Directory Enabled Applications Succinctly

by Giancarlo Lelli

Syncfusion | Technology Resource Portal

Directory Enabled Applications Succinctly

By
Giancarlo Lelli

Foreword by Daniel Jebaraj

Syncfusion
Deliver innovation with ease

Copyright © 2015 by Syncfusion Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Pedro Gomes
Copy Editor: Benjamin S. Ball
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books 7

Information is plentiful but harder to digest 7

The *Succinctly* series 7

The best authors, the best content 7

Free forever 8

Free? What is the catch? 8

Let us know what you think 8

About the Author 9

Chapter 1 Introduction 10

The Active Directory Service Interface (ADSI) 10

Event Tracing in ADSI 10

Active Directory Service Interface Architecture 11

Providers Supported by Active Directory Service Interface 11

Chapter 2 Active Directory 13

What is Active Directory 13

Active Directory Structure 13

Chapter 3 The LDAP Protocol 15

Generic Definition 15

LDAP Data Interchange Format 15

Differences Between LDAP 2 and LDAP 3 16

Chapter 4 Description of the Test Environment 17

Tooling 17

Environment 17

Chapter 5 System.DirectoryServices 18

Main Classes and Basic Concepts 18

The DirectorySearcher Class 18

The FindOne Method 19

The FindAll Method 19

The ClientTimeout Property 19

The ServerTimeLimit Property 19

The PropertiesToLoad Property 19

The Tombstone Property 19

The DirectoryEntry Class 19

Creating Child Nodes 20

The CommitChanges Method 20

The CopyTo Method 20

The DeleteTree Method 21

The Invoke, InvokeGet, and InvokeSet Methods 21

The MoveTo Method 21

The Rename Method 21

The SearchResult Class 22

The Path Property 22

The Properties Property 22

The GetDirectoryEntry Method 22

The SearchResultCollection Class 23

The UserPrincipal Class 23

The FindByBadPasswordAttempt Method 23

The FindByExpirationTime Method 23

The FindByLockOutTime Method 24

The FindByLogonTime Method 24

The PrincipalContext Class 24

The GroupPrincipal and ComputerPrincipal Classes 25

The MatchType Enumeration 25

Chapter 6 Code Samples 27

Connecting to the ADSI Service 27

Managing Users and Groups 28

Listing User and Groups 28

Editing Object Properties 30

Check the Existence of an Object 32

Creating Objects 33

Deleting Objects 34

Managing User Password 35

Adding User to a Group 37

Remove User from Group 37

Enumerate Directory Entry Settings 38
Error Handling with the Invoke Method in .NET..39
Handling Account Expiration...40
Setting User Account Expiration..40
Security Descriptor Management 41
Chapter 7 Implementing Single Sign-On (SSO) 44
Introduction 44
A Simpler Definition of Single Sign-On 47
Wrapping Up the Benefits and Security Concerns of SSO..47
Distinguishing Different Types of SSO...48
Implementing SSO with Microsoft Azure AD...49
Introducing Azure Active Directory 49
Creating an Azure AD 49
Creating a New Test User in the Azure AD 51
Integrating Our On-Premise Active Directory User in Azure AD 54
Important Notes About Syncing..60
Leveraging SSO Functionalities in Web Applications..62
An MVC Application That Authenticates Against Our AD 63
Building a Secure Web API Endpoint...73
A Brief Introduction to ADAL (Active Directory Authentication Library)...................................80
Conclusion 85

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Giancarlo Lelli is a passionate client and web developer focused on the Microsoft platform and technology stack.

He is currently studying for his Bachelor of Science in Computer Science at the University of Cagliari. In 2012, he earned the degree of expert accountant and programmer. In recent years, he has been a Microsoft Student Partner focused on providing support and technical training to students.

Giancarlo writes technical articles for the popular Italian community ASPItalia and leads a local user group based in Cagliari called Italian Developer Connection, where he organizes technical events about Microsoft technologies. Giancarlo has taken part as a speaker or as a logistic supporter in events such as Community Days, Startup Revolutionary Road, and Microsoft DevCamps.

Giancarlo currently works as a junior developer and IT consultant at iCubed, focusing on building client applications for the Windows 8.x and Windows Phone 8.x platforms, as well as providing consultancy on cloud solutions like Office 365.

Chapter 1 Introduction

The Active Directory Service Interface (ADSI)

The Active Directory Service Interfaces (ADSI) is a set of Component Object Model (COM) interfaces used to access the features of directory services from different network providers and from within client applications. ADSI enables developers and network administrators to easily manage network resources by providing a way to automate tasks. From a net-admin perspective, these tasks can include user management and printing device configuration, whereas from a developer perspective, ADSI provides a way to develop directory-enabled applications. Administrators and developers handle a single set of directory service interfaces, regardless of which directory services are installed.

A common scenario where ADSI is the solution to many problems is while working on a distributed computing environment; in this type of environment, we have multiple directory services and we have to find a way to access and manage all the resources inside of it. To address those problems, ADSI provides a consistent set of interfaces that can be used to seamlessly manage any directory contained within the environment.

The ADSI offers many benefits to system administrators, benefits that span between the simplicity of learning and implementation to the flexibility of usage. Listed below are some possible benefits that ADSI offers:

- Open: Any directory provider can implement ADSI.
- Directory Services (DS) Independent: It can be used without restrictions on the directory service provider.
- Multi-language support: Interfaces are accessible from multiple language (VB, C#, C++).
- Simple programming model.
- Scriptable: Can be used by any automation capable language, such as VBScript.
- Functionally rich: ISV have the chance to deploy serious and functionality rich applications.

Event Tracing in ADSI

Event tracing in ADSI is a feature that has been added to certain areas of the ADSI LDAP provider because of their complex under-the-hood implementation. This tracing makes it easier for developers to troubleshoot problems during the development of directory-enabled applications. Some of these areas include:

- Schema parsing and downloading.
- Changing and setting passwords.

Active Directory Service Interface Architecture

Figure 1 shows a COM-style diagram of the ADSI architecture. As we can see, at the top-level we have a system object that contains a list of all the ADSI providers currently installed. Each entry in the list is in turn a namespace Object specific for a particular ADSI provider. At the top-level root nodes of each namespace container object, we have defined whatever directory-system object the directory service uses. ADSI supplies a set of predefined objects and interfaces so that client applications can interact with directory services using a uniform set of methods. However, ADSI may not provide access to all features of a directory service. The root-node container objects, found within each provider Namespace object, include an ADSI schema container object. This object contains the definition of all features for that provider.

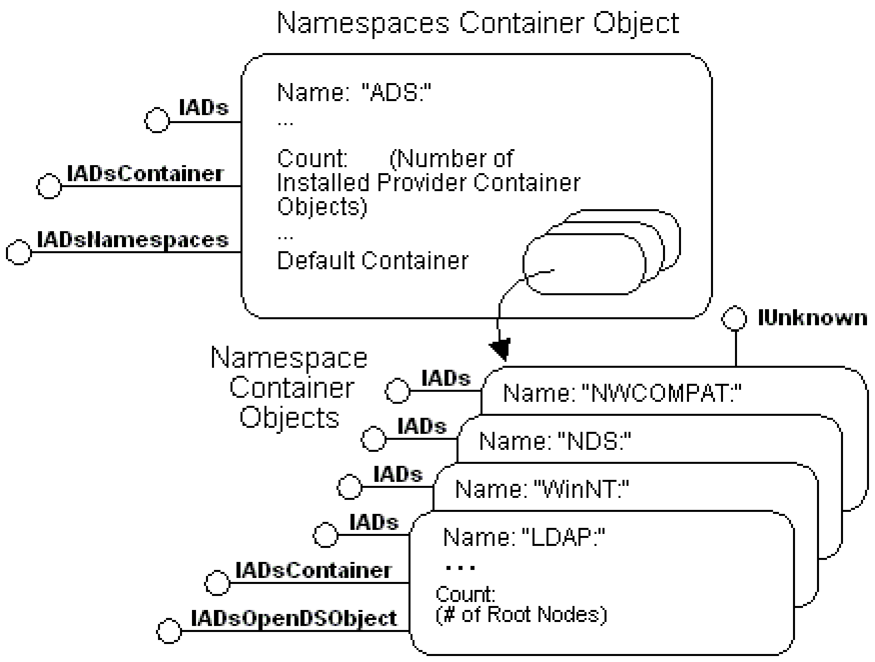


Figure 1: A diagram showing the namespaces container object

Providers Supported by Active Directory Service Interface

Service provider	Description
LDAP	Namespace implementation compatible with Lightweight Directory Access Protocol.
WinNT	Namespace implementation compatible with Windows.
NDS	Namespace implementation compatible with Novell NetWare Directory Service.
NWCOMPAT	Namespace implementation compatible with Novell NetWare 3.x.

Table 1: Supported ADSI

Not every service provider supports the methods and property methods exposed by ADSI interfaces. Because different directory services vary in the types of objects and properties stored, and use different protocols and authentication, ADSI is designed to work seamlessly with supported service providers. Thus, there are interfaces, methods, and property methods that work with one service provider, such as LDAP, that may not work on another, such as WinNT.

Chapter 2 Active Directory

What is Active Directory

Active Directory (AD) is a directory service that Microsoft built for Windows domain networks; it is included in most Windows Server operating systems as a set of processes and services. A server may act as domain controller (DC), where its roles are to authenticate and authorize all users and computers inside the Windows domain, and to assign and enforce security policies for all users and computers; these policies may be about OS updates or software restrictions.

For example, when a user logs into a computer that is part of a Windows domain, Active Directory checks the submitted password and determines whether the user is a system administrator or a normal user. Active Directory makes use of Lightweight Directory Access Protocol (LDAP) versions 2 and 3, Microsoft's version of Kerberos.

Active Directory Structure

As a directory service, Active Directory is mainly composed of a database and some executable binaries responsible for servicing all the incoming requests. The executable part is known as Directory System Agent (DSA), a collection of Windows services and processes. You can access the AD object in many ways, but the most common way is with an LDAP protocol or with ADSI.

Inside a generic AD, we can find a series of object that can be classified as the following:

- **Objects:** An object represents a single entity (user, computer, printer, or group) and its attributes. Certain objects can contain other objects and they are uniquely identified by their name and attributes.
- **Forests, Trees, and Domain:** The forest, tree, and domain are the logical divisions in an Active Directory network. Within a deployment, objects are grouped into domains. A domain is defined as a logical group of network objects (computers, users, devices) that share the same active directory database. A tree is a collection of one or more domains and domain trees in a contiguous namespace. At the top of the structure is the forest. A forest is a collection of trees that shares a common configuration. The forest represents the security boundary within which users, computers, groups, and other objects are accessible.
- **Organizational units:** The objects held within a domain can be grouped into Organizational Units (OUs). OUs can provide hierarchy to a domain. The OU is the recommended level at which to apply group policies (formally named Group Policy Objects (GPOs)).

Note: Microsoft recommends using OUs rather than domains for structure and to simplify the implementation of policies and administration.

There is a hierarchy classification between these objects. The diagram below provides a visual representation of this classification.

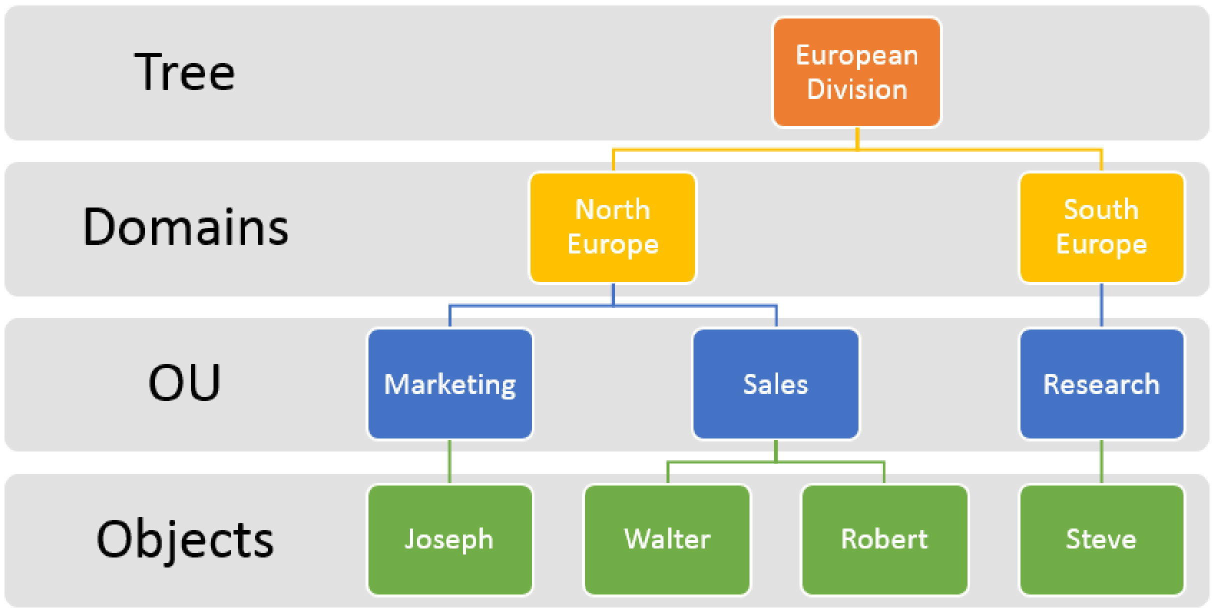


Figure 2: Chart showing the hierarchy of AD objects

Chapter 3 The LDAP Protocol

Generic Definition

The Lightweight Directory Access Protocol (LDAP) is an industry standard application protocol for accessing and maintaining distributed directory information services over an IP network. In matters of Active Directory, LDAP is the protocol we are going to use to talk with our AD. One important fact to point out is that LDAP cannot create directories or specify how a directory service operates. Over the years, LDAP has improved, and by the time of writing it has reached version 3 of the protocol; for more information about this release see RFC4511.

LDAP Data Interchange Format

The LDAP Data Interchange Format (LDIF) is a standard plain text data interchange format for representing LDAP (Lightweight Directory Access Protocol) directory content and update requests. LDIF conveys directory content as a set of records, one record for each object (or entry). It also represents update requests, such as Add, Modify, Delete, and Rename, as a set of records, one record for each update request. LDIF defines a set of fields used to identify the resource we want to deal with. Those fields are:

- **dn: distinguished name**: This refers to the name that uniquely identifies an entry in the directory. It is composed of dc's, ou's, and a cn.
- **dc: domain component**: This refers to each component of the domain. For example, www.google.com would be written as DC=www, DC=google, DC=com.
- **ou: organizational unit**: This refers to the organizational unit (or sometimes the user group) that the user is part of. If the user is part of more than one group, you may specify as such, e.g., OU= Lawyer, OU= Judge.
- **cn: common name**: This refers to the individual object (person's name; meeting room; recipe name; job title; etc.) for which you are querying.

Microsoft, since Windows Server 2003, provides a utility called LDIFDE that helps to construct LDIF queries. It is available if you have the AD DS or Active Directory Lightweight Directory Services (AD LDS) server role installed. To use LDIFDE, you must run the LDIFDE command from an elevated command prompt. You can find an example of LDIF file c:

```
dn: cn=Peter Michaels, ou=Artists, l=San Francisco, c=US
changetype: modify
add: telephonenumber
telephonenumber: +1 415 555 0002
```

The first line of this script identifies the object, in this case a user, we want to interact with. In order to avoid cases of homonymy, we try to be as precise as we can by specifying its organization unit (OU). The parameters “l” and “c” stand respectively for location and country. Both attributes are specified in the user object. The second line specifies the type of action we want to accomplish with this script; in this case, we want to modify the user by adding (line 3) its telephone number (line 4). Note that the attribute telephone number is a “standard” attribute supported by the user object.

Of course, as we said before, LDIF supports more than one ChangeType operation. Here below is a list of all the ChangeType operations supported.

ChangeType	Description
Add	Specifies that new content is contained in the import file.
Modify	Specifies that existing content has been modified in the import file.
Delete	Specifies that content has been deleted in the import file.

Table 2: Possible types of changes in LDIF

Differences Between LDAP 2 and LDAP 3

LDAP 3 defines a number of improvements that allow a more efficient implementation of the Internet directory user agent access model. These changes include:

- Use of UTF-8 for all text string attributes to support extended character sets.
- Operational attributes that the directory maintains for its own use; for example, to log the date and time when another attribute has been modified.
- Referrals allow a server to direct a client to another server that may have the data that the client requested.
- Schema publishing with the directory, allowing a client to discover the object classes and attributes that a server supports.
- Extended searching operations to allow paging and sorting of results, and client-defined searching and sorting controls.
- Stronger security through an SASL-based authentication mechanism.
- Extended operations, providing additional features without changing the protocol version.

LDAP 3 is compatible with LDAP 2. An LDAP 2 client can connect to an LDAP 3 server (this is a requirement of an LDAP 3 server). However, an LDAP 3 server can choose not to talk to an LDAP 2 client if LDAP 3 features are critical to its application.

Chapter 4 Description of the Test Environment

During the course of this book, a series of code examples are going to be provided that will enable you to reproduce the scenarios described. The following sections describe a test environment and the tools used to produce the demo code.

Tooling

All the code samples are going to be written in C# using Visual Studio 2014 CTP2 installed on a VM hosted on Azure. This presumes that you are familiar with C# syntax. However, to avoid misunderstanding, the code will be explained.

Environment

We are going to perform our AD interrogation to a Windows Server 2012 R2 Datacenter Edition virtual machine hosted on Microsoft Azure. Of course, this VM will be already configured with the Active Directory and DNS role. The virtual machine will be located in West Europe.

Tip: In order for you to test the code samples by yourself it is important that your domain controller and development machine are in the same LAN, otherwise they won't be able to communicate with each other (unless you expose the AD service to external users). However, exposing the AD services to the internet is something that should be avoided. If you are hosting the Virtual Machines in Azure, you can set them to be in the same LAN by using the same cloud service for both.

Chapter 5 System.DirectoryServices

Main Classes and Basic Concepts

The **System.DirectoryServices** is a namespace that contains classes and methods that allow us to access an Active Directory from managed code. The core components of this namespace are the **DirectoryEntry** and **DirectorySearcher** classes; further on, we will focus specifically on each of these classes, providing some more information about their methods and, more importantly, their core functionality. Both of these classes use the ADSI technology, and both can be used with any Active Directory Domain Services service providers. The current providers are Internet Information Services (IIS), Lightweight Directory Access Protocol (LDAP), Novell NetWare Directory Service (NDS), and WinNT.

The DirectorySearcher Class

You can use the DirectorySearcher object to search and perform query against any LDAP-compliant server, and in particular to an Active Directory Domain Service. LDAP is the only system supplied ADSI provider that supports directory searching. When creating a DirectorySearcher, you specify the root you want to retrieve and an optional list of properties to retrieve. This task is carried out by the **SearchRoot** property.

The SearchRoot property enables you to set additional properties to perform the following tasks:

- Cache the search results on the local computer. Set the **CacheResults** property to true to store directory information on the local computer. Updates are made to this local cache only when the **DirectoryEntry.CommitChanges** method is called.
- Specify the length of time to search, using the **ServerTimeLimit** property.
- Retrieve attribute names only. Set the **PropertyNamesOnly** property to true to retrieve only the names of attributes to which values have been assigned.
- Perform a paged search. Set the **PageSize** property to specify the maximum number of objects that are returned in a paged search. If you do not want to perform a paged search, set the PageSize property to zero (this is the default).
- Specify the maximum number of entries to return using the **SizeLimit** property. If you set the **SizeLimit** property to its default of zero, then the server-determined default is 1,000 entries.

Note: If the maximum number of returned entries and time limits exceed the limitations that are set on the server, the server settings override the component settings.

The FindOne Method

The FindOne method executes the search and returns only the first entry that is found. If more than one entry is found during the search, only the first entry is returned. If no entries are found to match the search criteria, a null reference is returned.

The FindAll Method

The FindAll method executes the search and returns a collection of the entries that are found. However, due to the diversity of providers we could deal with, there is the chance that this operation may not complete successfully; in that case, a NotSupportedException is thrown. Another unsuccessful scenario is when we try to perform a search inside an object that is not a container; in this case the FindAll method raises an InvalidOperationException.

The ClientTimeout Property

This property represents the maximum amount of time that the client waits for the server to return results. If the server does not respond within this time, the search is aborted and no results are returned.

The ServerTimeLimit Property

This property gets or sets a value indicating the maximum amount of time the server spends searching. If the time limit is reached, only entries that are found up to that point are returned.

The PropertiesToLoad Property

This property is a collection of string (the exact type name is StringCollection) that gets a value indicating the list of properties to retrieve during the search.

The Tombstone Property

This property gets or sets a value indicating whether the search should also return deleted objects that match the search filter. This option turns out to be very helpful if you don’t know whether the object inside the AD has been deleted or is still present.

The DirectoryEntry Class

You can use the DirectoryEntry class for binding to Active Directory objects to take advantage of functionalities like reading and updating attributes. This class, along with some helper classes, provides support for lifecycle management and navigation methods.

By lifecycle, we mean the complete management of a child node (creating, deleting, renaming, and moving); you also have the ability to enumerate all the children of a node. This is useful when you are considering a group of objects such as users, computers, or printers. In order for the changes to be propagated to the AD tree, you must commit your changes. As with our case scenario (the AlphaData Company), real world companies are in continuous expansion, adding new employees or computers to their forests or domains; the ADSI technology provides ways to programmatically add these objects to the directory tree.

Creating Child Nodes

To create a directory entry in the hierarchy, use the Children property. The Children property is a collection that provides an Add method through which you add a node to the collection directly below the parent node that you are currently bound to.

When adding a node to the collection, you must specify a name for the new node and the name of a schema template that you want to associate with the node. For example, you might want to use a schema titled "Computer" to add new computers to the hierarchy. This class also contains attribute caching, which can be useful for optimizing network traffic. To use attribute caching, see the **UsePropertyCache** property.

The CommitChanges Method

This method saves changes that are made to a directory entry to the underlying directory store.

Note: If you call RefreshCache before calling CommitChanges, any uncommitted changes to the property cache will be lost.

Tip: Use the DirectoryServicesPermission class for reading, writing, deleting, changing, and adding to the Active Directory Domain Services hierarchy. Associated enumeration: DirectoryServicesPermissionAccess.

The CopyTo Method

The first overload of the CopyTo method, which only takes another DirectoryEntry as a parameter, creates a copy of this entry as a child of the specified parent. The second overload takes a string as a parameter that specifies the name of the newly copied node. This method returns an instance of the copied DirectoryEntry and raises an InvalidOperationException in case the specified DirectoryEntry you want to copy into is not a container. In order to reflect your changes to the AD tree, you must call the CommitChanges method.

The DeleteTree Method

This method deletes the entry and its entire sub-tree from the Active Directory Domain Services hierarchy. Of the Active Directory Service Interfaces (ADSI) system-supplied providers, only the Lightweight Directory Access Protocol (LDAP) currently supports this operation.

The Invoke, InvokeGet, and InvokeSet Methods

These methods are used to invoke native methods or getter and setter properties from managed code.

1. The **Invoke** method takes the name of the method to invoke and an array of objects that defines the list as its parameters.
2. The **InvokeGet** method takes the name (as a string) of the property you want to get the value of.
3. The **InvokeSet** method differs only in the method firm; it tasks an array of objects that are used as possible values of the property you specify.

In case of failure, the Invoke methods may throw two types of exceptions. The first is the DirectoryServicesCOMException, when the native method throws a COMException. The second is a TargetInvocationException, when the native method throws a TargetInvocationException exception. The InnerException property contains a COMException that contains information about the actual error that occurred.

The MoveTo Method

Like the **CopyTo** method, the **MoveTo** method has two overloads. The first one only takes a DirectoryEntry and simply move the calling DirectoryEntry inside the DirectoryEntry passed as parameter. The second overloads like in the CopyTo case, takes a string parameter that specifies the name to use in the new DirectoryEntry parent. The MoveTo method throws a **InvalidOperationException** in case that the DirectoryEntry passed as parameter is not a container.

The Rename Method

The Rename method changes the name of this DirectoryEntry object.

Note: This will also affect the path that is used to refer to this entry.

The SearchResult Class

The **SearchResult** class encapsulates a node in the Active Directory Domain Services hierarchy that is returned during a search through DirectorySearcher. Instances of the SearchResult class share a similar way to access object properties as the DirectoryEntry class. The crucial difference is that the DirectoryEntry class retrieves its information from the Active Directory Domain Services hierarchy each time a new object is accessed, whereas the data for SearchResult is already available in the **SearchResultCollection**, where it gets returned from a query that is performed with the DirectorySearcher class. Only those properties that are specified through the DirectorySearcher.PropertiesToLoad collection in your query will be available from SearchResult.

The Path Property

Gets the path for this SearchResult. The Path property uniquely identifies this entry in the Active Directory Domain Services hierarchy. The entry can always be retrieved using this path.

The Properties Property

This gets a ResultPropertyCollection collection of properties for this object; of course, this collection only contains properties that were explicitly requested through DirectorySearcher.PropertiesToLoad. The ResultPropertyCollection inherits from the DictionaryBase class and so its members can be accessed by using the <key, value> notation with the help of the Item property. Inside the ResultPropertyCollection we also have fields that can be used to obtain the number of items contained in the dictionary, all the property names (that were previously specified through DirectorySearcher.PropertiesToLoad), keys, and values.

The GetDirectoryEntry Method

Retrieves the DirectoryEntry that corresponds to the SearchResult from the Active Directory Domain Services hierarchy.

Note: Calling GetDirectoryEntry on each SearchResult returned through DirectorySearcher can be slow.

The SearchResultCollection Class

The **SearchResultCollection** class contains the **SearchResult** instances that the Active Directory hierarchy returned during a DirectorySearcher query. Since it inherits from a collection, it has the Count and Item properties that respectively represent the number of items contained in the collection and the items itself inside the collection (given a valid index). Even inside the SearchResultCollection, using the field PropertiesLoad we can get the list of attributes explicitly specified in DirectorySearcher.PropertiesToLoad.

Note: Due to implementation restrictions, the SearchResultCollection class cannot release all of its unmanaged resources when it is garbage collected. To prevent a memory leak, you must call the Dispose method when the SearchResultCollection object is no longer needed.

The UserPrincipal Class

The **UserPrincipal** class encapsulates principals that are user accounts. We can define principals as AD objects. We can use this class to represent a user object starting from the more generic class DirectoryEntry. The UserPrincipal class has a primary constructor that takes as input a reference to the PrincipalContext (see next chapter) and in its variant a username a password and a Boolean flag.

The UserPrincipal class is contained inside the **System.DirectoryService.AccountManagement** namespace.

The user principal class exposes some very useful static methods that allow us to perform search operations based on specific properties or criteria. All of these static methods take as parameter a PrincipalContext object, a MatchType flag that defines the search criteria, and a third parameter that differs from method to method; some of them are explained below.

The FindByBadPasswordAttempt Method

This method searches for users who have a record of bad password inside the DateTime range specified as parameter; the type of match can be defined using the MatchType enumeration. This method returns an empty PrincipalResultCollection of UserPrincipal in case no results are found.

The FindByExpirationTime Method

This method, given a DateTime range, returns a **PrincipalSearchResult** object of **UserPrincipal** that has an account expiration date that satisfies the filter imposed by the MatchType enumeration.

The FindByLockOutTime Method

This method returns a **PrincipalSearchResult** object of **UserPrincipal** for users that have an account lockout time in the specified date and time range. Even in this case, the **MatchType** property defines the search criteria and an empty collection is returned in case of zero results.

The FindByLogonTime Method

This method returns a **PrincipalSearchResult** collection of UserPrincipal objects for users that have account logon recorded in the specified date and time range.

Besides these static methods, the UserPrincipal class makes it easier for us to perform actions on user objects such as changing/expiring their password, checking group membership, and listing group membership. Furthermore the, UserPrincipal class encapsulates in a more object-oriented way the property getter of a user object.

Note: More information about the UserPrincipal class can be found on the official MSDN page.

The PrincipalContext Class

This class encapsulates the server or domain against which all operations are performed, the container that is used as the base of those operations, and the credentials used to perform the operations. This class alone doesn't expose any particular method or property, but it has a valuable method that can state if a set of credentials (username and password) is valid. This validation is performed by executing a connect request to the server that is specified as parameter in the method's signature.

```
PrincipalContext principalContext = null;
try
{
    principalContext = new PrincipalContext(ContextType.Domain, IP_PORT,
"CN =Users,DC=alphadata,DC=adds");
    bool isValid = principalContext.ValidateCredentials("user", "pwd");
}
catch (PrincipalException e)
{ Console.WriteLine("Failed to create PrincipalContext. Exception: " + e);
return; }
```

The **ContextType** parameter can be Machine, Domain, or ApplicationDirectory. Notice how everything is wrapped inside a try-catch clause due to the fact that a PrincipalException can be thrown, for example, when the connection to the server was unsuccessful. The **IP_PORT** parameter is a constant that contains the local IP address of the AD server and the LDAP port (389).

The GroupPrincipal and ComputerPrincipal Classes

Since AD allows us to manage groups and computers, the System.DirectoryServices.AccountManagement namespace provides two classes that inherit from the abstract class named Principal to manage them. These classes are GroupPrincipal and ComputerPrincipal, and as we have seen with the UserPrincipal class, their main function is to wrap in an easier and more object-oriented way the AD/DirectoryEntry object. In this case, it can be a group or a computer. According to the type of object the class refers to, you have different methods and property exposed publicly. Remarking the fact that both GroupPrincipal and ComputerPrincipal inherit from the same base class as UserPrincipal, we shall not keep describing their public (and static) methods, since they are similar and in some case equal to the one discussed earlier in the chapter.

Note: In case you want to read in detail about the GroupPrincipal and ComputerPrincipal class please refer to these links:

GroupPrincipal: http://msdn.microsoft.com/en-us/library/system.directoryservices.accountmanagement.groupprincipal(v=vs.110).aspx

ComputerPrincipal: http://msdn.microsoft.com/en-us/library/system.directoryservices.accountmanagement.computerprincipal(v=vs.110).aspx

The MatchType Enumeration

While performing query in a **PrincipalContext** using static methods, it is necessary to specify a particular flag that defines the type of match we want to consider. The table below lists all the possible values of the **MatchType** enumeration.

Member Name	Description
Equals	The search results include values that equal the supplied value. If the supplied value specifies a date and time, the returned collection includes objects that have the same date and time.

Member Name	Description
GreaterThan	The search results include values that are greater than the supplied value. If the supplied value specifies a date and time, the returned collection includes objects that are dated after the specified date and time.
GreaterThanOrEquals	The search results include values that are greater than or equal to the supplied value. If the supplied value specifies a date and time, the returned collection includes objects that are dated on or after the specified date and time.
LessThan	The search results include values that are less than the supplied value. If the supplied value specifies a date and time, the returned collection includes objects that are dated prior to the specified date and time.
LessThanOrEquals	The search results include values that are less than or equal to the supplied value. If the supplied value specifies a date and time, the returned collection includes objects that are dated prior to or on the specified date and time.
NotEquals	The search results include values that are not equal to the supplied value. If the supplied value specifies a date and time, the returned collection includes objects that do not include the specified date and time.

Table 3: The list of members of the MatchType enumeration

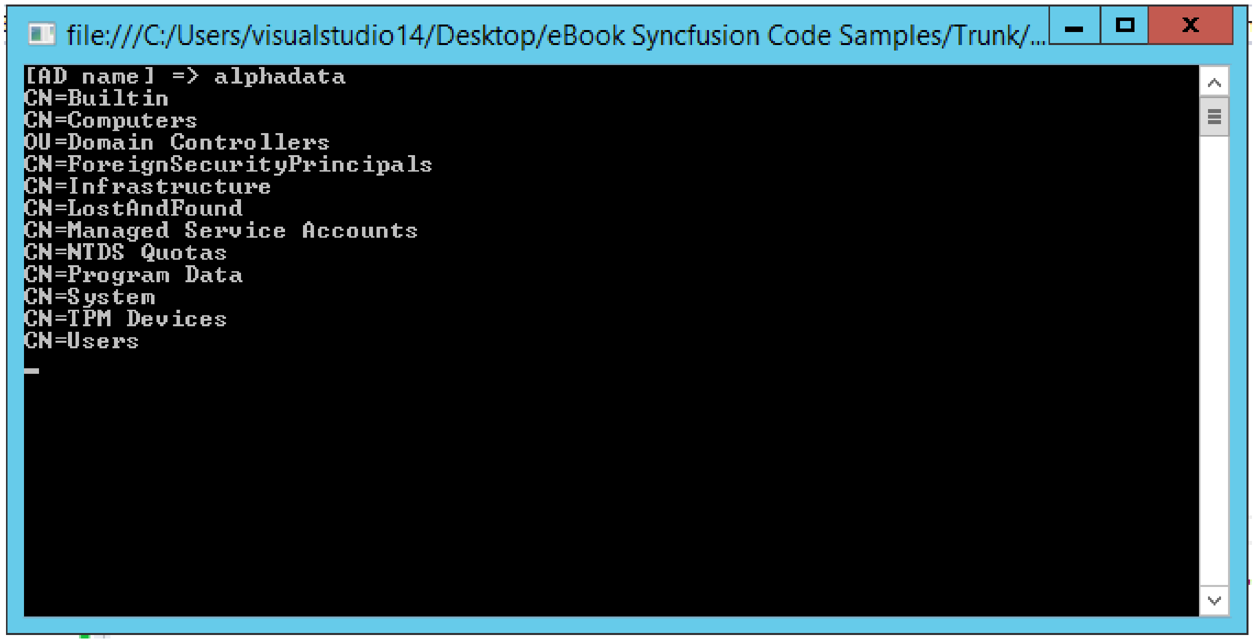
Chapter 6 Code Samples

Now that we have widely gone into all the theorist aspects of the System.DirectoryServices namespace, it is time to see some code examples showing how to accomplish some management task.

Connecting to the ADSI Service

Before we begin, we first need to connect to our AD service. Here's the code showing how to do that:

```
private static void Connection()
{
    DirectoryEntry conn = new DirectoryEntry("LDAP://"+ IP_PORT
+"/DC=alphadata, DC=adds", "user", "pass");
    Console.WriteLine("[DC name] => " + conn.Properties["dc"].Value);
    foreach (DirectoryEntry item in conn.Children)
        Console.WriteLine(item.Name); // We print all the children we have
in our root
}
```



file:///C:/Users/visualstudio14/Desktop/eBook Syncfusion Code Samples/Trunk/...

```
[AD name] => alphadata
CN=Builtin
CN=Computers
OU=Domain Controllers
CN=ForeignSecurityPrincipals
CN=Infrastructure
CN=LostAndFound
CN=Managed Service Accounts
CN=NTDS Quotas
CN=Program Data
CN=System
CN=TPM Devices
CN=Users
```

Figure 3: Connection successful, we output the list of children in our root

Managing Users and Groups

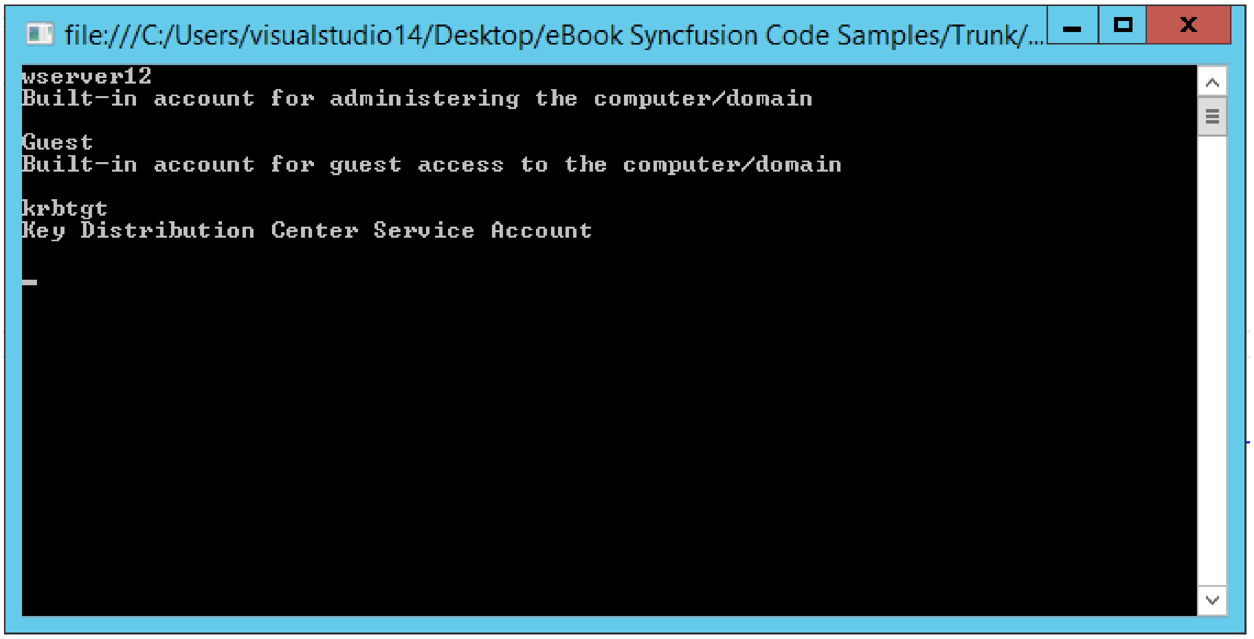
Listing User and Groups

The first and simplest management task that we are going to describe is when we need to create (maybe with some sort of filtering or ordering) a list of objects. These objects might be computers, users, or even printers. It is very common for companies to add these kinds of devices/entities inside the AD tree, because it simplifies their network allocation.

```
private static void ListObjects(Boolean users = true)
{
    DirectoryEntry userConn = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=Users,DC=alphadata, DC=adds", "user", "pass");

    // We will use "crawler" to search inside our AD with a given filter,
    // we also specify the properties we are interested in.
    // We use a flag to indicate if we want to search for users or groups.
    DirectorySearcher crawler = new DirectorySearcher(userConn);
    crawler.Filter = users ? "(objectCategory=user)" :
"(objectCategory=group)";
    crawler.PropertiesToLoad.AddRange(new string[] { "cn", "description"
});
    SearchResultCollection result = crawler.FindAll();

    // We now print the username and the account description for the object
we have found.
    foreach (SearchResult item in result)
    {
        Console.WriteLine(item.GetDirectoryEntry().Properties["cn"].Value);

Console.WriteLine(item.GetDirectoryEntry().Properties["description"].Value)
;
        Console.WriteLine();
    }
}
```

The previous code is very easy. First we create a DirectoryEntry object that points directly to our Users OU, then we instantiate a DirectorySearcher object specifying that we only want to return objects of type User (if the variable users is set to true). Then we call the method FindAll() and, thanks to the collection returned by it, with a for cycle we can iterate through the results. The following is a screenshot of the program output; in our case, we print on screen the user display name and description.

Note: In order to show the CN and description, you must add those properties to the .PropertiesToLoadCollection as shown.

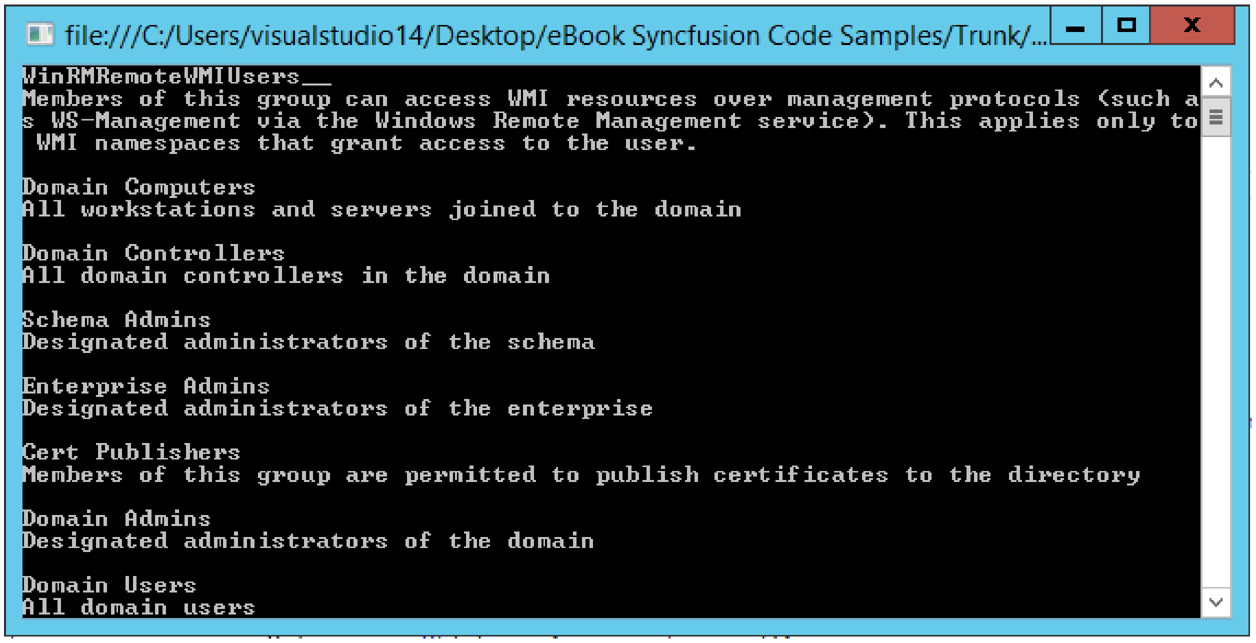


file:///C:/Users/visualstudio14/Desktop/eBook Syncfusion Code Samples/Trunk/...

```
wserver12
Built-in account for administering the computer/domain

Guest
Built-in account for guest access to the computer/domain

krbtgt
Key Distribution Center Service Account

_
```

Figure 4: The list of users in our AD

As we can see by the Boolean parameter **user**, we can easily change the type of query in order to get the list of groups contained inside our Users organizational unit.



file:///C:/Users/visualstudio14/Desktop/eBook Syncfusion Code Samples/Trunk/...

```
WinRMRemoteWMIUsers__
Members of this group can access WMI resources over management protocols (such a
s WS-Management via the Windows Remote Management service). This applies only to
 WMI namespaces that grant access to the user.

Domain Computers
All workstations and servers joined to the domain

Domain Controllers
All domain controllers in the domain

Schema Admins
Designated administrators of the schema

Enterprise Admins
Designated administrators of the enterprise

Cert Publishers
Members of this group are permitted to publish certificates to the directory

Domain Admins
Designated administrators of the domain

Domain Users
All domain users
```

Figure 5: The list of groups in our AD

Editing Object Properties

Of course, only listing entities is a useless task, and above all, it is not the reason that drives us in the need of a directory-enabled application. A more real and useful scenario is about editing the properties of an object. We will consider the case where we want to update the account description of a user.

```
private static void EditProperty()
{
    DirectoryEntry userConn = new
DirectoryEntry("LDAP://"+IP_PORT+"/CN=Users,DC=alphadata, DC=adds", "user",
"pwd");
    DirectoryEntry user = userConn.Children.Find("cn=wserver12");
    user.Properties["description"].Value = "Lorem ipsum dolor sit amet.
Admin account";
    user.CommitChanges();
}
```

The snippet of code above is almost identical to the one we talked about earlier, when we only needed to list AD entities, however this time we managed to get the reference of a single AD entity, a User more precisely. Once we have that, editing its properties is simple, we just need to use the correct index (in this case the index is represented by a literal string) in the Property collection and set the value that we want.

Tip: Remember to commit your changes after applying any consistent and valid modification.

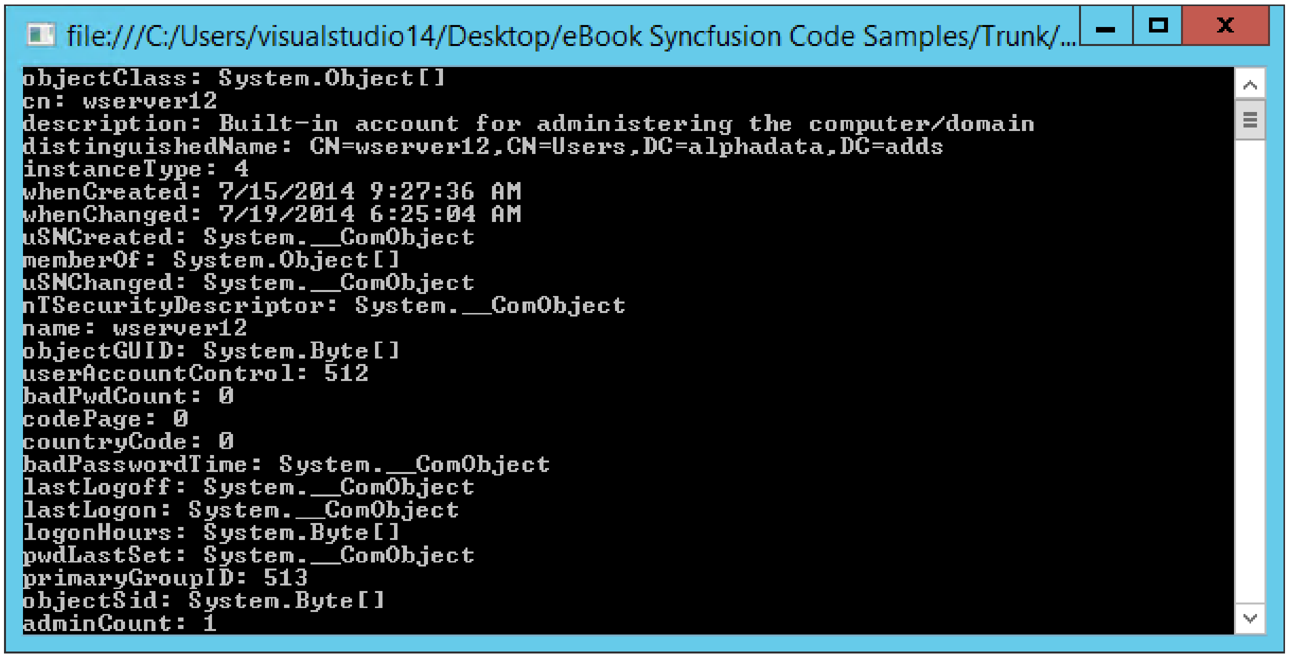
You may think the hard part is “how would I know that I am using a valid property indexer?” The answer is that you need to know them. A quite exhaustive list is indicated as follows; however, there are dozens more of them, but these are the more commonly used:

- sAMAccountName—The user ID of the account created.
- name—Same value as the sAMAccountName property.
- givenName—First name of the user.
- sn—Surname (last name) of the User.
- displayName—Typically the same as the name, and sAMAccountName.
- userPrincipalName—The same as name, sAMAccountName.
- co—Country.
- mail—Single email address value.

- telephoneNumber—User's phone number.
- description—a description of the account.
- userAccountControl—The enumerated property used to manage the User's account.
- wWWHomePage—User's default homepage (optional).
- parent—Parent owner object.
- cn—Canonical, or Common Name, usually the combination of givenName + sn (NOTE: If you have migrated from another system, like NT4 to AD 2000, this value is the same as the name, sAMAccountName, and displayName, by default, and it cannot be changed).

However, if you want to get the list of all available properties of an object, you can use this snippet of code:

```
private static void ListAllProps()
{
    DirectoryEntry userConn = new
DirectoryEntry("LDAP://"+IP_PORT+"/CN=Users,DC=alphadata, DC=adds", "user",
"pwd");
    DirectoryEntry user = userConn.Children.Find("cn=wserver12");
    foreach (PropertyValueCollection item in user.Properties)
        Console.WriteLine(item.PropertyName + ": " + item.Value);
}
```


file:///C:/Users/visualstudio14/Desktop/eBook Syncfusion Code Samples/Trunk/...

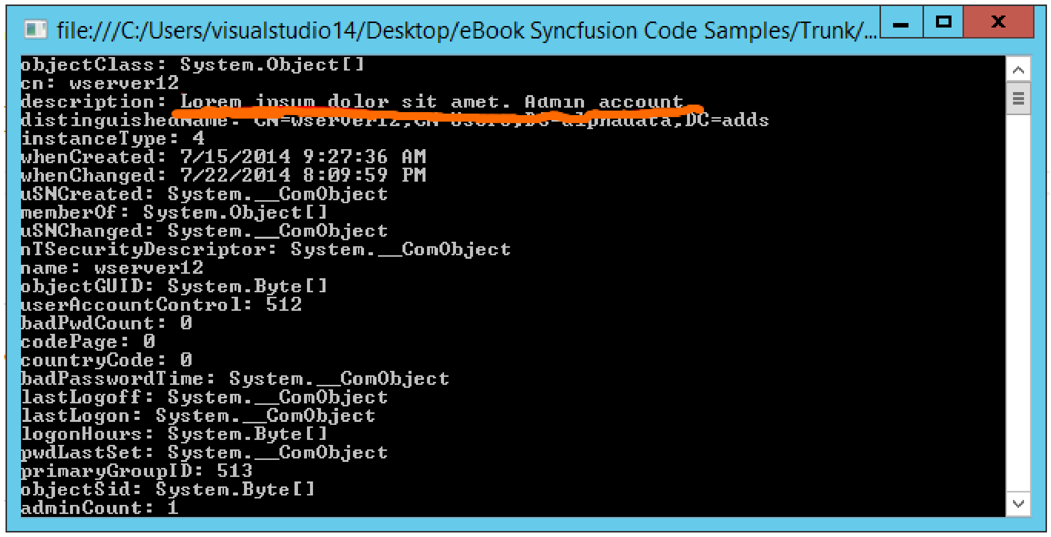
```
objectClass: System.Object[]
cn: wserver12
description: Built-in account for administering the computer/domain
distinguishedName: CN=wserver12,CN=Users,DC=alphadata,DC=adds
instanceType: 4
whenCreated: 7/15/2014 9:27:36 AM
whenChanged: 7/19/2014 6:25:04 AM
uSNCreated: System.__ComObject
memberOf: System.Object[]
uSNChanged: System.__ComObject
nTSecurityDescriptor: System.__ComObject
name: wserver12
objectGUID: System.Byte[]
userAccountControl: 512
badPwdCount: 0
codePage: 0
countryCode: 0
badPasswordTime: System.__ComObject
lastLogoff: System.__ComObject
lastLogon: System.__ComObject
logonHours: System.Byte[]
pwdLastSet: System.__ComObject
primaryGroupID: 513
objectSid: System.Byte[]
adminCount: 1
```

Figure 6: All the properties of a DirectoryEntry object

The picture above shows all the properties of the User objects that represents the account I'm currently using on the machine. The output is formatted as follows:

Tip: PropertyName : PropertyValue

By using the code indicated previously, we can now edit the description of the account. Here's a picture that proves that our operation completed successfully.


file:///C:/Users/visualstudio14/Desktop/eBook Syncfusion Code Samples/Trunk/...

```
objectClass: System.Object[]
cn: wserver12
description: Lorem ipsum dolor sit amet. Admin account
distinguishedName: [redacted],DC=adds
instanceType: 4
whenCreated: 7/15/2014 9:27:36 AM
whenChanged: 7/22/2014 8:09:59 PM
uSNCreated: System.__ComObject
memberOf: System.Object[]
uSNChanged: System.__ComObject
nTSecurityDescriptor: System.__ComObject
name: wserver12
objectGUID: System.Byte[]
userAccountControl: 512
badPwdCount: 0
codePage: 0
countryCode: 0
badPasswordTime: System.__ComObject
lastLogoff: System.__ComObject
lastLogon: System.__ComObject
logonHours: System.Byte[]
pwdLastSet: System.__ComObject
primaryGroupID: 513
objectSid: System.Byte[]
adminCount: 1
```

Figure 7: We have successfully edited a property

Check the Existence of an Object

Checking the existence of an object is always a useful task, because it avoids problems of possible collision due to duplicate objects.

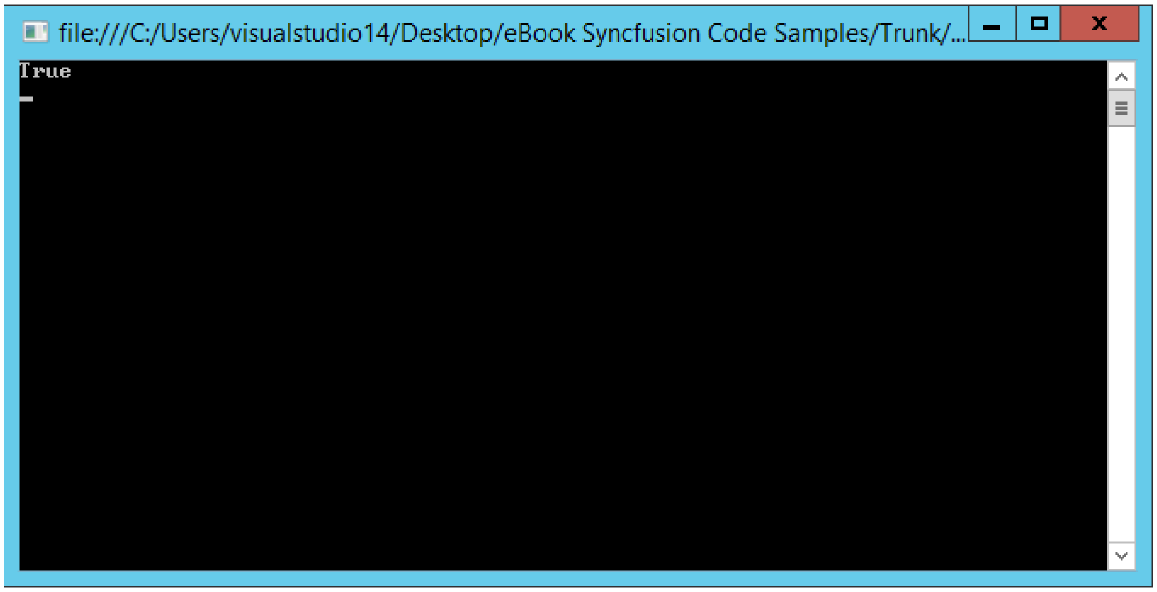
```
private static Boolean CheckIfUserExist(String cnName)
{
    DirectoryEntry userConn = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=Users,DC=alphadata, DC=adds", "user", "pwd");
    return userConn.Children.Find(cnName) != null;
}
```

The snippet above establishes a connection with our server pointing directly to the User's OU. Then with the Find() method we use the parameter passed as input to determine whether or not the user exists. You invoke the method in the following way:

```
static void Main(string[] args)
```

```
{
    Console.WriteLine(CheckIfUserExist("cn=wserver12"));
    Console.ReadLine();
}
```

We get this output (where true of course means that the object, in this case a user with that name, exists):



file:///C:/Users/visualstudio14/Desktop/eBook Syncfusion Code Samples/Trunk/...

```
True
_
```

Figure 8: The user exists

Creating Objects

When operating in a real environment, it is possible that the forest has to be modified during the course of the company existence, by creating new objects or removing existing objects, to better fit the real status of the company AD. Of course, the System.DirectoryServices namespace covers this case, providing a set of APIs that allows developers to pragmatically create objects inside an AD. Creating a user is a quite easy task, which can be achieved with the following code:

```
private static void CreateUser(String userName)
{
    try
    {
        DirectoryEntry dirEntry = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=Users,DC=alphadata, DC=adds", "user", "pass");
        DirectoryEntry newUser = dirEntry.Children.Add("CN=" + userName,
"user");
        newUser.Properties["samAccountName"].Value = userName;
        newUser.CommitChanges();
        String pwd = "G14ncarlo";
        Console.WriteLine("User pwd: " + pwd);
        newUser.Invoke("SetPassword", new object[] { pwd });
        newUser.CommitChanges();
        dirEntry.Close();
        newUser.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.InnerException);
    }}
```

Users are stored as children in the root of our AD, so that when creating an object you add a new item to the User collection of type **DirectoryEntries** (note how we specify the parameter *User*). After adding it to the children collection, we can specify additional properties to better describe our object. Since we are talking about a user, the same property naming conventions are used when editing a user's objects.

Deleting Objects

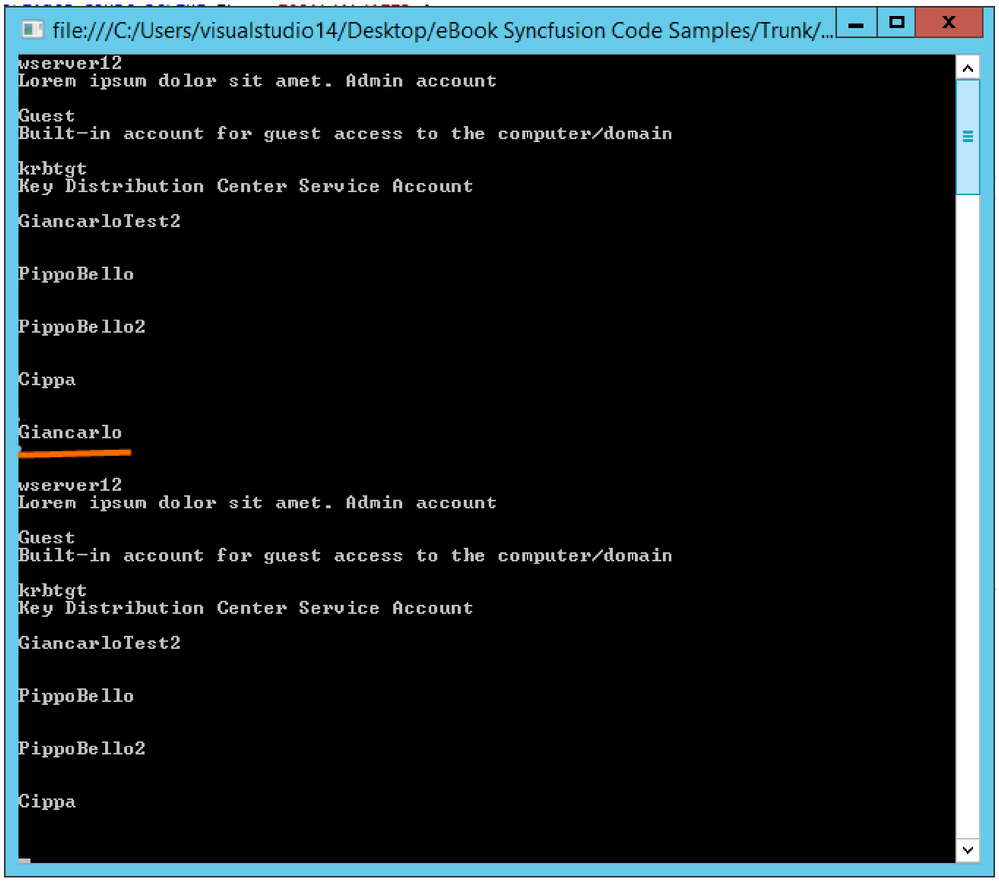
Deleting an object, like many other dangerous actions, is quite easy. In order to delete an AD object, you must first obtain a reference to the entity you want to delete, and then you have to call the Remove() method exposed by the children collection. In the following code snippet, we use the Find() method exposed by the Children collection to get an instance of the entity we want to delete.

```
private static void DeleteObject(String username)
{
    DirectoryEntry dirEntry = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=Users,DC=alphadata, DC=adds", "user", "pwd");
    DirectoryEntry user = dirEntry.Children.Find(username);
    if (user != null)
        dirEntry.Children.Remove(user);
}
```

In this example, we have invoked the method in this way:

```
static void Main(string[] args)
{
    ListObjects();
    DeleteObject("cn=Giancarlo");
    ListObjects();
    Console.ReadLine();
}
```

And this is what we get as output:



file:///C:/Users/visualstudio14/Desktop/eBook Syncfusion Code Samples/Trunk/...

```
wserver12
Lorem ipsum dolor sit amet. Admin account

Guest
Built-in account for guest access to the computer/domain

krbtgt
Key Distribution Center Service Account

GiancarloTest2


PippoBello


PippoBello2


Cippa


Giancarlo


wserver12
Lorem ipsum dolor sit amet. Admin account

Guest
Built-in account for guest access to the computer/domain

krbtgt
Key Distribution Center Service Account

GiancarloTest2


PippoBello


PippoBello2


Cippa
```

Figure 9: New user created

Managing User Password

Unlike most Active Directory and user management tasks, which we perform through simple manipulation of Active Directory objects and attributes via LDAP, managing passwords is a bit more complex. Password changes require very special semantics that are enforced by the server, and developers need to understand these semantics for password management applications to be successful. In order to try to facilitate the password management process, ADSI exposes two methods on the **IADsUser** interface: **SetPassword** and **ChangePassword**.

SetPassword is used to perform an administrative reset of a user's password and is typically performed by an administrator. Knowledge of the previous password is not required. ChangePassword is used simply to change the password from one value to another and is typically performed only by the user represented by the directory object. It does require knowledge of the previous password, and thus it takes the old and new passwords as arguments. Since the DirectoryEntry object does not directly expose the **IADsUser** ADSI interface, this is one case where we must use the DirectoryEntry.Invoke method to call these ADSI methods via late-bound reflection.

There are some specifics to understand when dealing with user passwords and boundaries around passwords, such as forcing a user to change their password on the next logon, denying the user the right to change their own passwords, setting passwords to never expire and when to expire; these tasks can be accomplished using **UserAccountControl.** Below is a table with all the available flags.

Constant Name	HEX Value
SCRIPT	0x0001
ACCOUNTDISABLE	0x0002
HOMEDIR_REQUIRED	0x0008
LOCKOUT	0x0010
PASSWD_NOTREQD	0x0020
PASSWD_CANT_CHANGE	0x0040
ENCRYPTED_TEXT_PWD_ALLOWED	0x0080
TEMP_DUPLICATE_ACCOUNT	0x0100
NORMAL_ACCOUNT	0x0200
INTERDOMAIN_TRUST_ACCOUNT	0x0800
WORKSTATION_TRUST_ACCOUNT	0x1000
SERVER_TRUST_ACCOUNT	0x2000
DONT_EXPIRE_PASSWORD	0x10000
MNS_LOGON_ACCOUNT	0x20000
SMARTCARD_REQUIRED	0x40000
TRUSTED_FOR_DELEGATION	0x80000
NOT_DELEGATED	0x100000
USE_DES_KEY_ONLY	0x200000
DONT_REQ_PREAUTH	0x400000
PASSWORD_EXPIRED	0x800000
TRUSTED_TO_AUTH_FOR_DELEGATION	0x1000000

We have already seen how to set a user password and how to create a User in the previous paragraph. In fact, if you look at line 11 of the code snippet, those lines of code show exactly how to set a user password.

However, we have not talked about the case of resetting the user password; here is the code showing how we do that.

```
private static void ResetPassword(string password)
{
    DirectoryEntry uEntry = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=TestUser, CN=Users, DC=alphadata, DC=adds", "user", "password");
    uEntry.Invoke("SetPassword", new object[] { password });
    uEntry.Properties["LockOutTime"].Value = 0; //unlock account
    uEntry.CommitChanges();
    uEntry.Close();
}
```

At first, we get the DirectoryEntry object of the user we are interested in. In a real scenario, the CN name of the user might be passed as a parameter and checked on the existence of the users. After obtaining the DirectoryEntry object, invoke the SetPassword method specified as parameter a new password that we receive as input. Since we are resetting the user password, the user might have entered it several times causing AD to block his account. In order to unblock it and give the user the possibility to login again, we unlock his account by setting the **LockOutTime** property to zero. Just to be sure, we commit our changes and we are done.

Adding User to a Group

```
private static void AddToGroup(DirectoryEntry user, string
groupConnectionString)
{
    try
    {
        DirectoryEntry dirEntry = new DirectoryEntry("LDAP://" +
groupConnectionString);
        dirEntry.Properties["member"].Add(user);
        dirEntry.CommitChanges();
        dirEntry.Close();
    }
    catch (System.DirectoryServices.DirectoryServicesCOMException ex)
    {
        throw ex;
    }
}
```

Remove User from Group

```
private void RemoveUserFromGroup(DirectoryEntry userDn, string
groupConnectionString)
```

```
{
    try
    {
        DirectoryEntry dirEntry = new DirectoryEntry("LDAP://" +
groupConnectionString);
        dirEntry.Properties["member"].Remove(userDn);
        dirEntry.CommitChanges();
        dirEntry.Close();
    }
    catch (System.DirectoryServices.DirectoryServicesCOMException ex)
    {
        throw ex;
    }
}
```

In this case, we omit the check of existence of the user inside the group, but that should always be performed.

Enumerate Directory Entry Settings

One of the nice things about the 2.0 classes is the ability to get and set a configuration object for your directoryEntry objects.

```
private static void DirectoryEntryConfigurationSettings()
{
    DirectoryEntry entry = new DirectoryEntry("LDAP://" + IP_PORT +
"/DC=alphadata, DC=adds", "user", "pwd");
    DirectoryEntryConfiguration entryConfiguration = entry.Options;

    Console.WriteLine("Server: " +
                entryConfiguration.GetCurrentServerName());
    Console.WriteLine("Page Size: " +
                entryConfiguration.PageSize.ToString());
    Console.WriteLine("Password Encoding: " +
                entryConfiguration.PasswordEncoding.ToString());
    Console.WriteLine("Password Port: " +
                entryConfiguration.PasswordPort.ToString());
    Console.WriteLine("Referral: " +
                entryConfiguration.Referral.ToString());
    Console.WriteLine("Security Masks: " +
                entryConfiguration.SecurityMasks.ToString());
    Console.WriteLine("Is Mutually Authenticated: " +
                entryConfiguration.IsMutuallyAuthenticated().ToString());
}
```

Error Handling with the Invoke Method in .NET

Error handling for these two methods is certainly relevant; it is a larger .NET topic that affects any code that uses reflection via the Invoke method. When the Invoke method is used to call any ADSI interface method, any exception thrown by the target method, including a COM error triggered via COM interop, will be wrapped in a **System.Reflection.TargetInvocationException** exception, with the actual exception in the **InnerException** property. Therefore, we will need to use a pattern like this with the Invoke method.

```
private static void DangerousCode(String userName)
{
    try
    {
        DirectoryEntry dirEntry = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=Users,DC=alphadata, DC=adds", "wserver12", "@Wserver12@");
        DirectoryEntry newUser = dirEntry.Children.Add("CN=" + userName,
"user");
        newUser.Properties["samAccountName"].Value = userName;
        newUser.CommitChanges();
        dirEntry.Close();
        newUser.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.InnerException);
        throw ex;
    }}
```

Note: The example above is a slightly modified version of the one used to create a User object.

Obviously, we may wish to do something different from simply printing and re-throwing the InnerException. The point here is that InnerException contains the information we are interested in. The other issue here is that the exceptions coming back from ADSI vary from the vaguely helpful to the truly bewildering. We will not attempt to list all of them, but here are a few hints.

- System.UnauthorizedAccessException always indicates a permissions problem.
- A System.Runtime.InteropServices.COMException exception with **ErrorCode 0x80072035** "Unwilling to perform" generally means that an LDAP password modification failed due to a password policy issue.
- A System.Runtime.InteropServices.COMException exception from one of the Net*APIs is usually pretty specific about what the problem was.

Handling Account Expiration

In many corporate networks, even in the small ones, account expiration is a common policy. Of course account expirations is something that resides inside the AD, hence it can be managed by the System.DirectoryServices namespace. In order for us to do that, the first thing to do is to characterize a particular user, and as we discussed before, this can be easily done with a few lines of code. However, what about the expiration?

Setting User Account Expiration

Setting the user account expiration is done by using the Invoke() method, hence this time we will be using reflection. However, this time the parameters passed to the Invoke() method are not as straightforward as they were in the previous examples. Here is a method that shows how to set the account expiration for a user.

```
private static void UserExpiration()
{
    DirectoryEntry usr = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=TestUser, CN=Users, DC=alphadata, DC=adds", "user", "pwd");

    // Get the native object.
    Type type = usr.NativeObject.GetType();
    Object adsNative = usr.NativeObject;

    // Use the Type.InvokeMember method to invoke the
    // AccountExpirationDate property setter.
    type.InvokeMember(
        "AccountExpirationDate",
        BindingFlags.SetProperty, // using System.Reflection
        null,
        adsNative,
        new object[] { "12/29/20015" });

    usr.CommitChanges(); // Commit the changes.
}
```

This is a complex piece of code. It uses reflection and operates directly on the native object in order to invoke the native method that will eventually set the expiration date. Of course, this code is very error prone and an easier solution that does not use reflection is indicated below.

```
private static void UserExpiration()
{
    DirectoryEntry usr = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=TestUser, CN=Users, DC=alphadata, DC=adds", "user", "pwd");
    // Use the DirectoryEntry.InvokeSet method to invoke the
    // AccountExpirationDate property setter.
    usr.InvokeSet(
"AccountExpirationDate",
new object[] { new DateTime(2015, 12, 29) });

    // Commit the changes.
    usr.CommitChanges();
}
```

In the second snippet, the first parameter is the property we want to modify. After that, we have an object array that contains the **DateTime**, which express when the account is going to expire. As always, in order for the changes to be propagated to the AD tree, we must call the CommitChanges() method exposed by the native object (in this case a User) that we are dealing with.

Security Descriptor Management

Unfortunately, not all kinds of operations can be done by modifying an object's attributes; in fact, sometimes it's necessary to modify the security descriptor of an object in order to allow some sort of action. These operations are accomplished by adding an access control entry (ACE) to the discretionary access control list (DACL) of the desired objects.

Note: In version 1.0 and 1.1 of the .NET Framework, there is no native support for modifying the security descriptors of Active Directory objects. The security descriptor of an Active Directory object can be manipulated by invoking ADSI and using the IADsSecurityDescriptor and associated interfaces. In .NET Framework 2.0, this has changed since the namespace contains several classes to allow the manipulation of a security descriptor without having to invoke ADSI.

The primary class used with Active Directory object security descriptors is **ActiveDirectorySecurity**; it has methods and properties that allow the discretionary access control list and secondary access control list of an object to be read and modified. An access control entry on a discretionary access control list is known as an access rule. The primary class that represents an access rule is the **ActiveDirectoryAccessRule** class.

An access control entry on a system access control list is known as an audit rule, which is represented by the **ActiveDirectoryAuditRule** class. Setting an access rule can be a very repetitive task, especially if we decide to manually set each audit rule for every object we have. In this case, a best practice is to directly edit the DACL of the container object and let all the audit rules to be inherited. This also ensures that all further objects added to that container are going to inherit that rule. The **ActiveDirectoryAccessRule** class is used as a base class to define a series of sub classes that describe in a more precise way a type of access rule. The following is the tabbed list of those sub classes.

Class	Usage
CreateChildAccessRule	Represents an access rule that is used to allow or deny an AD object the right to create child objects.
DeleteChildAccessRule	Represents an access rule that is used to allow or deny an AD object the right to delete child objects.
DeleteTreeAccessRule	Represents an access rule that is used to allow or deny an AD object the right to delete all child objects.
ExtendedRightAccessRule	Represents an access rule that is used to allow or deny an AD object an extended right.
ListChildrenAccessRule	Represents an access rule that is used to allow or deny an AD object the right to list child objects.
PropertyAccessRule	Represents an access rule that is used to allow or deny access to an AD property.
PropertySetAccessRule	Represents an access rule that is used to allow or deny access to an AD property set.

The following code shows how to set a security descriptor for a user so that the telephone number property can be edited; this policy is applied to the User container, so it gets propagated to its children.

```
private static void SecurityDescriptor()
{
    try
    {
        DirectoryEntry container = new DirectoryEntry("LDAP://" + IP_PORT +
"/CN=Users, DC=alphadata, DC=adds", "user", "pwd");

        // Get the ActiveDirectorySecurity for the container.
        ActiveDirectorySecurity containerSecurity =
container.ObjectSecurity;

        // using System.Security.Principal
        // Create a SecurityIdentifier object for "self".
        SecurityIdentifier selfSid =
            new SecurityIdentifier(WellKnownSidType.SelfSid, null);

        // Get the schema for the currently logged on user.
        // using System.DirectoryServices.ActiveDirectory
        ActiveDirectorySchema schema =
ActiveDirectorySchema.GetCurrentSchema();

        // Get the telephoneNumber schema property object.
        ActiveDirectorySchemaProperty phoneProperty =
schema.FindProperty("telephoneNumber");

        // Get the user schema class object.
        ActiveDirectorySchemaClass userClass = schema.FindClass("user");

        // Create a property access rule to allow a user to write to their
own telephoneNumber property.
        PropertyAccessRule allowWritePhoneRule =
            new PropertyAccessRule(
                selfSid,
                AccessControlType.Allow, // using
System.Security.AccessControl
                PropertyAccess.Write,
                phoneProperty.SchemaGuid,
                ActiveDirectorySecurityInheritance.Descendents,
                userClass.SchemaGuid);

        // Add the access rule to the DACL.
        container.ObjectSecurity.AddAccessRule(allowWritePhoneRule);

        // Commit the changes.
        container.CommitChanges();
    }
    catch (ActiveDirectoryObjectNotFoundException)
    { // The schema class or property could not be found.
    }
}
```

Chapter 7 Implementing Single Sign-On (SSO)

Introduction

With the increasing support provided by IT to business processes, users and system admins are faced with an increasingly complicated task: managing and maintaining users' accounts in a coordinated manner, in order to maintain the integrity of security policy enforcement. On the other side, users typically have to sign-on to multiple systems, necessitating an equivalent number of sign-on dialogues, each of which may involve different usernames and authentication information.

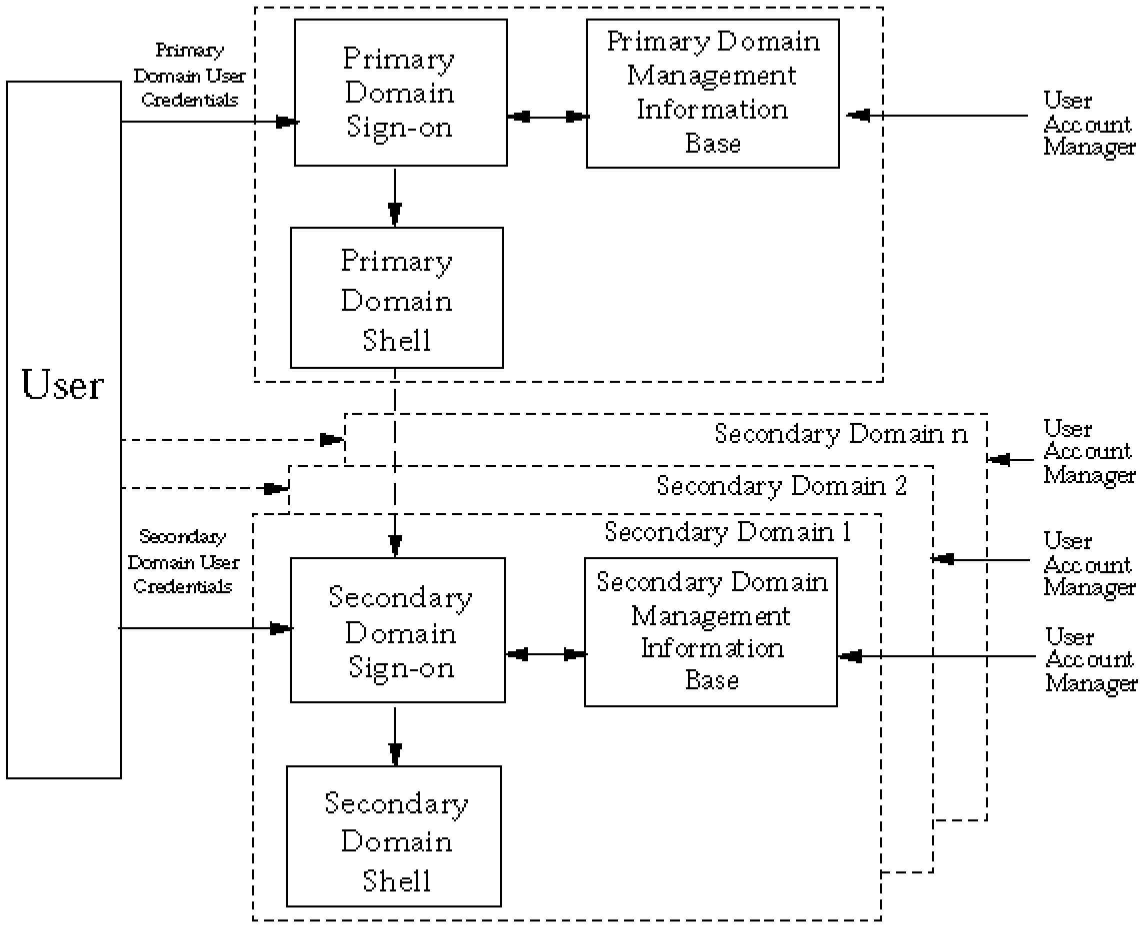


Figure 10: The classic schema of multiple independent user account managers

In general, distributed systems have always been assembled from components that act as independent security domains. These components act as independent domains in the sense that the end user has to identify and authenticate himself independently to each of the domains with which he wishes to interact.

This scenario is illustrated in Figure 10. The end user interacts initially with a Primary Domain to establish a session with that domain. This is done with the Primary Domain Sign-On in the above diagram and requires the end user to supply a set of user credentials applicable to the primary domain, for example, a username and password.

The primary domain session is typically represented by an operating system session shell executed on the end user's workstation within an environment. From this primary domain session shell, the user is able to invoke the services of the other domains, such as platforms or applications.

To invoke the services of a secondary domain, an end user is required to perform a Secondary Domain Sign-On. This requires the end user to supply a further set of user credentials applicable to that secondary domain. An end user has to conduct a separate sign-on dialogue with each secondary domain that the end user requires to use.

From the management perspective, the legacy approach requires independent management of each domain and the use of multiple user account management interfaces. Usability and security both require coordinating and, where possible, the integration of user sign-on functions and user-account management functions for the multitude of different domains in an enterprise. A service that provides such coordination and integration can provide real cost benefits to an enterprise through the following:

- **Reduction** in the time taken by users in sign-on operations to individual domains, including reducing the possibility of such sign-on operations failing.
- **Improved security** through the reduced need for a user to handle and remember multiple sets of authentication information.
- **Reduction** in the time taken, and improved response, by system administrators in adding and removing users in the system or modifying their access rights.
- **Improved security** through the enhanced ability of system administrators to maintain the integrity of user account configuration, including the ability to inhibit or remove an individual user's access to all system resources in a coordinated and consistent manner.

This type of service has been named **single sign-on**, after the end user's perception of the service. However, both the end user and management aspects of the service are equally important. In the single sign-on approach, the system is required to collect all the identification and user credential information necessary to support authentication as part of the primary sign-on. This means all information needed to authenticate each of the secondary domains that the user may potentially interact with must be collected here. Single sign-on services then use the information to support the end user's authentication to each of the secondary domains the user actually requests to interact with.

The information supplied by the end user as part of the Primary Domain Sign-On procedure may be used in support of secondary domain sign-on in several ways:

- **Directly**: the information supplied by the user is passed to a secondary domain as part of a secondary sign-on.
- **Indirectly**: the information supplied by the user is used to retrieve other user identification and user credential information stored within the single sign-on management information base. The retrieved information is then used as the basis for a secondary domain sign-on operation.
- **Immediately**: to establish a session with a secondary domain as part of the initial session establishment. This implies that application clients are automatically invoked and communications established at the time of the primary sign-on operation.
- **Temporarily:** stored or cached and used at the time a request for the secondary domain services is made by the end user.

From a management perspective, the single sign-on model provides a single account management interface through which all the component domains may be managed in a coordinated and synchronized manner. Significant security aspects of the single sign-on model are as follows.

The secondary domains have to trust the primary domain in:

- Correctly asserting the identity and authentication credentials of the end user.
- Protecting the authentication credentials used to verify the end user identity to the secondary domain from unauthorized use.

Authentication credentials must be protected against interception when transferred between the primary and secondary domains.

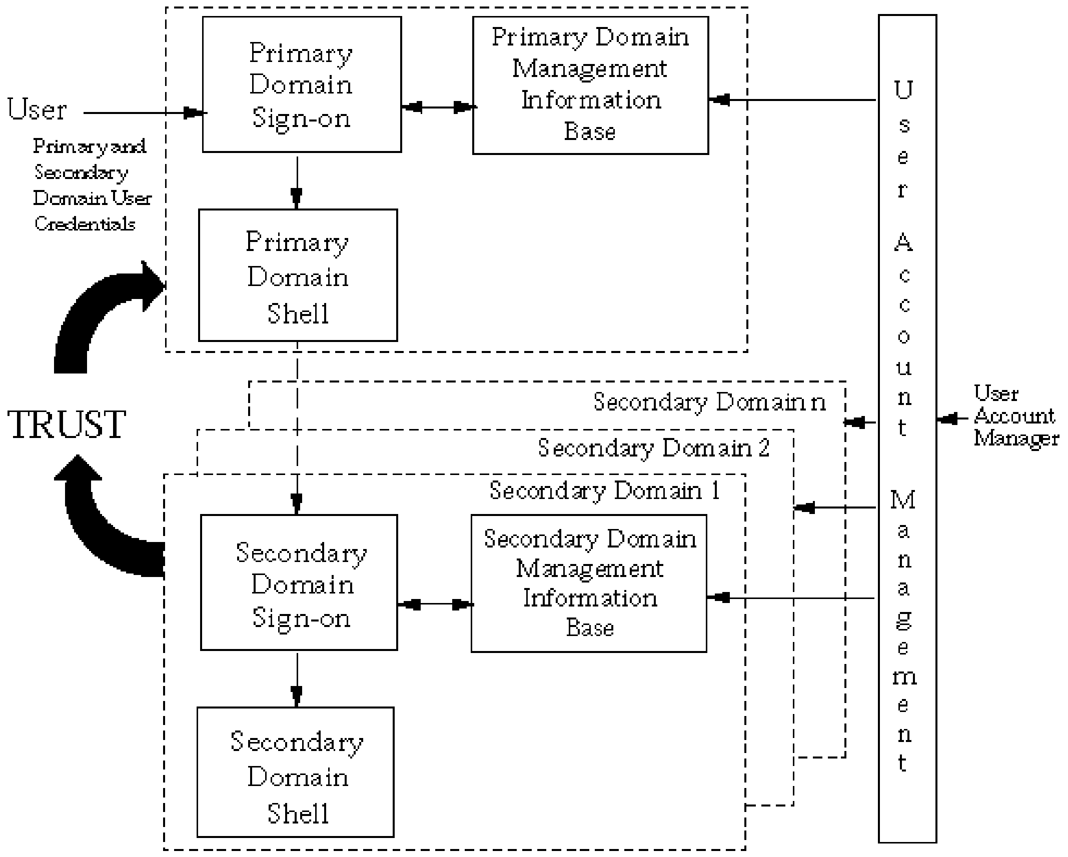


Figure 11: The modern scenario for leveraging SSO

A Simpler Definition of Single Sign-On

SSO is the ability to access different controls and multiple systems using the same credentials; it can be seen as a property of the system that allows the user to login without being prompted for credentials every time they try to access corporate resources or portals. This can be accomplished using the Lightweight Directory Access Protocol (LDAP) and stored LDAP databases on servers. A simple version of single sign-on can be achieved using cookies, but only if the sites are on the same domain. On the other hand, Single Sign-Off is the property whereby a single action of signing out terminates access to multiple software systems.

Wrapping Up the Benefits and Security Concerns of SSO

The benefits of SSO (along with web access management) apply in many areas, such as:

- **User experience**: The most apparent benefit is that users can move between services securely and uninterrupted without specifying their credentials each time. SSO effectively joins these individual services into portals and removes the service boundaries—switching from one application to the next appears seamless to the user.
- **Security**: The user's credentials are provided directly to the central SSO server, not the actual service that the user is trying to access; therefore, the credentials cannot be cached by the service. The central authentication point—the SSO service—limits the possibility of phishing.
- **Resource savings**: IT administrators can save time and resources by utilizing the central web access management service. Application and web developers receive a complete authentication and authorization framework that they can use to build secure, user customized services.

However, some security concerns may also come up. A single sign-on provides access to many resources once the user is initially authenticated. This increases the negative impact in the event the credentials are available to other persons and misused. Therefore, single sign-on requires an increased focus on the protection of the user credentials, and should ideally be combined with strong authentication methods like smart cards and one-time password tokens. Single sign-on also makes the authentication systems highly critical; a loss of their availability can result in denial of access to all systems unified under the SSO.

Distinguishing Different Types of SSO

Businesses have a couple of choices when it comes to implementing Sign Sign-On. They can choose a **"full sign-on"** system, where a user authenticates into a system once and then has access to all associated systems without having to enter their credentials again (unless there is an inactivity timeout).

Another option that poses fewer security risks is known as a "**reduced sign-on**" system. Users within a business will be able to access all of their systems with the same username and password. The difference between this and a "**full sign-on**" system is the user will have to enter their credentials for each system. The benefit to this is that it increases the level of security; however, the users will have to enter their credentials multiple times. In addition, systems have to be put in place to ensure the user's credentials are replicated across the various systems that they are authorized to access. If these systems break, users may have trouble accessing the systems they need, which may drive additional calls to the helpdesk.

An additional method of implementing single sign-on is called "**federated logins**". From a user perspective, Federated Logins are similar to that of a full sign-on system. However, on the backend, there are some big differences. With traditional single sign-on systems, each of the different systems involved has their own authentication systems.

Implementing SSO with Microsoft Azure AD

The goal of this section of the book is to guide you through the steps needed to integrate your on premise Active Directory (AlphaData) into a web application, thus letting all the AlphaData employees log into the company portal with the username and password they use to log onto their workstations. This task is easily accomplished with Windows Azure Active Directory. Azure AD enables you to make enterprise line-of-business (LOB) apps available over the Internet.

Introducing Azure Active Directory

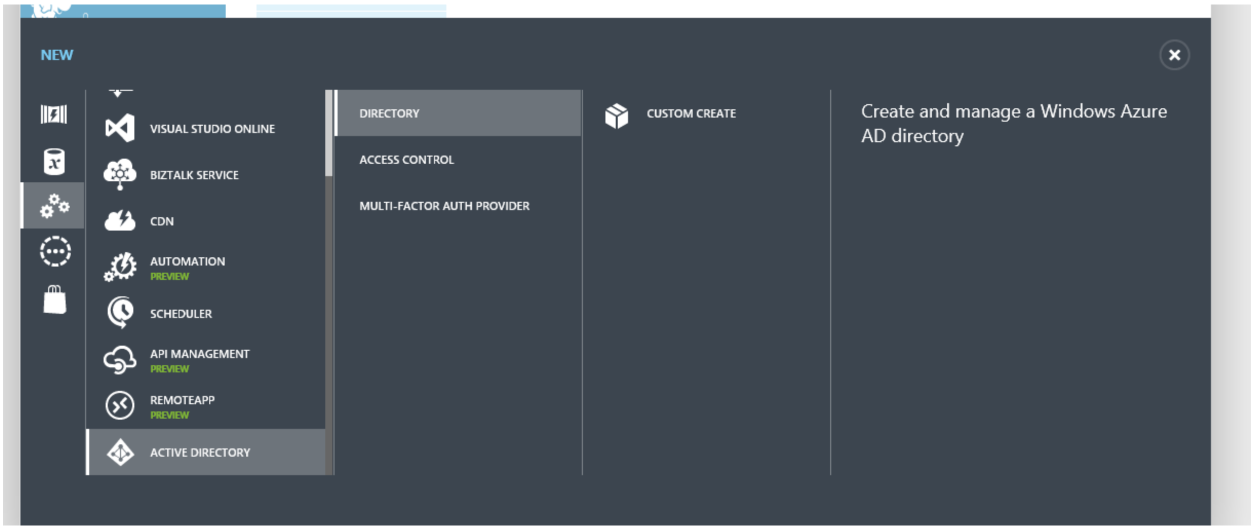
Azure AD provides active directory features on the cloud, with the possibility to integrate or work alongside an on premise AD environment. Azure Active Directory (Azure AD) is a service that is made available through Azure for cloud-based identity management. When you use Azure AD, it is Microsoft's responsibility to keep Active Directory running in the cloud with high scale, high availability, and integrated disaster recovery, while fully respecting your requirements for the privacy and security of your organization's information. The key features of Azure AD are:

- It integrates with on-premises Active Directory.
- It enables single sign-on with your apps.
- It supports directory sync
- It supports open standards such as SAML, WS-Fed, and OAuth 2.0.
- It supports Enterprise Graph REST API.

Azure AD can be entirely independent from your on-premises Active Directory; you can put anyone you want in it and authenticate them in Internet apps.

Creating an Azure AD

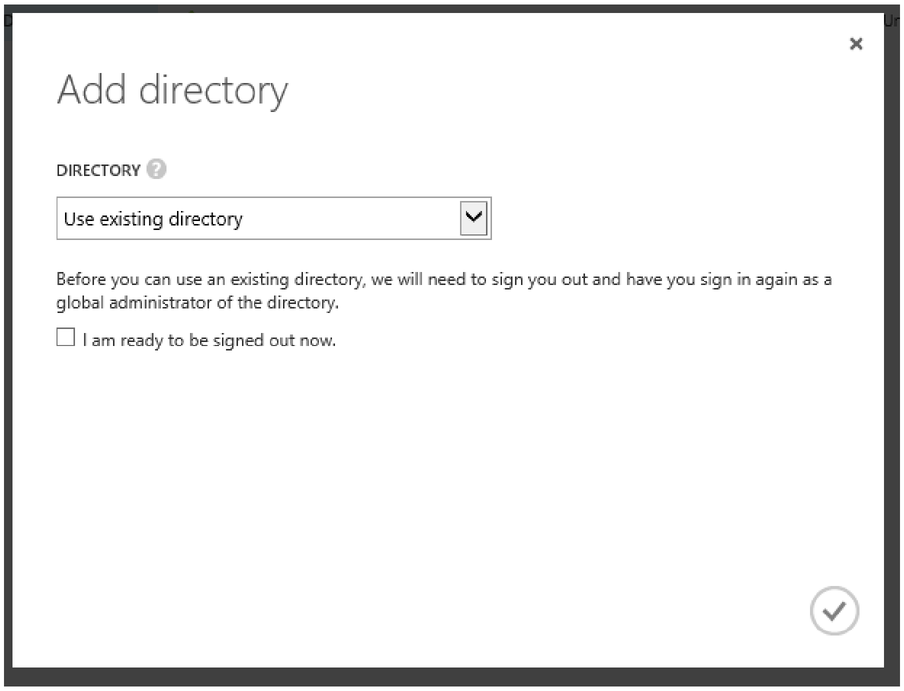
Creating an Azure AD is a far easier task than creating a normal AD on Windows Server. We can create an Azure AD through the wizard by selecting from the menu **New** > **App Services** > **Active Directory** > **Directory** > **Custom Create**.



Create and manage a Windows Azure AD directory

Figure 12: Wizard for creating an AD inside Azure

In order to fully implement and take advantage of the SSO feature provided by Azure AD, we choose to create an Azure AD associated with an existing directory. This directory has an associated Office 365 subscription, hence in a real world scenario it will contain all the users of our company, along with their credentials for accessing online resources.



Add directory

DIRECTORY

Use existing directory

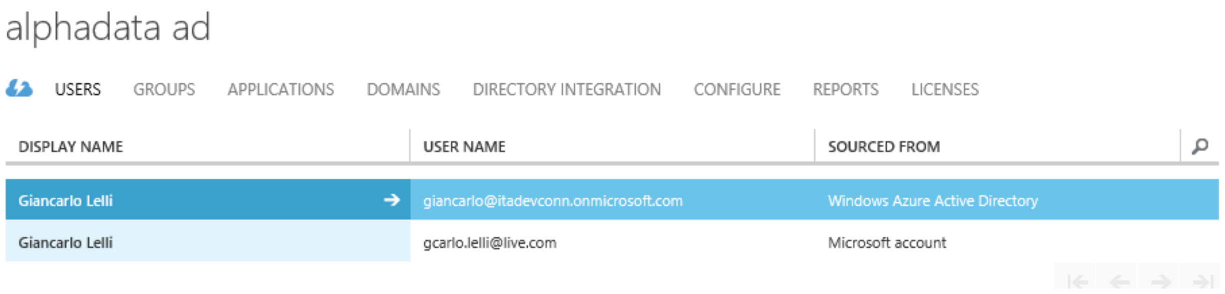
Before you can use an existing directory, we will need to sign you out and have you sign in again as a global administrator of the directory.

☐ I am ready to be signed out now.

Figure 13: A dialog where we can choose how we want to create our Azure AD

Once our Azure AD has been created, you can navigate inside it. In the top bar you should see all the possible configuration options available, such as Users, Groups, and Custom Domains.

As you can see in the following picture, Azure automatically added my Microsoft account and Office 365 administrator account to the user list, both accounts have Global Administrator privileges. However, in our case, we are mostly interested in the **Directory Integration** tab.



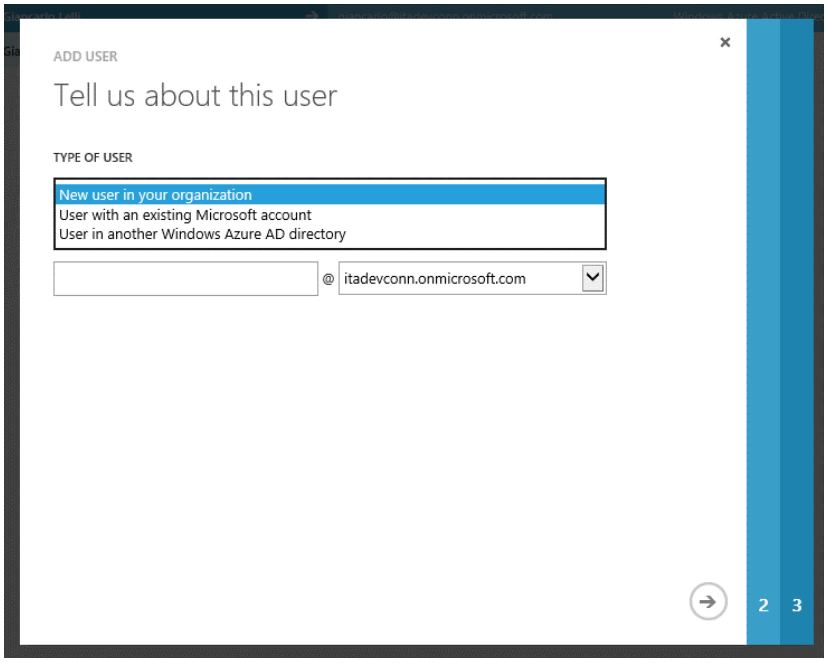
DISPLAY NAME	USER NAME	SOURCED FROM
Giancarlo Lelli	giancarlo@itadevconn.onmicrosoft.com	Windows Azure Active Directory
Giancarlo Lelli	gcarlo.lelli@live.com	Microsoft account

Figure 14: The users in our Azure AD

Note: For testing purposes, the domain of our Office 365 subscription does not match the name of our company; in a real scenario, they'*re supposed to match to avoid confusion. However, this difference enhances the problem of having multiple sets of credentials for different company resources. In this case, our scenario can be described as follows: one on-premise Active Directory used as the Account Manager for local resources, plus one Azure AD connected to an Office 365 subscription used for online resources. Our goal is to sync or merge these two environments.

Creating a New Test User in the Azure AD

Before we move along with our integration, let's see how we can create a new user inside our Azure AD through the portal. This user will be used later in the configuration process. As we can see in Figure 15, we have three different options for creating a user. We can create one with an existing Microsoft account, add a user that is already part of another AD namespace, or we can create one from scratch. We shall now proceed by creating a new user from scratch and granting him Global Administrator privileges.



ADD USER

Tell us about this user

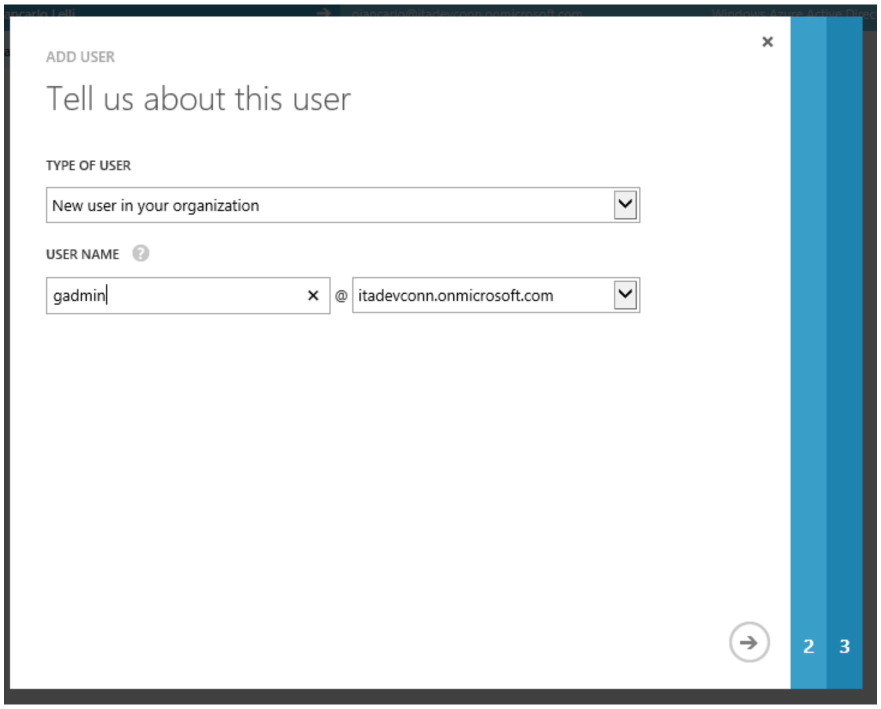
TYPE OF USER

New user in your organization
User with an existing Microsoft account
User in another Windows Azure AD directory

@ itadevconn.onmicrosoft.com

Figure 15: The source or type of user

Creating a user is a straightforward operation—all we have to do is follow the wizard as indicated in the following pictures. First we click **Add User** at the bottom of the screen; this will let a popup appear asking us to fill out a form (in multiple steps) specifying all the basic information of our new user. Figures 16 and 17 show how I've filled the form in order to create a Global Administrator user.



ADD USER

Tell us about this user

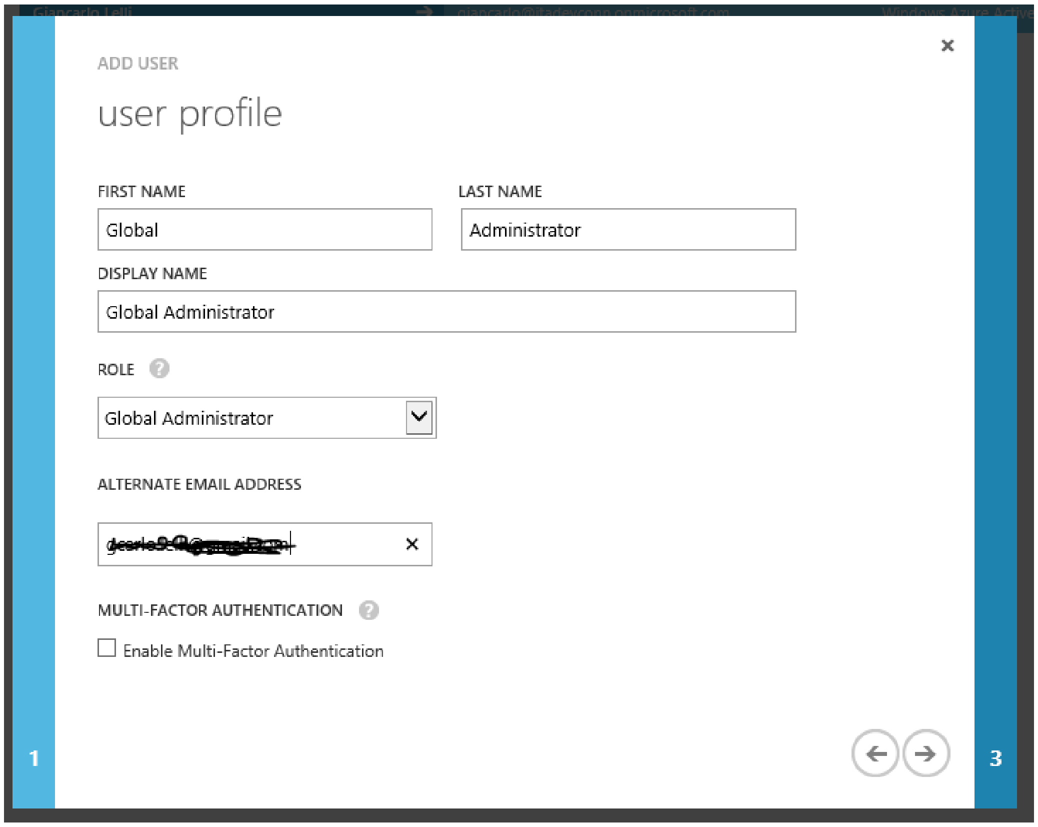
TYPE OF USER

New user in your organization

USER NAME

gadmin @ itadevconn.onmicrosoft.com

Figure 16: Giving our user an organizational email address


ADD USER

user profile

FIRST NAME: Global

LAST NAME: Administrator

DISPLAY NAME: Global Administrator

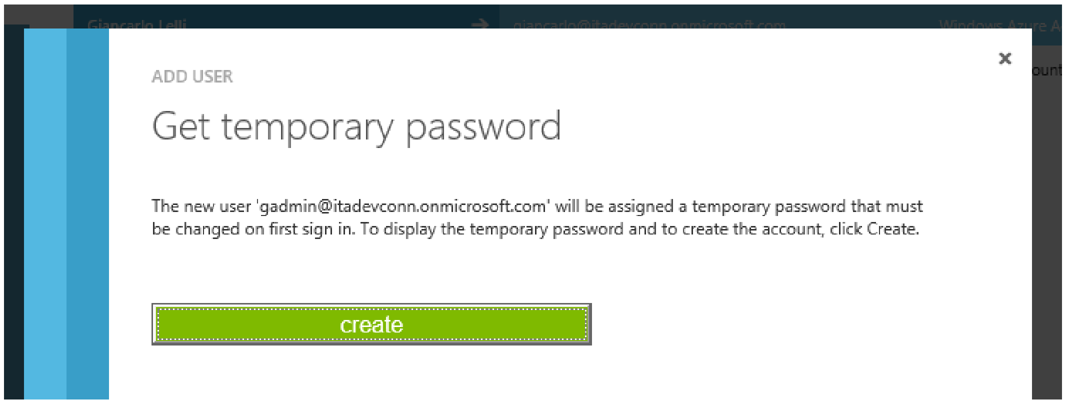
ROLE: Global Administrator

ALTERNATE EMAIL ADDRESS: [redacted]

MULTI-FACTOR AUTHENTICATION

☐ Enable Multi-Factor Authentication

Figure 17: Some generic information about the user


ADD USER

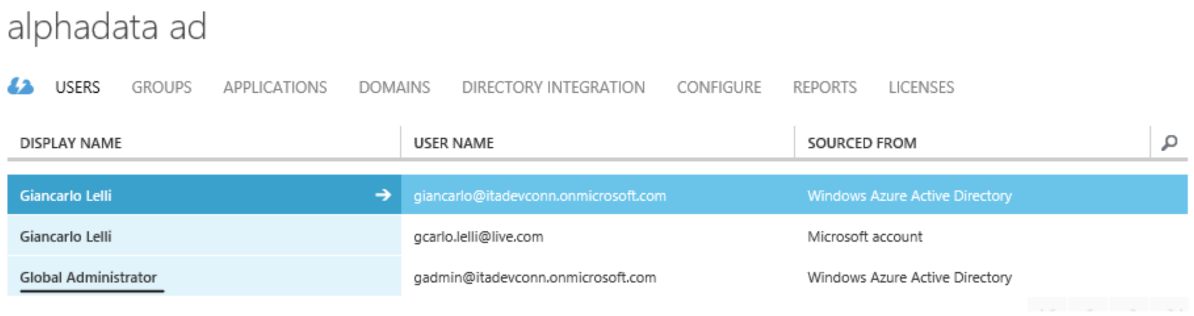
Get temporary password

The new user 'gadmin@itadevconn.onmicrosoft.com' will be assigned a temporary password that must be changed on first sign in. To display the temporary password and to create the account, click Create.

Figure 18: Azure creates the user with a temporary password

Now that our user has been created, we can proceed with the next step, integrating our on premise AD with this new Azure Active Directory.

Tip: Make sure to sign in at least one time with this new user. This will give a working and final password to our user.



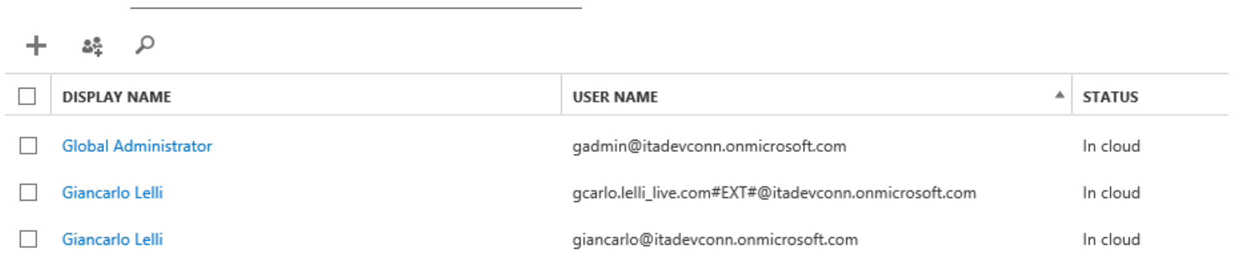
alphadata ad

USERS GROUPS APPLICATIONS DOMAINS DIRECTORY INTEGRATION CONFIGURE REPORTS LICENSES

DISPLAY NAME	USER NAME	SOURCED FROM
Giancarlo Lelli	giancarlo@itadevconn.onmicrosoft.com	Windows Azure Active Directory
Giancarlo Lelli	gcarlo.lelli@live.com	Microsoft account
Global Administrator	gadmin@itadevconn.onmicrosoft.com	Windows Azure Active Directory

Figure 19: User created

To verify the success of this operation, we can simply login from our Office 365 admin portal to see the entire set of available users.



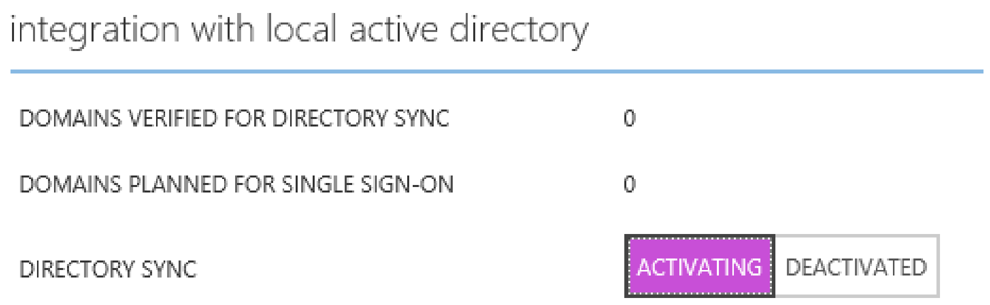
	DISPLAY NAME	USER NAME	STATUS
☐	Global Administrator	gadmin@itadevconn.onmicrosoft.com	In cloud
☐	Giancarlo Lelli	gcarlo.lelli_live.com#EXT#@itadevconn.onmicrosoft.com	In cloud
☐	Giancarlo Lelli	giancarlo@itadevconn.onmicrosoft.com	In cloud

Figure 20: Our new user is also available inside the Office 365 environment

Integrating Our On-Premise Active Directory User in Azure AD

As we said before, the goal of our Azure AD is to create some sort of bridge between our on-premise AD, giving the opportunity to our employees or users to leverage the SSO functionalities and benefits in our company's web application. Later in this book, we will also see how this integration can be useful to secure a Web API endpoint.

Let's now move into the **Directory Integration** tab inside our Azure AD management portal. The first thing to do is change the directory integration switch from **Deactivated** to **Activating** and then select **Save**.



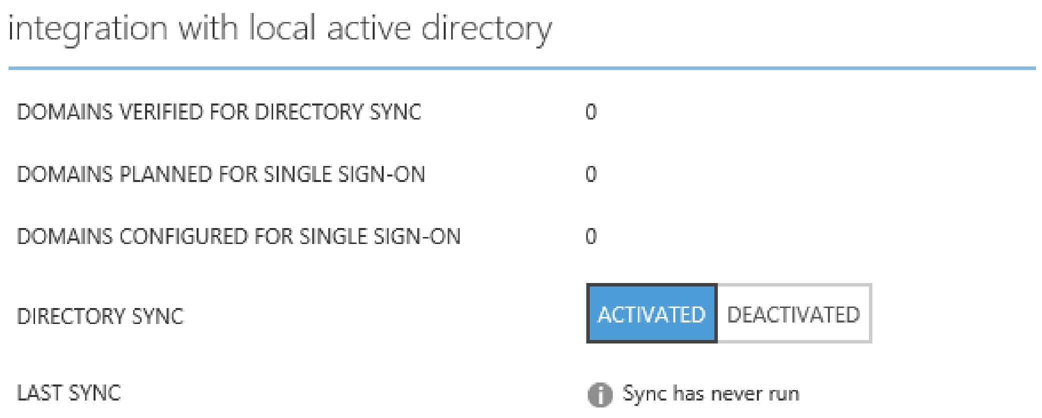
integration with local active directory

DOMAINS VERIFIED FOR DIRECTORY SYNC	0
DOMAINS PLANNED FOR SINGLE SIGN-ON	0
DIRECTORY SYNC	ACTIVATING DEACTIVATED

Figure 21: The switch for activating the Directory Sync feature

Note: Make sure to read and remember in the warning prompt that Azure displays when Save is clicked.

Once the operation completes, we should see a screen similar to Figure 22. Now we are ready to download the **Directory Sync Tool** and run it on our on premise AD server.



integration with local active directory

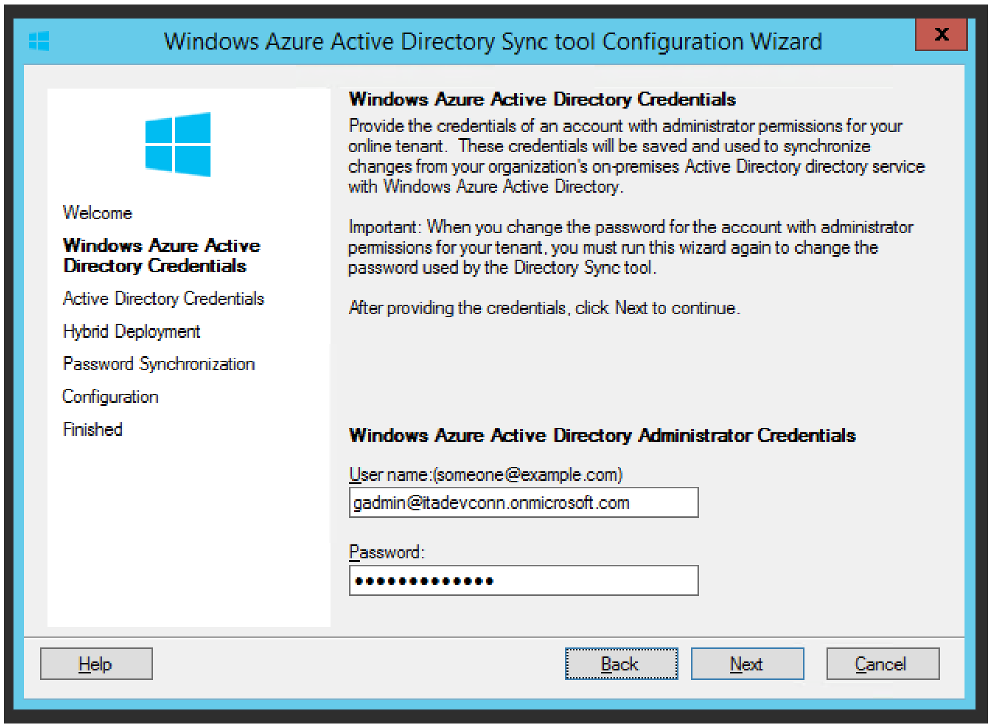
DOMAINS VERIFIED FOR DIRECTORY SYNC	0
DOMAINS PLANNED FOR SINGLE SIGN-ON	0
DOMAINS CONFIGURED FOR SINGLE SIGN-ON	0
DIRECTORY SYNC	ACTIVATED DEACTIVATED
LAST SYNC	Sync has never run

Figure 22: The Directory Sync feature is active

At the time of writing, the Directory Sync tool is available here and requires .NET Framework 3.5 and 4.5 to be installed on your server. In case you are using Windows Server 2012 or newer, you need to manually install version 3.5 since it is not installed by default. Once you complete the installation, it will take about 10 minutes. We are now ready to proceed with the integration.

Tip: Restart your PC before proceeding with this operation; I have experienced many configuration fails without restarting.

The first step will require us to enter the credentials of a global administrator account inside our online tenant.


Windows Azure Active Directory Sync tool Configuration Wizard

Welcome
Windows Azure Active Directory Credentials
Active Directory Credentials
Hybrid Deployment
Password Synchronization
Configuration
Finished

Windows Azure Active Directory Credentials

Provide the credentials of an account with administrator permissions for your online tenant. These credentials will be saved and used to synchronize changes from your organization's on-premises Active Directory directory service with Windows Azure Active Directory.

Important: When you change the password for the account with administrator permissions for your tenant, you must run this wizard again to change the password used by the Directory Sync tool.

After providing the credentials, click Next to continue.

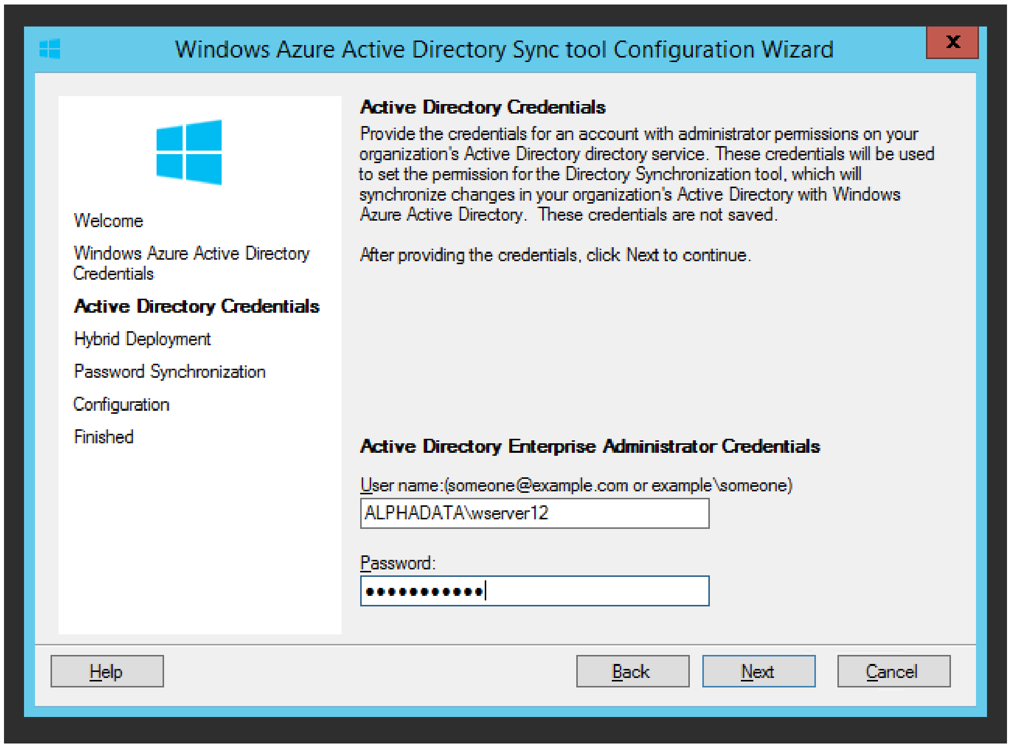
Windows Azure Active Directory Administrator Credentials

User name:(someone@example.com)
gadmin@itadevconn.onmicrosoft.com

Password:
••••••••••••••

Figure 23: Insert the credentials of the user we created earlier

Once the tool accepts our credentials, we are prompted to enter another set of credentials, this time of our on-premise Active Directory tenant.


Windows Azure Active Directory Sync tool Configuration Wizard
Welcome
Windows Azure Active Directory Credentials
Active Directory Credentials
Hybrid Deployment
Password Synchronization
Configuration
Finished
Help
Back
Next
Cancel

Active Directory Credentials

Provide the credentials for an account with administrator permissions on your organization's Active Directory directory service. These credentials will be used to set the permission for the Directory Synchronization tool, which will synchronize changes in your organization's Active Directory with Windows Azure Active Directory. These credentials are not saved.

After providing the credentials, click Next to continue.

Active Directory Enterprise Administrator Credentials

User name:(someone@example.com or example\someone)
ALPHADATA\wserver12

Password:
•••••••••••

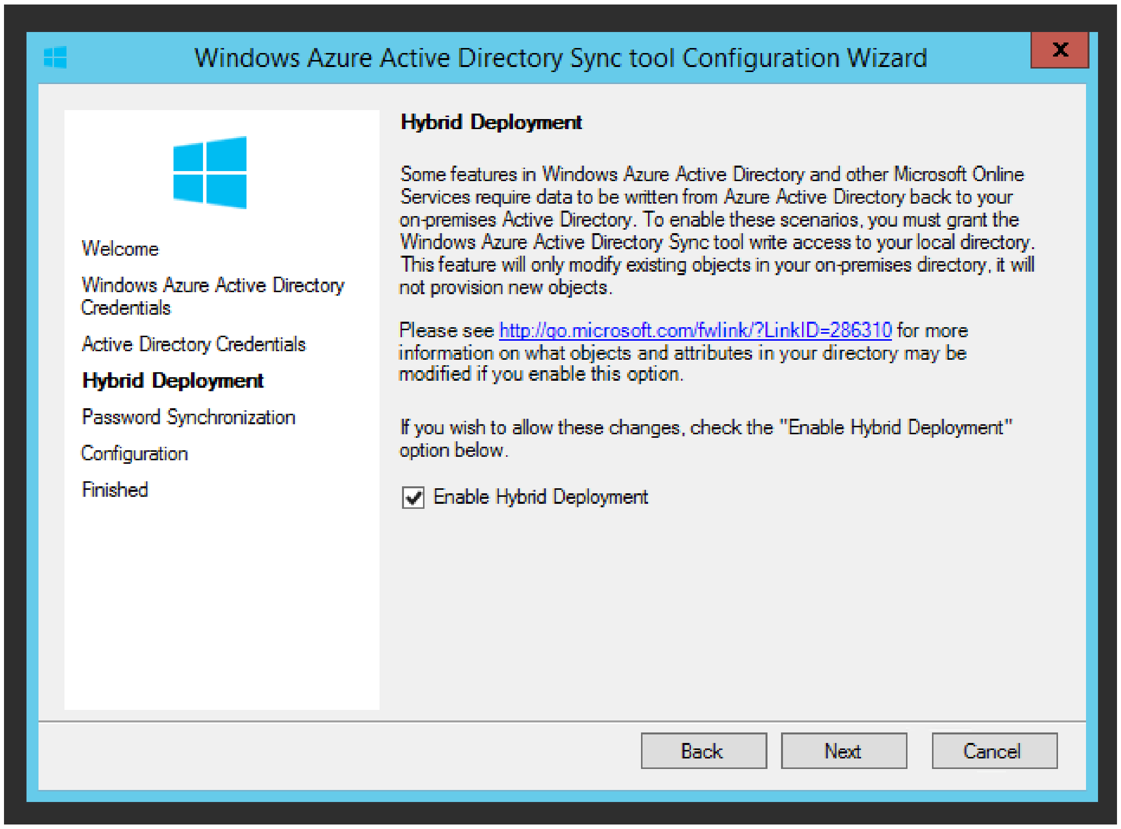
Figure 24: We type in the credential of a local user with Admin privileges

The last two steps before the actual synchronization process starts are to choose whether to enable the Hybrid Deployment and Password Sync features. The Hybrid Deployment features allow our Azure AD to write data back to our on premise AD. Basically, this step grants write access inside our on-premise tenant to our cloud tenant. The Hybrid Deployment does not create new objects inside our AD but only modifies those that already exist.

In our case, we choose to enable the Hybrid Deployment feature.

After enabling/discarding the **Hybrid Deployment** feature, we are prompted to choose whether or not to enable **Password Sync**. This feature, as can be clearly understood by its name, will allow users to access all Microsoft Online Services with the same username/password set they used to access their on premise resources. We choose to enable Password Sync since we want to propagate possible password changes to our cloud AD.

Note: If you'd like to read more about Hybrid Deployment please refer to this link: http://technet.microsoft.com/en-us/library/hh967642.aspx


Windows Azure Active Directory Sync tool Configuration Wizard
Welcome
Windows Azure Active Directory Credentials
Active Directory Credentials
Hybrid Deployment
Password Synchronization
Configuration
Finished
Back
Next
Cancel

Hybrid Deployment

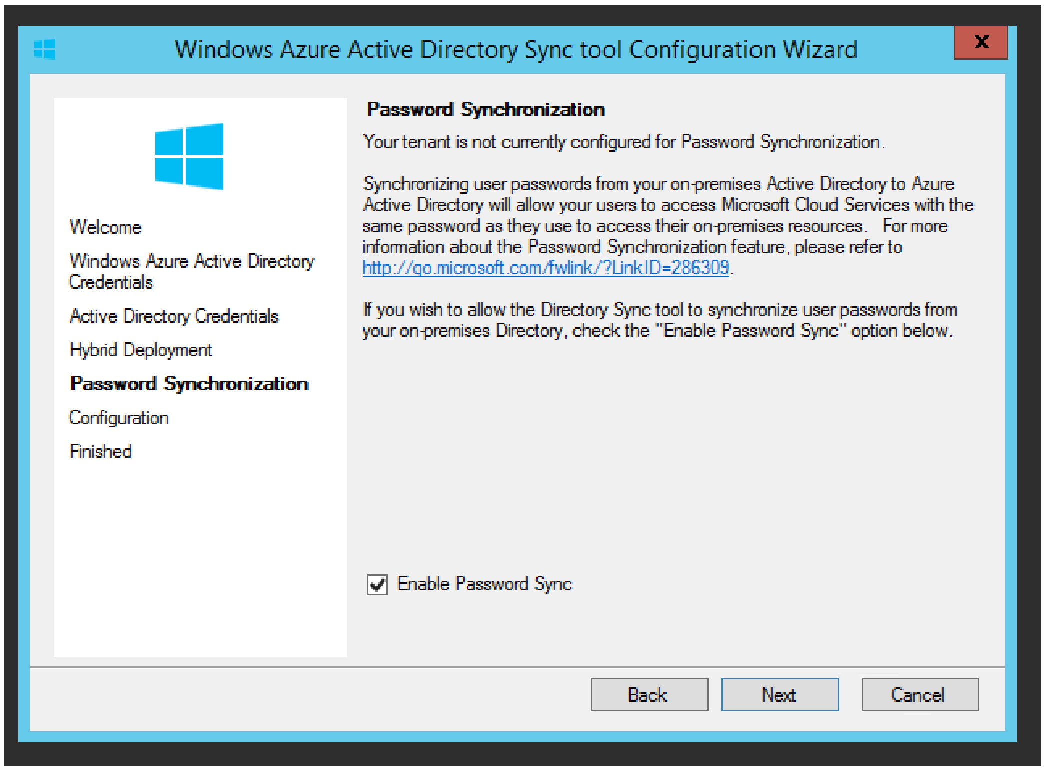
Some features in Windows Azure Active Directory and other Microsoft Online Services require data to be written from Azure Active Directory back to your on-premises Active Directory. To enable these scenarios, you must grant the Windows Azure Active Directory Sync tool write access to your local directory. This feature will only modify existing objects in your on-premises directory, it will not provision new objects.

Please see http://go.microsoft.com/fwlink/?LinkID=286310 for more information on what objects and attributes in your directory may be modified if you enable this option.

If you wish to allow these changes, check the "Enable Hybrid Deployment" option below.

☑ Enable Hybrid Deployment

Figure 25: The Hybrid Deployment screen


Windows Azure Active Directory Sync tool Configuration Wizard
Welcome
Windows Azure Active Directory Credentials
Active Directory Credentials
Hybrid Deployment
Password Synchronization
Configuration
Finished
Back
Next
Cancel

Password Synchronization

Your tenant is not currently configured for Password Synchronization.

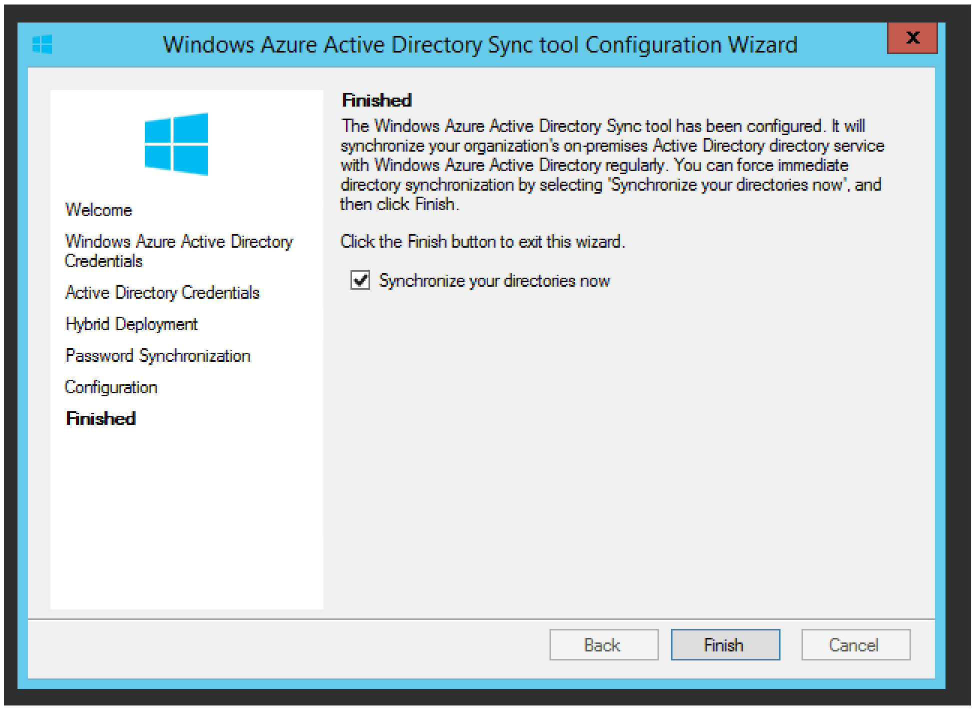
Synchronizing user passwords from your on-premises Active Directory to Azure Active Directory will allow your users to access Microsoft Cloud Services with the same password as they use to access their on-premises resources. For more information about the Password Synchronization feature, please refer to http://go.microsoft.com/fwlink/?LinkID=286309.

If you wish to allow the Directory Sync tool to synchronize user passwords from your on-premises Directory, check the "Enable Password Sync" option below.

☑ Enable Password Sync

Figure 26: Password Sync is enabled

After completing these steps, the Sync Tool will start the configuration process. It'll take about ten minutes. After that we can force the first synchronization process by simply clicking **Finish**.



Windows Azure Active Directory Sync tool Configuration Wizard

Welcome
Windows Azure Active Directory Credentials
Active Directory Credentials
Hybrid Deployment
Password Synchronization
Configuration
Finished

Finished

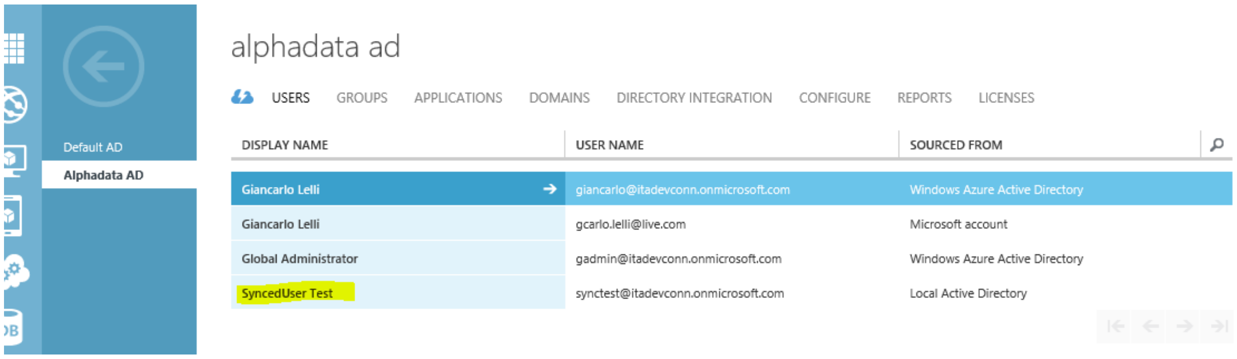
The Windows Azure Active Directory Sync tool has been configured. It will synchronize your organization's on-premises Active Directory directory service with Windows Azure Active Directory regularly. You can force immediate directory synchronization by selecting 'Synchronize your directories now', and then click Finish.

Click the Finish button to exit this wizard.

[✓] Synchronize your directories now

Figure 27: Run the first sync right after the wizard finishes

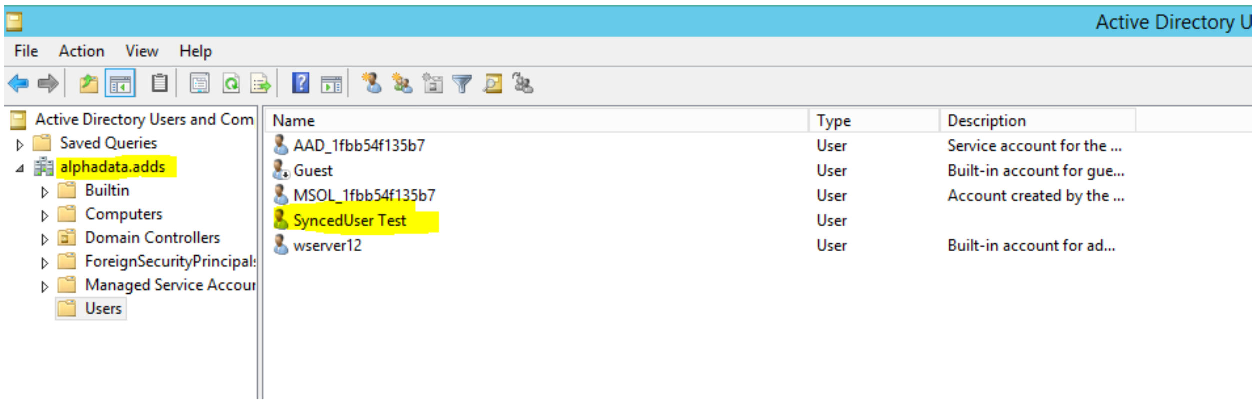
After the process starts, it could take up to 3 hours; after it is finished, the user list in our Azure AD directory should be as shown in Figure 28.



DISPLAY NAME	USER NAME	SOURCED FROM
Giancarlo Lelli	giancarlo@itadevconn.onmicrosoft.com	Windows Azure Active Directory
Giancarlo Lelli	gcarlo.lelli@live.com	Microsoft account
Global Administrator	gadmin@itadevconn.onmicrosoft.com	Windows Azure Active Directory
SyncedUser Test	synctest@itadevconn.onmicrosoft.com	Local Active Directory

Figure 28: The directory has successfully synced our users

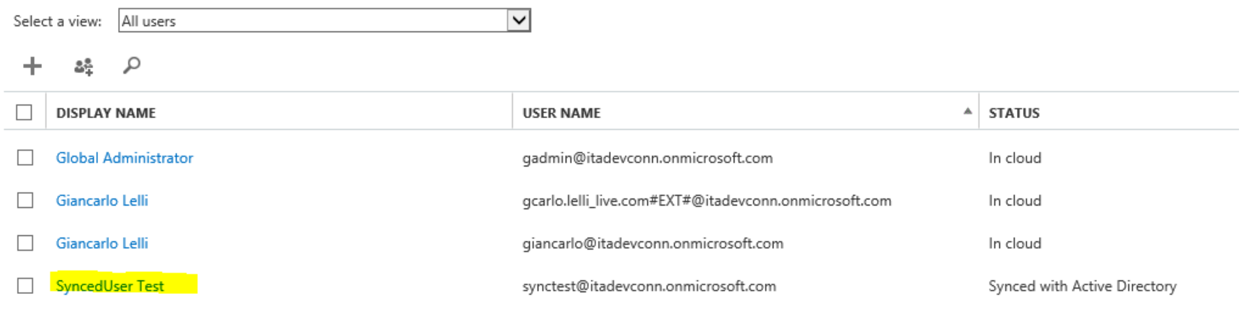
This is the list of users that are available in our on premise Active Directory.



Name	Type	Description
AAD_1fbb54f135b7	User	Service account for the ...
Guest	User	Built-in account for gue...
MSOL_1fbb54f135b7	User	Account created by the ...
SyncedUser Test	User	
wserver12	User	Built-in account for ad...

Figure 29: Proof that the user exists even in the local environment

Finally, the list of users inside our Office 365 tenant is as follows.



DISPLAY NAME	USER NAME	STATUS
Global Administrator	gadmin@itadevconn.onmicrosoft.com	In cloud
Giancarlo Lelli	gcarlo.lelli_live.com#EXT#@itadevconn.onmicrosoft.com	In cloud
Giancarlo Lelli	giancarlo@itadevconn.onmicrosoft.com	In cloud
SyncedUser Test	synctest@itadevconn.onmicrosoft.com	Synced with Active Directory

Figure 30: It also exists in our Office 365 tenant

Important Notes About Syncing

In order to get the synchronization to work when adding a new user to the on premise Active Directory, we had to tweak our tenant, and added to the list of supported UPNs the name “**itadevconn.onmicrosoft.com**.” The reason for this is that DirSync will only synchronize DNS suffixes that are available on your Windows Azure Active Directory as a verified domain. When we look at the verification process, it shows us that in either case, with TXT or MX, we need to add a record to the public available DNS for that domain. In our scenario, we are using a DNS suffix such as “**alphadata.adds**” which is not publicly routable, and this is a problem. The easiest solution is to register one, or more if needed, new UPN suffixes on your Active Directory. This can be done through **Server Manager** > **Tool** > **Active Directory Domains and Trusts.**

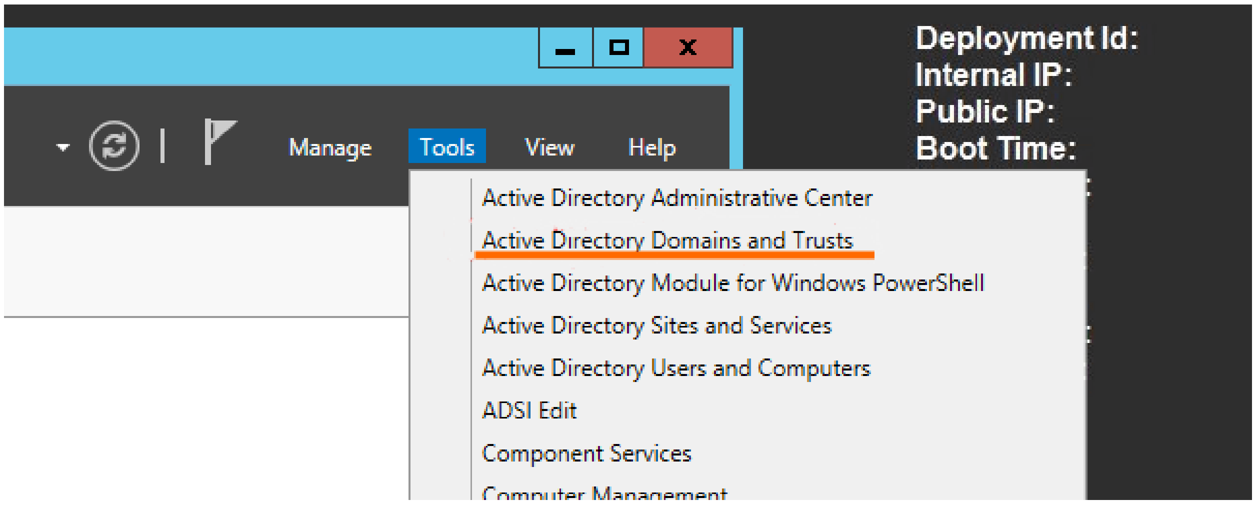
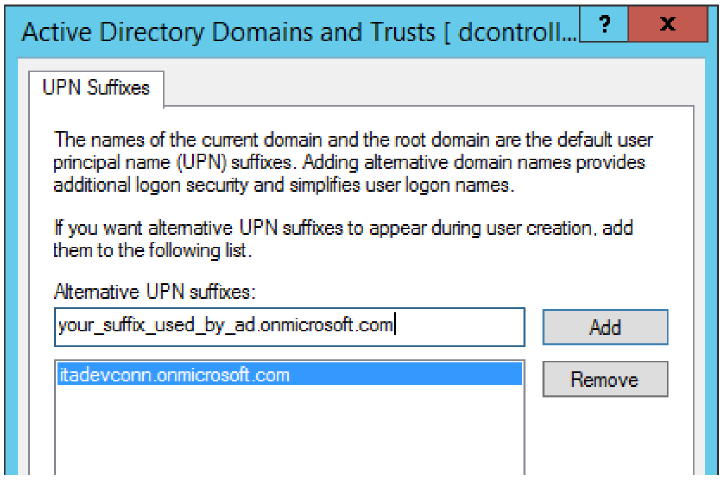

Deployment Id:
Internal IP:
Public IP:
Boot Time:
Manage
Tools
View
Help
Active Directory Administrative Center
Active Directory Domains and Trusts
Active Directory Module for Windows PowerShell
Active Directory Sites and Services
Active Directory Users and Computers
ADSI Edit
Component Services

Figure 31: Click Active Directory Domains and Trusts"

Right-click **Active Directory Domain and Trusts** and select **Properties**. Now type the UPN suffix that our Azure AD uses inside the textbox and click **Add**.



The names of the current domain and the root domain are the default user principal name (UPN) suffixes. Adding alternative domain names provides additional logon security and simplifies user logon names.

If you want alternative UPN suffixes to appear during user creation, add them to the following list.

Alternative UPN suffixes:

your_suffix_used_by_ad.onmicrosoft.com

itadevconn.onmicrosoft.com

Figure 32: Add a new UPN prefix

Now that we have done this, when we try to create a new user, we will have the ability to choose a different suffix. However, if we now choose the old one during the synchronization process on Azure, it will be replaced with the new one.

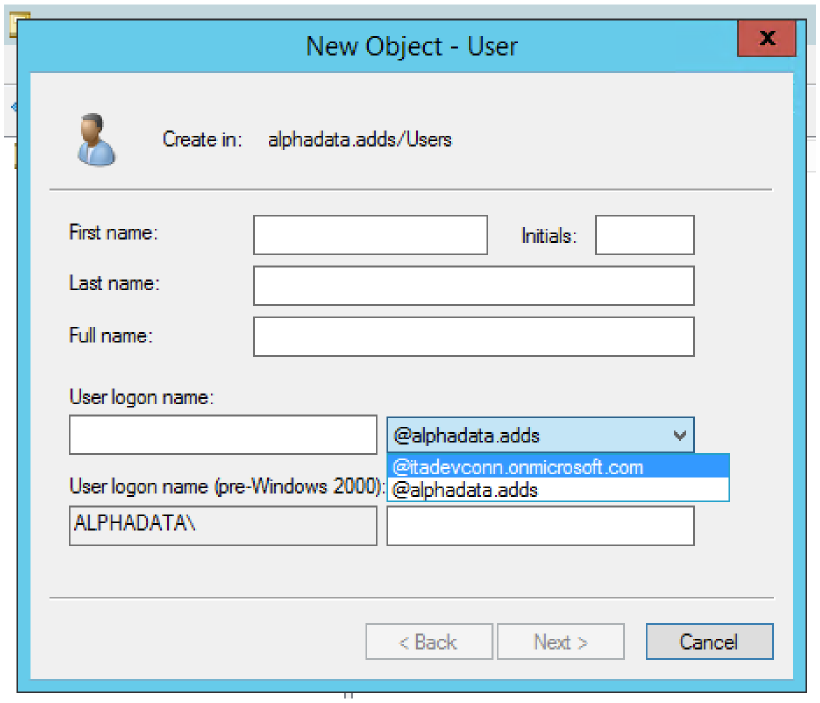


Figure 33: The new prefix is now available when creating a new user object

Note: If you want to test this functionality, you can create a test user (with either the new or old UPN suffix) and then start the synchronization process manually by typing in the PowerShell console the following commands:

- ***Import-Module DirSync***
- ***Start-OnlineCoExistenceSync -fullsync***

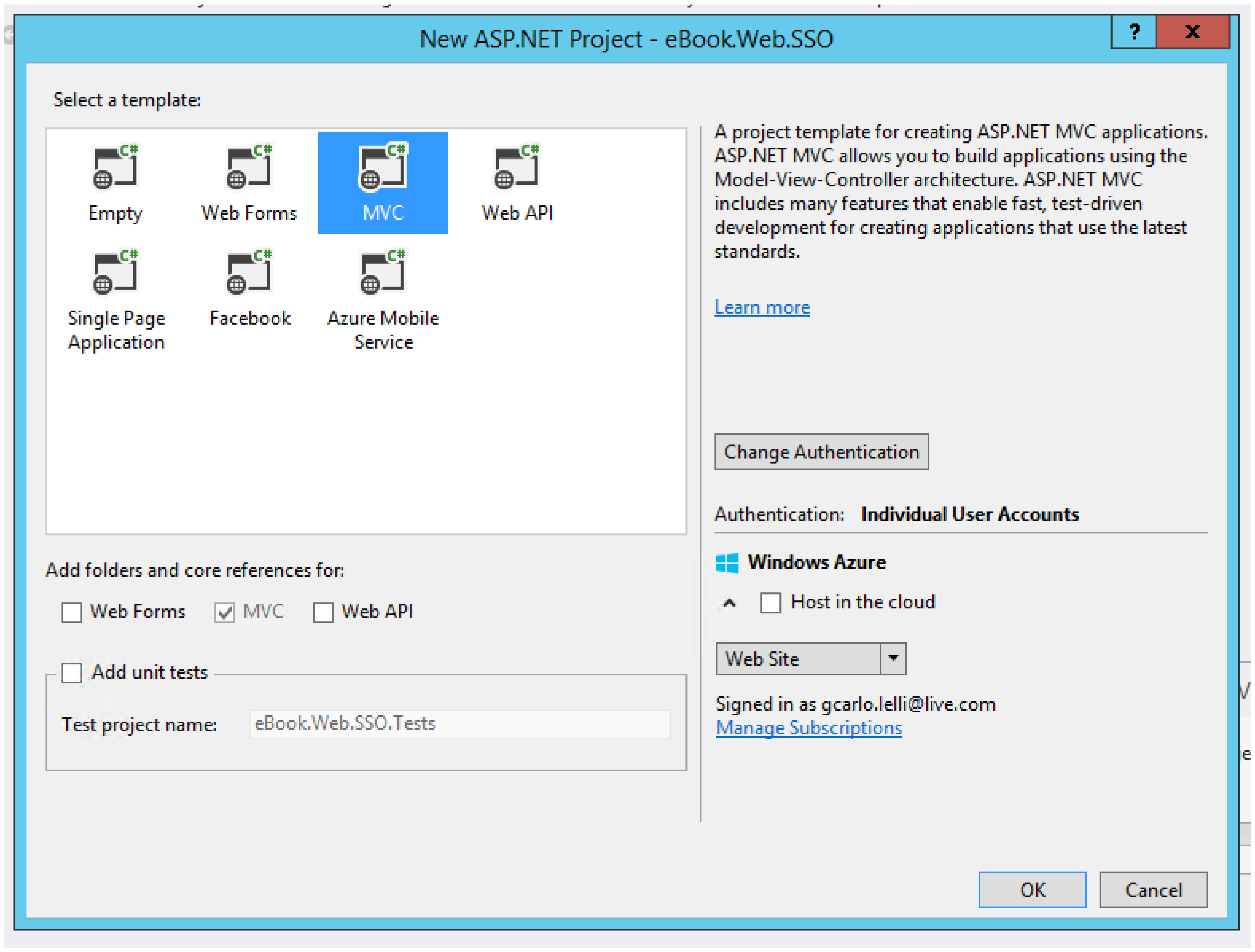
Now both on premise and cloud AD are in harmony; we can now start building web applications that leverage the SSO functionalities.

Leveraging SSO Functionalities in Web Applications

In this part of the book, we will see how easy it is to create a web application that leverages SSO functionalities. In addition, we will see how to secure a Web API endpoint by making it accessible only to our company employees (AD users). All of these operations require an initial configuration process, but Visual Studio (versions 2013 and above) takes care of all the low-level work.

An MVC Application That Authenticates Against Our AD

We shall create a basic ASP.NET MVC 5 application that supports SSO. In order to do so, open Visual Studio (Professional 2014 CTP2 in my case) and click **File**, **New Project**, then select **ASP.NET Web Application;** at this point this popup should appear:


New ASP.NET Project - eBook.Web.SSO

Select a template:

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Learn more

Change Authentication

Authentication: **Individual User Accounts**

Windows Azure

Host in the cloud

Web Site

Signed in as gcarlo.lelli@live.com

Manage Subscriptions

Add folders and core references for:

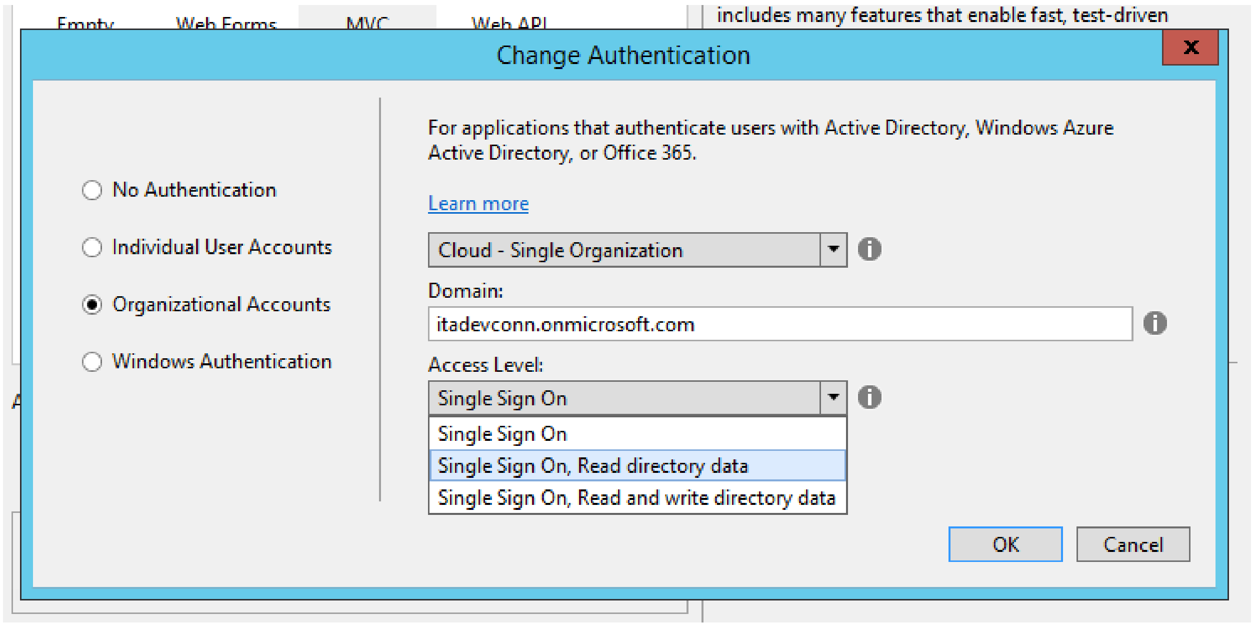
Web Forms [✓ MVC] Web API

Add unit tests

Test project name: eBook.Web.SSO.Tests

Figure 34: Choose an MVC project template

Of course, we now select the MVC template, but before clicking ok there are a couple of things that are worth underlining. First is the ability (provided by the latest version of the Azure SDK) to host our projects in the cloud on an Azure Website, and second is the ability to change the type of authentication; this is where we should focus now. Click **Change Authentication**. This popup should appear:



Change Authentication

- ( ) No Authentication
- ( ) Individual User Accounts
- (●) Organizational Accounts
- ( ) Windows Authentication

For applications that authenticate users with Active Directory, Windows Azure Active Directory, or Office 365.

Learn more

Cloud - Single Organization

Domain: itadevconn.onmicrosoft.com

Access Level: Single Sign On

- Single Sign On
- Single Sign On, Read directory data
- Single Sign On, Read and write directory data

Figure 35: The three different types of access level

Click on the radio button labeled **Organizational Account** and fill the **Domain** field with the domain associated with our Azure AD. For testing purposes, I have used the default domain that Azure gave me, but in a real case scenario, you can also use your own custom domain.

Talking about the **Access Level**, we have three different possibilities. We are supposed to choose the one the most fits our needs. However, these are some suggestions that might help you to make this decision.

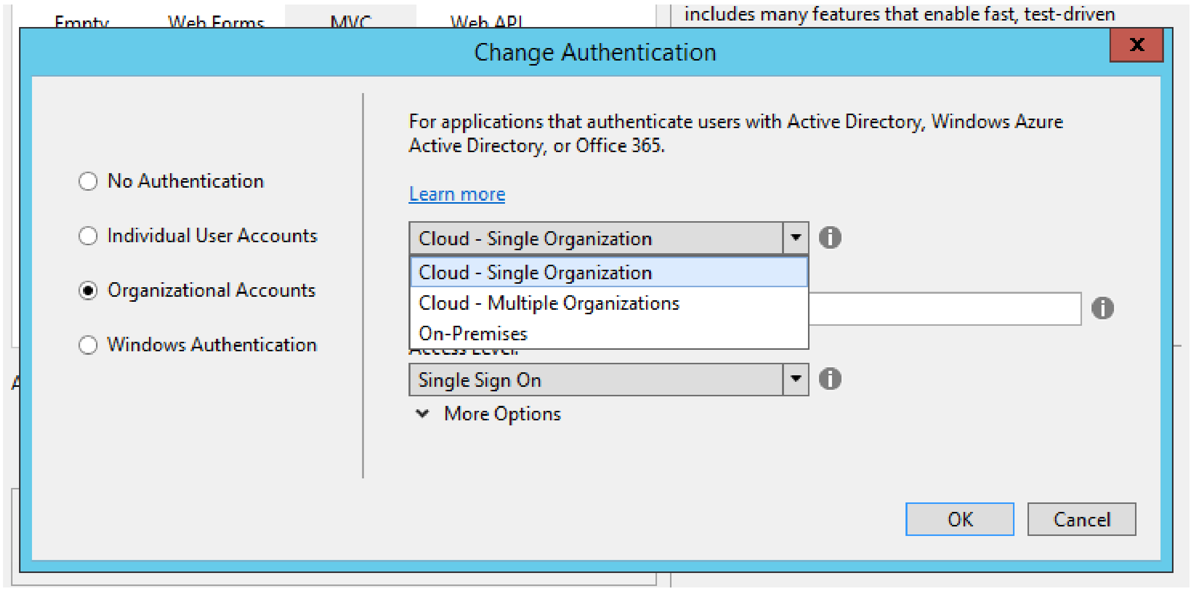
- **Single Sign On** option: Your application will be able to login using your AD credential, giving you access to a very limited set of information about the user, such as first, last, and full name. The token that the application will receive will not be valid (your application will be rejected) in case you try to reach out to your AD for querying it.
- **Single Sign On, Read Directory Data** option: You will be able to login and the token your application will obtain will be valid for querying the directory.
- **Single Sign On, Read and write directory data** option: You have full control of your Active Directory.

Both reading and writing operations are done via the Graph API, a REST-based programmatic interface designed to let apps from any platform with an HTTP stack gain delegated access to the directory.

Concerning the first “combo-box” that lets us choose what type of AD we want to target and how many organization we are targeting, the decision is quite easier.

- **Cloud—Single Organization:** Only the employees of your company will use the application you are developing.

- **Cloud—Multiple Organization:** You are developing a SaaS (Software-as-a-Service) platform and you are planning to sell it to various customers, each with their own AD/domain.
- **On Premise:** You have an AD tenant “inside” your company that will take care of handling the login flow; in this case, you only have to specify the URL of the metadata document.



Change Authentication

For applications that authenticate users with Active Directory, Windows Azure Active Directory, or Office 365.

Learn more

( ) No Authentication
( ) Individual User Accounts
(●) Organizational Accounts
( ) Windows Authentication

Cloud - Single Organization
- Cloud - Single Organization
- Cloud - Multiple Organizations
- On-Premises

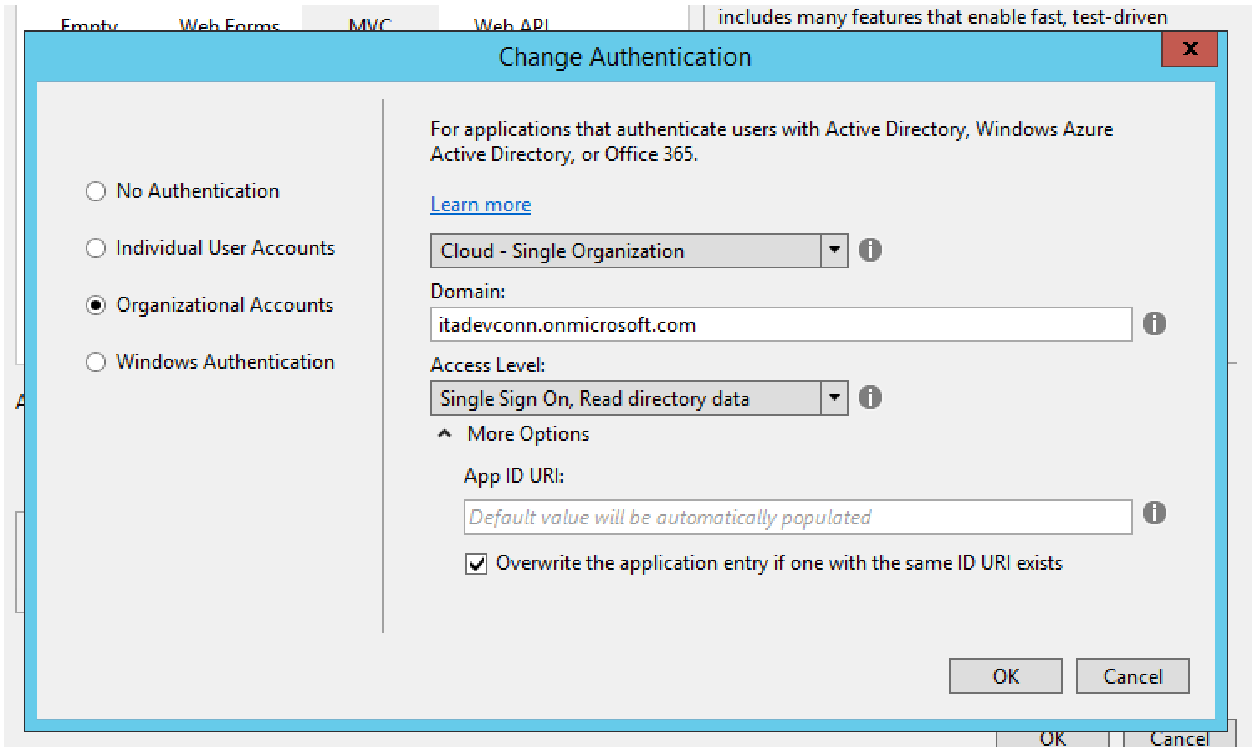
Access Level: Single Sign On

More Options

OK Cancel

Figure 36: The different scenarios for our web application

The **More Options** selection is something that we can ignore since it consists of a set of parameters that Visual Studio automatically generates; however, for the sake of completeness, we will briefly talk about them too.



Change Authentication

- [ ] No Authentication
- [ ] Individual User Accounts
- [✓] Organizational Accounts
- [ ] Windows Authentication

For applications that authenticate users with Active Directory, Windows Azure Active Directory, or Office 365.

Learn more

Cloud - Single Organization

Domain: itadevconn.onmicrosoft.com

Access Level: Single Sign On, Read directory data

More Options

App ID URI: *Default value will be automatically populated*

✓ Overwrite the application entry if one with the same ID URI exists

Figure 37: The More Options menu

The **App ID URI** is a unique identifier that distinguishes our application in the list of all the trusted applications that are allowed to connect/authenticate/query our AD. This value is automatically generated by Visual Studio and, in case a duplication occurs, the old entry is overwritten by the more recent (as you can predict by the checked checkbox in the picture).

You may be wondering, what is the difference between an organizational account and a Windows Authentication? Well, it resides in the kind of application we are developing. A web application that supports windows authentication will be hosted and available through our intranet; hence, it will use NTLM as the authentication protocol instead of WS-Federation, which is what we use in our case.

We are now ready to create our project. In our case, we choose a **Single Organization** with **SSO and read directory data permissions**. As soon as we click, a dialog will appear asking us to login; this dialog is reaching out to our Azure AD in order to get a token to write an entry inside the whitelisted application list of our Azure AD. We now login to a global administrator.

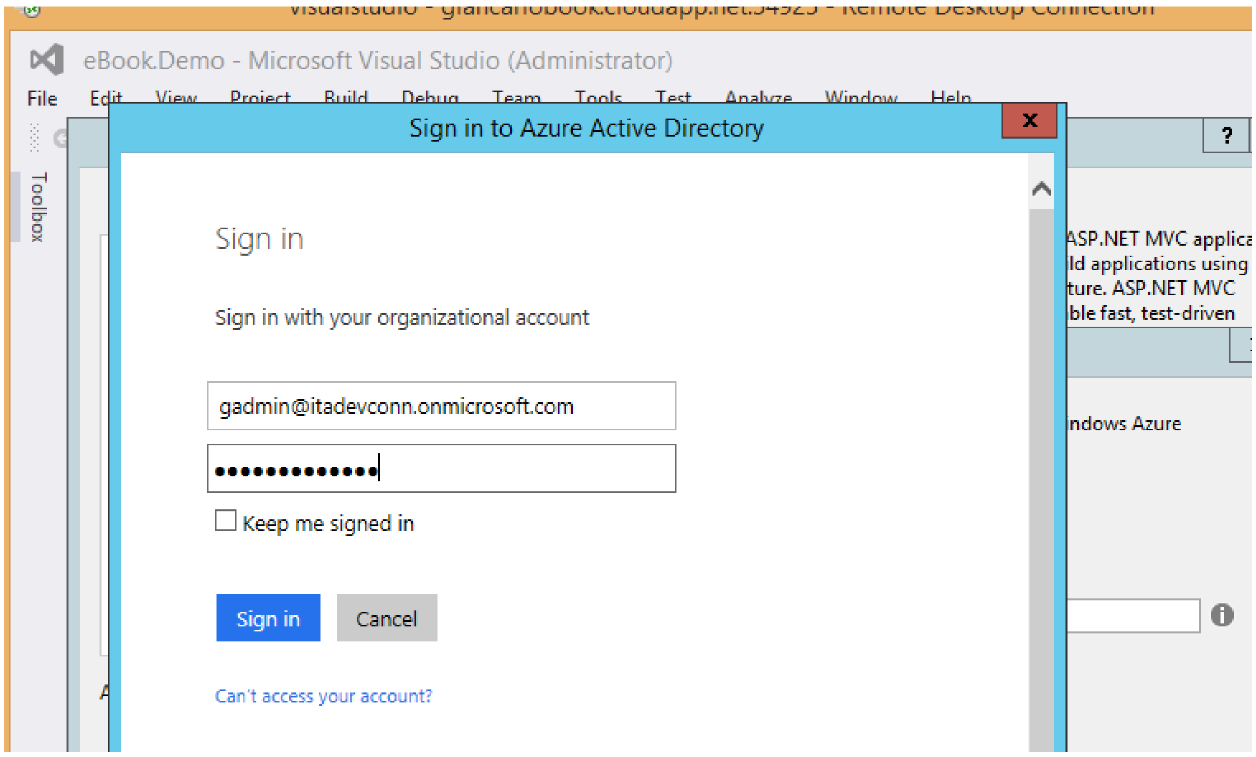
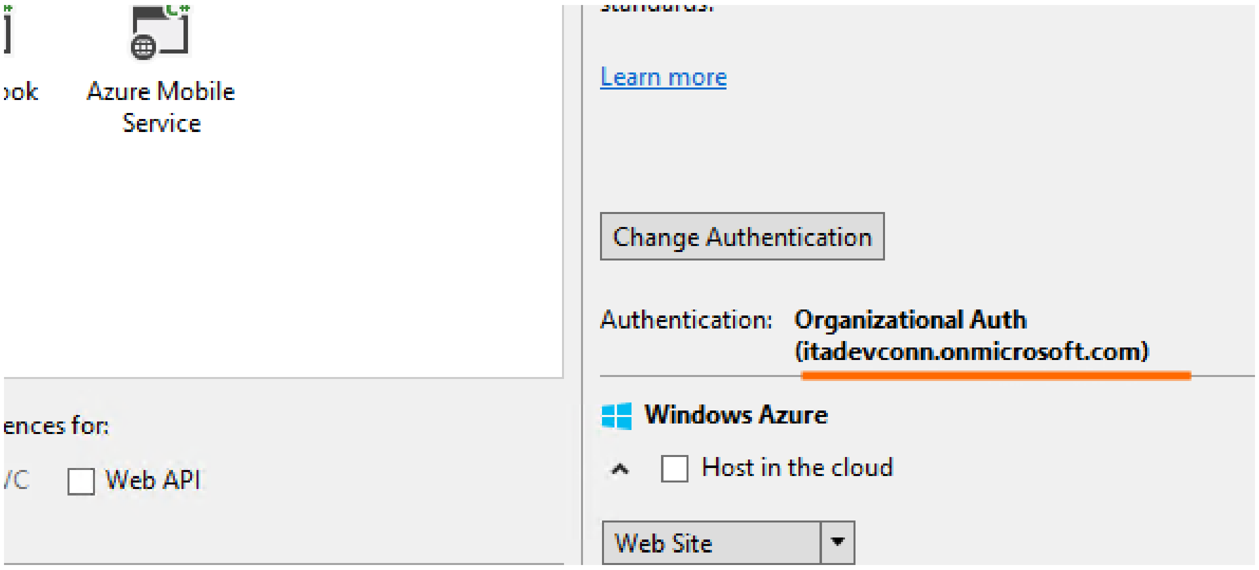

eBook.Demo - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Sign in to Azure Active Directory
Sign in
Sign in with your organizational account
gadmin@itadevconn.onmicrosoft.com
Keep me signed in
Sign in
Cancel
Can't access your account?

Figure 38: Login to authenticate the app

As soon as the process ends, we should now see that our application is trusted:



Authentication: **Organizational Auth (itadevconn.onmicrosoft.com)**

Figure 39: Successfully authenticated

Click **Ok** and Visual Studio will start creating our project. Let's spend some time analyzing the changes to our **Web.config** file. The first thing we notice is that Visual Studio added some key and values to our configuration file during project setup.

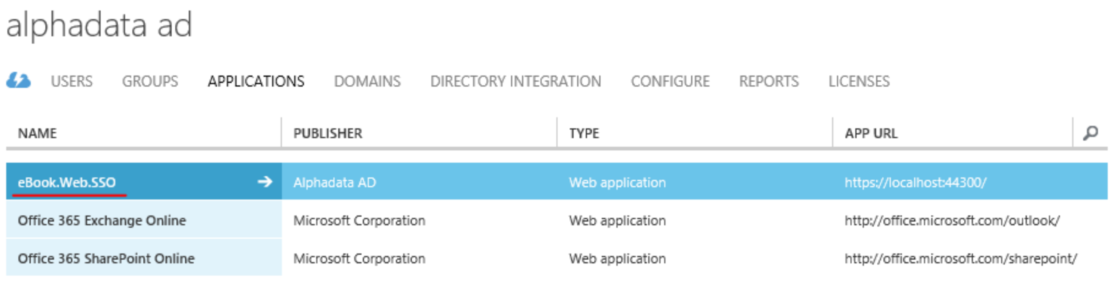
```
  <appSettings>
    <add key="ida:FederationMetadataLocation"
value="https://login.windows.net/itadevconn.onmicrosoft.com/FederationMetad
ata/2007-06/FederationMetadata.xml" />
    <add key="ida:Realm"
value="https://itadevconn.onmicrosoft.com/eBook.Web.SSO" />
    <add key="ida:AudienceUri"
value="https://itadevconn.onmicrosoft.com/eBook.Web.SSO" />
    <add key="ida:ClientID" value="9797a190-91ca-4dde-b430-509c20549154" />
    <add key="ida:Password"
value="Nu0CLukD1xHTgxfpvjBfoyMi6n7H5UECVD33Ef9QU5w=" />
  </appSettings>
```

As we can see, these values represent the identifiers that Azure AD uses to recognize our application during the user sign-in process.

If we scroll toward the end of the file, we also see some other changes that represent the parameter that we pass to the WS-Federation protocol during the authentication process.

```
  <system.identityModel.services>
    <federationConfiguration>
      <cookieHandler requireSsl="true" />
      <wsFederation passiveRedirectEnabled="true"

issuer="https://login.windows.net/itadevconn.onmicrosoft.com/wsfed"

realm="https://itadevconn.onmicrosoft.com/eBook.Web.SSO"
                    requireHttps="true" />
    </federationConfiguration>
  </system.identityModel.services>
```

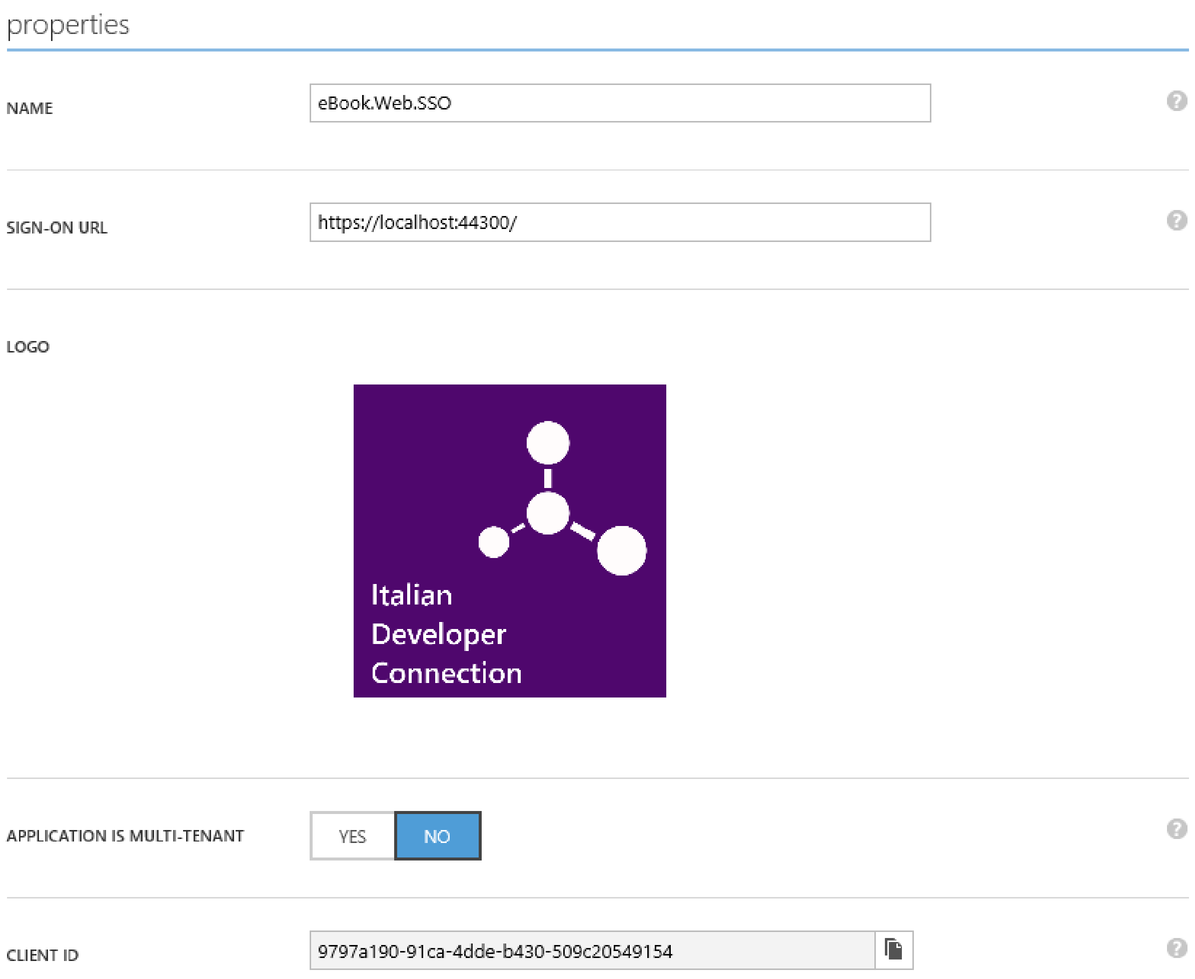
This is basically all that happens client-side. However, what has been the server-side cause of this configuration process (in our Azure AD settings)? Let's see: if we navigate to the Azure management portal and then inside our Azure AD (AlphaData AD) application tab, we should see something like the following:



NAME	PUBLISHER	TYPE	APP URL
eBook.Web.SSO	Alphadata AD	Web application	https://localhost:44300/
Office 365 Exchange Online	Microsoft Corporation	Web application	http://office.microsoft.com/outlook/
Office 365 SharePoint Online	Microsoft Corporation	Web application	http://office.microsoft.com/sharepoint/

Figure 40: The list of trusted apps by our Azure AD

Our application (still under development) is listed. The Azure management portal gives us the ability to manually create an entry in this list by clicking the **Add** button at the bottom of the page. However, that is only needed if you are planning to "trust" an application that you do not own or that is being developed with a different IDE. Otherwise, you just need to give it an App ID and specify its endpoints. If we navigate inside the details of this application, we can see some more settings and info about our application.


properties

NAME: eBook.Web.SSO

SIGN-ON URL: https://localhost:44300/

LOGO

APPLICATION IS MULTI-TENANT: YES [✓ NO]

CLIENT ID: 9797a190-91ca-4dde-b430-509c20549154

Figure 41: The configuration page

Moving into the **Configuration** tab, we have various sub-sections. The **Property** section reports all the fields that were automatically populated by Visual Studio and that are also stored inside the web.config file. All the other information on this page is about SSO parameters, such as the **Reply URL** that is used to redirect the user once that the login process is completed; then we have a brief recap of the application permissions together with the application key. However, as I said earlier, Visual Studio takes care of this entire configuration, so we do not need to dive into the details.

Now that we have described both the client and server sides of our application, if we run in locally, as soon as IIS is ready, then we should be asked to login using our company credentials. After a bit of magic (our app reaching out to our Azure AD), we can see in the following picture that we should be logged in as a company user:

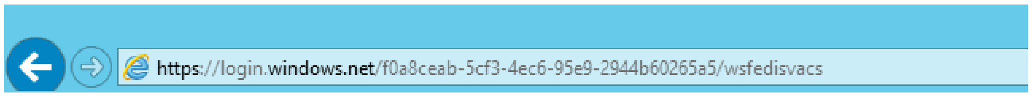
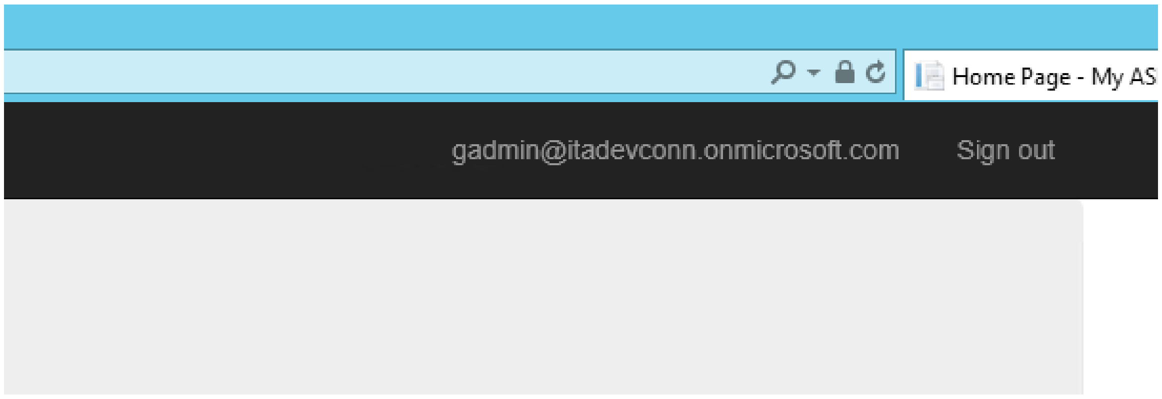

https://login.windows.net/f0a8ceab-5cf3-4ec6-95e9-2944b60265a5/wsfedisvacs

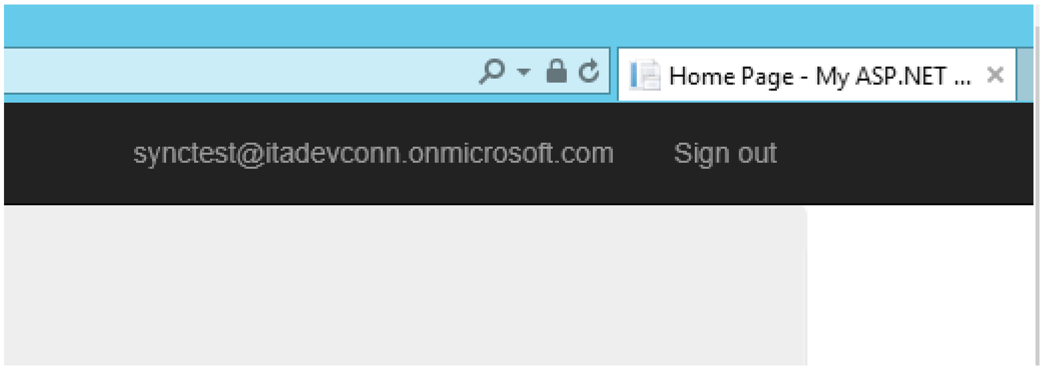
Figure 42: This is where the magic happens



gadmin@itadevconn.onmicrosoft.com

Figure 43: We are logged in

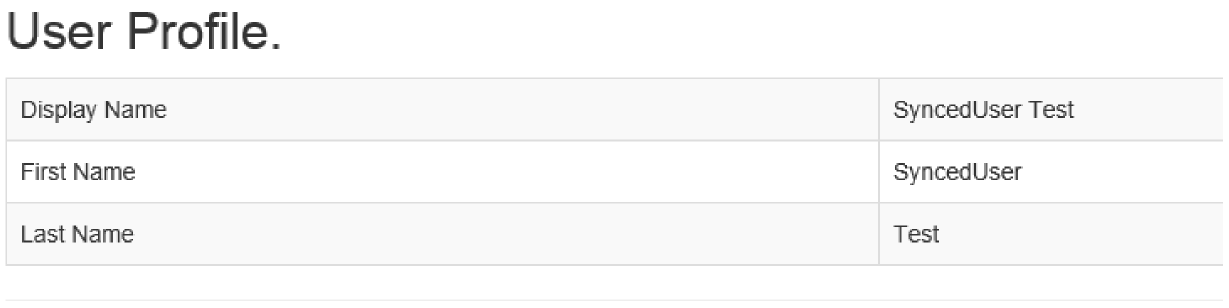
However, this user wasn't really synced with our on-premise AD, so we will now try to login with a user that has been synced, such as **SyncedUser Test**.



synctest@itadevconn.onmicrosoft.com

Figure 44: We successfully logged in with a synced account

And that's it, we now have successfully implemented SSO inside our MVC application. Let's move inside the default user profile page to see what information are available at the moment.

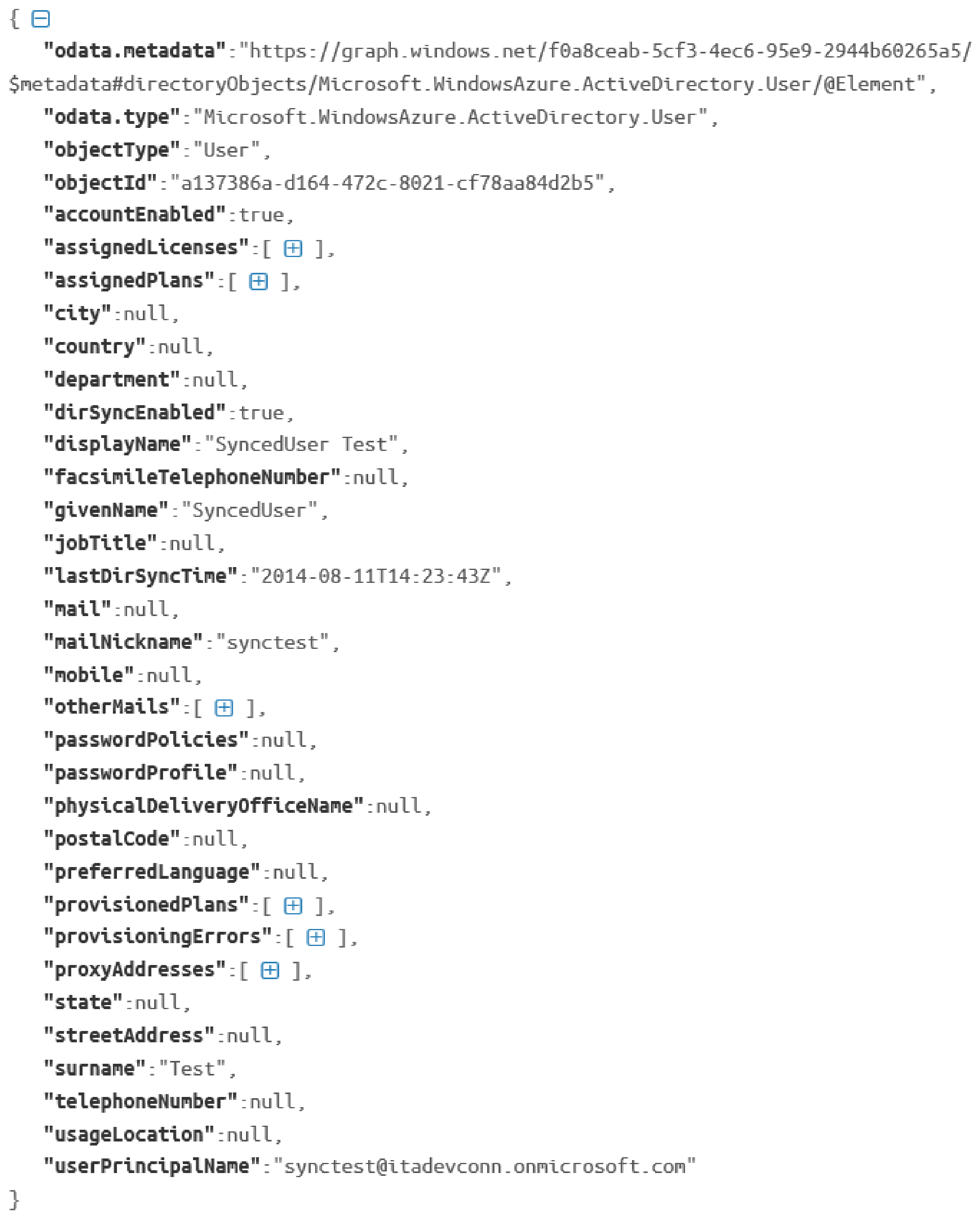


User Profile.

Display Name	SyncedUser Test
First Name	SyncedUser
Last Name	Test

Figure 45: Some basic info about our user

The default MVC template for Organizational Login provides a quick glance of what information is available out of the box; in this case (and in every new project), we are able to see the Display, First, and Last names. However, if we look at the JSON that is available after querying AD, we see that we only displayed this information because these are the only fields that are deserialized properly. You can find the full JSON in Figure 46:



```
{ ⊟
   "odata.metadata":"https://graph.windows.net/f0a8ceab-5cf3-4ec6-95e9-2944b60265a5/
$metadata#directoryObjects/Microsoft.WindowsAzure.ActiveDirectory.User/@Element",
   "odata.type":"Microsoft.WindowsAzure.ActiveDirectory.User",
   "objectType":"User",
   "objectId":"a137386a-d164-472c-8021-cf78aa84d2b5",
   "accountEnabled":true,
   "assignedLicenses":[ ⊞ ],
   "assignedPlans":[ ⊞ ],
   "city":null,
   "country":null,
   "department":null,
   "dirSyncEnabled":true,
   "displayName":"SyncedUser Test",
   "facsimileTelephoneNumber":null,
   "givenName":"SyncedUser",
   "jobTitle":null,
   "lastDirSyncTime":"2014-08-11T14:23:43Z",
   "mail":null,
   "mailNickname":"synctest",
   "mobile":null,
   "otherMails":[ ⊞ ],
   "passwordPolicies":null,
   "passwordProfile":null,
   "physicalDeliveryOfficeName":null,
   "postalCode":null,
   "preferredLanguage":null,
   "provisionedPlans":[ ⊞ ],
   "provisioningErrors":[ ⊞ ],
   "proxyAddresses":[ ⊞ ],
   "state":null,
   "streetAddress":null,
   "surname":"Test",
   "telephoneNumber":null,
   "usageLocation":null,
   "userPrincipalName":"synctest@itadevconn.onmicrosoft.com"
}
```

Figure 46: The response JSON by the Azure Graph

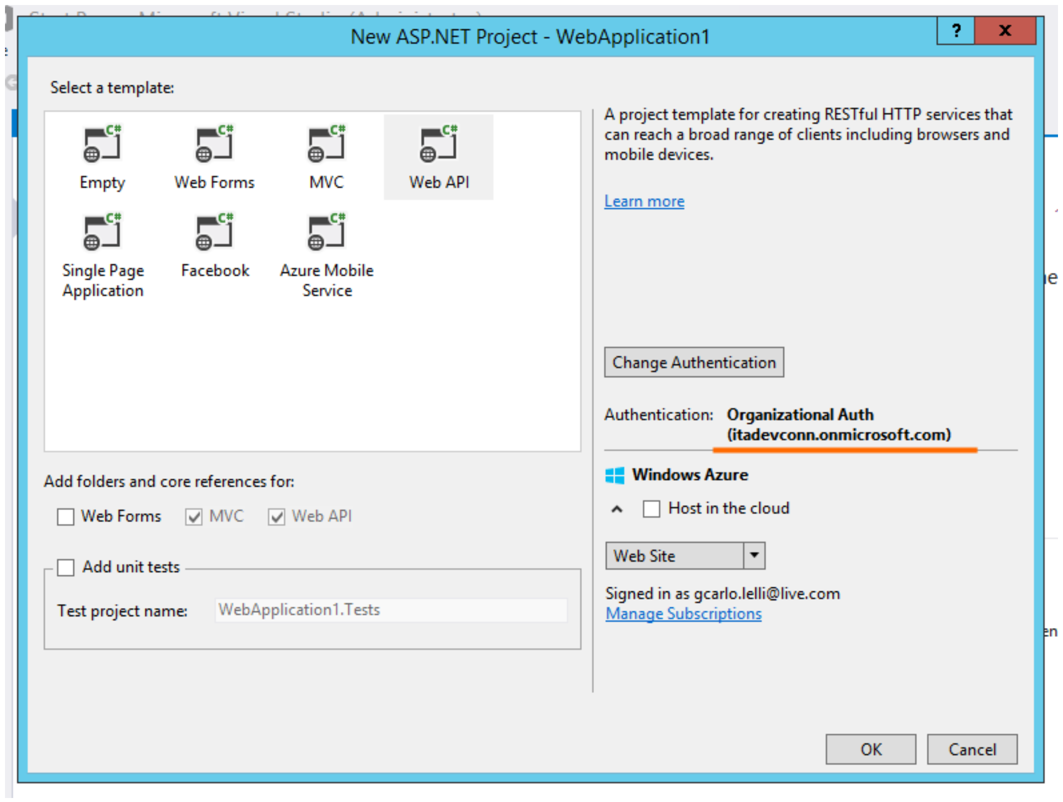
That is a huge amount of data, isn’t it? Later in this book, we will cover in detail how this query towards AD was possible and what tooling supports Visual Studio provides.

Building a Secure Web API Endpoint

Exposing data in a RESTful manner is becoming more and more convenient these days, because it allows every device that supports or has an HTTP stack to talk with our web endpoint and consume our data. However, exposing a web endpoint that provides sensitive information about our company to its clients, without designing some kind of security level that blocks unauthorized clients, may give rise to some serious security issues. In this part of the book, we will build a basic ASP.NET Web API project which uses our organizational directory (AlphaData AD) on the cloud as a security layer, in order to allow only authenticated clients to consume our services.

With the release of Visual Studio 2013, you can leave all that behind. This edition introduces innovative ASP.NET tooling and security middleware from the Microsoft Open Web Interface for .NET (OWIN) components that make protecting your Web API straightforward. The new ASP.NET tools and templates let you configure a Web API project to outsource authentication to Windows Azure Active Directory (AD) directly, emitting the necessary code in the local project and in the corresponding entries in Windows Azure AD.

Building a secured Web API endpoint is pretty easy; in fact, all we have to do is repeat almost every step we’ve made when we talked about a normal MVC app. However, in this case, there’s a little difference. First we choose Web API as the project type:


New ASP.NET Project - WebApplication1

Select a template:

A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

Authentication: Organizational Auth (itadevconn.onmicrosoft.com)

Add folders and core references for:

☐ Web Forms ☑ MVC ☑ Web API

☐ Add unit tests

Test project name: WebApplication1.Tests

Windows Azure

☐ Host in the cloud

Web Site

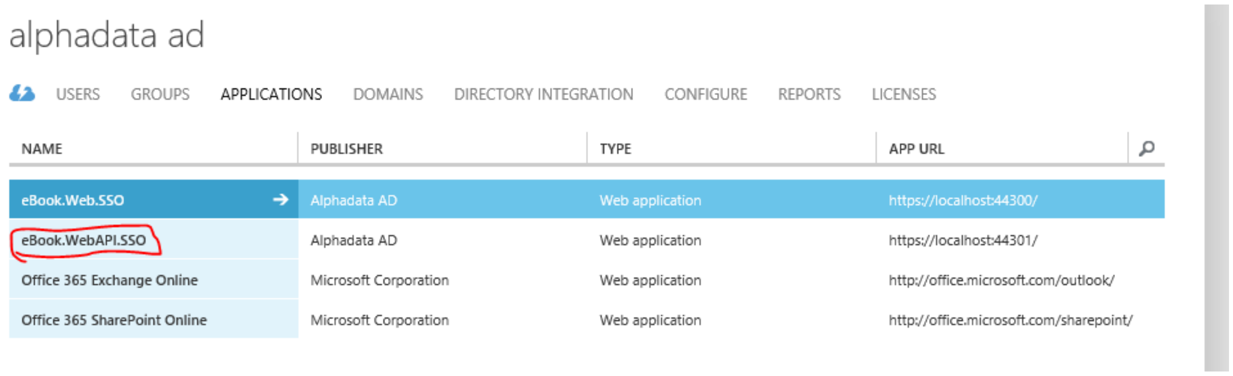
Signed in as gcarlo.lelli@live.com

Figure 47: Choose the Web API Template

Change the authentication type to **Organizational Account** and select the best options for your case. As before, I chose **Single Organization** and **Single Sign On, Read Directory data**.

Note: At the time of writing, Web API project templates do not yet support multi-tenant authorization.

Once you are done with the preliminary steps, you can verify by checking in the Azure portal if your newly created web application is listed in the **Application** tab.


alphadata ad

NAME	PUBLISHER	TYPE	APP URL
eBook.Web.SSO	Alphadata AD	Web application	https://localhost:44300/
eBook.WebAPI.SSO	Alphadata AD	Web application	https://localhost:44301/
Office 365 Exchange Online	Microsoft Corporation	Web application	http://office.microsoft.com/outlook/
Office 365 SharePoint Online	Microsoft Corporation	Web application	http://office.microsoft.com/sharepoint/

Figure 48: The new web application is now trusted by our AD

We now have a fully configured and secure Web API endpoint; let's take a closer look at the project in Visual Studio. If we open the file *Startup.cs,* we see a method called *ConfigureAuth(IAppBuilder)*; this method is defined inside the file *Startup.Auth.cs*, which is contained in the *App_Start* folder. Let's analyze this method's implementation.

```
public void ConfigureAuth(IAppBuilder app)
{
    app.UseWindowsAzureActiveDirectoryBearerAuthentication(
        new WindowsAzureActiveDirectoryBearerAuthenticationOptions
        {
            Audience = ConfigurationManager.AppSettings["ida:Audience"],
            Tenant = ConfigurationManager.AppSettings["ida:Tenant"]
        });
}
```

The **app.Use*** naming convention suggests the method adds a middleware implementation to the OWIN pipeline. In this case, the added middleware inspects the incoming request to see if the HTTP header Authorization contains a security token. If it finds a token, it is validated via a number of standard checks. If the token looks good, the middleware projects its content in a principal, assigns the principal to the current user, and cedes control to the next element in the pipeline. If the token does not pass the checks, the middleware sends back the appropriate error code. If there is no token, the middleware simply lets the call go through without creating a principal.

The single parameter passed to the middleware, **WindowsAzureActiveDirectoryBearerAuthenticationOptions**, supplies the settings for determining a token's validity. It captures the raw values during project creation and stores them in the **Web.config** file. The **Audience value** is the identifier by which the Web API is known to Windows Azure AD. Any tokens carrying a different Audience are meant for another resource and should be rejected. The Tenant property indicates the Windows Azure AD tenant used to outsource authentication. The middleware uses that information to access the tenant and read all the other properties that determine the validity of a token, such as which key should be used to verify the token's signatures.

Those few auto-generated lines of code are all that is needed to authenticate callers with Windows Azure AD. By default, the **Values** controller is decorated with the **[Authorize]** attribute; it will require an **Authorization** header in the request, otherwise it will return an error.

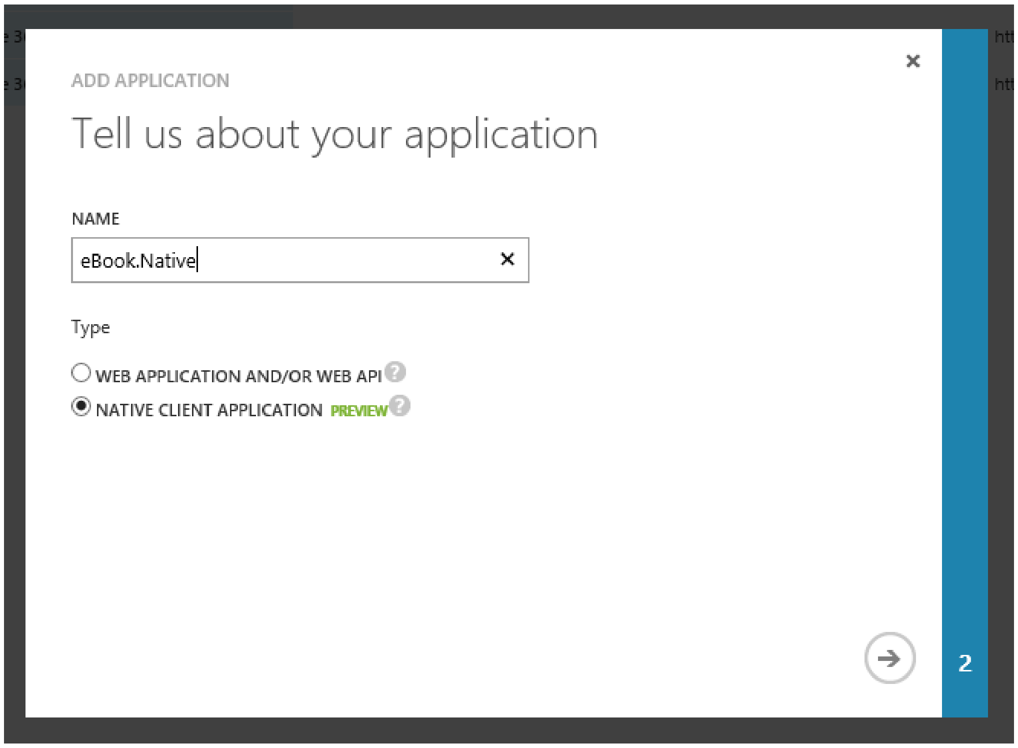
```
[Authorize]
public class ValuesController : ApiController
{
    // GET api/values
    public string Get()
    {
        return "Greetings from a secure Web API";
    }

    // GET api/values/5
    public string Get(int id)
    {
        return "value";
    }
}
```

The only thing we have left to do is to test our Web API to make sure it is secure; in order to do that, we need to create a native application.

Adding a Native Application to Azure Active Directory

To test our Web API, we will need a client that will make requests to our endpoint; in this scenario, a WinForms application will work just fine. Let's add the native application inside our AlphaData AD.

The first thing to do is to log into the Azure management portal and then navigate all the way down to the **Application** tab of the AlphaData AD. After that, click **Add** in the bar at the bottom of the page, then select **Add an app my organization is developing**; after that you should see a dialog like the one shown in Figure 49:


ADD APPLICATION

Tell us about your application

NAME

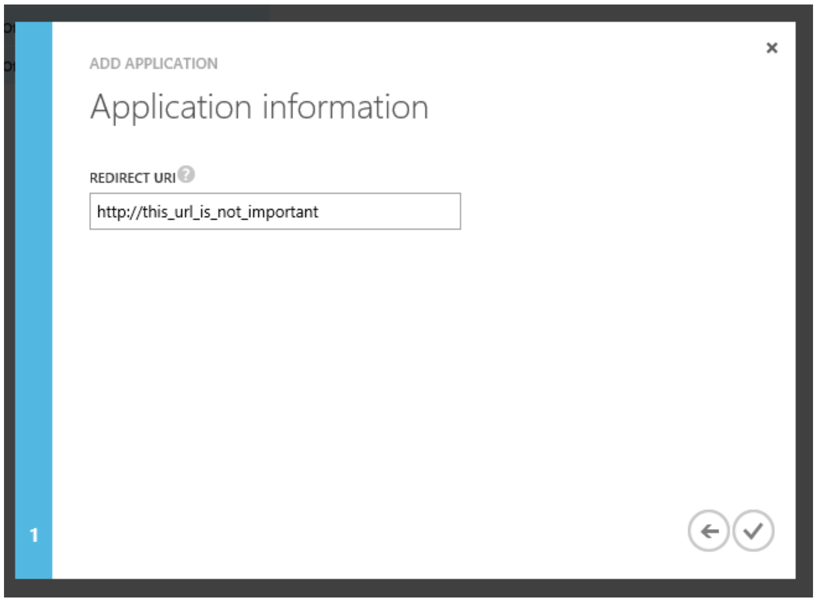
eBook.Native

Type

WEB APPLICATION AND/OR WEB API [✓ NATIVE CLIENT APPLICATION] PREVIEW

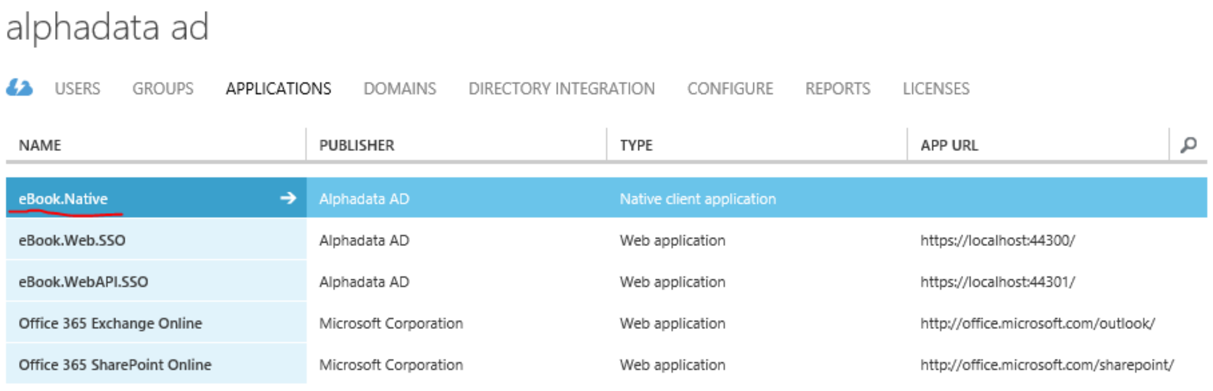
Figure 49: The name of our native app

Proceed to step two and fill the textbox with whatever URL you like; it does not matter if it is a fake URL. Since it is a native application, this URL will serve only as warning that will inform our native application to close the login dialog that appears (the dialog looks like an IE window). Click **Finish** and now we have a new entry in the application list.



http://this_url_is_not_important

Figure 50: A fake callback URL


alphadata ad

NAME	PUBLISHER	TYPE	APP URL
eBook.Native	Alphadata AD	Native client application	
eBook.Web.SSO	Alphadata AD	Web application	https://localhost:44300/
eBook.WebAPI.SSO	Alphadata AD	Web application	https://localhost:44301/
Office 365 Exchange Online	Microsoft Corporation	Web application	http://office.microsoft.com/outlook/
Office 365 SharePoint Online	Microsoft Corporation	Web application	http://office.microsoft.com/sharepoint/

Figure 51: Our native app is created

Exposing a Web API to Our Native Application

Unfortunately, our native application is not ready to communicate with our Web API because we haven't granted any permissions to do so. Your Web API is made available by configuring an application manifest, which is a JSON file that represents your application's identity configuration. You can expose your permission scopes by navigating to your application in the Azure Management Portal and clicking the **Application Manifest** button on the command bar. After downloading the manifest file, open it and replace the **appPermissions** node with the following JSON snippet. This snippet is just an example; make sure that you change the text and values for your own application. Once you have finished, upload the edited manifest to Azure.

Note: The "permissionId" value must be a new generated GUID that you create by using a GUID generation tool or programmatically. It represents a unique identifier for the permission that is exposed by the Web API. Once your client is appropriately configured to request access to your Web API and calls the Web API, it will present an OAuth 2.0 JWT token that has the scope claim set to the "claimValue" above, which in this case is "user_impersonation." You can expose additional permission scopes later as necessary.

```
"appPermissions": [

{

    "claimValue": "user_impersonation",

    "description": "Access company's Web API endpoints on user's behalf",

    "directAccessGrantTypes": [],

    "displayName": "Grant access to our Web API endpoint",

    "impersonationAccessGrantTypes": [

    {

        "impersonated": "User",

        "impersonator": "Application"

    }

    ],

    "isDisabled": false,

    "origin": "Application",

    "permissionId": "b69ee3c9-c40d-4f2a-ac80-961cd1534e40",

    "resourceScopeType": "Personal",
    "userConsentDescription": "Allow the application full access company's Web API",
    "userConsentDisplayName": "Grant access to our Web API endpoint"}]
```

Now our native application “sees” our Web API application and can add the required permissions.

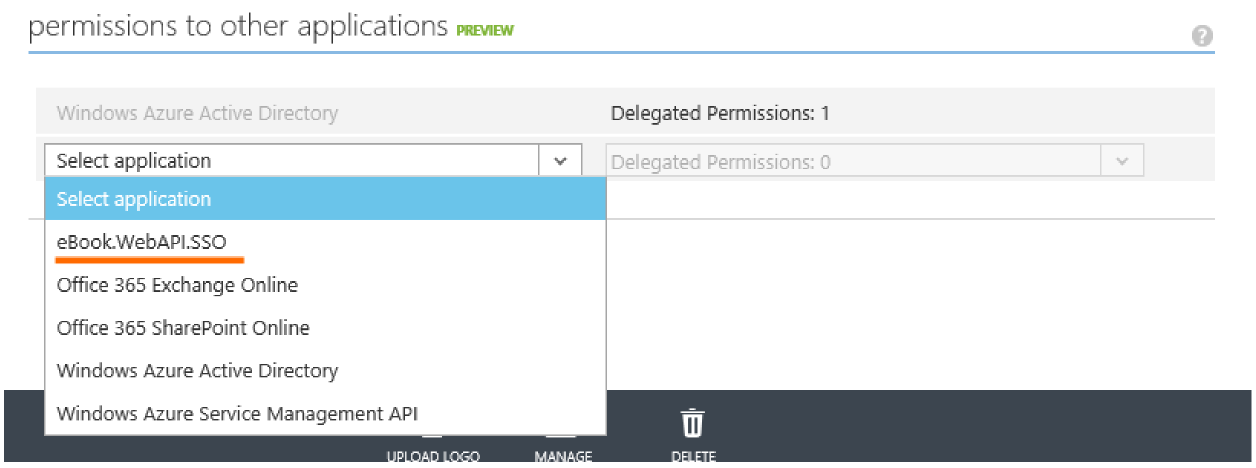

permissions to other applications PREVIEW
Windows Azure Active Directory
Delegated Permissions: 1
Select application
Delegated Permissions: 0
Select application
eBook.WebAPI.SSO
Office 365 Exchange Online
Office 365 SharePoint Online
Windows Azure Active Directory
Windows Azure Service Management API
UPLOAD LOGO
MANAGE
DELETE

Figure 52: Adding the permission to our native app.

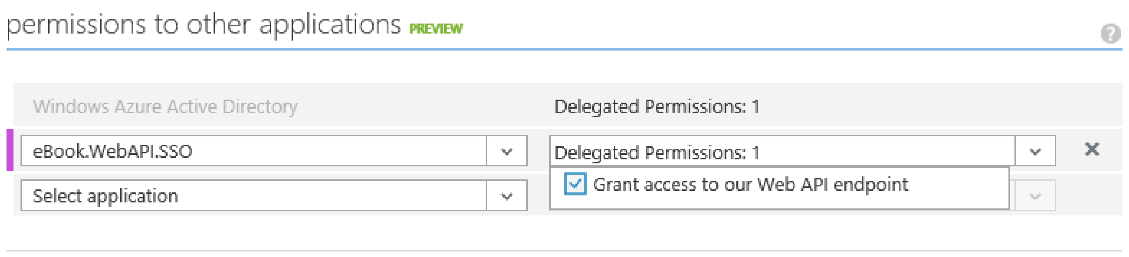

permissions to other applications PREVIEW
Windows Azure Active Directory
Delegated Permissions: 1
eBook.WebAPI.SSO
Delegated Permissions: 1
Grant access to our Web API endpoint
Select application

Figure 53: Permission added

We are now ready to see some code. In this next part of the book, we will not only see if we managed to secure our Web API, but we’ll also introduce the **Active Directory Authentication Library** (ADAL) that earlier queried our tenant inside the MVC application.

A Brief Introduction to ADAL (Active Directory Authentication Library)

Tip: The following code samples will require you to install the following NuGet packages in your Windows Desktop project: Microsoft HTTP Client Libraries and Active Directory Authentication Library (ADAL).

MSDN defines ADAL as follows:

“The Azure Active Directory Authentication Library (ADAL) for .NET enables client application developers to easily authenticate users to cloud or on premise Active Directory (AD), and then obtain access tokens for securing API calls. ADAL for .NET currently supports three authorities: Azure AD, Windows Server Active Directory Federation Services (ADFS) for Windows Server 2012 R2, and the Azure Access Control service (ACS). By handling most of the complexity, ADAL can help a developer focus on business logic in their application and easily secure resources without being an expert on security.”

ADAL for .NET offers the following features:

- **Token Acquisition**: ADAL for .NET facilitates the process of acquiring tokens from Azure AD, Windows Server ADFS for Windows Server 2012 R2, and the Azure ACS by using a variety of identity providers, protocols, and credential types. ADAL can manage the entire token acquisition process in just a few lines of code, including the authentication user experience. Alternatively, you can provide raw credentials that represent your user or application, and ADAL will manage obtaining a token for you.
- **Persistent Token Cache**: ADAL for .NET stores all access tokens in a persistent token cache by default, but you can also write your own cache implementation or disable it entirely.
- **Automatic Token Refresh**: In addition to the persistent token cache, ADAL supports automatic refresh of tokens when they expire. ADAL will both query the token cache to check if the token has expired, and then attempt to get a new token using the stored credentials.

Without knowing it, we have already used ADAL; it is installed as a NuGet package in every ASP.NET MVC/Web API project that uses organizational accounts. Listed below is the code that did the user magic back in the MVC app:

```
[Authorize]
public async Task<ActionResult> UserProfile()
{
    string tenantId =
ClaimsPrincipal.Current.FindFirst(TenantIdClaimType).Value;

    // Get a token for calling the Windows Azure Active Directory Graph
    AuthenticationContext authContext = new
AuthenticationContext(String.Format(CultureInfo.InvariantCulture, LoginUrl,
tenantId));
    ClientCredential credential = new ClientCredential(AppPrincipalId,
AppKey);
    AuthenticationResult assertionCredential =
authContext.AcquireToken(GraphUrl, credential);
    string authHeader = assertionCredential.CreateAuthorizationHeader();
    string requestUrl = String.Format(
        CultureInfo.InvariantCulture,
        GraphUserUrl,
        HttpUtility.UrlEncode(tenantId),
        HttpUtility.UrlEncode(User.Identity.Name));

    HttpClient client = new HttpClient();
    HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get,
requestUrl);
    request.Headers.TryAddWithoutValidation("Authorization", authHeader);
    HttpResponseMessage response = await client.SendAsync(request);
    string responseString = await response.Content.ReadAsStringAsync();
    UserProfile profile =
JsonConvert.DeserializeObject<UserProfile>(responseString);

    return View(profile);
}
```

What this code does is pretty straightforward to understand, but first let’s look at the list of all the variables declared on top of the file as follows (otherwise understanding this code would not be so straightforward).

```
private const string TenantIdClaimType =
"http://schemas.microsoft.com/identity/claims/tenantid";
private const string LoginUrl = "https://login.windows.net/{0}";
private const string GraphUrl = "https://graph.windows.net";
private const string GraphUserUrl =
"https://graph.windows.net/{0}/users/{1}?api-version=2013-04-05";
private static readonly string AppPrincipalId =
ConfigurationManager.AppSettings["ida:ClientID"];
private static readonly string AppKey =
ConfigurationManager.AppSettings["ida:Password"];
```

The first line of code gets our tenant ID, which consists of a unique identifier in our AD in the form of a GUID string. Then we build the **AuthenticationContext** that will contain the address of the resource that we want to use to authenticate. The **ClientCredential** object is then constructed; we will use it to give an “identity” to our application so that AD can recognize it. With the .AcquireToken() method we get an OAuth token for the Graph API, which we will use the graph to get the actual information about a user. The remaining lines of code are pretty self-explanatory: they create a string that represents the full address of the web resource we want to “query.” Of course, before making the actual request, we need to append the request object (an instance of the System.Net.HttpClient), the **Authorization** header, with the proper OAuth 2.0 authorization header. Finally, we execute the request and deserialize the JSON returned by Azure Graph.

Note: If you want to learn more about the Azure Graph API, please refer to this URL: http://msdn.microsoft.com/en-us/library/azure/hh974476.aspx

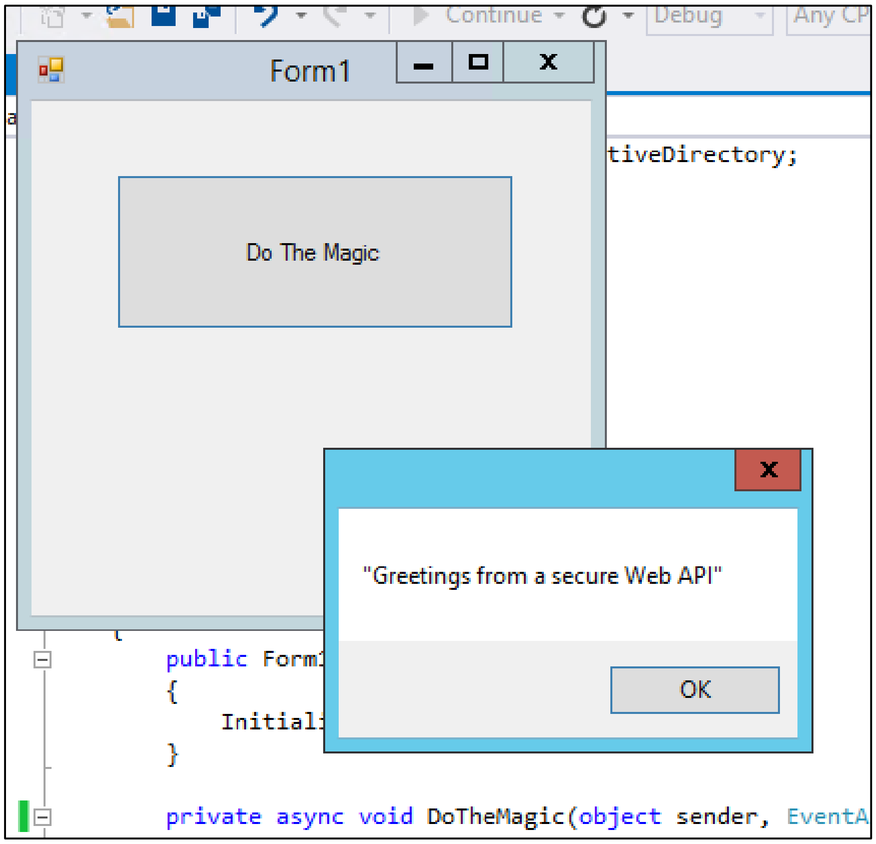
Now that we have discussed what happened earlier in our MVC app, let’s talk about our native application. Here is the code that will do all the magic for us.

```
private async void DoTheMagic(object sender, EventArgs e)
{
    // Get token
    AuthenticationContext ac = new
AuthenticationContext("https://login.windows.net/itadevconn.onmicrosoft.com
");
    AuthenticationResult ar  =
ac.AcquireToken("https://itadevconn.onmicrosoft.com/eBook.WebAPI.SSO",
                "5913d32b-1865-4d6f-8452-695dac3119d6",
                 new Uri("http://this_url_is_not_important"));

    // Call Web API
    string authHeader = ar.CreateAuthorizationHeader();
    HttpClient client = new HttpClient();
    HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get,
"https://localhost:44301/api/Values");
    request.Headers.TryAddWithoutValidation("Authorization", authHeader);
    HttpResponseMessage response = await client.SendAsync(request);
    string responseString = await response.Content.ReadAsStringAsync();
    MessageBox.Show(responseString);
}
```

The first line initializes a new **AuthenticationContext**. In the app’s code, an **AuthenticationContext** instance represents the authority with which to work. In this case, the authority is our Windows Azure AD tenant represented by the URI “**https://login.windows.net/[mydomain]**”.

The second line asks the **AuthenticationContext** for a token. There are many ways of crafting a token request—every scenario and application type calls for different parameters. In this case, we want to specify that the resource for which I want a token is my Web API. You then pass the identifier of its Windows Azure AD entry. This is the same value used as the audience in the Web API project. Then, because it is the native client requesting the token, we need to pass the client ID and redirect the URI of my client from the app configuration page I left open in the browser. That is all we need to do to obtain a token. The rest of the code puts that token in the correct HTTP header according to the OAuth 2.0 specification, and performs the actual call.

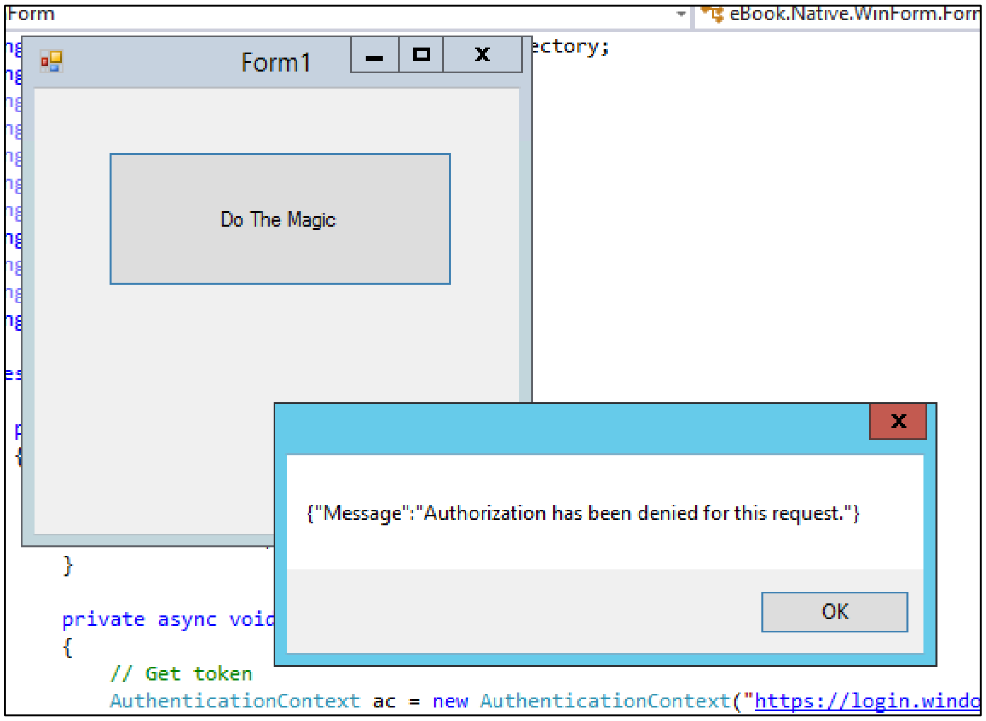
If we run this code and call the **DoTheMagic** method (in my case it is invoked when I click on a button), a dialog will appear asking us to log in with our organizational account. Once logged in, if you have set all the permissions correctly, this is what you should see. Figure 54 shows a successful request, Figure 55 shows a denied request.



"Greetings from a secure Web API"

```
tiveDirectory;
        public Form1
        {
            Initial
        }

        private async void DoTheMagic(object sender, EventA
```

Figure 54: An authorized request



```
{"Message":"Authorization has been denied for this request."}
```

```
ectory;
    }

    private async void
    {
        // Get token
        AuthenticationContext ac = new AuthenticationContext("https://login.windo
```

Figure 55: An unauthorized request

The unauthorized case can be tested by commenting the line where we add the "**Authorization**" header:

Conclusion

In this book, we have covered some basic and advanced aspects of developing directory-enabled applications by leveraging the ADSI, as well as some theoretical aspects of Active Directory and the LDAP protocol. We have also seen some code examples that clearly show how easy it is to build directory-enabled applications that are able to perform management and basic tasks with our on premise Active Directory. We talked about how to extend and expose our AD to the internet, and about giving the opportunity to our users to leverage the benefits of the SSO. We have seen how to configure an ASP.NET MVC 5 application to support organizational account and how to secure a Web API endpoint with our organizational credentials.

All of this has been possible thanks to Azure Active Directory and its syncing functionalities with on premise environments. In order to make a working bridge with our applications and the Azure AD, we have seen how, through Visual Studio, it is possible to add trusted applications to our Azure AD configuration. This enables applications to obtain a valid OAuth 2.0 token, making them able to query our tenant.