Microsoft

Inside OUT

The ultimate, in-depth reference

Hundreds of timesaving solutions

Supremely organized, packed with expert advice

Microsoft Exchange Server 2013: Mailbox and High Availability

Tony Redmond *Award-winning author + Microsoft MVP*

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

Library of Congress Control Number: 2013948703
ISBN: 978-0-7356-7858-3

Printed and bound in the United States of America.

First Printing

Acquisitions Editor: Anne Hamilton
Developmental Editor: Karen Szall
Project Editor: Karen Szall
Editorial Production: nSight, Inc.
Technical Reviewer: Paul Robichaux; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Copyeditor: Kerin Forsyth
Indexer: Lucie Haskins
Cover: Twist Creative • Seattle

Contents at a Glance

Table of Contents

Foreword for Exchange 2013 Inside Out books

Those seeking an in-depth tour of Exchange Server 2013 couldn't ask for better guides than Tony Redmond and Paul Robichaux. Tony and Paul have a relationship with the Exchange team that goes back two decades, to the days of Exchange 4.0. Few people have as much practical knowledge about Exchange, and even fewer have the teaching skills to match. You are in good hands.

Over the past few years, we have seen significant changes in the way people communicate; a growing number of devices, an explosion of information, increasingly complex compliance requirements, and a multigenerational workforce. This world of communication challenges has been accompanied by a shift toward cloud services. As we designed Exchange 2013, the Exchange team worked hard to build a product and service that address these challenges. As you read these books, you'll get an up-close look at the outcome of our efforts.

Microsoft Exchange Server 2013 Inside Out: Mailbox and High Availability covers foundational topics such as the Exchange Store, role-based access control (RBAC), our simplified approach to high availability, and the new public folder architecture. It also covers our investments in eDiscovery and in-place hold. As you read, you'll see how Exchange 2013 helps you achieve world-class reliability and provides a way to comply with internal and regulatory compliance requirements without the need for third-party products.

Microsoft Exchange Server 2013 Inside Out: Connectivity, Clients, and UM explores the technologies that give users anywhere access to their email, calendar, and contacts across multiple devices. It also explains how to protect your email environment from spam, viruses, and other threats and describes how Exchange 2013 can connect with Office 365 so you can take advantage of the power of the cloud.

From our new building-block architecture to data loss prevention, there's a lot to explore in the newest version of Exchange. I hope that as you deploy and use Exchange 2013, you'll agree that this is an exciting and innovative release.

Enjoy!

Rajesh Jha
Corporate Vice President - Exchange
Microsoft Corporation

Introduction

This book is for experienced Exchange administrators who want to get inside the soul of Exchange Server 2013, the latest version of the Microsoft enterprise messaging server first released in October 2012 and updated on a frequent basis since. You might learn how to work with Exchange 2013 by reading this book, but I sincerely doubt that this will happen simply because I have written it with experience in mind.

The book does not cover every possible topic relating to Exchange 2013. In fact, it focuses primarily on the Mailbox server role. Let me explain why. After completing *Microsoft Exchange Server 2010 Inside Out* (Microsoft Press, 2010), it became very clear that attempting to cover all of a complex product such as Exchange in any depth in just one book was a fool's errand. There are too many details to master, too much work to do, too much information that can only be skimmed over to keep to a reasonable page count. The result would probably be a book that weighs 2 kilos, spanning 1,400 pages that takes 2 years to write. All in all, an unacceptable situation in both commercial and practical terms.

Paul Robichaux and I ran a number of Exchange 2010 Maestro seminars in the 2010–2011 period. Despite the infamous cockroach sandwich affair, the events were good fun, and we enjoyed discussing the technology in some depth, even if we tended to ramble on at times. Brian Desmond, an Active Directory MVP who did an excellent job of lab master and stand-in speaker when required, helped us. Because we worked well together and because Paul has an excellent record of writing both books and articles, it seemed like a good idea to consider a joint approach for *Microsoft Exchange Server 2013 Inside Out*. We arrived at the basic idea quickly—we would split coverage into the two server roles. I'd write about the Mailbox role and Paul took on client access, including all the various clients Exchange supports, and unified messaging, which, strictly speaking, is part of an Exchange 2013 Mailbox server. However, Paul is an acknowledged expert in this space, and it would have made no sense to have me write about a subject of which Paul is the master.

Because Exchange 2013 is an evolution of Exchange 2010, we decided to use *Microsoft Exchange Server 2010 Inside Out* as the base for the new book. An evolution it might be, but an extensive level of change at the detail level exists in Exchange 2013. The upshot is that I'm not sure how much of that book remains in the current text—maybe 20 percent. One thing I am glad of is that we did not rush to press after Exchange 2013 first appeared. Given the amount of change that has occurred in updates from Microsoft since, a book that describes the release to manufacturing (RTM) version of Exchange 2013 would have been obsolete very soon after publication. We hope that these volumes will last longer.

I hope that you enjoy this book and that you'll read it alongside Paul's *Microsoft Exchange Server 2013 Inside Out: Clients, Connectivity, and UM*. The two books really do go together. Paul has scrutinized every word in this book and I have done the same for his. We therefore share the blame for any error you might find.

Acknowledgments

I owe enormous thanks to the many people who agreed to look over chapters or portions of the book. Each has deep expertise in specific areas and all contributed greatly to eradicating errors and increasing clarity. These folks include Sanjay Ramaswamy, Jürgen Hasslauer, David Espinoza, William Rall, Todd Luttinen, Tim McMichael, Vineetha Kalvakunta, Fred Monteiro da Cruz Filho, Kanika Ramji, Lokesh Bhoobalan, Astrid McClean, Alfons Staerk, Kern Hardman, Andrew Friedman, Abram Jackson, and Scott Schnoll. Even if they didn't realize it, many of the Exchange MVPs played their part in improving the book by prompting me to look into topics that I had forgotten to cover. I should also acknowledge the huge contribution made by my editor, Karen Szall. We fought many times about page counts, content, and too many other topics to list here but always kept the project moving.

I apologize sincerely if I have omitted to mention anyone who has contributed to making the text of the book as accurate and as informative as possible.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://aka.ms/ExIOv1/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

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

We want to hear from you

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

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

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

Stay in touch

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

CHAPTER 3

The Exchange Management Shell

Windows PowerShell is an extensible automation engine consisting of a command-line shell and a scripting language. Exchange Server 2007 was the first major Microsoft application to support Windows PowerShell in a comprehensive manner. Although not every administrator welcomed the opportunity to learn a new scripting language, the overall impact was extremely positive. The role of Windows PowerShell continues to expand across Microsoft products, and it now extends into the newest Microsoft offerings, including the deployment and management of applications on the Azure cloud computing platform.

Windows PowerShell is built on top of the Microsoft .NET Framework and is implemented in the form of cmdlets, specialized .NET classes that contain the code to implement a particular operation such as the creation of a new mailbox or the enumeration of the processes that are currently active on a server. Applications implement Windows PowerShell support by providing sets of application-specific cmdlets that collectively represent the functionality required to support the application, or they can be used to access different data stores such as the file system or system registry. Cmdlets can be run separately or combined by piping the output generated by one cmdlet to become the input of the next. Cmdlets can also be combined into scripts (with a .ps1 file extension) to provide more comprehensive processing and logic or included in executables when the need exists to launch a standalone application. Many scripts are available on different Internet sites to assist with Exchange management.

How Exchange uses Windows PowerShell

From an Exchange perspective, Windows PowerShell provides a way to perform tasks quickly and simply in a variety of manners, from one-off interventions to process one or more Exchange objects to complex scripts to perform tasks such as mailbox provisioning. Most administrators cut their teeth on PowerShell by using the Exchange Management Shell (EMS) to do simple things, such as using Get-Mailbox to report on a mailbox's properties and Set-Mailbox or Set-CASMailbox to set a property, before moving on to the more esoteric commands to manipulate connectors or control the ability of devices to connect through ActiveSync and so on. The saying is that almost anything is possible with Windows PowerShell, and this is certainly true when you dedicate enough energy and time to mastering the language, not to mention the time necessary to scan the Internet for useful examples of scripts that can be adapted to meet your needs.

Prior to Exchange Server 2007, business logic was scattered in components throughout the product. The management console did things—even simple things like setting a property on a server—by using different code and logic than in the setup program, and the application programming interfaces (APIs) included in the product usually provided a third way to approach a problem. The result was a total lack of consistency, duplication of code, and a tremendous opportunity to create bugs in multiple places. In addition, administrators could not automate common tasks to meet the needs of their organization; essentially, if an Exchange engineer didn't code something into the product, it couldn't be done.

Figure 3-1 illustrates the central role Windows PowerShell now plays in the Exchange architecture and shows how it provides a central place to encapsulate business logic that underpins the Exchange setup program, the Exchange Administration Center (EAC), the mailbox options that users can update through Outlook Web App, and the Exchange Management Shell (EMS).

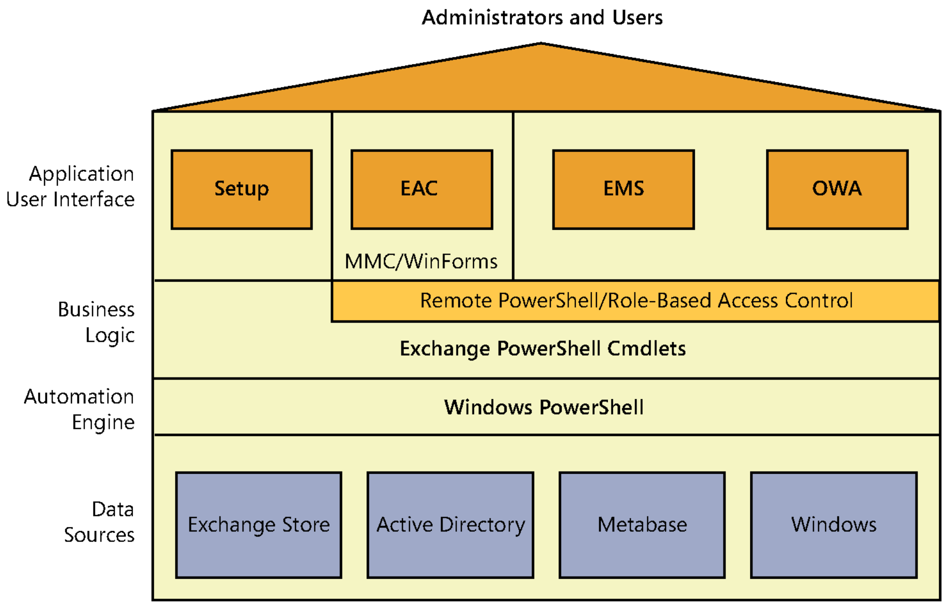


Figure 3-1 Windows PowerShell at the heart of Exchange

The way Exchange uses Windows PowerShell to implement business functionality is probably the most extensive of any Microsoft application. As explored throughout this book, the options presented by EAC to work with mailboxes, connectors, servers, and other objects invariably result in a call to one or more PowerShell cmdlets that actually do the work. The functionality presented to administrators, specialist users (those who perform a subset of administrative tasks such as maintaining user details), and normal users is all based on PowerShell.

The exact scope and range of the functionality presented to any individual user is determined by the permissions granted to him through role-based access control (RBAC). RBAC is designed to function across a range of environments, from a single-server organization to an organization composed of a mixture of on-premises and hosted servers. The need to accommodate such a wide range of environments is also why Microsoft has moved from local PowerShell (by which all commands are executed on a local server) to remote PowerShell (by which commands are redirected through Internet Information Services [IIS] for execution on a target server). The details of just how remote PowerShell and RBAC work together in EMS are covered shortly.

Simplifying the implementation of new functionality

The administrative interfaces in Exchange all lead to the same place and execute the same business logic. Apart from removing redundant and overlapping code, having a single place to implement business logic enables the Exchange engineers to concentrate on implementing new functionality rather than re-implementing features specifically for use by EAC, EMS, or the setup program. The approach enables Exchange to deliver a more consistent administrative environment and a comprehensive method to automate tasks to deal with mailboxes, databases, connectors, and all the other components that collectively make up an Exchange organization.

At the time of writing, Exchange 2013 RTM CU2 includes 965 cmdlets that are added to the standard set of Windows PowerShell cmdlets, including cmdlets to work with the system registry, file system, variables (including environmental variables), and so on that are available in an EMS session. Depending on the RBAC role groups of which your account is a member, the number of cmdlets available to you might vary.

Collectively, the set of EMS cmdlets manages the objects and the properties of the objects that form Exchange. Objects include mailboxes, servers, transport rules, connectors, and so on. You can determine the exact number of cmdlets Exchange owns by using the following command (this command doesn't work with Exchange Online):

```
Get-ExCommand | Measure-Object | Select Count
```

INSIDE OUT Finding the cmdlets available to you

As you learn in the discussion about RBAC in Chapter 4, "Role-based access control," an EMS session allows you access only to the cmdlets and parameters that are defined in the roles included in the role groups of which your account is a member. Accounts that are highly permissioned, such as those belonging to the Organization Management role group, can use many more cmdlets than those that belong to a less-permissioned role group, such as Help Desk or Recipient Management. You can use this command to generate a full list of all the Exchange 2013 cmdlets your account can access:

```
Get-ExCommand > C:\Temp\ExCommands.txt
```

By comparison, Exchange 2007 includes 394 cmdlets, Exchange 2010, 584; and the RTM version of Exchange 2013, 958. The hundreds of new cmdlets included in Exchange 2013 and subsequently augmented through cumulative updates reflect the new functionality in the product such as the introduction of site mailboxes and data loss protection policies, along with the expansion of existing functionality such as the changes to compliance.

PowerShell use and syntax are fundamental skills for Exchange administrators to master. In fact, many Exchange administrators prefer EMS to EAC because of the additional flexibility that EMS provides. This chapter lays out the basics of Windows PowerShell and sets the stage for the examples of PowerShell found in other chapters. To begin, review how the Exchange management tools actually connect to PowerShell.

Using remote Windows PowerShell

Exchange 2010 began the necessary transformation from a model that assumed an administrator would always have some form of physical access to a server to the point at which remote management has become the norm. Remote PowerShell provides the fundamental building block for connectivity to remote systems. The combination of remote PowerShell and RBAC enables administrators to manage objects residing on a server in a remote datacenter as easily as managing objects on a local server.

Note

You can think of Windows PowerShell as implemented in Exchange Server 2007 as "local PowerShell" because cmdlets are executed in a local process. The only element of remote access in Exchange 2007 is when you pass the –Server parameter to identify a server against which to execute a command. Even so, if data are needed from a remote server, such as fetching a set of mailbox objects, it is retrieved across the network and processed locally.

Exchange 2010 and Exchange 2013 support the remote execution of commands in a secure manner, using HTTPS and a Kerberos-based encryption mechanism that is easily manageable through firewalls (assuming that port 80 is open). Remote PowerShell is now used for all EMS sessions. Even if you are logged on to an Exchange server and want to use EMS to change a property of that server, EMS still creates a remote session on the local server to do the work. The same applies for EAC because Exchange creates a remote session when you log on to connect to a server in the local Active Directory site to retrieve information about the organization and then display it in the console. In effect, remote PowerShell has replaced local PowerShell for all server roles except edge servers. The sole exception is for commands used during setup, which continue to execute locally. Remote PowerShell separates business logic into code that runs on the client and code that runs on the Exchange server. The logic for replacing local PowerShell with the remote model is simple. Just as the change in Exchange 2007 forced all messages to flow through the transport system so that a common place existed to apply features such as transport rules, remote PowerShell forces all aspects of Exchange administration to flow through RBAC so that tight control can be achieved over the actions an administrator—or, indeed, a user—is allowed to perform.

An RBAC role group defines the set of administrative actions a user is allowed to perform inside Exchange and can be resolved into a set of PowerShell cmdlets the user is allowed to use within her PowerShell session. After it is created, a PowerShell session is populated with cmdlets by reference to the RBAC role groups of which the user is a member so that it will only ever include the cmdlets that have been assigned to an administrator through her membership in role groups. Thus, the fact that an administrator is not a member of a particular role group is reflected in that any PowerShell session she initiates will never be able to call the cmdlets associated with the management group. Consider the case of an administrator who is a member of the Organization Management role group, typically believed to be the all-powerful role for Exchange management. Even though membership in the Organization Management role group grants a user access to the vast majority of Exchange cmdlets, out of the box, it does not grant access to the set that controls movement of data into and out of mailboxes. To protect user data, the Mailbox Import-Export role has to be explicitly assigned to any user who wants to perform these tasks, even those who are already members of the Organization Management role group.

The need to support massively scalable, multitenant platforms such as Office 365 was a major influence on the Exchange move to remote PowerShell. Providing a secure and controllable mechanism to permit administrators to execute privileged commands to control the subset of objects they own inside an infrastructure that is controlled and managed by someone else is always a difficult task, especially when all the data have to pass across the Internet. When you use PowerShell to create a session with Exchange Online, the session is by default remote because you cannot connect to the servers running in Microsoft datacenters in any other way. After it accepts the credentials you provide in the connection request, PowerShell loads in the cmdlets you are allowed to use through membership in

RBAC role groups. The fact that Exchange Online restricts the number of cmdlets available to administrators when compared to on-premises deployments is immaterial. Exactly the same mechanism populates your PowerShell session with cmdlets whether you connect to an on-premises deployment or to a cloud-based service.

CAUTION!

Until Microsoft removes the functionality, it is possible to use local PowerShell with the Exchange 2013 snap-in to perform management operations on a server. However, Microsoft is not testing local PowerShell with Exchange anymore, and it is possible that problems will appear in local PowerShell that will never be resolved. In addition, running local PowerShell means that you bypass the controls otherwise imposed by RBAC. Given the engineering and strategic focus on remote PowerShell, it makes sense for everyone to make the transition now and embrace this platform as the future of command-line Exchange management.

Connecting to remote PowerShell

To understand how remote PowerShell and RBAC work together, examine how an administrator might create a new mailbox on a remote server. In this example, the administrator works on a help desk and has been assigned a role that enables him to create new mailboxes and update the properties of existing mailboxes. Further, assume that the user's account is enabled to use remote PowerShell. In many cases, people in specialist roles such as help desk personnel use EAC to perform tasks, but an experienced Exchange administrator might prefer to use a command-line interface because of its power and flexibility, especially when processing multiple objects, when compared to EAC.

Figure 3-2 lays out the various components remote PowerShell uses from the local PowerShell host on a workstation or server across the network to IIS and the PowerShell application running there. The other components are the PowerShell engine and the complete set of cmdlets available to Exchange 2013, the Exchange authorization library that handles the interpretation of roles in terms of the cmdlets that each RBAC role can use, and the Active Directory driver that reads data from Active Directory. For the purpose of this discussion, assume that the account has been assigned a role such as Recipient Management and is enabled for remote PowerShell. If you are unsure about the account's status, you can enable it to use remote PowerShell as follows:

```
Set-User –Identity AccountName –RemotePowerShellEnabled $True
```

All PowerShell sessions flow through IIS because even a local connection goes through localhost. All Exchange 2013 servers support IIS and the PowerShell virtual directory, or vdir; all are members of the Exchange Trusted Subsystem security group and therefore can manipulate any object in the organization.

If you run EMS on a workstation or server on which the Exchange management components are installed, EMS creates a remote session automatically as part of its initialization process. If you run PowerShell on a workstation that doesn't have the Exchange management components installed, you must specify the name of the server with which you want to work. This is done by using the New-PSSession cmdlet, passing the name of the server to which to connect in the *https://fqdn/PowerShell/* form. This cmdlet creates a secure, authenticated connection to IIS running on the target server and begins a session there by checking the role held by the account that initiates the connection.

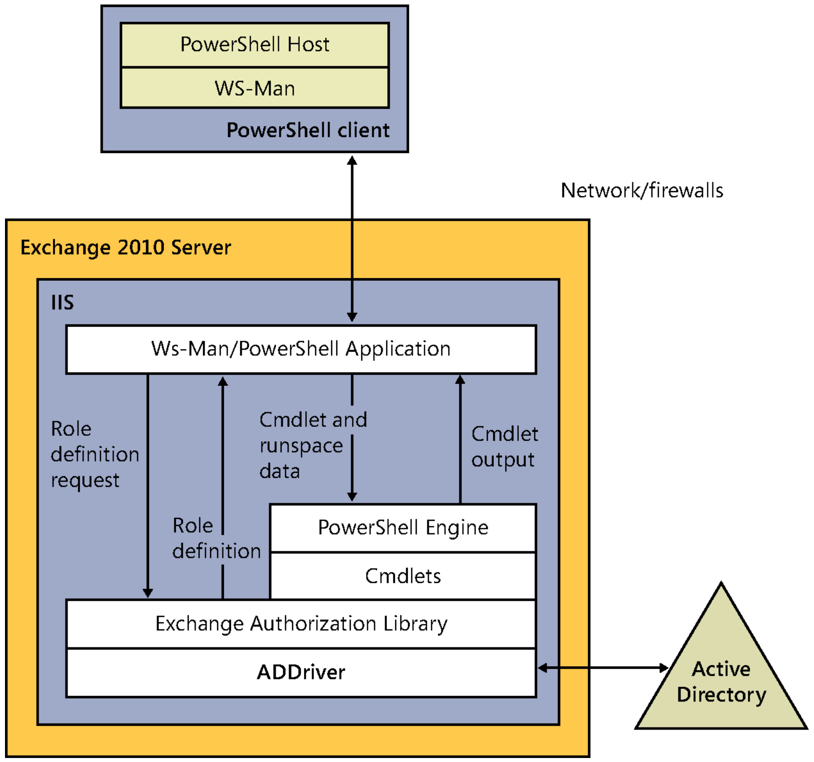


Figure 3-2 Remote PowerShell architecture

When you run EMS on a server on which the Exchange 2013 management components are installed, the EMS initialization script creates an environment by executing the code in the RemoteExchange.ps1 script (in the Exchange binaries folder), which first attempts to create a remote session with the local host. If successful, it then identifies your account to Exchange, uses RBAC to determine the cmdlet set you are allowed to use, and so on.

The RemoteExchange.ps1 script is quite complex because it has to handle the initiation of sessions in different circumstances, so it's easier to look at the code that connects a new session with Exchange Online to see how the basic requirements to create a remote PowerShell session are met. The following code defines the Connect-ExchangeOnline function, which is suitable for inclusion in your PowerShell profile (a document that initializes a session with common commands and settings that you might want to use during the session). In fact, this is how I connect to Exchange Online when I want to manage my tenant domain. Three commands are executed:

- Get-Credential gathers the name of the account with which to connect and its password.
- New-PSSession establishes the remote PowerShell session with Exchange Online.
- Import-PSSession imports the Exchange commands from the Exchange Online session into your current session.

```
function Connect-ExchangeOnline
{
  $LiveCred = Get-Credential
  $global:Session365 = New-PSSession -ConfigurationName Microsoft.Exchange
-ConnectionUri https://ps.outlook.com/powershell/ -Credential $LiveCred
-Authentication Basic -AllowRedirection
   Import-PSSession $global:Session365
}
```

How the initialization script finds a server

When a user creates a remote PowerShell session on an Exchange server, the initialization script attempts to connect him to the same server. If the attempt to establish a connection with the local server fails, the initialization script then enumerates the full set of Exchange servers in the local site and attempts to make a connection to one of the servers chosen at random. If this attempt fails, the script moves on to the next server and continues until a successful connection is established or all available servers have been attempted and have failed. The initialization script works in the same way when executed on a workstation that has the Exchange management components installed on it, except that the initialization begins by randomly selecting one of the servers from the site.

IIS uses the RBAC mechanism to check the user's role and associated permissions through the Exchange Authorization Library. The Exchange Authorization Library (or ADDriver) connects to Active Directory to use it as the definitive source of information about accounts and supplements these data with its knowledge about the Exchange-specific roles that administrators have assigned to users. During a PowerShell session, ADDriver connects to a domain controller in the local site to fetch data from Active Directory and keeps this connection throughout the session (referred to as DC affinity). Many PowerShell cmdlets support the DomainController parameter to enable you to connect to a specific domain controller (specifying the fully qualified domain name [FQDN]) if the need arises.

Because this user works with mailboxes as defined by the Recipient Management role group, the set of cmdlets he can use includes commands with easily identified purposes such as New-Mailbox, Set-Mailbox, Get-Mailbox, and so on.

Tip

Permissions granted through RBAC are evaluated during session initialization. If you are assigned a new role, you have to create a new session with EMS or EAC before you can access the cmdlets made available through the newly assigned role.

Users are not normally aware that they are restricted in terms of available cmdlets unless they attempt to use one to which they do not have access. The point is that they shouldn't care that they can't use hundreds of cmdlets, many of which do obscure things such as setting properties on messaging connectors or performing one-off operations such as creating a new Database Availability Group (DAG) or mailbox database. Instead, RBAC makes sure that users can access only the cmdlets they need to perform their jobs.

Limiting user functionality

When a new PowerShell session is created, you see no evidence that your role has forced RBAC to restrict the cmdlet set or the parameters you can use with cmdlets because the initialization of a session progresses just as it would for a fully privileged user. However, after you start to execute cmdlets, you quickly realize that you can't do as much as you'd like. For instance, if you log on with a restricted user account and attempt to use the Get-Mailbox cmdlet to fetch a list of mailboxes, all you'll see is your own mailbox. This is logical because your role allows you to see details of your own mailbox but not others'. In the same way, if you then attempt to use the Set-Mailbox cmdlet to update a property that only administrators can access, you won't be able to use even tab completion to reveal a restricted property. However, unless the default role assignment policy has been amended to block access to basic user options, you can use the Set-Mailbox cmdlet to update properties that are generally exposed for user update through Outlook Web App options, so (assuming JSmith is the alias for your mailbox) you'll be able to do things such as this:

```
Set-Mailbox -Identity JSmith –MailTip 'Hello World'
```

or this:

```
Set-Mailbox –Identity JSmith –Languages 'EN-US', 'EN-IE'
```

INSIDE OUT **You can do some things; you can’t do others**

Somewhat strangely, you’ll also be able to execute Get-MailboxStatistics to report the number of items in your mailbox but not Get-MailboxFolderStatistics to report on the folders and the items that each contains. This is all controlled by RBAC, the roles that your account holds, and the scope for the roles in terms of the cmdlets and parameters defined in each role. From this discussion, you should now understand how critical RBAC is to remote PowerShell and, by extension, to every aspect of the Exchange 2013 management toolset.

EMS basics

Exchange 2013 RTM CU2 includes 965 cmdlets, but you’re not likely to use the vast majority of these simply because many are designed for one-time use. For example, after you configure a receive connector, you probably will not revisit the Set-ReceiveConnector cmdlet very often after the connector is working. However, you’ll use cmdlets such as Get-Mailbox daily. Some examples (in no particular order) of frequently used Exchange cmdlets are the following:

- **Get-ExchangeServer** Return a list of Exchange servers in the organization.
- **Disable-Mailbox** Disable a user’s mailbox.
- **Add-DistributionGroupMember** Add a new member to a distribution group.
- **Set-Mailbox** Set a property of a user’s mailbox.
- **Get-MailboxDatabase** Retrieve properties of a mailbox database.
- **Get-MailboxStatistics** Return statistics about user mailboxes such as the total item count, quota used, and so on.

Note the consistent syntax of verb (Get, Set, Move, Remove, or Disable) and noun (Mailbox, User, and so on). Along with commands that operate on objects, you find commands that help you work with data, such as Where-Object, Sort-Object, and Group-Object. Where-Object, Sort-Object, and Group-Object are commonly shortened by using their aliases of

Where, Sort, and Group. You can type **Help** followed by a cmdlet name at any time to get help on the syntax of the command.

Tip

When you start to write scripts, consider spelling out cmdlet names completely and avoiding the use of aliases. This is important because you can never know in what environment a script will be run and therefore cannot assume that an alias will be defined and available for use in your code.

The Exchange developers have provided very accessible help for the EMS cmdlets. Apart from using the Help cmdlet, there are other ways of seeking help. RBAC controls limit help content so that a user sees help only for the set of cmdlets available to the roles that user holds. You can do the following:

- Use the Get-Command cmdlet to list the cmdlets you can use with different objects. The set of cmdlets will be limited to whatever is permitted by the RBAC roles held by your account. For example, Get-Command *contact* lists all the cmdlets available to work with contacts (shown in the following example). You can also use the shortened alias of gcm for Get-Command.

```
CommandType          Name                        Definition
-----------          ----                        ----------
Function             Disable-MailContact         ...
Function             Enable-MailContact          ...
Function             Get-Contact                 ...
Function             Get-MailContact             ...
Function             New-MailContact             ...
Function             Remove-MailContact          ...
Function             Set-Contact                 ...
Function             Set-MailContact             ...
```

- Use the /detailed switch to get more detailed help about a cmdlet. For example: Get-Help Get-CASMailbox –Detailed.
- Use the /full switch to have EMS return every bit of information it knows about a cmdlet. For example, Get-Help Get-DistributionGroup –Full.
- Use the /examples switch to see whatever examples of a cmdlet in use EMS help includes. For example, Get-Help Get-MailboxServer –Examples.

- Use the /parameter switch to get information about a selected parameter for a cmdlet. For example, Get-Help Get-Mailbox –Parameter Server. This switch supports wildcards, so you can do something like Get-Help Set-Mailbox –Parameter *Quota*.

INSIDE OUT Getting to know the cmdlets

You will probably begin by using the /full switch to retrieve all available help for a cmdlet to get to know what each cmdlet does. After you learn more about the cmdlet, you can move on to the default view as you become more accustomed to working with EMS. Remember that the Exchange help file contains information about all the EMS cmdlets. The advantage of using the help file (which is always present on a server) is that you can use the help file's index to search for specific entries.

Most of the time, you will probably work with commands by invoking EMS interactively and then typing whatever individual commands or scripts are necessary to perform a task. The user interface of EMS is based on the Win32 console with the addition of features such as customizable tab completion for commands. After you become accustomed to working with EMS, things flow smoothly, and work is easy. It is then usually faster to start EMS and issue the necessary code to change a property on a mailbox or a server than to start EAC and navigate to the right place to make the change through the graphical user interface (GUI).

Tip

Working through EMS is especially valuable if you have to perform management operations across an extended network link when waiting for the GUI to display can be painful. If you have a programmatic mind, you can also call EMS cmdlets through C# code, which is how Microsoft invokes them in EAC and other places throughout Exchange, such as to set up servers and databases in the setup program. (The blog Glen Scales writes at *http://gsexdev.blogspot.com/* provides many good examples of how to call EMS cmdlets from code.) In the past, the different groups that contributed to Exchange had to build their own programming interfaces, whereas now everyone uses PowerShell.

You can see that EMS focuses on performing tasks rather than taking the more object-focused approach implemented in the GUI, something that reflects a desire to accommodate administrators who think about how to do things rather than how to work with objects. After all, it is human nature to think in terms of the task of moving a mailbox to a different server rather than thinking about how to manipulate the properties of a mailbox object to reflect its new location.

Cmdlets accept structured pipelined input from one another in a common manner to allow them to process data in a consistent manner, no matter which cmdlet provides the data. Programmers therefore do not have to reformat data for input to specific cmdlets, so the task of assembling different cmdlets into a script to do a job is much easier. Microsoft built PowerShell around the concept of objects, so objects are accepted as input, and the output is in the form of objects that you can then pipe to other cmdlets. Even if the output from a cmdlet looks like plaintext, what you see is one or more objects that you can manipulate in a much more powerful manner than you can ever work with text output. The implementation is elegant.

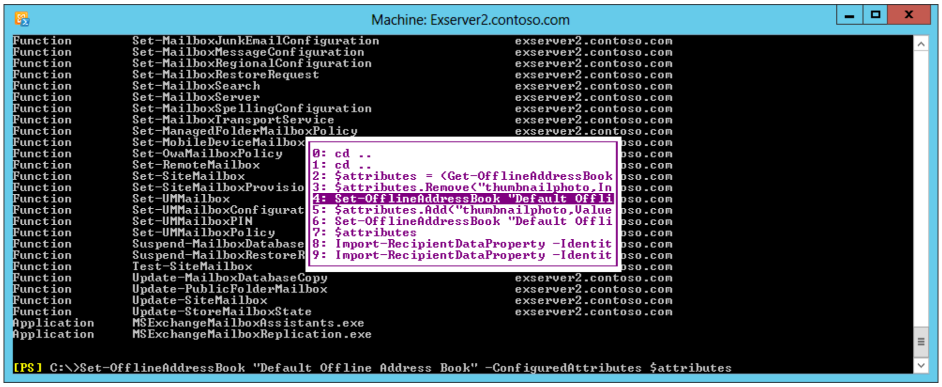
Command editing

It should be apparent that you could do a lot of typing to enter commands into PowerShell, make the inevitable mistakes, correct them, and try again. To make the task a little easier, PowerShell supports the same kind of command-line editing as the Win32 console (CMD) does. Some of the more important keys you can use are described in Table 3-1.

TABLE 3-1 Command editing keystrokes for PowerShell

Keyboard command	Effect
F2	Creates a new command based on your last command. A pop-up screen appears in which to enter a character. PowerShell then creates a new command, using the last entered command up to the character you specify. For example, if the last command is Get-MailboxStatistics –Identity TRedmond, and you enter F2 followed by c, PowerShell inserts "Get-MailboxStatistics". You can then complete the command as you like.
F4	Deletes characters in the current command up to a specified position. For example, if the cursor is located at the "M" of Get-MailboxStatistics, and you enter F4 followed by x, PowerShell deletes "Mailbo" and the result is "Get-xStatistics". Although this example wouldn't result in a useful command, F4 is useful when you need to edit many parameters in a complex command.
F7	Opens a list of the last 50 commands used in the current session to enable you to select a command for reuse.
F8	Moves backward through the command history.
Tab	Requests PowerShell to complete a command based on what you've typed.
Left/Right arrows	Moves the cursor left and right through the current command line.
Up/Down arrows	Moves up and down through the history of previous commands.
Delete	Deletes the character under the cursor.
Insert	Toggles between character insert and character overwrite mode.
Backspace	Deletes the character before the cursor.

Most of these keys are straightforward. The two most interesting keys are F7 and Tab. F7 opens a list of the last 50 commands you have run in the current session (Figure 3-3) so that you can both see what you've done in the immediate past and select one of the commands to re-execute. You can type a couple of characters into the F7 list, and EMS will look for the first matching command, or you can use the Up and Down arrows to navigate through the command history. At times, it's more convenient to use Up and Down arrows because you can retrieve more commands and edit a command before executing it. (F7 selects the command and executes it immediately.)



```
Function        Set-MailboxJunkEmailConfiguration                  exserver2.contoso.com
Function        Set-MailboxMessageConfiguration                    exserver2.contoso.com
Function        Set-MailboxRegionalConfiguration                   exserver2.contoso.com
Function        Set-MailboxRestoreRequest                          exserver2.contoso.com
Function        Set-MailboxSearch                                  exserver2.contoso.com
Function        Set-MailboxServer                                  exserver2.contoso.com
Function        Set-MailboxSpellingConfiguration                   exserver2.contoso.com
Function        Set-MailboxTransportService                        exserver2.contoso.com
Function        Set-ManagedFolderMailboxPolicy                     exserver2.contoso.com
Function        Set-MobileDeviceMailbox                                          oso.com
Function        Set-OwaMailboxPolicy    0: cd ..                                 oso.com
Function        Set-RemoteMailbox       1: cd ..                                 oso.com
Function        Set-SiteMailbox         2: $attributes = (Get-OfflineAddressBook oso.com
Function        Set-SiteMailboxProvisio 3: $attributes.Remove("thumbnailphoto,In oso.com
Function        Set-UMMailbox           4: Set-OfflineAddressBook "Default Offli oso.com
Function        Set-UMMailboxConfigurat 5: $attributes.Add("thumbnailphoto,Value oso.com
Function        Set-UMMailboxPIN        6: Set-OfflineAddressBook "Default Offli oso.com
Function        Set-UMMailboxPolicy     7: $attributes                           oso.com
Function        Suspend-MailboxDatabase 8: Import-RecipientDataProperty -Identit oso.com
Function        Suspend-MailboxRestoreR 9: Import-RecipientDataProperty -Identit oso.com
Function        Test-SiteMailbox                                                 oso.com
Function        Update-MailboxDatabaseCopy                         exserver2.contoso.com
Function        Update-PublicFolderMailbox                         exserver2.contoso.com
Function        Update-SiteMailbox                                 exserver2.contoso.com
Function        Update-StoreMailboxState                           exserver2.contoso.com
Application     MSExchangeMailboxAssistants.exe
Application     MSExchangeMailboxReplication.exe

[PS] C:\>Set-OfflineAddressBook "Default Offline Address Book" -ConfiguredAttributes $attributes
```

Figure 3-3 Using F7 to recall EMS commands

INSIDE OUT An easy way to type a command

Tab completion is a wonderful feature that Windows PowerShell inherited from CMD. You can partially enter a command and then press Tab to have PowerShell fill in the rest of the cmdlet name followed by its parameters. For example, type:

```
Get-Dist
```

This isn't the name of a valid cmdlet, but it is the root of several cmdlets, so when you press Tab, PowerShell completes the first valid cmdlet that matches and inserts:

```
Get-DistributionGroup
```

If you press Tab again, PowerShell moves to the next cmdlet that matches and inserts:

```
Get-DistributionGroupMember
```

If you press Tab again, PowerShell returns to Get-DistributionGroup because there are only two valid matches. PowerShell also supports completion for parameters. If you insert a dash to indicate a parameter value after Get-DistributionGroup and press Tab,

PowerShell starts with the first parameter and continues through all valid parameters. If you press Tab too many times and pass the parameter you want to use, you can press Shift+Tab to go back through the parameter list. If you add some characters to help PowerShell identify the parameter, it attempts to complete using that value. For example:

```
PowerShell completes Get-DistributionGroup –Ma into the command
Get-DistributionGroup –ManagedBy.
```

Even better, tab completion is context-sensitive, so it understands the structure of the object you are navigating. For example, if you want to move through the system registry, tab completion understands the hive structure, so you can type a location in the registry and then use the Tab key to move through the available choices from that point. For example, type:

```
CD HKLM:\Software\Microsoft\Exchange
```

Now press Tab, and PowerShell leads you through all the registry locations Exchange uses.

Windows PowerShell supports both named and positional parameters. Identifiers are a good example of a positional parameter. For example, if you enter Get-Mailbox Tony, PowerShell assumes that *Tony* is the value for the –Identity parameter.

Finally, PowerShell completes variables and even the properties of variables (such as their length) in a way similar to how the Microsoft Visual Studio IntelliSense feature works. If you type the incomplete name of a variable and press Tab, PowerShell completes it from the list of known variables. For example, if you fill a variable with details of a mailbox as in the following:

```
$Mailbox = Get-Mailbox –Identity Redmond
```

and then type **$Ma** and press Tab, PowerShell completes it and returns $Mailbox. This is a useful feature if you forget the names of variables you've defined. To see how properties are completed, type:

```
$Mailbox.Di
```

Pressing Tab now will request PowerShell to go through the list of properties beginning with Di. For a mailbox, the list is DistinguishedName and DisplayName.

Handling information EMS returns

Any cmdlet such as Get-EventLog that retrieves some information about an object will output a default set of properties about the object (or references to an object). Sometimes those properties are not exactly the ones you want to examine, so you will inevitably use the Format-List and Format-Table cmdlets to expand the set of properties a command returns. For example, if you use the Get-Mailbox cmdlet to view the properties of a mailbox, the information returned isn't interesting:

```
Get-Mailbox –Identity TRedmond
```

```
Name                 Alias              ServerName           ProhibitSendQuota
----                -----              ----------           -----------------
Tony Redmond        TRedmond           ExServer1            unlimited
```

However, if you pipe the output to Format-List, you see much more information—far too much to review comfortably on screen—so it's better to pipe the output to a text file and compare it at your leisure.

The Get-Mailbox cmdlet does not return every property you can set on a user object because EMS differentiates between general Active Directory properties for a user object and those that are specific to Exchange. For example, Get-Mailbox does not list the Office property for a user because every user object in Active Directory has this property regardless of whether it is mail-enabled. Thus, if you want to retrieve or update the Office property, you have to use the Get-User and Set-User cmdlets, respectively. The same differentiation exists for groups and contacts when the Get-Group/Set-Group and Get-Contact /Set-Contact cmdlets are available.

Selective output

It is easy to list every property, but when you have limited screen space, you need to be more selective about the properties you want to output, and that's why it's often a good idea to use the Select-Object cmdlet to select the data you need before you pipe to Format-Table. In this case, you use the Select alias for Select-Object just because this cmdlet is used so often and it is nice to use shorthand.

```
Get-Mailbox –Identity Pelton | Select Name, PrimarySmtpAddress, Database
```

```
Name                PrimarySmtpAddress                    Database
----                ------------------                    --------
David Pelton        David.Pelton@contoso.com              ExServe1\DB1
```

PowerShell output can obscure data because it contains too many spaces. For example:

```
Get-ExchangeServer
```

```
Name        Site                   ServerRole   Edition      AdminDisplayVersion
----        ----                   ----------   -------      -------------------
EXSERVER1   contoso.com/Conf....   Mailbox,...  Enterprise   Version 15.0 (Bu...
EXSERVER2   contoso.com/Conf....   Mailbox      Enterprise   Version 15.0 (Bu...
```

To force PowerShell to remove spaces and display more useful data, pipe the output to the Format-Table cmdlet and use the –AutoSize parameter to fit the output columns into the available space:

```
Get-ExchangeServer | Format-Table -AutoSize
```

```
Name       Site      ServerRole                Edition        AdminDisplayVersion
----       ----      ----------                -------        -------------------
EXSERVER1  contoso.com/Configuration/Sites/Default-First-Site-Name
                     Mailbox, ClientAccess   Enterprise     Version 1...
EXSERVER2  contoso.com/Configuration/Sites/Default-First-Site-Name
                     Mailbox, ClientAccess   Enterprise     Version 1...
```

Another way of extracting and then working with data is to direct the output of a command into a variable, in which case you have a complete picture of the object's properties in the variable. For example, this command loads all the available information about the ExServer2 server into the $Server variable:

```
$Server = Get-ExchangeServer –Identity 'ExServer2' -Status
```

You can extract additional information about the server to use by including the name of the property in which you're interested. (Specifying the –Status parameter requests Get-ExchangeServer to provide some additional information about the current domain controller and global catalog the server is using.) You can also use a variable as an array and populate the array with a call to a command.

In this example, you populate a $Mailboxes array with a call to Get-Mailbox, using a filter to extract details of all the mailboxes stored in a particular database. This output is a good example of how cmdlets can generate individual objects or an array of objects with each object being individually accessible within the array.

```
$Mailboxes = Get-Mailbox –Database DB2
```

When it is populated, you can then navigate through the array as follows:

```
$Mailboxes[0]
$Mailboxes[1]
$Mailboxes[2] etc etc etc.
```

You can reference specific properties of the objects by using the "." operator.

```
$Mailbox[2].Name
$Mailbox[53].PrimarySmtpAddress
```

INSIDE OUT Finding what you want when there's a lot of output

The output from a cmdlet such as Get-Mailbox can easily result in a lot of data that are hard to read to find the piece of information in which you are really interested. One technique that helps is to pipe the output to the Out-String cmdlet and then use the FindStr cmdlet to search the output for a particular term. For example, here's how to use the two cmdlets to search the output from Get-Mailbox to find a particular term. In this instance, EMS lists any occurrence of the word "Tony" if it exists in the list of mailbox names Get-Mailbox returns:

```
Get-Mailbox | Out-String | FindStr "Tony"
```

By default, EMS truncates the output of multivalue properties after 16 values. For example:

```
Get-Mailbox –Identity 'Pelton, David' | Format-List Name, EmailAddresses


   Name           : Pelton, David
   EmailAddresses : {smtp:dp12@contoso.com, smtp:dp11@contoso.com, smtp:dp10@
   contoso.com, smtp:dp9@contoso.com, smtp:dp8@contoso.com, smtp:dp7@contoso.com,
   smtp:dp6@contoso.com, smtp:dp4@contoso.com, smtp:dp5@contoso.com, smtp:dp3@
   contoso.com, smtp:dp2@contoso.com, smtp:dp1@contoso.com...}
```

Truncation can hide some valuable data. In the preceding example, many of the email addresses are defined for a mailbox, but the default Simple Mail Transfer Protocol (SMTP) address is not shown. If this limitation becomes a concern, you can force EMS to output more values for a property by amending a *$FormatEnumerationLimit* variable. This variable is defined in the EMS initialization script (\bin\Exchange.ps1), and the default value of 16 is usually more than sufficient for normal purposes. If you want to see more variables, you can set the variable to a different limit or set it to -1 to instruct EMS that it can enumerate as many values as are available for any property. For example:

```
$FormatEnumerationLimit = -1
Get-Mailbox –Identity 'Pelton, David' | Format-List Name, EmailAddresses
```

```
Name           : Pelton, David
EmailAddresses : {smtp:dp12@contoso.com, smtp:dp11@contoso.com, smtp:dp10@
contoso.com, smtp:dp9@contoso.com, smtp:dp8@contoso.com, smtp:dp7@contoso.com,
smtp:dp6@contoso.com, smtp:dp4@contoso.com, smtp:dp5@contoso.com, smtp:dp3@
contoso.com, smtp:dp2@contoso.com, smtp:dp1@contoso.com, smtp:dp@contoso.com,
smtp:dp16@contoso.com, smtp:dp15@contoso.com, smtp:dp14@contoso.com, smtp:dp13@
contoso.com, smtp:Pelton@contoso.com, SMTP:David.Pelton@contoso.com}
```

Using common and user-defined variables

PowerShell includes a number of variables you will use a lot. *$True* and *$False* are variables you can pass to shell commands and scripts to check for true and false conditions. Usually, *$True* is equivalent to setting a check box for an option in EMC, and *$False* is equivalent to clearing a check box. If you prefer numeric values, you can replace *$True* and *$False* with 1 (one) and 0 (zero), respectively. Other global variables you commonly meet as you work with PowerShell include *$Null* (no value), *$home*, which returns the user's home folder, and *$pwd*, which returns the current working folder. Important Exchange variables include the following:

- ***$ExBin*** Points to the directory in which Exchange binaries and other important files are kept. On an Exchange 2013 server, this variable normally resolves to disk: \Program Files\Microsoft\Exchange Server\V15\bin.
- ***$ExScripts*** Points to the directory in which important Exchange .ps1 scripts are kept. On an Exchange 2013 server, this variable resolves to disk: \Program Files \Microsoft\Exchange Server\V15\Scripts.
- ***$ExInstall*** Points to the root directory for Exchange. On an Exchange 2013 server, this variable resolves to disk: \Program Files\Microsoft\Exchange Server\V15.

You can use these variables to access files in these directories. For example, to see a list of scripts Exchange provides, type **Dir $ExScripts**.

Checking that a value is *$True* or *$False* is a common occurrence. For positive conditions, you can shorten the check by just passing the property against which to check, and PowerShell will assume that you want to check whether it is true. For example, assume that you want to find out which mailboxes are enabled to use Outlook Web App. You can use this command and, as you can see, there is no mention of *$True*, but it works:

```
Get-CASMailbox | Where-Object {$_.OWAEnabled} | Select Name
```

Note the use of *$_* in the last command. *$_* is a very important variable because it points to the current object in the pipeline. Scripting languages on other platforms such as UNIX and Linux also support pipelines, which compose complex commands by allowing the output of one command to be passed as the input to another. The | operator indicates that a pipeline is in place. Data are passed as fully formed objects rather than as a text stream. This enables PowerShell to operate on the full structure of data that are pipelined, including the attributes and types that define the objects piped from one cmdlet to another.

For example, if you create a filter to look for people in a certain department because you want to update the name of the department, you might do this:

```
Get-User | Where-Object {$_.Department –eq 'Legal'} | Set-User –Department 'Law'
```

The Department property is prefixed with $_ to indicate that you want to check this property for every object the call to Get-User passes through the pipeline. You actually use $_. as the prefix because it includes the "." operator to specify that you want to access a property. If you just passed $_ the comparison would not work because PowerShell would compare "Legal" against the complete object.

User-defined variables can be integer, decimal, or string—you decide by passing a value to the variable you want to use. For example:

```
$Tony = 'Tony Redmond'
$Figure = 15.16
```

This creates a string variable, and the second variable holds a decimal value. Variables are case-insensitive and case-preserving. Using the preceding example, you can refer to $Tony as $TONY or $tony or even $ToNY, and PowerShell will refer to the same variable. Variables are local unless you declare them to be global by prefixing them with Global, as in:

```
$Global:Tony = 'Tony Redmond'
```

When a variable is global, you can reference it interactively and in scripts you can call from anywhere.

A word of caution about PowerShell and quotation marks

Be careful how you use quotation marks in PowerShell because although it might appear that double and single quotation marks are interchangeable, there is a subtle difference that might catch you out. Single quotation marks represent a literal string, one that PowerShell will use exactly as you provide it. Double quotation marks mean that PowerShell should examine the string and resolve any variable it finds inside through a process called variable expansion. Consider this example:

```
$n = Date
$n1 = 'Right now, it is $n'

Right now it is $n
```

```
$n2 = "Right now, it is $n"
$n2

Right now, it is Tue Jan 16 17:59:54 2013
```

Can you see the difference a little quotation mark makes? Best practice is to use single quotation marks whenever you are sure that you want a string variable to stay exactly as you have typed it and to use double quotation marks elsewhere. Be careful about using editors that insert smart quotation marks because PowerShell cannot deal with them; it is best to use a simple text editor whenever you create or edit a script. You cannot mix and match the different types of quotation marks to enclose a variable because PowerShell will refuse to accept the command. You will not do any great harm if you use double quotation marks instead of single quotation marks, but it is best to use single quotation marks as the default.

Tip

Do not include hyphens when you name variables because PowerShell interprets the hyphens as parameters. In other words, $ServerName is a good name for a variable, but $Server-Name is not.

Like any good scripting language, PowerShell supports conditional checking with IF and ELSEIF that you will mostly use in scripts. It’s easy to generate code that goes through a certain number of iterations with constructs such as `1..100 | ForEach-Object <command…>`. You will see examples of these constructs as you see more sophisticated PowerShell code in later chapters.

Using PowerShell ISE with Exchange

If you don’t like the bare-bones nature of EMS, you might prefer to use ISE, the PowerShell Integrated Scripting Environment. ISE is installed on Windows 2008 R2 SP1 and Windows 2012 servers to provide a GUI for PowerShell that allows users to write, test, and debug scripts. PowerShell ISE is also installed by default on Windows 7 and Windows 8 workstations.

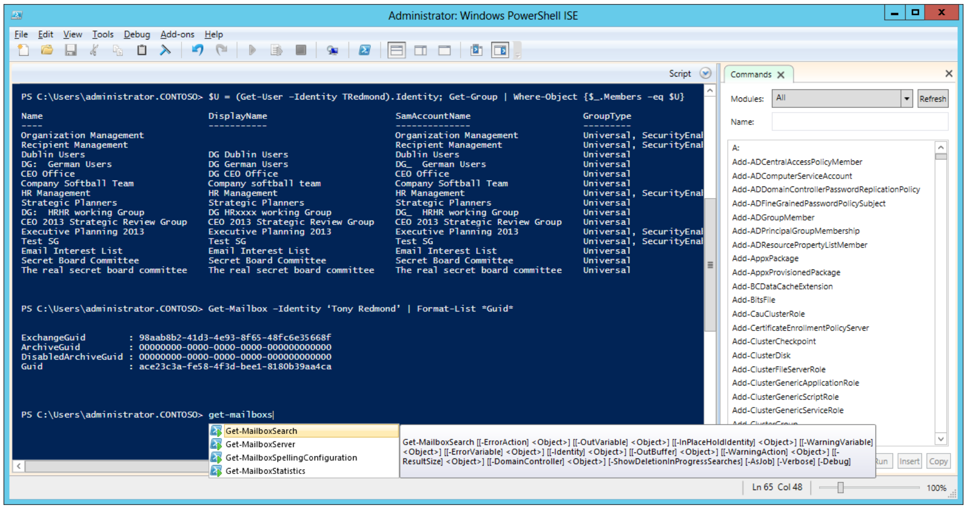
ISE supports multiline editing, tab completion, syntax coloring (or highlighting of different parts of commands), context-sensitive help, and keyboard shortcuts. Because of its debug features, ISE is a good way to write complex scripts for use with Exchange 2013. All the code included in this book can be worked on through ISE.

When you start ISE, it has no knowledge of Exchange or how to create the kind of remote session with an Exchange server in the way EMS does when it starts. Some work is therefore necessary to integrate ISE with Exchange. The easiest way to do this is to insert some code in the PowerShell profile so that ISE learns enough about Exchange when it initializes to access Exchange when you need it to.

The code you need to use with ISE is very similar to the code you met earlier when discussing the basics of creating a remote PowerShell session. Start ISE and type **Notepad $Profile** to edit your PowerShell profile, and then insert the following code (amending the reference to contoso.com to reflect your own environment):

```
$PSISE.CurrentPowerShellTab.AddOnsMenu.SubMenus.Add(
 "Connect to Exchange", {
 $user = Get-Credential
 $Server = Read-Host "Connect to what Exchange server "
 $connectpoint = $Server + ".contoso.com/PowerShell/"
 $ExSession= New-PSSession -ConfigurationName Microsoft.Exchange -ConnectionUri
$connectpoint -Credential $user
 Import-PSSession $ExSession
 },
 "Control+Alt+1"
)
```

The code defines a new menu choice called Connect to Exchange that appears on the ISE Add-ins menu. The option can also be invoked with the Control/Alt/1 key combination. In either case, when invoked, the code prompts for user credentials and the server to which to connect and then initiates a new remote PowerShell session with the selected Exchange server. After the connection is established, you can work as with EMS except that extra information and facilities are available to you, such as a context-sensitive list of cmdlets that appears when you start typing a cmdlet name (Figure 3-4).


Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script
Commands
Modules: All
Refresh
Name:
A:
Add-ADCentralAccessPolicyMember
Add-ADComputerServiceAccount
Add-ADDomainControllerPasswordReplicationPolicy
Add-ADFineGrainedPasswordPolicySubject
Add-ADGroupMember
Add-ADPrincipalGroupMembership
Add-ADResourcePropertyListMember
Add-AppxPackage
Add-AppxProvisionedPackage
Add-BCDataCacheExtension
Add-BitsFile
Add-CauClusterRole
Add-CertificateEnrollmentPolicyServer
Add-ClusterCheckpoint
Add-ClusterDisk
Add-ClusterFileServerRole
Add-ClusterGenericApplicationRole
Add-ClusterGenericScriptRole
Add-ClusterGenericServiceRole
Run
Insert
Copy
Ln 65 Col 48
100%

```
PS C:\Users\administrator.CONTOSO> $U = (Get-User -Identity TRedmond).Identity; Get-Group | Where-Object {$_.Members -eq $U}

Name                           DisplayName                    SamAccountName                 GroupType
----                           -----------                    --------------                 ---------
Organization Management                                       Organization Management        Universal, SecurityEnal
Recipient Management                                          Recipient Management           Universal, SecurityEnal
Dublin Users                   DG Dublin Users                Dublin Users                   Universal
DG:  German Users              DG German Users                DG_  German Users              Universal
CEO Office                     DG CEO Office                  CEO Office                     Universal
Company Softball Team          Company softball team          Company Softball Team          Universal
HR Management                  HR Management                  HR Management                  Universal, SecurityEnal
Strategic Planners             Strategic Planners             Strategic Planners             Universal
DG:  HRHR working Group        DG HRxxxx working Group        DG_  HRHR working Group        Universal
CEO 2013 Strategic Review Group CEO 2013 Strategic Review Group CEO 2013 Strategic Review Group Universal
Executive Planning 2013        Executive Planning 2013        Executive Planning 2013        Universal, SecurityEnal
Test SG                        Test SG                        Test SG                        Universal, SecurityEnal
Email Interest List            Email Interest List            Email Interest List            Universal
Secret Board Committee         Secret Board Committee         Secret Board Committee         Universal
The real secret board committee The real secret board committee The real secret board committee Universal


PS C:\Users\administrator.CONTOSO> Get-Mailbox -Identity 'Tony Redmond' | Format-List *Guid*


ExchangeGuid        : 98aab8b2-41d3-4e93-8f65-48fc6e35668f
ArchiveGuid         : 00000000-0000-0000-0000-000000000000
DisabledArchiveGuid : 00000000-0000-0000-0000-000000000000
Guid                : ace23c3a-fe58-4f3d-bee1-8180b39aa4ca



PS C:\Users\administrator.CONTOSO> get-mailboxs|
```

```
Get-MailboxSearch
Get-MailboxServer
Get-MailboxSpellingConfiguration
Get-MailboxStatistics
```

```
Get-MailboxSearch [[-ErrorAction] <Object>] [[-OutVariable] <Object>] [[-InPlaceHoldIdentity] <Object>] [[-WarningVariable]
<Object>] [[-ErrorVariable] <Object>] [[-Identity] <Object>] [[-OutBuffer] <Object>] [[-WarningAction] <Object>] [[-
ResultSize] <Object>] [[-DomainController] <Object>] [-ShowDeletionInProgressSearches] [-AsJob] [-Verbose] [-Debug]
```

Figure 3-4 Working with Exchange 2013 through the PowerShell ISE

Identities

You might have noticed the –Identity parameter in some of the cmdlets you have explored so far. In many cases, a call to an Exchange cmdlet results in a set of objects being returned (for example, all the mailboxes on a server). In these instances, you might need to identify a specific object within the chosen set with which to work. (Think of a pointer to an item in an array.) For example, if you issue the Get-ExchangeServer cmdlet, you retrieve a list of all the Exchange servers in the organization. If you want to work with one server, you have to tell EMS which server you want to select by passing its identity. For example, to work with just the server named ExServer1:

```
Get-ExchangeServer –Identity 'ExServer1'
```

Apart from its obvious use to identify the object with which you want to work, –Identity has a special meaning within PowerShell because it is a positional parameter. You can specify the parameter's value without specifying the parameter's name, so the example previously used is just as valid if you use:

```
Get-ExchangeServer 'ExServer1'
```

INSIDE OUT **Best practice to include the –Identity parameter**

Although you might find it faster to omit the –Identity parameter when you're working interactively with EMS, it is best practice always to include the –Identity parameter when you write code for reusable scripts because this ensures that there is no possibility that another administrator or programmer will mistake the value passed for the identity for anything else.

If you want, you can retrieve a list of objects and store them in a variable and retrieve the values as you wish. The variable holds the objects as an array. For example, to populate a variable with a set of mailboxes hosted by a server:

```
$Mbx= Get-Mailbox –Server 'ExServer1'
```

To retrieve the different objects in the array, pass the number of the object with which you want to work, starting from zero. For example, to fetch the first mailbox in the array:

```
$Mbx[0]
```

You can be more specific and ask for one of the object's properties. For example, to get the identity of the first mailbox in the array:

```
$Mbx[0].Identity
```

```
IsDeleted           : False
Rdn                 : CN=Eoin P. Redmond
Parent              : contoso.com/Exchange Mailboxes
Depth               : 3
DistinguishedName   : CN=Eoin P. Redmond,OU=Exchange Mailboxes,DC=contoso,DC=com
IsRelativeDn        : False
DomainId            : contoso.com
ObjectGuid          : 0bcd15b3-c418-43be-b678-2658614f732b
Name                : Eoin P. Redmond
```

You might be surprised by the amount of information returned here for the mailbox's identity (it's all defined in the schema), but it contains all the ways you can navigate to this object through its relative distinguished name (shown here as the rdn property), distinguished name, globally unique identifier (GUID), and name. Normally, you'll just use the name of a mailbox to find it, but you can use the other methods, and Exchange will find the mailbox. There is no requirement to parse out a specific piece of the identity you want to use or to trim values; PowerShell does it all for you. For example, you can use an identity to discover the groups to which a user belongs. Here's the code:

```
$U = (Get-User –Identity TRedmond).Identity; Get-Group | Where-Object {$_.Members –eq
$U}
```

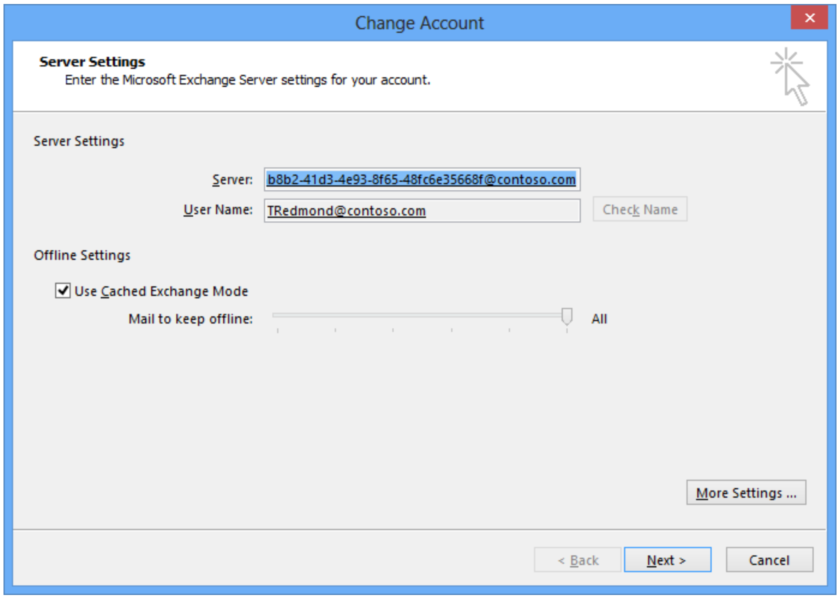
The Get-User cmdlet loads the user's identity into a variable, and then the Get-Group and the Where-Object cmdlets scan all groups to discover any that include the user in their membership. Scanning the membership list of groups to discover string matches is never going to be as quick (and will get slower as the number of groups in the forest grows) because a string compare will never get close to the backward pointers that consoles such as Active Directory Users and Computers or EMC use to display group membership in terms of speed of access, so don't be surprised. Scanning for group membership in this way takes some time to complete.

If you don't like user-friendly forms such as email addresses or mailbox names, Exchange also allows you to use GUIDs as identifiers. Because they are obscure and long, GUIDs are difficult to type, but you can still use them. One slightly complicating factor is that you must know which GUID to use where. You might want the GUID that points to a user's mail-box, the GUID pointing to her Active Directory account, or even the one pointing to her archive mailbox. For example, this command displays all GUIDs registered for a mailbox:

```
Get-Mailbox –Identity 'Tony Redmond' | Format-List *Guid*
```

```
ExchangeGuid        : c2c4a3b5-c1a6-5a17-971d-8549123a78d0
ArchiveGuid         : 00000000-0000-0000-0000-000000000000
DisabledArchiveGuid : 00000000-0000-0000-0000-000000000000
Guid                : 288617d1-4592-4211-bb20-26ab755458c8
```

The ExchangeGuid property points to the user's mailbox. This is a tremendously important property because the GUID pointing to a mailbox can be guaranteed to be unique across an Exchange organization, which is why the Store uses this value to locate a user's mailbox. It's also why Outlook users see the ExchangeGuid of their mailbox instead of the server name when viewing the server name property shown when viewing the server settings of an Exchange 2013 mailbox (Figure 3-5).



Server: b8b2-41d3-4e93-8f65-48fc6e35668f@contoso.com
User Name: TRedmond@contoso.com
[✓] Use Cached Exchange Mode
Mail to keep offline: All

Figure 3-5 How Outlook displays the ExchangeGuid

It is confusing, but if you run Get-MailboxStatistics to retrieve summary details of the contents of a mailbox, EMS returns a MailboxGuid property. This is the same value as the ExchangeGuid when reported by Get-Mailbox. Why Microsoft felt that two names were required for the same GUID is beyond me.

The Guid property identifies the user's Active Directory account and thus provides the essential link between a mailbox and an account. In this case, the ArchiveGuid is shown as all zeros, so no archive mailbox is associated with this mailbox. The DisabledArchiveGuid value is also all zeros. This GUID is used only when a user has been assigned an archive mailbox that was subsequently disabled for some reason. Exchange maintains the GUID so the archive can be reconnected to the mailbox up to the point at which it is permanently removed from a database after the expiry of the deleted mailboxes' retention period.

Now that you know what the GUIDs are, you could use them to reference a mailbox. For example:

```
$GUID = (Get-Mailbox –Identity 'Tony Redmond').Guid
Get-User | Where {$_.Guid –eq $GUID} | Format-Table Name
```

The great thing about identities is that you sometimes don't need to use them. This situation occurs when you pipe information from one cmdlet for processing by another because the shell understands that it needs to operate on the current object that has been fetched through the pipe. For example, this command pipes a list of mailbox identities passed in strings to the Set-Mailbox cmdlet:

```
"TRedmond", "JSmith", "JDoe" | Set-Mailbox –Office "Dublin"
```

Piping

You'll pipe output from one cmdlet to another frequently as you work with Exchange data. The important thing to remember is that PowerShell outputs fully formed objects that can be manipulated when fed as input to other cmdlets through the pipeline. This wouldn't be possible if PowerShell output text strings. For example, assume that you want to change the value of the Office property for a set of users who have moved to a new building. It would be tedious if you had to fetch the identity of each user individually, determine each identity, and then pass the value to make the change to each user's properties. A simple pipe works because PowerShell knows that it can use the stream of data from one command to identify the objects it has to process with another. Here's how you might update the Office property for a complete set of users without any mention of an identity. You'll see that the two cmdlets that do the work are separated by the pipe character, "|". This is the character that tells PowerShell to pipe the output from the first cmdlet to become the input to the second.

```
Get-User –Filter {Office –eq 'Building A'} | Set-User –Office "Building B"
```

Too many objects

By default, EMS returns up to 1,000 objects in response to cmdlets. (The value in Exchange 2007 is 5,000.) Therefore, if you run a cmdlet such as Get-Mailbox, Exchange will return up to 1,000 mailboxes if they are available. If you work in a small Exchange organization that supports fewer than 1,000 mailboxes, you don't need to worry too much about the number of objects you have to deal with because PowerShell will likely return relatively few objects, and things usually progress quickly. However, it's a different situation in large organizations, in which you have to pay attention to the filters you specify to retrieve data or override the default limit for returned objects by specifying the ResultSize parameter for cmdlets. For example, to let EMS return as many mailboxes as it can find, you could use a command like this:

```
Get-Mailbox –ResultSize Unlimited
```

This command will work, but it will be very slow because EMS has to read every mailbox in the organization. Think about how long this might take to execute in an organization that supports more than 300,000 mailboxes. In these situations it's always better to specify a filter to restrict the number of objects EMS looks for and returns.

OPATH filters

OPATH is the basic syntax used for PowerShell queries. It is similar in concept to but uses different syntax from Lightweight Directory Access Protocol (LDAP) queries. Dynamic distribution groups (see Chapter 6, “Groups and other objects”) also use OPATH queries to locate objects in Active Directory when the transport system builds addressee lists to deliver message addresses to these groups.

Some base guidelines about the syntax OPATH queries are as follows:

- OPATH requires a hyphen before –and, –or, and –not operators.
- Comparison operators include –eq (equal), –ne (not equal), –lt (less than), –gt (greater than), –like (like), –ilike, and –notlike. –Like and –notlike are wildcard string compares. –iLike and –inotlike are case-insensitive.
- Filters should be expressed within braces; for example, {Office –eq ‘London’}.

You’ll see many more examples of OPATH queries in the remainder of this book.

Server-side and client-side filters

Windows PowerShell supports server-side and client-side filters. There’s a big difference in performance between the two types of filters, especially when you have to process more than a hundred objects. Client-side filters are the default. Any code that uses the Where cmdlet executes a client-side filter. Client-side filters request data from a server and then perform the filtering on the client. This is an effective approach if you only have 10 or 15 objects to process, but it obviously doesn’t scale too well as the number of objects increases.

Server-side filters have better scalability because the request for data forces the server to return a filtered data set to the client. Because Exchange servers often have to deal with tens of thousands of objects, a number of the Exchange cmdlets support server-side filters. If a cmdlet supports the –Filter parameter, it supports server-side filters. Usually, these are cmdlets that deal with objects that output large numbers, such as mail-enabled recipients or message queues. All the precanned filters generated for dynamic distribution groups, address lists, and email address policies use server-side filters.

As an example of server-side and client-side filtering in action, two methods are available to find all the mailboxes with “James” in their name, as demonstrated in these commands:

```
Get-Mailbox –Filter {Name –like '*James*'} –ResultSize 5000
Get-Mailbox –ResultSize 5000 | Where {$_.Name –like '*James*'}
```

On the surface, these two pieces of code seem reasonably similar, but they are very different in reality. The first difference is that the first code example uses a server-side filter, and the second uses a client-side filter. The second difference is that the two filter types can generate very different results because of the way the filters operate. If you omit the –ResultSize parameter, the same query is generated: Find all the mailboxes with a name that contains "James." (The ResultSize parameter in the first example limits the total number of objects returned to 5,000.) However, if you time both queries, the server-side filter invariably executes faster than the client-side filter, largely because fewer data are transferred between server and client. To understand why the filters generate different results, you have to appreciate how the filters work:

- The server-side filter returns the first 5,000 mailboxes it finds that include "James" in the mailbox name.
- The client-side filter fetches data for the first 5,000 mailboxes and then applies the filter to find the mailboxes that include "James" in the mailbox name. However, the filter applies only to the set the client fetched and might not find all the mailboxes you actually want to discover.

Even though you ask the server-side filter to do more work (working with any reasonably sized set of mailboxes, the server-side filter will have to process significantly more data to find the first 5,000 mailboxes that match), it still executes faster. For example, when I executed similar commands within a very large Exchange organization (170,000 mailboxes), the server-side filter completed processing in 43 seconds, whereas the client-side filter completed in 81 seconds. The rule here is that the effect of server-side filtering gets better as the number of objects increases.

INSIDE OUT PowerShell and memory limits

Another aspect to consider is that PowerShell cannot fetch and cache data on disk temporarily the way a database might. This is not an issue if you want to process only a few objects, but it can lead to memory issues if you attempt to process tens of thousands of mailboxes at one time, especially if you use client-side filters and want to pipe the output to another command. In this case, you ask PowerShell to find all the objects that match the specified filter, store the data in memory, process the data, and pipe the matching objects to the second command. Experience shows that these operations can cause PowerShell to complain that it is running out of memory. This is likely to be one of the growing pains through which all software goes and, apart from using loops to process data, no good solution to the memory exhaustion problem is available today.

Sometimes people make the mistake of assuming that client-side filters are faster because server-side filters provide the data in one motion after the server processes all the data. You therefore wait for a while without seeing anything and then see all the filtered records at one time. By comparison, client-side filters fetch and filter data continuously, so you see output as the command finds each matching record. However, the important indicator of performance is how long each type of filter takes to complete, and server-side filters are always faster.

The commands you are most likely to use with server-side filters are as follows:

- **Get-User** Retrieve basic Active Directory properties for any user account, including mail-enabled accounts.
- **Get-Mailbox** Retrieve Exchange-specific properties for mailboxes.
- **Get-DistributionGroup** Retrieve Exchange-specific properties for mail-enabled groups.

Each of the commands you can use to work with user accounts, groups, and mailboxes supports a different set of filterable properties. To discover which properties are available for filtering, you can use PowerShell to query the properties of a returned object. For example:

```
Get-Mailbox -Identity Redmond | Get-Member | Where-Object {$_.MemberType –eq
'Property'} | Sort-Object Name | Format-Table Name
```

This set of commands calls a command to return some information about an object. It then pipes the information returned by the first command to the Get-Member cmdlet, which extracts information about the properties. You sort the properties by name and output them in table format. Here's an excerpt from the output:

```
Name
----
AcceptMessagesOnlyFrom
AcceptMessagesOnlyFromDLMembers
AddressListMembership
Alias
AntispamBypassEnabled
CustomAttribute1
CustomAttribute10
...
WindowsEmailAddress
```

This method works for the Get-Mailbox, Get-CASMailbox, Get-User, Get-Recipient, Get-DistributionGroup, and Get-DynamicDistributionGroup cmdlets. You can use any of the values reported in a –Filter statement. For instance, the call you just made to Get-Mailbox reports that the custom attributes are available, so to find all mailboxes that have a value in the CustomAttribute10 property, you can generate a command like this:

```
Get-Mailbox -Filter {CustomAttribute10 -ne $Null}
```

If you look at the filterable properties reported by the Get-DynamicDistributionGroup cmdlet, you can see that the ManagedBy property is available for this dynamic distribution group, whereas it is not for mailboxes. Hence, you can execute a filter like this:

```
Get-DynamicDistributionGroup -Filter {ManagedBy -ne $Null}
```

When you create a filter, it is best to be as specific as possible. You can state several conditions within a filter. An example of a server-side filter that returns all the mailboxes in the Dublin office where the user name contains “Tony” is shown next. The Get-User cmdlet also works with this filter, but Get-Mailbox executes a tad faster because the server does not have to process accounts that are not mail-enabled.

```
Get-Mailbox -Filter {Office -eq 'Dublin' -and Name -like '*Tony*'}
```

After you have mastered server-side filtering, you will use it all the time to work with sets of users. For example, assume that you want to give a new mailbox quota to members of a certain department but no one else.

```
Get-User -Filter {Department -Eq 'Advanced Technology'} | Set-Mailbox
-UseDatabaseQuotaDefaults:$False
-IssueWarningQuota 5000MB -ProhibitSendQuota 5050MB -ProhibitSendReceiveQuota 5075MB
```

INSIDE OUT WhatIf and Confirm

Before you execute any command to perform a bulk update of objects, you can run the command with the /whatIf switch added to force EMS to show you which objects will be altered. After you are sure that the correct set of objects will be updated, you can run the command without /whatIf, and EMS will perform the changes. The /confirm switch is also useful in terms of stopping administrators before they do something they should not. If you include the Confirm parameter, EMS prompts the administrator with “Are you sure that you want to perform this action” and waits for a “Y” or “Yes” response (or “A” for “all” if multiple objects are involved) before continuing. Act in haste, repent in leisure.

Transcripts

If you encounter a problem executing some EMS commands and need to produce some debug information to give to your support team or Microsoft, you can do this by generating a transcript. A transcript captures details of all commands executed in a session and is useful in terms of capturing the steps necessary to solve a problem or documenting steps to expose an issue that you want to report to Microsoft. You can combine this by adding the –Verbose parameter to most commands to gather a lot of information about what you've tried to do and what happened when you tried it. Use the Start-Transcript cmdlet to force EMS to capture debug information. For example:

```
Start-Transcript c:\Temp\Transcript.txt
```

All commands and output will be captured until you stop the transcript by using the Stop-Transcript cmdlet. At this point, you can examine the output with any text editor, and you'll see something like the output shown in the following example.

```
**********************
Windows PowerShell Transcript Start
Start time: 20130313093116
Username  : CONTOSO\Administrator
Machine   : ExServer1 (Microsoft Windows NT 6.2.9200.0)
**********************
PS C:\temp> $env:path
C:\Windows\system32\WindowsPowerShell\v1.0\;C:\Windows\system32;C:\Windows;C:
\Windows\System32\
Wbem;C:\Windows\System32
\WindowsPowerShell\v1.0\;C:\Windows\idmu\common;C:\Program Files\System Center
Operations Manager 2007\;C:\Program Files\Microsoft\Exchange Server\V14
\bin;c:\temp
```

Bulk updates

Those faced with the task of bulk updates (either to create a lot of new mailboxes or other objects or to modify many existing objects) before the advent of PowerShell support for Exchange had quite a lot of work ahead of them because Exchange offered no good way to perform the work. You could create comma-separated value (CSV) or other load files and use utilities such as CSVDE or LDIFDE to process data in the files against Active Directory, or you could write your own code to use CDOEXM or ADSI to update Active Directory. Either approach involved a lot of detailed work and made it quite easy to make a mistake. Using a console to make the necessary changes was boring and an invitation to make a mistake. The cause of Exchange's problems with bulk changes was the lack of a programmable way to automate common management operations, a situation that changed with the arrival of EMS.

You can combine the Get-User and Set-Mailbox cmdlets effectively to solve many problems. Here is an example in which you need to update the send quota property on every mailbox for a set of users whose business group has decided to fund additional storage. You can identify these users by their department, which always starts with "Advanced Tech" but sometimes varies into spellings such as "Advanced Technology" and "Advanced Technology Group." Conceptually, the problem is easy to solve:

1. Look for all users who have a department name beginning with "Advanced Tech."
2. Update the send quota property for each user.

You could use the Find option in Active Directory Users and Computers to build a suitable filter to establish the set of users, but then you have to open each user's mailbox that Active Directory Users and Computers locates to update his quota through the GUI, which could become boring after several accounts. You could also export a CSV-formatted list of users to a text file, manipulate the file to find the desired users, and then process that list through CSVDE to make the changes, but you have to search for all matching users across the complete directory first. That is a lot of work to do.

The process is easier in EMS. First, you use the Get-User cmdlet with a suitable filter to establish the collection of mailboxes you want to change. The following command returns all users who have a department name that begins with "Advanced Tech" and then updates the ProhibitSendQuota property to the desired amount (say, 20 GB). Because you have a collection of user objects established, you can use the Set-Mailbox cmdlet to perform the update. Note that some of these users might not be mail-enabled, but error handling is another day's work.

```
Get-User | Where {$_.Department –like '*Advanced Tech*'} | Set-Mailbox
–ProhibitSendQuota 20GB –UseDatabaseQuotaDefaults $False
```

Mergers, acquisitions, and internal reorganizations pose all sorts of problems for email administrators. EMS will not solve the big problems, but it can automate many of the mundane tasks that are necessary. For example, department names tend to change during these events. EMS makes it easy to find all users who belong to a specific department and update their properties to reflect the new organizational naming conventions. If only executing organizational change were as easy as this one-line command, which transfers everyone who works for the Old Designs department over to the Cutting Edge Design department, things would be much easier:

```
Get-User | Where {$_.Department –eq 'Old Designs'} | Set-User –Department 'Cutting
Edge Design'
```

Note the use of $_.Department; this indicates a value fetched from the current pipeline object. In this case, it is the department property of the current user object that Get-User fetched. To verify that you have updated all the users you wanted to (and maybe provide a report to human resources or management), you can use code like this:

```
Get-User | Where {$_.Department -eq 'Cutting Edge Design'} | Select Name,
Department | Sort-Object Name | Format-Table > c:\temp\Cutting-Edge.tmp
```

A variation on this theme is to output the data to a CSV file to make the data easier to work with in Microsoft Excel, Microsoft Access, or another tool that can read CSV data.

```
Get-User | Where {$_.Department -eq 'Cutting Edge Design'} | Select Name, Department
| Sort Name | Export-CSV c:\temp\Cutting-Edge.CSV
```

Things are even easier if you just need to change everyone's company name after your company is acquired.

```
Get-User | Set-User -Company 'New Company'
```

You can even do such things as alter only the users whose mailbox belongs to a particular database:

```
Get-Mailbox -Database 'VIP Mailboxes' | Set-User -company 'Big Bucks'
-Department 'Executives'
```

Tip

All the examples discussed so far depend on you being able to identify some property you can use as the basis for a filter. But what about when you do not have a common property value to check for? In this case, you can build a simple list of mailbox names (or any other format the –Identity parameter will accept, such as a Universal Principal Name [UPN]), use the Get-Content cmdlet to read the names one by one, and pipe these values to whatever other command you need to use. For example, here is how you can use that trick to enable ActiveSync access for a set of users. In this example, the Get-Content cmdlet reads lines containing the identities of the mailboxes you want to change from a text file and pipes them as input to the Set-CASMailbox cmdlet:

```
Get-Content c:\temp\Users.txt | Set-CASMailbox -ActiveSyncEnabled $True
```

Another example of when EMS excels is when you want to apply a common setting across all servers in your organization. For example, assume that you want to apply a new deleted item retention limit of 150 days (perhaps mandated by the legal department) to all servers:

```
Get-MailboxDatabase | Set-MailboxDatabase -DeletedItemRetention 150.00:00:00
```

These simple examples demonstrate the value of having a scripting language that supports automation of common management tasks.

Calling scripts

After you have written a script, you have to decide where to keep it. You could put the new script in the directory that stores the Exchange binaries, but this is a bad idea for many reasons, not least because your script could be overwritten by the installation of a future Exchange service pack, a roll-up update, or even a completely new version.

INSIDE OUT **A wise practice**

It is wise to maintain a clear separation between the code for which you are responsible and the code Microsoft distributes with Exchange. Therefore, you should create a directory to hold all the scripts you use to work with Exchange. You can then call your scripts safely in the knowledge that they will be available.

The basic rule of calling a script is that if the script is in the working directory (the directory you are currently in), you prefix the name with ".\"

```
C:>.\Get-All-Users.ps1
```

If you're not in the right directory, you can move to where you want to be by using the cd command:

```
C:> cd c:\Scripts\
```

Alternatively, you can supply the full path to where the script is located:

```
C:>c:\Scripts\Get-All-Users.ps1
```

If there are spaces in the directory names, then you need to enclose the path in single or double quotation marks:

```
C: '\Program Files\Microsoft\Exchange Server\V15\Scripts\CollectOverMetrics.ps1'
```

Even better, you can amend the path PowerShell uses by looking for scripts and adding your directory to it. For example, running this command adds the C:\MyScripts directory to the path:

```
$env:path = $env:path + ";c:\MyScripts'
```

After a script is in a directory that's included in the path, you can invoke it by just typing its name.

Execution policies

EMS is powerful, and just a few cmdlets can have a tremendous effect on many objects throughout Exchange. You might have thought about how to control the ability of users to execute EMS commands.

RBAC provides the first line of protection. As you recall, users are permitted access only to the set of cmdlets and parameters available to the roles each user holds. Even though trusted users are assigned the roles they need to do their work, you still don't want them to execute scripts they download from the Internet or obtain elsewhere.

A second line of defense is therefore provided by Execution Policies, which define the conditions under which Windows PowerShell loads files for execution. There are four policies: Restricted, AllSigned, RemoteSigned, and Unrestricted. You configure the execution policy used for a server by using the Set-ExecutionPolicy cmdlet. The default is RemoteSigned, which you can verify by using the Get-ExecutionPolicy cmdlet. In this mode, EMS permits the execution of any script created locally and any script downloaded from the Internet, provided the script includes a digital signature. All the scripts that come with Exchange are signed for this purpose (see Table 3-2). The caveat is that any script you attempt to run can contain only Exchange cmdlets that are supported by the role the user holds who invokes the script. Table 3-2 lists the alternate modes together with the potential trade-off in security that you might have to make for each mode.

TABLE 3-2 Windows PowerShell execution policies

Execution Policy mode	Meaning
Restricted	No scripts can be run, even if they are signed by a trusted publisher.
AllSigned	Scripts must be digitally signed by a trusted partner before EMS will run them.
RemoteSigned	EMS will run any script created locally. Scripts that originate outside the system (such as those downloaded from the Internet) cannot be run.
Unrestricted	EMS will run any script. This mode should be used for test environments only.

If you attempt to run an unsigned script that doesn't comply with policy, Windows PowerShell signals that it cannot load the script. Scripts are signed with the Set-AuthenticodeSignature cmdlet, but you need to get a valid certificate first. The certificate can be one you generate yourself or one you buy from a commercial vendor such as VeriSign.

See *http://technet.microsoft.com/en-us/library/bb125017.aspx* for further details of how to generate and apply certificates to sign scripts.

CAUTION!

Obviously, running an Exchange server with an unrestricted execution policy is a bad idea. In fact, you should avoid any deviation from the default policy unless you have an excellent reason to change. For example, you might decide that you want to run scripts you find on the Internet. This might be acceptable if you run the scripts on a test system only, but it's a much better idea to take the time to go through the code to understand exactly what it does before you think of deploying to a production system. Remember that if you edit a script to create a new version on your computer, that version of the script is now considered local and can be run without changing the execution policy. Opening a downloaded script and saving it can lead to unintended consequences, so be sure that you only save a script that you didn't write when you absolutely intend to create a new version.

If you deem it necessary to change the policy, use the Set-ExecutionPolicy command to update the default execution policy on an Exchange 2013 server. For example:

```
Set-ExecutionPolicy –ExecutionPolicy Unrestricted
```

The change to the execution policy is effective immediately. Be sure to test any change you want to make before you enable the change in production because it might break scripts on which you or applications depend. Execution policy is a server-specific setting. However, its setting is recorded in the system registry, and it is possible to use Group Policy to apply the same setting to every server within the organization. To do this, configure Group Policy to set the value of ExecutionPolicy to the desired execution mode. The key is located under:

```
HKLM\Software\Microsoft\PowerShell\1\ShellIds\Microsoft\PowerShell
```

Note that because the setting for the execution policy is held in the system registry, Windows will deny any attempt to update the value unless your account has the privilege to change the system registry.

Profiles

When you start EMS, PowerShell runs a script called Bin\RemoteExchange.ps1 to initialize EMS by loading the Exchange snap-in and defining a set of variables that EMS uses, such as the default scope for Active Directory queries. The script also prints some welcome information for EMS.

If you use EMS frequently, consider creating a profile EMS can load when it initializes a new session. If it finds a profile, PowerShell executes the commands in it before it runs

Exchange ps1 to create the EMS session. This order ensures that you can't interfere with the creation of the EMS session.

I like profiles because they remind me of the convoluted logon command procedures I used to create for OpenVMS. Typical examples of commands included in profiles are the following:

- Define some aliases (shorthand for commands). For example, you could use Set-Alias gmbx Get-Mailbox to use gmbx any time you want to run the Get-Mailbox cmdlet.
- Add one or more directories containing scripts to the path, as discussed earlier.
- Position your session in a specific directory in which you prefer to work.

PowerShell defines a global variable called $Profile to hold the location of your profile. The exact location varies across different versions of Windows. The profile doesn't exist by default, and you might have to create it before you can edit it to add some commands. First, see whether a profile is available for the account you use:

```
Test-Path $Profile
```

If the response is $True, you know that a profile exists. If not, you have to create it with:

```
New-Item –Path $Profile –Type File –Force
```

After you have a profile, you can edit it as follows:

```
Notepad $Profile
```

Here's a simple profile that you could begin with:

```
$env:path = $env:path + ";c:\Scripts"
'You are now entering PowerShell: ' + $env:Username
$StartTime = (Get-Date)
Write-Host "Session starting at $StartTime"
Set-Location c:\temp
```

After you finish updating the profile, save the file and restart EMS to see whether your changes are effective. There are endless possibilities for inventive code to run within a profile.

Active Directory for PowerShell

Active Directory is a huge dependency for Exchange, and it makes a lot of sense to be able to manage Active Directory through PowerShell. This was not always possible, but on Windows Server 2008 R2 SP1 servers, all you need to do is load the Active Directory Module for PowerShell that's installed under Administrative Tools. Assuming that the Active

Directory module is available on a server or client, you can load it into any PowerShell session by using the following command:

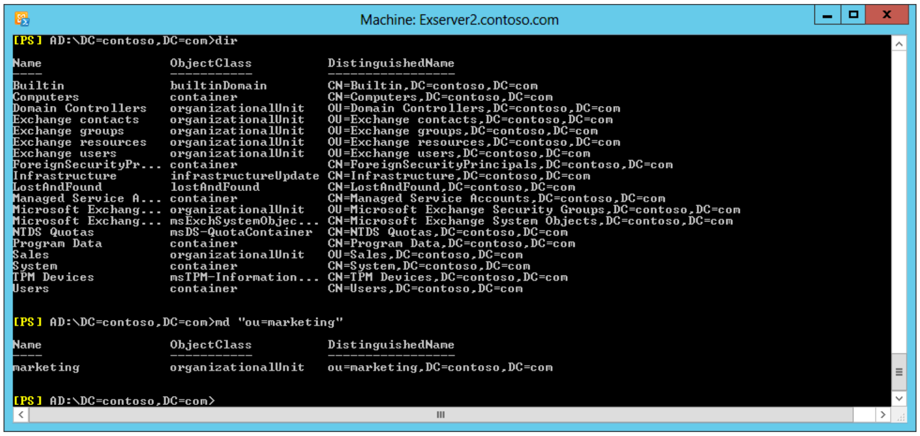
```
PS C:\> Import-Module ActiveDirectory
```

The Active Directory module is loaded automatically into EMS on Windows 2012 servers, so you can execute commands against Active Directory data immediately. To get a list of the Active Directory cmdlets, type:

```
PS C:\> Get-Help *-AD*
```

In terms of navigation through the directory structure, Active Directory is represented to PowerShell like files on a hard drive that is referenced as the AD: drive. If your system is joined to a domain, you can then navigate Active Directory. For example, here's how to create a new organizational unit (OU) called Marketing after navigating to the desired location in Active Directory. You can see the same in Figure 3-6.

```
PS C:\> CD AD:
PS AD:\> CD "DC=contoso,DC=com"
PS AD:\DC=contoso, DC=com> MD "OU=Marketing"
```



```
[PS] AD:\DC=contoso,DC=com>dir

Name                 ObjectClass          DistinguishedName
----                 -----------          -----------------
Builtin              builtinDomain        CN=Builtin,DC=contoso,DC=com
Computers            container            CN=Computers,DC=contoso,DC=com
Domain Controllers   organizationalUnit   OU=Domain Controllers,DC=contoso,DC=com
Exchange contacts    organizationalUnit   OU=Exchange contacts,DC=contoso,DC=com
Exchange groups      organizationalUnit   OU=Exchange groups,DC=contoso,DC=com
Exchange resources   organizationalUnit   OU=Exchange resources,DC=contoso,DC=com
Exchange users       organizationalUnit   OU=Exchange users,DC=contoso,DC=com
ForeignSecurityPr... container            CN=ForeignSecurityPrincipals,DC=contoso,DC=com
Infrastructure       infrastructureUpdate CN=Infrastructure,DC=contoso,DC=com
LostAndFound         lostAndFound         CN=LostAndFound,DC=contoso,DC=com
Managed Service A... container            CN=Managed Service Accounts,DC=contoso,DC=com
Microsoft Exchang... organizationalUnit   OU=Microsoft Exchange Security Groups,DC=contoso,DC=com
Microsoft Exchang... msExchSystemObjec... CN=Microsoft Exchange System Objects,DC=contoso,DC=com
NTDS Quotas          msDS-QuotaContainer  CN=NTDS Quotas,DC=contoso,DC=com
Program Data         container            CN=Program Data,DC=contoso,DC=com
Sales                organizationalUnit   OU=Sales,DC=contoso,DC=com
System               container            CN=System,DC=contoso,DC=com
TPM Devices          msTPM-Information... CN=TPM Devices,DC=contoso,DC=com
Users                container            CN=Users,DC=contoso,DC=com


[PS] AD:\DC=contoso,DC=com>md "ou=marketing"

Name                 ObjectClass          DistinguishedName
----                 -----------          -----------------
marketing            organizationalUnit   ou=marketing,DC=contoso,DC=com


[PS] AD:\DC=contoso,DC=com>
```

Figure 3-6 Creating a new OU in Active Directory

To compare how much easier it is to access Active Directory data by using the new module, the command to retrieve a list of domain controllers is:

```
PS C:\> Get-ADDomainController  | Format-Table Name, OperatingSystem
```

```
Name                       Operatingsystem
----                       ---------------
CONTOSO-DC07               Windows Server 2012 Standard
```

```
CONTOSO-DC01                 Windows Server 2012 Standard
CONTOSO-DC02                 Windows Server 2008 R2 Enterprise
```

Another useful example is when you want to scan for inactive Active Directory accounts so that you can clean up the directory. In this command, you scan for any account that has not been logged on to in the past 120 days and report the account name and the date the user last logged on.

```
Search-ADAccount –UsersOnly –AccountInActive –TimeSpan 120 | Format-Table Name,
LastLogonDate
```

You could then disable these accounts by piping the discovered list to the Disable-ADAccount cmdlet. However, this is a dangerous thing to do in an Exchange environment because so many accounts are never logged on to because they are used for purposes such as room and discovery mailboxes.

Another one-liner that is extremely useful on test systems searches for all Active Directory accounts that have an email address and sets the accounts so that the passwords never expire. This gets rid of a lot of annoying prompts you might otherwise encounter because passwords expire!

```
Get-ADUser –Filter {EmailAddress –Like "*@contoso.com"}  | Set-ADUser
–PasswordNeverExpires $True
```

See *http://technet.microsoft.com/en-us/library/dd378937(v=ws.10).aspx* for information about how to perform Active Directory management by using PowerShell for Windows 2008 R2.

Setting the right scope for objects in a multi-domain forest

When you start EMS, Exchange sets the default scope for queries performed against Active Directory to the domain to which the server belongs. This is fine if you operate a single-domain forest, but it is definitely not if you have to manage objects in a multi-domain forest because it means that any query you perform will return only objects from the local domain. To control the scope for Active Directory objects, use the Set-ADServerSettings cmdlet. Set the ViewEntireForest parameter to be $True (to see the entire forest) or $False (to see just the objects owned by the default domain). The logical place to do this is in your personal PowerShell profile. For example:

```
Set-ADServerSettings -ViewEntireForest $True
```

You can also use this command to point to a particular domain controller to retrieve Active Directory data. For example:

```
Set-ADServerSettings –PreferredServer 'DC1.contoso.com'
```

If you do not want to set your scope to the entire forest, a partial workaround is to specify a global catalog server in the remote domain to use for the query. Another way of forcing EMS to operate on a forest-wide basis is to specify the –IgnoreDefaultScope parameter for cmdlets such as Get-Mailbox. This parameter tells EMS to ignore the default recipient scope setting for EAC (typically the domain into which a server is installed) and use the entire forest instead. For example, if you wanted to set up a batch of mailboxes to move from an Exchange 2007 server to Exchange 2013 that used accounts in multiple domains, you could use a command like this:

```
Get-Mailbox –Server 'Exchange2007' –ResultSize Unlimited –IgnoreDefaultScope |
New-MoveRequest -TargetDatabase 'Mailbox Database 1002' –BatchName 'Move Group from
Exchange 2007'
```

The natural question at this point is whether changing the scope for Active Directory queries will affect how you work with EMS. The answer is yes because when you set a forest-wide scope, EMS fetches data from across the forest rather than from the local domain. Unless you use parameters to focus on particular groups of objects, such as specifying that you want to work with the mailboxes from one server, you will probably have to wait longer for a response. This is because you will ask EMS to process cmdlets that deal with servers, mailboxes, databases, or other objects across a complete forest rather than with just one domain, but in most cases, the wait is worthwhile because you see the complete picture and do not run the risk of missing something.

Exploring useful EMS examples

A scan of the Internet results in many interesting EMS code snippets that can be usefully employed by an Exchange administrator. This section discusses some good examples. The idea is not to present complete solutions. Rather, I hope to inspire you to experiment with EMS to see just how much value you can get from a few lines of reasonably straightforward code. After all, if you can do a lot of work in a couple of lines that take just a few minutes to type in and get running, think of how much you can do if you really set your mind to exploiting EMS!

Before reviewing the examples of EMS in use, I have two specific pieces of advice for the aspiring EMS aficionado. Because this book is emphatically not designed to be a reference guide for EMS, if you think that you will become heavily involved with EMS, purchase a copy of *Microsoft Exchange 2013 PowerShell Cookbook*, Second edition (Packt Publishing, 2013). The book is packed full of guidance, tips, and programming examples that are extremely useful for both on-premises and Exchange Online administrators.

Second, many of the Exchange MVPs provide an extremely valuable service to the Exchange community by publishing what become de facto standards for how to write a script to solve certain problems. You should download these scripts and use them as a starting point for understanding just how to approach writing industrial-strength EMS code. I would also bookmark their websites and keep up to date with their activities so that you can learn from their future work. At the risk of offending others, among my favorite sites are:

- **Pat Richard (*http://www.ehloworld.com/*)** Features a great script (Send-NewUserWelcome.ps1) that shows how to build a welcome message to new Exchange users on a scheduled basis.
- **Andy Grogan (*http://www.telnetport25.com*)** Look at his script for automating the setup of an Exchange lab environment.
- **Mike Crowley (*http://mikecrowley.wordpress.com/*)** Contains a nice script to report on the proxy addresses assigned to email users.
- **Steve Goodman (*http://www.stevieg.org/*)** Shows an extremely useful Exchange environment report, a comprehensive overview of lots of information about your Exchange organization; output in HTML format.
- **Paul Cunningham (*http://exchangeserverpro.com/*)** Offers the best mailbox report script around (Get-MailboxReport.ps1). Paul also maintains a nice server health monitoring script that generates and sends an HTML format message to administrators on a regular basis.

These scripts can be downloaded from these sites; the code is fully revealed and is easily adapted to meet any particular needs that exist in your environment. New sites that feature great tips appear all the time, and I'm sure you will accumulate your own list of go-to people you consult when you meet a problem. In the meantime, look at some examples to get started with EMS.

Looking for large folders

The first example shows how to discover users who might be suffering from performance problems because they have very large folders in their mailboxes. The number of items that is considered bad has grown over time in line with the updates Microsoft has made to tune the database schema. With Exchange 2000 or Exchange 2003, the danger mark is around 5,000 items. The threshold increases to 20,000 with Exchange 2007 and leaps to 100,000 for Exchange 2010 onward. The client used is also important because Outlook 2010 and Outlook 2013 are better at dealing with large folders than Outlook 2007 is. Having more than 20,000 items in a folder is evidence of solid pack-rat behavior by anyone, and it marks a folder that probably will never be cleaned out simply because it takes too much effort to explore the contents and decide what should be kept and what should be deleted. Assume that you want to flag potential issues to users who have more than 5,000 items in a folder. You can use code like this:

```
Get-Mailbox –Server ExServer2 | Get-MailboxFolderStatistics | Where {$_.ItemsInFolder
–GT 5000} | Sort ItemsInFolder –Descending | Format-Table Identity, ItemsInFolder
–AutoSize
```

```
Identity                                                    ItemsInFolder
---------                                                   -----------------
contoso.com/Exchange Users/Redmond, Eoin\I                  5271
contoso.com/Exchange Users/Ruth, Andy\Inbox                 5265
contoso.com/Exchange Users/Andrews, Ben\Inbox               5263
contoso.com/Exchange Users/Pelton, David\Inbox              5230
contoso.com/Exchange Users/Simpson, David\Inbox             5218
contoso.com/Exchange Users/Redmond, Tony\Sent Items         5215
```

Of course, it would be impolite to send a note to these users to remind them that good filing practices lead to clean mailboxes, but you can still think about it!

This code does the following:

- Calls Get-Mailbox to generate a list of all mailboxes located on databases hosted by a server. It is possible to process all mailboxes in an organization by changing the code to Get-Mailbox –ResultSize Unlimited, but such a command will take a long time to process in any organization with more than a couple of thousand mailboxes (though you could use a server-side filter when appropriate).
- Calls Get-MailboxFolderStatistics to extract a count of items in each folder.
- Filters any folder with more than 5,000 items.
- Sorts the filtered folders by descending order.
- Outputs the information.

If you run this command against an Exchange 2010 or Exchange 2013 server, even details of the folders in the dumpster (for example, Deletions) will be shown that are not reported by an Exchange 2007 server.

Outputting a CSV file

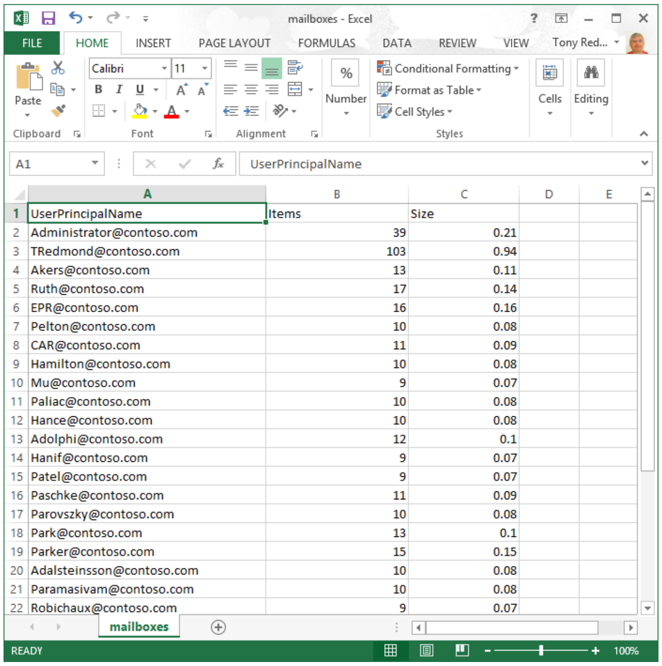
Many examples of outputting CSV files from Exchange data use the Export-CSV cmdlet. For instance, here's a two-line script that looks for any mailbox that has an ActiveSync partnership created, which indicates that the user has connected a mobile device to the mailbox by using ActiveSync. An expression is included to force a call to the Get-ActiveSyncDevice cmdlet to retrieve the count of devices associated with each user. This kind of information is useful when understanding how many people actually connect mobile devices to Exchange!

```
$Mbx = Get-CASMailbox –Filter {HasActiveSyncDevicePartnership –eq $True} |
Get-Mailbox
$Mbx | Select DisplayName, UserPrincipalName, @{Name="Devices";Expression=
{(Get-ActiveSyncDevice –Mailbox $_.Identity).Count)} | Export-CSV
"c:\temp\ActiveSync.csv" –NoTypeInformation
```

Export-CSV is great because it takes care of all the formatting issues required to create a valid CSV file that will be recognized by applications such as Excel. However, there are other ways to generate CSV data. This script creates a CSV file you can use to analyze mailbox usage (Figure 3-7). A check in the code limits processing to the mailboxes found in a specific database and ignores anything but user mailboxes. (Objects such as room or arbitration mailboxes are ignored.) This script could take quite a while to finish if there are more than a few hundred mailboxes in the selected database, so be sure to test it on perhaps a smaller group before you launch it to process larger collections.

```
$Outputfile = "C:\temp\Mailboxes.csv"
Out-File -FilePath $OutputFile -InputObject "UserPrincipalName, Items, Mailbox Size"
-Encoding UTF8
$mbx = Get-Mailbox –Database DB2
Foreach ($M in $Mbx)
   {

   if ($M.RecipientTypeDetails -eq "UserMailbox")
        {

        # Fetch information about the mailbox
        $UserMailbox = Get-Mailboxstatistics -Identity $($M.Identity)

       $UserPrincipalName = $M.UserPrincipalName
       $ItemSizeString = $UserMailbox.TotalItemSize.ToString()
       $MailboxSize = "{0:N2}" -f ($ItemSizeString.SubString(($ItemSizeString.
IndexOf("(") + 1),($itemSizeString.IndexOf(" bytes") - ($ItemSizeString.IndexOf("(")
+ 1))).Replace(",","")/1024/1024)
       $ItemCount = $UserMailbox.ItemCount

    #Prepare the user details in CSV format for writing to file and append line
      $UserDetails = $UserPrincipalName + "," + $ItemCount + "," + $MailboxSize
      Out-File -FilePath $OutputFile -InputObject $UserDetails -Encoding UTF8 -append
        }
   }
```

This script generates fairly basic data about mailboxes, and if you scan the Internet, you can find many other approaches to the problem of mailbox reporting, some of which are much better than others. With anything to do with mailboxes, the key is speed because code that is quite good at processing one or two mailboxes might not be smart when confronted with a few thousand. It's also a good idea to consider what information needs to be output and make sure that the data reported is formatted in a way that is most useful to the reader.



	A	B	C	D	E
1	UserPrincipalName	Items	Size		
2	Administrator@contoso.com	39	0.21		
3	TRedmond@contoso.com	103	0.94		
4	Akers@contoso.com	13	0.11		
5	Ruth@contoso.com	17	0.14		
6	EPR@contoso.com	16	0.16		
7	Pelton@contoso.com	10	0.08		
8	CAR@contoso.com	11	0.09		
9	Hamilton@contoso.com	10	0.08		
10	Mu@contoso.com	9	0.07		
11	Paliac@contoso.com	10	0.08		
12	Hance@contoso.com	10	0.08		
13	Adolphi@contoso.com	12	0.1		
14	Hanif@contoso.com	9	0.07		
15	Patel@contoso.com	9	0.07		
16	Paschke@contoso.com	11	0.09		
17	Parovszky@contoso.com	10	0.08		
18	Park@contoso.com	13	0.1		
19	Parker@contoso.com	15	0.15		
20	Adalsteinsson@contoso.com	10	0.08		
21	Paramasivam@contoso.com	10	0.08		
22	Robichaux@contoso.com	9	0.07		

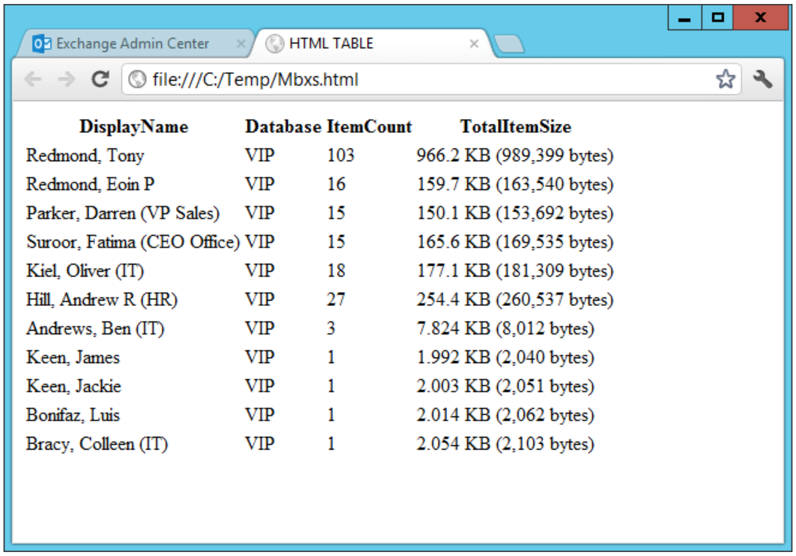
Figure 3-7 User mailbox CSV data

Creating a report in HTML

PowerShell is flexible in terms of processing output. Generated reports can show management and others the kind of work that servers do. The typical reports EMS generates are plaintext. You can also generate HTML reports by piping objects through the ConvertTo-HTML cmdlet. (The Out-HTML cmdlet at *http://poshcode.org/1612* is also useful for generating HTML content.) This example explores how to generate a useful report that shows mailboxes that have exceeded their storage quota. You could use a report like this to check proactively for users who are experiencing problems with their quota and perhaps allocate them some additional quota to enable them to resume working. The output is shown in Figure 3-8.

```
Get-Mailbox -Database VIP | Get-MailboxStatistics | Sort TotalItemSize -Descending |
ConvertTo-HTML DisplayName, Database, ItemCount, TotalItemSize > C:\Temp\Mbxs.html
```

You can enhance the output further by formatting the HTML with a style sheet or adding other information such as the date and time of the report. I leave that as an exercise for the reader.



DisplayName	Database	ItemCount	TotalItemSize
Redmond, Tony	VIP	103	966.2 KB (989,399 bytes)
Redmond, Eoin P	VIP	16	159.7 KB (163,540 bytes)
Parker, Darren (VP Sales)	VIP	15	150.1 KB (153,692 bytes)
Suroor, Fatima (CEO Office)	VIP	15	165.6 KB (169,535 bytes)
Kiel, Oliver (IT)	VIP	18	177.1 KB (181,309 bytes)
Hill, Andrew R (HR)	VIP	27	254.4 KB (260,537 bytes)
Andrews, Ben (IT)	VIP	3	7.824 KB (8,012 bytes)
Keen, James	VIP	1	1.992 KB (2,040 bytes)
Keen, Jackie	VIP	1	2.003 KB (2,051 bytes)
Bonifaz, Luis	VIP	1	2.014 KB (2,062 bytes)
Bracy, Colleen (IT)	VIP	1	2.054 KB (2,103 bytes)

Figure 3-8 Viewing the HTML version of the mailbox report

It's worth noting that when you run the Get-MailboxStatistics cmdlet, you force EMS to make a remote procedure call (RPC) to the Information Store to retrieve the latest data for the mailboxes (individual, database, or server). The information is completely up to date and reflects the exact state of the mailbox rather than cached data that could be a couple of hours old. The Store caches information about mailbox quotas and updates the cache every two hours to avoid the overhead of the I/O that it would otherwise need to generate to check quotas every time a user attempts to send a message or to check that a mailbox can accept a new message.

TROUBLESHOOTING

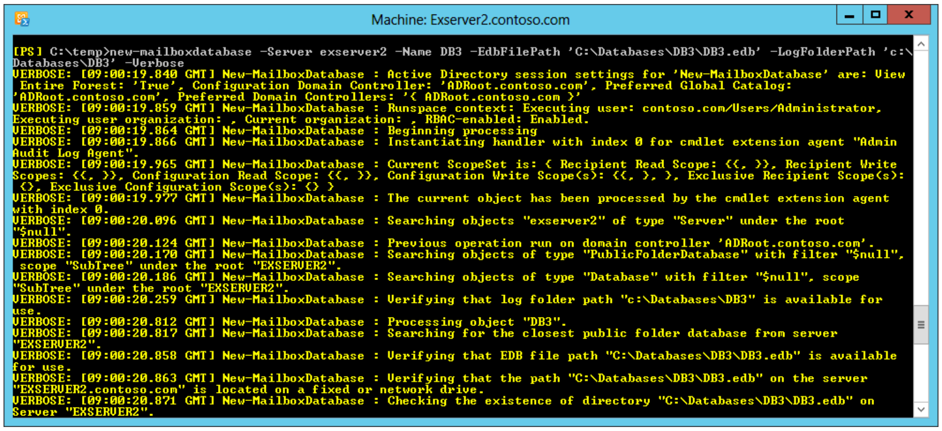
Users report that they've deleted messages but still exceed quota

Given the dynamic flow of messages in and out of mailboxes, it's likely that a small difference exists between the cached data and the actual state. This sometimes causes confusion when a user reports that she has exceeded quota and can't send mail even though she has deleted many messages, and she has to wait until the Store refreshes its cache to determine the new mailbox size and respect the fact that she has reduced the size under quota. If this becomes a problem and users complain that Exchange takes too long before it allows them to resume email activity, you can amend the system registry to force Exchange to refresh the cache more often with the caveat that more frequent refreshes impose an extra overhead on the server. See *http://technet.microsoft.com/en-us/library/aa996988(EXCHG.80).aspx* for details.

Verbose PowerShell

Usually, EMS gets on with whatever you ask it to do and doesn't give any indication of the processing it performs in the background. You ask for a new mailbox to be created, and it's created, or some problem occurs that stops the command from executing. If the problem originates with an error introduced by the user, such as an error in syntax or attempting to do something that doesn't make sense, such as creating a mailbox in a database that doesn't exist, you can just fix the problem and try again.

Sometimes you need to know exactly what EMS does to help track down a problem, perhaps to provide information to Microsoft support to help them figure out what's going on in your Exchange deployment. You might just want to know what's happening when you execute a command. In either case, you can add the /verbose switch to a command to have PowerShell generate details of exactly what it does as it proceeds. Figure 3-9 shows some of the output when the New-MailboxDatabase cmdlet is used to create a new mailbox database. You can see how EMS validates the context within which it is executing, including checks to locate a global catalog server, validate RBAC authorization, and confirm that the mailbox database doesn't already exist.



```
[PS] C:\temp>new-mailboxdatabase -Server exserver2 -Name DB3 -EdbFilePath 'C:\Databases\DB3\DB3.edb' -LogFolderPath 'c:\
Databases\DB3' -Verbose
VERBOSE: [09:00:19.840 GMT] New-MailboxDatabase : Active Directory session settings for 'New-MailboxDatabase' are: View
 Entire Forest: 'True', Configuration Domain Controller: 'ADRoot.contoso.com', Preferred Global Catalog:
'ADRoot.contoso.com', Preferred Domain Controllers: '{ ADRoot.contoso.com }'
VERBOSE: [09:00:19.859 GMT] New-MailboxDatabase : Runspace context: Executing user: contoso.com/Users/Administrator,
Executing user organization: , Current organization: , RBAC-enabled: Enabled.
VERBOSE: [09:00:19.864 GMT] New-MailboxDatabase : Beginning processing
VERBOSE: [09:00:19.866 GMT] New-MailboxDatabase : Instantiating handler with index 0 for cmdlet extension agent "Admin
Audit Log Agent".
VERBOSE: [09:00:19.965 GMT] New-MailboxDatabase : Current ScopeSet is: { Recipient Read Scope: {{, }}, Recipient Write
Scopes: {{, }}, Configuration Read Scope: {{, }}, Configuration Write Scope(s): {{, }, }, Exclusive Recipient Scope(s):
 {}, Exclusive Configuration Scope(s): {} }
VERBOSE: [09:00:19.977 GMT] New-MailboxDatabase : The current object has been processed by the cmdlet extension agent
with index 0.
VERBOSE: [09:00:20.096 GMT] New-MailboxDatabase : Searching objects "exserver2" of type "Server" under the root
"$null".
VERBOSE: [09:00:20.124 GMT] New-MailboxDatabase : Previous operation run on domain controller 'ADRoot.contoso.com'.
VERBOSE: [09:00:20.170 GMT] New-MailboxDatabase : Searching objects of type "PublicFolderDatabase" with filter "$null",
 scope "SubTree" under the root "EXSERVER2".
VERBOSE: [09:00:20.186 GMT] New-MailboxDatabase : Searching objects of type "Database" with filter "$null", scope
"SubTree" under the root "EXSERVER2".
VERBOSE: [09:00:20.259 GMT] New-MailboxDatabase : Verifying that log folder path "c:\Databases\DB3" is available for
use.
VERBOSE: [09:00:20.812 GMT] New-MailboxDatabase : Processing object "DB3".
VERBOSE: [09:00:20.817 GMT] New-MailboxDatabase : Searching for the closest public folder database from server
"EXSERVER2".
VERBOSE: [09:00:20.858 GMT] New-MailboxDatabase : Verifying that EDB file path "C:\Databases\DB3\DB3.edb" is available
for use.
VERBOSE: [09:00:20.863 GMT] New-MailboxDatabase : Verifying that the path "C:\Databases\DB3\DB3.edb" on the server
"EXSERVER2.contoso.com" is located on a fixed or network drive.
VERBOSE: [09:00:20.871 GMT] New-MailboxDatabase : Checking the existence of directory "C:\Databases\DB3\DB3.edb" on
Server "EXSERVER2".
```

Figure 3-9 Examining some verbose PowerShell output

Controlling access to Exchange

EMS is a great way to get work done with Exchange as long as you don't mind grappling with the command-line interface. If no control were exerted, you could do massive damage to an Exchange organization with EMS, such as selecting all the mailboxes in a database and removing them with a single line of code. Only the people who need to control the full scope of the organization should be able to take such drastic action. Traditionally, control is given through permissions and privileges. Exchange takes a different approach and adopts the RBAC model. All administrators need a solid grounding in RBAC and its implementation in Exchange, and that's the next subject of discussion.

Index