a lot of code, and I present some of it in the form of guided walk-throughs. The goal is always to unveil the concepts you need to understand in manageable chunks, as opposed to the classic recipes you get in traditional labs or exercises. Also, I often discuss alternatives in the text, but the code can't always reflect all possible options. Expect the code to demonstrate the mainline approach; where possible and appropriate, alternatives are provided in code comments.

You can find the code I use in the book on my GitHub, at the following address:

*<http://aka.ms/modauth/files>*

You will notice a number of repositories with the form <ModAuthBook\_ChapterN>, where *N* represents the chapter number in which the repository code is described and demonstrated. (Not every book chapter contains code; only the chapters that do have a corresponding repository on GitHub.) If you are not familiar with GitHub, just click the repository name for the chapter you are interested in; somewhere on the page (at this time, at the bottom-right corner of the layout), you'll find the Download ZIP button, which you can use to save a local copy of the code.

## Using the code samples

Every repository contains a Visual Studio solution and a readme file. The readme provides a quick indication about the topic covered by the corresponding chapter, prerequisites, and basic instructions on how to provision the sample in your own Azure AD tenant. I'll do my best to keep the setup instructions up to date.

Once again, don't expect too much handholding: the code is provided mostly for reference. (Microsoft's official step-by-step samples and quick starts are provided at *<http://aka.ms/aaddev>*.) If a sample requires extra steps to fully demonstrate a scenario (for example, the presence in your tenant of an admin and at least a nonadmin user), I've assumed that you'll get that information by reading the book and don't repeat it in the sample's readme. The code provided at *<http://aka.ms/modauth/files>* is meant to support and complement the reading of the book rather than as a standalone asset.

## Acknowledgments

*We have to go deeper.*

—Cobb in *Inception*, a film by Christopher Nolan, 2010

This book has been a labor of love, written during nights, weekends, and occasional time off. I have willingly put that yolk on myself, but my wife, **Iwona Bialynicka-Birula**, did not . . . she endured nearly one year of missed hikes, social jet lag, and a silence curfew “because I have to write.” Thank you for your patience, darling—as I promised in the acknowledgments for *Programming Windows Identity Foundation* back in 2011: No more books for a few years!

This book would not have happened at all if **Devon Musgrave**, my acquisitions editor, would not have relentlessly pursued it, granting me a level of trust and freedom I am not sure I fully deserve. Thank you, Devon!

**John Pierce** has been an absolutely incredible project editor, driving everything from editing to project management to illustrations. He has this magic ability of turning my broken English into correct sentences while preserving my original intent. I wish every technical writer would have the good fortune of working with somebody as gifted as John. **Rob Nance** and **Carrie Wicks** also made significant contributions to producing this book.

I will be forever grateful to **Mark Russinovich** for the fantastic foreword he wrote for the book and for the kind words he offered about me. I am truly humbled to have my book begin with the words of a legend in software engineering.

Big thanks to my management chain for supporting this side project. **Alex Simons**, **Eric Doerr**, **Stuart Kwan**—thank you! I never quite managed to write on Fridays, but it was a great attempt.

I need to call out Stuart for a special thanks—from welcoming me to the product team to mentoring me through the transition from evangelism to product management. A large part of whatever success I have achieved is thanks to our work together. Thank you!

**Rich Randall**, the development lead on the Azure AD developer experience team, is my partner in crime and recipient of my utmost respect and admiration. Without his amazing work, none of the libraries described in this book would be around. And without the contribution of **Afshin Sephetri**, **Kanishk Panwar**, **Brent Schmaltz**, **Tushar Gupta**, **Wei Jia**, **Sasha Tokarev**, **Ryan Pangrle**, **Chris Chartier**, and

**Omer Cansizoglu**—developers on Rich’s team—those libraries would not be nearly as usable and powerful as they are.

**Danny Strockis** has been on the PM team for a relatively short time, but his contributions are already monumental. **Ariel Gordon**, responsible for designing many of the experiences that the Azure AD users go through every day, is a source of never-ending insights. **Dushyant Gill** drove the authorization features in Azure AD, and he patiently explained those to me every single time I barged into his office.

**Igor Sakhnov**, developer manager for Azure AD authentication, and his then-PM counterpart **David Howell** have my gratitude for trusting us on the decision to move the web authentication stack to OWIN. It worked out pretty well!

Speaking of OWIN. **Chris Ross, Tushar Gupta, Brent Schmaltz, Daniel Roth, Louis Dejardin, Eilon Lipton, and Barry Dorrans** all did a fantastic job, both in developing and driving the libraries and in handling my mercurial outbursts. Dan, I told you we’d get there! Special thanks to Chris Ross and Tushar Gupta for reviewing Chapter 7 in record time.

I started working with **Scott Hunter** on ASP.NET tooling and templates back in 2012 and loved every second. The man cares deeply about customers, understands the importance of identity, and is a force to reckon with. It is thanks to him and to my good friends **Pranav Rastogi, Brady Gaster, and Dan Roth** that web apps in Visual Studio can be enabled for Azure AD in just a few clicks.

In my opinion, Visual Studio 2015 has the most sophisticated identity management features in all of Microsoft’s rich clients, and that’s largely thanks to the relentless work that **Anthony Cangialosi, Ji Eun Kwon**, and all the Visual Studio and Visual Studio Online gang poured into it. That made it possible for many other teams to build on that core and deliver first-class identity support in Visual Studio for Azure, Office 365, and more. Among others, we have **Chakkaradeep (Chaks) Chinnakonda Chandran, Dan Seefeldt, Steve Harter, Xiaoying Guo, Yuval Mazor, Sean Laberee, and Paul Yuknewicz** to thank for that.

The Azure AD authentication service is for developers and maintained by some of the finest developers I know—**Shiung Yong, Ravi Sharma, Matt Rimer, and Maxim Yaryn** are the ones patiently fielding my questions and listening to my crazy scenarios. The architects behind the service, **Yordan Rouskov** and **Murli Satagopan**, are an inexhaustible source of insight.

The guys working on the directory data model, portal, and Graph API are also amazing in all sorts of ways: **Dan Kershaw, Edward Wu, Yi Li, Dmitry Pugachev,**

**Vijay Srirangam, Jeff Staiman, and Shane Oatman** are always there to help. Special mention to **Yi Li** who reviewed Chapter 8 and deals with my questions nearly every day.

Besides doing a fantastic job with ADFS in Windows Server 2016, **Samuel Devashayam, Mahesh Unnikrishnan, Jen Field, Jim Uphaus, and Saket Kataruka** from the ADFS team were of great help for Chapter 10.

The people on the partner teams are the ones who keep things real: they won't be satisfied until the services and libraries address their scenarios, and in so doing they push the services to excellence. **Mat Velloso** from Evangelism; **Rob Howard, Matthias Leibmann, Yina Arenas, and Tim McConnell** from Office 365; **Shriram Natarajan (Shri)** and **Pavel Tsurbelevu** from Azure Stack; **Dave Brankin, David Messner, Yugang Wang,** and **George Moore** from Azure; and **Hadeel Elbitar** from Power BI are all people who keep asking the right questions and offer priceless insights. Thank you guys!

The contribution from people in the development community is of paramount importance, especially now that our libraries are open source. **Dominick Baier** and **Brock Allen** are the most prominent sources of insight I can think of and are a beacon in the world of claims-based identity and modern authentication.

The identirati community plays a key role in moving modern authentication forward, divining what the industry wants and translating it into the form of RFC stone tablets. I am super grateful to **John Bradley** for our beer-fueled chats every time we meet at the Cloud Identity Summit and to the excellent **Brian Campbell** and, well, Canadian **Paul Madsen** for the friendly banter; to **Bob Blakley** and **Ian Glazer** for never failing to inspire; and to our own **Mike Jones** and **Anthony Nadalin** for being dependable, in-house protocol oracles. Although I cannot stop myself from reminding Tony that it is imperative that he work on his focus—he'll know what that means.

Last but not least, I want to thank the readers of my blog, my Twitter followers, the people I engage with on StackOverflow, and the people I meet at conferences during my sessions and afterward. It is your passion, your desire to know more and be more effective, and, yes, your affection, that made me decide to invest time in writing this book. Thank you for your incredible energy. *This book is for you.*

## Errata, updates, & book support

---

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

<http://aka.ms/modauth>

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at [mspinput@microsoft.com](mailto:mspinput@microsoft.com).

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to <http://support.microsoft.com>.

## Free ebooks from Microsoft Press

---

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

<http://aka.ms/mspressfree>

Check back often to see what is new!

## 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://aka.ms/tellpress>

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

## Stay in touch

---

Let's keep the conversation going! We're on Twitter:

<http://twitter.com/MicrosoftPress>





# The OWIN OpenID Connect middleware

In this chapter I focus on the OpenID Connect middleware and supporting classes. These are the cornerstones of ASP.NET's support for web sign-on.

As you saw in Chapter 5, "Getting started with web sign-on and Active Directory," in the most common case, the OpenID Connect middleware requires very few parameters to enable web sign-on. Beneath that simple surface, however, are knobs for practically anything you want to control: protocol parameters, initialization strategies, token validation, message processing, and so on. This chapter will reveal the various layers of the object model for you, showing how you can fine-tune the authentication process to meet your needs.

## OWIN and Katana

---

When I wrote *Programming Windows Identity Foundation* (Microsoft Press) in 2009, I didn't have to spend much time explaining `HttpContext`, the well-established ASP.NET extensibility technology on which WIF was built. This time around, however, I cannot afford the luxury of assuming that you are already familiar with OWIN and its implementation in ASP.NET—this is the foundational technology of the new generation of authentication libraries.

OWIN is a stable standard at this point, but its implementations are still relatively new technologies. You can find plenty of information online, but the details are rapidly changing (as I write, ASP.NET vNext is in the process of renaming lots of classes and properties), and you need to have a solid understanding of the pipeline and model underlying the identity functionality.

In this section I provide a quick tour of OWIN (as implemented in Katana 3.0.1) and the features that are especially relevant for the scenarios I've described throughout the book. For more details, you can refer to the online documentation from the ASP.NET team.

## What is OWIN?

OWIN stands for Open Web Interface for .NET. It is a community-driven specification: Microsoft is just a contributor, albeit a very important one. Here's the official definition, straight from the specifications' home page at <http://owin.org/>.

*OWIN defines a standard interface between .NET web servers and web applications. The goal of the OWIN interface is to decouple server and application, encourage the development of simple modules for .NET web development, and, by being an open standard, stimulate the open source ecosystem of .NET web development tools.*

In essence, OWIN suggests a way of building software modules (called *middlewares*) that can process HTTP requests and responses. It also describes a way in which those modules can be concatenated in a processing pipeline and defines how that pipeline can be hosted without relying on any specific web server or host or the features of a particular development stack.

The core idea is that, at every instant, the state of an HTTP transaction and the server-side processing of it is represented by a dictionary of the following form:

```
IDictionary<string, object>
```

This is commonly known as the *environment dictionary*. You can expect to find in it the usual request and response data (host, headers, query string, URI scheme, and so on) alongside any data that might be required for whatever processing an app needs to perform. Where does the data come from? Some of it, like the request details, must eventually come from the web server. The rest is the result of the work of the middleware in the pipeline.

Oversimplifying, a middleware is a module that implements the following interface:

```
Func<IDictionary<string, object>, Task>;
```

I am sure you have already guessed how things might work. The middleware receives the environment dictionary as input, acts on it to perform the middleware's function, and then hands it over to the next middleware in the pipeline. For example, logging middleware might read the dictionary and pass it along unmodified, but an authentication middleware might find a 401 code in the dictionary and decide to transform it into a 302, modifying the response to include an authentication request. By using the dictionary as the way of communicating and sharing context, as opposed to calling each other directly, middlewares achieve a level of decoupling that was not possible in older approaches.

How do you bootstrap all this? At startup, the middleware pipeline needs to be constructed and initialized: you need to decide what middlewares should be included and in which order and ensure that requests and responses will be routed through the pipeline accordingly. The OWIN specification has a section that defines a generic mechanism for this, but given that you will be working mostly with the ASP.NET-specific implementation, I won't go into much detail on that.

I skipped an awful lot of what the formulaic descriptions of OWIN normally include (like the formal definitions of application, middleware, server, and host), but I believe that this brief description should provide you enough scaffolding for understanding Katana, ASP.NET's implementation of OWIN.

## Katana

Katana is the code name for a set of Microsoft's .NET 4.5–based components that utilize the OWIN specification to implement various functionalities in ASP.NET 4.6. It's what you used in Chapter 1 and Chapter 5 and includes base middleware classes, a framework for initializing the pipeline, pipeline hosts for ASP.NET, and a large collection of middlewares for all sorts of tasks.

### Katana != OWIN

OWIN is an abstract specification. Katana is a set of concrete classes that implement that spec, but it also introduces its own implementation choices for tasks that aren't fully specified or in scope for the OWIN spec. In giving technical guidance, it's easy to say something to the effect "in OWIN, you do X," but it is often more proper to say "in Katana, you do X." I am sure I will be guilty of this multiple times: please accept my blanket apologies in advance.

In Chapter 5 you encountered most of the Katana NuGet packages and assemblies that appear in common scenarios. You also successfully used them by entering the code I suggested. Here I'll reexamine all that, explaining what really happens.

### *Startup and IAppBuilder*

In Chapter 5, in the section "Host the OWIN pipeline," you created a `Startup` class and decorated its source file with the `assembly:OwinStartup` attribute. The function of `Startup` is to initialize the OWIN pipeline by having its `Configure` method automatically invoked at initialization. To follow current practices, I instructed you to make `Startup` partial and to put the actual pipeline-building code in another file—but you could have just as well added the code in line in `Startup`.

Using the attribute is only one of several ways of telling Katana which class should act as `Startup`. You can also do the following:

- Have one class named `Startup` in the assembly or the global namespace.
- Use the `OwinStartup` attribute. The attribute wins against the naming convention (using `Startup`): if both techniques are used, the attribute will be the one driving the behavior.
- Add an entry under `<appSettings>` in the app config file, of the form

```
<add key="owin:appStartup" value=" WebAppChapter5.Startup" />.
```

This entry wins over both the naming convention and the attribute.

Fun times! I've listed these alternatives so that you know where to look if your app appears to magically pick up code without an obvious reason. I used to feel like that all the time when I first started with Katana.

Let's now turn our attention to `Startup.Configure`. Observe the method's signature:

```
public void Configure(IApplicationBuilder app)
```

`IApplicationBuilder` is an interface designed to support the initialization of the application. It looks like this:

```
public interface IApplicationBuilder
{
    IDictionary<string, object> Properties { get; }

    object Build(Type returnType);
    IApplicationBuilder New();
    IApplicationBuilder Use(object middleware, params object[] args);
}
```

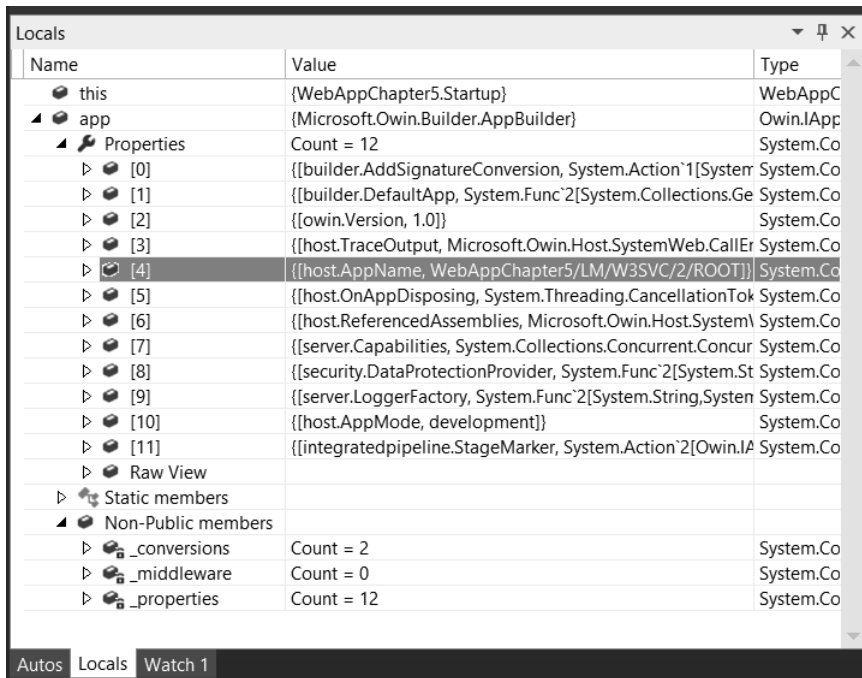
The `Properties` dictionary is used in turn by the server and host to record their capabilities and initialize data, making it available to the application's initialization code—that is to say, whatever you put in `Configure`. In the case of our sample app, the server is IIS Express and the host is the `System.Web` host we referenced when adding the NuGet package with the same name.



**Note** That host is actually an `HttpModule` designed to host the OWIN pipeline. That's the trick Katana uses to integrate with the traditional `System.Web` pipeline.

The `Build` method is rarely called in your own code, so I'll ignore it here. The `Use` method enables you to add middleware to the pipeline, and I'll get to that in a moment.

To prove to you that the host does indeed populate `app` at startup, let's take a peek at the `app` parameter when `Configure` is first invoked. Open Visual Studio, open the solution from Chapter 5, place a breakpoint on the first line of `Configure`, and hit F5. Once the breakpoint is reached, navigate to the Locals tab and look at the content of `app`. You should see something similar to Figure 7-1.



**FIGURE 7-1** The content of the app parameter at Configure time.

Wow, we didn't even start, and look at how much stuff is there already!

Katana provides a concrete type for `IAppBuilder`, named `AppBuilder`. As expected, the `Properties` dictionary arrives already populated with server, host, and environment properties. Feel free to ignore the actual values at this point. Just as an example, in Figure 7-1 I highlighted the `host.AppName` property holding the IIS metabase path for the app.

The nonpublic members hold a very interesting entry: `_middleware`. If you keep an eye on that entry as you go through the pipeline-initialization code in the next section, you will see the value of `Count` grow at every invocation of `Use*`.

## Middlewares, pipeline, and context

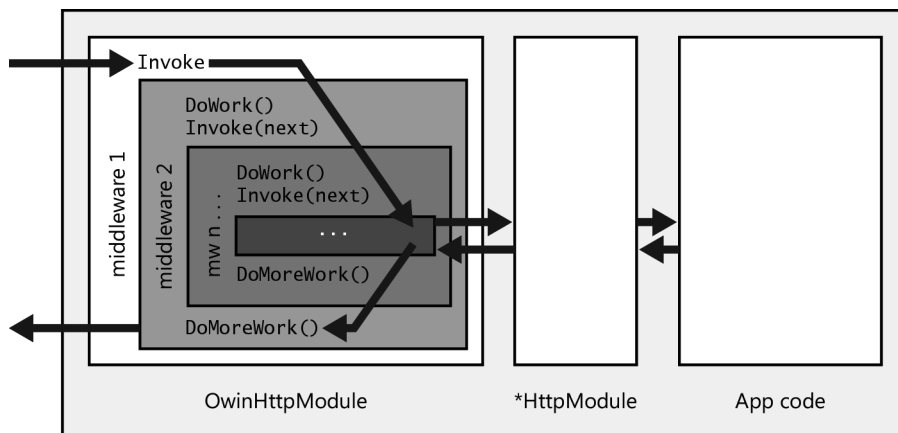
Stop the debugger and head to `Startup.Auth.cs`, where you will find the implementation of `ConfigureAuth`. This is where you actually add middleware to the pipeline, through the calls to `UseCookieAuthentication` and `UseOpenIdConnectAuthentication`. Those are convenience extension methods. `UseCookieAuthentication` is equivalent to this:

```
app.Use(typeof(CookieAuthenticationMiddleware), app, options);
```

The effect of `Use` is to add the corresponding middleware to the pipeline in `ApplicationBuilder`—as you can observe by watching the aforementioned `_middleware`. Although technically a middleware might simply satisfy the `Func` interface mentioned at the beginning, Katana offers patterns that are a bit more structured. One easy example can be found by examining `OwinMiddleware`, a base class for middlewares. It's super simple:

```
public abstract class OwinMiddleware
{
    protected OwinMiddleware(OwinMiddleware next)
    {
        Next = next;
    }
    protected OwinMiddleware Next { get; set; }
    public abstract Task Invoke(IOwinContext context);
}
```

Every middleware provides an `Invoke` method, accepting an `IOwinContext`, which is a convenience wrapper of the environment dictionary from the OWIN specs. In addition, every middleware can hold a pointer to the next entry in the pipeline. The idea is that when you call a middleware's `Invoke` method, the middleware can do its work on the context (typically, the request part of it), await the `Invoke` call of the next middleware in the pipeline, and do more work (this time on the response) once the `Invoke` method of the next middleware returns. As mentioned earlier, middlewares communicate via shared context: each middleware can examine the `IOwinContext` instance to find out what the preceding middleware did. You can see a diagram of this flow in Figure 7-2. The diagram is specific to the sample application scenario—hence IIS and the `System.Web` model—to make things as concrete as possible. However, I want to stress that the middleware activation sequence would remain the same even if it were hosted elsewhere.



**FIGURE 7-2** The OWIN pipeline as implemented by Katana in the sample application scenario: an `HttpModule`, hosting a cascade of middlewares.

Note that one middleware can always decide that no further processing should happen. In that case the middleware will not call the `Invoke` method of the next middleware in the sequence, effectively short-circuiting the pipeline.



**Note** `OwinMiddleware` is great for explaining the base functionality of the middleware, but in practice it raises interop issues. If you plan to build your own middleware (which is far outside the scope of this book), you should consider achieving the same behaviors without using it.

There's a neat trick you can use for observing firsthand how the middleware pipeline unfolds. You can interleave the `Use*` sequence with your own debug middlewares, and then place strategic breakpoints or debug messages. Here's the pattern for a minimal middleware-like debug implementation:

```
app.Use(async (Context, next) =>
{
    // request processing - do something here
    await next.Invoke();
    // response processing - do something here
});
```

That's pretty easy. Here's the sequence from the sample app, modified accordingly:

```
app.SetDefaultSignInAsAuthenticationType(CookieAuthenticationDefaults.AuthenticationType);
app.Use(async (Context, next) =>
{
    Debug.WriteLine("1 ==>request, before cookie auth");
    await next.Invoke();
    Debug.WriteLine("6 <==response, after cookie auth");
});

app.UseCookieAuthentication(new CookieAuthenticationOptions());

app.Use(async (Context, next) =>
{
    Debug.WriteLine("2 ==>after cookie, before OIDC");
    await next.Invoke();
    Debug.WriteLine("5 <==after OIDC");
});

app.UseOpenIdConnectAuthentication(
    new OpenIdConnectAuthenticationOptions
    {
        ClientId = "c3d5b1ad-ae77-49ac-8a86-dd39a2f91081",
        Authority = "https://login.microsoftonline.com/DeveloperTenant.onmicrosoft.com",
        PostLogoutRedirectUri = https://localhost:44300/
    }
);

app.Use(async (Context, next) =>
{
    Debug.WriteLine("3 ==>after OIDC, before leaving the pipeline");
    await next.Invoke();
    Debug.WriteLine("4 <==after entering the pipeline, before OIDC");
});
```



The numbers in front of every debug message express the sequence you should see when all the middlewares have a chance to fire. Any discontinuity in the sequence will tell you that some middleware decided to short-circuit the pipeline by not invoking its next middleware buddy.

Run the sample app and see whether everything works as expected. But before you do, you need to disable one Visual Studio feature that interferes with our experiment: it's the Browser Link. The Browser Link helps Visual Studio communicate with the browser running the app that's being debugged and allows it to respond to events. The unfortunate side effect for our scenario is that Browser Link produces extra traffic. In Chapter 6, "OpenID Connect and Azure AD web sign-on," we solved the issue by hiding the extra requests via Fiddler filters, but that's not an option here. Luckily, it's easy to opt out of the feature. Just add the following line to the `<appSettings>` section in the `web.config` file for the app:

```
<add key="vs:EnableBrowserLink" value="false"></add>
```

That done, hit F5. As the home page loads, the output window will show something like the following:

```
1 ==>request, before cookie auth
2 ==>after cookie, before OIDC
3 ==>after OIDC, before leaving the pipeline
'iisexpress.exe' (CLR v4.0.30319: /LM/W3SVC/2/ROOT-1-130799278910142565): Loaded 'C:\windows\assembly\GAC_MSIL\Microsoft.VisualStudio.Debugger.Runtime\14.0.0.0_b03f5f7f11d50a3a\Microsoft.VisualStudio.Debugger.Runtime.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
4 <==after entering the pipeline, before OIDC
5 <==after OIDC
6 <==response, after cookie auth
```

You can see that all the middlewares executed, and all in the order that was predicted when you assigned sequence numbers. Never mind that this doesn't appear to do anything! You'll find out more about that in the next section.

Click Contact or Sign In on the home page. Assuming that you are not already signed in, you should see pretty much the same sequence you've seen earlier (so I won't repeat the output window content here), but at the end of it your browser will redirect to Azure AD for authentication. Authenticate, and then take a look at the output window to see what happens as the browser returns to the app. You should see something like this:

```
1 ==>request, before cookie auth
2 ==>after cookie, before OIDC
5 <==after OIDC
6 <==response, after cookie auth
1 ==>request, before cookie auth
2 ==>after cookie, before OIDC
3 ==>after OIDC, before leaving the pipeline
4 <==after entering the pipeline, before OIDC
5 <==after OIDC
6 <==response, after cookie auth
```

This time you see a gap. As the request comes back with the token, notice that the first part of the sequence stops at the OpenID Connect middleware—the jump from 2 to 5 indicates that the last debug middleware was not executed, and presumably the same can be said for the rest of the following stages.

What happened? Recall what you studied in the section “Response” in Chapter 6: when the OpenID Connect middleware first receives the token, it does not grant access to the app right away. Rather, it sends back a 302 for honoring any internal redirect and performs a set-cookie operation for placing the session cookie in the browser. That’s exactly what happens in the steps 1, 2, 5, and 6: the OpenID Connect middleware decides that no further processing should take place and initiates the response sequence. The full 1–6 sequence that follows is what happens when the browser executes the 302 and comes back with a session cookie.

That’s it. At this point, you should have a good sense of how middlewares come together to form a single, coherent pipeline. The last generic building block you need to examine is the context that all middlewares use to communicate.

Sign out of the app and stop the debugger so that the next exploration will start from a clean slate.

### IIS integrated pipeline events and middleware execution

By now you know that Katana runs its middleware pipeline in an `HttpContext`, which participates in the usual IIS integrated pipeline. If you are familiar with that, you also know that `HttpContext`s can subscribe to multiple predefined events, such as `AuthenticateRequest`, `AuthorizeRequest`, and `PreExecuteRequestHandler`.

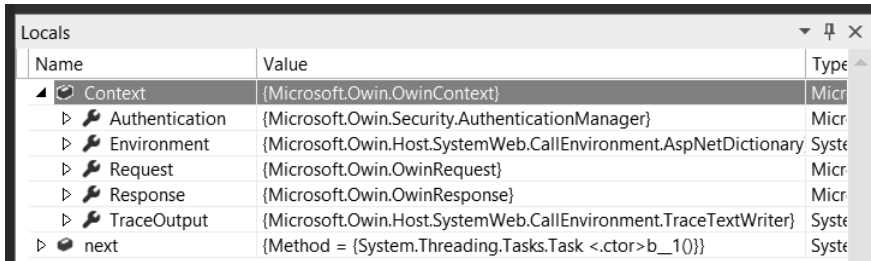
By default, Katana middleware executes during `PreExecuteRequestHandler`, although there are exceptions. There is a mechanism you can use for requesting execution of given segments of the middleware pipeline at a specific event in the IIS integrated pipeline, and that’s by using the extension method `UseStageMarker`.

Adding `app.UseStageMarker(PipelineStage.Authenticate)` tells Katana to execute in the `AuthenticateRequest` IIS event all the middlewares registered so far, or as far as the first preceding `UseStageMarker` directive.

This is not the whole story: for example, it’s possible to use stage markers for requesting sequences that are incompatible with the natural sequencing of events in the IIS pipeline. There are a number of rules that determine Katana’s behavior in those cases. Please refer to the ASP.NET documentation for details.

**Context** Before getting to the specifics of authentication, let’s invest a few moments to get to know the OWIN context better.

Place a breakpoint in the first diagnostic middleware, on the line that writes the message marked with 1. Hit F5, and once the execution reaches your breakpoint, head to the Locals tab and take a look at the content of the Context parameter. You should see what's depicted in Figure 7-3.

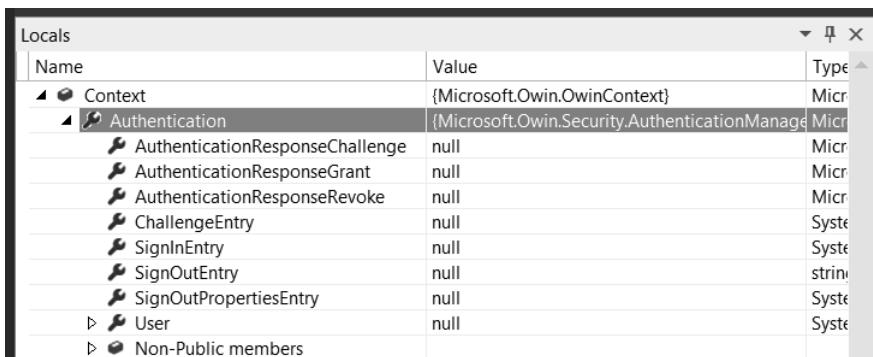


**FIGURE 7-3** The structure of the Katana context.

Let's cover each of the entries here.

- Authentication** The `Authentication` property is used for exposing authentication capabilities of the current pipeline. You saw this in action when you implemented the sign-in and sign-out features in Chapter 5, via the `Challenge` and `SignInOut` methods, respectively.

`Authentication` is also used by authentication middlewares for communicating with one another, as you will see in the next section. As Figure 7-4 shows, when the request first enters the pipeline, `Authentication` is empty. You will learn about this property in detail when we focus on the authentication middleware.



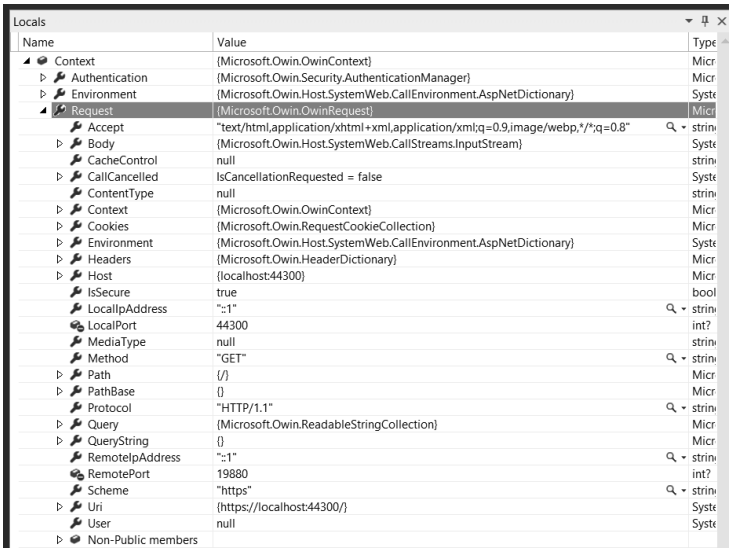
**FIGURE 7-4** The `Context.Authentication` property content upon receiving the first unauthenticated request.

- Environment** As the OWIN specification states, the core status of an OWIN pipeline is captured by the environment dictionary. Figure 7-5 shows how the Katana implementation features all the values prescribed by the OWIN specification, plus a few more.



**FIGURE 7-5** The content of the OWIN environment dictionary on first request.

- Request and Response** If you are familiar with HTTP request and response manipulation in traditional ASP.NET, you should be quite comfortable with their similarly named Context properties in Katana. Figure 7-6 shows an example.



**FIGURE 7-6** The first request, represented by the `Context.Request` property.

- **TraceOutput** This property is mainly a clever way of exposing a standard trace at the OWIN level, regardless of the specific host used to run the pipeline.

Add more breakpoints for the other debug middlewares and see how `Context` changes as the execution sweeps through the pipeline. After you have experimented with this, head to the next section, where I review the authentication flow through the OWIN pipeline in detail.

## Authentication middleware

The authentication functionality emerges from the collaboration of a protocol middleware (like those for OpenID Connect or WS-Federation) and the cookie middleware. The protocol middleware reacts to requests and responses by generating and processing protocol messages, with all that entails (token validation and so on). The cookie middleware persists sessions in the form of cookies at sign-in and enforces the presence and validity of such cookies from the instant of authentication onward. All communication between the two middlewares takes place via the `AuthenticationManager` instance in the `Context`. Let's break down the sign-in flow we captured earlier into three phases: generation of the sign-in challenge, response processing and session generation, and access in the context of a session.

**Sign-in redirect message** Assume that you triggered the sign-in flow by clicking `Contact`. As you observed, this action results in all the middlewares firing in the expected order, and it concludes with the redirection of the browser toward Azure AD with an authorization request message.

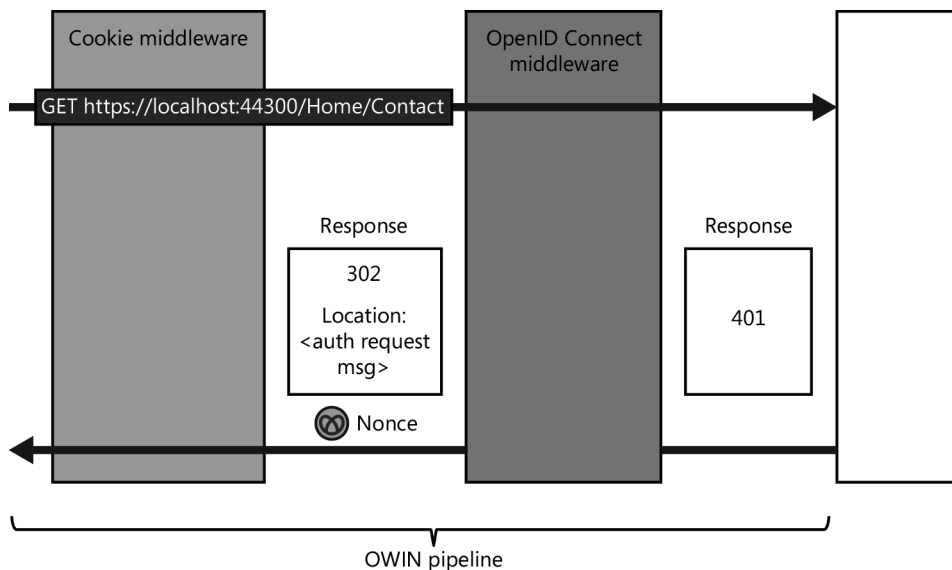
If you go through the flow while keeping an eye on `Context.Response`, you will notice that after the request leaves the OWIN pipeline (after the debug message marked 3), something changes the `Response.StatusCode` to 401. In this case, that was the good old `[Authorize]`, which does its job to enforce authenticated access regardless of the presence of the OWIN pipeline.

If you go beyond the breakpoint on debug message 4 and let the OpenID Connect middleware execute, you will observe that `Response.StatusCode` changes again, this time to 302. If you dig into the `Response.Headers` collection, you will notice a new entry, `Location`, containing the OpenID Connect authorization request. Moreover, you will find a new `Set-Cookie` entry for saving the OpenID Connect nonce.

Walking through the rest of the breakpoint, you will see the response message go unmodified through the remainder of the pipeline and back to the browser.

In Katana parlance, the OpenID Connect middleware is Active by default. That means that its options class's `AuthenticationMode` property is set to `Active`, which makes it react to 401 responses by generating its sign-in challenge message. That is not always what you want: for example, if you have multiple protocol middlewares configured to talk to different IdPs, you will want explicit control (via calls to `Authentication.Challenge`) over what middleware should be in charge to generate the sign-in message at a given moment.

Figure 7-7 displays the steps in the sequence of the sign-in message generation phase.



**FIGURE 7-7** The sequence through which an unauthenticated request elicits the generation of a sign-in message.

**Token validation and establishment of a session** The sequence that processes the Azure AD response carrying the token (characterized by the debug sequence 1, 2, 5, 6 earlier) is the one requiring the most sophisticated logic.

The request goes through the cookie middleware (breakpoints on messages 1 and 2) unmodified. However, as soon as you step over the `Invoke` call in the diagnostic middleware that calls the OpenID Connect middleware, you'll observe that the execution goes straight to the breakpoint on debug message 5, skipping the rest of the pipeline and the app itself and initiating the response.

Once again, the Response object carries a 302. If you recall the explanations in the earlier section, you know that this 302 means that the middleware successfully validated the token received and is now using a redirect operation to perform any local redirect and persist the session cookie in the client browser. If you take a look at the Response.Header collection, you will find a Location entry redirecting to "https://localhost:44300/Home/Contact", which is the route originally requested. You will also find a Set-Cookie entry meant to delete the nonce, which is no longer necessary at this point. However, you will not find any Set-Cookie for the session cookie. Where is it?

Saving the session is the job of the cookie middleware, which at this point has not yet had a chance to process the response. In fact, saving a session might be a far more complicated matter than simply sending back a Set-Cookie header. For example, you might want to save the bulk of the session on a server-side store: the cookie middleware provides that ability as a service so that any protocol middleware can leverage it without having to reinvent the process every time.

The OpenID Connect middleware uses Context.Authentication to communicate to the cookie middleware the content of the validated token to be persisted as well as other session-related details, such as duration. Right after the OpenID Connect middleware processes the request, you'll see the Authentication properties AuthenticationResponseGrant, SignInEntry, and User populated.

The cookie middleware is mostly interested in AuthenticationResponseGrant. When its turn comes to process the response, the cookie middleware will find the AuthenticationResponseGrant and use its content to generate a session. In Figure 7-8 you can see an example of AuthenticationResponseGrant content.

| Name  | Value  |
|---|--|
| (((Microsoft.Owin.OwinContext)Context).Authentication | {Microsoft.Owin.Security.AuthenticationManager}                |
| AuthenticationResponseChallenge                       | null   |
| AuthenticationResponseGrant                           | {Microsoft.Owin.Security.AuthenticationResponseGrant}          |
| Identity  | {System.Security.Claims.ClaimsIdentity}                        |
| Actor   | null   |
| AuthenticationType                                    | "Cookies"  |
| BootstrapContext                                      | null   |
| Claims  | {System.Security.Claims.ClaimsIdentity.<get_Claims>d__1}       |
| CustomSerializationData                               | null   |
| IsAuthenticated                                       | true   |
| Label   | null   |
| Name  | "mario@developertenant.onmicrosoft.com"                        |
| NameClaimType   | "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"   |
| RoleClaimType   | "http://schemas.microsoft.com/ws/2008/06/identity/claims/role" |
| Static members  |  |
| Non-Public members                                    |  |
| Principal   | {System.Security.Claims.ClaimsPrincipal}                       |
| Properties  | {Microsoft.Owin.Security.AuthenticationProperties}             |
| AllowRefresh  | false  |
| Dictionary  | Count = 6  |
| ExpiresUtc  | {7/19/2015 11:45:38 PM +00:00}                                 |
| IsPersistent  | false  |
| IssuedUtc   | {7/19/2015 10:40:38 PM +00:00}                                 |
| RedirectUri   | "https://localhost:44300/Home/Contact"                         |
| Static members  |  |
| Non-Public members                                    |  |

**FIGURE 7-8** The AuthenticationResponseGrant content right after the OpenID Connect middleware successfully validates a sign-in response from Azure AD.

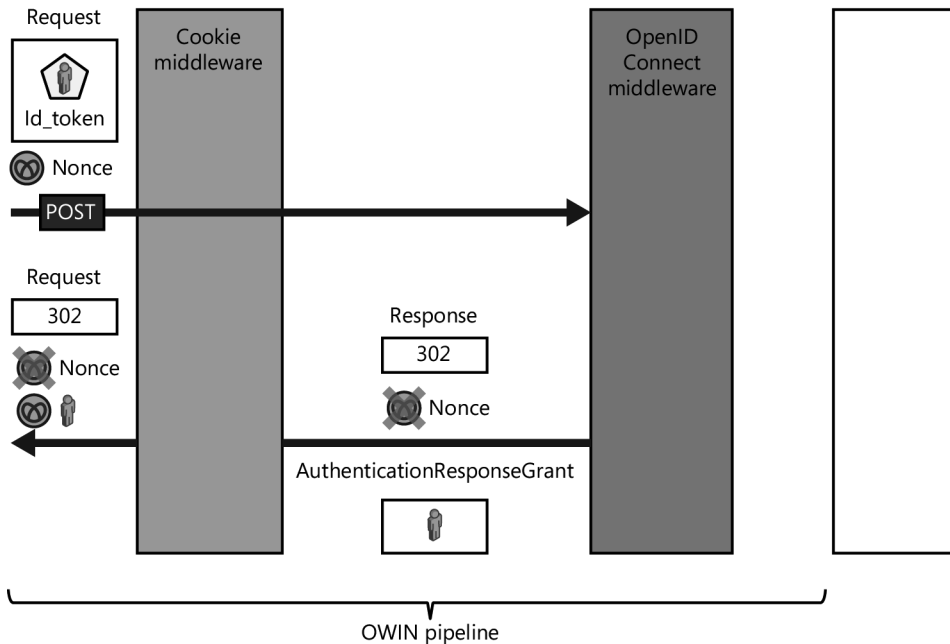
Properties refers to generic session properties, such as the validity window (derived from the validity window of the token itself, as declared by Azure AD). Identity, as you guessed, is the

ClaimsIdentity representing the authenticated user. The most important thing to notice at this point is the AuthenticationType value that's shown: that's a hint left by the OpenID Connect middleware for the cookie middleware, indicating that the ClaimsIdentity instance should be persisted in the session. Recall that when the pipeline is initialized in Startup.Auth.cs, you started the method with the following line:

```
app.SetDefaultSignInAsAuthenticationType(CookieAuthenticationDefaults.AuthenticationType);
```

That told the protocol middlewares in the pipeline that in the absence of local overrides, the identifier to use for electing an identity to be persisted in a session is CookieAuthenticationDefaults.AuthenticationType, which happens to be the string "Cookies". When the OpenID Connect middleware validates the incoming token and generates the corresponding ClaimsPrincipal and nested ClaimsIdentity, it uses that value for the AuthenticationType property. When the cookie middleware starts processing the response and finds that ClaimsIdentity, it verifies that the AuthenticationType it finds there corresponds to the AuthenticationType value it has in its options. Given that here we used the defaults everywhere, it's a match; hence, the cookie middleware proceeds to save the corresponding ClaimsPrincipal in the session.

If you examine the Response.Headers collection after the cookie middleware has a chance to execute, you will see that the Set-Cookie value now includes a new entry for an .AspNet.Cookies, which contains the ClaimsPrincipal information. Figure 7-9 summarizes the sequence.

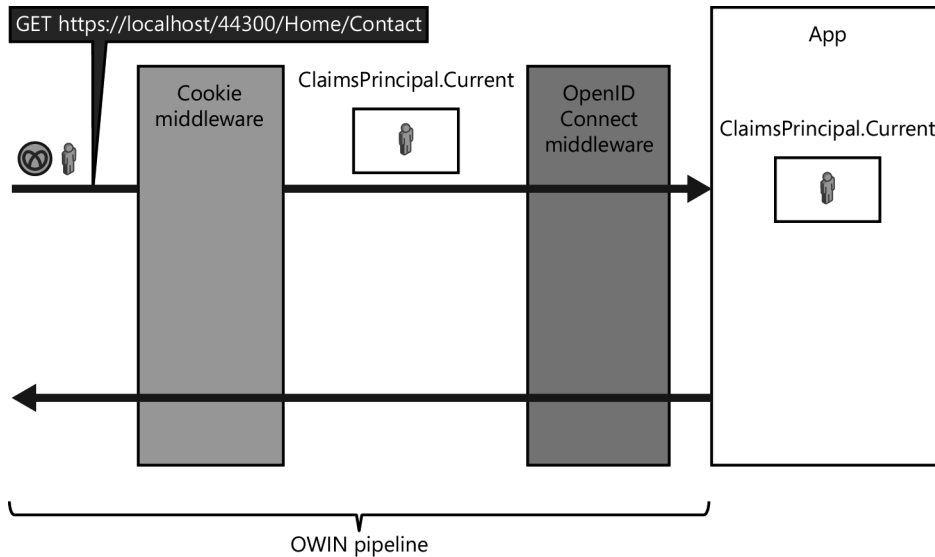


**FIGURE 7-9** The token-validation and session-creation sequence. The OpenID Connect middleware processes the incoming token, passing the user identity information it carries to the cookie middleware. In turn, the cookie middleware saves the user identity information in a session cookie.



**Authenticated access as part of a session** Once the session has been established, all requests within its validity window are handled in the same way: as soon as the request is processed by the cookie middleware (between debug messages marked with 1 and 2), the incoming cookie is retrieved, validated, and parsed. The `ClaimsPrincipal` it carries is rehydrated, as shown by the value of `Authentication.User` being populated with a `ClaimsPrincipal`; the rest of the pipeline just lets the message through without further processing.

Figure 7-10 shows how this all plays out through the middleware pipeline.



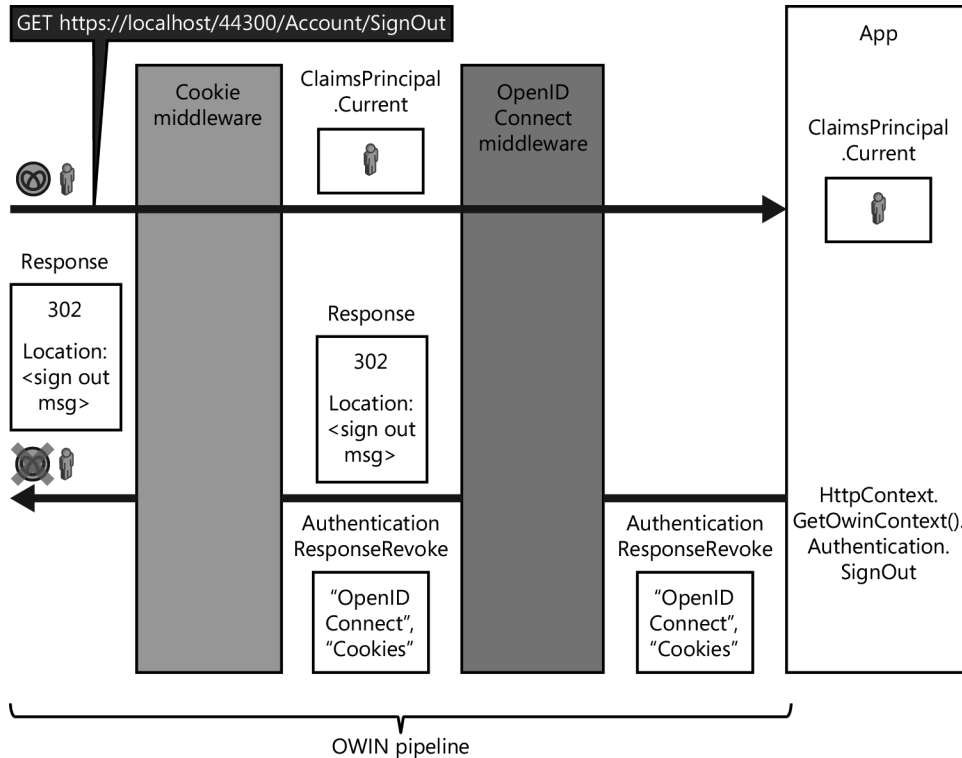
**FIGURE 7-10** During a session, every request carries the session token, which is validated, decrypted, and parsed by the cookie middleware. The user claims are made available to the application.

**Explicit use of *Challenge* and *SignOut*** The explicit sign-in and sign-out operations you implemented in the `AccountController` of the sample app also use the `Authentication` property of `Context` to communicate with the middleware in the pipeline.

If you want to see how `Challenge` works, repeat the sign-in flow as described earlier, but this time trigger it by clicking Sign In. Stop at the breakpoint on debug message 4. You will see that the response code is a 401, just like the case we examined earlier. However, here you will also see populated entries in `Authentication`, in particular `AuthenticationResponseChallenge`. If you peek into it, you'll see that `AuthenticationResponseChallenge` holds the `AuthenticationType` of the middleware you want to use for signing in ("OpenIdConnect") and the local redirect you want to perform after sign-in (in this case, the root of the app). If the OpenID Connect middleware is set to `Passive` for `AuthenticationMode`, the presence of the 401 response code alone is not enough to provoke the sign-in message generation, but the presence of `AuthenticationResponseChallenge` guarantees that it will kick in. Other than that, the rest of the flow goes precisely as described.

The sign-out flow is very similar. Hit the Sign Out link. Stopping at the usual breakpoint 4, you'll observe that `Authentication` now holds a populated `AuthenticationResponseRevoke`

property, which in turn contains a collection of `AuthenticationTypes`, including `"OpenIdConnect"` and `"Cookies"`. When it's their turn to process the response, the middlewares in the pipeline check whether there is a nonnull `AuthenticationResponseRevoke` entry containing their `AuthenticationTypes`. If they find one, they have to execute their sign-out logic. As you advance through the breakpoints in the response flow, you can see that behavior unfolding. The OpenID Connect middleware reacts by changing the return code to 302 and placing the sign-out message for Azure AD in the Location header. The cookie middleware simply adds a Set-Cookie entry that sets the session cookie expiration date to January 1, 1970, invalidating the session. Figure 7-11 provides a visual summary of the operation.



**FIGURE 7-11** The contributions to the sign-out sequence from each middleware in the pipeline.

## Diagnostic middleware

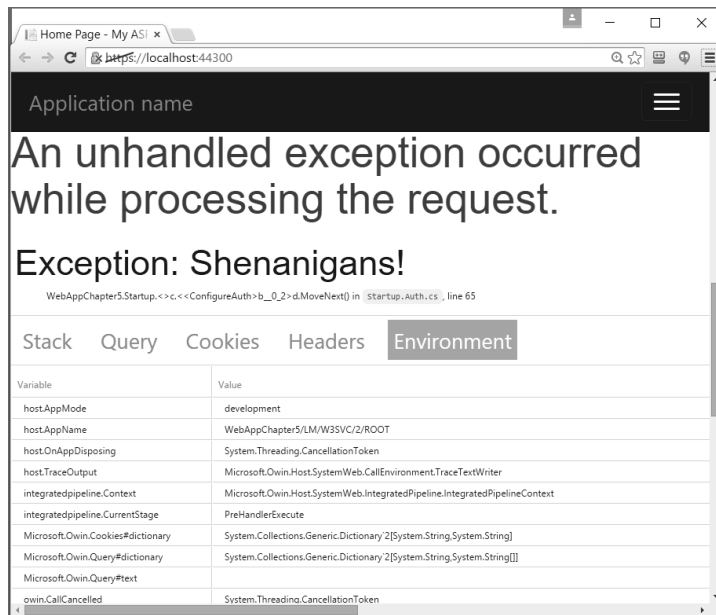
When something goes wrong in the OWIN pipeline, finding the culprit is often tricky. Adding breakpoints to an “in line” middleware, as I have done in this chapter to highlight how the pipeline works, is definitely an option. Alternatively, Katana offers a specialized diagnostic middleware that can render useful debugging information directly in the browser when an unhandled exception occurs in the pipeline. Setting it up is super easy.

Add a reference to the NuGet package `Microsoft.Owin.Diagnostics`. In your `Startup.Auth.cs`, add the associated `using` directive. Right on top of your main configuration routine (in our sample, `ConfigureAuth`), add something along the lines of the following:

```
app.UseErrorPage(new ErrorPageOptions()
{
    ShowCookies = true,
    ShowEnvironment = true,
    ShowQuery = true,
    ShowExceptionDetails = true,
    ShowHeaders = true,
    ShowSourceCode = true,
    SourceCodeLineCount = 10
});
```

The extension method `UseErrorPage` injects into the pipeline some logic for dumping diagnostic information on the current page in case an exception is raised in the pipeline. For that reason, it's important to place this method at the beginning of the pipeline (otherwise, any exceptions occurring before it has a chance to fire would not be captured). All the options you see in the call control what diagnostic information you want to display; the property names are self-explanatory.

If you want to test the contraption, you can artificially raise an exception in any of our debugging middlewares, and then hit F5 to see what happens. Figure 7-12 shows a typical diagnostic page.



**FIGURE 7-12** The page displayed by the diagnostic middleware from `Microsoft.Owin.Diagnostics`.



**Important** You should never use this middleware in production applications, as it might reveal information you don't want an attacker to obtain. Please use this only for debugging. Moreover, this middleware will not help in the case of exceptions raised in the application itself. It is really specialized for handling issues occurring in the OWIN pipeline.

## OpenID Connect middleware

---

With the exception of the cookie tracking the nonce, all the considerations so far apply to the OpenID Connect middleware as well as the WS-Federation middleware. In this section I dive deeper into the features and options of the OpenID Connect middleware.

### *OpenIdConnectAuthenticationOptions*

The options you pass in at initialization are the main way that you control the OpenID Connect middleware. The Azure AD and ASP.NET teams have taken a lot of care to ensure that only the absolute minimum amount of information is required for the scenario you want to support. The sample app you have studied so far shows the essential set of options: the `ClientId` (which identifies your app in your requests to the authority) and the `Authority` (which identifies the trusted source of identities and, indirectly, all the information necessary to validate the tokens it issues). If you want to exercise more fine-grained control, you can use the middleware initialization options class to provide the following:

- More protocol parameters that define your app and the provider you want to trust.
- What kind of token requests you want the app to put forth.
- What logic you want to execute during authentication, choosing from settings offered out of the box and from custom logic you want to inject.
- The usual array of choices controlling all Katana middleware mechanics.

In this section I describe the most notable categories. Two special properties, `Notifications` and `TokenValidationParameters`, are so important that I've dedicated sections to them.

For your reference, Figure 7-13 shows the default values in `OpenIdConnectAuthenticationOptions` for our app, right after initialization.

| Name                            | Value  |
|---------------------------------|--|
|                                 | {Microsoft.Owin.Security.OpenIdConnect.OpenIdConnectAuthenticationOptions}       |
| AuthenticationMode              | Active   |
| AuthenticationType              | "OpenIdConnect"  |
| Authority                       | "https://login.microsoftonline.com/DeveloperTenant.onmicrosoft.com"              |
| BackchannelCertificateValidator | null   |
| BackchannelHttpHandler          | null   |
| BackchannelTimeout              | {00:01:00}   |
| CallbackPath                    | {}   |
| Caption                         | "OpenIdConnect"  |
| ClientId                        | "c3d5b1ad-ae77-49ac-8a86-dd39a2f91081"   |
| ClientSecret                    | null   |
| Configuration                   | null   |
| ConfigurationManager            | null   |
| Description                     | {Microsoft.Owin.Security.AuthenticationDescription}                              |
| MetadataAddress                 | null   |
| Notifications                   | {Microsoft.Owin.Security.OpenIdConnect.OpenIdConnectAuthenticationNotifications} |
| PostLogoutRedirectUri           | "https://localhost:44300/"   |
| ProtocolValidator               | {Microsoft.IdentityModel.Protocols.OpenIdConnectProtocolValidator}               |
| RedirectUri                     | null   |
| RefreshOnIssuerKeyNotFound      | true   |
| Resource                        | null   |
| ResponseType                    | "code id_token"  |
| Scope                           | "openid profile"   |
| SecurityTokenHandlers           | null   |
| SignInAsAuthenticationType      | null   |
| StateDataFormat                 | null   |
| TokenValidationParameters       | {System.IdentityModel.Tokens.TokenValidationParameters}                          |
| UseTokenLifetime                | true   |

**FIGURE 7-13** The values in `OpenIdConnectAuthenticationOptions` after a typical initialization sequence.

## Application coordinates and request options

Besides the already-mentioned `ClientId`, you can supply the following application details.



**Note** Parameters in the options class corresponding to OpenID Connect protocol parameters have the same name, with the notation adjusted to match .NET naming conventions. In early iterations, the Active Directory team tried to use the protocol names verbatim—lowercase, underscore, and all—but the community staged an uprising, and the team quickly settled on the format you see today.

- RedirectUri** This controls the value of `redirect_uri` included in the request, corresponding to the route in your app through which you want Azure AD to return the requested token. As I noted in Chapter 6, if you don't specify any value, the parameter will be omitted and Azure AD will pick the one registered at registration time. That's handy, but you should watch out for two possible issues. First, you might register multiple `redirect_uri` values for your app, in which case Azure AD will choose which one to use in a semirandom fashion (it always looks like it chooses the first one you registered, but you cannot count on that). Second, if you are connecting to providers other than Azure AD, they might require the request to comply with their spec and include a `redirect_uri`.

This setting is ingested at the time the app is initialized and won't change later on. In the section about notifications, you will learn ways of overriding this and other parameters on the fly in the context of specific requests and responses.

- `PostLogoutRedirectUri` You have seen this in use in Chapter 5. It determines where to redirect the browser in your app once the authority concludes its sign-out operations.
- `ClientSecret` This represents the `client_secret`, which is required when redeeming an authorization code. I covered this at a high level in Chapter 2, in the context of OAuth2, but did not look at it at the trace and code level. I'll do so later in the book.

Here are a few other parameters that control what's going to be sent in the request.

- `ResponseType` Maps to the OpenID Connect parameter of the same name. Although you can assign to it any of the values discussed in Chapter 6, only "id\_token" and "code id\_token" (the default) lead to the automatic handling of user sign-in. If you want to support other response types, such as "code", you need to inject custom code in the notifications described later in this chapter.
- `Resource` In case you are using "code id\_token", you can use this parameter to specify what resource you want an authorization code for. If you don't specify anything, the code you get back from Azure AD will be redeemable for an access token for the Graph API. As mentioned in Chapter 6, `resource` is a parameter specific to Azure AD.
- `Scope` Maps to the OAuth2/OpenID Connect scope parameter.

Barring any custom code that modifies outgoing messages on the fly, the settings described here are the ones used in every request and response.

## Authority coordinates and validation

The functional area of validation is one of the toughest to explain. It was one of the main pain points of working with WIF, where the object model expected all validation coordinates to be passed by value. Although Microsoft provided tools that generated those settings automatically from metadata, the obscurity and sheer sprawl of the resulting configuration settings came across as a bogeyman that kept the noninitiated at bay.

In the new middlewares, the default behavior is to obtain (most of) the validation coordinates by reference. You provide the authority from which you want to receive tokens, and the middleware takes care of retrieving the token validation coordinates it needs from the authority's metadata.

In Chapter 6 you saw how that retrieval operation takes place when you pass an Azure AD authority. If you want to customize that behavior, there is a hierarchy of options you can use. From accommodating providers that expose metadata differently from how Azure AD does, to supplying each and every setting for providers that don't expose metadata at all, these options cover the full spectrum.

Here's how it works.

The `ConfigurationManager` class is tasked to retrieve, cache, and refresh the validation settings published by the discovery documents. That class is fed whatever options you provide at initialization. There is a cascade of options it looks for:

- If the options include an `Authority` value, it will be used as you saw in Chapter 6.
- If you are working with a provider other than Azure AD, with a different URL structure, or if you prefer to specify a reference to the actual discovery document endpoint, you can do so by using the `Metadata` property.
- If your provider requires special handling of the channel validation, like picking a well-known certificate instead of the usual certification authority and subject matching checks, you can override the default logic via the properties `BackchannelCertificateValidator`, `BackchannelHttpHandler`, and `BackchannelTimeout`.
- If you acquire the token-issuance information—such as the authorization endpoint, the issuer value, the signing keys, and the like—out of band, you can use it to populate a new instance of `OpenIdConfiguration` and assign it to the `Configuration` property.
- Finally, if you need to run dynamic logic for populating the `Configuration` values, you can completely take over by implementing your own `IConfigurationManager` and assigning it to the `ConfigurationManager` property in the options.

The issuer coordinates are only part of the validation story. Following is a miscellany of options that affect the validation behavior, and there will be more to say about validation in the section about `TokenValidationParameters`.

- `SecurityTokenHandlers` This property holds a collection of `TokenHandlers`, classes that are capable of handling token formats. By default, the collection includes a handler capable of dealing with the JSON Web Token (JWT). You can take control of the collection and substitute your own implementation if you so choose.
- `RefreshOnIssuerKeyNotFound` The practice of publishing in metadata documents both the currently valid and next signing key should guarantee business continuity in normal times. In case of emergency key rolls, however, the keys you have acquired in your `Configuration` and the ones used by the provider might end up out of sync. This flag tells the middleware to react to a token signed with an unknown key by triggering a new metadata acquisition operation so that if the mismatch is the result of stale keys, it is fixed automatically.
- `CallbackPath` If for some reason (typically performance) you decide that you want to receive tokens only at one specific application URL, you can assign that URL to this property. That will cause the middleware to expect tokens only in requests to that specific URL and ignore all others. Use this with care because embedding paths in your code often results in surprise 401s when you forget about them and deploy to the cloud without changing the value accordingly.
- `ProtocolValidator` By default, this property contains an instance of `OpenIdConnectProtocolValidator`, a class that performs various static verifications on the incoming message to ensure that it complies with the current OpenID Connect specification. Besides those validations, the class gives you the option of adding extra constraints, like mandating the presence of certain claim types.

## Middleware mechanics

Finally, here's a list of options that are used for driving the general behavior of the middleware in the context of the Katana pipeline:

- `SignInAsAuthenticationType` This value determines the value of the `AuthenticationType` property of the `ClaimsPrincipal/ClaimsIdentity` generated from the incoming token. If left unspecified, it defaults to the value passed to `SetDefaultSignInAsAuthenticationType`. As you have seen earlier in the section about authentication middleware, if the cookie middleware finds this in an `AuthenticationResponseGrant`, that's what the cookie middleware uses to determine whether such `ClaimsPrincipal/ClaimsIdentity` should be used for creating a session.
- `AuthenticationType` This property identifies this middleware in the pipeline and is used to refer to it for authentication operations—think of the `Challenge` and `SignInOut` calls you have seen in action earlier in this chapter.
- `AuthenticationMode` As discussed earlier, when this parameter is set to `Active`, it tells the middleware to listen to outgoing 401s and transform them into sign-in requests. That's the default behavior: if you want to change it, you can turn it off by setting `AuthenticationMode` to `Passive`.
- `UseTokenLifetime` This property is often overlooked, but it's tremendously important. Defaulting to `true`, `UseTokenLifetime` tells the cookie middleware that the session it creates should have the same duration and validity window as the `id_token` received from the authority. If you want to decouple the session validity window from the token (which, by the way, Azure AD sets to one hour), you must set this property to `false`. Failing that, all the session-duration settings on the `CookieMiddleware` will be ignored.
- `Caption` This property has purely cosmetic value. Say that your app generates sign-in buttons for all your authentication middlewares. This property provides the label you can use to identify for the user the button triggering the sign-in implemented by this middleware.

## Notifications

Just like WIF before them, the Katana middlewares implementing claims protocols offer you hooks designed for injecting your own custom code to be executed during key phases of the authentication pipeline. Through the years, I have seen this extensibility point used for achieving all sorts of customizations, from optimized sign-in flows, where extra information in the request is used to save the end user a few clicks, to full-blown extensions that support entirely new protocol flavors.

Whereas in old-school WIF those hooks were offered in the form of events, in Katana they are implemented as a collection of delegates gathered in the class `OpenIdConnectNotifications`. The `OpenIdConnectAuthenticationOptions` class includes a property of that type, `Notifications`.

`OpenIdConnectNotifications` can be split into two main categories: notifications firing at sign-in/sign-out message generation, and notifications firing at token/sign-in message validation. The for-



mer category counts only one member, RedirectToIdentityProvider; all the other notifications are included in the latter.

Here is some code that lists all the notifications. You can add it to the initialization of the OpenID Connect middleware in the sample application.

```
app.UseOpenIdConnectAuthentication(
    new OpenIdConnectAuthenticationOptions
    {
        ClientId = "c3d5b1ad-ae77-49ac-8a86-dd39a2f91081",
        Authority = "https://login.microsoftonline.com/DeveloperTenant.onmicrosoft.com"
        PostLogoutRedirectUri = "https://localhost:44300/",
        Notifications = new OpenIdConnectAuthenticationNotifications()
        {
            RedirectToIdentityProvider = (context) =>
            {
                Debug.WriteLine("**** RedirectToIdentityProvider");
                return Task.FromResult(0);
            },
            MessageReceived = (context) =>
            {
                Debug.WriteLine("**** MessageReceived");
                return Task.FromResult(0);
            },
            SecurityTokenReceived = (context) =>
            {
                Debug.WriteLine("**** SecurityTokenReceived");
                return Task.FromResult(0);
            },
            SecurityTokenValidated = (context) =>
            {
                Debug.WriteLine("**** SecurityTokenValidated");
                return Task.FromResult(0);
            },
            AuthorizationCodeReceived = (context) =>
            {
                Debug.WriteLine("**** AuthorizationCodeReceived");
                return Task.FromResult(0);
            },
            AuthenticationFailed = (context) =>
            {
                Debug.WriteLine("**** AuthenticationFailed");
                return Task.FromResult(0);
            },
        },
    },
);
```

I'll discuss each notification individually in a moment, but before I do, give the app a spin so that you can see in which order the notifications fire. When you click the Sign In link, you can expect to see something like this in the output window:

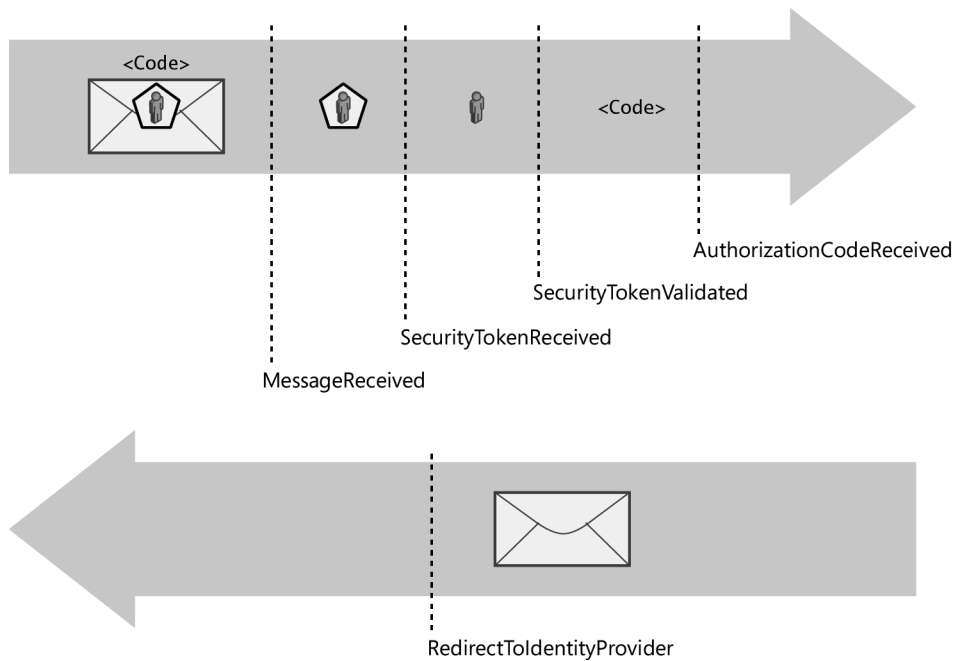
```
1 ==>request, before cookie auth
2 ==>after cookie, before OIDC
3 ==>after OIDC, before leaving the pipeline
4 <==after entering the pipeline, before OIDC
*** RedirectToIdentityProvider
5 <==after OIDC
6 <==response, after cookie auth
```

This shows that `RedirectToIdentityProvider` runs in the context of the OpenID Connect middleware, as expected.

Once you sign in with Azure AD and are redirected to the app, you can expect to see the following sequence:

```
1 ==>request, before cookie auth
2 ==>after cookie, before OIDC
*** MessageReceived
*** SecurityTokenReceived
*** SecurityTokenValidated
*** AuthorizationCodeReceived
5 <==after OIDC
6 <==response, after cookie auth
1 ==>request, before cookie auth
2 ==>after cookie, before OIDC
3 ==>after OIDC, before leaving the pipeline
4 <==after entering the pipeline, before OIDC
5 <==after OIDC
6 <==response, after cookie auth
```

This is the same token-processing and cookie-setting sequence you encountered earlier in this chapter. This time, you can see the other notifications fire and the order in which they execute. Figure 7-14 summarizes the sequence in which the notifications fire.



**FIGURE 7-14** The notifications sequence.

If you trigger a sign-out, you will see the usual sequence, but look between messages 4 and 5, and you will find that `RedirectToIdentityProvider` fires on sign-out as well.

Keep in mind also that notifications derive from a `BaseNotification` class from which they inherit a couple of methods exposing two fundamental capabilities. The first, `HandleResponse`, signals to the middleware pipeline that whatever logic has been executed in the notification concludes the processing of the current request, hence no other middleware should be executed. A notification calling this method has the responsibility of having everything in the context tidied up, including writing the full response. The second, `SkipToNextMiddleware`, signals to the middleware pipeline that whatever logic has been executed in the notification concludes the work that the current middleware should do on the request. Hence, any other request-processing code in the current middleware should not be executed, and the baton should be passed to the next middleware in the pipeline as soon as the notification concludes its work.

Now let's look at each notification in more detail.

### ***RedirectToIdentityProvider***

This is likely the notification you'll work with most often. It is executed right after the OpenID Connect middleware creates a protocol message, and it gives you the opportunity to override the option values the middleware uses to build the message, augment them with extra parameters, and so on. If you place a breakpoint in the notification and take a look at the context parameter, you'll see something like what's shown in Figure 7-15.

| Name                  | Value  | Type      |
|-----------------------|--|-----------|
| context               | (Microsoft.Owin.Security.Notifications.RedirectToIdentityProviderNotification<M... | Microsoft |
| HandledResponse       | false  | bool      |
| Options               | (Microsoft.Owin.Security.OpenIdConnect.OpenIdConnectAuthenticationOptions)         | Microsoft |
| OwinContext           | (Microsoft.Owin.OwinContext)   | Microsoft |
| ProtocolMessage       | (Microsoft.IdentityModel.Protocols.OpenIdConnectMessage)                           | Microsoft |
| AccessToken           | null   | string    |
| AcrValues             | null   | string    |
| AuthorizationEndpoint | null   | string    |
| ClaimsLocales         | null   | string    |
| ClientAssertion       | null   | string    |
| ClientAssertionType   | null   | string    |
| ClientId              | "c3d5b1ad-ae77-49ac-8a86-dd39a2f91081"   | string    |
| ClientSecret          | null   | string    |
| Code                  | null   | string    |
| Display               | null   | string    |
| DomainHint            | null   | string    |
| Error                 | null   | string    |
| ErrorDescription      | null   | string    |
| ErrorUri              | null   | string    |
| ExpiresIn             | null   | string    |
| GrantType             | null   | string    |
| IdToken               | null   | string    |
| IdTokenHint           | null   | string    |
| IdentityProvider      | null   | string    |
| Iss                   | null   | string    |
| IssuerAddress         | "https://login.microsoftonline.com/6c3d51dd-f0e5-4959-b4ea-a80c4e36fe..."          | string    |
| LoginHint             | null   | string    |
| MaxAge                | null   | string    |
| Nonce                 | "63573578222877558.NzhiNDQwNzYTYmNkZS00ZjNiLWJmOWItNWfKMGEG..."                    | string    |
| Parameters            | Count = 6  | System.C  |
| Password              | null   | string    |
| PostLogoutRedirectUri | null   | string    |
| PostTitle             | "Working..."   | string    |
| Prompt                | null   | string    |
| RedirectUri           | null   | string    |
| RequestType           | AuthenticationRequest  | Microsoft |
| RequestUri            | null   | string    |
| Resource              | null   | string    |
| ResponseMode          | "form_post"  | string    |
| ResponseType          | "code_id_token"  | string    |
| Scope                 | "openid profile"   | string    |
| ScriptButtonText      | "Submit"   | string    |
| ScriptDisabledText    | "Script is disabled. Click Submit to continue."                                    | string    |
| SessionState          | null   | string    |
| State                 | "OpenIdConnect.AuthenticationProperties=eyJzawDqKUBHDw24sv83Xt9qi..."              | string    |
| TargetLinkUri         | null   | string    |
| Token                 | null   | string    |
| TokenEndpoint         | null   | string    |
| TokenType             | null   | string    |
| UiLocales             | null   | string    |
| UserId                | null   | string    |
| Username              | null   | string    |
| Non-Public members    |  |           |
| Request               | (Microsoft.Owin.OwinRequest)   | Microsoft |
| Response              | (Microsoft.Owin.OwinResponse)  | Microsoft |
| State                 | Continue   | Microsoft |

**FIGURE 7-15** The content of the context parameter on a typical `RedirectToIdentityProvider` notification execution.

I expanded the `ProtocolMessage` in Figure 7-15 so that you can see that it already contains all the default parameters you have seen in the request on the traces in Chapter 6. There are a number of fun and useful things you can do here, so let's examine a couple of examples.

Say that my app is registered to run both on my local dev box (hence, on a localhost address) and on an Azure website (hence, on something like `myapp.azurewebsites.net`). That means that depending on where my app is running at the moment, I have to remember to set the correct `RedirectUri` and `PostLogoutRedirectUri` properties in the options right before deploying. Or do I? Consider the following code:

```
RedirectToIdentityProvider = (context) =>
{
```

```

string appBaseUrl = context.Request.Scheme + "://"
    + context.Request.Host + context.Request.PathBase;
context.ProtocolMessage.RedirectUri = appBaseUrl + "/";
context.ProtocolMessage.PostLogoutRedirectUri = appBaseUrl;
return Task.FromResult(0);
},

```

Here I simply read from the Request the URL being requested, indicating at which address my app is running at the moment and using it to inject the correct values of `RedirectUri` and `PostLogoutRedirectUri` in the message. Neat!

Or consider a case in which I want to guarantee that when an authentication request is sent, the user is always forced to enter credentials no matter what session cookies might already be in place. In Chapter 6 you learned that OpenID Connect will behave that way upon receiving a `prompt=login` parameter in the request, but how do you do it? Check out this code:

```

RedirectToIdentityProvider = (context) =>
{
    context.ProtocolMessage.Prompt = "login";
    return Task.FromResult(0);
},

```

That's it. From this moment on, every sign-in request will prompt the user for credentials. Easy. Now is the time to reap the benefits of having gone through all those nitty-gritty protocol details in Chapter 6; you can use this notification to control every aspect of the message to your heart's content. Of course, this applies to sign-out flows, too.

But before moving on to the next notification, I want to highlight that you don't have to put the code for your notifications in line. If you have notification-handling logic you want to reuse across multiple applications, you can put it in a function, package it in a class, and reuse it as you see fit. Explicitly creating a function is also indicated when the amount of code is substantial, or when you want to enhance readability. As a quick demonstration of this approach, let's rewrite the latest sample in an explicit function at the level of the Startup class:

```

public static Task RedirectToIdentityProvider(RedirectToIdentityProviderNotification<OpenIdConnectMessage, OpenIdConnectAuthenticationOptions> notification)
{
    notification.ProtocolMessage.Prompt = "login";
    return Task.FromResult(0);
}

```

Assigning it back in the `Notifications` is straightforward:

```

//...
Notifications = new OpenIdConnectAuthenticationNotifications()
{
    RedirectToIdentityProvider = Startup.RedirectToIdentityProvider,
// ...

```

I also like the aspect of this approach that makes more visible which parameters are being passed to the notification, which in turns makes it easier to understand what the notification is suitable for.

The `OpenIdConnectMessage` passed to `RedirectToIdentityProvider` is an excellent example of that.

### ***MessageReceived***

This notification is triggered when the middleware detects that the incoming message happens to be a known OpenID Connect message. You can use it for a variety of purposes; for example, for resources you want to allocate just in time (such as database connections), stuff you want to cache in memory before the message is processed further, and so on. Alternatively, you might use this notification for logging purposes. However, the main use I have seen for `MessageReceived` occurs when you want to completely take over the handling of the entire request (that's where `HandleResponse` comes into play, by the way). For example, you might use `MessageReceived` for handling `response_` types that the middleware currently does not automatically process, like a sign-in flow based on authorization code. That's not an easy endeavor, and as such not very common, but some advanced scenarios will sometimes require it, and this extensibility model makes doing so possible.

### ***SecurityTokenReceived***

`SecurityTokenReceived` triggers when the middleware finds an `id_token` in the request. Similar considerations as for `MessageReceived` apply, with finer granularity. Here, the entity being processed is the token, as opposed to the entire message.

### ***SecurityTokenValidated***

At the stage in which `SecurityTokenValidated` fires, the incoming `id_token` has been parsed, validated, and used to populate `context.AuthenticationTicket` with a `ClaimsIdentity` whose claims come from the incoming token.

This is the right place for adding any user-driven logic you want to execute before reaching the application itself. Common scenarios include user-driven access control and claims augmentation. Here are examples for each case.

Say that I run a courseware website where users can buy individual subscriptions for gaining access to training videos. I integrate with Azure AD, given that business users are very important to me, but my business model imposes on me the need to verify access at the user level. That means that the token validations you have studied so far aren't in themselves sufficient to decide whether a caller can gain access. Consider the following implementation of `SecurityTokenValidated`:

```
SecurityTokenValidated = (context) =>
{
    string userID = context.AuthenticationTicket.Identity.FindFirst(ClaimTypes.NameIdentifier).
Value;
    if (db.Users.FirstOrDefault(b => (b.UserID == userID)) == null)
        throw new System.IdentityModel.Tokens.SecurityTokenValidationException();
    return Task.FromResult(0);
},
```

The notification body retrieves a user identifier from the claims of the freshly created `AuthenticationTicket`. That done, it verifies whether that identifier is listed in a database of subscribers (whose existence I am postulating for the sake of the scenario). If the user does have an entry, everything goes on as business as usual. But if the user is not listed, the app throws an exception that creates conditions equivalent to the ones you would experience on receiving an invalid token. Simple!

Consider this other scenario. Say that your application maintains a database of attributes for its users—attributes that are not supplied in the incoming token by the identity provider. You can use `SecurityTokenValidated` to augment the set of incoming user claims with any arbitrary value you keep in your local database. The application code will be able to access those values just like any other IdP-issued claims, the only difference being the issuer value. Here's an example.

```
SecurityTokenValidated = (context) =>
{
    string userID = context.AuthenticationTicket.Identity.FindFirst(ClaimTypes.NameIdentifier).
    Value;
    Claim userHair = new Claim("http://mycustomclaims/hairlength", RetrieveHairLength(userID),
    ClaimValueTypes.Double, "LocalAuthority");
    context.AuthenticationTicket.Identity.AddClaim(userHair);
    return Task.FromResult(0);
},
```

Here I assume that you have a method that, given the identifier of the current user, queries your database to retrieve an attribute (in this case, hair length). Once you get the value back, you can use it to create a new claim (I invented a new claim type on the spot to show you that you can choose pretty much anything that works for you) and add that claim to the `AuthenticationTicket's ClaimsIdentity`. I passed `"LocalAuthority"` as the issuer identifier to ensure that the locally generated claims are distinguishable from the ones received from the IdP: the two usually carry a different trust level.

Now that the new claim is part of the ticket, it's going to follow the same journey we have studied so far for normal, nonaugmented identity information. Making use of it from the app requires the same code you already saw in action for out-of-the-box claim types.

```
public ActionResult Index()
{
    var userHair = ClaimsPrincipal.Current.FindFirst("http://mycustomclaims/hairlength");
    return View();
}
```

This is a very powerful mechanism, but it does have its costs. Besides the performance hit of doing I/O while processing a request, you have to keep in mind that whatever you add to the `AuthenticationTicket` will end up in the session cookie. In turn, that will add a tax for every subsequent request, and at times it might even blow past browser limits. For example, Safari is famous for allowing only 4 KB of cookies/headers in requests for a given domain. Exceed that limit and cookies will be clipped, signature checks will fail, nonces will be dropped, and all sorts of other hard-to-diagnose issues will arise.

## ***AuthorizationCodeReceived***

This notification fires only in the case in which the middleware emits a request for a hybrid flow, where the `id_token` is accompanied by an authorization code. I'll go into more details in a later chapter, after fleshing out the scenario and introducing other artifacts that come in handy for dealing with that case.

## ***AuthenticationFailed***

This notification gives you a way to catch issues occurring in the notifications pipeline and react to them with your own logic. Here's a simple example:

```
AuthenticationFailed = (context) =>
{
    context.OwinContext.Response.Redirect("/Home/Error");
    context.HandleResponse();
    return Task.FromResult(0);
},
```

In this code I simply redirect the flow to an error route. Chances are you will want to do something more sophisticated, like retrieving the culprit exception (available in the context) and then log it or pass it to the page. The interesting thing to notice here is the use of `HandleResponse`. There's nothing else that can make meaningful work in the pipeline after this, hence we short-circuit the request processing and send the response back right away.

## ***TokenValidationParameters***

---

You think we've gone deep enough to this point? Not quite, my dear reader. The rabbit hole has one extra level, which grants you even more control over your token-validation strategy.

`OpenIdConnectAuthenticationOptions` has a property named `TokenValidationParameters`, of type `TokenValidationParameters`.

The `TokenValidationParameters` type predates the RTM of Katana. It was introduced when the Azure AD team released the very first version of the JWT handler (a .NET class for processing the JWT format) as a general-purpose mechanism for storing information required for validating a token, regardless of the protocol used for requesting and delivering it and the development stack used for supporting such protocol. That was a clean break with the past: up to that moment, the same function was performed by special XML elements in the `web.config` file, which assumed the use of WIF and IIS. It was soon generalized to support the SAML token format, too.

The OpenID Connect middleware itself still uses the JWT handler when it comes to validating incoming tokens, and to do so it has to feed it a `TokenValidationParameters` instance with the desired validation settings. All the metadata inspection mechanisms you have been studying so far ultimately feed specific values—the issuer values to accept and the signing keys to use for validating incoming tokens' signatures—in a `TokenValidationParameters` instance. If you did not provide any values in the `TokenValidationParameters` property (I know, it's confusing) in the options, the



values from the metadata will be the only ones used. However, if you do provide values directly in `TokenValidationParameters`, the actual values used will be a merger of the `TokenValidationParameters` and what is retrieved from the metadata (using all the options you learned about in the “Authority coordinates and validation” section).

The preceding mechanisms hold for the validation of the parameters defining the token issuer, but as you know by now, there are lots of other things to validate in a token, and even more things that are best performed during validation. If you don’t specify anything, as is the case the vast majority of the time, the middleware fills in the blanks with reasonable defaults. But if you choose to, you can control an insane number of details. Figure 7-16 shows the content of `TokenValidationParameters` in OpenID Connect middleware at the initialization time for our sample application. I am not going to unearth all the things that `TokenValidationParameters` allows you to control (that would take far too long), but I do want to make sure you are aware of the most commonly used knobs you can turn.

| Name                      | Value   |
|---------------------------|---|
| TokenValidationParameters | {System.IdentityModel.Tokens.TokenValidationParameters}                         |
| AudienceValidator         | null  |
| AuthenticationType        | null  |
| CertificateValidator      | null  |
| ClientDecryptionTokens    | Count = 0   |
| ClockSkew                 | {00:05:00}  |
| IssuerSigningKey          | null  |
| IssuerSigningKeyResolver  | null  |
| IssuerSigningKeyValidator | null  |
| IssuerSigningKeys         | null  |
| IssuerSigningToken        | null  |
| IssuerSigningTokens       | null  |
| IssuerValidator           | {Method = {System.String <ConfigureAuth>b_0_7(System.String, System.IdentityMod |
| LifetimeValidator         | null  |
| NameClaimType             | "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name" 🔍                  |
| NameClaimTypeRetriever    | null  |
| RequireExpirationTime     | true  |
| RequireSignedTokens       | true  |
| RoleClaimType             | "http://schemas.microsoft.com/ws/2008/06/identity/claims/role" 🔍                |
| RoleClaimTypeRetriever    | null  |
| SaveSigninToken           | false   |
| TokenReplayCache          | null  |
| ValidAudience             | null  |
| ValidAudiences            | null  |
| ValidIssuer               | null  |
| ValidIssuers              | null  |
| ValidateActor             | false   |
| ValidateAudience          | true  |
| Validatelssuer            | true  |
| ValidatelssuerSigningKey  | false   |
| Validatelifetime          | true  |

**FIGURE 7-16** The `TokenValidationParameters` instance in `OpenIdConnectAuthenticationOptions`, as initialized by the sample application.

## Valid values

As you’ve learned, the main values used to validate incoming tokens are the issuer, the audience, the key used for signing, and the validity interval. With the exception of the last of these (which does not require reference values because it is compared against the current clock values), `TokenValidationParameters` exposes a property for holding the corresponding value: `ValidIssuer`, `ValidAudience`, and `IssuerSigningKey`.

What is less known is that `TokenValidationParameters` also has an `IEnumerable` for each of these—`ValidIssuers`, `ValidAudiences`, and `IssuerSigningKeys`—which are meant to make it easy for you to manage scenarios in which you need to handle a small number of alternative values. For example, your app might accept tokens from two different issuers simultaneously. Or you might use a different audience for your development and staging deployments but have a single codebase that automatically works in both.

## Validation flags

One large category of `TokenValidationParameters` properties allows you to turn on and off specific validation checks. These Boolean flags are self-explanatory: `ValidateAudience` turns on and off the comparison of the audience in the incoming claim with the declared audience (in the OpenID Connect case, the `clientId` value); `ValidateIssuer` controls whether your app cares about the identity of the issuer; `ValidateIssuerSigningKey` determines whether you need the key used to sign the incoming token to be part of a list of trusted keys; `ValidateLifetime` determines whether you will enforce the validity interval declared in the token or ignore it.

At first glance, each of these checks sounds like something you'd never want to turn off, but there are various occasions in which you'd want to. Think of the subscription sample I described for `SecurityTokenValidated`: in that case, the actual check is the one against the user and the subscription database, so the issuer check does not matter and can be turned off. There are more exotic cases: in the Netherlands last year, a gentleman asked me how his intranet app could accept expired tokens in case his client briefly lost connectivity with the Internet and was temporarily unable to contact Azure AD for getting new tokens.

There is another category of flags controlling constraints rather than validation flags. The first is `RequireExpirationTime`, which determines whether your app will accept tokens that do not declare an expiration time (the specification allows for this). The other, `RequireSignedTokens`, specifies whether your app will accept tokens without a signature. To me, a token without a signature is an oxymoron, but I did encounter situations (especially during development) where this flag came in handy for running some tests.

## Validators

Validation flags allow you to turn on and off validation checks. Validator delegates allow you to substitute the default validation logic with your own custom code.

Say that you wrote a SaaS application that you plan to sell to organizations instead of to individuals. As opposed to the user-based validation you studied earlier, now you want to allow access to any user who comes from one of the organizations (one of the issuers) who bought a subscription to your app. You could use the `ValidIssuers` property to hold that list, but if you plan to have a substantial number of customers, doing that would be inconvenient for various reasons: a flat lookup on a list might not work too well if you are handling millions of entries, dynamically extending that list without recycling the app would be difficult, and so on. The solution is to take full control of the issuer validation operation. For example, consider the following code:

```

TokenValidationParameters = new TokenValidationParameters
{
    IssuerValidator = (issuer, token, tvp) =>
    {
        if (db.Issuers.FirstOrDefault(b => (b.Issuer == issuer)) == null)
            return issuer;
        else
            throw new SecurityTokenInvalidIssuerException("Invalid issuer");
    }
}

```

The delegate accepts as input the issuer value as extracted from the token, the token itself, and the validation parameters. In this case I do a flat lookup on a database to see whether the incoming issuer is valid, but of course you can imagine many other clever validation schemes. The validator returns the issuer value for a less-than-intuitive reason: that string will be used for populating the `Issuer` value of the claims that will ultimately end up in the user's `ClaimsPrincipal`.

All the other main validators (`AudienceValidator`, `LifetimeValidator`) return Booleans, with the exception of `IssuerSigningKeyValidator` and `CertificateValidator`.

## Miscellany

Of the plethora of remaining properties, I want to point your attention to two common ones.

`SaveSignInToken` is used to indicate whether you want to save in the `ClaimsPrincipal` (hence, the session cookie) the actual bits of the original token. There are topologies in which the actual token bits are required, signature and everything else intact: typically, the app trades that token (along with its credentials) for a new token, meant to allow the app to gain access to a web API acting on behalf of the user. This property defaults to false, as this is a sizable tax.

The `TokenReplayCache` property allows you to define a token replay cache, a store that can be used for saving tokens for the purpose of verifying that no token can be used more than once. This is a measure against a common attack, the aptly called *token replay attack*: an attacker intercepting the token sent at sign-in might try to send it to the app again ("replay" it) for establishing a new session. The presence of the nonce in OpenID Connect can limit but not fully eliminate the circumstances in which the attack can be successfully enacted. To protect your app, you can provide an implementation of `ITokenReplayCache` and assign an instance to `TokenReplayCache`. It's a very simple interface:

```

public interface ITokenReplayCache
{
    bool TryAdd(string securityToken, DateTime expiresOn);
    bool TryFind(string securityToken);
}

```

In a nutshell, you provide the methods for saving new tokens (determining for how long they need to be kept around) and bringing a token up from whatever storage technology you decide to use. The cache will be automatically used at every validation—take that into account when you pit latency and storage requirements against the likelihood of your app being targeted by replay attacks.

## More on sessions

---

Before I close this long chapter, I need to spend a minute on session management. You already know that by default, session validity will be tied to the validity specified by the token itself, unless you decouple it by setting the option `UseTokenLifetime` to `false`. When you do so, the `CookieAuthenticationOptions` are now in charge of session duration: `ExpiresTimeSpan` and `SlidingExpiration` are the properties you want to keep an eye on.

You also know that the cookie middleware will craft sessions that contain the full `ClaimsPrincipal` produced from the incoming token, but as mentioned in discussing the use of `SaveSignInToken`, the resulting cookie size can become a problem. This issue can be addressed by saving the bulk of the session server-side and using the cookie just to keep track of a reference to the session data on the server. The cookie middleware allows you to plug in an implementation of the `IAuthenticationSessionStore` interface, which can be used for customizing how an `AuthenticationTicket` is preserved across calls. If you want to provide an alternative store for your authentication tickets, all you need to do is implement that interface and pass an instance to the cookie middleware at initialization. Here's the interface:

```
public interface IAuthenticationSessionStore
{
    Task<string> StoreAsync(AuthenticationTicket ticket);
    Task RenewAsync(string key, AuthenticationTicket ticket);
    Task<AuthenticationTicket> RetrieveAsync(string key);
    Task RemoveAsync(string key);
}
```

That's pretty much a CRUD interface for an `AuthenticationTicket` store, which you can use for any persistence technology you like. Add some logic for cleaning up old entries and keeping the store size under control, and you have your custom session store.

Considerations about I/O and latency are critical here, given that this guy will trigger every single time you receive an authenticated request. A two-level cache, where most accesses are in-memory and the persistence layer is looked up only when necessary, is one of the solutions you might want to consider.

## Summary

---

This chapter explored in depth what happens when the OpenID Connect middleware and its underlying technologies process requests and emit responses. You learned about the main functional components of the request-processing pipeline, how they communicate with one another, and what options you have to change their behavior.

The complexity you have confronted here is something that the vast majority of web developers will never have to face—or even be aware of. Even in advanced cases, chances are that you will always use a subset of what you have read here. Don't worry if you don't remember everything; you don't have to. After the first read, this chapter is meant to be a reference you can return to whenever you are trying to achieve a specific customization or are troubleshooting a specific issue. Now that you've had an opportunity to deconstruct the pipeline, you'll know where to look.

The next chapter will be significantly lighter. You'll learn more about how Azure AD represents applications.

# Azure Active Directory application model

It's time to take a closer look at how Azure AD represents applications and their relationships to other apps, users, and organizations.

You got a brief taste of the Azure AD application model in Chapter 3, "Introducing Azure Active Directory and Active Directory Federation Services." Later on you experienced firsthand a couple of ways to provision apps and use their protocol coordinates in authentication flows. Here I will go much deeper into the constructs used by Azure AD to represent apps, the mechanisms used to provision apps beyond one's own organization, and the consent framework, which is the backbone of pretty much all of this. I'll also touch on roles, groups, and other features that Azure AD offers to grant fine-grained access control to your application.

The application model in Azure AD is designed to sustain many different functions:

- It holds all the data required to support authentication at run time.
- It holds all the data for deciding what other resources an application might need to access and whether a given request should be fulfilled and under what circumstances.
- It provides the infrastructure for implementing application provisioning, both within the app developer's tenant and to any other Azure AD tenant.
- It enables end users and administrators to dynamically grant or deny consent for the app to access resources on their behalf.
- It enables administrators to be the ultimate arbiters of what apps are allowed to do and which users can use specific apps, and in general to be stewards of how the directory resources are accessed.

That is A LOT more than setting up a trust relationship, the basic provisioning step you perform with traditional on-premises authorities like ADFS. Remember how I often bragged about how much easier it is to provision apps in Azure AD? What makes that feat possible is the highly sophisticated application model in Azure AD, which goes to great lengths to make life easy for administrators and end users. Unfortunately, the total complexity of the system remains roughly constant, so somebody must work harder to compensate for that simplification, and this time that somebody is the developer. I could work around that complexity and simply give you a list of recipes to follow to the letter for the most common tasks, but by now you know that this book doesn't work that way. Instead, we'll

dig deep to understand the building blocks and true motivation of each moving part—and once we emerge, everything will make sense. Don't worry, the model is very manageable and, once you get the hang of it, even easy, but some investment is required to understand it. This chapter is here to help you do just that.

## The building blocks: *Application* and *ServicePrincipal*

---

Since Azure AD first appeared on the market, a lot of content has been published about its application model. A large part of that content was produced while the application model had not yet solidified in its current form. To avoid any confusion, I am going to open this section with a bit of history: by understanding how we got to where we are today, you won't risk getting confused if you happen to stumble on documentation and samples from another epoch.

In traditional Active Directory, every entity that can be authenticated is represented by a *principal*. That's true for users, and that's true for applications—in the latter case, we speak of *service principals*. In traditional Kerberos, service principals are used to ensure that a client is speaking to the intended service and that a ticket is actually intended for a given service. In other words, they are used for any activity that requires establishing the identity of the service application itself.

Although Azure AD has been designed from the ground up to address modern workloads, it remains a directory. As such, it retains many of the concepts and constructs that power its on-premises ancestor, and service principals are among those. If you use the Internet time machine and fish out content from summer 2012, describing the very first preview of Azure AD development features, you'll see that at that time, provisioning an application in Azure AD was done by using special Windows PowerShell cmdlets, which created a new service principal for the app in the directory. Even the format of the service principal name was a reminder of its Kerberos legacy, following a fixed schema based on the app's execution environment. Disregarding the protocols it enabled, that service principal already had all the things we know are needed for supporting authentication transactions: application identifiers, a redirect URI, and so on.

Service principals are a great way of representing an application's instance, but they aren't very good at supporting the development of the application itself. Their limitations stem from two key considerations:

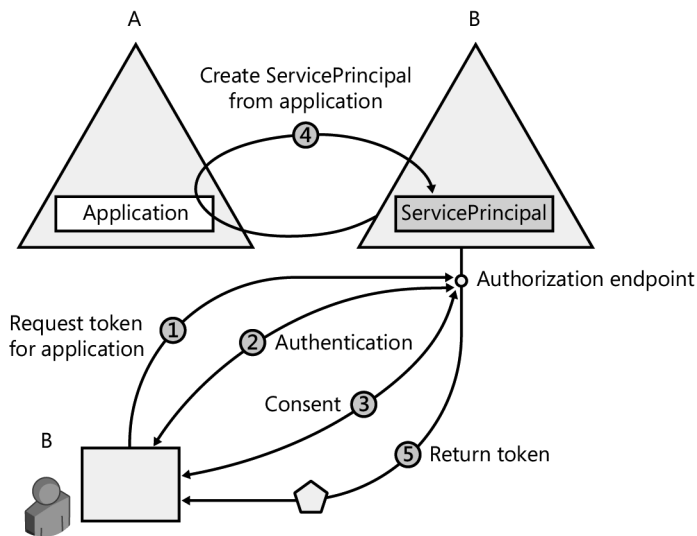
- Applications are usually an abstract entity, made of code and resources: the service principal represents a concrete instance of that abstract entity in a specific directory. You will want that abstract entity to have many concrete instances, especially if you build and sell software for a living: one or more instances for each of your customers' organizations. Even if you are building applications for your own organization, to be used by your colleagues, chances are that you'll want to work with multiple instances—for example, development, staging, and production. If the only building block at your disposal were app instances, development and deployments would be unnatural, denormalized, and repetitive. For one thing, every time you changed something, you'd have to go chase all your app instances and make the same change everywhere.

- Although so far we have seen applications mostly as resources one user gains access to, a directory sees applications as clients, which need to access resources under the control of the directory. Even the act of a user requesting a token for accessing an application is seen by the directory as the application itself gaining access to the user's identity information. Through this optic, you can see how some applications can be pretty powerful clients, performing functions that range from reading users' personally identifiable information (PII) to modifying the directory itself: deleting users, creating groups, changing passwords—the works. Application instances are normally put in operation by administrators, who enjoy those powers themselves. Hence, they have the faculty to imbue applications with such capabilities. If your company is big enough for employees not to have to juggle multiple hats, however, developers are traditionally not administrators. If service principals were the only way to create an application, very few employees in a company would have the power to develop apps. It gets worse: in today's software as a service (SaaS) push, it is in the developer's best interest that end users be empowered to elect to start using applications, but most users aren't administrators either. Even more than in the development case, this exposes the limits of perpetrating the service principal model "as is" in the cloud.

Given this, and for various other reasons, Azure AD defines a new entity, the `Application`, which is meant to describe an application as an abstract entity: a template, if you will. As a developer, you work with `Applications`. At deployment time a given `Application` object can be used as a blueprint to create a `ServicePrincipal` representing a concrete instance of an application in a directory. It's that `ServicePrincipal` that is used to define what the app can actually do in that specific target directory, who can use it, what resources it has access to, and so on.

Bear with me just a little longer, the abstract part is almost over. The main way through which Azure AD creates a `ServicePrincipal` from an `Application` is *consent*. Here's a simplified description of the flow: Say that you create an `Application` object in directory A, supplying all the protocol coordinates we've discussed so far in earlier chapters. Say that a user from tenant B navigates to the app's pages and triggers an authentication flow. Azure AD authenticates the user from B against its home directory, B. In so doing, it sees that there is no `ServicePrincipal` for the app in B; hence, it prompts the user about whether he or she wants to consent for that app to have access to the directory B (you'll see later in what capacity). If the user grants consent, Azure AD uses the `Application` object in A as a blueprint for creating a `ServicePrincipal` in B. Along with that, B records that the current user consented to the use of this application (expect lots of details on this later on). Once that's done, the user receives a token for accessing the app . . . and provisioning magically happens. No lengthy negotiations between administrators required. Isn't Azure AD awesome? Figure 8-1 summarizes the process.





**FIGURE 8-1** Simplified provisioning flow driven by consent: 1) a user from B attempts to sign in with the app; 2) the user credentials are acquired and verified; 3) the user is prompted to consent for the app to gain access to tenant B; the user consents; 4) Azure AD uses the `Application` object in A as a blueprint for creating a `ServicePrincipal` in B; 5) the user receives the requested token.

You can iterate the process shown in Figure 8-1 as many times as you want, for directory C, D, E, and so on. Directory A retains the blueprint of the app, in the form of its `Application` object. The users and admins of all the directories where the app is given consent retain control over what the application is allowed to do (and a lot more) through the corresponding `ServicePrincipal` object in each tenant.

### A special case: App creation via the Azure portal and Visual Studio

As I write, both of the application provisioning techniques you've experienced so far (using the Azure portal and using Visual Studio) assume that you want to run your application in the same tenant in which you are creating it. Hence, these techniques create both the `Application` and the `ServicePrincipal` objects. The presence of a `ServicePrincipal` right after creation time in the home tenant will cause differences in behavior in respect to what happens when the application is consumed through different tenants. That is especially true for native applications, which are out of scope for this book, but in general this is something you need to be aware of. Note that the current behavior is not set in stone and not part of any explicit contract. I cannot guarantee that it will not change after this book goes to the printer.

In the next two subsections, you'll take a look at the content of the `Application` and `ServicePrincipal` objects. This will give me an opportunity to introduce lots of new directory artifacts, which in turn will refine your understanding of what an application is for Azure AD and what it can do for you.



**Note** Your hands-on experience so far has been limited to implementing web sign-on to applications with a web interface, rendering their own user experience (UX) in a browser. The `Application` and `ServicePrincipal` objects are also used to model web APIs, which follow a different set of protocols. I am going to show you how to write web API projects in the next chapter, but I cannot wait until then to describe those concepts—they play such a central role in the Azure AD application model, in consent, and in provisioning that everything would sound weird without them. This is just to ensure that you know what's coming and don't get confused when I suddenly start to talk about OAuth and exposing scopes.

## The Application

The `Application` object in Azure AD is meant to describe three distinct aspects of an application:

- The identifiers, protocol coordinates, and authentication options that come into play when a token is requested for accessing the application.
- The resources that the application itself might need to access, and the actions it might need to take, in order to perform its functions. For example, an application might need to write back to the directory, or it might need to send email via Exchange as the authenticated user. You'll have to wait until the next chapter to learn how to actually perform these actions in code, but it's important to understand in this context the provisioning and consent mechanisms underpinning this aspect.
- The actions that the application itself offers. For example, an application representing a facade for a data store might allow for read and write operations—and make it possible for the directory to decide whether to grant a client permission to do only read operations, or both read and write, depending on the identity of the client. This feature is used when the application is a web API, but it rarely comes into play when doing web sign-on, so I won't spend much time on it in this chapter.

So far you've acted directly only on the first aspect. You indirectly took advantage of the defaults in the second point—every web app is configured to ask for permissions to sign in and access the user's profile. You have not interacted with the third aspect yet, but you will in Chapter 9, "Consuming and exposing a web API protected by Azure Active Directory."

Mercifully, neither the Azure portal or the Visual Studio ASP.NET project templates wizards ask you to provide values for all the properties that constitute an `Application` object. The vast majority of those properties are assigned default values that work great for most of the populace, who can get their web sign-on functionality by providing just a handful of strings (as you have seen, mainly name and `redirect_uri`) without ever being aware that there are customizations available.

That said, if you do want to know what's available in the `Application` object, how would you go about it? You have three strategies to choose from:

- Head to the Azure portal (<https://manage.windowsazure.com>), go to the Azure AD section, select the Applications tab, search for your app, select it, then click Configuration. You'll see far more info there than you provided at creation time. One example you are already familiar with is the `client_id`, which is assigned by Azure AD to your app when it's created.

The information shown there is what you would probably customize to meet the requirements of the most common scenarios. However, not all the application features are exposed there.

- Still in the Azure portal, with your app selected, you can use a link at the bottom of the page, Manage Manifest, to download a JSON file that contains the verbatim dump of the corresponding `Application` entity in the directory. You can edit this file to change whatever you want to control, then upload it again (through the same portal commands) to reflect your new options in the directory.
- Finally, you can use the Directory Graph API (mentioned in Chapter 3) to query the directory and GET the `Application` object, once again in JSON format.

The first method goes against the policy I am adopting in the book—the portal UX can change far too easily after the book is in print, so including screenshots of it would be a bad idea. Also, it does not go nearly deep enough for my purposes here.

The second method, the manifest, would work out well—and is the method I advise you to use when you work with your applications. However, there is something that makes it less suitable for explaining the anatomy of the `Application` object for the first time: the manifest is a true object dump from the directory, and for pure inheritance reasons it includes lots of properties that aren't useful or relevant for the `Application` itself.

To keep the signal-to-noise ratio as crisp as possible, the JSON snippets I'll show you here will all be obtained through the third method, direct queries through the Graph. I am using a very handy sample web app (which you can find at <https://graphexplorer.cloudapp.net>), which provides an easy UI for querying the graph. I cannot guarantee that the app will still be available when you read this book, but performing those queries through code, or with curl or via Fiddler, is extremely easy. In the next chapter you'll learn how.

Following is a dump of the `Application` object that corresponds to the sample app we've been working with so far. The query I used for obtaining it is as follows:

```
https://graph.windows.net/developertenant.onmicrosoft.com/
applications?$filter=appId+eq+'e8040965-f52a-4494-96ab-0ef07b591e3f'&api-version=1.5
```

You'll likely recognize the typical OData '\$' syntax. The GUID you see there is the `client_id` of the application. Here's the complete JSON from the result:

```
{
  "odata.metadata": "https://graph.windows.net/developertenant.onmicrosoft.com/$metadata#directoryObjects/Microsoft.DirectoryServices.Application",
  "value": [
    {
      "odata.type": "Microsoft.DirectoryServices.Application",
      "objectType": "Application",
```

```

    "objectId": "c806648a-f27d-43fd-9f18-999f7708fcfc",
    "deletionTimestamp": null,
    "appId": "e8040965-f52a-4494-96ab-0ef07b591e3f",
    "appRoles": [],
    "availableToOtherTenants": false,
    "displayName": "WebAppChapter5",
    "errorUrl": null,
    "groupMembershipClaims": null,
    "homepage": "https://localhost:44300/",
    "identifierUris": [
      "https://localhost:44300/WebProjectChapter5"
    ],
    "keyCredentials": [],
    "knownClientApplications": [],
    "logoutUrl": null,
    "oauth2AllowImplicitFlow": false,
    "oauth2AllowUrlPathMatching": false,
    "oauth2Permissions": [
      {
        "adminConsentDescription": "Allow the application to access WebAppChapter5 on behalf
of the signed-in user.",
        "adminConsentDisplayName": "Access WebAppChapter5",
        "id": "00431d04-5334-4da6-8396-0e6f54631f10",
        "isEnabled": true,
        "type": "User",
        "userConsentDescription": "Allow the application to access WebAppChapter5 on your
behalf.",
        "userConsentDisplayName": "Access WebAppChapter5",
        "value": "user_impersonation"
      }
    ],
    "oauth2RequirePostResponse": false,
    "passwordCredentials": [],
    "publicClient": null,
    "replyUrls": [
      "https://localhost:44300/"
    ],
    "requiredResourceAccess": [
      {
        "resourceAppId": "00000002-0000-0000-c000-000000000000",
        "resourceAccess": [
          {
            "id": "311a71cc-e848-46a1-bdf8-97ff7156d8e6",
            "type": "Scope"
          }
        ]
      }
    ],
    "samlMetadataUrl": null
  }
]
}

```

Feel free to ignore anything that starts with “odata” here. Also, some properties listed are for internal use only or are about to be deprecated, so I won’t talk about those.

The most “meta” properties here are `objectId` and `deletionTimestamp`.

- `objectId` is the unique identifier for this `Application` entry in the directory. Note, this is *not* the identifier used to identify the app in any protocol transaction—you can think of it as the ID of the row where the `Application` object is saved in the directory store. It is used for referencing the object in most directory queries and in cross-entity references.
- `deletionTimestamp` is always null, unless you delete the `Application`, which in that case it records the instant in which you do so. Azure AD implements most eliminations as soft deletes so that you can repent and restore the object without too much pain should you realize the deletion was a mistake.

## Properties used for authentication

The bulk of the properties of the `Application` object control aspects of the authentication, specifying parameters that define the app from the protocol’s perspective, turning options on and off, and providing experience customizations.

### Property naming galore

One important thing to keep in mind: Although in this book I am focusing on OAuth2 and OpenID Connect, the `Application` object must support all the protocols that Azure AD implements. As you have seen in previous chapters, all claims-oriented protocols share some common concepts—issuer, audience, URLs to receive returned tokens, and so on. That helps to keep the list of properties short, given that you need to specify the URL where you want to get the tokens back only once and use it with all protocols. However, it also creates a problem: If WS-Federation calls that URL *wsreply*, and OAuth2 calls it *redirect\_uri*, what should the corresponding property in the `Application` object be called? You’ll see that the question has been answered in many different ways through the object model, largely driven by historical circumstances (for example, which protocols were implemented first). That has led to some confusion, which prompted remediation attempts by surfacing those properties in the Azure portal UX under different labels . . . which led to further confusion. This is just a heads-up to highlight the importance of being very precise when you reason about Applications and protocol literature.

Here’s the complete list:

- `appId` This corresponds to the `client_id` of the application.
- `replyUrls` This multivalued property holds the list of registered `redirect_uri` values that Azure AD will accept as destinations when returning tokens. No other URI will be accepted. This property is the source of some of the most common errors: even the smallest mismatch (trailing slash missing, different casing) will cause the token-issuance operation to fail.

Although at creation time the only URL in the collection is the one you specified, as is the case with the localhost-based URL in the sample here, you’ll often find yourself adding more URLs

as your app moves past the development stage and gets deployed to staging and production. If you want to achieve complete isolation between application deployments, you can always create an entirely new `Application` for every environment, each with its own `client_id`.

- `identifierUri` This multivalue property holds a collection of developer-assigned application identifiers, as opposed to the directory-assigned `client_id`.

These values are used to represent the application as a resource in protocols such as SAML and WS-Federation, where they map to the concept of realm. The URIs are also used as audience in access tokens issued for the app via OAuth2, when the app is consumed as a web API (as opposed to a web app with an HTML UX). This often generates confusion, given that this scenario can also be implemented by using the app's `client_id` instead of one identifier URI. More about this in Chapter 9.

- `publicClient` In the current Azure AD model, applications can be either confidential clients (apps that can have their own credentials, usually run on servers, etc.) or public clients (mobile and native apps running on devices, with no credentials, hence no strong identity of their own). The security characteristics of the two app types are very different, and so is the set of protocols that the two types support. For example, a native client cannot obtain a token purely with its app identity because it has no identity of its own; and a confidential client cannot request tokens with user-only flows, where the identity of the app would not play a role.

This book focuses on web apps; hence, confidential clients. That means that the apps discussed here will always have the `publicClient` property set to null.

- `passwordCredentials`, `keyCredentials` These properties hold references to application-assigned credentials, string-based shared secrets and X.509 certificates, respectively. Only confidential clients can have nonempty values here. Those credentials come into play when requesting access tokens—in other words, when the app is acting as a client rather than as a resource itself. You'll see more of that in the next chapter.
- `displayName` This property determines how the application is called in interactions with end users, such as consent prompts. It's also the mnemonic moniker used to indicate the application for the developer in the Azure management portal. Given that the display name has no uniqueness requirements, it's not always a way to conclusively identify one app in a long list.
- `Homepage` The URL saved here is used to point to the application from its entry in application portals such as the Office 365 application store. It does not play any role in the protocol behavior of the app; it's just whatever landing page you want visitors, prospective buyers, and corporate users (who might get there through the list of applications their company uses) to use as an entry point. At creation time, the `Homepage` value is copied from the `replyUrls` property. A common bit of advice to software developers from Office is to ensure that the URL in `Homepage` corresponds to a protected page/route in your application so that if visitors are already authenticated when they click the link, they'll find themselves authenticated with the same identity in your app as well.

- `samlMetadataUrl` In case you are implementing SAML in your app, this property allows you to specify where your app publishes its own SAML metadata document.
- `oauth2AllowImplicitFlow` This flag, defaulting to false, determines whether your app allows requests for tokens for the app via implicit flow.
- `oauth2AllowUrlPathMatching` By default, Azure AD requires all `redirect_uris` in a request to be a perfect match of any of the entries in `replyURLs`. This is a very good policy, designed to mitigate the open redirector attack—an attack in which appending extra parameters to one `redirect_uri` might lead to the resulting token being forwarded to a malicious party. However, there are situations in which your app might need to have more flexibility and use `return_uris` that do have a tail of extra characters that aren't part of the registered values. Setting this property to true tells Azure AD that you want to relax the perfect match constraint, and allows you to use URLs that are a superset of the ones you registered. Before changing this value, make sure you truly need it and that you have mitigations in place.
- `oauth2RequirePostResponse` Azure AD expects all requests to be carried through a GET operation. Setting this property to true relaxes that constraint.
- `groupMembershipClaims` If you want to receive group membership information as claims in the tokens issued for your user, you can use this property to express that requirement. Setting `groupMembershipClaims` to `SecurityGroup` results in a token containing all the security group memberships of the user. Setting it to `All` results in a token containing both security group and distribution list memberships. The default, null, results in no group information in the token. Note that the group claims do not include the group name; rather, they carry a GUID that uniquely identifies the group within the tenant. I'll spend more time on this topic later in the chapter.
- `appRoles` This property is used for declaring roles associated with the application. I provide a complete explanation of this property in later sections of this chapter.
- `availableToOtherTenants` This property deviates from the strictly protocol-related functionalities: it's more about controlling the provisioning aspect. Every confidential client application starts its existence as an app that can be accessed only by accounts from the same directory tenant in which the application was created. That's the typical line-of-business application scenario, where the IT department of one company develops an app to be used by their fellow employees. Any attempt to get tokens for the app from a different tenant will not work (excluding guest scenarios, which will be mentioned later).

However, that clearly does not work if your intent is to make the application available across organizations: that is the case for SaaS scenarios, naturally. If you are in that situation, you can flip `availableToOtherTenants` to true. That will make Azure AD allow requests from other tenants to trigger the consent flow I described briefly earlier instead of carrying out the default behavior, in which the request would be rejected right away.

Applications available across tenants (what we commonly call “multitenant apps”) have extra constraints. For example, whereas `identifierURIs` can normally be any URI with

no restrictions, for multitenant apps those URLs must be proper URLs and their hostname must match a domain that is registered with the tenant. Also, only tenant administrators can promote an app to be multitenant. The consent for a multitenant app clearly identifies the tenant as the publisher of the app to potentially every other organization using Azure AD—with important repercussions on reputation should something go south.



**Note** Flipping this switch only tells Azure AD that you want your app to behave as a multitenant app. Actually promoting one application from line of business to multitenant requires some coding changes, which I'll discuss later on.

- `knownClientApplications` The last property listed here is also about provisioning. You have seen how consenting for one application to have access to your own directory results in the creation of a `ServicePrincipal` for the app in the target directory. To anticipate a bit the upcoming discussion on permissions, the idea is that the `ServicePrincipal` will also need to record the list of resources and actions on those resources that the user consented to. This is possible only if the requested resources are already present with their own `ServicePrincipal` entries in the target directory. That is usually the case for first-party resources: if your app needs access to the Directory Graph or Exchange online, you can expect those to already have an entry in the directory. It will occasionally happen that your solution includes both a client application and one custom web API application. You'll want your prospective customers to have to consent only once, when they first try to get a token for the client application. If consent can happen only when all the requested resources are present as a `ServicePrincipal` in the target directory, and one of the resources you need is your own API, you have a problem. It looks like you have to ask your user to first consent to the web API so that it can create its `ServicePrincipal` in the target directory, and only after that ask the user to go back and consent to the client application.

Well, this property exists to save you from having to do all that work. Say that the application you are working on is the web API project. If you save in `knownClientApplications` the `client_id` (the `appId`, that is) of the client application you want to use for accessing your API, Azure AD will know that consenting to the client means implicitly consenting to the web API, too, and will automatically provision `ServicePrincipals` for both the client and web API at the same time, with a single consent. Handy!

The main catch in all this is that both the client and the web API application must be defined within the same tenant. You cannot list in `knownClientApplications` the `client_id` of a client defined in a different tenant.

### ***oauth2Permissions: What actions does the app expose?***

The `oauth2Permissions` collection publishes the list of things that client applications can do with your app—the scopes the app admits, mostly, but that comes into play only in case your app is a web API. If your app is a web application with a UX, the expectation is that browsers will request tokens for your app with the goal of signing in. That does not require any entry for web sign-on, the scenario



considered in this chapter, so I thought of deferring coverage of this property until I get to exposing your own web API, but some of the concepts will come in handy sooner than that, so I'll give you a bit of background now. Let's take a closer look at the only entry in the `oauth2Permissions` collection for the sample application:

```
{
  "adminConsentDescription": "Allow the application to access WebAppChapter5 on behalf of the signed-in user.",
  "adminConsentDisplayName": "Access WebAppChapter5",
  "id": "00431d04-5334-4da6-8396-0e6f54631f10",
  "isEnabled": true,
  "type": "User",
  "userConsentDescription": "Allow the application to access WebAppChapter5 on your behalf.",
  "userConsentDisplayName": "Access WebAppChapter5",
  "value": "user_impersonation"
}
```

### Where does the default `oauth2Permissions` entry come from?

Answering this question requires a bit of history. For the way in which Azure AD is organized, a token obtained by a client for accessing a web API must contain at least a scope—which is, as you have seen, an action that the client obtains permissions to perform. An application representing a web API but not defining any scopes would be impossible to access because any token request would not have any scope to prompt consent for. That wasn't always the case! This constraint was added a few months after Azure AD was released, creating a lot of confusion for developers who were now expected to manually add at least one `oauth2Permission` entry before being able to use their API. This also influenced all the walk-through and sample readme files at the time, making it necessary to add instructions on how to add that entry. I am happy to report that such manual steps are no longer necessary. Every `Application` is created with one default permission, `user_impersonation`, so that if you want to implement your app as a web API you don't need any extra configuration step, and you can begin development right away. I am telling you all this because some of the walk-throughs from that phase are still around. Now you know that you don't need to follow them to the letter on this.

The schema is pretty straightforward:

The ID uniquely identifies the permission within this resource.

Each property ending in "description" or "name" indicates how to identify and describe this permission in the context of interactive operations, such as consent prompts or `Application` configuration at development time.

The `type` property indicates whether this permission can be granted by any user in the directory (in which case it is populated with the value `User`) or is a high-value capability that can be granted only by an administrator (in which case, the value is `Admin`).

The `value` property represents the value in the scope claim that a token will carry to signal the fact that the caller was granted this permission by the directory. That is what the app should look for in the incoming token to decide whether the caller should be allowed to exercise the function gated by this permission.

I'll come back to this collection in Chapter 9.

### ***requiredResourceAccess***: What resources the app needs

This is one of the most powerful entries in the `Application` object, which can lead to utter despair when things go wrong:

```
"requiredResourceAccess": [
  {
    "resourceAppId": "00000002-0000-0000-c000-000000000000",
    "resourceAccess": [
      {
        "id": "311a71cc-e848-46a1-bdf8-97ff7156d8e6",
        "type": "Scope"
      }
    ]
  }
]
```

You can think of `requiredResourceAccess` as the client-side partner of `oauth2Permissions`. The `requiredResourceAccess` entry lists all the resources and permissions the application needs access to, referring to the entries each of those resources expose through their own `oauth2Permissions` entries. For each resource, `requiredResourceAccess` specifies:

- The `resourceAppId` of the requested resource, via the `resourceAppId` property
- Which specific permissions it is after, via the `resourceAccess` collection, which contains
  - The permission ID—the same ID the resource declared (in its own `Application` object) for the permission in its own corresponding `oauth2Permission` entry
  - The type of access it intends to perform: possible values are `Scope` and `Role`.

In our sample, the resource we need access to is the directory itself, in the form of the Graph API—the identifier `00000002-0000-0000-c000-000000000000` is reserved for the Graph in all tenants. The permission we are requesting, of ID `311a71cc-e848-46a1-bdf8-97ff7156d8e6`, corresponds to “sign in and access the user’s profile.” I know it doesn’t sound that easy to remember . . . but it is not supposed to be. The Azure portal or the Visual Studio project wizards normally take care of putting those values there for you when you select the human-readable counterparts in their UIs.

The type of access `Scope` determines that the app request the permission in delegated fashion; that is to say, as the identity of the user who’s doing the request. Whether an admin user is required for successfully obtaining this permission at run time, or a normal user can suffice, is determined by the `Type` declared in the corresponding `oauth2Permission` entry—found in the `Application` object of the resource exposing the permission. As you have seen in the preceding section, the

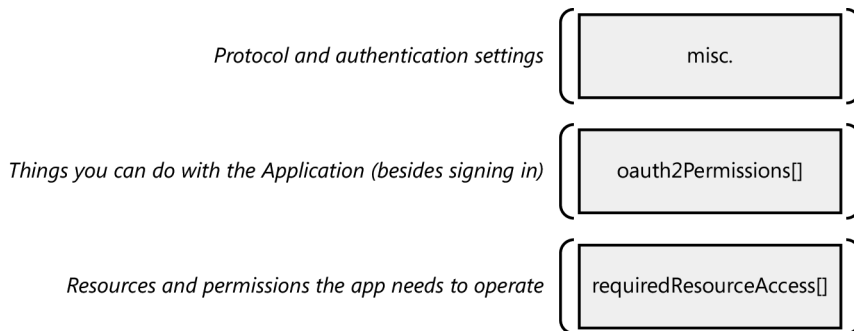
possible values are User and Admin. If the permission declares the latter, only an administrator can consent to it.

A `requiredResourceAccess` entry with a `Type` of value `Role` indicates that the application requires that permission with its own application identity, regardless of which user identity is used to obtain the token (if any—there are ways for an app to get tokens with no users involved, and I'll talk about that in the "Application permissions" section toward the end of this chapter). This option does require consent from an administrator.

Now here is a super important concept; put everything else down and read very carefully. In the current Azure AD model, one application must declare in advance *all* resources it needs access to, and *all* the associated permissions it requires. At the first request for a token for that app, that list will be presented to the user in its entirety, regardless of what resources are actually needed for that specific request. Once the user successfully grants consent, a `ServicePrincipal` will be provisioned, and that consent will be recorded in the target directory (you'll see later how that happens in practice) for all the requested resources. This makes it possible to prompt the user for consent only once.

The side effect of this approach is that the list of consented permissions is static. If you decide to add a new permission request to your application after a customer of yours already consented to it in its own directory, your customer will not be able to obtain the new permission for your app in the customer's own tenant until he or she revokes consent in its entirety and then grants it again. This can sometimes be painful. In version 2 of Azure AD, we are working hard to eliminate this constraint, but in version 1, that is the way it is today.

Figure 8-2 summarizes the main functional groups the `Application` object's properties fall into. Sure, there are a lot of details to keep in mind, but at the end of the day, more often than not, this simple subdivision will help you to ignore the noise and zero in on the properties you need for your scenario.



**FIGURE 8-2** A functional grouping of the properties of the `Application` object in Azure AD.

## The *ServicePrincipal* object

In later sections you will study in detail how an app goes from one `Application` object in one tenant to one or more `ServicePrincipals` in one or more tenants. In this section, I've assumed that such provisioning has already happened and will focus on the properties of the resulting `ServicePrincipal`: what properties are copied as is from the `Application`, what doesn't make it through, and what's added that is unique.

Following is the `ServicePrincipal` for our sample app. It is deployed on the same tenant as the `Application`, but for our analysis that doesn't matter. At first glance, it does look a lot like the `Application` itself, but it is in fact quite different.

```
{
  "odata.type": "Microsoft.DirectoryServices.ServicePrincipal",
  "objectType": "ServicePrincipal",
  "objectId": "29f565fd-0889-43ff-aa7f-3e7c37fd95b4",
  "deletionTimestamp": null,
  "accountEnabled": true,
  "appDisplayName": "WebAppChapter5",
  "appId": "e8040965-f52a-4494-96ab-0ef07b591e3f",
  "appOwnerTenantId": "6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e",
  "appRoleAssignmentRequired": false,
  "appRoles": [],
  "displayName": "WebAppChapter5",
  "errorUrl": null,
  "homepage": "https://localhost:44300/",
  "keyCredentials": [],
  "logoutUrl": null,
  "oauth2Permissions": [
    {
      "adminConsentDescription": "Allow the application to access WebAppChapter5 on behalf
of the signed-in user.",
      "adminConsentDisplayName": "Access WebAppChapter5",
      "id": "00431d04-5334-4da6-8396-0e6f54631f10",
      "isEnabled": true,
      "type": "User",
      "userConsentDescription": "Allow the application to access WebAppChapter5 on your
behalf.",
      "userConsentDisplayName": "Access WebAppChapter5",
      "value": "user_impersonation"
    }
  ],
  "passwordCredentials": [],
  "preferredTokenSigningKeyThumbprint": null,
  "publisherName": "Developer Tenant",
  "replyUrls": [],
  "samlMetadataUrl": null,
  "servicePrincipalNames": [
    "https://localhost:44300/WebProjectChapter5",
    "e8040965-f52a-4494-96ab-0ef07b591e3f"
  ],
  "tags": [
    "WindowsAzureActiveDirectoryIntegratedApp"
  ]
}
```

I am sure you are not surprised to find `objectId` and `deletionTimestamp` here, too.

Notably missing are all the flags determining protocol behaviors at run time: `availableToOtherTenants`, `groupMembershipClaims`, `oauth2AllowImplicitFlow`, `oauth2AllowUrlPathMatching`, `oauth2-RequirePostResponse`, and `publicClient`. Other properties that don't directly make it in the form of properties in `ServicePrincipal` are `knownClientApplications` and `requiredResourceAccess`, both of which are properties that influence the consent process and the very creation of this `ServicePrincipal`. As you will see later on, `requiredResourceAccess` gets recorded in a different form—one that makes it easier for the directory to track down who in the tenant has actually been granted the necessary permissions to use the app.

Properties that do transfer as is from the `Application` to its corresponding `ServicePrincipal` are the `appId` (containing the all-important `client_id`), various optional URLs (`errorUrl`, `logoutUrl`, `samlMetadataUrl`), the settings used when listing the app in some UX (`displayName`, `homepage`), the exposed `appRoles` and `oauth2Permissions`, and finally the credentials `keyCredentials` and `passwordCredentials`. The presence of the credentials in the `ServicePrincipal` has important implications: it means that your code can use the same credentials defined in the `Application` and those will work on every `ServicePrincipal` in every tenant.

Here's a list of the brand-new properties:

- `appOwnerTenantId` This property carries the `tenantId` of the tenant where you'll find the `Application` object that was used as a blueprint for creating this `ServicePrincipal`—in this case, `developertenant.onmicrosoft.com`. If you search Chapter 6 for the GUID value shown in our example's `ServicePrincipal`, you'll find it everywhere.
- `publisherName` Another property meant to be used for describing the app in user interactions, `publisherName` stores the display name of the tenant where the original `Application` was defined. This represents the organization that published the app.
- `servicePrincipalNames` This property holds all the identifiers that can be used for referring to this application in protocol flows: as you might have noticed in the sample, it contains the union of the values in the `identifierUris` collection and the `appId` value from the `Application` object. The former is used for OAuth2 and OpenID Connect flows, the latter for WS-Federation, SAML, or OAuth2 bearer token resource access requests.
- `appRoleAssignmentRequired` Administrators can decide to explicitly name the user accounts that they want to enable for the user of the app and gate the token issuance on this condition. If `appRoleAssignmentRequired` is set to `true`, only the token requests coming from explicitly assigned users will be fulfilled. I'll talk more about this later in the chapter.
- `tags` This property is used mostly by the Azure portal to determine the type of application and how to present it in the administrative interface. Without going into fine detail, an empty tag collection results in the corresponding `ServicePrincipal` not being shown as one of the resources that can be requested by other applications.

## Consent and delegated permissions

---

Now that you know what application aspects are defined in the `Application` and `ServicePrincipal` objects, it's time to understand how these two entities are used in the application provisioning and consent flows.

You have learned that all it takes for provisioning an app in a tenant (creating a `ServicePrincipal` for that app in the tenant) is one user requesting a token by using the app coordinates defined in the `Application` object, successfully authenticating, and granting to the app consent to the permissions it requires. To get to the next level of detail, you must take into account what kind of user created the application in the first place, what permissions the applications requires, and what kind of user actually grants consent to the app and in what terms. There is an underlying rule governing the entire process, but that's pretty complicated. Instead of enunciating it here and letting you wrestle with it, I am going to walk you through various common scenarios and explain what happens. Little by little, we'll figure out how things work.

Initially, I'll scope things down to the case in which you are creating line-of-business apps—applications meant to be consumed by users from the same directory in which they were created. If your company has an IT department that creates apps for your company's employees, you know what kind of apps I am referring to. Once you have a solid understanding of how consent works within a single directory, I'll venture to the multitenant case, where you'll see more of the provisioning aspect. I'll stick to delegated permissions, but there are other kinds of permissions, like the things that an app can do independently of which user is signed in at the moment, but I'll defer coverage of those and describe the basics here.

### Application created by a nonadmin user

In Chapter 5 you followed instructions to create an application in Azure AD via the Azure portal. Did you create it while being signed in with a user who is a global administrator in your tenant? If not, that's perfect—the app you have is already in the state I'll describe in this section. If you did, you can choose to believe that my description is accurate—or you can create a new app, following the same instructions (very important!) but using a nonadmin user. Note: to be able to sign in with that account in the Azure portal, you might need to promote that user to coadmin of your Azure subscription.

As you have seen in the preceding section, creating one app via the Azure portal has the effect of creating both the `Application` object and the corresponding `ServicePrincipal`. What you haven't seen yet is how the directory remembers what permissions have been granted to the `ServicePrincipal` and to which user. The `Application` object enumerates the permissions it needs in the `requiredResourceAccess` collection, but those aren't present in the `ServicePrincipal`. Where are they?

Azure AD maintains another collection of entities, named `oauth2PermissionGrants`, which records which clients have access to which resources and with what permissions. Critically, `oauth2PermissionGrants` also records which users that consent is valid for.

For example, when you created the sample app in the Azure portal, Azure AD automatically granted consent for that app on behalf of your user. Alongside the `Application` and `ServicePrincipal`, the process also created the following `oauth2PermissionGrants` entry:

```
{
  "odata.metadata": "https://graph.windows.net/developertenant.onmicrosoft.com/$metadata#oauth2PermissionGrants",
  "value": [
    {
      "clientId": "29f565fd-0889-43ff-aa7f-3e7c37fd95b4",
      "consentType": "Principal",
      "expiryTime": "2015-11-21T23:31:32.6645924",
      "objectId": "_WX1KYkI_00qfz58N_2VtEnIMYJNhOpOkFrsIuF86Y8",
      "principalId": "13d3104a-6891-45d2-a4be-82581a8e465b",
      "resourceId": "8231c849-844d-4eea-905a-ec22e17ce98f",
      "scope": "UserProfile.Read",
      "startTime": "0001-01-01T00:00:00"
    }
  ]
}
```

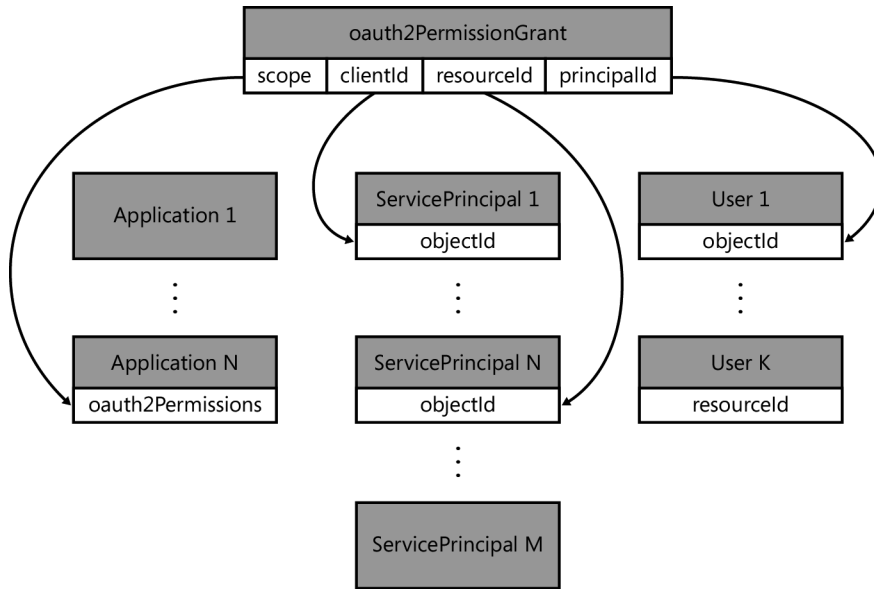


**Note** The query I used for retrieving this result was `https://graph.windows.net/developertenant.onmicrosoft.com/oauth2PermissionGrants?$filter=clientId+eq+'29f565fd-0889-43ff-aa7f-3e7c37fd95b4'`.

Let's translate that snippet into English. It says that the User with identifier `13d3104a-6891-45d2-a4be-82581a8e465b` (the `PrincipalId`) consented for the client `29f565fd-0889-43ff-aa7f-3e7c37fd95b4` (the `clientId`) to access the resource `8231c849-844d-4eea-905a-ec22e17ce98f` (the `resourceId`) with permission `UserProfile.Read` (the `scope`). Resolving references further, the client is our sample app, and the resource is the directory itself—more precisely, the Directory Graph API. Figure 8-3 shows how the consent for the first application user is recorded in the directory; Figure 8-4 shows how the `oauth2PermissionGrants` table grows as more users give their consent.



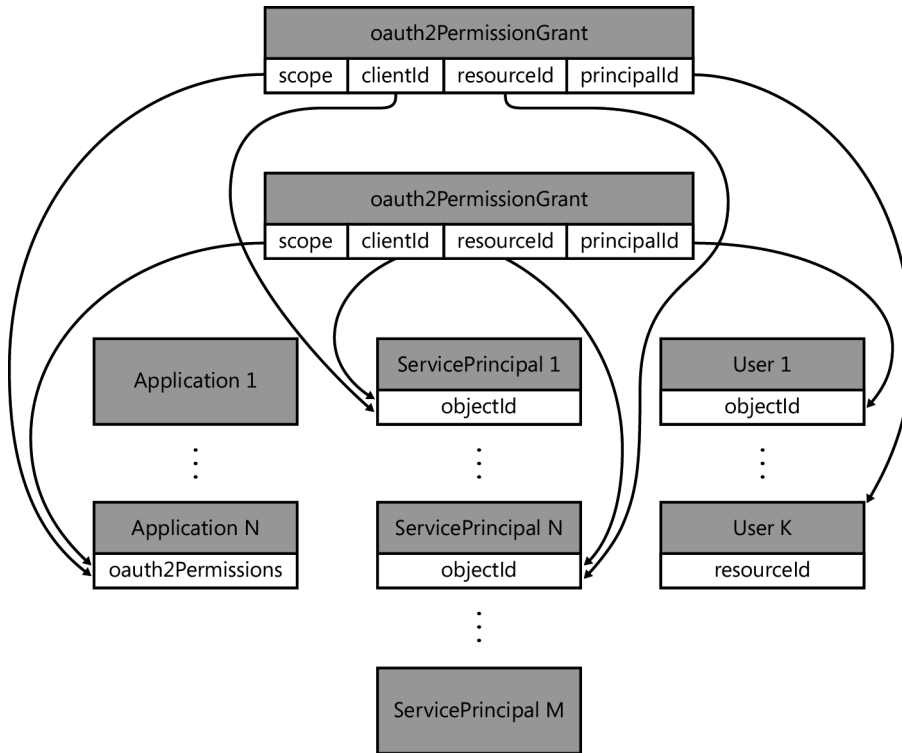
**Important** All the identifiers here refer to the `objectId` property of the respective entity they refer to. Given that `clientId` and `resourceId` ultimately refer to `ServicePrincipals`, it's easy to get confused and expect those values to represent the `appId`. But nope, it's the `objectId`. The `principalId` is the `objectId` property of the User object representing the user account used for consenting.



**FIGURE 8-3** The `oauth2PermissionGrant` recording in the directory that user 1 consented for the app represented by `ServicePrincipal 1` to access `ServicePrincipal N` with the permission stored in the property `scope`, in itself picked from one of the permissions exposed by the original `Application N` `oauth2Permissions` section.

When Azure AD receives a request for a token to be issued to the application defined here, it looks in the `oauth2PermissionGrants` collection for entries whose `clientId` matches the app. If the authenticated user has a corresponding entry, she or he will get back a token right away. If there's no entry, the user will see the consent prompt listing all the `requiredResourceAccess` permissions from the `Application` object. Upon successful consent, a new `oauth2PermissionGrant` entry for the current user will be created to record the consent. And so on and so forth.





**FIGURE 8-4** Subsequent consent operations create more `oauth2PermissionGrant` entries in the directory, one for each new user consenting for the application.

If you want to try, go ahead and launch the sample app again, but sign in as another user. This time, you will be presented with the consent page. Consent and then sign out. Sign in again with the new user: you will not be prompted for consent again. If you queried the directory (in the next chapter you'll learn how) to find all the `oauth2PermissionGrants` whose `clientId` matches the sample app, you'd see that there are now two entries, looking very much alike apart from the `principalId`, which would point to different users. Note that it doesn't matter whether your second user is an administrator or a low-privilege user; the resulting `oauth2PermissionGrant` will look just like the one described earlier when following this flow.

## Interlude: Delegated permissions to access the directory

One of the things you have learned in this chapter is that applications can declare the permissions that a client can request of them, via `oauth2Permissions`, as a way of partitioning the possible actions a user can perform over the resource represented by the app and to provide fine-grained access control over who can do what. As I've mentioned, in the next chapter you will learn how clients can actually take advantage of gaining such permissions; here, you're just studying how requesting and granting such permissions takes place.

Each and every resource protected by Azure AD works by exposing permissions—the Office 365 API, Azure management API, and any custom API all work that way. Covering all those would be a pretty hard task. Even ignoring the enormous surface I'd have to cover, chances are that the details would change multiple times from the time I'm writing and when you have this book in your hands. That said, I am going to describe in detail at least one resource: the directory itself. Like any other resource, Azure AD exposes a number of delegated permissions, which determine what actions your application is allowed to perform against the data stored in the directory. Such actions take the form of requests to embed information in issued tokens (what we have been working with until now) and reading or modifying directory data via API calls to the Graph API (what you'll see in the next chapter). You will likely have to deal with directory permissions in practically every app you write; hence, they're a great candidate for showing you how to deal with permissions in depth—well, except for the fact that they feature lots of exceptions, but you need to be aware of these anyway.

As of today, the directory itself is represented by a `ServicePrincipal` in your tenant. You already know both the `AppId` and the `ObjectId` of that principal, given that our sample app had to request at least the permission `UserProfile`. Read in order to sign users in. The `AppId`, `00000002-0000-0000-c000-000000000000`, comes from the `requiredResourceAccess` in the `Application` object representing our sample. The `ObjectId` of the `ServicePrincipal`, `8231c849-844d-4eea-905a-ec22e17ce98f`, comes from the `oauth2PermissionGrant` tracking the consent to our sample. The `objectId` is enough for crafting the resource URL referring to the Graph API `ServicePrincipal`: it's `https://graph.windows.net/developertenant.onmicrosoft.com/servicePrincipals/8231c849-844d-4eea-905a-ec22e17ce98f`.

I won't show the entire JSON for the `ServicePrincipal` here, as it contains a lot of stuff I want to cover later. But take a look at the `oauth2Permissions`, the collection of delegated permissions one client can request for interacting with the directory:

```
"oauth2Permissions": [
  {
    "adminConsentDescription": "Allows the app to create groups on behalf of the signed-in user and read all group properties and memberships. Additionally, this allows the app to update group properties and memberships for the groups the signed-in user owns.",
    "adminConsentDisplayName": "Read and write all groups",
    "id": "970d6fa6-214a-4a9b-8513-08fad511e2fd",
    "isEnabled": true,
    "type": "User",
    "userConsentDescription": "Allows the app to create groups on your behalf and read all group properties and memberships. Additionally, this allows the app to update group properties and memberships for groups you own.",
    "userConsentDisplayName": "Read and write all groups",
    "value": "Group.ReadWrite.All"  },
  {
    "adminConsentDescription": "Allows the app to read basic group properties and memberships on behalf of the signed-in user.",
    "adminConsentDisplayName": "Read all groups",
    "id": "6234d376-f627-4f0f-90e0-dff25c5211a3",
    "isEnabled": true,
    "type": "User",
    "userConsentDescription": "Allows the app to read all group properties and memberships on your behalf.",
```

```

    "userConsentDisplayName": "Read all groups",
    "value": "Group.Read.All"
  },
  {
    "adminConsentDescription": "Allows the app to read and write data in your company or school directory, such as users and groups. Does not allow user or group deletion.",
    "adminConsentDisplayName": "Read and write directory data",
    "id": "78c8a3c8-a07e-4b9e-af1b-b5ccab50a175",
    "isEnabled": true,
    "type": "Admin",
    "userConsentDescription": "Allows the app to read and write data in your company or school directory, such as other users, groups. Does not allow user or group deletion on your behalf.",
    "userConsentDisplayName": "Read and write directory data",
    "value": "Directory.Write"
  },
  {
    "adminConsentDescription": "Allows the app to have the same access to information in the directory as the signed-in user.",
    "adminConsentDisplayName": "Access the directory as the signed-in user",
    "id": "a42657d6-7f20-40e3-b6f0-cee03008a62a",
    "isEnabled": true,
    "type": "User",
    "userConsentDescription": "Allows the app to have the same access to information in your work or school directory as you do.",
    "userConsentDisplayName": "Access the directory as you",
    "value": "user_impersonation"
  },
  {
    "adminConsentDescription": "Allows the app to read data in your company or school directory, such as users, groups, and apps.",
    "adminConsentDisplayName": "Read directory data",
    "id": "5778995a-e1bf-45b8-affa-663a9f3f4d04",
    "isEnabled": true,
    "type": "Admin",
    "userConsentDescription": "Allows the app to read data in your company or school directory, such as other users, groups, and apps.",
    "userConsentDisplayName": "Read directory data",
    "value": "Directory.Read"
  },
  {
    "adminConsentDescription": "Allows the app to read the full set of profile properties of all users in your company or school, on behalf of the signed-in user. Additionally, this allows the app to read the profiles of the signed-in user's reports and manager.",
    "adminConsentDisplayName": "Read all users' full profiles",
    "id": "c582532d-9d9e-43bd-a97c-2667a28ce295",
    "isEnabled": true,
    "type": "Admin",
    "userConsentDescription": "Allows the app to read the full set of profile properties of all users in your company or school on your behalf. Additionally, this allows the app to read the profiles of your reports and manager.",
    "userConsentDisplayName": "Read all users' full profiles",
    "value": "User.Read.All"
  },
  {
    "adminConsentDescription": "Allows the app to read a basic set of profile properties of all users in your company or school on behalf of the signed-in user. Includes display name, first and last name, photo, and email address. Additionally, this allows the app to read basic

```

```

info about the signed-in user's reports and manager.",
  "adminConsentDisplayName": "Read all users' basic profiles",
  "id": "cba73afc-7f69-4d86-8450-4978e04ecd1a",
  "isEnabled": true,
  "type": "User",
  "userConsentDescription": "Allows the app to read a basic set of profile properties of
other users in your company or school on your behalf. Includes display name, first and last
name, photo, and email address. Additionally, this allows the app to read basic info about your
reports and manager.",
  "userConsentDisplayName": "Read all user's basic profiles",
  "value": "User.ReadBasic.All"
},
{
  "adminConsentDescription": "Allows users to sign in to the app, and allows the app to read
the profile of signed-in users. It also allows the app to read basic company information of
signed-in users.",
  "adminConsentDisplayName": "Sign in and read user profile",
  "id": "311a71cc-e848-46a1-bdf8-97ff7156d8e6",
  "isEnabled": true,
  "type": "User",
  "userConsentDescription": "Allows you to sign in to the app with your work account and let
the app read your profile. It also allows the app to read basic company information.",
  "userConsentDisplayName": "Sign you in and read your profile",
  "value": "User.Read"
}
],

```

Here's a quick description of each delegated permission listed, per their `Value` property. Please note that this list does change over time. Funny story: it changed a couple of weeks after I finished writing this chapter—I had to come back and revise much of what follows. In fact, the change is not fully complete, as the `ServicePrincipal` object shown above still shows some old values. The first four permissions described in what follows are the ones that Azure AD has offered since it started supporting consent as described in this book; the last four are brand-new and likely to be less stable. Wherever appropriate, I will hint at the old values so that if you encounter code based on older strings, you can map it back to the new permissions. Chances are the list will change again: please keep an eye on the permissions documentation, currently available at <https://msdn.microsoft.com/Library/Azure/Ad/Graph/howto/azure-ad-graph-api-permission-scopes>.

### ***User.Read (was UserProfile.Read)***

This is the permission that each app needs to authenticate users. Applications created in the Azure portal and Visual Studio are configured to automatically request this permission, which is why you don't see it mentioned in the UI you use for creating apps in either tool.

Besides the ability to request a token containing claims about the incoming user, this permission grants to the app the ability to query the Graph API for information about the currently signed-in user.

As you've experienced, this permission can be granted by nonadmin users. That is confirmed by the `type` property of value `User` in the permissions declaration.

### ***Directory.Read.All (was Directory.Read)***

As the name implies, obtaining this permission allows one application to read via the Graph API (I'll stop saying that; just assume that's what you use to interact with the directory) the content of the directory tenant of the user that is currently signed in.

Here's the first exception. In the general case, `Directory.Read` is an admin-only permission: only an admin user can consent to it. However, if the application is a web app (as opposed to a native client) defined in tenant A, and the user being prompted for consent is also from A, `Directory.Read` behaves like a `User-type` permission, which is to say that even a nonadmin user can consent to it. For the scenario we have been considering until now—app developer and app users are from the same tenant—this is effectively a `User-type` permission. When we consider the case in which the app is available to other tenants, you'll see that an app created in A that is requesting `Directory.Read` and being accessed by a user from B will be provisioned in B only if that user happens to be an administrator.

### ***Directory.ReadWrite.All (was Directory.Write)***

Once again, the name is self-explanatory: this permission grants to the app the ability to read, modify, and create directory data. No exceptions this time; only administrator users can consent to `Directory.Write`.

### ***Directory.AccessAsUser.All (was user\_impersonation)***

This permission, which today is surfaced in the Azure portal under the label "Access the directory as the signed-in user," allows the application to impersonate the caller when accessing the directory, inheriting his or her permissions. That is a pretty powerful thing to do, which is why for web applications this permission can be granted only by an admin user.

As a side note, for native applications, this permission behaves like a `User` permission instead. A native app does not have an identity per se, and it is already doing the direct user's bidding anyway. It stands to reason that the app should be able to do what the user is able to do, just as happens on-premises when a classic native client (say Word or Excel) can or cannot open a document from a network share depending on whether the user has the correct permissions on that folder.

### ***User.ReadBasic.All***

You can think of this permission as the minimum requirement allowing an app to enumerate all users from a tenant. Namely, `User.ReadBasic.All` will give access to the user attributes `displayName`, `givenName`, `surname`, `mail` and `thumbnailPhoto`. Anything beyond that requires higher permissions.

### ***User.Read.All***

This is an extension of `User.ReadBasic.All`. This permission allows an app to access all the attributes of `User`, the navigation properties manager, and `directReports`. `User.Read.All` can be exercised only by admin users.

## Group.Read.All, Group.ReadWrite.All

These new permissions are still in preview at this point, so I hesitate to give too detailed a description here. The idea is that groups and group membership are important information and deserve their own permissions so that access can be requested and granted explicitly. `Group.Read.All` allows an app to read the basic profile attributes of groups and the groups they are a member of. `Group.ReadWrite.All` allows an app to access the full profile of groups and to change the hierarchy by creating new groups and updating existing ones. Both permissions alone won't grant access to the users in the groups—to obtain that, the app also needs to request some `User.Read*` permission.

As usual, it's important to remember that scopes don't really add to what a user can do: an application obtaining `Group.ReadWrite.All` will only be able to manipulate the groups owned by the user granting the delegation to the app.

Table 8-1 summarizes how the out-of-the-box Azure AD permissions work. I've added a column for the permission identifier, which I find handy so that when I look at the `Application` object, which uses only opaque IDs, I know what permission the app is actually requesting. Let me stress that there's no guarantee these won't change in the future, so please use them advisedly.

**TABLE 8-1** A summary of the Azure AD permissions for accessing the directory.

| Permission description in the Azure portal | Identifier                           | Scope value                | Type  |
|--|--------------------------------------|----------------------------|---|
| Sign in and read user profile              | 311a71cc-e848-46a1-bdf8-97ff7156d8e6 | User.Read                  | User  |
| Read directory data                        | 5778995a-e1bf-45b8-affa-663a9f3f4d04 | Directory.Read.All         | Admin (except for users from the tenant where the Application is defined) |
| Read and write directory data              | 78c8a3c8-a07e-4b9e-af1b-b5ccab50a175 | Directory.ReadWrite.All    | Admin   |
| Access the directory as the signed-in user | a42657d6-7f20-40e3-b6f0-cee03008a62a | Directory.AccessAsUser.All | Admin (except native clients)   |
| Read all users' basic profiles             | cba73afc-7f69-4d86-8450-4978e04ecd1a | User.ReadBasic.All         | User  |
| Read all users' full profiles              | c582532d-9d9e-43bd-a97c-2667a28ce295 | User.Read.All              | Admin   |
| Read all groups                            | 6234d376-f627-4f0f-90e0-dff25c5211a3 | Group.Read.All             | Admin   |
| Read and write all groups                  | 970d6fa6-214a-4a9b-8513-08fad511e2fd | Group.ReadWrite.All        | Admin   |

Now that you have some permissions to play with, let's get back to the exploration of how consent operates.

## Application requesting admin-level permissions

Let's say that your application needs the ability to modify data in the directory. You might be surprised to learn that you can create such an application even with a nonadmin user: you'll simply not be able to use it at run time.



**Note** If you are keeping track of the identifiers in the JSON, technically I could modify the app we've been working on so far, but for the sake of clarity I'll create a new one.

Go back to the Azure portal, sign in as a nonadmin user, and go through the usual application creation flow. Once the app is created, head to the Configure tab and scroll all the way to the bottom of the page. As of today, you'll find a section labeled Permissions To Other Applications, already containing one entry for Azure Active Directory—specifically, the default delegated permission Sign In And Read User Profile. Figure 8-5 shows you the UI at the time of writing, but as usual you can be sure there will be something different (but I hope functionally equivalent) by the time you pick up the book.

REPLY URL:

permissions to other applications

Windows Azure Active Directory | Application Permissions: 0 | Delegated Permissions: 1

- Read and write all groups
- Read all groups
- Read and write directory data
- Access the directory as the signed-in user
- Read directory data
- Read all users' full profiles
- Read all users' basic profiles
- Sign in and read user profile

Add application

**FIGURE 8-5** The application permission selection UI in the Azure portal (fall 2015).

You'll also see an ominous warning: "You are authorized to select only delegated permissions which have personal scope." Today that isn't actually the case. Select Read And Write Directory Data, and then click Save.

You'll receive a warning that the portal was unable to update the configuration for the app, but that's only partially true. If you go take a look at the Application, you'll see that it was correctly updated. Here is its `requiredResourceAccess` section:

```
"requiredResourceAccess": [
{
  "resourceAppId": "00000002-0000-0000-c000-000000000000",
  "resourceAccess": [
    {
      "id": "78c8a3c8-a07e-4b9e-af1b-b5ccab50a175",
      "type": "Scope"
    },
    {
      "id": "311a71cc-e848-46a1-bdf8-97ff7156d8e6",
      "type": "Scope"
    }
  ]
}
]
```

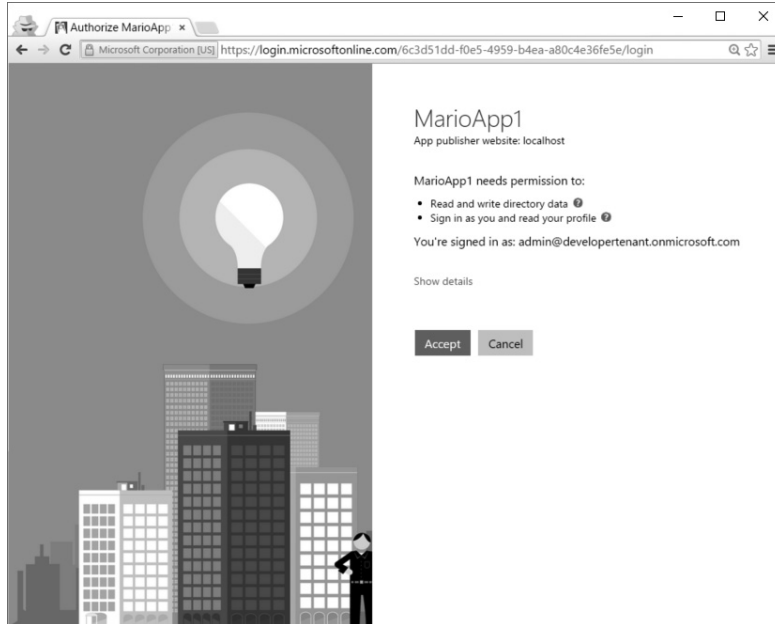
Thanks to our magical Table 8-1, we know those to be the correct permissions.

The part that the portal was *not* able to add was the `oauth2PermissionGrant` that would allow the current (nonadmin) user to have write access to the directory. If you list the `oauth2PermissionGrants` of the `ServicePrincipal`, you'll find only the original entry for `User.Read`.

That entry is the reason why, if you try to sign in to the app as the user who created it, you will succeed: the directory sees that entry, and that's enough to not show the consent prompt and issue the requested token. However, if after you sign in, your app attempts to get a token for calling the Graph, the operation would fail.

If you launch the application again and try to sign in as any other nonadmin user, instead of the consent prompt you'll receive an error along the lines of "AADSTS90093: Calling principal cannot consent due to lack of permissions," which is exactly what you should expect.

Finally, launch the app again and try to sign in as an administrator. You will be presented with the consent page as in Figure 8-6, just as expected.



**FIGURE 8-6** The consent prompt presented to an admin user.

Grant the consent—you'll find yourself signed in to the application. That done, take a look at what changed in `oauth2PermissionGrants`:



```

{
  "odata.metadata": "https://graph.windows.net/developertenant.onmicrosoft.com/$metadata#oauth2
PermissionGrants",
  "value": [
    {
      "clientId": "725a2d9a-6707-4127-8131-4f9106d771de",
      "consentType": "Principal",
      "expiryTime": "2016-02-26T18:17:06.8442687",
      "objectId": "mi1acgdnJ0GBMU-RBtdx3knIMYJNhOpOkFrsIuF86Y_VUmVPfKg_R6aK4EVKgQSW",
      "principalId": "4f6552d5-a87c-473f-a68a-e0454a810496",
      "resourceId": "8231c849-844d-4eea-905a-ec22e17ce98f",
      "scope": "Directory.Write UserProfile.Read",
      "startTime": "0001-01-01T00:00:00"
    },
    {
      "clientId": "725a2d9a-6707-4127-8131-4f9106d771de",
      "consentType": "Principal",
      "expiryTime": "2016-02-26T00:50:43.3860871",
      "objectId": "mi1acgdnJ0GBMU-RBtdx3knIMYJNhOpOkFrsIuF86Y9KENMTkwjSRaS-g1gajkZb",
      "principalId": "13d3104a-6891-45d2-a4be-82581a8e465b",
      "resourceId": "8231c849-844d-4eea-905a-ec22e17ce98f",
      "scope": "UserProfile.Read",
      "startTime": "0001-01-01T00:00:00"
    }
  ]
}

```

There's a new entry now, representing the fact that the admin user consented for the app to have `UserProfile.Read` and `Directory.Write` permissions. As discussed earlier, by the time you read this, those scopes will likely have their new values—`User.Read` and `Directory.ReadWrite.All`—but it is really exactly the same semantic.

Note that this did not change the access level for anybody but this particular admin user. If you try to sign in as a nonadmin user (other than the app's creator), you'll still get error AADSTS90093.

## Admin consent

If the consent styles you've encountered so far were the only ones available, you'd have a couple of serious issues:

- Each and every user, apart from the application developer, would need to consent upon their first use of the app.
- Only admin-level users would be able to consent for applications requiring more advanced access to the directory, even when a user did not plan to exercise those higher privileged capabilities.

Both issues would limit the usefulness of Azure AD. Luckily, there's a way of consenting to applications that results in a blanket grant to all users of a tenant, all at once, and regardless of the access level requested. That mechanism is known as *admin consent*, as opposed to user consent, which you've been studying so far. Achieving admin consent is just a matter of appending to the request to the authorization endpoint the parameter `prompt=admin_consent`.

## Scopes can't grant to the app more power than their user has!

I want to make sure you don't fall for a common misconception here. Scopes are a way of delegating to the app some of the capabilities of their current user. In the most extreme case, this means that an app can be as powerful as its current user (full user impersonation). What can never happen via delegated permissions is that an app can do more than what its user can. If a user cannot write to the directory, the fact that the app obtains `Directory.ReadWrite.All` does not mean that such user can now use the app for writing to the directory! What that scope really means is that if the current user of the app has that capability, the app has that capability, too. If the user does not have that capability, he or she cannot delegate it to the application. As you will see later, applications can have their own permissions (as opposed to delegated permissions) that are independent from their current user and that can be used when the app needs to perform things that would not normally be within the possibilities of its users.

Let's give it a try and see what happens. From Chapter 7, you now know how to modify authentication requests by adding the change you want to the `RedirectToIdentityProvider` notification. In a real app, you would add some conditional logic to weave this parameter in only at the time of first access, but for this test you can go with the brute-force solution in which you add it every time.

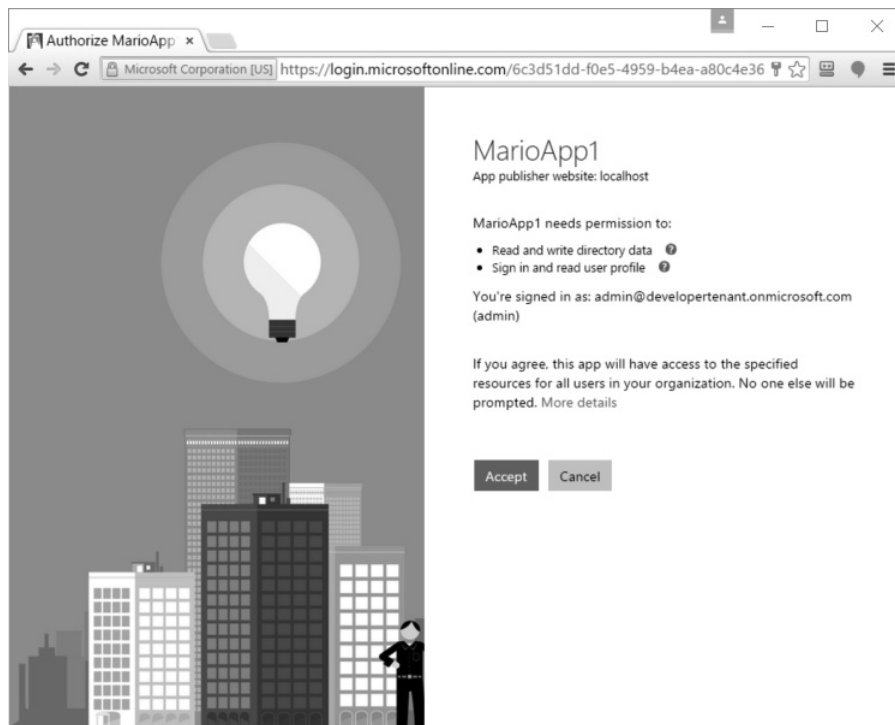


**Important** Here I am adding `Prompt=admin_consent` in the sign-in request for the sake of simplicity, but you would never do that in a production application without at least some conditional logic. In fact, more often than not, you would not include it in the sign-in action but wire it up to a dedicated sign-up action instead. Including `Prompt=admin_consent` in a request will result in the consent being shown to the user, regardless of the past consent history. You want to show this only when needed, and that's only the first time. Wire it up to some specific action in your app, like sign-up, onboarding, or any other label that makes sense for your application.

Here's the code:

```
public static Task RedirectToIdentityProvider(RedirectToIdentityProviderNotification<OpenIdConnectMessage,
    OpenIdConnectAuthenticationOptions> notification)
{
    notification.ProtocolMessage.Prompt = "admin_consent";
    return Task.FromResult(0);
}
```

After you've added that code, hit F5 and try signing in. You will be prompted by a dialog similar to the one shown in Figure 8-7.



**FIGURE 8-7** The admin consent dialog.

Superficially, the dialog in Figure 8-7 looks a lot like the one shown in Figure 8-6, but there is a very important difference! The dialog shown when admin consent is triggered has new text, which articulates the implications of granting consent in the admin consent case: “If you agree, this app will have access to the specified resources for all users in your organization. No one else will be prompted.”

Click OK—you’ll end up signing in as usual. The app will look the same, but its entries in the directory underwent a significant change. Once again, take a look at the `ServicePrincipal’s` `oauth2PermissionGrants`:

```
{
  "odata.metadata": "https://graph.windows.net/developertenant.onmicrosoft.com/$metadata#oauth2PermissionGrants",
  "value": [
    {
      "clientId": "725a2d9a-6707-4127-8131-4f9106d771de",
      "consentType": "AllPrincipals",
      "expiryTime": "2016-02-27T00:38:03.4045842",
      "objectId": "milacgdnJ0GBMU-RBtdx3knIMYJNhOp0kFrsIuF86Y8",
      "principalId": null,
      "resourceId": "8231c849-844d-4eea-905a-ec22e17ce98f",
      "scope": "Directory.Write UserProfile.Read",
      "startTime": "0001-01-01T00:00:00"
    },
  ],
}
```

```

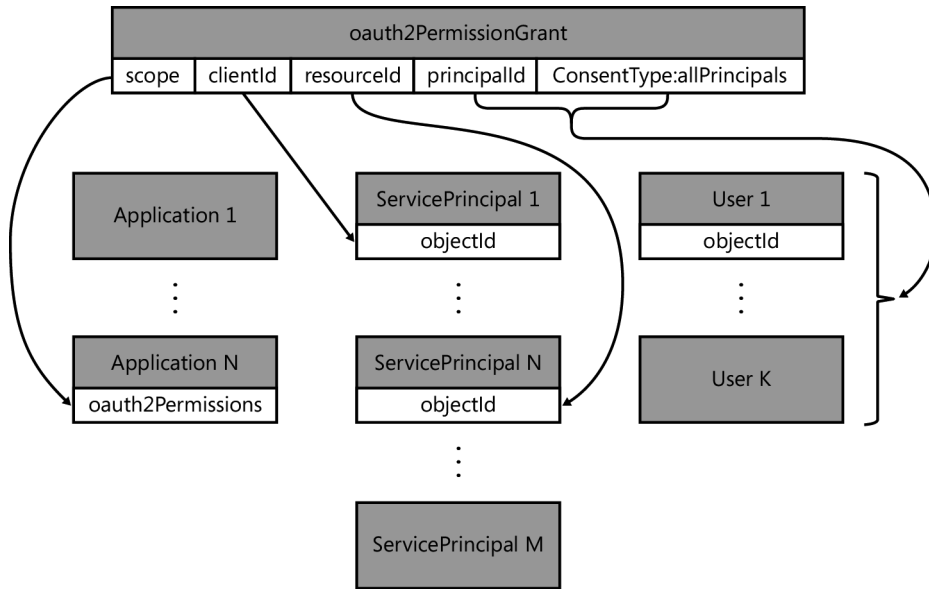
{
  "clientId": "725a2d9a-6707-4127-8131-4f9106d771de",
  "consentType": "Principal",
  "expiryTime": "2016-02-26T18:17:06.8442687",
  "objectId": "mi1acgdnJ0GBMU-RBtdx3knIMYJNhOpOkFrsIuF86Y_VUmVPfKg_R6aK4EVKqQSw",
  "principalId": "4f6552d5-a87c-473f-a68a-e0454a810496",
  "resourceId": "8231c849-844d-4eea-905a-ec22e17ce98f",
  "scope": "Directory.Write UserProfile.Read",
  "startTime": "0001-01-01T00:00:00"
},
{
  "clientId": "725a2d9a-6707-4127-8131-4f9106d771de",
  "consentType": "Principal",
  "expiryTime": "2016-02-26T00:50:43.3860871",
  "objectId": "mi1acgdnJ0GBMU-RBtdx3knIMYJNhOpOkFrsIuF86Y9KENMTkwjSRaS-g1gajkZb",
  "principalId": "13d3104a-6891-45d2-a4be-82581a8e465b",
  "resourceId": "8231c849-844d-4eea-905a-ec22e17ce98f",
  "scope": "UserProfile.Read",
  "startTime": "0001-01-01T00:00:00"
}
]
}

```



**Note** As I mentioned earlier in this chapter, `Directory.Write` and `UserProfile.Read` will change to `Directory.ReadWrite.All` and `User.Read`.

I highlighted the new entry for you: it has a `consentType` of `AllPrincipals`, as opposed to the usual `Principal`. Furthermore, its `principalId` property does not point to any user in particular; it just says `null`. This tells Azure AD that the application has been granted a blanket consent for any user coming from the current tenant. To prove that this is really the case, sign out from the app, stop it in Visual Studio, comment out the code you added for triggering admin consent, and start the app again. Sign in as a third user from the same tenant, one that you have never used before with this app. Figure 8-8 shows a visual summary of this `oauth2PermissionGrant` configuration.



**FIGURE 8-8** An `oauth2PermissionGrant` recording admin consent enables the app to operate with the requested scope with all users of a tenant at once.

After the credential gathering, you'll find yourself signed in right away, with no consent prompt of any form.

## Application created by an admin user

What happens when you sign in to the Azure portal as an admin user and you create an app in Azure AD? The portal creates the same list of entities: an `Application`, its `ServicePrincipal`, and an `oauth2PermissionGrant`. The difference from the nonadmin case is that the `oauth2PermissionGrant` for an app created by an admin looks exactly like the one you observed as an outcome of the admin consent flow: it includes `consentType allPrincipals`, which means that every user in the tenant can instantly get access to the application.



**Note** The creation of the `ServicePrincipal` and the associated grant is at the origin of the peculiar behavior of native apps created via the Azure portal by an admin. That is the only case in which a native app does not trigger consent for all users in a tenant. In all other cases, Azure AD today does not record consent for native apps in the directory, storing it in the refresh token instead—which means that each new native app instance running on a different device will prompt its user for consent regardless of its past consent history. This is really out of scope for this book, but given that you have the concept fresh in your mind, I thought I'd share this tidbit.

## Multitenancy

How to develop apps that can be consumed by multiple organizations is such a large topic that for some time I wondered whether I should devote an entire chapter to it. I ultimately decided against that. Even if this is going to be a very large section, it still is a logical extension of what you have been studying so far in this chapter.

The first part of this section will discuss how Azure AD enables authentication flows across multiple tenants, and how you can generalize what you have learned about configuring the Katana middleware to the case in which users are sourced from multiple organizations.

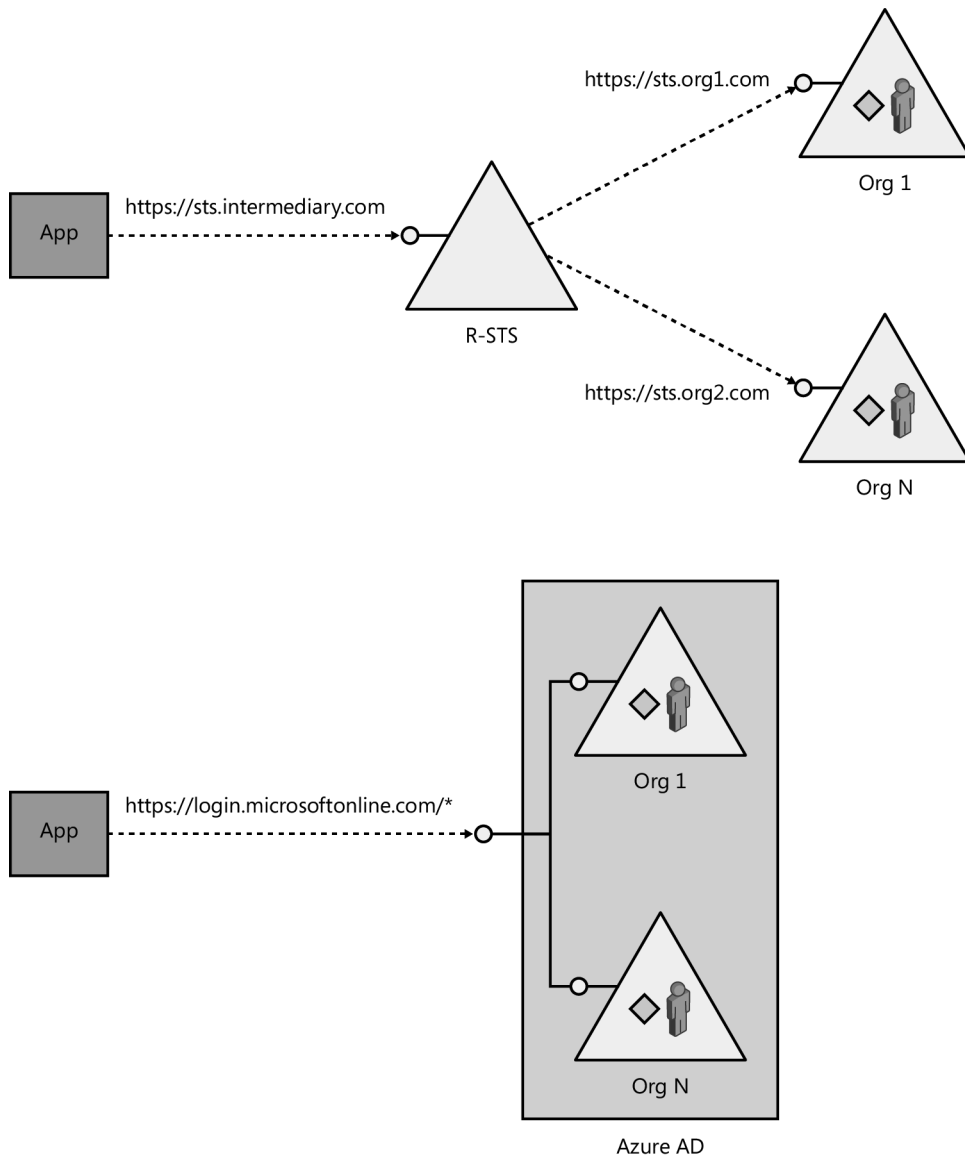
The second part will go back to the application model proper, showing you what happens to the directory data model when your app triggers consent flows across tenants.

### Azure AD as a parametric STS: The common endpoint

Ironically, if you are a veteran of federation protocols, you are at the highest risk of misunderstanding how Azure AD handles multitenancy. The approach taken here is very different from the classic solutions that preceded it, and I have to admit that I myself needed some time to fully grok it.

In traditional claims-based protocols such as SAML and WS-Federation, the problem of enabling access to one application from multiple IdPs has a canonical solution. It entails introducing one intermediary STS (often referred to as resource STS, R-STS or RP-STS) as the authority that the application trusts. In turn, the intermediate STS trusts all the IdPs that the application needs to work with—assuming the full burden of establishing and maintaining trust, implementing whatever protocol quirks each IdP demands. This is a very sensible approach, which isolates the application itself from the complexities of maintaining relationships with multiple authorities. It is also likely the best approach when you don't know anything about the IdPs you want to connect to, apart from the protocol they implement and the STS metadata they publish. ADFS, Azure Access Control Services (ACS), and pretty much any STS implementation supports this approach.

If you restrict the pool of possible IdPs to only the ones represented by a tenant in Azure AD, however, you have far more information than that, and as you'll see in the following, this removes the need to have an intermediary in the picture. Although each administrator retains full control over her or his own tenant, all tenants share the same infrastructure—same protocols, same data model, same provisioning pipes. Focusing on endpoints in particular (recall their description from Chapter 3), rather than a collection of STSs for each of its tenants, Azure AD can be thought of like a giant parametric STS, where each tenant is expressed by instantiating its ID in the right segment of the issuance endpoint. Figure 8-9 compares the R-STS approach with the multitenant pattern used by Azure AD.



**FIGURE 8-9** The R-STS brokered trust pattern and the parametric STS pattern. Besides allowing for directory queries that would be impossible via federation alone, the latter makes it possible to automate application provisioning and trust establishment.

In the hands-on chapters, you've experienced directly how the endpoint pattern `https://<instance>/<tenant>/<protocol-specific-path>` can be modulated to indicate tenant-specific token-issuance endpoints, sign-out endpoints, metadata document endpoints, and so on. You have also seen how the Katana middleware leverages those endpoints for tying one application to one specific tenant. For example, in Chapter 6 you saw how the metadata document published at `https://login.microsoftonline.com/DeveloperTenant.onmicrosoft.com/.well-known/openid-configuration` (which,

by the way, is equivalent to <https://login.microsoftonline.com/6c3d51dd-f0e5-4959-b4ea-a80c4e-36fe5e/well-known/openid-configuration>, where the GUID is the corresponding tenantID) asserts that tokens issued by that tenant will carry an `iss(uer)` claim value of <https://sts.windows.net/6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e/>. In Chapter 7, you saw how that information is used by the Katana middleware to ensure that only tokens coming from that tenant (that is, carrying that `iss` value) will be accepted. That's all well and good, and exactly what you want for line-of-business applications and single-tenant apps in general.

You can repeat the same reasoning for all tenants: all you need to do is instantiate the right domain (or tenantID) in the endpoints paths.

Azure AD makes it possible to deal with multitenant scenarios by exposing a particular endpoint, where the tenant parameter is not instantiated up front. There is a particular value, `common`, that can be instantiated in endpoints in lieu of a domain or tenantID. By convention, that value tells Azure AD that the requestor is not mandating any particular tenant—any Azure AD tenant will do.



**Very important: `Common` is not a tenant.** It is just an artifact used for constructing Azure AD endpoints when the tenant to be used is not known yet. This is a crucial point to keep in mind at all times when working with multitenant solutions, or you'll end up baking assumptions into your app that will inevitably turn out to be false and create all sorts of issues that are hard to debug.

When the endpoint being constructed is one that would serve authentication UI, as is the case for the OAuth2 authorization endpoints, the user is presented with a generic Azure AD credentials-gathering experience. As the user enters his or her credentials, the account he or she chooses will indirectly determine a specific tenant—the one the account belongs to. That will resolve the ambiguity about which tenant should be used for the present transaction, concluding the role of `common` in the flow. The resulting code or token will look exactly as it would have had it been obtained by specifying the actual tenant instead of `common` to begin with. In other words, whether you start the authentication flow using <https://login.microsoftonline.com/common/oauth2/authorize> or <https://login.microsoftonline.com/6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e/oauth2/authorize> for an OpenID Connect sign-in flow, if at run time you sign in with a user from the tenant with ID `6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e`, the resulting token will look the same, with no memory of what endpoint path led to its issuance. That should make it even clearer that `common` is not a real tenant: it's just an endpoint sleight of hand for late binding a tenant, if you will.

Now comes the fun part. Upon learning about the `common` endpoint, the typical (and healthy) developer reaction is "Awesome! Let me just change the OpenID Connect middleware options as shown here, and I'll be all set!"

```
app.UseOpenIdConnectAuthentication(  
    new OpenIdConnectAuthenticationOptions  
    {  
        ClientId = "c3d5b1ad-ae77-49ac-8a86-dd39a2f91081",  
        Authority = "https://login.microsoftonline.com/common",
```



Let's say that you do just that, and then you hit F5 and, just for testing purposes, use the same account you used successfully earlier—the one from the same tenant where the app was defined in the first place. Well, if you do that—surprise! The app won't work. Sure, upon sign-in you will be presented with your credential-gathering and consent experience, but the app won't accept the issued token. If you dig in a bit, as you learned in Chapter 7, you'll discover that the token failed the issuer validation test.

Recall the `id_token` validation logic from Chapter 7, and the comment about how the discovery document of each tenant establishes what `iss` value an app should expect. If your app is initialized with a tenant-specific endpoint, it will read from the metadata the tenant-specific issuer value to expect; but if it is initialized with `common`, what issuer value is it going to get? I'll save you the hassle of visiting <https://login.microsoftonline.com/common/well-known/openid-configuration> yourself: the discovery doc says "`issuer`": "`https://sts.windows.net/{tenantid}/`". No real tenant will ever issue a token with that value, given that it's just a placeholder, but the middleware does not know that. That's the value that the metadata asserts is going to be in the `iss` claim, and the default logic will refuse anything carrying a different value.



**Note** What about all the other values in the discovery doc? Issuer is the only problematic one, everything else (including keys, as you have seen in Chapter 6) is shared by all tenants.

This simply means that the default validation logic cannot work in case of multitenancy. What should you do instead? You already saw the main strategies for dealing with this in Chapter 7, although at the time I could not fully discuss the multitenant case. I recommend that you leaf back a few pages to get all the details, but just to summarize the key points here:

- If you have your own list of tenants that your application should accept, you have two main approaches. If the list is short and fairly static, you can pass it in at initialization time via `TokenValidationParameters.ValidIssuers`. If the list is long and dynamic, you can provide an implementation for `TokenValidationParameters.IssuerValidator` where you accommodate for whatever logic is appropriate for your case.
- If the decision about whether the caller should be allowed to get through is not strictly tied to the tenant the caller comes from, you can turn off issuer validation altogether by setting `TokenValidationParameters.ValidateIssuer` to `false`. You should be sure that you do add your own validation logic; for example, in the `SecurityTokenValidated` notifications or even in the app (custom authorization filters, etc.). Otherwise, your app will be completely open to access by anybody with a user in Azure AD. There are scenarios where this might be what you want, but in general, if you are protecting your app with authentication, that means that you have something valuable to gate access to. In turn, that might call for you to verify whether the requestor did pay his monthly subscription or whatever other monetization strategy you are using—and usually that verification boils down to checking the issuer or the user against your own subscription list.

Now that you know how Azure AD multitenancy affects the application's code, I'll go back to how consent, provisioning, and the data model are influenced.

## Consenting to an app across tenants

The section about the `Application` object earlier in this chapter, and specifically the explanation of the `availableToOtherTenants` property, already anticipated most of what you need to know about creating multitenant applications. All apps are created for being used exclusively within their own tenant, and only a tenant admin can promote an app to be available across organizations. Today, this is done by flipping a switch labeled “Application is multi-tenant” on the Configuration page of the application on the Azure portal, and this has the effect of setting the `availableToOtherTenants` app property to true. Also, an app is required to have an App ID Uri (one of the elements in the `identifierUris` collection in the `Application` object) whose host portion corresponds to a domain registered for the tenant. In the sample I have been using through the last couple of chapters, that means that you’d need to set the App ID Uri to something like `https://developertenant.onmicrosoft.com/MarioApp1`.

Let’s say that you signed in to the Azure portal and modified your app entry to be multitenant. Let’s also say that you modified your app code to correctly handle the validation for tokens coming from multiple organizations. Let’s give the app a spin by hitting F5.



**Note** If you promote the app you have been using in this chapter until now, be sure to comment out the logic that triggers the admin consent (for now). Consequently, make sure also that the app does not request any admin-only permissions.

### In case you did not code your validation logic yet

If you are just experimenting and didn’t set up your multitenant validation code yet, here’s the code you can use for turning off issuer validation while you play with the walk-through in this chapter:

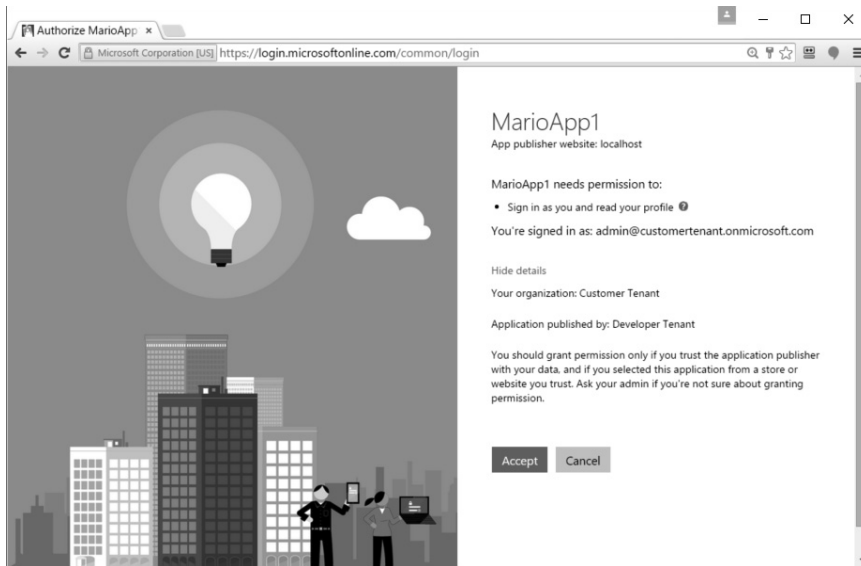
```
app.UseOpenIdConnectAuthentication(  
    new OpenIdConnectAuthenticationOptions  
    {  
        ClientId = "c3d5b1ad-ae77-49ac-8a86-dd39a2f91081",  
        Authority = "https://login.microsoftonline.com/common",  
        TokenValidationParameters = new System.IdentityModel.Tokens.TokenValidationParameters  
        {  
            ValidateIssuer = false,  
        },  
    },  
);
```

I cannot stress this enough: you should not go into production with the issuer validation logic disabled unless you have also added your own validation.

Once the app is running, click the Sign In link, but this time sign in with a user from a different Azure AD tenant. As explained in Chapter 3, in the section “Getting Azure Active Directory,” any Azure subscriber can create a number of Azure AD tenants, create users and apps in them, and so on. If you belong to a big-ish organization, you likely already did this in creating your development

tenant, as that's the best way of experimenting with admin-only features. If you already have a second tenant and an account in it, great! If you don't, create one tenant, create a user in it, then come back and pick up the flow from here.

Upon successful sign-in, you'll be presented with the consent page. As you can see in Figure 8-10, the consent page presents some important differences from the single-tenant case. For one, the tenant where the Application object was originally created is prominently displayed as the publisher. Moreover, there's now text telling you to consider whether you trust the publisher of the application. This is serious stuff—if you give consent to the wrong application for the wrong permissions, the damage to your own organization could be severe. That's why only admins can publish apps for multiple organizations, and that's why even the simple Directory.Read permission requires admin consent when it's requested by a multitenant app.



**FIGURE 8-10** The consent prompt for a multitenant page.

At the beginning of this chapter, you encountered a description of what happens in the tenants for this exact consent scenario: the Application object in the original development tenant is used as a blueprint for creating a ServicePrincipal in the target tenant. In fact, if you query the Applications collection in the target tenant (you'll learn how to do this in the next chapter), you'll find no entries with the ClientId of your application—but you will find a ServicePrincipal with that ClientId. From what you have learned a few pages ago, you know that if you look into the collection of OAuth2PermissionGrants for that ServicePrincipal, you will find an entry recording the consent of that particular user for this app and the permissions it requires. The principles of admin consent apply here as well: if you want the admin of your prospective customer tenants to be able to grant a blanket consent for all of his or her users, provide a way for your app to trigger an admin consent request.

## Changing consent settings

I touched on this earlier, but it's worth stressing that the list of permissions an app requires isn't very dynamic. More concretely, say that your application initially declares a certain list of permissions in its `requiredResourceAccess`, and some users in a few tenants consent to it. Say that after some time you decide to add a new permission. That change in the `Application` object in your development tenant will not affect the existing `oauth2PermissionGrants` attached to the `ServicePrincipals` that have been created at consent time. With this version of Azure AD, the only way of reflecting the new permission set for a given app in a tenant is to revoke the existing consent (typically done by the user visiting `myapps.microsoft.com`, the Office 365 portal, or any other means that might be available when you read this book) and ask for consent again.

This is less than ideal, especially if you consider that Azure AD offers you no way of warning your users that something changed—you have to handle that in your own app or subscription logic. The good news is that the next version of the Azure AD application model will allow for dynamic consent, solving this issue once and for all.

The last section discussed at length the consent framework used for driving delegated permissions assignment to applications. That is a super important aspect of managing application capabilities, but it is far from the only one. The next section will continue to explore how Azure AD helps you to control how users and applications have access to the directory itself, and to each other.

## App user assignment, app permissions, and app roles

---

This section describes a set of Azure AD features that seem unrelated but are in fact all implemented through the same primitive: the role. Here's the list of features I'll cover.

- **App user assignment** The ability to explicitly define which users should be allowed to get a token for a certain application, at the exclusion of everybody else.
- **App-level permissions** The ability to expose (and request) permissions that can be assigned to applications themselves, as opposed to the users of the apps.
- **App roles** The ability for developers to define application-specific roles, which can be used by administrators to establish in which capacity users can access an application.

All these features give you even more control over who can access your application and how.

### App user assignment

By default, every user in a tenant can request a token from any app. Whether the requested token will actually be issued depends on the outcome of user authentication, consent, and considerations of user versus admin permissions, as I've discussed in the preceding sections.

Azure AD offers the possibility for an administrator to restrict access to one application to a specific set of handpicked accounts. In terms of today's experience, an administrator can navigate to the Azure management portal at <https://manage.windowsazure.com>, select the target tenant, navigate to the appropriate app entry, select the Configure tab, and flip the setting for "User assignment required to access app" to On.

Given that this feature is related to specific instances of the app in specific tenants, the knobs used to control it are not in the `Application` object but in the `ServicePrincipal` and associated entities in the target tenant. You already encountered the `ServicePrincipal` property `AppRoleAssignmentRequired`: flipping the switch in the portal has the effect of setting this property to true.

The Users tab in the application entry in the Azure portal lists all the users that are assigned to the application. From now on, *no user not on that list* can successfully request a token for the application. If you flip the switch for one of the apps you've been playing with in the preceding section, you'll see that all the users that already gave consent for the app are present in the list. Every time a user gives consent to the app, Azure AD adds an entry to a list of `AppRoleAssignment`, an entity I haven't yet discussed. Here's how one typical entry looks:

```
{
  "odata.type": "Microsoft.DirectoryServices.AppRoleAssignment",
  "objectType": "AppRoleAssignment",
  "objectId": "Bkp-sDgT4kq5a-YB4HMF2q2NyOTf4hpKhVKXXQHxMhA",
  "deletionTimestamp": null,
  "creationTimestamp": "2015-09-06T08:53:30.1974755Z",
  "id": "00000000-0000-0000-0000-000000000000",
  "principalDisplayName": "Vittorio Bertocci",
  "principalId": "b07e4a06-1338-4ae2-b96b-e601e0731fda",
  "principalType": "User",
  "resourceDisplayName": "MarioApp1",
  "resourceId": "725a2d9a-6707-4127-8131-4f9106d771de"
}
```

That entry declares that the user Vittorio Bertocci (identified by its `objectId` `b07e4a06-1338-4ae2-b96b-e601e0731fda`) can have access to the app `MarioApp1` (object ID of the app's `ServicePrincipal` being `725a2d9a-6707-4127-8131-4f9106d771de`) in the capacity of role `00000000-0000-0000-0000-000000000000`.

This is where the role of Role (pun intended) comes into play. As you will see later, Azure AD allows developers to define application-specific roles. The `AppRoleAssignment` entity is meant to track that a certain app role has been assigned to one user for a certain app. What you are discovering here is that Azure AD uses `AppRoleAssignment` also for tracking app user assignments—but in this case, Azure AD automatically sets in the `AppRoleAssignment` a default role, `00000000-0000-0000-0000-000000000000`. It's as simple as that.

One notable property of `AppRoleAssignment` is `principalType`. The sample entry here has the value `User`, indicating that the entity being assigned the role is a user account. Other possible values are `Group` (in which case, all the members of the group are assigned the role) or `ServicePrincipal` (in which case, the role is being assigned to another client application).

If you use the Azure portal to assign more users to the app, you'll see corresponding new `AppRoleAssignment` entries appearing in the application. By the way, the query I used for getting the list of `AppRoleAssignments` for my app is:

```
https://graph.windows.net/developertenant.onmicrosoft.com/servicePrincipals/725a2d9a-6707-4127-8131-4f9106d771de/appRoleAssignedTo.
```

Just for kicks, try to access your application with a user that has not been assigned. Instead of the usual consent dialog, you'll get back a lovely error along the lines of:

```
"error=access_denied&error_description=AADSTS50105: The signed in user 'fabio@developertenant.onmicrosoft.com' is not assigned to a role for the application 'developertenant.onmicrosoft.com'."
```

The behavior described in this section is what you would observe if your application didn't define any app roles. In the next section, I'll explore app roles in more depth.

## App roles

Azure AD allows developers to define roles associated with an application. Traditionally, roles are handy ways of assigning collections of permissions to a bunch of users all at once: for example, people in a hypothetical Approver role might have read/write access to a certain set of resources, while people in the Reviewer role might have only read access to the same resources. Roles are handy because assigning a user to a role saves you the hassle of adding all the permissions a role entails one by one. Moreover, when a new resource is added to the milieu, access to that resource can be added to the role to enable access to it for all the users already assigned to the role, replacing the need to assign access individually, account by account. That said, roles in Azure AD do not necessarily need to represent permissions grouping: Azure AD does not offer you anything for representing such permissions anyway; it is your app's job to interpret each role. You can use application roles to represent any user category that makes sense for your application, like what is the primary spoken language of a user. True, there are many other ways of tracking the same info, but one big advantage of app roles over any other method is that Azure AD will send them in the form of claims in the token, making it extra easy for the app to consume the info they carry.

After you declare application roles, such roles are available to be assigned to users by the administrators of the tenants using your app. Let's take a look at how that cycle plays out.

The `Application` entity has one collection, `appRoles`, which is used for declaring the roles you want to associate with your application. As of today, the way in which you populate that property is by downloading the app manifest as described in "The *Application*" section at the beginning of this chapter, adding the appropriate entries in `appRoles`, and uploading it back via the portal. Here is what one `appRoles` collection looks like:

```
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
  },
]
```

```

    "description": "Approvers can mark documents as approved",
    "displayName": "Approver",
    "id": "8F29F99B-5C77-4FBA-A310-4A5C0574E8FF",
    "isEnabled": "true",
    "value": "approver"
  },
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Reviewers can read documents",
    "displayName": "Reviewer",
    "id": "0866623F-2159-4F90-A575-2D1D4D3F7391",
    "isEnabled": "true",
    "value": "reviewer"
  }
],

```

The properties of each entry are mostly self-explanatory, but there are a couple of nontrivial points.

The `displayName` and `description` strings are used in any experience in which the role is presented, such as the one in which an administrator can assign roles to users.

The `value` property represents the value that the role claim will carry in tokens issued for users belonging to this role. This is the value that your application should be prepared to receive and interpret at access time.

The `id` is the actual identifier of the role entry. It must be unique within the context of this `Application`.

The `allowedMemberTypes` property is the interesting one. Roles can be assigned to users, groups, and applications. An `allowedMemberTypes` collection including the entry `"User"` indicates a role that can be assigned to both users and groups. (In the next section, I'll cover roles assignable to applications.)

Once you have added the roles in the manifest file, don't forget to upload it back via the portal.



**Note** Sometimes the upload will fail, unfortunately without a lot of information to help you troubleshoot: watch out for silly errors—for example, nonmatching parentheses. I recommend using a syntax-aware JSON editor, which should take care of most issues up front.

If you head back to the Users tab and try to assign a new user to the app like you did in the preceding section, you'll see that you are no longer able to simply declare that you want to assign a user to the app: now you are presented with a choice between the various roles you declared in the manifest. Assign one of the roles to a random user, and then launch the app and try to sign in with that user.



**Note** Subscribers to Azure AD Premium will also see an experience allowing the assignment of groups.

If you go back to the `appRoleAssignedTo` property of the `ServicePrincipal` and inspect the role assignments there, you'll find the same user assignments from the preceding section, plus a new entry for the user you just assigned to a role. It should look something like this:

```
{
  "odata.type": "Microsoft.DirectoryServices.AppRoleAssignment",
  "objectType": "AppRoleAssignment",
  "objectId": "9pcRosZaC0a10yoa5r0IwZrIr_JYzUxFtCm1WBYn6w0",
  "deletionTimestamp": null,
  "creationTimestamp": null,
  "id": "8f29f99b-5c77-4fba-a310-4a5c0574e8ff",
  "principalDisplayName": "Fabio Bianchi",
  "principalId": "a21197f6-5ac6-460b-b5d3-2a1ae6bd08c1",
  "principalType": "User",
  "resourceDisplayName": "MarioApp1",
  "resourceId": "725a2d9a-6707-4127-8131-4f9106d771de"
},
```

As expected, the `id` property points to one of the roles just defined, as opposed to the default `00000000-0000-0000-0000-000000000000` used during user assignment.

Launch the app and sign in as the user you just assigned to the role. If you capture the traffic via Fiddler (as you learned about in Chapter 6) and peek at the JWT token sent in the `id_token`, you'll notice a new `roles` claim:

```
{
  "amr" : [ "pwd" ],
  "aud" : "1538b378-5829-46de-9294-6cfb4ad4bbaa",
  "c_hash" : "EOuY-5M5FXRyNCvRHe8Kg",
  "exp" : 1442740348,
  "family_name" : "Bianchi",
  "given_name" : "Fabio",
  "iat" : 1442736448,
  "iss" : "https://sts.windows.net/6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e/",
  "name" : "Fabio Bianchi",
  "nbf" : 1442736448,
  "nonce" : "635783335451569467.YzJiYmZjMGUtOWFhKMS00NzI3LWJkYjMtMzhiMjE0YjVmNWE0ZDcwZTk3YmYtNzQ4NC00YjkyLWFiY2YtYWViOWFhNjE0YjFj",
  "oid" : "a21197f6-5ac6-460b-b5d3-2a1ae6bd08c1",
  "roles" : [ "approver" ],
  "sub" : "0pcgG-Rxo_DSCJnuAf_7tdfXp7Xa0zpw6pF3x7Ga8Y0",
  "tid" : "6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e",
  "unique_name" : "fabio@developertenant.onmicrosoft.com",
  "upn" : "fabio@developertenant.onmicrosoft.com",
  "ver" : "1.0"
}
```



From your own application's code, you can find out the same information through the usual `ClaimsPrincipal.Current.FindFirst("roles")` or, given that this is a multivalued claim, `FindAll`. Once you have the value, you can do whatever the semantic you assigned to the role suggests that your code should do: allow or deny access to the method being called, change environment settings to match the preferences of the caller, and so on.

If you are using roles for authorization, classic ASP.NET development practices would suggest using `[Authorize]`, `<Authorization>`, or the evergreen `IsInRole()`. The good news is that they are all an option. The only thing you need to do is tell the identity pipeline that you want to use the claim type `roles` as the source for the role information used by those artifacts. That's done via one property of `TokenValidationParameters`, `RoleClaimType`. For example, you can add the following to your OpenID Connect middleware initialization options:

```
TokenValidationParameters = new TokenValidationParameters
{
    RoleClaimType = "roles",
}
```

Azure AD roles are a very powerful tool, which is great for modeling relationships between users and the functionality that the app provides. Although the concept is not new, Azure AD roles operate in novel ways. For example, developers are fully responsible for their creation and maintenance, while the administrators of the various tenants where the app is provisioned are responsible for actually assigning people to them. Also, Azure AD roles are always declared as part of one app—it is not possible to create a role and reuse it across multiple applications. There is no counterpart for this on-premises. The closest match is groups, but those have global scope, and a developer has no control over them. Before the end of the chapter, I will also touch on groups in Azure AD.

## Application permissions

All the features you encountered in this chapter are meant to give you control over how users have access to your app and how users can delegate your app to access other resources for them.

In some situations you want to be able to confer access rights to the application itself, regardless of what user account is using the app, or even when the app is running without any currently signed-in user. For example, imagine a long-running process that performs continuous integration—an app updating a dashboard with the health status of running tests against a solution and so on. Or more simply, think about all the situations in which an app must be able to perform operations that a low-privilege user would not normally be entitled to do—like provisioning users, assigning users to groups, reading full user profiles, and so on. Note that, once again, those kinds of permissions come into play when accessing the resource as a web API, so you won't see this feature really play out until the next chapter. Here I'll just discuss provisioning.

While delegated permissions are represented in Azure AD via `oauth2Permission` in the `Application` object and the `oauth2PermissionsGrants` collection in the `ServicePrincipal` table, Azure AD represents application permissions via `Application.appRoles` and `ServicePrincipal.appRoleAssignedTo`.

The `AppRole` entity is used to declare application permissions just as you have seen for the application roles case, with the difference that `allowedMemberTypes` must include an entry of value `"Application"`. To clarify that point, let's once again turn to the Directory Graph API Service-Principal and examine its `appRoles` collection:

```
"appRoles": [
  {
    "allowedMemberTypes": [
      "Application"
    ],
    "description": "Allows the app to read and write all device properties without a signed-in user. Does not allow device creation, device deletion, or update of device alternative security identifiers.",
    "displayName": "Read and write devices",
    "id": "1138cb37-bd11-4084-a2b7-9f71582aedd8",
    "isEnabled": true,
    "value": "Device.ReadWrite.All"
  },
  {
    "allowedMemberTypes": [
      "Application"
    ],
    "description": "Allows the app to read and write data in your organization's directory, such as users and groups. Does not allow create, update, or delete of applications, service principals, or devices. Does not allow user or group deletion.",
    "displayName": "Read and write directory data",
    "id": "78c8a3c8-a07e-4b9e-af1b-b5ccab50a175",
    "isEnabled": true,
    "value": "Directory.Write"
  },
  {
    "allowedMemberTypes": [
      "Application"
    ],
    "description": "Allows the app to read data in your organization's directory, such as users, groups, and apps.",
    "displayName": "Read directory data",
    "id": "5778995a-e1bf-45b8-af6a-663a9f3f4d04",
    "isEnabled": true,
    "value": "Directory.Read"
  }
],
```



**Note** `Directory.Write` and `Directory.Read` will follow the same update path as their delegated homonyms and become `Directory.ReadWrite.All` and `Directory.Read.All`, respectively.

You can think of each of those roles as permissions that can be requested by applications invoking the Graph API. Although in the case of user and group roles, administrators can perform role assignments directly in the Azure management portal, granting application roles works very much like delegated permissions—via consent at the first token request.

A client application needs to declare in advance what application permissions (that is, application roles) it requires. That is currently done via the Azure portal, in the Permission To Other Application section of the Configure tab. In Figure 8-5 earlier, you can see that the middle column of the screen contains a drop-down labeled Application Permissions, in that case specifying the options available for the Directory Graph API. It is operated much as you learned about for the Delegated Permissions list, but the entries exposed in Application Permissions are the ones in the target resource from its `appRoles` collection, and specifically the ones marked as `Application` in `allowedMemberTypes`.

What happens when you select an application permission, say Read Directory Data, for the Directory Graph API? Something pretty similar to what you have seen in the case of delegated permissions. Take a look at what changes in the Application's `requiredResourceAccess` collection:

```
"requiredResourceAccess": [
{
  "resourceAppId": "00000002-0000-0000-c000-000000000000",
  "resourceAccess": [
    {
      "id": "5778995a-e1bf-45b8-affa-663a9f3f4d04",
      "type": "Role"
    },
    {
      "id": "78c8a3c8-a07e-4b9e-af1b-b5ccab50a175",
      "type": "Scope"
    },
    {
      "id": "311a71cc-e848-46a1-bdf8-97ff7156d8e6",
      "type": "Scope"
    }
  ]
}
]
```

The resource you want to access remains the same, the Directory Graph API—represented by the ID 00000002-0000-0000-c000-000000000000. In addition to the old delegated permissions, of type `Scope`, you'll notice a new one, of type `Role`. The ID of this one corresponds exactly to the ID declared in the Directory Graph API's `ServicePrincipal` `appRoles` for the Read Directory Data permission.

As I mentioned, granting application permissions takes place upon successful request of a token from the app and positive consent granted by the user at authentication time. The presence of an entry of type `Role` in a `RequiredResourceAccessCollection` introduces a key constraint, however: only admin consent requests will be considered. This means that every time you develop an app requesting application permissions, you have to be sure that the first time you request a token from it, you append the `prompt=admin_consent` flag to your request.

If you actually launch the app and go through the consent dance, you'll find that after provisioning, the directory has added one new `AppRoleAssignment` entry to the `appRoleAssignedTo` property of the app's `ServicePrincipal` entry in the target tenant. Or better, you would find it if your app had requested permissions for *any* resource other than the Directory Graph API. As I am writing this chapter, the Directory Graph API is the only resource that received special treatment from

Azure AD: whereas every other resource has its consent settings recorded in the entities described in this chapter, as of today clients accessing the Graph API record the application permissions consent for it elsewhere. I won't go into further details for two reasons. One, it would not help you understand how application permissions work in general, given that each and every other resource does use `appRoleAssignedTo`. Two, there is talk of changing the Directory Graph API behavior so that it will start acting like any other resource—it's entirely possible that this will already be the case once the book is in your hands, but given that it's not for sure, I am not taking any chances.

With their permission/role dual nature, application permissions can be confusing. However, they are an extremely powerful construct, and the possibilities their use opens up are well worth the effort of mastering them.

## Groups

---

In closing this chapter about how Azure AD models applications, I am going to show you how to work with groups. Groups in Azure AD can be cloud-only sets of users, created and populated via the Azure portal or the Office 365 portal, or they can be synched from on-premises distribution lists and security groups. Groups have been a staple of access control for the last few decades. As a developer, you can count on groups to work across applications and to be assigned and managed by administrators: all you need to know is that a group exists and what its semantic is and then use that information to drive your app's decisions regarding the current user (access control, UI customization, and so on).

By default, tokens issued by Azure AD do not carry any group information: if your app is interested in which groups the current user belongs to, it has to use the Directory Graph API (cue the next chapter).

Just as with application roles, you can ask Azure AD to start sending group information in issued tokens in the form of claims—simply by flipping a switch property in the `Application` object. If you download your app manifest, modify the `groupMembershipClaims` property as follows, and then upload the manifest again, you will get group information in the incoming tokens:

```
"groupMembershipClaims": "All",
```

If you are interested in receiving just the security groups, enter `"SecurityGroup"` instead of `"All"`.

After changing the manifest as described, I used the portal to create in my test tenant a new group called `"Hippies,"` and assigned to it the test user Fabio. That done, I launched the app and signed in as Fabio. Here's the token I got:

```
{
  "amr" : [ "pwd" ],
  "aud" : "c3d5b1ad-ae77-49ac-8a86-dd39a2f91081",
  "c_hash" : "zit-F66pwRsDeJPtjpuzgA",
  "exp" : 1442822854,
  "family_name" : "Bianchi",
  "given_name" : "Fabio",
  "groups" : [ "d6f48969-725d-4869-a7a0-97956001d24e" ],
```

```

    "iat" : 1442818954,
    "iss" : "https://sts.windows.net/6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e/",
    "name" : "Fabio Bianchi",
    "nbf" : 1442818954,
    "nonce" : "635784160492173285.ZmIyMTM5NGYtZDEyNC00MThmLTgyN2YtNTZkNzViZjA1MDgxMz1jZDA10WmtNjVhOC00ZWl1LThkNmQtZDE4NGJ1OTU2ZGZj",
    "oid" : "a21197f6-5ac6-460b-b5d3-2a1ae6bd08c1",
    "sub" : "0vmQvSCoJqTYby1EE0XR94PgRuveuOWUbAHNkmf0xTk",
    "tid" : "6c3d51dd-f0e5-4959-b4ea-a80c4e36fe5e",
    "unique_name" : "fabio@developertenant.onmicrosoft.com",
    "upn" : "fabio@developertenant.onmicrosoft.com",
    "ver" : "1.0"
  }
}

```

You can see that there is indeed a groups claim, but what happened to the group name? Well, the short version of the story is that because Azure AD is a multitenant system, arbitrary group names like “People in building 44” or “Hippies” have no guarantee of being unique. Hence, if you wrote code relying on only a group name, your code would often be broken and subject to misuse (a malicious admin might create a group matching the name you expect in a fraudulent tenant and abuse your access control logic). As a result, today Azure AD sends only the `objectId` of the group. You can use that ID for constructing the URI of the group itself in the directory, in this case that’s:

```
https://graph.windows.net/developertenant.onmicrosoft.com/groups/d6f48969-725d-4869-a7a0-97956001d24e.
```

In the next chapter, you’ll learn how to use the Graph API to use that URI to retrieve the actual group description, which in my case looks like the following:

```

{
  "odata.metadata": "https://graph.windows.net/developertenant.onmicrosoft.com/$metadata#directoryObjects/Microsoft.DirectoryServices.Group/@Element",
  "odata.type": "Microsoft.DirectoryServices.Group",
  "objectType": "Group",
  "objectId": "d6f48969-725d-4869-a7a0-97956001d24e",
  "deletionTimestamp": null,
  "description": "Long haired employees",
  "dirSyncEnabled": null,
  "displayName": "Hippies",
  "lastDirSyncTime": null,
  "mail": null,
  "mailNickname": "363bdd6b-f73c-43a4-a3b4-a0bf8b528ee1",
  "mailEnabled": false,
  "onPremisesSecurityIdentifier": null,
  "provisioningErrors": [],
  "proxyAddresses": [],
  "securityEnabled": true
}

```

Your app could query the Graph periodically to find out what group identifiers to expect, or you could perform queries on the fly as you receive the group information, though that would somewhat defeat the purpose of getting groups in the form of claims.

Consuming groups entails more or less the same operations described for roles and ClaimsPrincipal. You can even assign groups as the RoleClaimType if that's the strategy you usually enact for groups (traditional IsInRole actually works against Windows groups on-premises, often creating a lot of confusion).

One last thing about groups. There are tenants in which administrators choose to use groups very heavily, resulting in each user belonging to very large numbers of groups. Adding many groups in a token would make the token itself too large to fulfil its usual functions (such as authentication and so on), so Azure AD caps at 200 the number of groups that can be sent via JWT format. If the user belongs to more than 200 groups, Azure AD does not pass any group claims; rather, it sends an overage claim that provides the app with the URI to use for retrieving the user's groups information via the Graph API. Azure AD does so by following the OpenID Connect core specification for aggregated and distributed claims: in a nutshell, a mechanism for providing claims by reference instead of passing the values. Say that Fabio belonged to 201 groups in our sample above. Instead of the groups claims, the incoming JWT would have contained the following claims:

```
"_claim_names": {
  "groups": "src1",
},
"_claim_sources": {
  "src1": {"endpoint": "https://graph.windows.net/developertenant.onmicrosoft.com/users/a21197f6-5ac6-460b-b5d3-2a1ae6bd08c1/getMemberObjects"}
}
```

In the next chapter, you'll learn how to use that endpoint for extracting group information for the incoming user.

## Summary

---

The Azure AD application model is designed to support a large number of important functions: to hold protocol information used at authentication time, provide a mechanism for provisioning applications within one tenant and across multiple tenants, allow end users and administrators to grant or deny consent for apps to access resources on their behalf, and supply access control knobs to administrators and developers to fine-tune user and application access control.

That's a tall order, but as you have seen throughout this chapter, the Azure AD application model supports all of those functions—though in so doing, it often needs to create complex castles of interlocking entities. Note that little of that complexity ever emerges all the way to the end user, and even for most development tasks, you don't need to dive as deep as we did in this chapter. However, as a reward for the extra effort, you now have a holistic understanding of how applications in Azure AD are represented, provisioned, and granted or denied access to resources. You will find that this skill will bring your proficiency with Azure AD to a new level.



# Index

## A

- About() action, 238–239, 241, 249–250
- access control
  - for applications, 216–219
  - enforcing, 82
  - groups, 219–221
  - risk levels, 59
  - for web APIs, 283
- Access Control Service (ACS), 78–79
- access delegation, 31–33
- AccessToken property, 229
- access tokens, 35, 256. *See also* tokens
  - claims, 263, 289–290, 292
  - invoking web API with, 232–251
  - JWT format, 271
  - life span, 238–239
  - opacity to token requestors, 72, 233
  - refresh tokens, 238–251. *See also* refresh tokens
  - renewing, 70–71
  - requests, 268–269
  - responses, 269–270
  - scope, 116. *See also* scopes
- AccountController sign-in and sign-out logic, 104
- AcquireTokenByAuthorizationCode method, 229, 244–245, 287–288
- AcquireToken method, 228, 269
- AcquireToken\* methods, 238–239, 241, 247
- AcquireTokenSilent method, 239, 241, 243–244
  - failed calls, 246
- acr claims, 133
- Active Directory (AD)
  - access token representation, 255
  - introduction, 15
  - as new directory in the cloud, 58
  - on-premises, 15–16
  - on-premises vs. cloud approach, 58–59
  - projection in the cloud, 58
  - setup in Windows Server 2016, 273–274
  - token requests, 70
  - Visual Studio integration, 85–87
- Active Directory Authentication Library (ADAL), 76–78
  - accessing APIs as application, 251–252
  - accessing APIs as arbitrary user, 252
  - cache, 243–247
  - handling AuthorizationCodeReceived notification, 227–230
  - JavaScript versions, 80
  - midtier client libraries, 81
  - native apps libraries, 48, 80–81
  - .NET NuGet package, referencing, 227–228
  - refresh tokens, 238–251
  - session management, 238–251
  - token-acquisition pattern, 77
  - token caches, 238–239
- Active Directory Federation Services (ADFS), 9, 25, 52–56
  - access control policies for web APIs, 283
  - access token representation, 255
  - access tokens for web APIs, 285–288
  - API and UX entries, 282
  - application groups, 274–275
  - application permissions for web APIs, 284
  - app provisioning, 287
  - Client-Server Applications section of management UX, 275
  - credentials gathering, 280–281
  - endpoints, 276–277
  - federated tenants, 65–66
  - JWT format for access tokens, 271
  - management UX, 274–276
  - multiresource refresh tokens, 286
  - Native Application and Web API template, 275
  - OAuth2 authorization code grants, 55
  - OpenID Connect support, 103
  - protocol support, 55–56
  - Server Application and Web API template, 275
  - setting up, 54, 273–274
  - signing keys, 280
  - tokens issued by, 289
  - web API identifiers, 282
  - web API invocation, 288–289
  - web API setup, 281–285



## Active Directory Federation Services (ADFS) “3”

- Active Directory Federation Services (ADFS), *continued*
  - web app setup, 277–280
  - web sign-on with OpenID Connect, 276–281
  - Windows Integrated Authentication credential, 279
  - in Windows Server 2016, 56, 103, 273–292
  - workplace-joined device detection, 56
- Active Directory Federation Services (ADFS) “3”
  - application provisioning, 271
  - client entity, 274
  - OAuth2 support, 272
  - web APIs, protecting, 271–272
- ActiveDirectoryFederationServicesBearerAuthentication method, 271
- ADAL4J, 81
- ADAL Android, 81
- ADAL Cordova, 81
- ADAL iOS, 80
- ADAL JS, 72, 85, 294
- ADAL .NET, 78–80
- Add Transform Claim Rule Wizard, 290
- admin consent, 173, 200–204. *See also* consent
  - dialog box for, 202
  - requests, 210
- administrators
  - ADFS management, 53–54, 57
  - application creation, 204
  - Azure portal, 64, 66
  - claims issued, managing, 9, 57
  - consent prompts, 199
  - control over trust establishment, 57
  - directory resource access, 173
  - directory sync, 65
    - global, 93
    - guest, 93
    - permissions, 59, 198–200
- AJAX calls, 235
- allowedMemberTypes collection, 214, 216–217
- amr claims, 133
- AngularJS, 47
- anonymous access, 97
- APIs. *See* web APIs
- <api version> component in URL template, 237
- AppBuilder type, 141–142
- appld property, 180, 188
- App ID Uri, 209, 256
- Application.appRoles object, 216–219
- application groups, 274–275, 281–282
- application identifiers, 181
- application-level authentication messages, 25
- application model, Azure AD. *See* Azure Active Directory
  - application model
- Application object, 175, 177–186
  - authentication properties, 180–183
  - deletion timestamp, 180
  - JSON file, 178–180
  - object ID, 180
  - properties by functional group, 186
  - for web APIs, 257–258
- Application Proxy, 67
- applications
  - access directory as user permission, 196
  - accessing resources as, 45
  - accessing web APIs, 252
  - access through Azure Active Directory, 66
  - actions, 177
  - adding to application groups, 281–282
  - ADFS code, libraries, protocol support, 53, 55
  - admin consent, 173, 200–204, 210
  - admin creation, 204
  - admin-level permissions, 198–200
  - app-level permissions, 216–219
  - assigned users, 188
  - authenticate users permission, 195–196
  - authentication options, 177
  - availability to other tenants, 182–183
  - client role, 70–72. *See also* clients
  - credentials, 226–227, 279
  - decoupling from web servers, 138
  - delegated permissions, 192–197
  - directory read and write permissions, 196
  - display name, 181
  - enumerate users permissions, 196–197
  - group read and write permissions, 197
  - homepage, 181
  - identifiers, 177, 188
  - identifying authentication protocols of, 64
  - IdP metadata, reading, 21
  - IdP trust, 18
  - initialization, 140–141
  - isolated and independent, 14
  - iss (issuer) value, 120–121, 208
  - key string assignments, 226–227
  - multitenancy, 205–211
  - nonadmin user creation, 189–192
  - OAuth2 permissions, 183–185
  - partitioning for consumption routes, 265–266
  - protecting with Azure AD, 60–61
  - protocol coordinates, 61, 177, 276, 278
  - provisioning, 53–54, 57, 189
  - public vs. confidential clients, 181
  - relying parties, 18. *See also* relying parties (RPs)
  - resource protectors, 69, 73–74
  - resources, 177, 185–187
  - roles, 182, 213–216
  - scopes, 201. *See also* scopes
  - single-page, 45–47
  - as token requestors, 69–72, 74
  - token validation, 22. *See also* token validation
  - user assignment, 211–213
- app manifest files, 214, 219

- appOwnerTenantId property, 188
- app parameter, 140–141
- app permissions, 216–219
- AppRoleAssignment entity, 212
- appRoleAssignedTo property, 215
- appRoleAssignmentRequired property, 188, 212
- AppRole entity, 216–217
- appRoles property, 182, 188
- ASP.NET
  - Katana. *See* Katana
  - membership providers, 14
  - project templates for web APIs, 287
  - support for web sign-on, 137. *See also* Open Web Interface for .NET (OWIN) middlewares
  - templates in Visual Studio 2015, 87
- ASP.NET 4.6
  - vs. ASP.NET 5, 90
  - initialization code, 95
  - web API projects, 254. *See also* web APIs
- ASP.NET 5, 85, 90
- ASP.NET applications, 3–7. *See also* web applications
  - building, 89
  - claims-based identity support, 82
  - MVC project type, 90–91
  - OWIN components, 83–84
- assembly:OwinStartup attribute, 139
- assertions, 26
- attributes, 20, 59, 290
- audience claims, 132, 282
- authentication
  - Application object properties, 180–183
  - Azure AD for, 1
  - claims-based identity, 17–23
  - default process, 4–7
  - defined, 12
  - failure notification, 166
  - indicating success, 98
  - mechanisms for, 7
  - mode, 152, 158–159
  - modern, 31–48
  - multitenant systems, 58
  - native apps vs. web apps, 94
  - pre-claims techniques, 12–16
  - round-trip web apps, 23–31
  - steps of, 73
  - triggering, 97–98, 100
  - type setting, 158
- AuthenticationContext class, 228
  - initialization, 287
- AuthenticationFailed notification, 167
- authentication flows
  - across multiple tenants, 205–208
  - authorization-code, 42–43
  - hybrid, 40–42, 108
  - OWIN middlewares pipeline, 148–153
  - state, preserving, 116–117
- AuthenticationManager instance, 148
- authentication middlewares, 148–153
  - Authentication property, 146
- AuthenticationMode property, 159
  - Active option, 149
- AuthenticationProperties settings, 100
- Authentication property, 146, 150, 152–153
- AuthenticationReponseGrant, 150
- authentication-request message type, 39
- authentication requests, 98, 113–119
  - authorization endpoints, 114
  - clientId, 114
  - nonce, 117
  - omitted parameters, 117–119
  - response mode and response type, 114–116
  - scope, 116
  - state, 116–117
- AuthenticationResult instance, 229
- AuthenticationTicket store, 171
- AuthenticationType property, 159
- authorities, 18
  - /adfs/, 287
  - control over user authentication experience, 122
  - validation in ADFS, 287
- authority coordinates, validation and, 157–158
- Authority property, 155
- authority types. *See* Active Directory Federation Services (ADFS); Azure Active Directory
- authorization, 33–39, 116
  - OAuth2 grants, 55, 252
  - <Authorization>, 216
- AuthorizationCodeReceived notification, 167, 227–230, 286
- authorization codes
  - acquiring, 225
  - client secrets, 156
  - code-redemption logic, 227–230
  - OpenID Connect flow, 42–43
  - redeeming, 224–232, 286
- authorization endpoint, 35, 63, 114, 207, 285
- Authorization HTTP headers, tokens embedded in, 232–234
- authorization requests, 149
- authorization server (AS), 34–35
- [Authorize] attribute, 97–98, 148–149
  - on entire class, 257
  - role information, 216
  - scope-verification logic, 264
- auth\_time claims, 133
- availableToOtherTenants property, 182–183, 209
- Azure Access Control Service (ACS), 41, 78–79
- Azure Active Directory, 1, 56–67
  - access token representation, 255
  - application access, 66

## Azure Active Directory application model

- application entry permissions, 224–225
- application model. *See* Azure Active Directory application model
- Application Proxy, 67
- apps, adding entry for, 60
- authorization endpoints, 114
- B2C (business to consumer), 294–295
- client-credentials grants, 251
- client IDs, 94, 97
- cloud workload functionality, 59
- consent prompt, 5–6
- cookies on user browser, 124
- credential gathering, 122–123
- credentials prompt, 5
- default domain, 62–63
- development and, 2, 60–61
- Directory Graph API, 10, 59. *See also* Directory Graph API
- directory sync, 65–66
- discovery, 119–120
- functional components, 59–60, 63–65
- group information in tokens, 219–220
- libraries, 75–86
- multitenancy, 58, 205–208
- oauth2PermissionGrants collection, 189–192
- obtaining tenant, 61–62
- online developer guide, 293
- OpenID Connect endpoints, 109
- permissions for directory access, 193–197
- private/public key pair information, 227
- programmatic access to entities, 236–237
- programmatic interface, 64–65. *See also* Directory Graph API
- projection of on-premises, 65
- protocol endpoints, 63–64
- protocols supported, 58
- redirect URIs, 100
- refresh tokens, 240–243
- registering apps, 93–94
- resource identifiers in token requests, 256
- resource-protector library references, 92
- response to POST, 123–125
- response types, 124
- service deployments, 63
- sessions, cleaning, 135
- synchronizing users and groups to, 65–66
- tenantID, 63
- tenants, 62, 93
- tokens, 61. *See also* tokens
- token-signing keys, 120–122
- token validation, 149–151
- trial, 2
- tying to Visual Studio, 2–3
- user information, accessing, 7–10
- Visual Studio 2015 connected services, 87
- web API provisioning, 253
- Azure Active Directory application model, 64, 173–221
  - admin consent, 200–204
  - admin-level permissions, 198–200
  - admin user application creation, 204
  - app-level permissions, 216–219
  - Application object, 175, 177–186
  - app roles, 213–216
  - app user assignments, 211–213
  - consent, 175, 189–192
  - delegated permissions, 192–197
  - functions, 173
  - groups, 219–221
  - multitenancy, 205–211
  - provisioning flow, 175–176
  - ServicePrincipal object, 187–188
  - service principals, 174–177
- Azure Active Directory Basic, 62, 66–67
- Azure Active Directory Connect, 65
- Azure Active Directory Free tier, 61–62
- Azure Active Directory Premium, 62, 66–67, 215
- Azure Active Directory vNext, 295
- Azure management portal, 60–61, 64
  - application configuration section, 178
  - application credentials, assigning, 226
  - Application entity JSON file, 178
  - application permission selection UI, 198
  - application permissions screen, 217–218
  - application tags, 188
  - manifest management section, 178
  - multitenancy setting, 209
  - provisioning apps in Azure AD, 93–94
  - Users tab, 212
- Azure subscription, 2, 93

## B

- back ends, HTTP requests to, 46
- Balfanz, Dirk, 41
- BaseNotification class, 161
- bearer token middleware
  - diagnosing issues, 261
  - notifications, 264
  - Provider, specifying, 265
  - tokens from ADFS, validation, 271
- bearer tokens, 232–237, 262
  - extraction and validation, 255
- BootstrapContext property, 268
- broker apps, 48
- browsers
  - hosting prompting logic in, 48
  - network tracing features, 110
  - presentation layer, 45–46
- business to consumer (B2C) Azure AD, 294–295

## C

- CallbackPath property, 158
- caller identity class, 7–10
- callers
  - attributes, 7
  - identifying, 23–24
  - retrieving names of, 7–8
- Caption property, 159
- Challenge method, 99–100
- Challenge sequence in OpenID Connect middleware, 152
- claims, 7, 20
  - from access tokens, 263
  - adding to access tokens, 289–290, 292
  - vs. attributes, 290
  - group information, 182, 219–220
  - in ID tokens, 132–134
  - information in, 57
  - JWT types, 131–132
  - OAuth2 and, 36–37
  - sourcing values, 52
  - type identifiers, 8–9
- claims-based identity, 17–23
  - authentication process, 21–22
  - identity providers, 17–18
  - just-in-time identity information, 57
  - protocols, 20–23
  - tokens, 18–20
  - trust and claims, 20
- claims-oriented protocols, communication across boundaries, 36–37
- ClaimsPrincipal class, 7–10, 82
  - Claims list, 263
  - Current.FindFirst(“roles”), 215–216
  - Current property, 8
  - in OWIN, 83
  - saving, 151
  - source location, 8
- ClaimsPrincipalSelector delegate, 8
- claims rules engine, 52
- claims transformation engine, 60
- ClaimTypes enumeration, 8–9
- ClientAssertionCertificate, 231
- ClientCredential class, 229, 251, 287
- client credentials grants, 44–45, 251, 266
- client IDs, 94, 97, 114, 155, 190
  - of application, 256
  - overriding at registration, 276
  - refresh tokens and, 243
- client-resource interactions, tokens for, 70–71
- clients, 70. *See also* token requestors
  - access control policies for web APIs, 283
  - in ADFS “3,” 274
  - ADFS support, 55
  - application permissions for web APIs, 284
  - confidential, 181, 275
  - definitions of term, 72
  - entries in target directories, 183
  - granted permissions, 189–192
  - identity and resource consumption, 265–266
  - public, 181, 275
  - scopes, 284–285
  - as token requestors, 72
  - of web APIs, 291
- client secrets, 156, 227
- cloud applications, 57–58
- cloud-based Active Directory, 58–59. *See also* Azure Active Directory
- cloud-based authentication, 56. *See also* Azure Active Directory
- cloud-based directories, 60
- cloudidentity.com blog, 80
- cloud services, 58. *See also* Azure Active Directory
- cloud stores, 59
- code reuse, 71
- common endpoint, 121, 207
- confidential clients, 181, 275
- ConfigurationManager class, 157–158
- ConfigureAuth, 141
- consent, 189–192
  - across tenants, 209–211
  - admin, 173, 200–204, 210
  - AppRoleAssignment entries, 212
  - provisioning flow, 175–176
  - for resource access, 186
  - revoking, 259
  - settings, 211
  - for web APIs, 258
- consent prompts, 44, 191
  - for admin users, 199
  - for multitenant pages, 210
- constrained delegation, 43
- context
  - AuthenticationManager instance, 148
  - Authentication property, 146, 150, 152–153
  - environment dictionary, 147
  - middlewares, 142, 145–148
  - Request and Response properties, 147–149
  - TraceOutput property, 148
- contracts, 23
- controllers, MVC 5 Controller, 100
- cookie-based sessions, 92
- cookie middleware
  - adding to pipeline, 96
  - adding to web apps, 92
  - ClaimsPrincipal, saving, 151
  - collaboration with protocol middleware, 148
  - response processing, 150
  - sessions, managing, 171
  - sessions, saving, 150

## cookies

- cookie middleware, *continued*
  - sign-out, 100–101
- cookies
  - domain-bound, 24–25
  - life cycles, 24, 46
  - limitations, 46
  - nonce value, tracking, 124–125
  - session. *See* session cookies
  - on user browser, 124
  - for web API protection, 235–236
- Cordova ADAL library, 81
- credentials
  - application, 226–227
  - assigning, 226
  - gathering, 122–123
  - grants, 44–45, 251, 266
  - keys, 181. *See also* keys
  - passwords, 181
  - in ServicePrincipal, 188
  - sharing among apps, 32–33
  - storage, 226
  - types, 13
- credentials validation and session cookie authentication pattern, 23–24
- cross-collaboration scenarios, 17
- cross-domain single sign-on, problems, 23–25
- Current property, 8

## D

- decoupling web servers from apps, 138. *See also* middlewares; Open Web Interface for .NET (OWIN)
- default authentication process, 4–7
- delegated access, 34–36
- delegated permissions, 185, 192–197
  - scopes, 201
- deletionTimestamp property, 180, 188
- Devasahayam, Samuel, 15
- developer-assigned application identifiers, 181
- development certificates, 91
- development libraries
  - in Active Directory, 75. *See also* libraries
  - for native clients, 294
  - for other platforms, 293
- development on dedicated machines, 91
- diagnostic middleware, 153–154
- digital signatures, 19
- directories, defined, 62
- Directory.AccessAsUser.All permission, 196
- directory access permissions, 193–196
- directory entities, programmatic access to, 236–237
- directory entries for web APIs, 257–258
- Directory Graph API, 10, 59–60, 64–65, 236–237
  - Application object JSON file, 178–180

- application permissions, 217–218
  - calling, 233–234
  - group information, 219
- directory permissions, 193–197
- directory queries in cloud applications, 57–58
- Directory.Read.All permission, 196
- Directory.Read permission, 196
- Directory.ReadWrite.All permission, 196
- directory services for multitenant systems, 58
- directory sync, 65–66
- directory tenants, 58
- Directory.Write permission, 196
- discovery document, 119, 208
  - keys document, 120–121
  - location, 277
- displayName property, 181, 188
- distributed sign-out, 27, 29, 101, 109
- domain-based identifiers, 63
- domain controllers (DCs), 15, 20, 23
- domain\_hint parameter, 118
- domain-joined servers, ADFS on, 54
- domain-joined workstations, 14–16
- domains, 14–16, 62

## E

- email claims, 133
- endpoints, 18
  - ADFS, 276–277
  - common, 207
  - multitenancy and, 206–207
  - network, 52
  - OAuth2, 64
  - protocol, 60, 63–64
  - protocol/credential type, 60
  - turning on and off, 52
- entities, 22–23, 70
- environment dictionary, 138, 147
- errorUrl property, 188
- exp claims, 132
- ExpiresOn property, 230

## F

- family\_name claims, 133
- federated tenants, 65–66, 122
- federation. *See also* Active Directory Federation Services (ADFS)
  - for integrating with Azure AD, 66
  - for synchronized deployments of Azure AD, 65
- Fiddler, 110
  - capturing trace, 112
  - HttpClient traffic tracing, 261
  - setup, 111

Fiddler inspector, 127  
 first-name claim type, 8  
 form post response mode, 115  
 fragment response mode, 115  
 functions, creating, 163–164

## G

GET operations  
   of Account/SignOut, 134  
   for authenticated resource requests, 125  
   requests through, 182  
 given\_name claims, 133  
 Goland, Yaron, 41  
 grants  
   admin consent, 203–204  
   AuthenticationResponseGrant, 150  
   client credentials, 44–45, 251, 266  
   implicit, 46–47  
   OAuth2 grants, 252  
   oauth2PermissionGrants collection, 189–192  
   refresh token, 239–240, 242  
 Graph API. *See* Directory Graph API  
 groupMembershipClaims property, 182, 219  
 Group.Read.All permission, 197  
 Group.ReadWrite.All permission, 197  
 groups, 219–221  
   assigning, 215  
   consuming, 220–221  
   names, 220  
   number of, 221  
 guest Microsoft account users, 122

## H

HandleResponse method, 162  
 hero apps, 48  
 homepage property, 181, 188  
 HostAuthenticationFilter attribute, 266  
 hosts in OWIN pipeline, 140  
 HTTP 302s, 98  
   redirects, 113–119, 145, 149–150  
   requests, 29–30, 44, 46  
   responses, 125  
 HTTP 401 responses, 149, 235, 261–262  
 HTTP claims-based identity, 22  
 HttpClient traffic tracing, 261  
 HttpContext.Current.User, 8  
 HttpContext.GetOwinContext().Authentication method, 100  
 HttpContext.GetOwinContext().Authentication.SignOut method, 134  
 HttpModules, 83, 137  
   as host for OWIN pipeline, 140

  predefined events, 145  
 HTTP requests to back end, 46  
 HTTPS URL for projects, 91, 94  
 HttpWatch, 110  
 hybrid authentication flow  
   APIs, obtaining tokens, 224–232  
   authorization code redemption, 227–232  
   authorization codes, 166  
   initialization, 113  
   OpenID Connect, 40–42, 108  
   token validation requirements, 133  
 hybrid token-requestor and resource-protector role  
   development libraries, 85–86

## I

IApplicationBuilder interface, 140  
 iat claims, 132  
 IAuthenticationSessionStore interface, 171  
 identifierUri property, 181  
 identity libraries. *See* libraries  
 identity party trusts in ADFS, 274  
 identity providers (IdPs), 17–18, 20  
   endpoints, 18  
   metadata, 18, 21, 108  
   public-private key pairs, 18–19  
   redirecting to, 160  
   SAML, 25–26  
   string identifiers, 18  
   WS-Federation, 28–29  
 identity transactions, 17–23  
 ID tokens, 39–40, 116, 127–134, 230, 280–281, 286  
   claims in, 133–134  
   decoding, 127–129  
   from server-to-server calls, 42  
   user information in, 269  
   validating, 42, 133  
 IIS Express, 91  
 IIS integrated pipeline, 145  
 impersonation, 44  
 implicit flow, 182  
 implicit grants, 46–47  
 integrated authentication, 14–16  
 interceptors, 74  
 intranets, authentication on, 14–16  
 Intune API, 61  
 Invoke method, 142  
 IOwinContext wrapper, 142  
 IsInRole() role information, 216  
 IsMultipleRefreshToken property, 230  
 iss claims, 132  
 IssuerSigningKey property, 167  
 issuer validation, 208–209  
 iss (issuer) value, 120–121, 208

### J

#### JavaScript

- HTTP requests to back end, 46
- logic-layout management, 45–46
- native apps, 81
- token bits, retrieving, 46–47

Jones, Mike, 41

JSON Tokens, 41

JSON Web Algorithms (JWA), 110, 131

JSON Web encryption (JWE), 129

JSON Web Keys set (JWKS), 280

JSON Web Signature (JWS), 129–131

JSON Web Token (JWT), 19, 40

- access token representation as, 255
- for access tokens, 271
- ADFS support, 55
- claim set, 131–132
- components, 129–130
- handlers, 84, 92
- header types, 131
- specification, 110, 129
- tokens, 84

just-in-time provisioning, 58

### K

Katana, 139–154. *See also* Open Web Interface for .NET (OWIN)

- assembly:OwinStartup attribute, 139
- context, 145–148
- diagnostic middleware, 153–154
- appSettings entry, 139
- middleware behavior settings, 158–159
- middleware execution, 145
- notifications, 159–166
- OwinStartup attribute, 139
- Startup class, 139–141
- UseStageMarker method, 145

“Katana” 3.x, 83–84

“Katana” vNext, 84

Kerberos

- native applications and, 47
- service principals, 174

Kerberos federation, 17

keyCredentials property, 181, 188, 226–227

keys

- assigning to applications, 226–227
- credentials, 181
- IssuerSigningKey property, 167
- JSON Web Keys set, 280
- keyCredentials property, 181, 188, 226–227
- public-private, 18–19, 227
- RefreshOnIssuerKeyNotFound property, 158

- signing, 280
- symmetric, 19
- token-signing, 120–122, 158
- token-validation, 167
- ValidateIssuerSigningKey property, 168

keys document, 120–121

Klout web application, 248–249

knownClientApplications property, 183, 258

### L

libraries

- in Active Directory, 75
- authentication tasks, 73–74
- for hybrid token-requestor and resource-protector role, 74–75, 85–86
- for native clients, 294
- open source, 76
- for other platforms, 293
- reasons for using, 71
- for resource-protector role, 73–74, 82–85
- for token-requestor role, 70–71, 76–81

line-of-business (LOB) applications, 4–5

local networks, authentication on, 14–15

localStorage, 47

login\_hint parameter, 117

logoutUrl property, 188

### M

managed tenants, 65–66, 122

manifest files, 214, 219

/me alias, 237

MessageReceived notification, 165

messages

- SAML, 26–27
- signed, 26
- WS-Federation, 28–31

metadata, 18, 21

MetadataAddress, 104, 277

metadata documents, 158

- discovery document, 119
- OpenID Connect format, 39
- SAML format, 26
- WS-Federation format, 29

MetadataEndpoint, 287

Microsoft.AspNet.WebApi.Owin NuGet package, 266

Microsoft Azure. *See* Azure Active Directory

Microsoft cloud service, 61

Microsoft Enterprise Agreement, 62

Microsoft.IdentityModel.Protocol.Extensions NuGet package, 84, 92

Microsoft Office 365. *See* Office 365

- Microsoft Online Directory Service (MSODS), 60
  - Microsoft.Owin.Diagnostics NuGet package, 154
  - Microsoft.Owin NuGet package, 92
  - Microsoft.Owin.Security.ActiveDirectory NuGet package, 83, 254
  - Microsoft.Owin.Security.Jwt NuGet package, 254
  - Microsoft.Owin.Security.OAuth NuGet package, 254
  - Microsoft.Owin.Security.OpenIdConnect NuGet package, 84
  - Microsoft.Owin.Security NuGet package, 92
  - Microsoft.Owin.Security.WsFederation NuGet package, 83
  - Microsoft Visual Studio. *See* Visual Studio
  - \_middleware entry, 141
  - middleware initialization options class, 155–159
  - middlewares
    - activation sequence, 142–145
    - behavior settings, 158–159
    - building, 138. *See also* Open Web Interface for .NET (OWIN)
    - caption setting, 159
    - context, 142, 145–148
    - environment dictionary, 138
    - initialization pipeline, 265–266
    - Invoke method, 142
    - message received notification, 164
    - observing pipeline, 143–145
    - pipeline of web APIs, 254–255
    - pointers to next entries, 142
    - requesting execution, 145
    - resource protectors, 74, 81
    - response handling, 161
    - security token received notification, 164
    - security token validated notification, 164–165
    - sign-in and sign-out flow, 99–103
    - skipping to next, 161
    - stopping processing, 142, 145
    - UseStageMarker method, 145
  - midtier clients ADAL libraries, 81
  - MMC (Microsoft Management Console), 60
  - mobile operating systems, native apps on, 80
  - modern authentication techniques, 31–48
  - multiple authentication factors (MFA), 122
  - Multiple Response Type specifications, 109
  - multiresource refresh tokens (MRRT), 242–243, 260
  - multitenancy, 205–211
  - MVC 5 Controller, 100
  - /myorganization alias, 237
- N**
- native applications, 47–48
    - ADAL libraries, 48, 80–81
    - ADFS support, 55
    - ADFS template, 275
    - admin creation in Azure portal, 204
    - authentication flows, 94
    - broker apps and, 48
    - development libraries, 75–76
    - Kerberos and, 47
    - modern authentication for, 294
    - popularity, 47–48
    - tokens, obtaining, 21–22
  - nbf claims, 132
  - .NET-based applications, 78
  - .NET core, OWIN middleware for, 84
  - .NET Framework
    - caller identity class, 7–10
    - SAML and, 25
    - version 4.5, 82
    - Windows Identity Foundation classes, 82–83
  - .NET JWT handler, 84
  - .NET web development, 138
  - network endpoints, 52
  - network tracing features, 110
  - nickname claims, 133
  - Node.js, 81
  - nonadmin users, application creation, 189–192. *See also* users
  - nonce value
    - of authentication requests, 117
    - cookie tracking, 124–125
    - OpenID Connect, 149
  - notifications, 159–166
    - AuthenticationFailed, 166
    - AuthorizationCodeReceived, 166
    - in bearer token middleware, 264
    - MessageReceived, 164
    - RedirectToIdentityProvider, 162–164
    - SecurityTokenReceived, 164
    - SecurityTokenValidated, 164–165
    - sequence, 159–161
    - of TokenCache class, 244–245
  - Notifications property, 155
  - NuGet packages
    - adding references, 92
    - Microsoft.AspNet.WebApi.Owin, 266
    - Microsoft.IdentityModel.Protocol.Extensions, 84, 92
    - Microsoft.Owin, 92
    - Microsoft.Owin.Diagnostics, 154
    - Microsoft.Owin.Security, 92
    - Microsoft.Owin.Security.ActiveDirectory, 254
    - Microsoft.Owin.Security.Jwt, 254
    - Microsoft.Owin.Security.OAuth, 254
    - Microsoft.Owin.Security.OpenIdConnect, 83
    - .NET, 227–228
    - System.IdentityModel.Tokens.Jwt, 84, 92
    - SystemWeb, 92
    - for web APIs, 254
    - web apps referencing, 92



**O**

- OAuth, 33–37
- OAuth2, 33–37
  - ADAL and, 76–77
  - ADFS “3” support, 272
  - authorization grants, 55, 252
  - bearer token usage, 232–237, 262
  - claims and, 36–37
  - client credentials grants, 44–45
  - endpoints, 64
  - ID token, 39
  - interoperability, 37
  - limitations, 118
  - Multiple Response Type, 109
  - “on-behalf-of” security token requests, 44
  - OpenID Connect extensions, 39, 110
  - permissions in applications, 183–185
  - Post Response Mode, 109
  - refresh token grants, 239–240
  - refresh tokens, 238–251
  - scope, 116
  - support for, 37
  - Token Exchange extensions on-behalf-of flow, 267–270
    - web sign-in, 37–39
  - oauth2AllowImplicitFlow property, 182
  - oauth2AllowUrlPathMatching property, 182
  - OAuth2 Authorization Framework specification, 110
  - OAuth2 bearer token middleware, 287
  - OAuth2 Bearer Token Usage specification, 110
  - oauth2PermissionGrants collection, 189–192
    - admin consent, 203–204
    - consent entries, 210
  - oauth2Permissions collection, 183–185, 188, 192–195
    - default entry for web APIs, 257
    - value property, 257–258
  - oauth2RequirePostResponse property, 182
  - OAuth WRAP (Web Resource Authorization Profile), 33, 40–41
- objectId property, 180, 188, 190
- odata parameters in URL template, 237
- Office 365, 61
  - cloud-based issues, 58
  - Visual Studio 2015 tools, 87
- oid claims, 133
- on-behalf-of flow, 267–270
  - security token requests, 44
- on-premises Active Directory, 15–16, 58–59
- on-premises directories
  - functional components, 60
  - querying protocols, 59
- OnValidateIdentity, 265
- opaque channels, 72, 91
- OpenID, 37–38
  - OpenID Connect, 9, 38–43, 108–109
    - authentication, 122–123
    - authentication-request message type, 39
    - authentication requests, 113–119
    - authorization-code flow, 42–43
    - authorization requests, 98, 149
    - discovery, 119–122
    - document format, 39
    - endpoints, advertising by Azure AD, 109
    - ID token, 127–134
    - initialization code, 95–97
    - JWT format, 129–132
    - nonce, 149
    - opaque channels, 91
    - response, 123–125
    - session management, 109
    - sign-in sequence, 110–112, 126–127
    - sign-out, 134–136
    - support for, 43
    - supporting specifications, 110
    - web sign-on with ADFS, 276–281
  - OpenIdConnectAuthenticationOptions class, 159
  - OpenIdConnectAuthenticationOptions parameter, 96, 155–159, 276
    - TokenValidationParameters property, 166–169
  - OpenID Connect Core 1.0, 108
  - OpenID Connect Discovery 1.0, 109
  - OpenID Connect hybrid flow, 40–42, 224–232
  - OpenID Connection Session Management specification, 109
  - OpenID Connect middleware, 92, 155–166
    - ADFS and, 276–277
    - authentication flow control, 96
    - authority value, 97
    - Challenge sequence, 152
    - client ID, 94, 97, 256
    - distributed sign-out, 101
    - initializing, 95–97, 277
    - notifications, 159–166
    - OpenIdConnectAuthenticationOptions, 155–159
    - outgoing 401s, 98
    - Passive authentication mode, 152, 159
    - postlogout redirects, 102
    - session management, 149–151, 171
    - sign-out, 100–101, 152–153
    - token validation, 149–151
    - TokenValidationParameters property, 166–169
  - OpenIdConnectNotifications class, 159–166
  - OpenIdConnectProtocolValidator class, 158
  - OpenID Connect Session Management specification, 135
  - openid scope, 116, 286
  - open redirector attacks, 182
  - open source libraries, 76
  - Open Web Interface for .NET (OWIN), 83–84, 137–138.
    - See also* middlewares

- ASP.NET-specific implementation, 138
- context, 145–148
- defined, 138
- environment dictionary, 138
- Katana and, 139–154. *See also* Katana
- Open Web Interface for .NET (OWIN) middlewares
  - adding to web apps configuration, 92
  - authentication capabilities, 146
  - authentication flow, 148–153
  - for claims-based identity, 83
  - core status, 147
  - diagnostic middleware, 153–154
  - environment dictionary, 147
  - hosting, 92, 95–96
  - for .NET core, 85
  - OpenID Connect, 137–170
  - sign-in flow, 148–152
  - sign-out flow, 152–153
  - WS-Federation support, 103
- Open Web Interface for .NET (OWIN) pipeline
  - adding middleware, 141–142
  - hosts, 140
  - initializing, 139–141
  - \_middleware entry, 141
  - servers, 140
- OS X apps, ADAL libraries, 81
- OwinMiddleware class, 142–143
- OwinStartup attribute, 139

## P

- parametric STS, 205–208
- password-based authentication, 12–14
- passwordCredentials property, 181, 188, 226
- passwords, 13–14
- password sharing antipattern, 32–33
- path matching, 182
- permissions
  - admin-level, 198–200
  - app-level, 216–219
  - on application entry in Azure AD, 224–225
  - consented, 186
  - delegated, 192–197
  - directory, 193–197
  - for directory access, 193–196
  - fine-grained, 59
  - granted, storage of, 189–192
  - roles and, 213
- Permissions To Other Applications, 259
- platform as a service (PaaS), 57
- postlogout redirects, 102, 156
- PostLogoutRedirectUri property, 102, 156, 276
- Post Response Mode specifications, 109
- pre-claims authentication techniques, 12–16

- principalType property, 212
- private/public key pairs, 227
- profile scope value, 116
- profile stores, 12–14, 20
- programmable web, 31–33
- Programming Windows Identity Foundation*, 82, 137
- prompt=admin\_consent flag, 200–201, 218
- prompt parameter, 117–118
- Properties dictionary, 140–141
- protected APIs. *See also* web APIs
  - accessing, 232–251
  - exposing, 253–272
  - refresh tokens, 238
- protected clients, 78
- protocol coordinates, 73–74
- protocol/credential type endpoints, 60
- protocol endpoints, 60, 63–64
- protocol enforcement, 73
- protocol libraries, 77
- protocol middleware. *See also* OpenID Connect
  - middleware
    - collaboration with cookie middleware, 148
- protocols, application identifiers, 181
- protocol URLs, 63, 94
- protocol validation, 158
- ProtocolValidator property, 158
- providers
  - claims issued, 9
  - specifying, 265
- provisioning
  - in ADFS, 271, 287
  - applications, 53–54, 57, 189
  - in Azure management portal, 93–94
  - just-in-time, 58
  - relying parties, 52
  - ServicePrincipal, 186
  - web APIs, 253
- provisioning flow, 175–176
- provisioning resources, 183
- proxy role, 52
- proxy utilities, 110
- publicClient property, 181
- public clients, 78, 275
- public key cryptography, 19
- public-private key pairs, 18
- publisherName property, 188
- pwd\_exp claims, 133
- pwd\_url claims, 133

## Q

- querying protocols, 59
- query response mode, 115

## reauthorization

### R

- reauthorization, 248–249
- redirects, 35
- RedirectToIdentityProvider notification, 160–164
  - modifying authentication requests, 201
- redirect URIs, 100, 115, 117, 135, 156, 180–181, 276
  - for web apps, 278
- RefreshOnIssuerKeyNotFound property, 158
- refresh token grants, 239–240, 242
- RefreshToken property, 230
- refresh tokens, 35, 238–251
  - in Azure AD, 240–242
  - expiration, 246–251
  - invalidating, 240
  - multiresource, 286
  - opacity to client, 242
  - validity times, 240
- relying parties (RPs), 18
  - distributed sign-out, 109
  - IdP metadata, 108
  - provisioning, 52
  - user sign-in status inquiries, 109
  - WS-Federation, 29
- relying party trusts, 274–276
- renewal operations, 71
- replyUrls property, 180–181
- Request and Response methods, 148–149
- Request and Response properties, 147–148
- requests
  - ClientAssertionCertificate, 231
  - client\_secret property, 227
  - interception, 73
  - redirect URI, 156
  - resource for authorization code, 156
  - response type, 156
  - scope parameter, 156
  - through GET operations, 182
  - through middleware pipeline, 138
  - token inclusion, 70
- requiredResourceAccess, 198–199
- RequiredResourceAccessCollection, 185–187
  - Role type entries, 218
- resource apps
  - configuring by IdP's metadata, 73
  - token acquisition, 73
  - token validation, 73
- resource consumption
  - identity of clients, 265–266
  - patterns, 43–45
- resource identifiers in token requests, 256
- resourceId property, 190
- Resource parameter, 118, 156, 231
- <resource path> component in URL template, 236–237
- resource protectors, 69, 73–74

- development libraries, 81–85
- interceptors, 74
- resources
  - accessing, 185–187
  - accessing as application, 44–45
  - access requests, 70–71. *See also* requests
  - authorizing access, 97–98
  - client libraries, 71–72
  - multiple, refresh tokens for, 242–243
  - third-party access, 34
  - type of access scope, 185–186
- resource STS, 205–206
- response mode and response type parameters of
  - authentication requests, 114–116
- Response object, 149–150
- responses
  - handling, 161
  - ID token, 127–134
  - OpenID Connect message, 123–125
  - through middleware pipeline, 138
- ResponseType parameter, 156
- response types, 124
- REST API calls, 233–235
- REST-based protocols, 28
- REST operations for directory queries, 59
- RoleClaimType property, 216
  - groups as, 220
- roles
  - allowedMemberTypes property, 214
  - application, 212–219
  - assigning, 213–214
  - claims, 133, 215
  - displayName and description strings, 214
  - id property, 214
  - value property, 214
  - WS-Federation, 28–29
- round trips
  - performance and, 45
  - request-response pattern, 22–23, 45
  - web apps, 23–31
- RS256 signatures, 131

### S

- samlMetadataUrl property, 182, 188
- SaveSignInToken property, 171, 268
- scope-driven authorization, 262–265
- scopes, 116, 118, 156, 201
  - openid, 286
  - of web APIs, 284–285
- security
  - HTTPS, 91
  - nonce values, 117
  - for web API calls, 46–47

- Security Assertion Markup Language (SAML), 8, 25–27, 55, 182
- security code, custom, 71
- security groups, 219
- SecurityTokenHandlers property, 158
- SecurityTokenReceived notification, 165
- Security Token Service (STS), 29
  - Access Control Service, 78–79
  - resource, 205–206
- SecurityTokenValidated notification, 165–166
- server applications, ADFS template, 275
- servers
  - in OWIN pipeline, 140
  - server-to-server calls, 42
- ServicePrincipal, 174–177, 187–188
  - AppId, 193
  - oauth2Permissions, 193–196
  - ObjectId, 193
  - properties, 187–188
  - provisioning, 186
  - for web APIs, 257
- ServicePrincipal.appRoleAssignedTo object, 216–219
- servicePrincipalNames property, 188
- service providers (SPs), 26
- session artifacts, 73
- session cookies, 24–25, 45, 92, 122
  - discarding, 135
  - in OpenID Connect hybrid flow, 42
  - persisting, 150
  - validation, 73, 125
- session data, 24
- session management, 70–71
  - by ADAL, 238–251
  - in OpenID Connect middleware, 109, 171
- sessions
  - ClaimsPrincipal, saving, 151
  - cleaning, 135
  - ending, 134–136
  - establishing, 22, 73, 149–151
  - properties, 151
  - request token validation, 152
  - saving, 150
  - validation, 73
- sessionStorage, 47
- Set-Cookie value, 135, 149–151
- shared secrets, 279–280, 287
- signatures, 19
- signature verification, SAML and, 26
- signed tokens, 20
- sign-in, 37–39, 99–103, 126. *See also* web sign-on
  - notifications, 159–160
  - response phase, 224–225
  - sequence, 126–127, 224
  - UI elements, 102–103
  - user credentials prompts, 163
- sign-in and sign-out flow, 110–112
- SignInAsAuthenticationType property, 159
- sign-in flow
  - access in context of session, 148, 152
  - challenge generation, 148–149, 152–153
  - OpenID Connect for, 107–134
  - response processing, 149–151
  - session generation, 149–151
  - specifications and dependencies, 107–108
  - WS-Federation, 29–31
- signing keys for web apps, 280
- sign-in messages
  - generation of, 149
  - redirects, 148–149
  - request generation, 73–74
- sign-out, 99–103
  - distributed, 101, 109
  - flow sequence, 136
  - ID hint, 135
  - notifications, 161
  - OpenID Connect, 134–136
  - postlogout redirects, 156
  - PostLogoutRedirectUri property, 102
  - redirect URI, 135
  - request syntax, 135
  - state preservation, 135
  - target endpoint, 135
  - UI elements, 102–103
  - user credentials prompts, 163
- sign-out flow, 152–153
- SignOut method, 99
- Simple Web Token (SWT), 40–41
- Single Logout messages, 27
- single-page applications (SPAs), 45–47, 294
  - ADAL JS library for, 85
- single sign-on, hack for, 38
- single sign-out, 27
- SkipToNextMiddleware method, 161
- software as a service (SaaS) apps, 17
- \_sso\_data claim, 289
- SSO sessions, 27
- stage markers, 145
- Startup.Auth.cs file
  - ADFS identity provider code, 103–104
  - identity pipeline initialization code, 96–97
- Startup class, 139–141
- Startup.Configure, 140
- Startup.cs file, 95
  - call to activate authentication, 97
- state, preserving at sign-out, 135
- state parameter
  - of authentication requests, 116–117
  - local URL of resource, 125
- storing tokens, 70–71
- string identifiers, 18

## sub claims

- string identifiers, *continued*
    - verification, 19
  - sub claims, 132
  - subjects, 25
  - symmetric keys, 19
  - synchronized deployments of Azure AD, 65
  - synchronizing users and groups to Azure AD tenants, 65–66
  - System.IdentityModel.Tokens.Jwt NuGet package, 84, 92
  - System.Security.Claims namespace, 7
  - SystemWeb NuGet package, 92
  - System.Web pipeline, 140
- ## T
- tags property, 188
  - target directories
    - consent, recording, 186
    - resource entries in, 183
  - target platforms, native libraries for, 81
  - <tenant> component in URL template, 236
  - tenant IDs, 63, 188, 230
  - Tenant parameter, 255
  - tenants
    - application availability, 182–183
    - defined, 62
    - display name, 188
    - federated and managed, 65–66
    - ServicePrincipal and, 176, 193
    - tenant IDs, 63, 188, 230
  - third-party access to resources, 34
  - Thread.CurrentPrincipal, 8
  - tid claims, 133
  - token acquisition, 70
    - ADAL pattern, 77
  - TokenCache class, 244–245
  - token endpoint, 35
    - authenticated requests against, 226
    - response to token request, 231–232
  - token handlers, 158
  - token replay attacks, 117
  - TokenReplayCache property, 170
  - token requestors, 69–72, 74
    - access token format and, 72
    - client applications as, 72
    - development libraries, 76–81
  - token requests, 70
    - on-behalf-of, 44
    - resource identifiers in, 256
    - user consent, 64
  - tokens, 18–20. *See also* access tokens
    - accessing independent of protocol, 268
    - acquiring by authorization code, 227–229
    - assertions, 26
    - audience claims, 282
    - in Authorization HTTP headers, 232–234
    - Azure AD, 61
    - bearer, 232–237, 262
    - broker apps, 48
    - caching, 70–71, 238
    - callback path, 158
    - for client-resource interactions, 70
    - cross-domain, 25
    - group information, 219–220
    - group membership claims, 182
    - HTTP carrier mechanisms, 109
    - ID. *See* ID tokens
    - issuance of, 21–22
    - issuers, 120
    - JWT format, 40, 129–132
    - life-cycle management, 159, 238
    - refresh, 35, 238–251, 286
    - replaying, 169
    - in requests, 70–71
    - response mode, 114
    - response type, 109
    - SAML structure, 26
    - saving, 169
    - scope, 257–258
    - security of, 42–43
    - signed, 20, 120–122
    - Simple Web Token, 40–41
    - with user attributes from cloud store, 59
    - user information, 42, 268
    - validation. *See* token validation
      - for web API calls, 46–47
  - token validation, 22, 73, 119, 149–151
    - audience, 167
    - discovery of criteria, 119–120
    - issuer, 167
    - key for signing, 167
    - notification of, 164–165
    - parameters, 166–169
    - signature check, 129–130
    - validation flags, 168
    - validator delegates, 168–169
    - validity interval, 167
  - TokenValidationParameters class, 155, 157, 167–170, 256–257, 261, 264
    - IssuerValidator property, 208
    - ValidIssuers property, 208
  - TraceOutput property, 148
  - traces
    - capturing, 110–112
    - exposing, 148
  - traffic, capturing in trace, 110–112
  - trusts, 18
    - between app and IdP, 20–21
    - establishment, 57

type identifiers, claim, 8–9

## U

UI, sign-in and sign-out, 102–103  
 unique identifiers, 18  
 unique\_name claims, 133  
 upn claims, 133  
 URI fragments, 46–47  
 UseCookieAuthentication method, 96, 141  
 UseErrorPage method, 154  
 Use method, 140, 142  
 UseOpenIdConnectAuthentication method, 96, 141  
 UserAssertion class, 268–269  
 user assignment, 211–213  
 user attributes, 7, 12  
 user consent for token requests, 64  
 user credentials. *See also* credentials  
   for synchronized deployments of Azure AD, 65  
   synching to cloud, 65–66  
 user\_impersonation permission, 196  
 UserInfo property, 42, 230  
 username-password-profiles authentication, 13–14  
 UserProfile.Read permission, 195–196  
 User.Read.All permission, 197  
 User.ReadBasic.All permission, 196  
 User.Read permission, 195–196  
 users  
   accessing web APIs, 252  
   application creation, 189–192  
   assignment, 211–213  
   authentication experience, 122–123  
   Azure AD landing page, 66  
   consent prompts, 191  
   identity, 12–13  
   life cycles, 14  
   roles and, 213  
 Use\* sequence, 143  
 UseStageMarker method, 145  
 UseTokenLifetime property, 159  
 UseWindowsAzureActiveDirectoryBearerAuthentication  
   method, 255, 271  
 UseXXX extension methods, 96

## V

validate-and-drop-a-cookie approach, 23–24  
 ValidateAudience property, 168  
 ValidateIssuer property, 168  
 ValidateIssuerSigningKey property, 168  
 validation  
   authority coordinates and, 157–158  
   components, 9  
   flags, 169

  of ID tokens, 133  
   issuer, 208–209  
   session, 73  
   of session cookies, 73  
   token, 73  
   validator delegates, 169–170  
 ValidAudience property, 167, 256  
 ValidIssuer property, 167  
 verification, 19, 24  
 Visual Studio  
   application credentials, assigning, 226  
   ASP.NET 4.6 Web API projects, 254, 287  
   authentication preferences settings, 288  
   Browser Link, 144  
   creating new web app, 90–91  
   F5 verification procedure, 91  
   identity-integration features, 86–87  
   Immediate window, 247  
   Multiple Startup Projects option, 288  
   MVC 5 Controller, 100  
   Package Manager Console, 92  
   Startup.cs file, 95  
   using directives, 98  
   web API project setup, 253–258  
   web API project template on-premises option, 271  
   Windows Identity Foundation tools, 82  
 Visual Studio 2013, 86  
 Visual Studio 2015, 2–3  
   accounts, associating with, 3  
   AD integration features, 86  
   keychain, 87  
   tying to Azure user account, 2–3

## W

web API calls  
   handling, 258–265  
   securing, 46–47  
 web APIs  
   access control policies, 283  
   accessing as an application, 251–252  
   accessing as arbitrary user, 252  
   application permissions for, 284  
   calling, 260  
   calling another API, 266–270  
   claims in token, 289–290  
   client access, 291  
   clients, adding, 291  
   client setup, 258–262  
   consent for, 258  
   directory entries, 257–258  
   exposing, 253–272  
   failed token requests, 261–262  
   identifiers, 282

## web applications

- web APIs, *continued*
  - invoking from web app, 223–252, 285–289
  - invoking with access tokens, 232–251
  - invoking with bearer tokens, 232–237
  - middleware pipeline, 254–255
  - modeling, 177
  - NuGet packages for, 254
  - project setup, 253–258
  - protecting with ADFS, 271–272, 281–292
  - request processing, 262–265
  - scope-driven authorization, 262–265
  - scopes of, 284–285
  - ServicePrincipal, 257
  - tokens, obtaining, 21–22
  - troubleshooting, 261
  - unauthorized caller errors, 261
- web applications
  - ADAL cache considerations, 243–246
  - ADFS as identity provider, 103–104
  - ADFS support, 55
  - application credentials, 279
  - authentication flows, 94
  - claims, 98
  - client ID, 94, 278
  - creating, 90–91
  - delegated access, 34–36
  - HTTPS, 91
  - hybrid role of token requestor and resource protector, 74–75
  - interaction pattern, 22–23
  - invoking web API from, 285–289
  - middlewares, adding and initializing, 95
  - OpenID Connect initialization code, 95–97
  - OWIN pipeline, adding, 95
  - Permissions To Other Applications, 259
  - protocol coordinates, 276, 278
  - redirect URIs, 278
  - referencing NuGet packages, 92
  - registering in Azure AD, 93–94
  - roundtrip-based request-response pattern, 22–23, 45
  - running, 98–99, 103
  - setup in ADFS, 277–280
  - shared secrets, 279–280
  - sign-in and sign-out, 99–103
  - signing keys, 280
  - sign-in message generation, 73
  - single-page applications, 46
  - single sign-on, 38
  - SSL Enabled, 91
  - third-party access to resources, 34
  - triggering authentication, 97–98
  - unique resource identifier, 94
  - user authentication, 21
  - web API, consuming, 223–252
  - Windows Integrated Authentication credential, 279
  - web browser-based SSO, 25–27
  - web.config files, 83
  - web servers, decoupling from apps, 138
  - web sign-on, 29–31. *See also* sign-in
    - ASP.NET support for, 137. *See also* Open Web Interface for .NET (OWIN) middlewares
    - hybrid authentication flow, 108
    - OpenID Connect Core 1.0, 108
    - with OpenID Connect in ADFS, 276–281
    - testing, 280–281
    - URLs, 94
  - web UX, exposing, 265–266
  - Wells, Dean, 15
  - WindowsAzureActiveDirectoryBearerAuthenticationOptions
    - initialization, 255
  - Windows Identity Foundation (WIF), 82–83
  - Windows Internal Database (WID), 52
  - Windows Server. *See also* Active Directory Federation Services (ADFS)
    - ADFS server role, 54
  - Windows Server 2016, ADFS in, 56, 103, 273–292
  - workplace-joined device detection, 56
  - WS-Federation, 8, 27–31
    - ADFS support, 55
    - messages, 29–31
    - metadata document format, 29
    - OWIN middlewares support, 103
    - relying parties, 29
    - roles, 28–29
    - Security Token Service, 29
    - sign-in flow, 29–31
    - support, 31
    - support in .NET core, 85
    - tokens, 29
  - WS-Federation middleware. *See* OpenID Connect middleware
  - WS-\* specifications, 27–28
    - native apps and, 47
  - WS-Trust, ADFS support, 55
  - Wtrealm, 104
  - WWW-Authenticate header, 262

## X

- X.509 certificates, 18–19
- Xamarin, 80

# About the author



**VITTORIO BERTOCCI** is principal program manager on the Azure Active Directory team, where he works on the developer experience: Active Directory Authentication Library (ADAL), OpenID Connect and OAuth2 OWIN components in ASP.NET, Azure AD integration in various Visual Studio workstreams, and other things he can't tell you about (yet).

Vittorio joined the product team after years as a virtual member in his role as principal architect evangelist, during which time he contributed to the inception and launch of Microsoft's claims-based platform components (Windows Identity Foundation, ADFS 2.0) and owned SaaS and identity evangelism for the .NET developers community.

Vittorio holds a masters degree in computer science and began his career doing research on computational geometry and scientific visualization. In 2001 he joined Microsoft Italy, where he focused on the .NET platform and the nascent field of web services security, becoming a recognized expert at the national and European level.


In 2005 Vittorio moved to Redmond, where he helped launch the .NET Framework 3.5 by working with Fortune 100 and Global 100 companies on cutting-edge distributed systems. He increasingly focused on identity themes until he took on the mission of evangelizing claims-based identity for mainstream use. After years of working with customers, partners, and the community, he decided to contribute the experience he had accumulated back to the product and joined the identity product team.

Vittorio is easy to spot at conferences. He has spoken about identity in 23 countries on four continents, from keynote addresses to one-on-one meetings with customers. Vittorio is a regular speaker at Ignite, Build, Microsoft PDC, TechEd (US, Europe, Australia, New Zealand, Japan), TechDays, Gartner Summit, European Identity Conference, IDWorld, OreDev, NDC, IASA, Basta, and many others. At the moment his Channel 9 speaker page at <https://channel9.msdn.com/events/speakers/vittorio-bertocci> lists 44 recordings.

Vittorio is a published author, both in the academic and industry worlds, and has written many articles and papers. He is the author of *Programming Windows Identity Foundation* (Microsoft Press, 2010) and coauthor of *A Guide to Claims-Based Identity and Access Control* (Microsoft patterns & practices, 2010) and *Understanding Windows CardSpace* (Addison-Wesley, 2008). He is a prominent authority and blogger on identity, Azure, .NET development, and related topics: he shares his thoughts at [www.cloudidentity.com](http://www.cloudidentity.com) and via his twitter feed, <http://www.twitter.com/vibronet>.

Vittorio lives in the lush green 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 place on the beach, be it the South Pacific or Camogli, his home town in Italy.





Now that  
you've  
read the  
book...

Tell us what you think!

Was it useful?

Did it teach you what you wanted to learn?

Was there room for improvement?

**Let us know at <http://aka.ms/tellpress>**

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!

