Microsoft

Building Cloud Apps with Microsoft Azure

Best practices for DevOps, data storage, high availability, and more

Professional

Scott Guthrie, Mark Simms, Tom Dykstra,
Rick Anderson, Mike Wasson

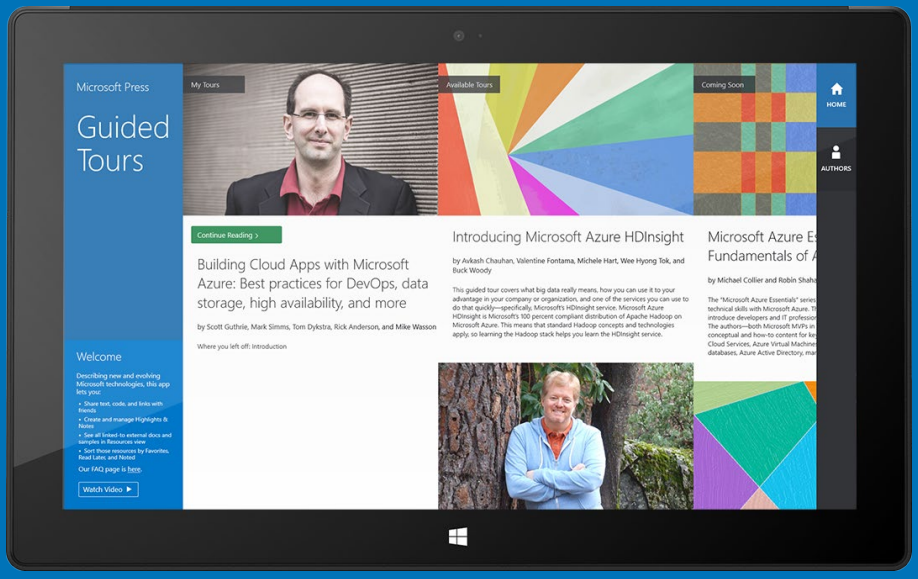
Visit us today at

microsoftpressstore.com

- **Hundreds of titles available** – Books, eBooks, and online resources from industry experts
- **Free U.S. shipping**
- **eBooks in multiple formats** – Read on your computer, tablet, mobile device, or e-reader
- **Print & eBook Best Value Packs**
- **eBook Deal of the Week** – Save up to 60% on featured titles
- **Newsletter and special offers** – Be the first to hear about new releases, specials, and more
- **Register your book** – Get additional benefits

Microsoft

Wait, there's more...



Find more great content and resources in the **Microsoft Press Guided Tours** app.

The Microsoft Press Guided Tours app provides insightful tours by Microsoft Press authors of new and evolving Microsoft technologies.

- Share text, code, illustrations, videos, and links with peers and friends
- Create and manage highlights and notes
- View resources and download code samples
- Tag resources as favorites or to read later
- Watch explanatory videos
- Copy complete code listings and scripts

Download from Windows Store

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

ISBN: 978-0-7356-9565-8

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

Acquisitions, Developmental, and Project Editor: Devon Musgrave
Editorial Production: Flyingspress and Rob Nance
Copyeditor: John Pierce
Cover: Twist Creative • Seattle and Joel Panchot

Table of Contents

Introduction

This ebook walks you through a patterns-based approach to building real-world cloud solutions. The patterns apply to the development process as well as to architecture and coding practices.

The content is based on a presentation developed by Scott Guthrie and delivered by him at the Norwegian Developers Conference (NDC) in June of 2013 (part 1, part 2), and at Microsoft Tech Ed Australia in September 2013 (part 1, part 2). Many others updated and augmented the content while transitioning it from video to written form.

Who should read this book

Developers who are curious about developing for the cloud, are considering a move to the cloud, or are new to cloud development will find here a concise overview of the most important concepts and practices they need to know. The concepts are illustrated with concrete examples, and each chapter includes links to other resources that provide more in-depth information. The examples and the links to additional resources are for Microsoft frameworks and services, but the principles illustrated apply to other web development frameworks and cloud environments as well.

Developers who are already developing for the cloud may find ideas here that will help make them more successful. Each chapter in the series can be read independently, so you can pick and choose topics that you're interested in.

Anyone who watched Scott Guthrie's "Building Real World Cloud Apps with Windows Azure" presentation and wants more details and updated information will find that here.

Assumptions

This book expects that you have experience developing web applications by using Visual Studio and ASP.NET. Familiarity with C# would be helpful in places.

This book might not be for you if...

This book might not be for you if you are looking for information specific to cloud environments other than Microsoft Azure.

Organization of this book

This ebook explains thirteen recommended patterns for cloud development. "Pattern" is used here in a broad sense to mean a recommended way to do things: how best to go about developing, designing, and coding cloud apps. These are key patterns that will help you "fall into the pit of success" if you follow them.

- Automate everything
  - Use scripts to maximize efficiency and minimize errors in repetitive processes.
  - Demo: Azure management scripts.
- Source control
  - Set up branching structures in source control to facilitate a DevOps workflow.
  - Demo: add scripts to source control.
  - Demo: keep sensitive data out of source control.
  - Demo: use Git in Visual Studio.
- Continuous integration and delivery
  - Automate build and deployment with each source control check-in.
- Web development best practices
  - Keep web tier stateless
  - Demo: scaling and autoscaling in Azure Websites.
  - Avoid session state.
  - Use a Content Delivery Network (CDN).
  - Use an asynchronous programming model.
  - Demo: async in ASP.NET MVC and Entity Framework.
- Single sign-on
  - Introduction to Azure Active Directory.
  - Demo: create an ASP.NET app that uses Azure Active Directory.

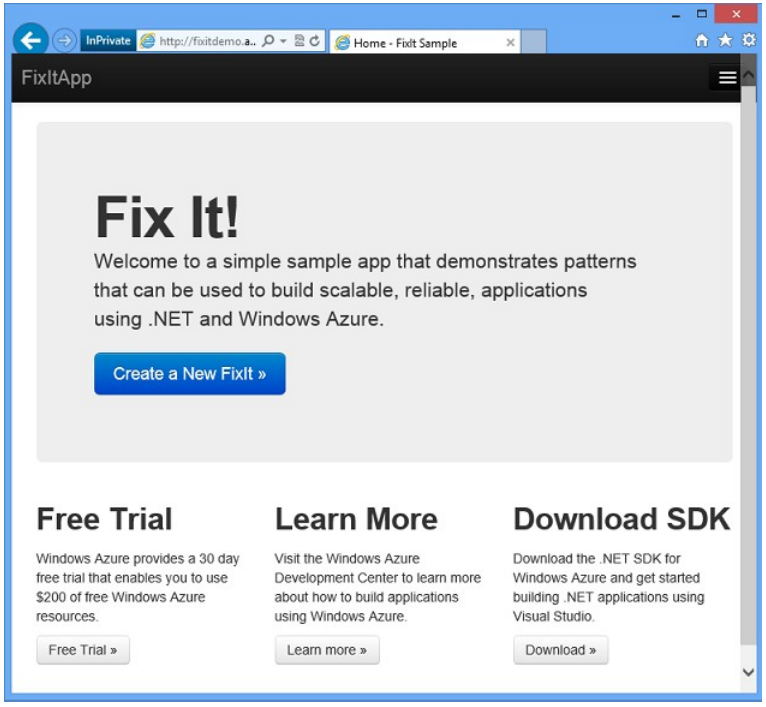
- Data storage options
  - Types of data stores.
  - How to choose the right data store.
  - Demo: Azure SQL Database.
- Data partitioning strategies
  - Partition data vertically, horizontally, or both to facilitate scaling a relational database.
- Unstructured blob storage
  - Store files in the cloud by using the Blob service.
  - Demo: using blob storage in the Fix It app.
- Design to survive failures
  - Types of failures.
  - Failure scope.
  - Understanding SLAs.
- Monitoring and telemetry
  - Why you should both buy a telemetry app and write your own code to instrument your app.
  - Demo: New Relic for Azure
  - Demo: logging code in the Fix It app.
  - Demo: built-in logging support in Azure.
- Transient fault handling
  - Use smart retry/back-off logic to mitigate the effect of transient failures.
  - Demo: retry/back-off in Entity Framework 6.
- Distributed caching
  - Improve scalability and reduce database transaction costs by using distributed caching.
- Queue-centric work pattern
  - Enable high availability and improve scalability by loosely coupling web and worker tiers.
  - Demo: Azure storage queues in the Fix It app.

- More cloud app patterns and guidance
- Appendix: The Fix It Sample Application
    - Known issues.
    - Best practices.
    - Download, build, run, and deploy instructions.

These patterns apply to all cloud environments, but we'll illustrate them by using examples based on Microsoft technologies and services, such as Visual Studio, Team Foundation Service, ASP.NET, and Azure.

The Fix It sample application

Most of the screen shots and code examples shown in this ebook are based on the Fix It app originally developed by Scott Guthrie to demonstrate recommended cloud app development patterns and practices.

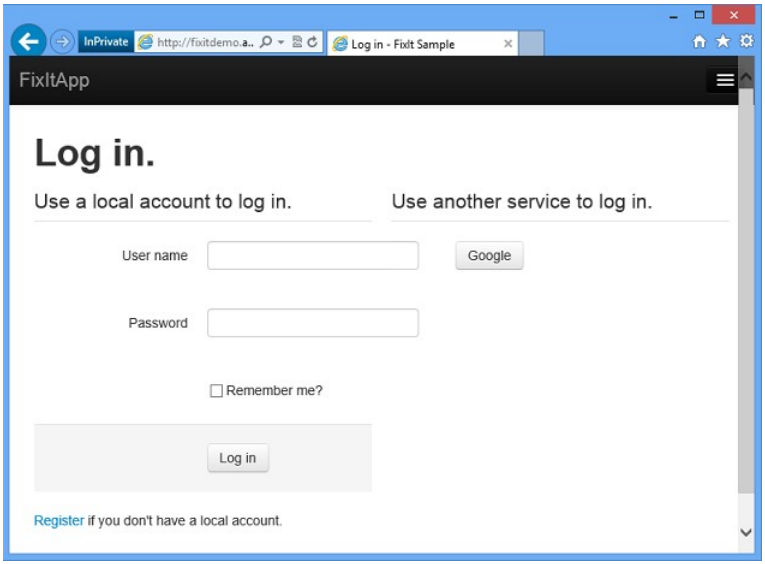


The sample app is a simple work item ticketing system. When you need something fixed, you create a ticket and assign it to someone, and others can log in and see the tickets assigned to them and mark

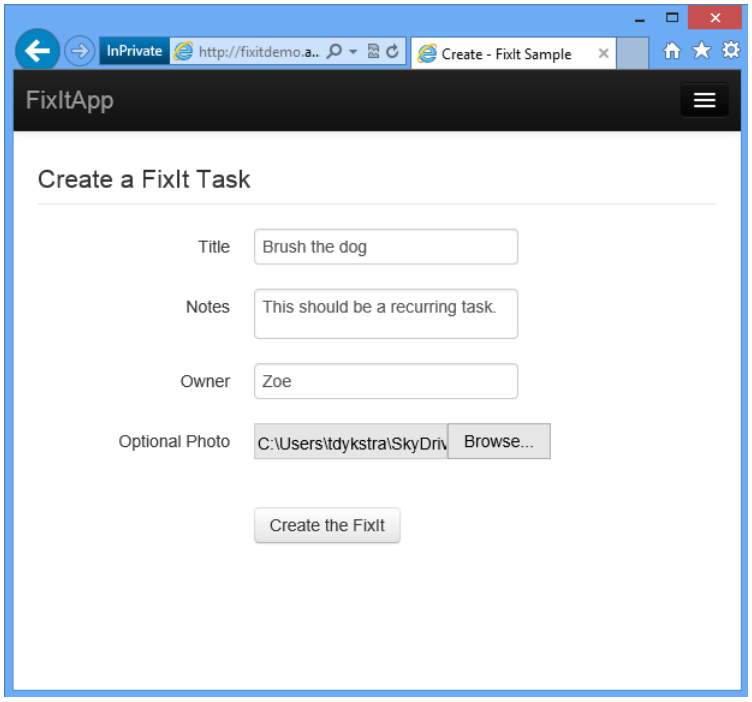
tickets as completed when the work is done.

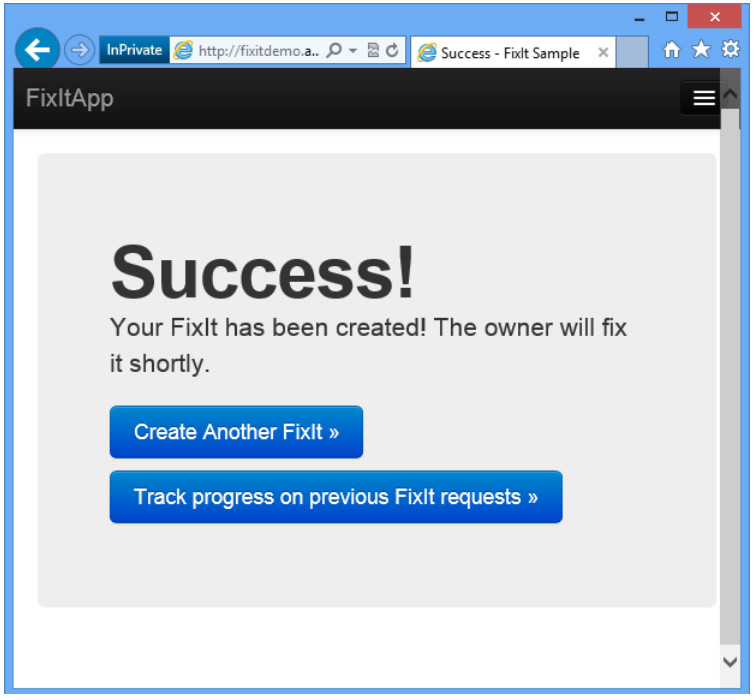
The Fix It app is a standard Visual Studio web project. It is built on ASP.NET MVC and uses a SQL Server database. It can run locally in IIS Express and can be deployed to an Azure website to run in the cloud.

You can log in using forms authentication and a local database or by using a social provider such as Google. (Later we'll also show how to log in with an Active Directory organizational account.)



Once you're logged in you can create a ticket, assign it to someone, and upload a picture of what you want to get fixed.


InPrivate
http://fixitdemo.a..
Create - FixIt Sample
FixItApp
Create a FixIt Task
Title
Brush the dog
Notes
This should be a recurring task.
Owner
Zoe
Optional Photo
C:\Users\tdykstra\SkyDriv
Browse...
Create the FixIt


InPrivate
http://fixitdemo.a..
Success - FixIt Sample
FixItApp
Success!
Your FixIt has been created! The owner will fix it shortly.
Create Another FixIt »
Track progress on previous FixIt requests »

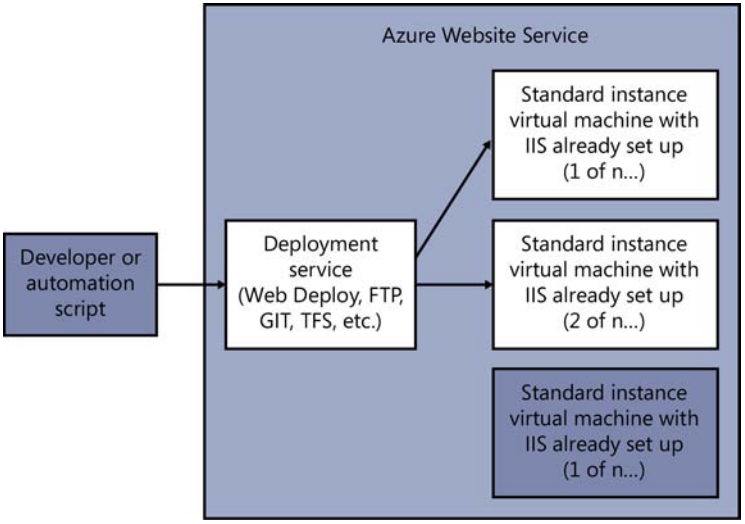
You can track the progress of work items you created, see tickets assigned to you, view ticket details, and mark items as completed.

This is a very simple app from a feature perspective, but you'll see how to build it so that it can scale to millions of users and will be resilient to things such as database failures and connection terminations. You'll also see how to create an automated and agile development workflow, which enables you to start simple and make the app better and better by iterating the development cycle efficiently and quickly.

Azure Websites

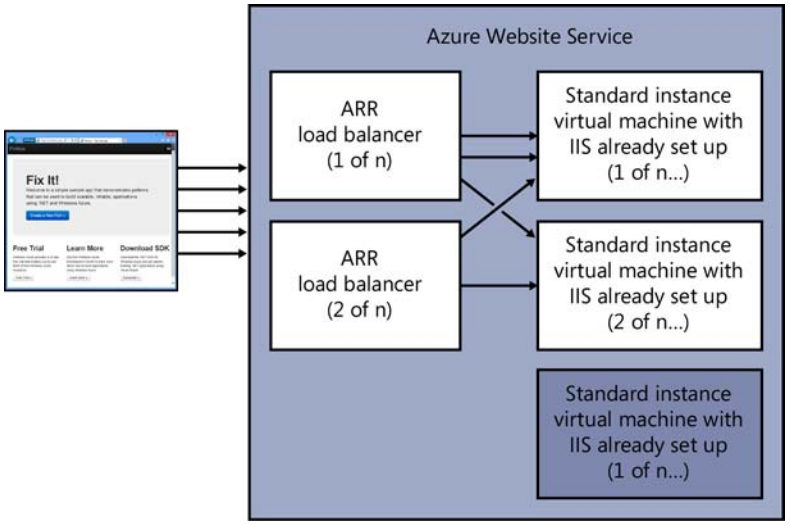
The cloud environment used for the Fix It application is a service of Azure that Microsoft calls Websites. This service is a way in which you can host your own web app in Azure without having to create VMs and keep them updated, install and configure IIS, and so on. Microsoft hosts your site on its VMs and automatically provides backup and recovery and other services for you. The Websites service works with ASP.NET, Node.js, PHP, and Python. It enables you to deploy very quickly using Visual Studio, Web Deploy, FTP, Git, or TFS. It's usually just a few seconds between the time you start a deployment and the time your update is available over the Internet. It's all free to get started, and you can scale up as your traffic grows.

Behind the scenes the Azure Websites service provides a number of architectural components and features that you'd have to build yourself if you were going to host a website using IIS on your own VMs. One component is a deployment end point that automatically configures IIS and installs your application on as many VMs as you want to run your site on.

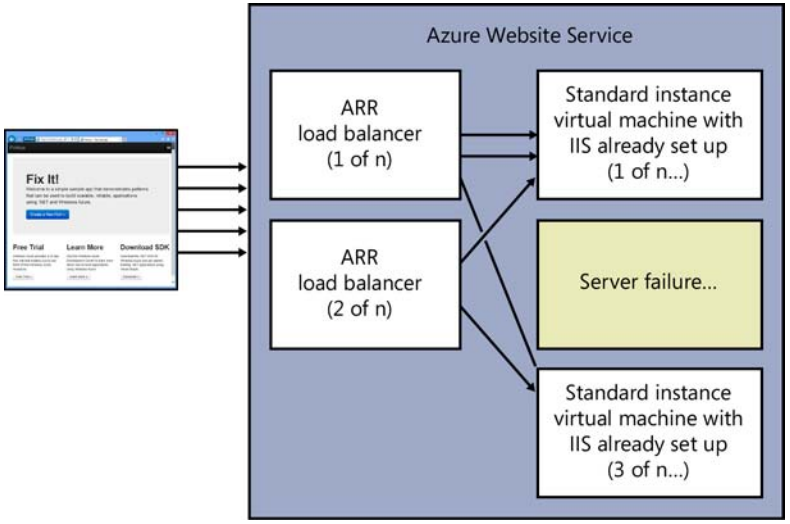


When a user hits the website, they don't hit the IIS VMs directly, they go through Application

Request Routing (ARR) load balancers. You can use these with your own servers, but the advantage here is that they're set up for you automatically. They use a smart heuristic that takes into account factors such as session affinity, queue depth in IIS, and CPU usage on each machine to direct traffic to the VMs that host your website.



If a machine goes down, Azure automatically pulls it from the rotation, spins up a new VM instance, and starts directing traffic to the new instance—all with no down time for your application.



All of this takes place automatically. All you need to do is create a website and deploy your application to it, using Windows PowerShell, Visual Studio, or the Azure management portal.

For a quick and easy step-by-step tutorial that shows how to create a web application in Visual Studio and deploy it to the Azure Websites service, see Get started with Azure Websites and ASP.NET.

For more information about the topics we've introduce here, see the following resources.

Documentation:

- Azure Websites Portal page for azure.microsoft.com documentation about Azure Websites.
- Azure Websites, Cloud Services, and Virtual Machines Comparison Azure Websites as shown in this introduction is just one of three ways you can run web apps in Azure. Read this article for guidance on how to choose which one is right for your scenario. Like Websites, Cloud Services is a Platform as a Service (PaaS) feature of Azure. VMs are an Infrastructure as a Service (IaaS) feature. For an explanation of PaaS versus IaaS, see Chapter 6, "Data storage options."

Videos:

- Scott Guthrie starts at Step 0 - What is the Azure Cloud OS?
- Websites Architecture - with Stefan Schackow.
- Windows Azure Websites Internals with Nir Mashkowski.

System requirements

You will need the following software to run the sample application that you can download:

- Windows 7, 8, or 8.1; or Windows Server 2003, 2008, or 2012
- Visual Studio 2013, any edition

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio.

Downloads: Code samples

You can download the Fix It sample application here. The download site includes an overview of the Fix It app and instructions for building the sample. You can find more detailed information about building the app, as well as known issues and best practices for using it, in the appendix.

Acknowledgments

This content was written by Tom Dykstra, Rick Anderson, and Mike Wasson. Most of the original content came from Scott Guthrie, and he in turn drew on material from Mark Simms and the Microsoft Customer Advisory Team (CAT).

Many other colleagues at Microsoft reviewed and commented on drafts and code:

- Tim Ammann reviewed the automation chapter.
- Christopher Bennage reviewed and tested the Fix It code.
- Ryan Berry reviewed the CD/CI chapter.
- Vittorio Bertocci reviewed the SSO chapter.
- Conor Cunningham reviewed the data storage options chapter.
- Carlos Farre reviewed and tested the Fix It code for security issues.
- Larry Franks reviewed the telemetry and monitoring chapter.
- Jonathan Gao reviewed Hadoop and MapReduce sections of the data storage options chapter.
- Sidney Higa reviewed all chapters.
- Gordon Hogenson reviewed the source control chapter.
- Tamra Myers reviewed data storage options, blob, and queues chapters.
- Pranav Rastogi reviewed the SSO chapter.
- June Blender Rogers added error handling and help text to the PowerShell automation scripts.
- Mani Subramanian reviewed all chapters and led the code review and testing process for the Fix It code.
- Shaun Tinline-Jones reviewed the data partitioning chapter.
- Selcin Tukarslan reviewed chapters that cover SQL Database and SQL Server.
- Edward Wu provided sample code for the SSO chapter.
- Guang Yang wrote the PowerShell automation scripts.

Members of the Microsoft Developer Advisory Council (DAC) also reviewed and commented on drafts: Jean-Luc Boucho, Catalin Gheorghiu, Wouter de Kort, Carlo dos Santos, Neil Mackenzie, Dennis Persson, Sunil Sabat, Aleksey Sinyagin, Bill Wagner, and Michael Wood.

Other members of the DAC reviewed and commented on the preliminary outline: Damir Arh, Edward Bakker, Srdjan Bozovic, Ming Man Chan, Gianni Rosa Gallina, Paulo Morgado, Jason Oliveira, Alberto Poblacion, Ryan Riley, Perez Jones Tsisah, Roger Whitehead, and Pawel Wilkosz.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book. If you discover an error, please submit it to us via mspinput@microsoft.com. You can also reach the Microsoft Press Book Support team for other support via the same alias. Please note that product support for Microsoft software and hardware is not offered through this address. For help with Microsoft software or hardware, go to http://support.microsoft.com.

Free ebooks from Microsoft Press

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

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

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

http://aka.ms/tellpress

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

Stay in touch

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

Chapter 1

Automate everything

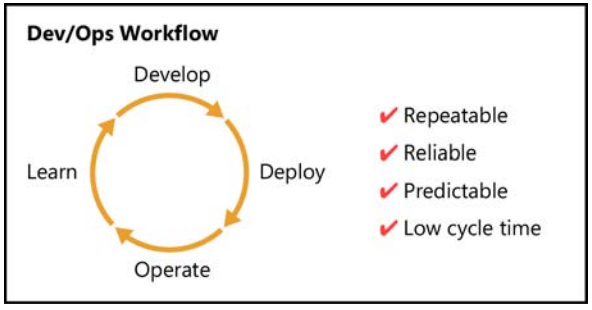
The first three patterns we'll look at actually apply to any software development project, but especially to cloud projects. The pattern we describe in this chapter is about automating development tasks. It's an important topic because manual processes are slow and prone to error; automating as many of them as possible helps set up a fast, reliable, and agile workflow. It's uniquely important for cloud development because you can easily automate many tasks that are difficult or impossible to automate in an on-premises environment. For example, you can set up whole test environments, including new web server and back-end virtual machines (VMs), databases, blob storage (file storage), queues, etc.

DevOps workflow

Increasingly, you hear the term "DevOps." The term developed out of a recognition that you have to integrate development and operations tasks to develop software efficiently. The kind of workflow you want to enable is one in which you can develop an app, deploy it, learn from production usage of it, change it in response to what you learn, and repeat the cycle quickly and reliably.

Some successful cloud development teams deploy multiple times a day to a live environment. The Azure team used to deploy a major update every two to three months, but now it releases minor updates every two to three days and major releases every two to three weeks. Getting into that cadence really helps you be responsive to customer feedback.

To do that, you have to enable a development and deployment cycle that is repeatable, reliable, predictable, and has low cycle time.



In other words, the period of time between when you have an idea for a feature and when customers are using it and providing feedback must be as short as possible. The first three patterns—automate everything, source control, and continuous integration and delivery—are all about best

practices that we recommend to enable that kind of process.

Azure management scripts

In the introduction to this ebook, you saw the web-based console, the Azure management portal. The management portal enables you to monitor and manage all of the resources that you have deployed on Azure. It's an easy way to create and delete services such as websites and VMs, configure those services, monitor service operation, and so forth. It's a great tool, but using it is a manual process. If you're going to develop a production application of any size, and especially in a team environment, we recommend that you go through the portal UI to learn and explore Azure and then automate the processes that you'll be doing repetitively.

Nearly everything that you can do manually in the management portal or from Visual Studio can also be done by calling the REST management API. You can write scripts using Windows PowerShell, or you can use an open source framework such as Chef or Puppet. You can also use the Bash command-line tool in a Mac or Linux environment. Azure has scripting APIs for all these environments, and it has a .NET management API in case you want to write code instead of script.

For the Fix It app, we created some Windows PowerShell scripts that automate the processes of creating a test environment and deploying the project to that environment, and we'll review some of the contents of those scripts in the following sections.

Environment creation script

The first script we'll look at is named *New-AzureWebsiteEnv.ps1*. It creates an Azure environment that you can deploy the Fix It app to for testing. The main tasks that this script performs are the following:

- Create a website.
- Create a storage account. (Required for blobs and queues, as you'll see in later chapters.)
- Create an Azure SQL Database server and two databases: an application database and a membership database.
- Store settings in Azure that the app will use to access the storage account and databases.
- Create settings files that will be used to automate deployment.

Run the script

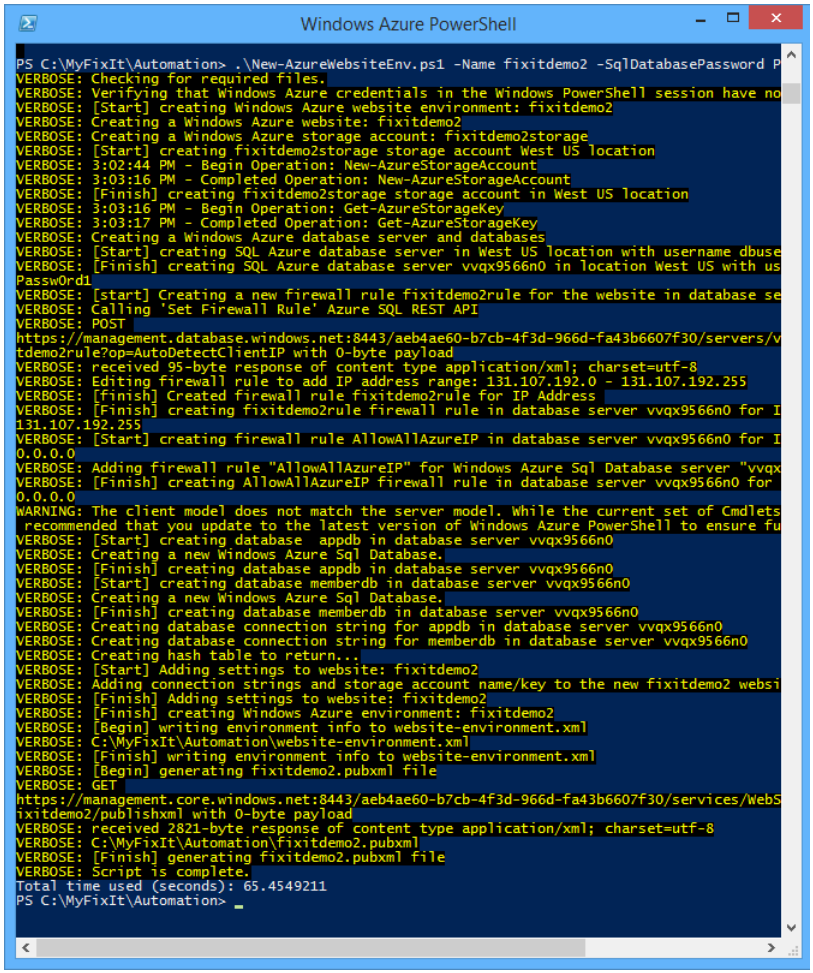
Note: This part of the chapter shows examples of scripts and the commands that you enter to run them. This a demo and doesn't provide everything you need to know to run the scripts. For step-by-

step how-to-do-it instructions, see the appendix, "The Fix It Sample Application."

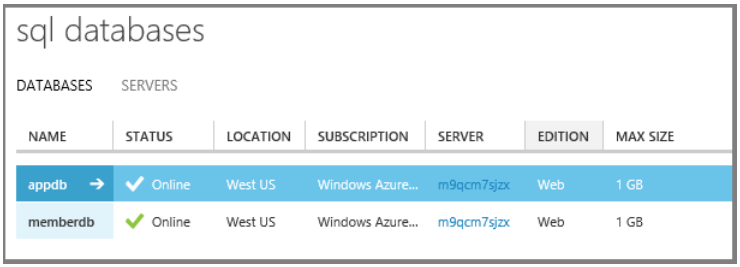
To run a PowerShell script that manages Azure services you have to install the Azure PowerShell console and configure it to work with your Azure subscription. Once you're set up, you can run the Fix It environment creation script with a command like this one:

```
.\New-AzureWebsiteEnv.ps1 -Name <websitename> -SqlDatabasePassword <password>
```

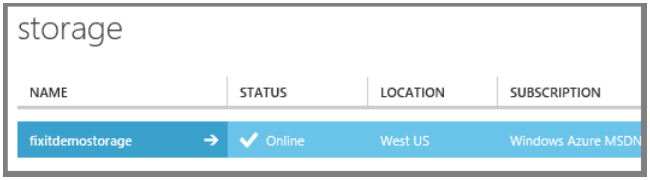
The `Name` parameter specifies the name to be used when creating the database and storage accounts, and the `SqlDatabasePassword` parameter specifies the password for the admin account that will be created for the SQL Database server. There are other parameters you can use that we'll look at later.



After the script finishes, you can see in the management portal what was created. You'll find two databases:

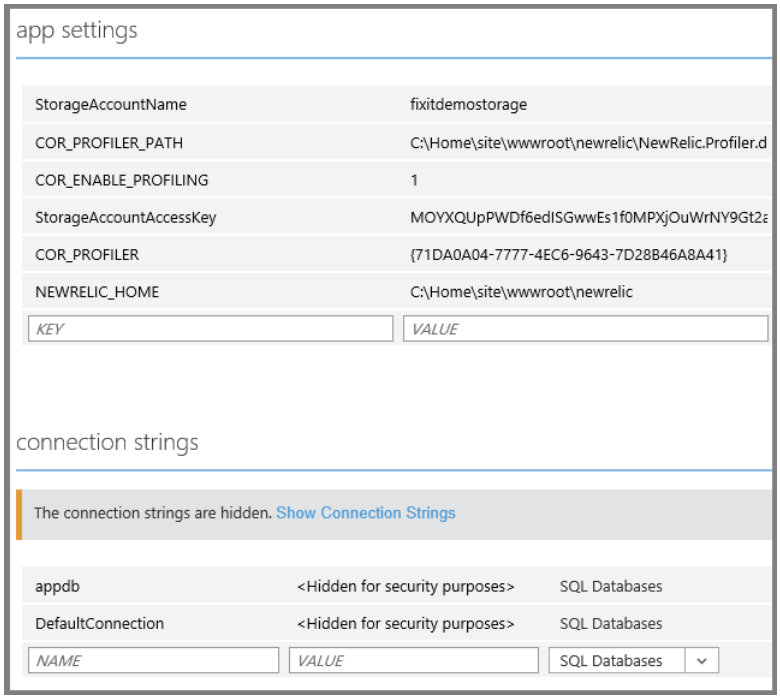


A storage account:



And a website:

On the **Configure** tab for the website, you can see that it has the storage account settings and SQL database connection strings set up for the Fix It app.



The Automation folder now also contains a *<websitename>.pubxml* file. This file stores settings that MSBuild will use to deploy the application to the Azure environment that was just created. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>MSDeploy</WebPublishMethod>

<SiteUrlToLaunchAfterPublish>http://fixitdemo.azurewebsites.net</SiteUrlToLaunchAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <MSDeployServiceURL>waws-prod-bay-
        003.publish.azurewebsites.windows.net:443</MSDeployServiceURL>
    <DeployIisAppPath>fixitdemo</DeployIisAppPath>
    <RemoteSitePhysicalPath />
    <SkipExtraFilesOnServer>True</SkipExtraFilesOnServer>
    <MSDeployPublishMethod>WMSVC</MSDeployPublishMethod>
    <EnableMSDeployBackup>True</EnableMSDeployBackup>
    <UserName>$fixitdemo</UserName>
    <PublishDatabaseSettings></PublishDatabaseSettings>
  </PropertyGroup>
</Project>
```

As you can see, the script has created a complete test environment, and the whole process is done in about 90 seconds.

If someone else on your team wants to create a test environment, they can just run the script. Not only is it fast, but they can be confident that they are using an environment identical to the one you're using. You couldn't be quite as confident of that if everyone was setting things up manually by using the management portal UI.

A look at the scripts

There are actually three scripts that do this work. You call one from the command line and it automatically uses the other two to do some of the tasks:

- *New-AzureWebSiteEnv.ps1* is the main script.
    - *New-AzureStorage.ps1* creates the storage account.
    - *New-AzureSql.ps1* creates the databases.

Parameters in the main script

The main script, New-AzureWebSiteEnv.ps1, defines several parameters:

```
[CmdletBinding(PositionalBinding=$True)]
Param(
    [Parameter(Mandatory = $true)]
    [ValidatePattern("^[a-z0-9]*$")]
    [String]$Name,
    [String]$Location = "West US",
    [String]$SqlDatabaseUserName = "dbuser",
    [String]$SqlDatabasePassword,
    [String]$StartIPAddress,
    [String]$EndIPAddress
    )
```

Two parameters are required:

- The name of the website that the script creates. (This is also used for the URL: `<name>.azurewebsites.net`.)
- The password for the new administrative user of the database server that the script creates. Optional parameters enable you to specify the data center location (defaults to "West US"), database server administrator name (defaults to "dbuser"), and a firewall rule for the database server.

Create the website

The first thing the script does is create the website by calling the `New-AzureWebsite` cmdlet, passing in to it the website name and location parameter values:

```
# Create a new website
```

```
$website = New-AzureWebsite -Name $Name -Location $Location -Verbose
```

Create the storage account

Then the main script runs the *New-AzureStorage.ps1* script, specifying "<*websitename*>storage" for the storage account name and the same data center location as the website.

```
$storageAccountName = $Name + "storage"

$storage = $scriptPath\New-AzureStorage.ps1" -Name $storageAccountName - Location $Location
```

New-AzureStorage.ps1 calls the `New-AzureStorageAccount` cmdlet to create the storage account, and it returns the account name and access key values. The application will need these values to access the blobs and queues in the storage account:

```
# Create a new storage account
New-AzureStorageAccount -StorageAccountName $Name -Location $Location – Verbose

# Get the access key of the storage account
$key = Get-AzureStorageKey -StorageAccountName $Name

# Generate the connection string of the storage account
$connectionString =
"BlobEndpoint=http://$Name.blob.core.windows.net/;QueueEndpoint=http://$Name.
queue.core.windows.net/;TableEndpoint=http://$Name.table.core.windows.net/;Ac
countName=$Name;AccountKey=$primaryKey"

#Return a hashtable of storage account values
Return @{AccountName = $Name; AccessKey = $key.Primary; ConnectionString =
$connectionString}
```

You might not always want to create a new storage account; you could enhance the script by adding a parameter that optionally directs it to use an existing storage account.

Create the databases

The main script then runs the database creation script, *New-AzureSql.ps1*, after setting up default database and firewall rule names:

```
$sqlAppDatabaseName = "appdb"
$sqlMemberDatabaseName = "memberdb"
$sqlDatabaseServerFirewallRuleName = $Name + "rule"
# Create a SQL Azure database server, app and member databases
$sql = $scriptPath\New-AzureSql.ps1 `
    -AppDatabaseName $sqlAppDatabaseName `
    -MemberDatabaseName $sqlMemberDatabaseName `
    -UserName $SqlDatabaseUserName `
    -Password $SqlDatabasePassword `
    -FirewallRuleName $sqlDatabaseServerFirewallRuleName `
    -StartIPAddress $StartIPAddress `
    -EndIPAddress $EndIPAddress `
```

```
-Location $Location
```

The database creation script retrieves the dev machine's IP address and sets a firewall rule so that the dev machine can connect to and manage the server. The database creation script then goes through several steps to set up the databases:

- Creates the server by using the `New-AzureSqlDatabaseServer` cmdlet.

```
$databaseServer = New-AzureSqlDatabaseServer -AdministratorLogin
$UserName -AdministratorLoginPassword $Password -Location $Location
```

- Creates firewall rules to enable the dev machine to manage the server and to enable the website to connect to it.

```
# Create a SQL Azure database server firewall rule for the IP address of the machine in
which this script will run
# This will also put all the Azure IP on an allowed list so that the website can access
the database server
New-AzureSqlDatabaseServerFirewallRule -ServerName $databaseServerName
-RuleName $FirewallRuleName -StartIpAddress $StartIPAddress
-EndIpAddress $EndIPAddress -Verbose
New-AzureSqlDatabaseServerFirewallRule -ServerName
$databaseServer.ServerName -AllowAllAzureServices
-RuleName "AllowAllAzureIP" -Verbose
```

- Creates a database context that includes the server name and credentials by using the `New-AzureSqlDatabaseServerContext` cmdlet.

```
# Create a database context which includes the server name and credential
# These are all local operations. No API call to Microsoft Azure
$credential = New-PSCredentialFromPlainText -UserName $UserName - Password $Password
$context = New-AzureSqlDatabaseServerContext -ServerName
$databaseServer.ServerName -Credential $credential
```

`New-PSCredentialFromPlainText` is a function in the script that calls the `ConvertTo-SecureString` cmdlet to encrypt the password and returns a `PSCredential` object, the same type that the `Get-Credential` cmdlet returns.

- Creates the application database and the membership database by using the `New-AzureSqlDatabase` cmdlet.

```
# Use the database context to create app database
New-AzureSqlDatabase -DatabaseName $AppDatabaseName -Context $context - Verbose

# Use the database context to create member database
New-AzureSqlDatabase -DatabaseName $MemberDatabaseName -Context
$context -Verbose
```

- Calls a locally defined function to create a connection string for each database. The application will use these connection strings to access the databases.

```
$appDatabaseConnectionString = Get-SQLAzureDatabaseConnectionString - DatabaseServerName
```

```
$databaseServerName -DatabaseName $AppDatabaseName - UserName $UserName -Password
$Password
$memberDatabaseConnectionString = Get-SQLAzureDatabaseConnectionString
-DatabaseServerName $databaseServerName -DatabaseName
$MemberDatabaseName -UserName $UserName -Password $Password
```

`Get-SQLAzureDatabaseConnectionString` is a function defined in the script that creates the connection string from the parameter values supplied to it.

```
Function Get-SQLAzureDatabaseConnectionString
{
    Param(
        [String]$DatabaseServerName,
        [String]$DatabaseName,
        [String]$UserName,
        [String]$Password
    )

    Return "Server=tcp:$DatabaseServerName.database.windows.net,
      1433;Database=$DatabaseName;User ID=$UserName@$DatabaseServerName;
      Password=$Password;Trusted_Connection=False;Encrypt=True;
      Connection Timeout=30;"
}
```

- Returns a hash table with the database server name and the connection strings.

```
Return @{ `
    Server = $databaseServer.ServerName; UserName = $UserName; Password = $Password; `
    AppDatabase = @{Name = $AppDatabaseName; ConnectionString =
        $appDatabaseConnectionString}; `
    MemberDatabase = @{Name = $MemberDatabaseName; ConnectionString =
        $memberDatabaseConnectionString} `
}
```

The Fix It app uses separate membership and application databases. It's also possible to put both membership and application data in a single database. For an example that uses a single database, see Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to an Azure Website.

Store app settings and connection strings

Azure has a feature that enables you to store settings and connection strings that automatically override what is returned to the application when it reads the `appSettings` or `connectionStrings` collections in the Web.config file. This is an alternative to applying Web.config transformations when you deploy. For more information, see "Store sensitive data in Azure" later in this ebook.

The environment creation script stores in Azure all of the `appSettings` and `connectionStrings` values that the application needs to access the storage account and databases when it runs in Azure.

```
# Configure app settings for storage account and New Relic
```

```
$appSettings = @{ `
    "StorageAccountName" = $storageAccountName; `
    "StorageAccountAccessKey" = $storage.AccessKey; ` "COR_ENABLE_PROFILING" = "1"; `
    "COR_PROFILER" = "{71DA0A04-7777-4EC6-9643-7D28B46A8A41}"; `
    "COR_PROFILER_PATH" =
"C:\Home\site\wwwroot\newrelic\NewRelic.Profiler.dll"; `
    "NEWRELIC_HOME" = "C:\Home\site\wwwroot\newrelic" `
}

# Configure connection strings for appdb and ASP.NET member db
$connectionStrings = ( `
    @{Name = $sqlAppDatabaseName; Type = "SQLAzure"; ConnectionString =
$sql.AppDatabase.ConnectionString}, `
    @{Name = "DefaultConnection"; Type = "SQLAzure"; ConnectionString =
$sql.MemberDatabase.ConnectionString}
)

# Add the connection string and storage account name/key to the website
Set-AzureWebsite -Name $Name -AppSettings $appSettings -ConnectionStrings
$connectionStrings
```

New Relic is a telemetry framework that we demonstrate in Chapter 5, "Monitoring and telemetry." The environment creation script also restarts the website to make sure that it picks up the New Relic settings.

```
# Restart the website to let New Relic hook kick in
Restart-AzureWebsite -Name $websiteName
```

Preparing for deployment

At the end of the process, the environment creation script calls two functions to create files that will be used by the deployment script.

One of these functions creates a publish profile (<*websitename*>.pubxml file). The code calls the Azure REST API to get the publish settings, and it saves the information in a .publishsettings file. Then it uses the information from that file along with a template file (pubxml.template) to create the .pubxml file that contains the publish profile. This two-step process simulates what you do in Visual Studio: download a .publishsettings file and import that to create a publish profile.

The other function uses another template file (*website-environment.template*) to create a website-environment.xml file that contains settings the deployment script will use along with the .pubxml file.

Troubleshooting and error handling

Scripts are like programs: they can fail, and when they do you want to know as much as you can about the failure and what caused it. For this reason, the environment creation script changes the value of the `VerbosePreference` variable from `SilentlyContinue` to `Continue` so that all verbose messages are displayed. It also changes the value of the `ErrorActionPreference` variable from `Continue` to

`Stop` so that the script stops even when it encounters nonterminating errors:

```
# Set the output level to verbose and make the script stop on error
$VerbosePreference = "Continue"
$ErrorActionPreference = "Stop"
```

Before it does any work, the script stores the start time so that it can calculate the elapsed time when it's done:

```
# Mark the start time of the script execution
$startTime = Get-Date
```

After it completes its work, the script displays the elapsed time:

```
# Mark the finish time of the script execution
$finishTime = Get-Date
# Output the time consumed in seconds
Write-Output ("Total time used (seconds): {0}" -f ($finishTime -
$startTime).TotalSeconds)
```

And for every key operation, the script writes verbose messages; for example:

```
Write-Verbose "[Start] creating $websiteName website in $Location location"
$website = New-AzureWebsite -Name $websiteName -Location $Location –Verbose
Write-Verbose "[Finish] creating $websiteName website in $Location location"
```

Deployment script

What the New-AzureWebsiteEnv.ps1 script does for environment creation, the Publish-AzureWebsite.ps1 script does for application deployment.

The deployment script gets the name of the website from the website-environment.xml file created by the environment creation script.

```
[Xml]$envXml = Get-Content "$scriptPath\website-environment.xml"
$websiteName = $envXml.environment.name
```

It gets the deployment user password from the .publishsettings file:

```
[Xml]$xml = Get-Content $scriptPath\$websiteName.publishsettings
$password = $xml.publishData.publishProfile.userPWD[0]
$publishXmlFile = Join-Path $scriptPath -ChildPath ($websiteName + ".pubxml")
```

It executes the MSBuild command that builds and deploys the project:

```
& "$env:windir\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe" $ProjectFile `
    /p:VisualStudioVersion=12.0 `
    /p:DeployOnBuild=true `
    /p:PublishProfile=$publishXmlFile `
    /p:Password=$password
```

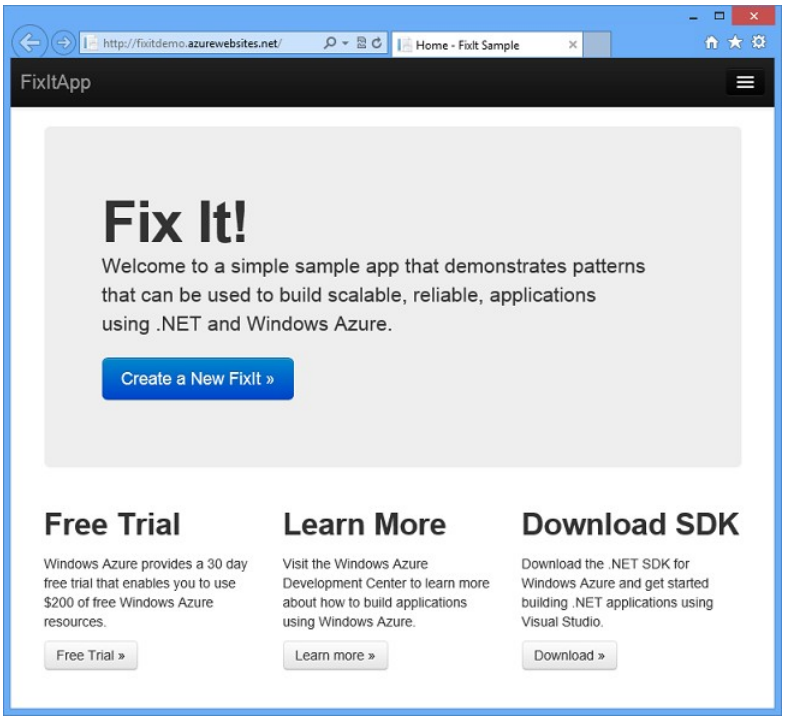
And if you've specified the `Launch` parameter on the command line, it calls the `Show-AzureWebsite` cmdlet to open your default browser to the website URL.

```
If ($Launch)
{
    Show-AzureWebsite -Name $websiteName
}
```

You can run the deployment script with a command like this one:

```
.\Publish-AzureWebsite.ps1 ..\MyFixIt\MyFixIt.csproj -Launch
```

And when it's done, the browser opens with the site running in the cloud at the `<websitename>.azurewebsites.net` URL



Summary

With these scripts you can be confident that the same steps will always be executed in the same order using the same options. This helps ensure that each developer on the team doesn't miss something or mess something up or deploy something custom on his or her own machine that won't actually work the same way in another team member's environment or in production.

In a similar way, you can automate most Azure management functions that you do manually in the management portal by using the REST API, Windows PowerShell scripts, a .NET language API, or a Bash utility that you can run on Linux or Mac.

In the next chapter we'll look at source code and explain why it's important to include your scripts in your source code repository.

Resources

- How to install and configure Azure PowerShell Explains how to install the Azure PowerShell cmdlets and how to install the certificate that you need on your computer to manage your Azure account. This is a great place to get started because it also has links to resources for learning PowerShell itself.
- Azure Script Center A portal to resources for developing scripts that manage Azure services, with links to getting started tutorials, cmdlet reference documentation and source code, and sample scripts
- Weekend Scripter: Getting Started with Windows Azure and PowerShell In a blog dedicated to Windows PowerShell, this post provides a great introduction to using PowerShell for Azure management functions.
- Install and Configure the Azure Cross-Platform Command-Line Interface Getting-started tutorial for an Azure scripting framework that works on Mac and Linux as well as Windows systems.
- Azure Command-line tools Portal page for documentation and downloads related to command line tools for Azure.
- Penny Pinching in the Cloud: Automating everything with the Windows Azure Management Libraries and .NET Scott Hanselman introduces the .NET management API for Azure.
- Using Windows PowerShell Scripts to Publish to Dev and Test Environments MSDN documentation.

Chapter 2

Source control

Source control is essential for all cloud development projects, not just team environments. You wouldn't think of editing source code or even a Word document without an undo function and automatic backups, and source control gives you those functions at a project level, where they can save even more time when something goes wrong. With cloud source control services, you no longer have to worry about complicated setup, and you can use Visual Studio Online source control free for up to five users.

The first part of this chapter explains three key best practices to keep in mind:

- Treat automation scripts as source code and version them together with your application code.
- Never check in secrets (sensitive data such as credentials) to a source code repository.
- Set up source branches to enable the DevOps workflow.

The remainder of the chapter gives some sample implementations of these patterns in Visual Studio, Azure, and Visual Studio Online:

- Add scripts to source control in Visual Studio
- Store sensitive data in Azure
- Use Git in Visual Studio and Visual Studio Online

Treat automation scripts as source code

When you're working on a cloud project, you change things frequently and you want to be able to react quickly to issues reported by your customers. Responding quickly involves using automation scripts, as explained in Chapter 1, "Automate everything." All of the scripts that you use to create your environment, deploy to it, scale it, and so on, need to be in sync with your application source code.

To keep scripts in sync with code, store them in your source control system. Then, if you ever need to roll back changes or make a quick fix to production code that is different from development code, you don't have to waste time trying to track down which settings have changed or which team members have copies of the version you need. You're assured that the scripts you need are in sync with the code base that you need them for, and you're assured that all team members are working with the same scripts. Then, whether you need to automate testing and deployment of a hot fix to

production or a new feature development, you'll have the right script for the code that needs to be updated.

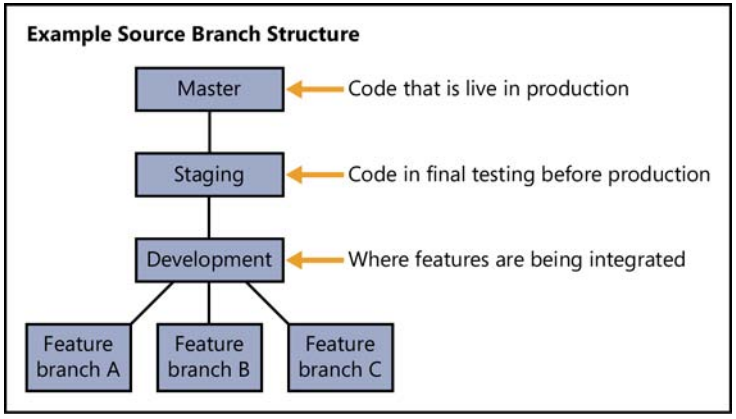
Don’t check in secrets

A source code repository is typically accessible to too many people for it to be an appropriately secure place for sensitive data such as passwords. If scripts rely on secrets such as passwords, parameterize those settings so that they don’t get saved in source code, and store your secrets somewhere else.

For example, Azure lets you download files that contain publish settings in order to automate the creation of publish profiles. These files include user names and passwords that are authorized to manage your Azure services. If you use this method to create publish profiles, and if you check in these files to source control, anyone with access to your repository can see those user names and passwords. You can safely store the password in the publish profile itself because it's encrypted and it's in a .pubxml.user file that by default is not included in source control.

Structure source branches to facilitate DevOps workflow

How you implement branches in your repository affects your ability to both develop new features and fix issues in production. Here is a pattern that a lot of medium sized teams use:



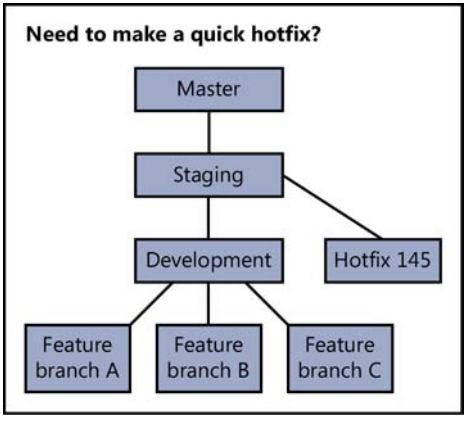
The master branch always matches code that is in production. Branches underneath master correspond to different stages in the development life cycle. The development branch is where you implement new features. For a small team you might have just master and development, but we often recommend that people have a staging branch between development and master. You can use staging for final integration testing before an update is moved to production.

For big teams there may be separate branches for each new feature; for a smaller team, you might have everyone checking in to the development branch.

If you have a branch for each feature, when Feature A is ready you merge its source code changes up into the development branch and down into the other feature branches. This source code merging process can be time-consuming, and to avoid that work while still keeping features separate, some teams implement an alternative called feature toggles (also known as feature flags). This means all of the code for all of the features is in the same branch, but you enable or disable each feature by using switches in the code. For example, suppose Feature A is a new field for Fix It app tasks, and Feature B adds caching functionality. The code for both features can be in the development branch, but the app will display the new field only when a variable is set to true, and it will use caching only when a different variable is set to true. If Feature A isn't ready to be promoted but Feature B is ready, you can promote all of the code to production with the Feature A switch off and the Feature B switch on. You can then finish Feature A and promote it later, all with no source code merging.

Whether or not you use branches or toggles for features, a branching structure like this enables you to flow your code from development into production in an agile and repeatable way.

This structure also enables you to react quickly to customer feedback. If you need to make a quick fix to production, you can also do that efficiently in an agile way. You can create a branch off of master or staging, and when the fix is ready, merge it up into master and down into development and the feature branches.



Without a branching structure like this, with its separation of production and development branches, a production problem could put you in the position of having to promote new feature code along with your production fix. The new feature code might not be fully tested and ready for production, and you might have to do a lot of work backing out changes that aren't ready. Or you might have to delay your fix in order to test changes and get them ready to deploy.

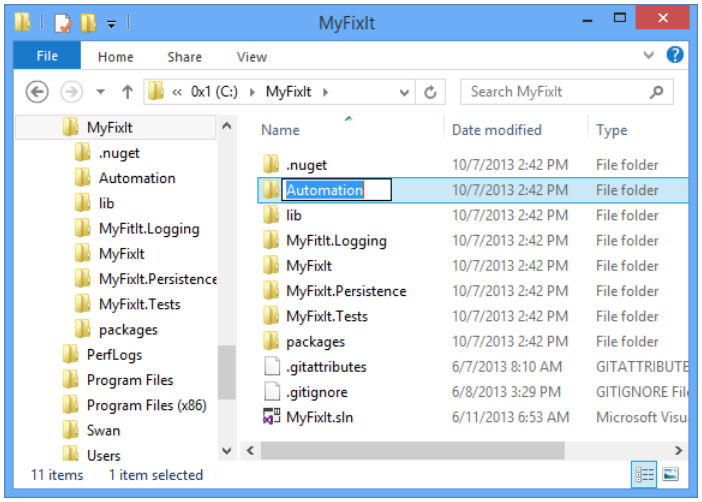
Next, you'll see examples of how to implement these three patterns in Visual Studio, Azure, and Visual Studio Online. These are examples rather than detailed step-by-step, how-to-do-it instructions;

for detailed instructions that provide all of the context necessary, see the Resources section at the end of this chapter.

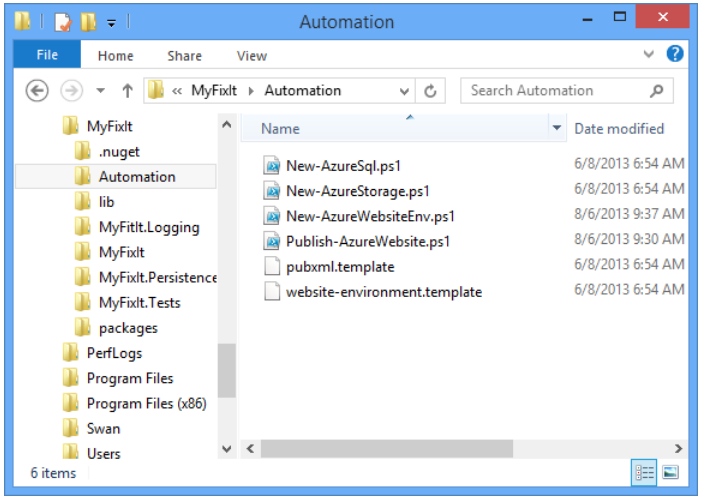
Add scripts to source control in Visual Studio

You can add scripts to source control in Visual Studio by including them in a Visual Studio solution folder (assuming your project is in source control). Here's one way to do it.

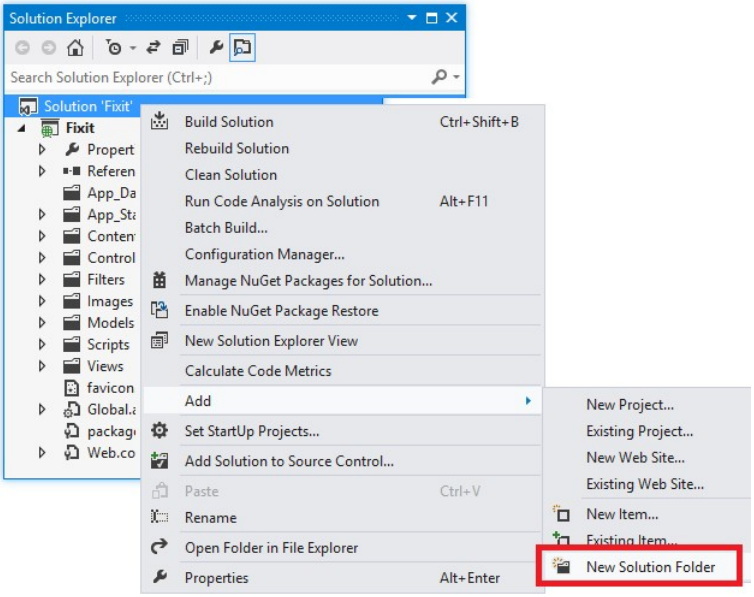
Create a folder for the scripts in your solution folder (the same folder that has your .sln file).



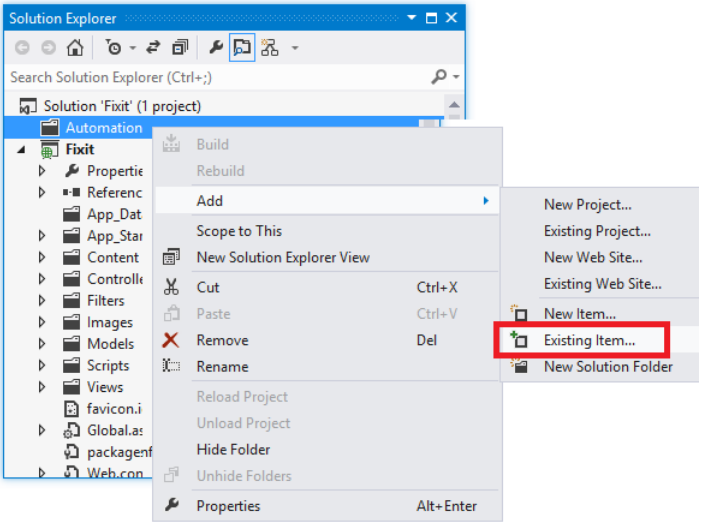
Copy the script files into the folder.

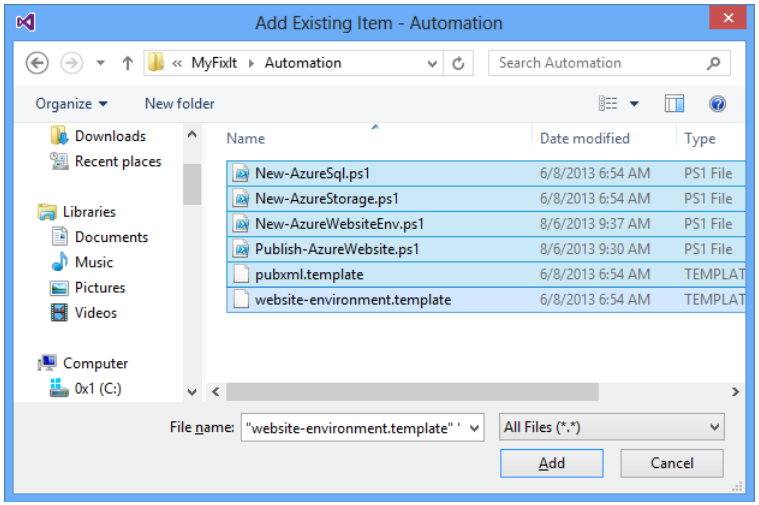


In Visual Studio, add a solution folder to the project.



And add the script files to the solution folder.





The script files are now included in your project and source control is tracking their version changes along with corresponding source code changes.

Store sensitive data in Azure

If you run your application in an Azure website, one way to avoid storing credentials in source control is to store them in Azure instead.

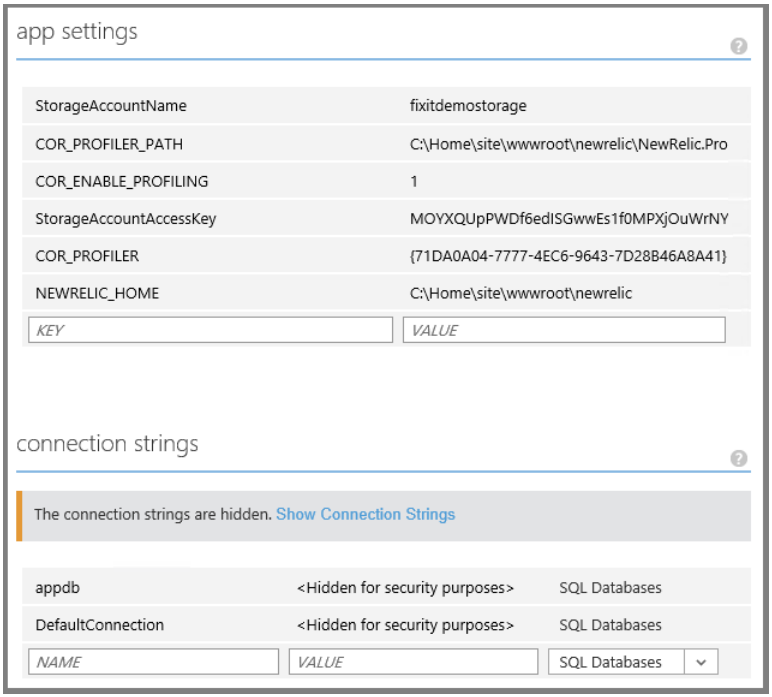
For example, the Fix It application stores in its Web.config file two connection strings that will have passwords in production and a key that gives access to an Azure Storage account.

```
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\MyFixItMembership.mdf
;Initial Catalog=MyFixItMembership;Integrated Security=True"
providerName="System.Data.SqlClient" />
    <add name="appdb" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\MyFixItTasks.mdf;Initial Catalog=aspnet-
MyFixItTasks;Integrated Security=True" providerName="System.Data.SqlClient" />
</connectionStrings>
<appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="StorageAccountName" value="fixitdemostorage" />
    <add key="StorageAccountAccessKey" value="[accesskeyvalue]" />
</appSettings>
```

If you put actual production values for these settings in your Web.config file, or if you put them in

the Web.Release.config file to configure a Web.config transform to insert them during deployment, they'll be stored in the source repository. If you enter the database connection strings into the production publish profile, the password will be in your .pubxml file. (You could exclude the .pubxml file from source control, but then you lose the benefit of sharing all the other deployment settings.)

Azure gives you an alternative for the appSettings and connection strings sections of the Web.config file. Here is the relevant part of the **Configuration** tab for a website in the Azure management portal:



When you deploy a project to this website and the application runs, whatever values you have stored in Azure override whatever values are in the Web.config file.

You can set these values in Azure by using either the management portal or scripts. The environment creation automation script you saw in Chapter 1, creates an Azure SQL Database, gets the storage and SQL Database connection strings, and stores these secrets in the settings for your website.

```
# Configure app settings for storage account and New Relic
$appSettings = @{ `
    "StorageAccountName" = $storageAccountName;`
    "StorageAccountAccessKey" =     $storage.AccessKey; `
    "COR_ENABLE_PROFILING" = "1"; `
    "COR_PROFILER" = "{71DA0A04-7777-4EC6-9643-7D28B46A8A41}"; `
    "COR_PROFILER_PATH" =
"C:\Home\site\wwwroot\newrelic\NewRelic.Profiler.dll"; `
```

```
    "NEWRELIC_HOME" = "C:\Home\site\wwwroot\newrelic" `
}
# Configure connection strings for appdb and ASP.NET member db
$connectionStrings = ( `
    @{Name = $sqlAppDatabaseName; Type = "SQLAzure"; ConnectionString =
$sql.AppDatabase.ConnectionString}, `
    @{Name = "DefaultConnection"; Type = "SQLAzure"; ConnectionString =
$sql.MemberDatabase.ConnectionString}
)
```

Notice that the scripts are parameterized so that actual values don't get persisted to the source repository.

When you run locally in your development environment, the app reads your local Web.config file and your connection string points to a LocalDB SQL Server database in the App_Data folder of your web project. When you run the app in Azure and the app tries to read these values from the Web.config file, what it gets back and uses are the values stored for the website, not what's actually in the Web.config file.

Use Git in Visual Studio and Visual Studio Online

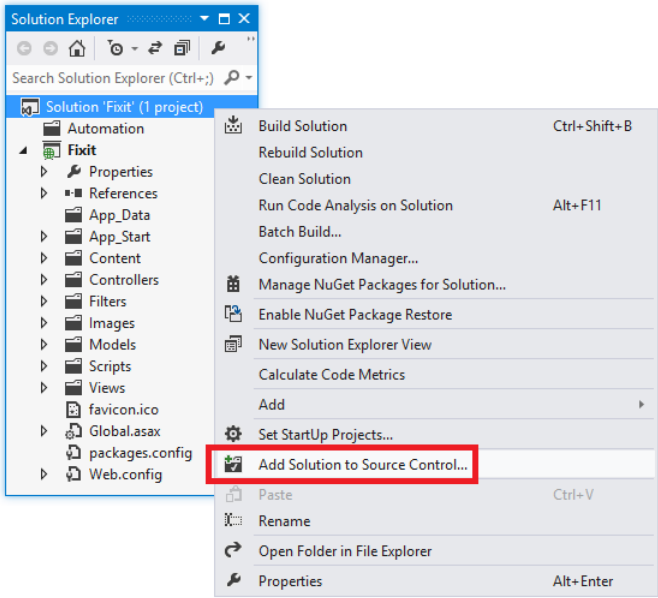
You can use any source control environment to implement the DevOps branching structure presented earlier. For distributed teams a distributed version control system (DVCS) might work best; for other teams a centralized system might work better.

Git is a DVCS that has become very popular. When you use Git for source control, you have a complete copy of the repository with all of its history on your local computer. Many people prefer that because it's easier to continue working when you're not connected to the network—you can continue to do commits and rollbacks, create and switch branches, and so forth. Even when you're connected to the network, it's easier and quicker to create branches and switch branches when everything is local. You can also do local commits and rollbacks without having an impact on other developers. And you can batch commits before sending them to the server.

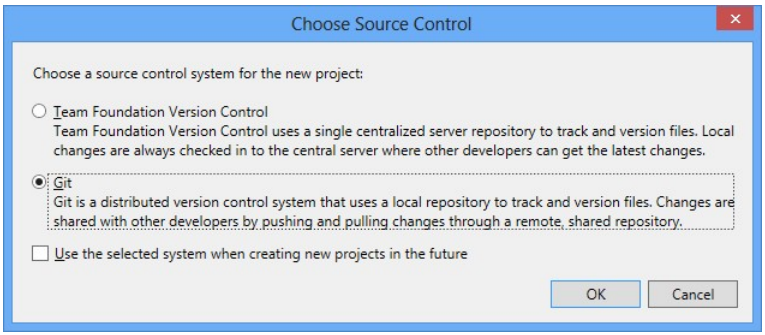
Microsoft Visual Studio Online, formerly known as Team Foundation Service, offers both Git and Team Foundation Version Control (TFVC; centralized source control). Here at Microsoft in the Azure group, some teams use centralized source control, some use distributed, and some use a mix (centralized for some projects and distributed for other projects). The Visul Studio Online service is free for up to five users. You can sign up for a free plan here.

Visual Studio 2013 includes built-in, first-class Git support; here's a quick demo of how that works.

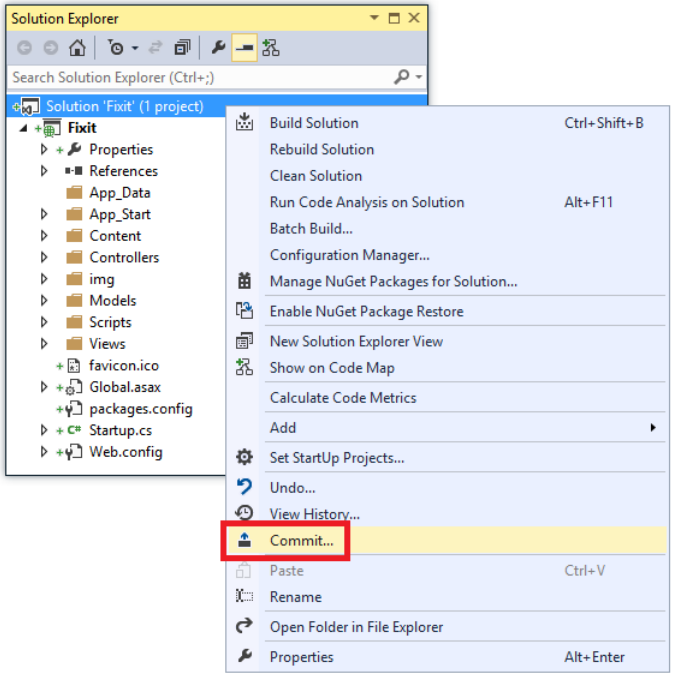
With a project open in Visual Studio 2013, right-click the solution in **Solution Explorer**, and choose **Add Solution to Source Control**.



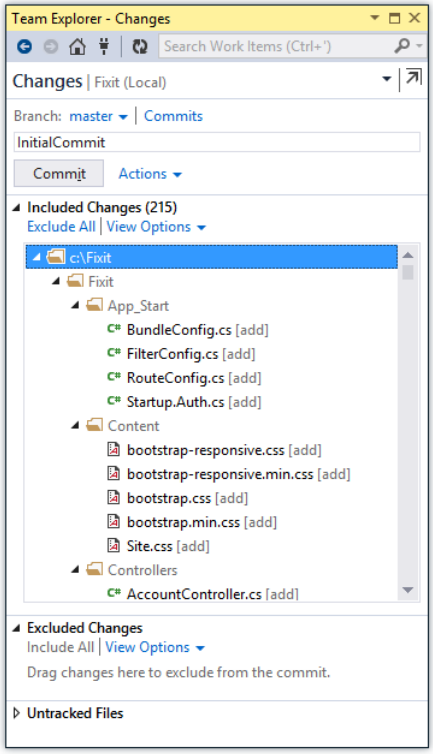
Visual Studio asks if you want to use TFVC (centralized version control) or Git.



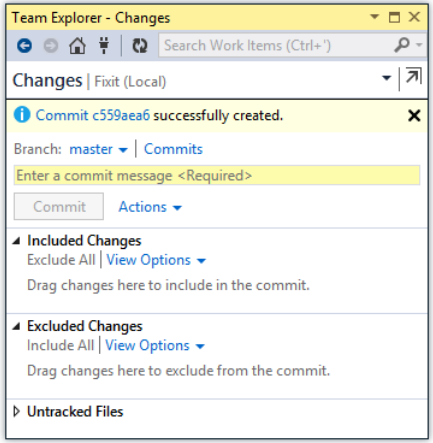
When you select Git and click OK, Visual Studio creates a new local Git repository in your solution folder. The new repository has no files yet; you have to add them to the repository by doing a Git commit. Right-click the solution in **Solution Explorer**, and then click **Commit**.



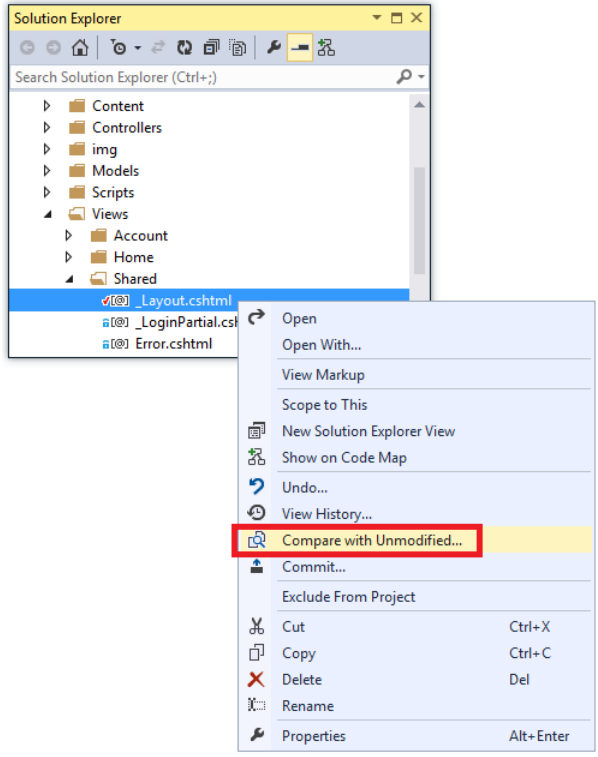
Visual Studio automatically stages all of the project files for the commit and lists them in **Team Explorer** in the **Included Changes** pane. (If there are some files you don't want to include in the commit, you can select them, right-click, and click **Exclude**.)

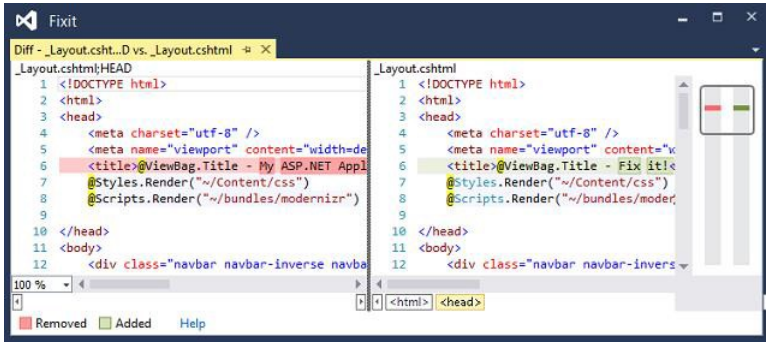


Enter a commit comment and click **Commit**, and Visual Studio executes the commit and displays the commit ID.



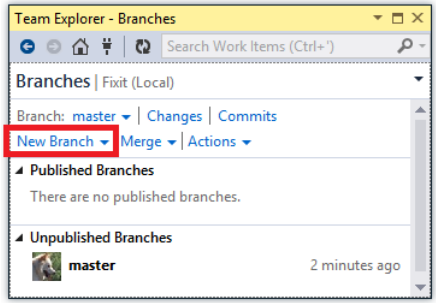
Now, if you change some code so that it's different from what's in the repository, you can easily view the differences. Right-click a file that you've changed, select **Compare with Unmodified**, and you get a comparison display that shows your uncommitted change.



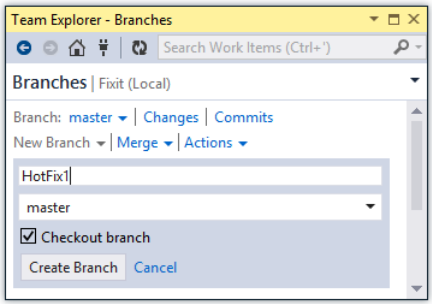


You can easily see what changes you're making and check them in.

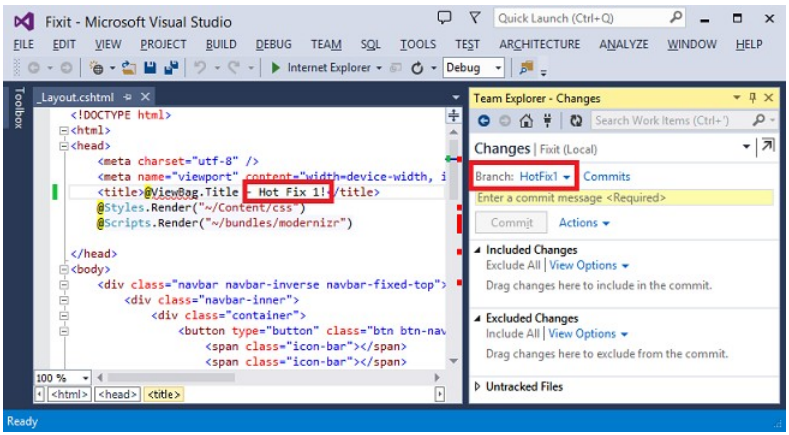
Suppose you need to make a branch—you can do that in Visual Studio, too. In **Team Explorer**, click **New Branch**.



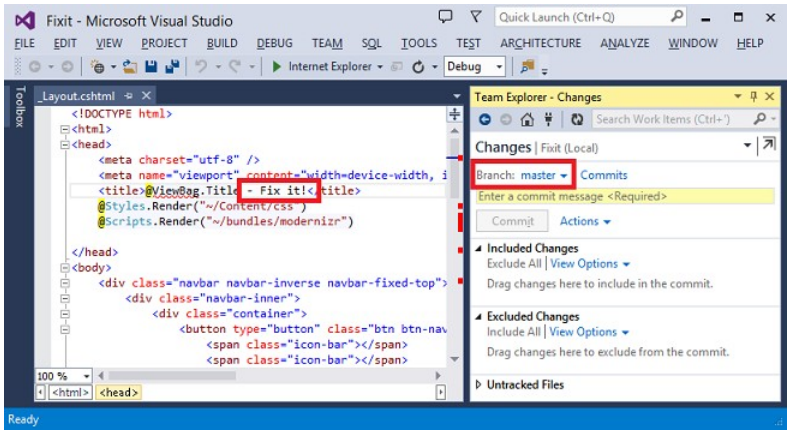
Enter a branch name, click **Create Branch**, and if you select **Checkout Branch**, Visual Studio automatically checks out the new branch.



You can now make changes to files and check them in to this branch. And you can easily switch between branches, and Visual Studio automatically syncs the files to whichever branch you have checked out. In this example, the webpage title in _Layout.cshtml has been changed to “Hot Fix 1” in the HotFix1 branch.



If you switch back to the master branch, the contents of the _Layout.cshtml file automatically revert to what they are in the master branch.



This is a simple example of how you can quickly create a branch and flip back and forth between branches. This feature enables a highly agile workflow using the branch structure and automation scripts presented in Chapter 1. For example, you can be working in the development branch, create a hot fix branch off of master, switch to the new branch, make your changes there and commit them, and then switch back to the development branch and continue what you were doing.

What you've seen here is how you work with a local Git repository in Visual Studio. In a team environment you typically also push changes up to a common repository. The Visual Studio tools also enable you to point to a remote Git repository. You can use GitHub.com for that purpose, and Microsoft is adding Git to Visual Studio Online so that you can use Git integrated with all the other capabilities of Visual Studio Online, such as work item and bug tracking.

This isn't the only way you can implement an agile branching strategy, of course. You can enable the same agile workflow by using a centralized source control repository.

Summary

Measure the success of your source control system based on how quickly you can make a change and get it live in a safe and predictable way. If you find yourself scared to make a change because you have to do a day or two of manual testing on it, you might ask yourself what you have to do process-wise or test-wise so that you can make that change in minutes or, at worst, no longer than an hour. One strategy for doing that is to implement continuous integration and continuous delivery, which we'll cover in the next chapter.

Resources

For an overview of Visual Studio Online, see Introducing Visual Studio Online. The Visual Studio Online

portal provides documentation and support services, and you can sign up for an account.

If you have Visual Studio 2012 and would like to use Git, see Visual Studio Tools for Git.

For more information about TFVC (centralized version control) and Git (distributed version control), see the following resources:

- Which version control system should I use: TFVC or Git? MSDN documentation, includes a table summarizing the differences between TFVC and Git.
- Well, I like Team Foundation Server and I like Git, but which is better? Comparison of Git and TFVC.

For more information about branching strategies, see the following resources:

- Building a Release Pipeline with Team Foundation Server 2012 Microsoft Patterns & Practices documentation. See Chapter 6 for a discussion of branching strategies. Advocates feature toggles over feature branches, and if branches for features are used, advocates keeping them short-lived (hours or days at most).
- Version Control Guide Guide to branching strategies by the ALM Rangers. See Branching Strategies.pdf on the Downloads tab.
- Feature Toggle Introduction to feature toggles/feature flags on Martin Fowler's blog.
- Feature Toggles vs. Feature Branches Another blog post about feature toggles, by Dylan Smith.

For more information about storing settings in Azure, see the following resources:

- Windows Azure Web Sites: How Application Strings and Connection Strings Work Explains the Azure feature that overrides appSettings and connectionStrings data in the Web.config file.
- Custom configuration and application settings in Azure Web Sites—with Stefan Schackow.
- For information about other methods for keeping sensitive information out of source control, see ASP.NET MVC: Keep Private Settings Out of Source Control.

Chapter 3

Continuous integration and continuous delivery

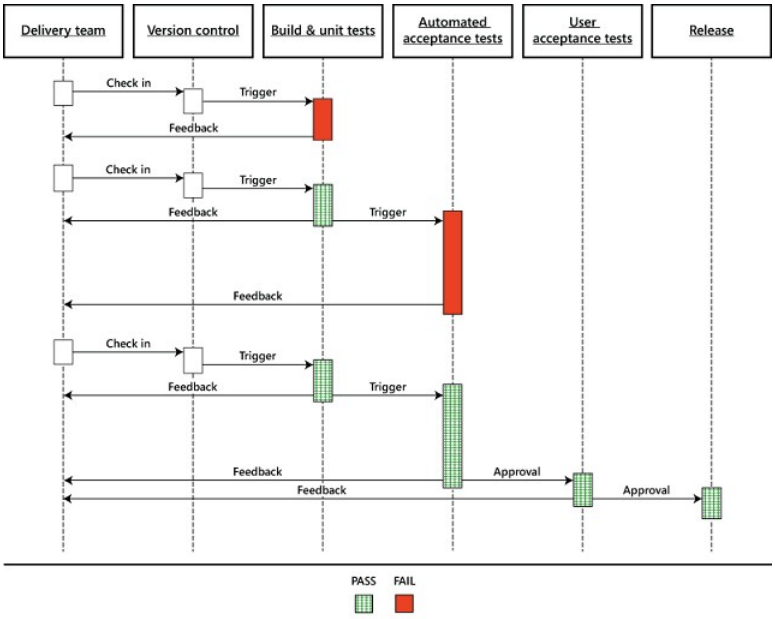
The first two recommended development process patterns were automate everything and source control. The third process pattern, which we cover in this chapter, combines them. Continuous integration (CI) means that whenever a developer checks in code to the source repository, a build is automatically triggered. Continuous delivery (CD) takes this one step further: after a build and automated unit tests are successful, you automatically deploy the application to an environment where you can do more in-depth testing.

The cloud enables you to minimize the cost of maintaining a test environment because you only pay for the environment resources as long as you're using them. Your CD process can set up the test environment when you need it, and you can take down the environment when you're done testing.

Continuous integration and continuous delivery workflow

Generally, we recommend that you perform continuous delivery in your development and staging environments. Most teams, even at Microsoft, require a manual review and approval process for production deployment. For a production deployment you might want to be sure the deployment happens when key people on the development team are available for support or during low-traffic periods. But there's nothing to prevent you from completely automating your development and test environments so that all a developer has to do is check in a change and an environment is set up for acceptance testing.

The following diagram from a Microsoft Patterns and Practices ebook about continuous delivery illustrates a typical workflow. (Click the image to see it full size in its original context.)



How the cloud enables cost-effective CI and CD

Automating these processes in Azure is easy. Because you're running everything in the cloud, you don't have to buy or manage servers for your builds or your test environments. And you don't have to wait for a server to be available to do your testing on it. With every build that you do, you could spin up a test environment in Azure by using your automation script, run acceptance tests or more in-depth tests against it, and then just tear it down when you're done. And if you run that server for only two hours or eight hours or a day, the amount of money that you have to pay for it is minimal, because you're paying only for the time that a machine is actually running. For example, the environment required for the Fix It application basically costs about one cent per hour if you go one tier up from the free level. Over the course of a month, if you ran the environment only an hour at a time, your testing environment would probably cost less than a latte that you buy at Starbucks.

Visual Studio Online

One of the things we're working on with Visual Studio Online, formerly known as Team Foundation Service, is to make it work really well with continuous integration and delivery. Here are some key features of Visual Studio Online:

- It supports both Git (distributed) and TFVC (centralized) source control.

- It offers an elastic build service, which means it dynamically creates build servers when they're needed and takes them down when they're done. You can automatically kick off a build when someone checks in source code changes, and you don't have to allocate and pay for your own build servers that lie idle most of the time. The build service is free as long as you don't exceed a certain number of builds. If you expect to do a high volume of builds, you can pay a little extra for reserved build servers.
- It supports continuous delivery to Azure.
- It supports automated load testing. Load testing is critical to a cloud app but is often neglected until it's too late. Load testing simulates heavy use of an app by thousands of users, enabling you to find bottlenecks and improve throughput—before you release the app to production.
- It supports team room collaboration, which facilitates real-time communication and collaboration for small agile teams.
- It supports agile project management.

For more information about the continuous integration and delivery features of Visual Studio Online, see Visual Studio Lab Management and Visual Studio Release Management. An application monitoring feature, Application Insights for Visual Studio Online, is in preview (available to try but not released for production use yet).

If you're looking for a turn-key project management, team collaboration, and source control solution, check out Visual Studio Online. As mentioned earlier, the service is free for up to five users, and you can sign up for it at http://www.visualstudio.com.

Summary

The first three cloud development patterns have been about how to implement a repeatable, reliable, predictable development process with low cycle time. In the next chapter we start to look at architectural and coding patterns.

Resources

For more information, see How to Deploy an Azure Web Site.

The following white papers and tools are about Team Foundation Server rather than Visual Studio Online, but they explain concepts and procedures that apply generally to continuous delivery:

- Building a Release Pipeline with Team Foundation Server 2012 Ebook, hands-on labs, and sample code by Microsoft Patterns & Practices provides an in-depth introduction to continuous

delivery. Covers use of Visual Studio Lab Management and Visual Studio Release Management.

- ALM Rangers DevOps Tooling and Guidance The ALM Rangers introduced the DevOps Workbench sample companion solution and practical guidance in collaboration with the Patterns & Practices book Building a Release Pipeline with Team Foundation Server 2012, as a great way to start learning the concepts of DevOps and Release Management for TFS 2012 and to kick the tires. The guidance shows how to build once and deploy to multiple environments.
- Testing for Continuous Delivery with Visual Studio 2012 Ebook by Microsoft Patterns & Practices, explains how to integrate automated testing with continuous delivery.
- WindowsAzureDeploymentTracker Source code for a tool designed to capture a build from TFS (based on a label), build it, package it, allow someone in the DevOps role to configure specific aspects of it, and push it into Azure. The tool tracks the deployment process to enable operations to roll back to a previously deployed version. The tool has no external dependencies and can function as a stand-alone tool by using TFS APIs and the Azure SDK.

Videos:

- Deploying to Web Sites with GitHub using Kudu—with David Ebbo Scott Hanselman and David Ebbo show how to deploy a web site directly from GitHub to an Azure website.

Hard-copy books:

- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation By Jez Humble.
- Release It! Design and Deploy Production-Ready Software By Michael T. Nygard.

Chapter 4

Web development best practices

The first three patterns were about setting up an agile development process; the rest are about architecture and code. The pattern covered in this chapter is a collection of web development best practices:

- Stateless web servers behind a smart load balancer.
- Avoid session state (or if you can't avoid it, use distributed cache rather than a database).
- Use a CDN to edge-cache static file assets (images and scripts).
- Use .NET 4.5's async support to avoid blocking calls.

These practices are valid for all web development, not just for cloud apps, but they're especially important for cloud apps. They work together to help you make optimal use of the highly flexible scaling offered by the cloud environment. If you don't follow these practices, you'll run into limitations when you try to scale your application.

Stateless web tier behind a smart load balancer

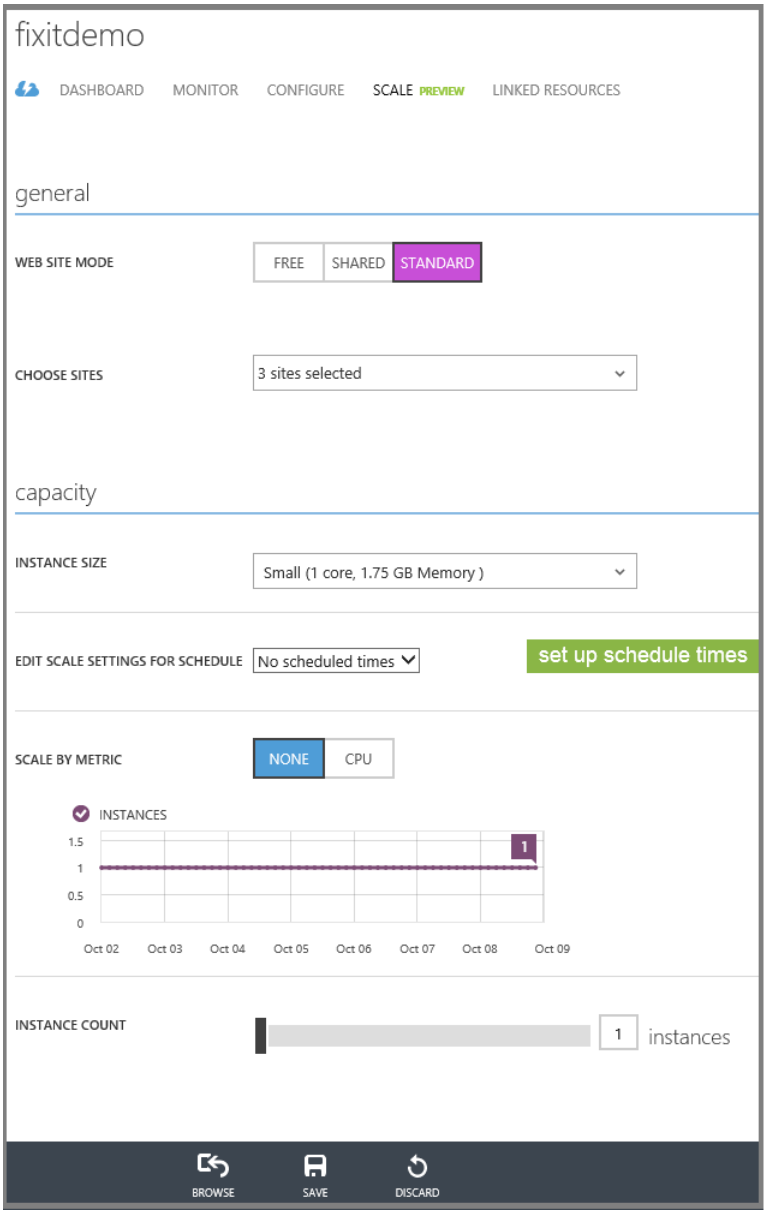
Stateless web tier means you don't store any application data in the web server memory or file system. Keeping your web tier stateless enables you to both provide a better customer experience and save money:

- If the web tier is stateless and it sits behind a load balancer, you can quickly respond to changes in application traffic by dynamically adding or removing servers. In the cloud environment, where you pay for server resources only for as long as you actually use them, that ability to respond to changes in demand can translate into huge savings.
- A stateless web tier is architecturally much simpler for scaling out the application. That enables you to respond to scaling needs more quickly, and spend less money on development and testing in the process.
- Cloud servers, like on-premises servers, need to be patched and rebooted occasionally. If the web tier is stateless, rerouting traffic when a server goes down temporarily won't cause errors or unexpected behavior.

Most real-world applications do need to store state for a web session; the main point here is not to store it on the web server. You can store state in other ways, such as on the client in cookies or out of

process server-side in ASP.NET session state by using the Redis cache provider. You can store files in Azure Blob storage instead of the local file system.

As an example of how easy it is to scale an application in Azure Websites if your web tier is stateless, look at the Scale tab for an Azure website in the management portal:

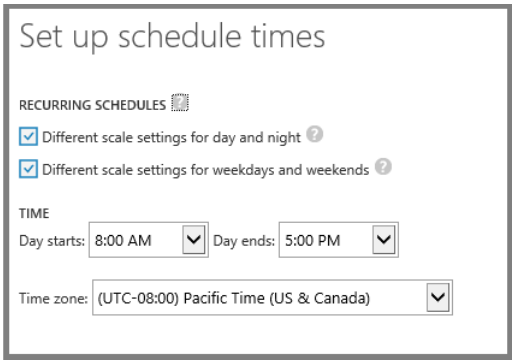


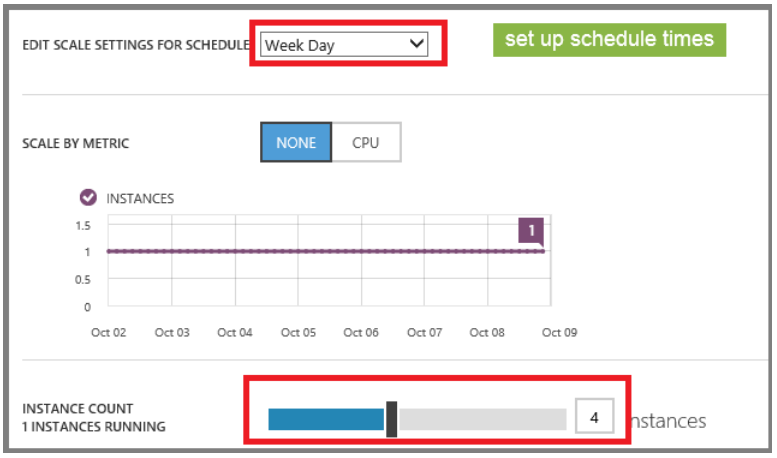
If you want to add web servers, you can just drag the instance count slider to the right. Set it to 5 and click Save, and within seconds you have five web servers in Azure handling your website's traffic.

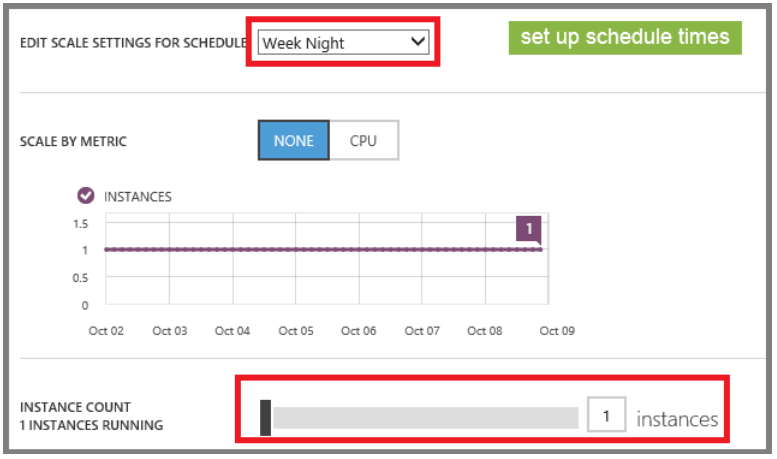
You can just as easily set the instance count to 3 or back down to 1. When you scale back, you start saving money immediately because Azure charges by the minute not by the hour.

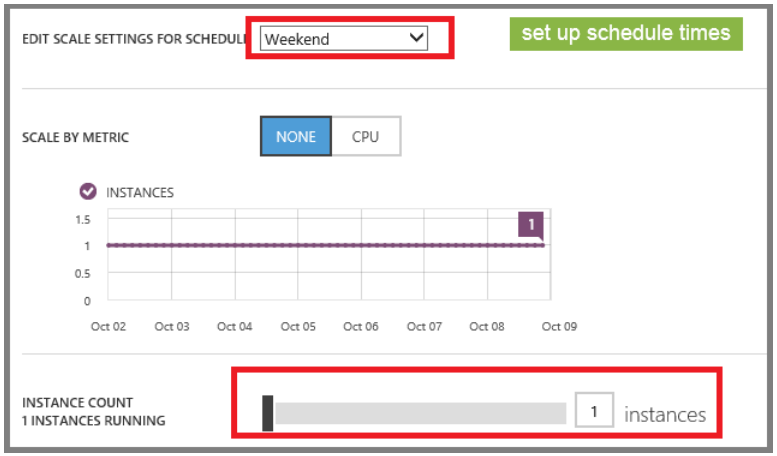
You can also tell Azure to automatically increase or decrease the number of web servers based on CPU usage. In the following example, when CPU usage goes below 60 percent, the number of web servers will decrease to a minimum of two, and if CPU usage goes above 80 percent, the number of web servers will be increased up to a maximum of four.

Or, what if you know that your site will be busy only during working hours? You can tell Azure to run multiple servers during daytime and decrease to a single server evenings, nights, and weekends. The following series of screen shots shows how to set up the website to run one server in off hours and four servers during work hours, from 8 A.M. to 5 P.M.


Set up schedule times
RECURRING SCHEDULES
Different scale settings for day and night
Different scale settings for weekdays and weekends
TIME
Day starts: 8:00 AM
Day ends: 5:00 PM
Time zone: (UTC-08:00) Pacific Time (US & Canada)


EDIT SCALE SETTINGS FOR SCHEDULE
Week Day
set up schedule times
SCALE BY METRIC
NONE
CPU
INSTANCES
1.5
1
0.5
0
1
Oct 02
Oct 03
Oct 04
Oct 05
Oct 06
Oct 07
Oct 08
Oct 09
INSTANCE COUNT
1 INSTANCES RUNNING
4
nstances


EDIT SCALE SETTINGS FOR SCHEDULE
Week Night
set up schedule times
SCALE BY METRIC
NONE
CPU
INSTANCES
1.5
1
0.5
0
1
Oct 02
Oct 03
Oct 04
Oct 05
Oct 06
Oct 07
Oct 08
Oct 09
INSTANCE COUNT
1 INSTANCES RUNNING
1
instances



And, of course, all this work can be done through scripts as well as in the portal.

The ability of your application to scale out is almost unlimited in Azure, so long as you avoid impediments to dynamically adding or removing server VMs by keeping the web tier stateless.

Avoid session state

It's often not practical in a real-world cloud app to avoid storing some form of state for a user session, but some approaches impact performance and scalability more than others. If you have to store state, the best solution is to keep the amount of state small and store it in cookies. If that isn't feasible, the next best solution is to use ASP.NET session state with a provider for distributed, in-memory cache. (See "ASP.NET session state using a cache provider," in Chapter 12, "Distributed Caching.") The worst solution from a performance and scalability standpoint is to use a database-backed session state provider.

Use a CDN to cache static file assets

CDN is an acronym for Content Delivery Network. You provide static file assets such as images and script files to a CDN provider, and the provider caches these files in data centers all over the world so that wherever people access your application, they get relatively quick response and low latency for the cached assets. This speeds up the overall load time of the site and reduces the load on your web servers. CDNs are especially important if you are reaching an audience that is widely distributed geographically.

Azure has a CDN, and you can use other CDNs in an application that runs in Azure or any web hosting environment.

Use .NET 4.5's async support to avoid blocking calls

.NET 4.5 enhanced the C# and Visual Basic programming languages to make it much simpler to handle tasks asynchronously. The benefit of asynchronous programming applies not just to parallel processing situations, such as when you want to kick off multiple web service calls simultaneously. It also enables your web server to perform more efficiently and reliably under high load conditions. A web server has only a limited number of threads available, and under high load conditions, when all of the threads are in use, incoming requests have to wait until threads are freed up. If your application code doesn't handle tasks like database queries and web service calls asynchronously, many threads are unnecessarily tied up while the server is waiting for an I/O response. This limits the amount of traffic the server can handle under high load conditions. With asynchronous programming, threads that are waiting for a web service or database to return data are freed up to service new requests until the data is received. In a busy web server, hundreds or thousands of requests that would otherwise be waiting for threads to be freed up can then be processed promptly.

As you saw earlier, it's as easy to decrease the number of web servers handling your website as it is to increase them. So, if a server can achieve greater throughput, you don't need as many of them, and you can decrease your costs because you need fewer servers for a given traffic volume than you otherwise would.

Support for the .NET 4.5 asynchronous programming model is included in ASP.NET 4.5 for Web Forms, MVC, and Web API; in Entity Framework 6; and in the Azure Storage API.

Async support in ASP.NET 4.5

In ASP.NET 4.5, support for asynchronous programming has been added not just to the language but also to the MVC, Web Forms, and Web API frameworks. For example, an ASP.NET MVC controller action method receives data from a web request and passes the data to a view, which then creates the HTML to be sent to the browser. Frequently, the action method needs to get data from a database or web service to display it in a webpage or to save data entered in a webpage. In those scenarios it's easy to make the action method asynchronous: instead of returning an ActionResult object, you return Task<ActionResult> and mark the method with the async keyword. Inside the method, when a line of code kicks off an operation that involves wait time, you mark it with the *await* keyword.

Here is a simple action method that calls a repository method for a database query:

```
public ActionResult Index()
{
    string currentUser = User.Identity.Name;
    var result = fixItRepository.FindOpenTasksByOwner(currentUser);

    return View(result);
}
```

Here, the same method handles the database call asynchronously:

```
public async Task<ActionResult> Index()
{
    string currentUser = User.Identity.Name;
    var result = await fixItRepository.FindOpenTasksByOwnerAsync(currentUser);

    return View(result);
}
```

Under the covers the compiler generates the appropriate asynchronous code. When the application makes the call to `FindTaskByIdAsync`, ASP.NET makes the `FindTask` request and then unwinds the worker thread and makes it available to process another request. When the `FindTask` request is done, a thread is restarted to continue processing the code that comes after that call. During the interim, between when the `FindTask` request is initiated and when the data is returned, you have a thread available to do useful work which otherwise would be tied up waiting for the response.

There is some overhead for asynchronous code, but under low load conditions, that overhead is negligible, while under high load conditions you're able to process requests that otherwise would be held up waiting for available threads.

It has been possible to do this kind of asynchronous programming since ASP.NET 1.1, but it was difficult to write, prone to error, and difficult to debug. Now that the coding for it is simplified in ASP.NET 4.5, there's no reason anymore not to do it.

Async support in Entity Framework 6

As part of .NET 4.5, Microsoft provided async support for web service calls, sockets, and file system I/O, but the most common pattern for web applications is to hit a database, and Microsoft's data libraries didn't support async programming for this situation. Entity Framework 6 now adds async support for database access.

In Entity Framework 6, all methods that cause a query or command to be sent to the database have async versions. The example here shows the async version of the `Find` method.

```
public async Task<FixItTask> FindTaskByIdAsync(int id)
{
    FixItTask fixItTask = null;
    Stopwatch timespan = Stopwatch.StartNew();
    try
    {
        fixItTask = await db.FixItTasks.FindAsync(id);
        timespan.Stop();
        log.TraceApi("SQL Database", "FixItTaskRepository.FindTaskByIdAsync",
            timespan.Elapsed, "id={0}", id);
    }
    catch(Exception e)
    {
```

```
            log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
        }

        return fixItTask;
    }
```

And this async support works not just for inserts, deletes, updates, and simple finds, it also works with LINQ queries:

```
public async Task<List<FixItTask>> FindOpenTasksByOwnerAsync(string userName)
{
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        var result = await db.FixItTasks
            .Where(t => t.Owner == userName)
            .Where(t=>t.IsDone == false)
            .OrderByDescending(t => t.FixItTaskId).ToListAsync();

        timespan.Stop();
        log.TraceApi("SQL Database",
"FixItTaskRepository.FindTasksByOwnerAsync", timespan.Elapsed, "username={0}", userName);
        return result;
    }
    catch (Exception e)
    {
        log.Error(e, "Error in FixItTaskRepository.FindTasksByOwnerAsync(userName={0})",
            userName);
        return null;
    }
}
```

There's an `Async` version of the `ToList` method because in this code that's the method that causes a query to be sent to the database. The `Where` and `OrderByDescending` methods only configure the query, while the `ToListAsync` method executes the query and stores the response in the `result` variable.

Summary

You can implement the web development best practices outlined here in any web programming framework and any cloud environment, but there are tools in ASP.NET and Azure that make it easy. If you follow these patterns, you can easily scale out your web tier, and you'll minimize your expenses because each server will be able to handle more traffic.

The next chapter looks at how the cloud enables single sign-on scenarios.

Resources

For more information, see the following resources.

Stateless web servers:

- Microsoft Patterns & Practices—Autoscaling Guidance.
- Disabling ARR's Instance Affinity in Windows Azure Web Sites Blog post by Erez Benari, explains session affinity in Azure Websites.

CDN:

- Using CDN for Azure. Step-by-step tutorial that explains how to use the Azure CDN.
- FailSafe: Building Scalable, Resilient Cloud Services Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. See the CDN discussion in episode 3, starting at 1:34:00.
- Microsoft Patterns & Practices Static Content Hosting Pattern.
- CDN Reviews Overview of many CDNs.

Asynchronous programming:

- Using Asynchronous Methods in ASP.NET MVC 4 Tutorial by Rick Anderson.
- Best Practices in Asynchronous Programming *MSDN Magazine* article by Stephen Cleary.
- Asynchronous Programming with Async and Await (C# and Visual Basic) MSDN white paper that explains rationale for asynchronous programming, how it works in ASP.NET 4.5, and how to write code to implement it.
- How to Build ASP.NET Web Applications Using Async Video presentation by Rowan Miller. Includes a graphical demonstration of how asynchronous programming can facilitate dramatic increases in web server throughput under high load conditions.
- FailSafe: Building Scalable, Resilient Cloud Services Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. For discussions about the impact of asynchronous programming on scalability, see episode 4 and episode 8.
- The Magic of using Asynchronous Methods in ASP.NET 4.5 plus an important gotcha Blog post by Scott Hanselman, primarily about using async in ASP.NET Web Forms applications.

For additional web development best practices, see the following resources:

- The Fix It Sample Application - Best Practices The appendix to this ebook lists a number of best practices that were implemented in the Fix It application.
- Web Developer Checklist.

Chapter 5

Single sign-on

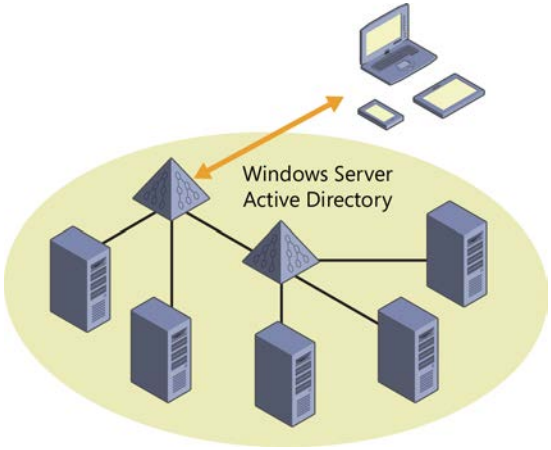
There are many security issues to think about when you're developing a cloud app, but for this chapter we'll focus on just one: single sign-on. A question people often ask is this: "I'm primarily building apps for the employees of my company; how do I host these apps in the cloud and still enable them to use the same security model that my employees know and use in the on-premises environment when they're running apps that are hosted inside the firewall?" One of the ways we enable this scenario is called Azure Active Directory (Azure AD). Azure AD enables you to make enterprise line-of-business (LOB) apps available over the Internet, and it enables you to make these apps available to business partners as well.

Introduction to Azure Active Directory

Azure AD provides Active Directory in the cloud. Key features include the following:

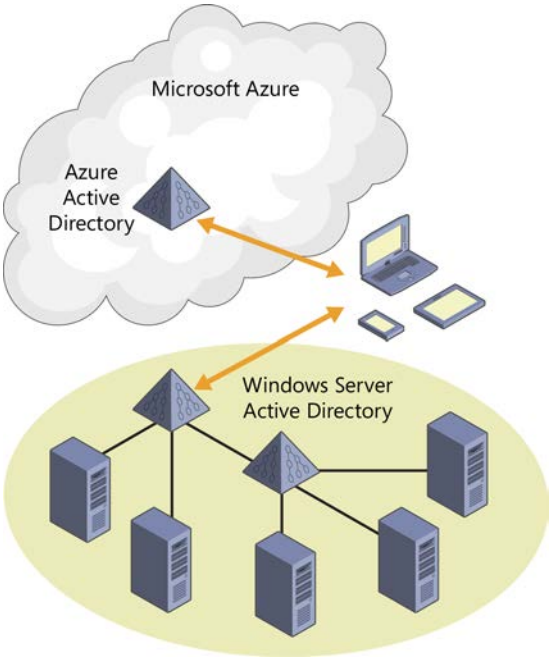
- It integrates with on-premises Active Directory.
- It enables single sign-on with your apps.
- It supports open standards such as SAML, WS-Fed, and OAuth 2.0.
- It supports Azure AD Graph REST API.

Suppose you have an on-premises Windows Server Active Directory environment that you use to enable employees to sign on to intranet apps:

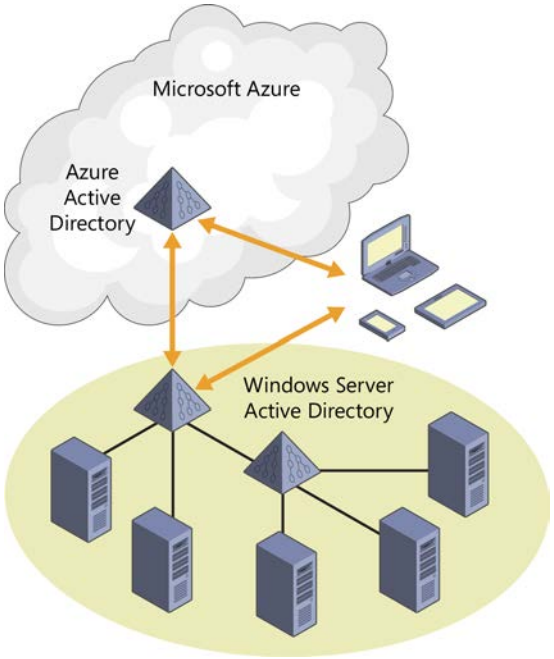


What Azure AD enables you to do is create a directory in the cloud. It's a free feature and easy to set up.

This directory can be entirely independent from your on-premises Active Directory; you can put any user you want in it and authenticate them in Internet apps.

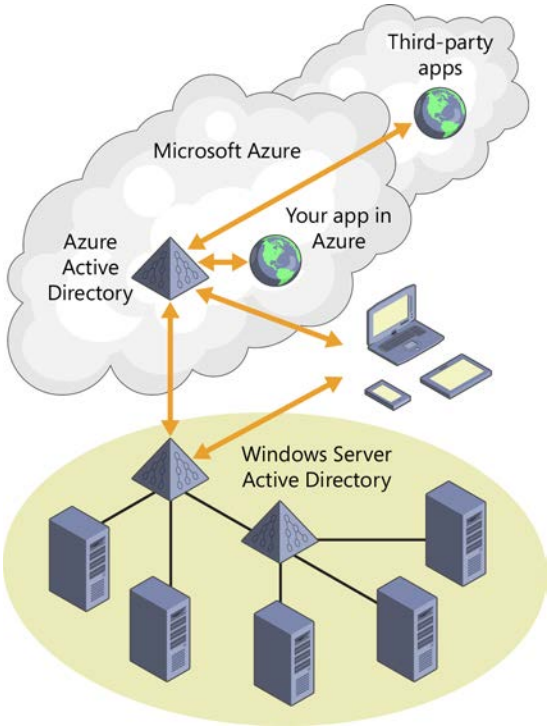


Or you can integrate it with your on-premises AD.



Now all the employees who can authenticate on-premises can also authenticate over the Internet—without you having to open up a firewall or deploy any new servers in your data center. You can continue to leverage all the capabilities of the existing Active Directory environment that you know and use today to give your internal apps single-sign on capability.

Once you've made this connection between AD and Azure AD, you can also enable your web apps and your mobile devices to authenticate your employees in the cloud, and you can enable third-party apps, such as Office 365, SalesForce.com, or Google apps, to accept your employees' credentials. If you're using Office 365, you're already set up with Azure AD because Office 365 uses it for authentication and authorization.



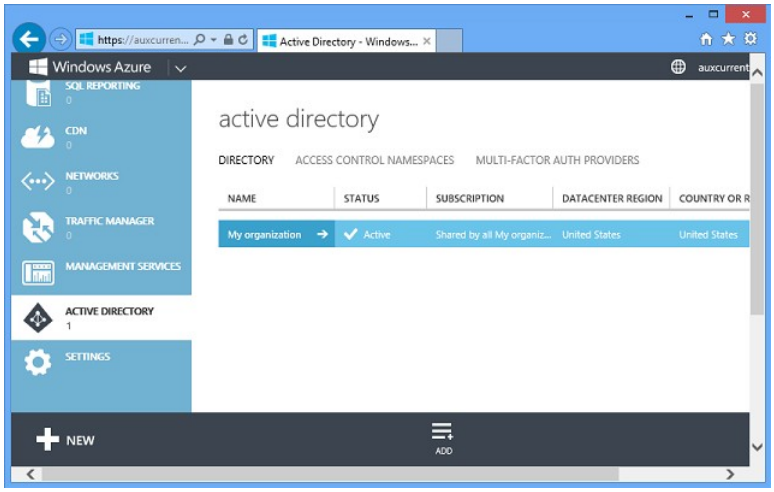
The beauty of this approach is that any time your organization adds or deletes a user, or a user changes a password, you use the same process that you use today in your on-premises environment. All of your on-premises AD changes are automatically propagated to the cloud environment.

If your company is using or moving to Office 365, the good news is that Azure AD will be set up automatically. So you can easily use the same authentication that Office 365 uses in your own apps.

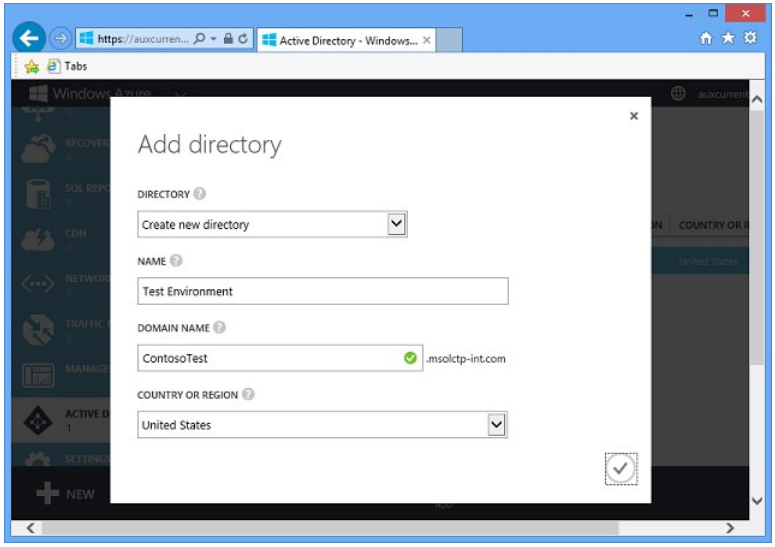
Set up an Azure AD tenant

An Azure AD directory is called an Azure AD tenant, and setting up a tenant is pretty easy. We'll show you how it's done in the Azure management portal to illustrate the concepts, but of course you can also set up a tenant by using a script or management API.

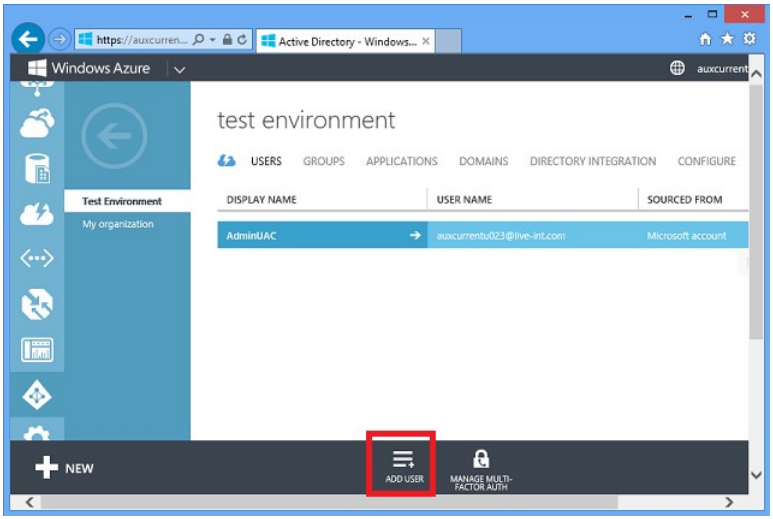
In the management portal click the **Active Directory** tab.



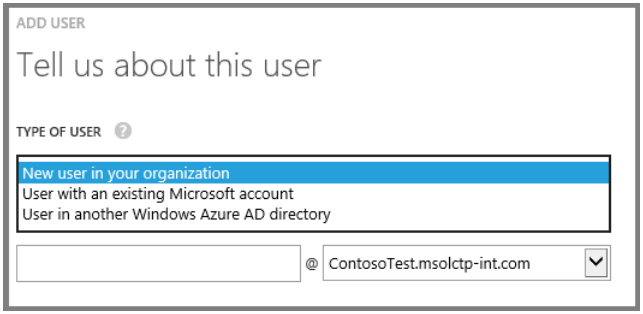
You automatically have one Azure AD tenant for your Azure account, and you can click the Add button at the bottom of the page to create additional directories. You might want one for a test environment and one for production, for example. Think carefully about what you name a new directory. If you use your name for the directory and then use your name again as one of the users, that can be confusing.

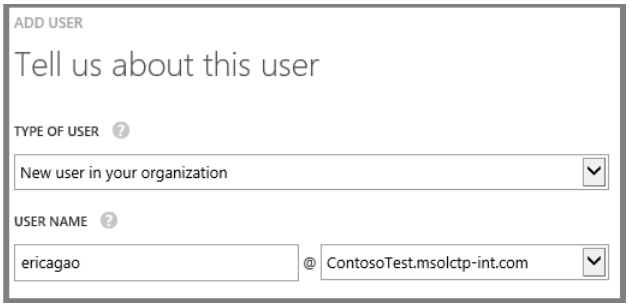


The portal has full support for creating, deleting, and managing users within this environment. For example, to add a user, go to the Users tab and click the Add User button.

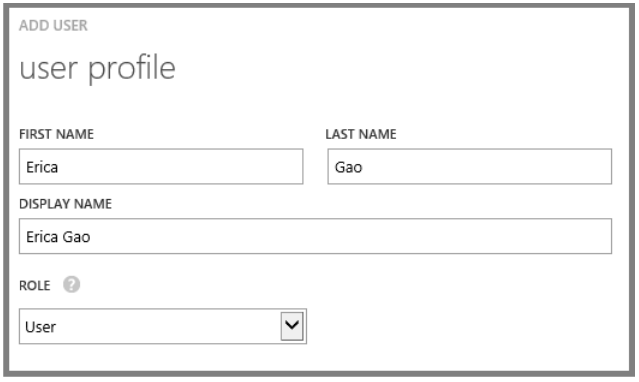


You can create a new user who exists only in this directory, or you can register a Microsoft Account or a user from another Azure AD directory as a user in this directory. (In a real directory, the default domain would be ContosoTest.onmicrosoft.com. You can also use a domain of your own choosing, such as contoso.com.)

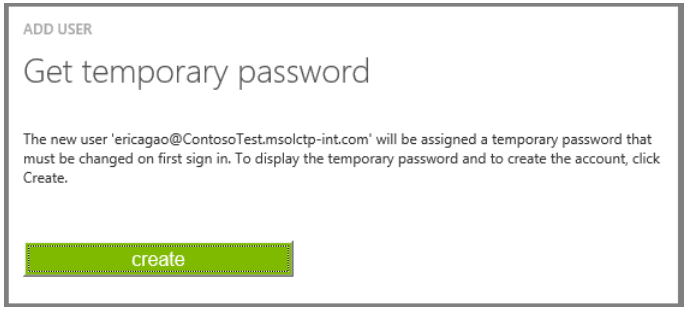




You can assign the user to a role.

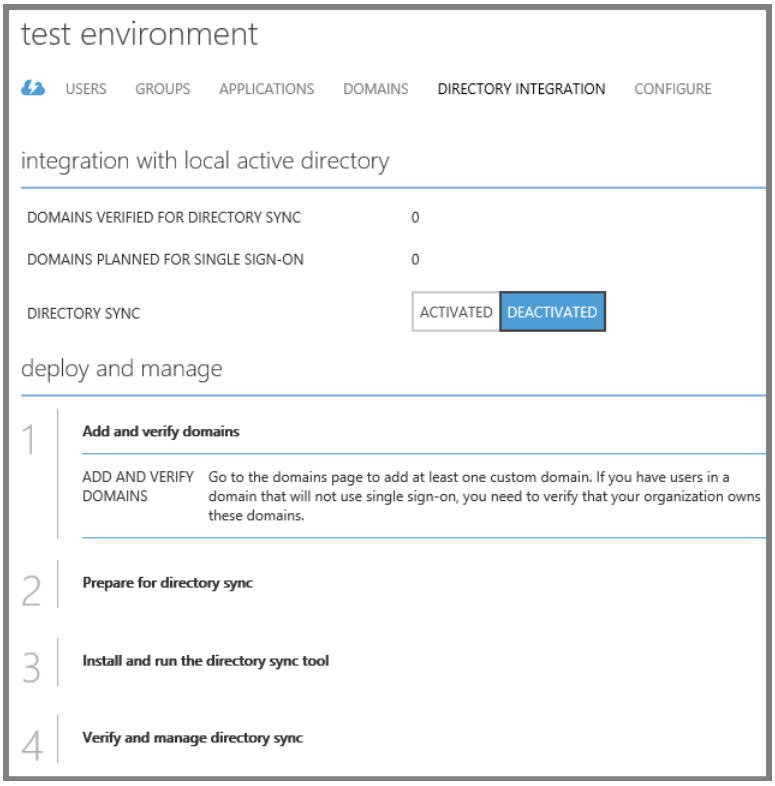


And the account is created with a temporary password.



The users you create this way can immediately sign in to your web apps using this cloud directory.

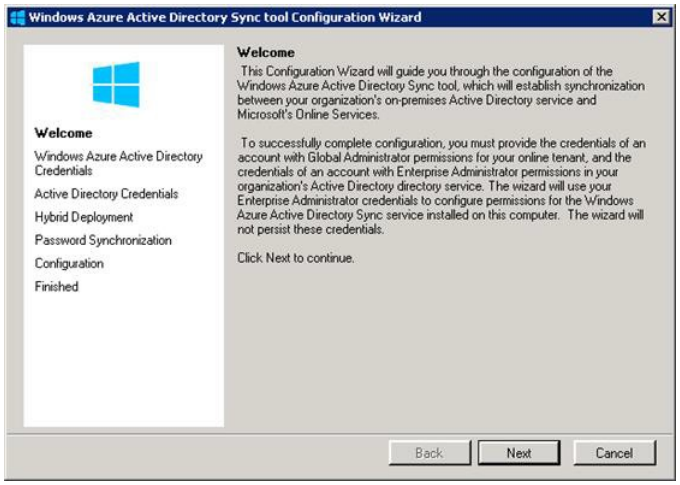
What's great for enterprise single sign-on, though, is the **Directory Integration** tab:



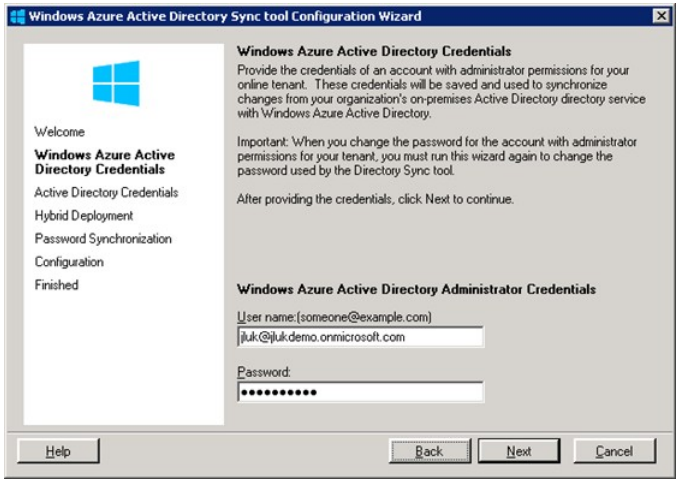
If you enable directory integration and download a tool, you can sync this cloud directory with your existing on-premises Active Directory that you're already using inside your organization. Then, all of the users stored in your directory will show up in this cloud directory. Your cloud apps can now authenticate all of your employees using their existing Active Directory credentials. And all this is free—both the sync tool and Azure AD itself.

The tool is a wizard that is easy to use, as you can see from the following screen shots. These are not complete instructions, just an example showing you the basic process. For more detailed how-to-do-it information, see the links in the Resources section at the end of the chapter.

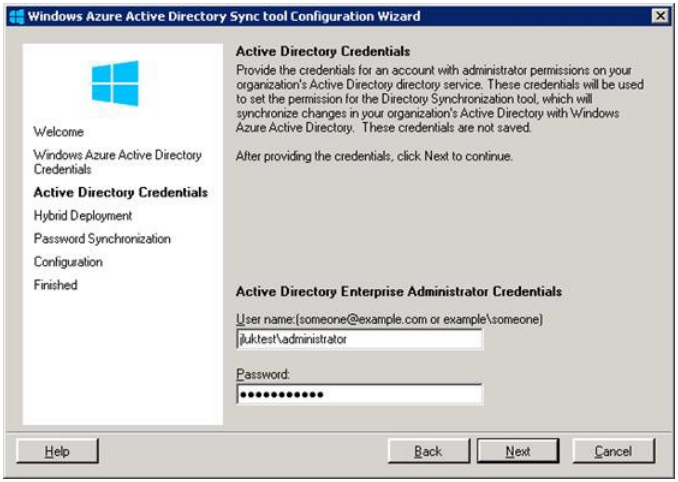
First you see the Welcome page.



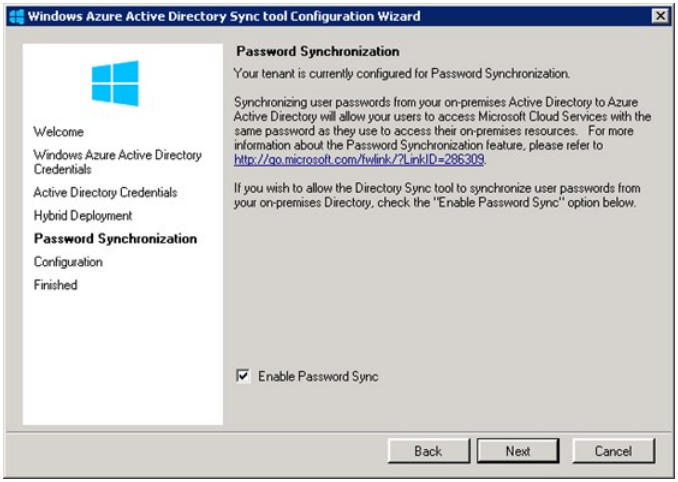
Click **Next**, and then enter your Azure Active Directory credentials.



Click **Next**, and then enter your on-premises AD credentials.

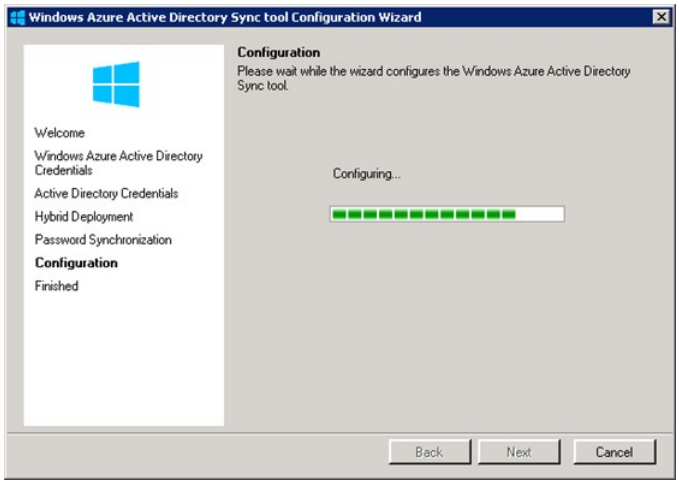


Click **Next**, and then indicate whether you want to store a hash of your AD passwords in the cloud.

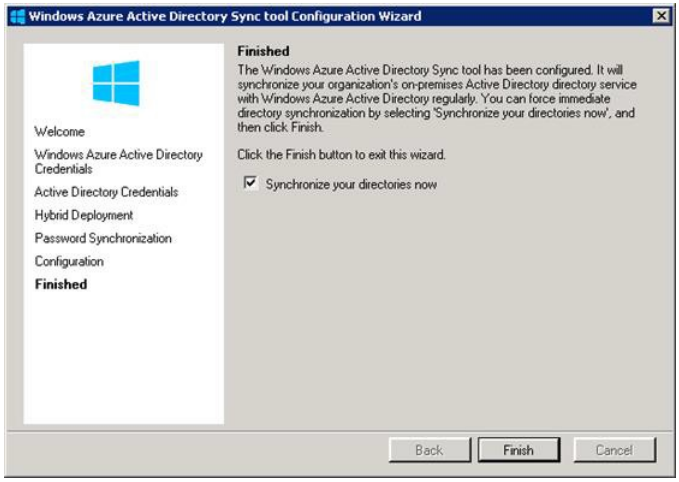


The password hash that you can store in the cloud is a one-way hash; actual passwords are never stored in Azure AD. If you decide against storing hashes in the cloud, you'll have to use Active Directory Federation Services (ADFS). There are also other factors to consider when choosing whether to use ADFS. The ADFS option requires a few additional configuration steps.

If you choose to store hashes in the cloud, you're done, and the tool starts synchronizing directories when you click **Next**.

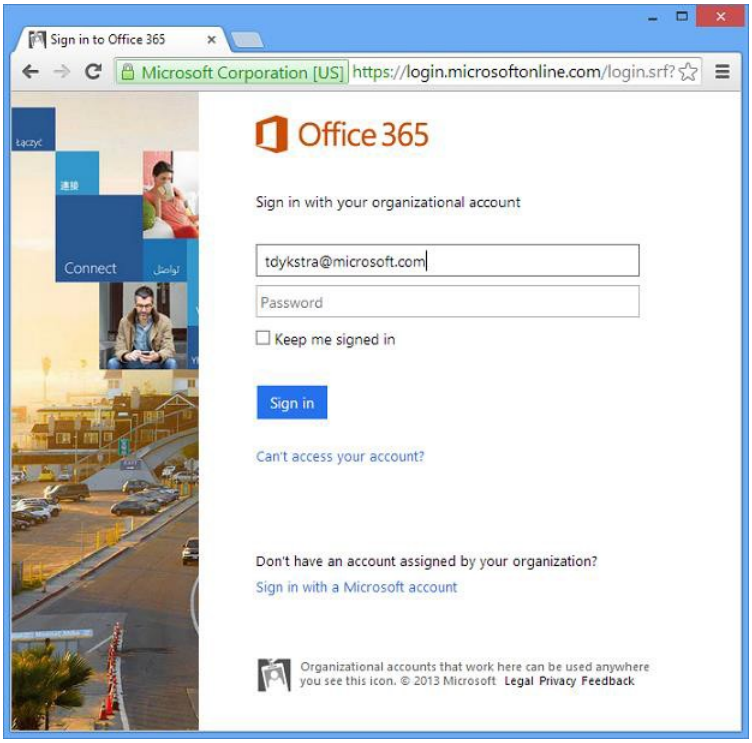


And in a few minutes you're done.

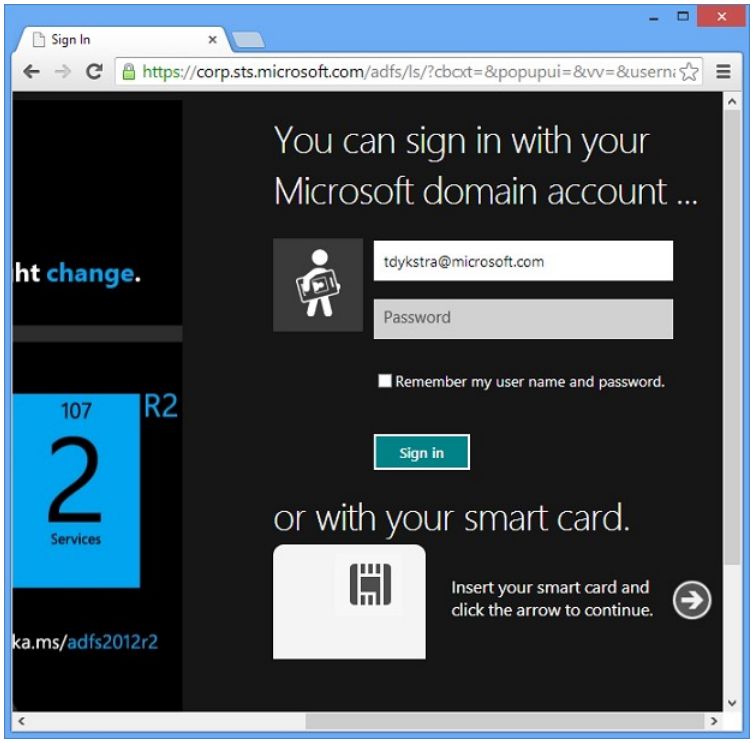


You only have to run this wizard on one domain controller in the organization; the server must be running Windows Server 2003 or higher. And no need to reboot. When you're done, all of your users are set up in the cloud, and you can do single sign-on from any web or mobile application, using SAML, OAuth, or WS-Fed.

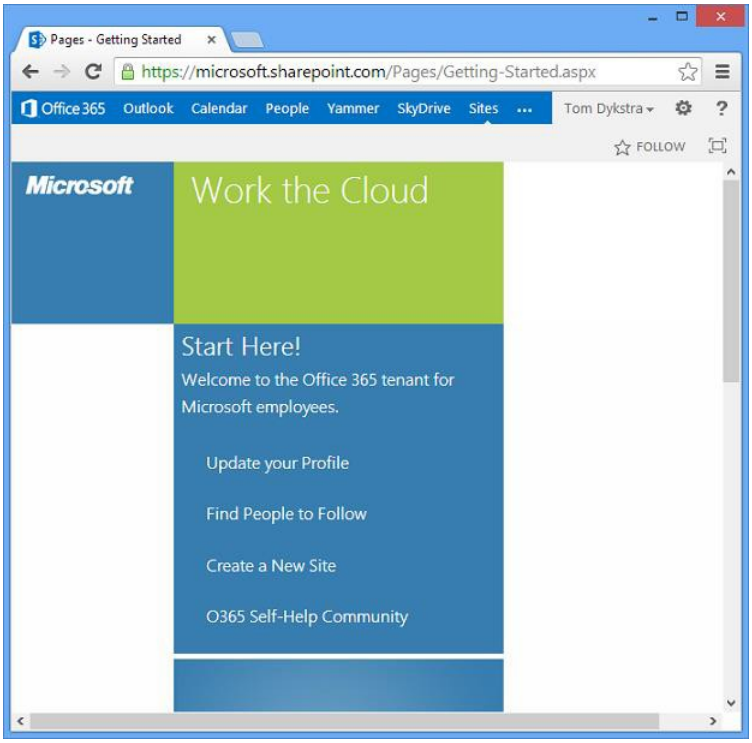
Sometimes we're asked about how secure this approach is—does Microsoft use it for its own sensitive business data? And the answer is yes it does. For example, if you go to an internal Microsoft SharePoint site, such as *http://microsoft.sharepoint.com*, you get prompted to log in.



Microsoft has enabled ADFS, so when you enter a Microsoft ID, you're redirected to an ADFS log-in page.



And once you enter credentials that are stored in an internal Microsoft AD account, you have access to this internal application.

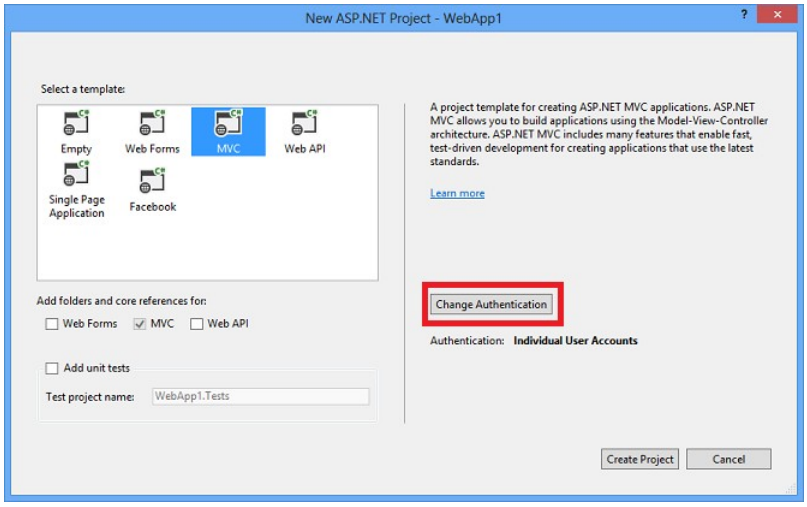


Microsoft uses an AD sign-in server mainly because it already had ADFS set up before Azure AD became available, but the log-in process is going through an Azure AD directory in the cloud. Microsoft puts its important documents, source control, performance management files, sales reports, and more in the cloud and is using this exact solution to secure them.

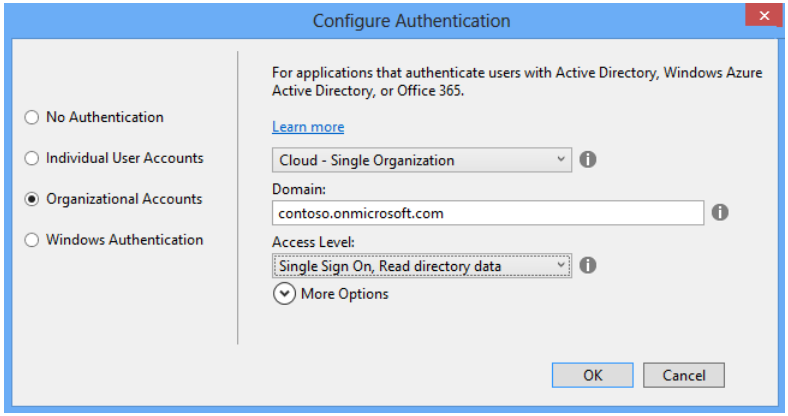
Create an ASP.NET app that uses Azure AD for single sign-on

Visual Studio makes it really easy to create an app that uses Azure AD for single sign-on, as you can see from a few screen shots.

When you create a new ASP.NET application, either MVC or Web Forms, the default authentication method is ASP.NET Identity. To change that to Azure AD, you click the **Change Authentication** button.



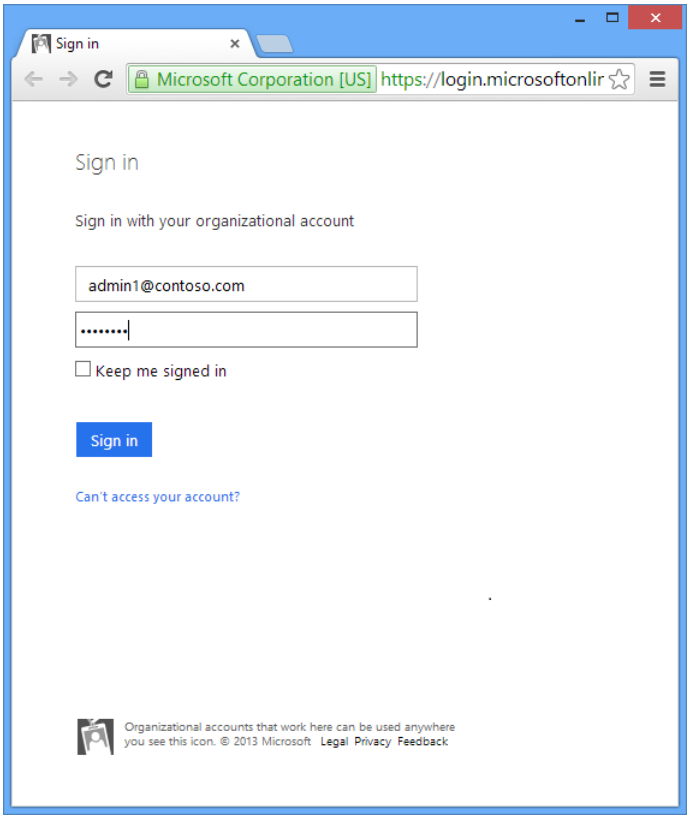
Select **Organizational Accounts**, enter your domain name, and then select **Single Sign On**.

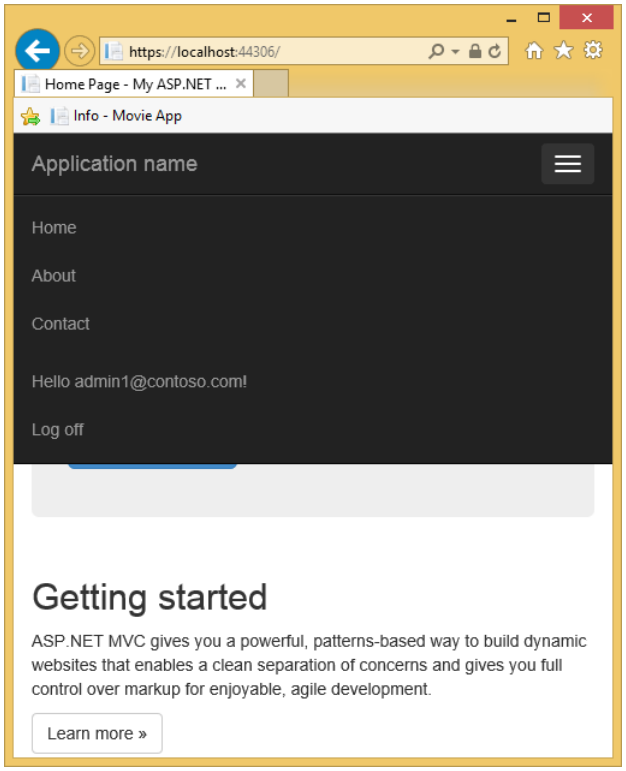


You can also give the app read or read/write permission for directory data. If you do that, it can use the Azure Active Directory Graph REST API to look up users' phone numbers, find out whether they're in the office, when they last logged on, and other information.

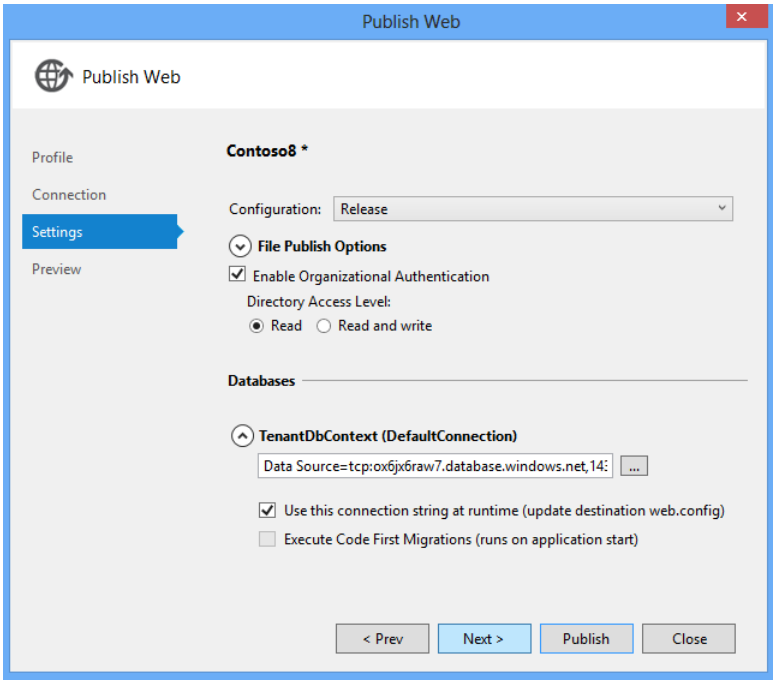
That's all you have to do—Visual Studio asks for the credentials for an administrator of your Azure AD tenant and then it configures both your project and your Azure AD tenant for the new application.

When you run the project, you'll see a sign-in page, and you can sign in with the credentials of a user in your Azure AD directory.


Sign in
Microsoft Corporation [US]
https://login.microsoftonlir
Sign in
Sign in with your organizational account
admin1@contoso.com
Keep me signed in
Sign in
Can't access your account?
Organizational accounts that work here can be used anywhere you see this icon. © 2013 Microsoft
Legal
Privacy
Feedback



When you deploy the app to Azure, all you have to do is select the **Enable Organizational Authentication** check box, and once again Visual Studio takes care of all the configuration for you.



These screen shots come from a complete step-by-step tutorial that shows how to build an app that uses Azure AD authentication: Developing ASP.NET Apps with Windows Azure Active Directory.

Summary

In this chapter you saw that Azure Active Directory, Visual Studio, and ASP.NET make it easy to set up single sign-on in Internet applications for your organization's users. Your users can sign on in Internet apps using the same credentials they use to sign on using Active Directory in your internal network.

The next chapter looks at the data storage options available for a cloud app.

Resources

For more information, see the following resources:

- Azure Active Directory Documentation Portal page for Azure AD documentation on azure.microsoft.com. For step by step tutorials, see the Develop section.
- Multi-Factor Authentication Portal page for documentation about multi-factor authentication in Azure.

- Organizational account authentication options Explanation of the Azure AD authentication options in the Visual Studio 2013 New-Project dialog box.
- Microsoft Patterns and Practices - Federated Identity Pattern.
- HowTo: Install the Windows Azure Active Directory Sync Tool
- Active Directory Federation Services 2.0 Content Map Links to documentation about ADFS 2.0.

Chapter 6

Data storage options

Most developers are used to relational databases, and they tend to overlook other data storage options when they're designing a cloud app. The result can be suboptimal performance, high expenses, or worse, because NoSQL (nonrelational) databases can handle some tasks more efficiently than relational databases can. When an organization encounters a critical data storage problem, it's often because it is using a relational database in a situation where one of the NoSQL options would have worked better. In these situations, the organization would have been better off if it had implemented a NoSQL solution before deploying its app to production.

On the other hand, it is also a mistake to assume that a NoSQL database can do everything well or well enough. There is no single best data management choice for all data storage tasks; different data management solutions are optimized for different tasks. Most real-world cloud apps have a variety of data storage requirements and are often served best by a combination of data storage solutions.

The purpose of this chapter is to give you a broader sense of the data storage options available to a cloud app and some basic guidance on how to choose the options that fit your scenario. It's best to be aware of the options available to you and to think about their strengths and weaknesses before you develop an application. Changing data storage options in a production app can be extremely difficult—like having to change a jet engine while the plane is in flight.

Data storage options on Azure

The cloud makes it relatively easy to use a variety of relational and NoSQL data stores. Here are some of the data storage platforms that you can use in Azure.

Relational	Key/Value	Column Family	Document	Graph
• Azure SQL Database • SQL Server • Oracle • MySQL • SQL Compact • SQLite • Postgres	• Azure Blob Storage • Azure Table Storage • Azure Cache • Redis • Memcached • Riak	• Cassandra • HBase	• MongoDB • RavenDB • CouchDB	• Neo4J

The illustration shows four types of NoSQL databases:

- Key/value databases store a single serialized object for each key value. They're good for storing large volumes of data in situations where you want to get one item for a given key value and you don't have to query based on other properties of the item.
- Azure Blob storage is a key/value database that functions like file storage in the cloud, with key values that correspond to folder and file names. You retrieve a file by its folder and file name, not by searching for values in the file contents.
- Azure Table storage is also a key/value database. Each value is called an entity (similar to a row, identified by a partition key and row key) and contains multiple properties (similar to columns, but not all entities in a table have to share the same columns). Querying on columns other than the key is extremely inefficient and should be avoided. For example, you can store user profile data, with one partition storing information about a single user. You could store data such as user name, password hash, birth date, and so forth, in separate properties of one entity or in separate entities in the same partition. But you wouldn't want to query for all users with a given range of birth dates, and you can't execute a join query between your profile table and another table. Table storage is more scalable and less expensive than a relational database, but it doesn't enable complex queries or joins.
- Document databases are key/value databases in which the values are documents. "Document" here isn't used in the sense of a Word or an Excel document but means a collection of named fields and values, any of which could be a child document. For example, in an order history table, an order document might have order number, order date, and customer fields, and the customer field might have name and address fields. The database encodes field data in a format such as XML, YAML, JSON, or BSON, or it can use plain text. One feature that sets document databases apart from other key/value databases is the capability they provide to query on nonkey fields and define secondary indexes, which makes querying more efficient. This capability makes a document database more suitable for applications that need to retrieve data on the basis of criteria more complex than the value of the document key. For example, in a sales order history document database, you could query on various fields, such as product ID, customer ID, customer name, and so forth. MongoDB is a popular document database.
- Column-family databases are key/value data stores that enable you to structure data storage into collections of related columns called *column families*. For example, a census database might have one group of columns for a person's name (first, middle, last), one group for the person's address, and one group for the person's profile information (date of birth, gender, and so on). The database can then store each column family in a separate partition while keeping all of the data for one person related to the same key. You can then read all profile information without having to read through all of the name and address information as well. Cassandra is a popular column-family database.

- Graph databases store information as a collection of objects and relationships. The purpose of a graph database is to enable an application to efficiently perform queries that traverse the network of objects and the relationships between them. For example, the objects might be employees in a human resources database, and you might want to facilitate queries such as "find all employees who directly or indirectly work for Scott." Neo4j is a popular graph database.

Compared with relational databases, the NoSQL options offer far greater scalability and are more cost effective for storage and analysis of unstructured data. The tradeoff is that they don't provide the rich querying and robust data integrity capabilities of relational databases. NoSQL options would work well for IIS log data, which involves high volume with no need for join queries. NoSQL options would not work so well for banking transactions, which require absolute data integrity and involve many relationships to other account-related data.

A newer category of database platforms, called NewSQL, combines the scalability of a NoSQL database with the querying capability and transactional integrity of a relational database.

NewSQL databases are designed for distributed storage and query processing, which are often hard to implement in "OldSQL" databases. NuoDB is an example of a NewSQL database that can be used on Azure.

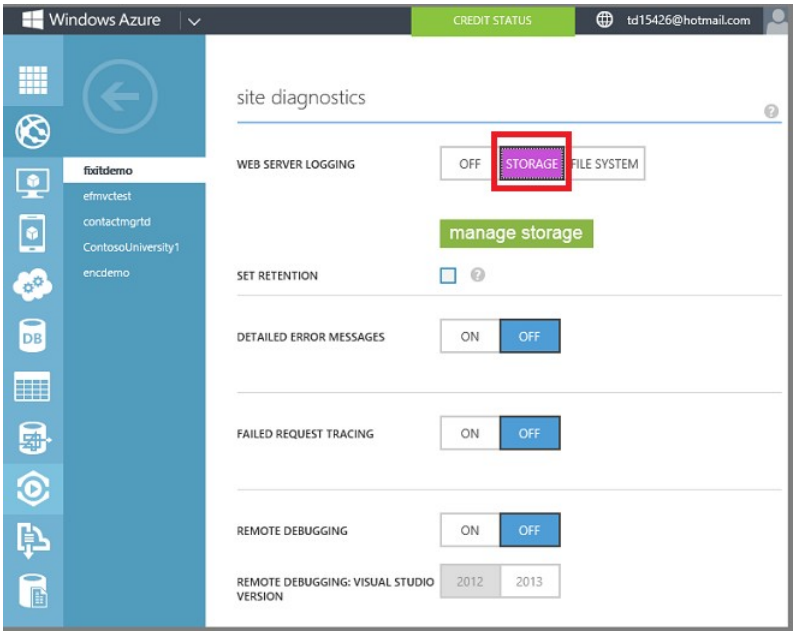
Hadoop and MapReduce

The high volumes of data that you can store in NoSQL databases may be difficult to analyze efficiently in a timely manner. To perform this type of analysis, you can use a framework such as Hadoop, which implements MapReduce functionality. Essentially, what a MapReduce process does is the following:

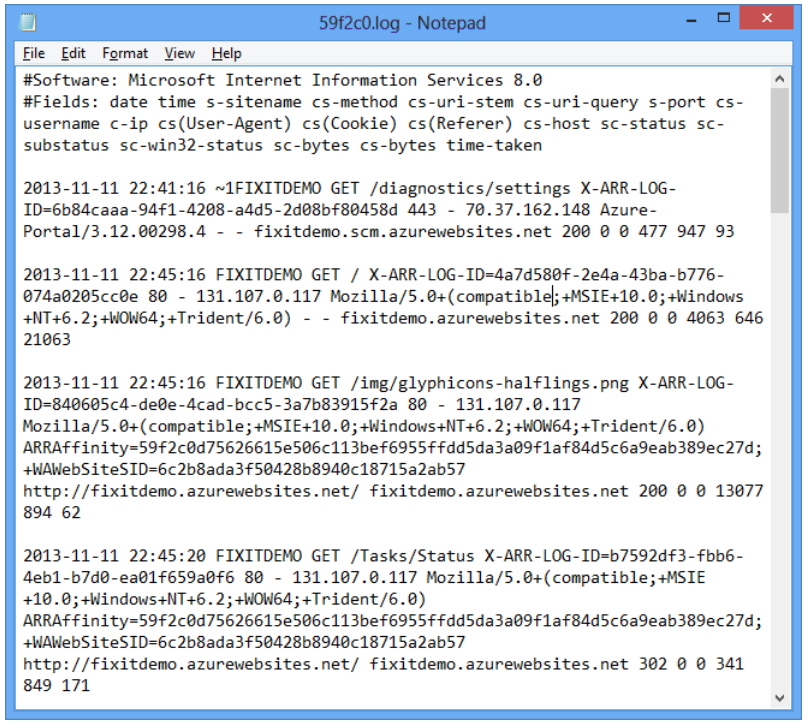
- Limits the size of the data that needs to be processed by selecting out of the data store only the data you actually need to analyze. For example, if you want to know the makeup of your user base by birth year, the process selects only birth years out of your user profile data store.
- Breaks down the data into parts and sends them to different computers for processing. Computer A calculates the number of people with dates between 1950 and 1959, computer B works on dates between 1960 and 1969, and so on. This group of computers is called a Hadoop cluster.
- Puts the results of each part back together after the processing on the parts is complete. You now have a relatively short list of how many people have each birth year, and the task of calculating percentages in this overall list is manageable.

On Azure, HDInsight enables you to process, analyze, and gain new insights from big data by using the power of Hadoop. For example, you could use HDInsight to analyze web server logs in the following manner:

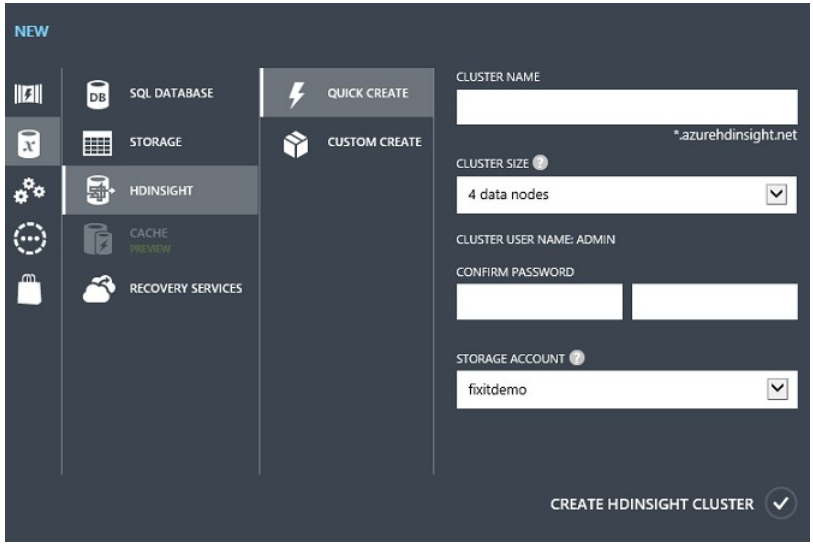
- Enable web server logging to your storage account. This sets up Azure to write logs to the Blob service for every HTTP request to your application. The Blob service is basically cloud file storage and integrates nicely with HDInsight.



- As the app gets traffic, web server IIS logs are written to Blob storage.



- In the Azure management portal, click **New**, **Data Services**, **HDInsight**, **Quick Create**, and then specify an HDInsight cluster name, cluster size (number of HDInsight cluster data nodes), and a user name and password for the HDInsight cluster.



You can now set up MapReduce jobs to analyze your logs and get answers to questions such as:

- What times of day does my app get the most or least traffic?
- What countries is my traffic coming from?
- What is the average neighborhood income of the areas my traffic comes from? (There's a public dataset that provides neighborhood income by IP address, and you can match that data against the IP addresses in the web server logs.)
- How does neighborhood income correlate to specific pages or products in the site?

You could then use the answers to questions such as these to target ads based on the likelihood that a customer would be interested in or would be likely to buy a particular product.

As explained in Chapter 1, "Automate everything," most functions that you can perform in the management portal can be automated, and that includes setting up and executing HDInsight analysis jobs. A typical HDInsight script might contain the following steps:

- Provision an HDInsight cluster and link it to your storage account for Blob storage input.
- Upload the MapReduce job executables (.jar or .exe files) to the HDInsight cluster.
- Submit a MapReduce job that stores the output data to Blob storage.
- Wait for the job to complete.
- Delete the HDInsight cluster.
- Access the output from Blob storage.

By running a script that performs these steps, you minimize the amount of time that the HDInsight cluster is provisioned, which minimizes your costs.

Platform as a Service (PaaS) versus Infrastructure as a Service (IaaS)

The data storage options listed earlier include both Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) solutions.

In a PaaS solution, Microsoft manages the hardware and software infrastructure and you just use the service. SQL Database is a PaaS feature of Azure. You ask for databases, and behind the scenes Azure sets up and configures the virtual machines (VMs) and sets up the databases on them. You don't have direct access to the VMs and don't have to manage them.

In an IaaS solution, you set up, configure, and manage VMs that run in Microsoft's data center

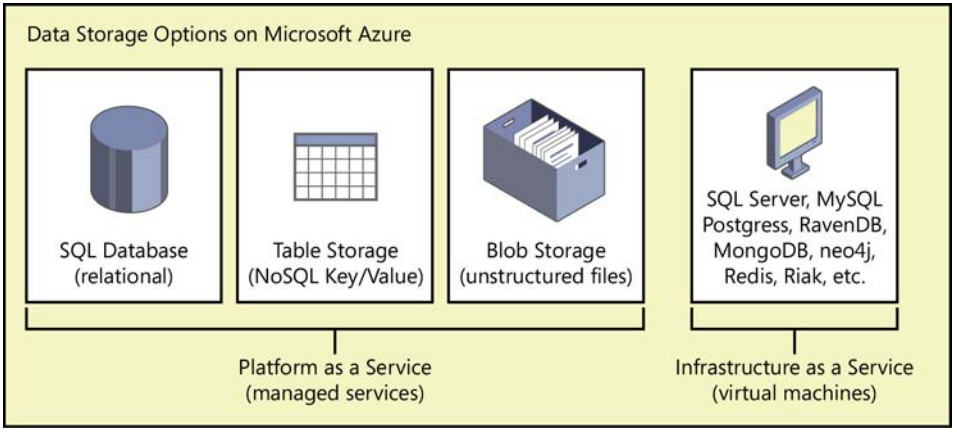
infrastructure, and you put whatever you want on them. Microsoft provides a gallery of preconfigured VM images for common VM configurations. For example, you can install preconfigured VM images for Windows Server 2008, Windows Server 2012, BizTalk Server, Oracle WebLogic Server, Oracle Database, and others.

PaaS data solutions that Azure offers include:

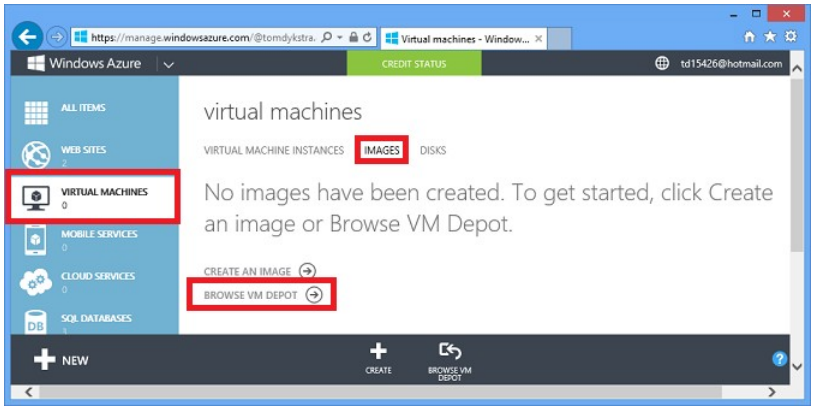
- Azure SQL Database (formerly known as SQL Azure) A cloud relational database based on SQL Server.
- Azure Table storage A column-oriented NoSQL database.
- Azure Blob storage File storage in the cloud.

For IaaS, you can run any software that you can load onto a VM, for example:

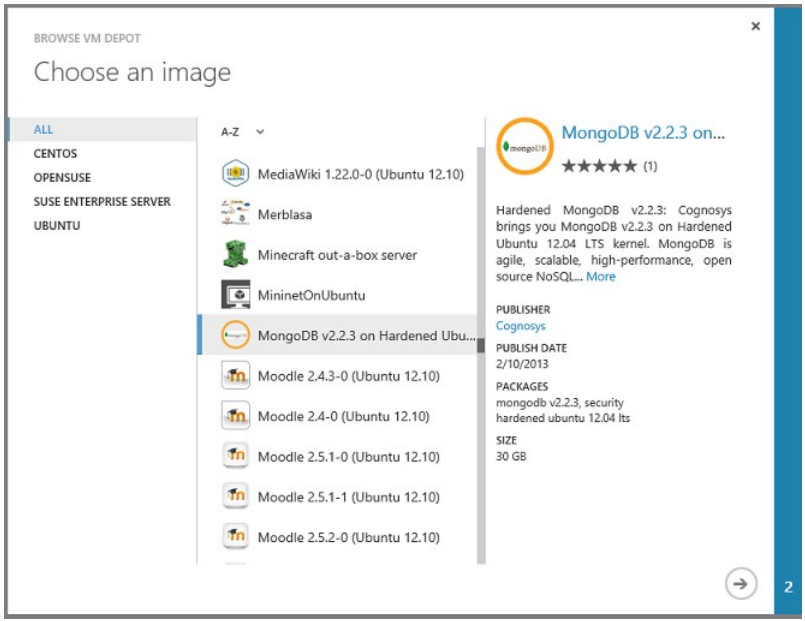
- Relational databases such as SQL Server, Oracle, MySQL, SQL Compact, SQLite, or Postgres.
- Key/value data stores such as Memcached, Redis, Cassandra, and Riak.
- Column data stores such as HBase.
- Document databases such as MongoDB, RavenDB, and CouchDB.
- Graph databases such as Neo4j.



The IaaS option gives you almost unlimited data storage options, and many of them are especially easy to use because you can create VMs using preconfigured images. For example, in the management portal, go to **Virtual Machines**, click the **Images** tab, and then click **Browse VM Depot**.



You then see a list of [hundreds of preconfigured VM images](), and you can create a VM from an image that has a database management system such as MongoDB, Neo4J, Redis, Cassandra, or CouchDB preinstalled:



Azure makes IaaS data storage options as easy to use as possible, but the PaaS offerings have many advantages that make them more cost-effective and practical for many scenarios:

- You don't have to create VMs; you just use the portal or a script to set up a data store. If you want a 200-terabyte data store, you just click a button or run a command, and in seconds it's ready for you to use.
- You don't have to manage or patch the VMs used by the service; Microsoft does that for you

automatically.

- You don't have to worry about setting up infrastructure for scaling or high availability; Microsoft handles all that for you.
- You don't have to buy licenses; license fees are included in the service fees.
- You pay only for what you use.

PaaS data storage options in Azure include offerings by third-party providers. For example, you can choose the MongoLab Add-On from the Azure Store to provision a MongoDB database as a service.

Choosing a data storage option

No one approach is right for all scenarios. If anyone says that a particular technology is the answer, the first thing to ask is "What is the question?" because different solutions are optimized for different things. The relational model has definite advantages; that's why it's been around for so long. But there are also downsides to SQL that can be addressed with a NoSQL solution.

Often, what we see work best is a composite approach in which SQL and NoSQL are used in a single solution. Even when people say they're embracing NoSQL, a closer looks reveals that they're using several different NoSQL frameworks—they're using CouchDB, Redis, and Riak for different things. Even Facebook, which uses NoSQL solutions extensively, uses different NoSQL frameworks for different parts of the service. The flexibility to mix and match data storage approaches is one of the qualities that's nice about the cloud; it's easy to use multiple data solutions and integrate them in a single app.

Here are some questions to think about when you're choosing an approach:

Data semantic	What is the core data storage and data access semantic (are you storing relational or unstructured data)? Unstructured data such as media files fits best in Blob storage; a collection of related data such as products, inventories, suppliers, customer orders, etc., fits best in a relational database.
Query support	How easy is it to query the data? What types of questions can be efficiently asked? Key/value data stores are very good at getting a single row when given a key value, but they are not so good for complex queries. For a user-profile data store in which you are always getting the data for one particular user, a key/value data store could work well. For a product catalog from which you want to get different groupings based on various product attributes, a relational database might work better. NoSQL databases can store large volumes of data efficiently, but you have to structure the database around how the app queries the data, and this makes ad hoc queries harder to do. With a relational database, you can build almost any kind of query.
Functional projection	Can questions, aggregations, and so on be executed on the server? If you run SELECT COUNT(*) from a table in SQL, the DBMS will very efficiently do all the work on the server and return the number you're looking for. If you want the same calculation from a NoSQL data store that doesn't support aggregation, this operation is an inefficient "unbounded query" and will probably time out. Even if the query succeeds, you have to retrieve all the data from the server and bring

	it to the client and count the rows on the client. What languages or types of expressions can be used? With a relational database, you can use SQL. With some NoSQL databases, such as Azure Table storage, you'll be using OData, and all you can do is filter on the primary key and get projections (select a subset of the available fields).
Ease of scalability	How often and how much will the data need to scale? Does the platform natively implement scale-out? How easy is it to add or remove capacity (size and throughput)? Relational databases and tables aren't automatically partitioned to make them scalable, so they are difficult to scale beyond certain limitations. NoSQL data stores such as Azure Table storage inherently partition everything, and there is almost no limit to adding partitions. You can readily scale Table storage up to 200 terabytes, but the maximum database size for Azure SQL Database is 500 gigabytes. You can scale relational data by partitioning it into multiple databases, but setting up an application to support that model involves a lot of programming work.
Instrumentation and Manageability	How easy is the platform to instrument, monitor, and manage? You need to remain informed about the health and performance of your data store, so you need to know up front what metrics a platform gives you for free and what you have to develop yourself.
Operations	How easy is the platform to deploy and run on Azure? PaaS? IaaS? Linux? Azure Table storage and Azure SQL Database are easy to set up on Azure. Platforms that aren't built-in Azure PaaS solutions require more effort.
API Support	Is an API available that makes it easy to work with the platform? The Azure Table Service has an SDK with a .NET API that supports the .NET 4.5 asynchronous programming model. If you're writing a .NET app, the work to write and test the code will be much easier for the Azure Table Service than for a key/value column data store platform that has no API or a less comprehensive one.
Transactional integrity and data consistency	Is it critical that the platform support transactions to guarantee data consistency? For keeping track of bulk emails sent, performance and low data-storage cost might be more important than automatic support for transactions or referential integrity in the data platform, making the Azure Table Service a good choice. For tracking bank account balances or purchase orders, a relational database platform that provides strong transactional guarantees would be a better choice.
Business continuity	How easy are backup, restore, and disaster recovery? Sooner or later production data will become corrupted and you'll need an undo function. Relational databases often have more fine-grained restore capabilities, such as the ability to restore to a point in time. Understanding what restore features are available in each platform you're considering is an important factor to consider.
Cost	If more than one platform can support your data workload, how do they compare in cost? For example, if you use ASP.NET Identity, you can store user profile data in Azure Table Service or Azure SQL Database. If you don't need the rich querying facilities of SQL Database, you might choose Azure Table storage in part because it costs much less for a given amount of storage.

Microsoft generally recommends that you should know the answer to the questions in each of these categories before you choose your data storage solutions.

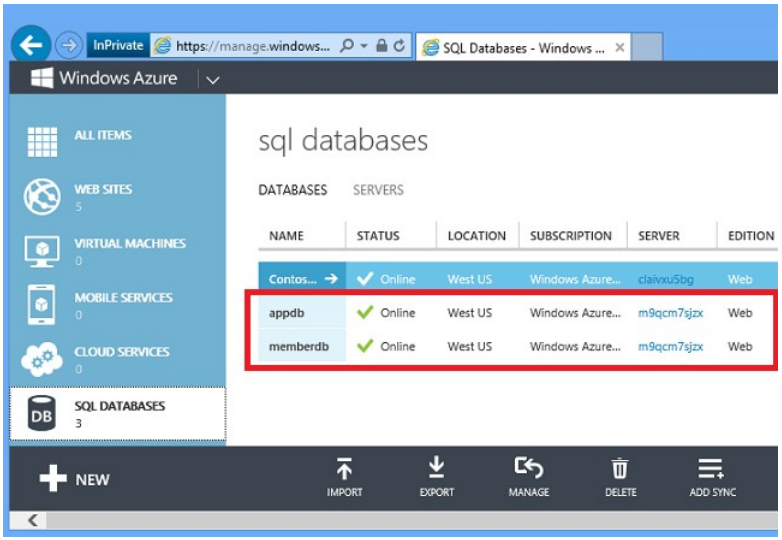
In addition, your workload might have specific requirements that some platforms can support better than others. For example:

- Does your application require audit capabilities?

- What are your data longevity requirements—do you require automated archival or purging capabilities?
- Do you have specialized security needs? For example, your data might include personally identifiable information (PII), but you have to be sure that PII is excluded from query results.
- If you have some data that can't be stored in the cloud for regulatory or technological reasons, you might need a cloud data storage platform that facilitates integration with your on-premises storage.

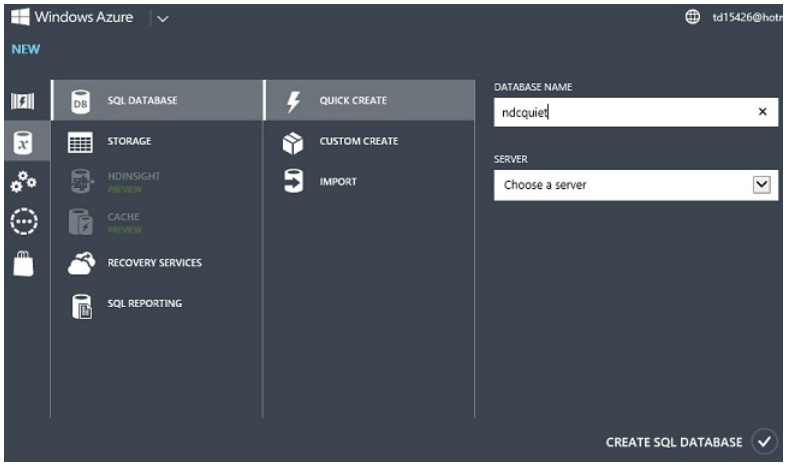
Demo—Using SQL Database in Azure

The Fix It app uses a relational database to store tasks. The Windows PowerShell environment creation script shown in Chapter 1 creates two SQL Database instances. You can see these in the portal by clicking the SQL Databases tab.

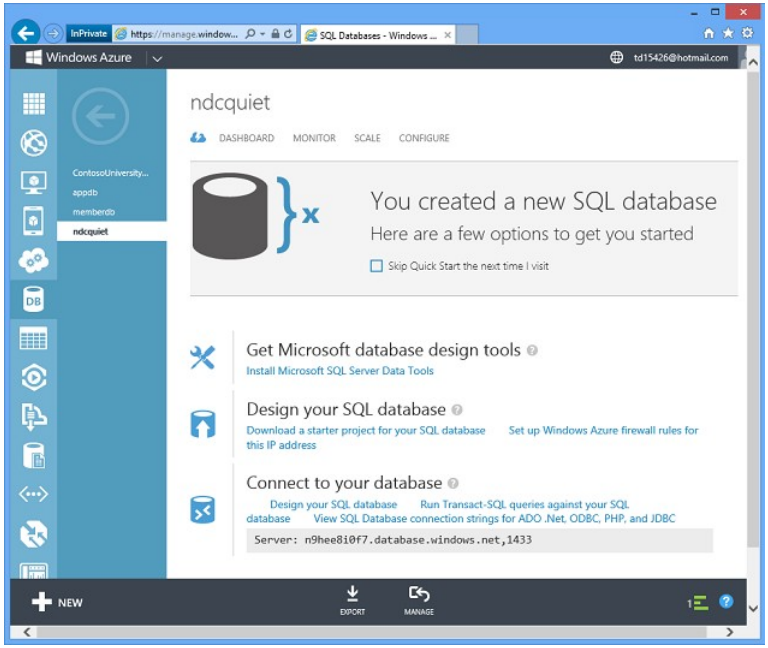


It's also easy to create databases by using the portal.

Click **New, Data Services, SQL Database, Quick Create**, enter a database name, choose a server you already have in your account or create a new one, and then click **Create SQL Database**.



Wait several seconds, and you have a database in Azure ready to use.



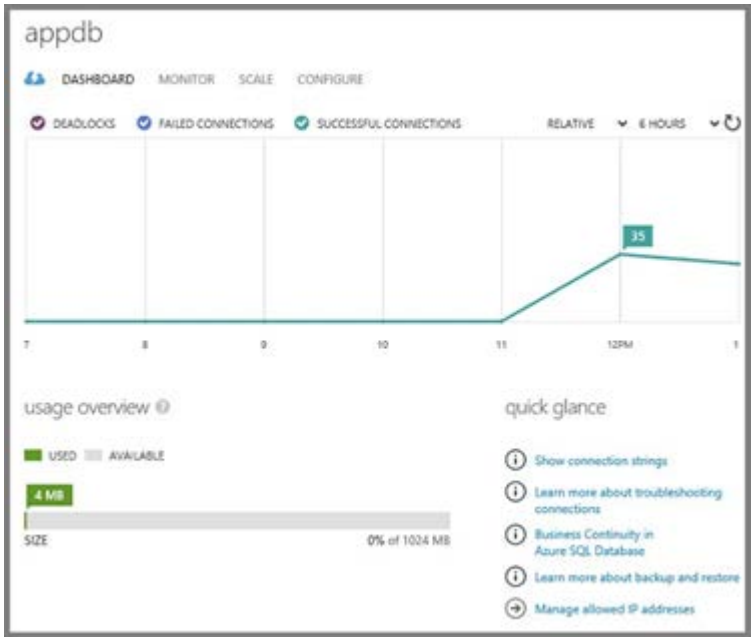
So, Azure does in a few seconds what it might take you a day, a week, or longer to accomplish in the on-premises environment. And since you can just as easily create databases automatically with a script or by using a management API, you can dynamically scale out by spreading your data across multiple databases, so long as your application has been programmed for that.

This is an example of Microsoft's Platform-as-a-Service model. You don't have to manage the servers, Microsoft does that. You don't have to worry about backups, Microsoft does it. The database is

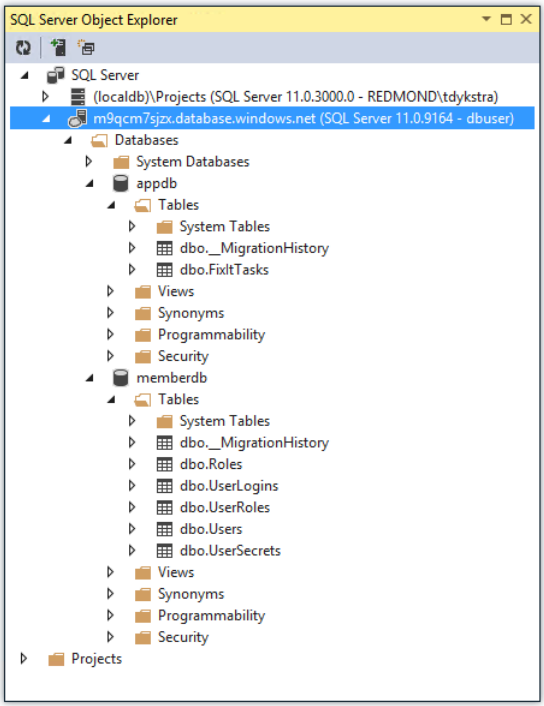
running in high availability, and the data in the database is replicated across three servers automatically. If a machine dies, it automatically fails over, and you lose no data. And, the server is patched regularly, you don't need to worry about that.

Click a button, and you get the exact connection string you need, and you can immediately start using the new database.

The Dashboard shows you the connection history and the amount of storage used.

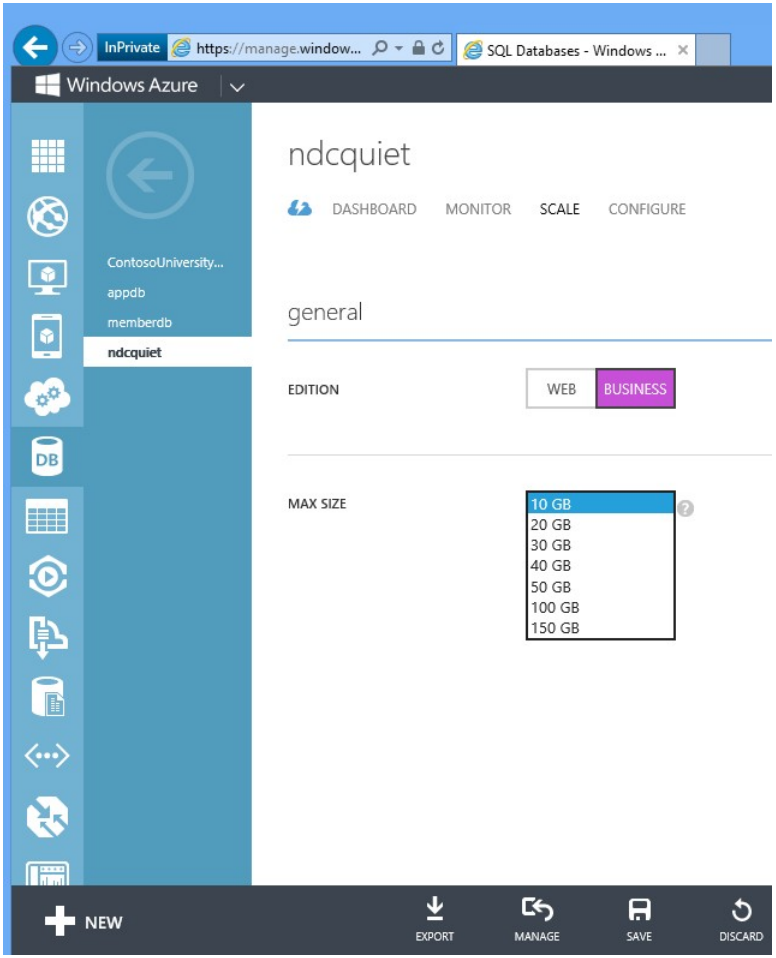


You can manage databases in the portal or by using SQL Server tools you're already familiar with, including SQL Server Management Studio (SSMS) and the Visual Studio tools SQL Server Object Explorer (SSOX) and Server Explorer.



Another nice feature is the pricing model. You can start development with a free 20-MB database, and a production database starts at about $5 per month. You pay only for the amount of data you actually store in the database, not the maximum capacity. You don't have to buy a license.

SQL Database is easy to scale. For the Fix It app, the database we create in our automation script is capped at 1 GB. If you want to scale up to 500 GB, you just go to the portal and change that setting, or you can execute a REST API command. In seconds, you have a 500-GB database that you can deploy data to.



That's the power of the cloud to stand up infrastructure quickly and easily and let you start using it immediately.

The Fix It app uses two SQL Databases, one for membership (authentication and authorization) and one for data, and this is all you have to do to provision and scale them. You saw earlier how to provision the databases through Windows PowerShell scripts, and now you've seen how easy it is to do this work in the portal.

Entity Framework versus direct database access using ADO.NET

The Fix It app accesses these databases by using the Entity Framework (EF), Microsoft's recommended

ORM (object-relational mapper) for .NET applications. An ORM is a great tool that facilitates developer productivity, but productivity comes at the expense of degraded performance in some scenarios. In a real-world cloud app you won't be making a choice between using EF or using ADO.NET directly—you'll use both. Most of the time when you're writing code that works with the database, getting maximum performance is not critical, and you can take advantage of the simplified coding and testing that you get with the Entity Framework. In situations where the EF overhead would cause unacceptable performance, you can write and execute your own queries using ADO.NET, ideally by calling stored procedures.

Whatever method you use to access the database, you want to minimize "chattiness" as much as possible. In other words, if you can get all the data you need in one larger query result set rather than dozens or hundreds of smaller ones, that's usually preferable. For example, if you need to list students and the courses they're enrolled in, it's usually better to get all of the data in one join query instead of getting the students in one query and executing separate queries for each student's courses.

SQL databases and the Entity Framework in the Fix It app

In the Fix It app, the `FixItContext` class, which derives from the Entity Framework `DbContext` class, identifies the database and specifies the tables in the database. The context specifies an entity set (table) for tasks, and the code passes in to the context the connection string name. That name refers to a connection string that is defined in the Web.config file.

```
public class MyFixItContext : DbContext
{
    public MyFixItContext()
    : base("name=appdb")
    {
    }

    public DbSet<MyFixIt.Persistence.FixItTask> FixItTasks { get; set; }
}
```

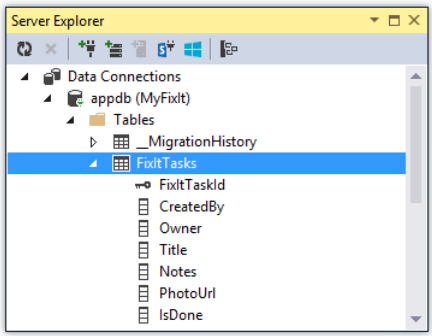
The connection string in the Web.config file is named *appdb* (here, pointing to the local development database):

```
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;Initial Catalog=aspnet-MyFixIt-
20130604091232_4;Integrated Security=True"
providerName="System.Data.SqlClient" />
    <add name="appdb" connectionString="Data Source=(localdb)\v11.0; Initial
Catalog=MyFixItContext-20130604091609_11;Integrated Security=True;
MultipleActiveResultSets=True" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

The Entity Framework creates a FixItTasks table based on the properties included in the `FixItTask`

entity class. This is a simple POCO (Plain Old CLR Object) class, which means it doesn't inherit from or have any dependencies on the Entity Framework. But the Entity Framework knows how to create a table based on it and execute CRUD (create-read-update-delete) operations with it.

```
public class FixItTask
{
    public int FixItTaskId   { get; set; }
    public string CreatedBy  { get; set; }
    [Required]
    public string Owner      { get; set; }
    [Required]
    public string Title      { get; set; }
    public string Notes      { get; set; }
    public string PhotoUrl   { get; set; }
    public bool IsDone       { get; set; }
}
```



The Fix It app includes a repository interface that it uses for CRUD operations working with the data store.

```
public interface IFixItTaskRepository
{
    Task<List<FixItTask>> FindOpenTasksByOwnerAsync(string userName);
    Task<List<FixItTask>> FindTasksByCreatorAsync(string userName);
    Task<MyFixIt.Persistence.FixItTask> FindTaskByIdAsync(int id);
    Task CreateAsync(FixItTask taskToAdd);
    Task UpdateAsync(FixItTask taskToSave);
    Task DeleteAsync(int id);
}
```

Notice that the repository methods are all async, so all data access can be done in a completely asynchronous way.

The repository implementation calls Entity Framework async methods to work with the data, including LINQ queries as well as for performing insert, update, and delete operations. Here's an example of the code for looking up a Fix It task.

```
public async Task<FixItTask> FindTaskByIdAsync(int id)
```

```
{
    FixItTask fixItTask = null;
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        fixItTask = await db.FixItTasks.FindAsync(id);

        timespan.Stop();
        log.TraceApi("SQL Database", "FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed,
          "id={0}", id);
    }
    catch(Exception e)
    {
        log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
    }

    return fixItTask;
}
```

You'll notice that there's also some timing and error-logging code here. We'll look at that code later in Chapter 10, "Monitoring and telemetry."

Choosing SQL Database (PaaS) versus SQL Server in a VM (IaaS) in Azure

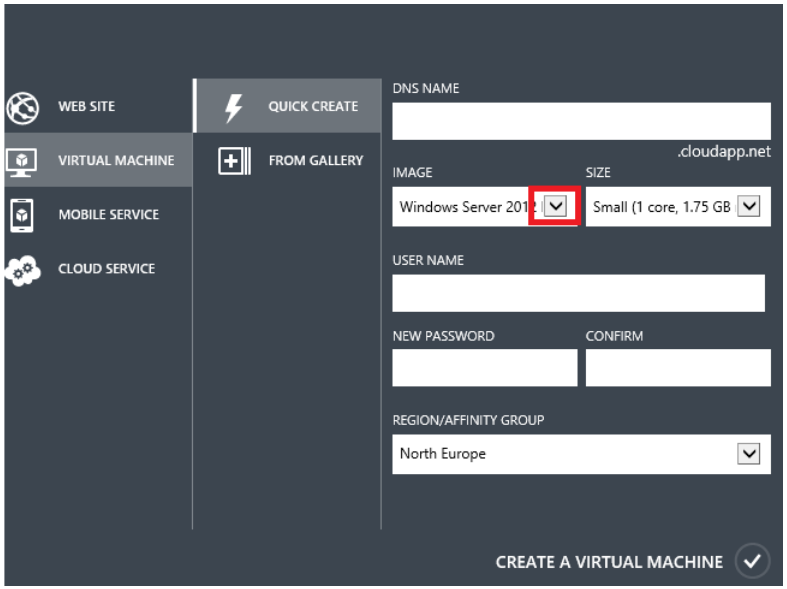
A nice thing about SQL Server and Azure SQL Database is that the core programming model for both is identical. You can use most of the same skills in both environments. You can even use a SQL Server database in development and a SQL Database instance in the cloud, which is how the Fix It app is set up.

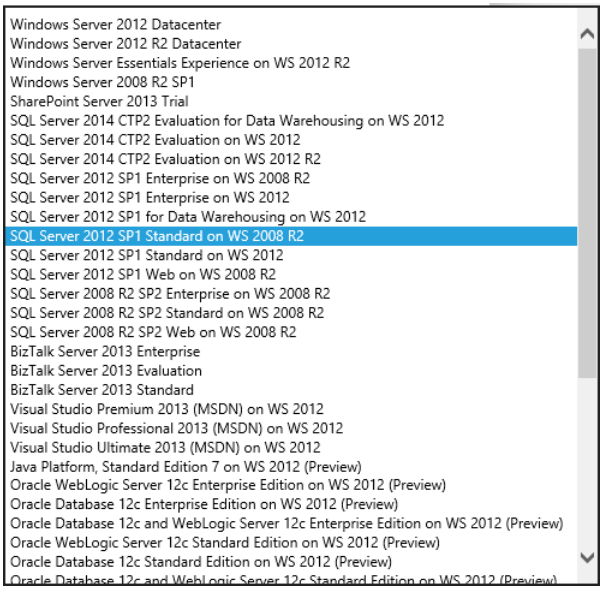
As an alternative, you can run the same edition of SQL Server in the cloud that you run on-premises by installing it on IaaS VMs. For some legacy applications, running SQL Server in a VM might be a better solution. Because a SQL Server database runs on a dedicated VM, it has more resources available to it than does a SQL Database database that runs on a shared server. That means a SQL Server database can be larger and still perform well. In general, the smaller the database size and table size, the better the use case works for SQL Database (PaaS).

Here are some guidelines on how to choose between the two models.

Azure SQL Database (PaaS)	SQL Server in a Virtual Machine (IaaS)
Pros You don't have to create or manage VMs or update or patch the OS or SQL Server; Azure does that for you. Built-in High Availability, with a database-level SLA. Low total cost of ownership (TCO) because you pay only for what you use (no license required). Good for handling large numbers of smaller databases (<=500 GB each). Easy to dynamically create new databases to enable scale-out.	Pros Feature compatible with on-premises SQL Server. Can implement SQL Server High Availability via AlwaysOn in two or more VMs, with VM-level SLA. You have complete control over how SQL Server is managed. Can reuse SQL Server licenses you already own or pay by the hour for one. Good for handling fewer but larger (1 TB+)
Cons Some feature gaps compared with on-premises SQL Server (lack of CLR integration, TDE, compression support, SQL Server Reporting Services, etc.) Database size limit of 500 GB.	Cons Updates/patches (OS and SQL) are your responsibility. Creation and management of DBs are your responsibility. Disk IOPS (input/output operations per second) limited to about 8,000 (via 16 data drives).

If you want to use SQL Server in a VM, you can use your own SQL Server license, or you can pay for one by the hour. For example, in the portal or via the REST API, you can create a new VM using a SQL Server image.





When you create a VM with a SQL Server image, Microsoft prorates the SQL Server license cost by the hour on the basis of your use of the VM. If you have a project that's going to run only for a couple of months, it's cheaper to pay by the hour. If you think your project is going to last for years, it's cheaper to buy the license the way you normally do.

Summary

Cloud computing makes it practical to mix and match data storage approaches to best fit the needs of your application. If you're building a new application, think carefully about the questions listed in this chapter to pick approaches that will continue to work well when your application grows. The next chapter explains some partitioning strategies that you can use to combine multiple data storage approaches.

Resources

For more information, see the following resources.

Choosing a database platform:

- Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence Ebook by Microsoft Patterns & Practices that goes in depth into the different kinds of data stores available for cloud applications.

- Microsoft Patterns and Practices - Cloud Development. See Data Consistency Primer, Data Replication and Synchronization Guidance, Index Table pattern, Materialized View pattern.
- BASE: An Acid Alternative Article about trade-offs between data consistency and scalability.
- Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement Book by Eric Redmond and Jim R. Wilson. Highly recommended for introducing yourself to the range of data storage platforms available today.

Choosing between SQL Server and SQL Database:

- Basic, Standard, and Premium Preview for Azure SQL Database An introduction to SQL Database Premium, and guidance on when to choose it over the SQL Database Web and Business editions.
- Azure SQL Database Guidelines and Limitations Portal page that links to documentation about limitations of SQL Database, including one that focuses on SQL Server features that SQL Database doesn't support.
- SQL Server in Azure Virtual Machines Portal page that links to documentation about running SQL Server in Azure.
- Scott Guthrie explains SQL Databases in Azure Six-minute video introduction to SQL Database by Scott Guthrie.
- Application Patterns and Development Strategies for SQL Server in Azure Virtual Machines. MSDN technical article.

Using Entity Framework and SQL Database in an ASP.NET Web app:

- Getting Started with EF 6 using MVC 5 Nine-part tutorial that walks through building an MVC app that uses the Entity Framework and deploys the database to Azure and SQL Database.
- ASP.NET Web Deployment using Visual Studio Twelve-part tutorial that goes into more depth about how to deploy a database by using EF Code First.
- Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to an Azure Web Site Step-by-step tutorial that walks through creating a web app that uses authentication, stores application tables in the membership database, modifies the database schema, and deploys the app to Azure.
- ASP.NET Data Access—Recommended Resources Links to resources for working with EF and SQL Database.

Using MongoDB on Azure:

- MongoLab—MongoDB on Windows Azure Portal page for documentation about running MongoDB on Azure.

- Create an Azure web site that connects to MongoDB running on a virtual machine in Azure
  Step-by-step tutorial that shows how to use a MongoDB database in an ASP.NET web application.

HDInsight (Hadoop on Azure):

- HDInsight Portal to HDInsight documentation on azure.microsoft.com.
- Hadoop and HDInsight: Big Data in Windows Azure *MSDN Magazine* article by Bruno Terkaly and Ricardo Villalobos, introducing Hadoop on Azure.
- Microsoft Patterns & Practices—Azure Guidance See the MapReduce pattern.

Chapter 7

Data partitioning strategies

Earlier, we showed how easy it is to scale the web tier of a cloud application by adding and removing web servers. But if the servers are all hitting the same data store, your application's bottleneck moves from the front end to the back-end data tier, and the data tier is the hardest to scale. In this chapter we look at how you can make your data tier scalable by partitioning data into multiple relational databases or by combining relational database storage with other data storage options.

Setting up a partitioning scheme is best done up front for the same reason mentioned earlier: it's very difficult to change your data storage strategy after an app is in production. If you think carefully beforehand about different approaches, you can avoid having a "Twitter moment," when your app crashes or goes down for a long time while you reorganize your app's data and data access code.

The three Vs of data storage

To determine whether you need a partitioning strategy and what it should be, consider three questions about your data:

- **Volume** How much data will you ultimately store? A couple gigabytes? A couple hundred gigabytes? Terabytes? Petabytes?
- **Velocity** What is the rate at which your data will grow? Is it an internal app that isn't generating a lot of data? An external app to which customers will be uploading images and videos?
- **Variety** What type of data will you store? Relational, images, key-value pairs, social graphs?

If you think you're going to have a lot of volume, velocity, or variety, you have to carefully consider what kind of partitioning scheme will best enable your app to scale efficiently and effectively as it grows, and to ensure that you don't run into any bottlenecks.

There are basically three approaches to partitioning:

- Vertical partitioning
- Horizontal partitioning
- Hybrid partitioning

Vertical partitioning

Vertical portioning is like splitting up a table by columns: one set of columns goes into one data store, and another set of columns goes into a different data store.

For example, suppose my app stores data about people, including images:

FirstName	LastName	Email	Thumbnail	Photo
Bobbie	Angelo	bobbiea@contoso.com	3kb	3MB
Stacy	Heyward	staceyh@contoso.com	3kb	3MB
Claude	Vanburen	claudev@contoso.com	3kb	3MB
Marla	Weber	marlaw@contoso.com	3kb	3MB
Fay	Whitehead	fayw@contoso.com	3kb	3MB

When you represent this data as a table and look at the different varieties of data, you can see that the three columns on the left have string data that can be efficiently stored by a relational database, whereas the two columns on the right are essentially byte arrays that come from image files. It's possible to storage image-file data in a relational database, and a lot of people do that because they don't want to save the data to the file system. They might not have a file system capable of storing the required volumes of data, or they might not want to manage a separate backup and restore system. This approach works well for on-premises databases and for small amounts of data in cloud databases. In the on-premises environment, it might be easier to just let the database administrator (DBA) take care of everything.

But in a cloud database, storage is relatively expensive, and a high volume of images could make the size of the database grow beyond the limits at which it can operate efficiently. You can address these problems by partitioning the data vertically, which means you choose the most appropriate data store for each column in your table of data. What might work best for this example is to put the string data in a relational database and the images in Blob storage.

FirstName	LastName	Email	Thumbnail	Photo
Bobbie	Angelo	bobbiea@contoso.com	3kb	3MB
Stacy	Heyward	staceyh@contoso.com	3kb	3MB
Claude	Vanburen	claudev@contoso.com	3kb	3MB
Marla	Weber	marlaw@contoso.com	3kb	3MB
Fay	Whitehead	fayw@contoso.com	3kb	3MB

SQL Database BLOBS

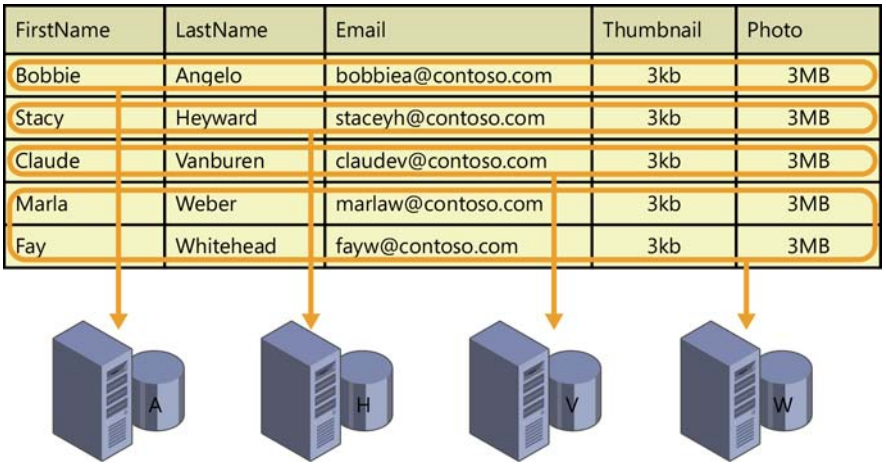
Storing images in Blob storage instead of in a database is more practical in the cloud than in an on-premises environment because you don't have to worry about setting up file servers or managing backup and restore of data stored outside the relational database: all that is handled for you by the Blob storage service.

This is the partitioning approach we implemented in the Fix It app, and we'll look at the code for that in Chapter 8, "Unstructured blob storage." Without this partitioning scheme, and assuming an average image size of 3 megabytes (MB), the Fix It app would be able to store only about 40,000 tasks before it hit the maximum database size of 150 gigabytes. After removing the images, the database can store 10 times as many tasks; the application can handle a much larger number of people before you have to think about implementing a horizontal partitioning scheme. And as the app scales, your expenses grow more slowly because the bulk of your storage needs are going into very inexpensive Blob storage.

Horizontal partitioning (sharding)

Horizontal portioning is like splitting up a table by rows: one set of rows goes into one data store, and another set of rows goes into a different data store.

Given the same set of data, another option would be to store different ranges of customer names in different databases.

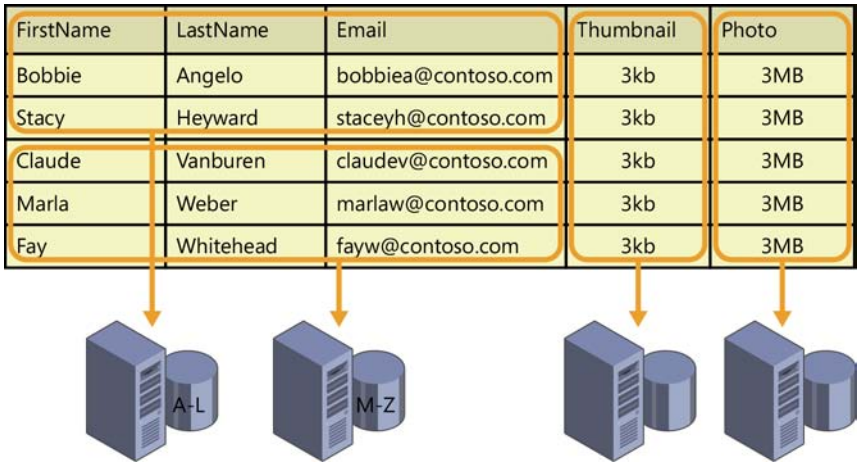


You want to be very careful about your sharding scheme to be sure that your data is evenly distributed to avoid hot spots. This simple example (using the first letter of the last name) doesn't meet that requirement because a lot of peoples' last names start with common letters. You'd hit table size limitations earlier than you might expect because some databases would become very large while most would remain small.

A downside of horizontal partitioning is that it might be hard to run queries across all of the data. In this example, a query would have to draw from up to 26 databases to get all of the data stored by the app.

Hybrid partitioning

You can combine vertical and horizontal partitioning. For example, with the sample data we're using, you could store the images in Blob storage and horizontally partition the string data.



Partitioning a production application

Conceptually, it's easy to see how a partitioning scheme would work, but any partitioning scheme does increase code complexity and introduces many new complications that you have to deal with. If you're moving images to Blob storage, what do you do when the storage service is down? How do you handle blob security? What happens if the database and Blob storage get out of sync? If you're sharding, how will you handle querying across all of the databases?

The complications are manageable so long as you plan for them before you go into production. Many people who haven't done that wish they had later. On average, Microsoft's Customer Advisory Team (CAT) gets phone calls about once a month from panicked customers whose apps are taking off in a really big way, but they didn't do this kind of planning. They say something like, "Help! I put everything in a single data store, and in 45 days I'm going to run out of space on it!" And if you have a lot of business logic built into how you access your data store—and you have customers who are using your app—there's no good time to go down for a day while you migrate. Microsoft ends up going through herculean efforts to help the customer partition their data on the fly with no downtime. It's very exciting and very scary, and not something you want to be involved in if you can avoid it! Thinking about data partitioning up front and integrating it into your app will make your life a lot easier if the app grows later.

Summary

An effective partitioning scheme can enable your cloud app to scale to petabytes of data without bottlenecks. And you don't have to pay up front for massive machines or extensive infrastructure, as you might if you were running the app in an on-premises data center. In the cloud, you can

incrementally add capacity as you need it, and you pay only for as much as you use when you use it.

In the next chapter, you'll see how the Fix It app implements vertical partitioning by storing images in Blob storage.

Resources

For more information about partitioning strategies, see the following resources.

Documentation:

- Best Practices for the Design of Large-Scale Services on Azure Cloud Services White paper by Mark Simms and Michael Thomassy.
- Microsoft Patterns and Practices - Cloud Design Patterns See Data Partitioning guidance, Sharding pattern.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from the Microsoft Customer Advisory Team's experience with actual customers. See the partitioning discussion in episode 7.
- Building Big: Lessons learned from Windows Azure customers—Part I Mark Simms discusses partitioning schemes, sharding strategies, how to implement sharding, and SQL Database Federations, starting at 19:49. Similar to the Failsafe series but goes into more how-to details.

Sample code:

- Cloud Service Fundamentals in Windows Azure Sample application that includes a sharded database. For a description of the sharding scheme implemented, see DAL—Sharding of RDBMS on the Azure blog.

Chapter 8

Unstructured blob storage

In the previous chapter we looked at partitioning schemes and explained how the Fix It app stores images in the Azure Blob Storage service, and other task data in Azure SQL Database. In this chapter we go deeper into the Blob service and show how it's implemented in Fix It project code.

What is Blob storage?

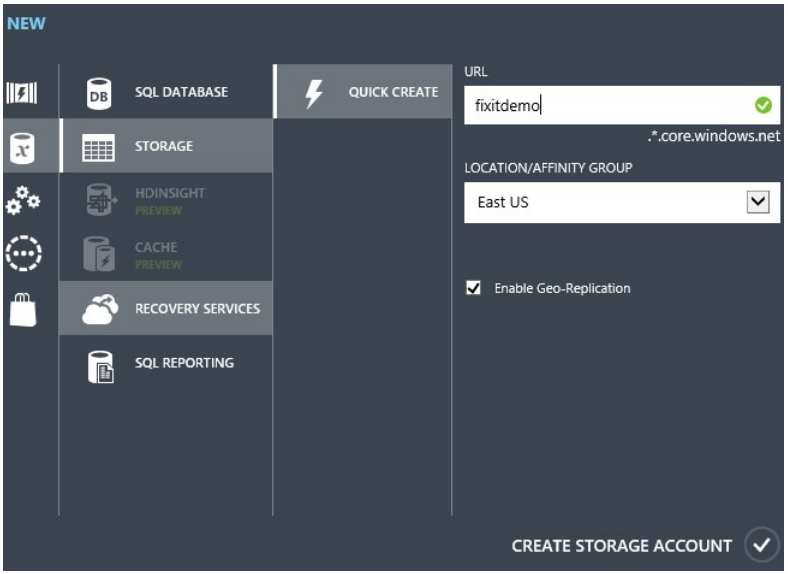
The Azure Blob Storage service provides a way to store files in the cloud. The Blob service has a number of advantages over storing files in a local network file system:

- It's highly scalable. A single storage account can store 100 terabytes, and you can have multiple storage accounts. Some of the biggest Azure customers store hundreds of petabytes. Microsoft OneDrive uses Blob storage.
- It's durable. Every file you store in the Blob service is automatically backed up.
- It provides high availability. The SLA for Storage promises 99.9 percent or 99.99 percent uptime, depending on which geo-redundancy option you choose.
- It's a platform-as-a-service (PaaS) feature of Azure, which means you just store and retrieve files, paying only for the actual amount of storage you use, and Azure automatically takes care of setting up and managing all of the VMs and disk drives required for the service.
- You can access the Blob service by using a REST API or by using a programming language API. SDKs are available for .NET, Java, Ruby, and other languages.
- When you store a file in the Blob service, you can easily make it publicly available over the Internet.
- You can secure files in the Blob service so that they can accessed only by authorized users, or you can provide temporary access tokens that makes the files available to someone only for a limited period of time.

Anytime you're building an app for Azure and you want to store a lot of data that in an on-premises environment would go in files—such as images, videos, PDFs, spreadsheets, and so on—consider the Blob service.

Creating a storage account

To get started with the Blob service you create a storage account in Azure. In the portal, click **New, Data Services, Storage, Quick Create,** and then enter a URL and a data center location. The data center location should be the same as your website.



Notice the **Enable Geo-Replication** check box under the location drop-down list. You pick the primary region where you want to store your content, and if you select this option, Azure creates backups of all your data in a different data center in another region of the country. For example, if you choose the West US data center, a file you store goes to the West US data center, but in the background Azure also copies it to one of the other US data centers. If a disaster happens in one region of the country, your data is still safe.

Azure won't replicate data across geopolitical boundaries: if your primary location is in the United States, your files are replicated only to another region within the United States; if your primary location is Australia, your files are replicated only to another data center in Australia.

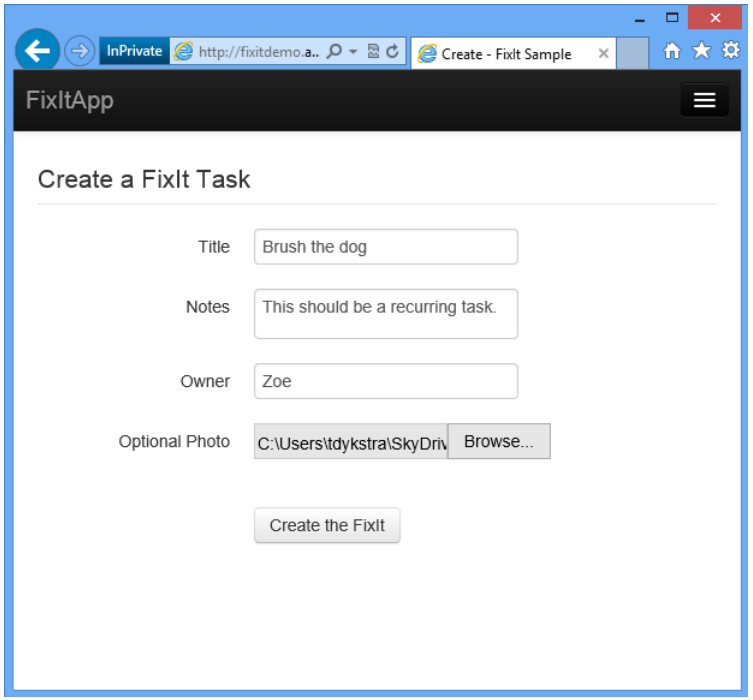
Of course, you can also create a storage account by executing commands from a script, as you saw earlier. Here's a Windows PowerShell command to create a storage account:

```
# Create a new storage account
New-AzureStorageAccount -StorageAccountName $Name -Location $Location - Verbose
```

Once you have a storage account, you can immediately start storing files in the Blob service.

Using Blob storage in the Fix It app

The Fix It app enables you to upload photos.



When you click **Create the FixIt**, the application uploads the specified image file and stores it in the Blob service.

Set up the Blob container

To store a file in the Blob service, you need a container to store it in. A Blob service container corresponds to a file system folder. The environment creation scripts that we reviewed in Chapter 1, "Automate everything," create the storage account, but they don't create a container. So the purpose of the `CreateAndConfigure` method of the `PhotoService` class is to create a container if it doesn't already exist. This method is called from the `Application_Start` method in Global.asax.

```
async public void CreateAndConfigureAsync()
{
    try
    {
        CloudStorageAccount storageAccount = StorageUtils.StorageAccount;

        // Create a blob client and retrieve reference to images container
```

```
        CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
        CloudBlobContainer container =
blobClient.GetContainerReference("images");

        // Create the "images" container if it doesn't already exist.
        if (await container.CreateIfNotExistsAsync())
        {
            // Enable public access on the newly created "images" container
            await container.SetPermissionsAsync(
                new BlobContainerPermissions
                {
                    PublicAccess = BlobContainerPublicAccessType.Blob
                });
        log.Information("Successfully created Blob Storage Images
Container and made it public");
        }
    }
    catch (Exception ex)
    {
        log.Error(ex, "Failure to Create or Configure images container in
Blob Storage Service");
    }
}
```

The storage account name and access key are stored in the `appSettings` collection of the Web.config file, and code in the `StorageUtils.StorageAccount` method uses those values to build a connection string and establish a connection:

```
string account = CloudConfigurationManager.GetSetting("StorageAccountName");
string key = CloudConfigurationManager.GetSetting("StorageAccountAccessKey");
string connectionString =
String.Format("DefaultEndpointsProtocol=https;AccountName={0};AccountKey={1}", account, key);
return CloudStorageAccount.Parse(connectionString);
```

The `CreateAndConfigureAsync` method then creates an object that represents the Blob service and an object that represents a container (folder) named "images" in the Blob service:

```
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
CloudBlobContainer container = blobClient.GetContainerReference("images");
```

If a container named "images" doesn't exist yet—which will be true the first time you run the app against a new storage account—the code creates the container and sets permissions to make it public. (By default, new blob containers are private and are accessible only to users who have permission to access your storage account.)

```
if (await container.CreateIfNotExistsAsync())
{
    // Enable public access on the newly created "images" container
    await container.SetPermissionsAsync(
        new BlobContainerPermissions
        {
            PublicAccess = BlobContainerPublicAccessType.Blob
```

```
        });

    log.Information("Successfully created Blob Storage Images Container and made it public");
}
```

Store the uploaded photo in Blob storage

To upload and save the image file, the app uses an `IPhotoService` interface and an implementation of the interface in the `PhotoService` class. The PhotoService.cs file contains all of the code in the Fix It app that communicates with the Blob service.

The following MVC controller method is called when the user clicks Create the FixIt. In this code, `photoService` refers to an instance of the `PhotoService` class, and `fixittask` refers to an instance of the `FixItTask` entity class that stores data for a new task.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create([Bind(Include =
    "FixItTaskId,CreatedBy,Owner,Title,Notes,PhotoUrl,IsDone")]FixItTask
    fixittask, HttpPostedFileBase photo)
{
    if (ModelState.IsValid)
    {
        fixittask.CreatedBy = User.Identity.Name;
        fixittask.PhotoUrl = await photoService.UploadPhotoAsync(photo);
        await fixItRepository.CreateAsync(fixittask);
        return RedirectToAction("Success");
    }

    return View(fixittask);
}
```

The `UploadPhotoAsync` method in the `PhotoService` class stores the uploaded file in the Blob service and returns a URL that points to the new blob.

```
async public Task<string> UploadPhotoAsync(HttpPostedFileBase photoToUpload)
{
    if (photoToUpload == null || photoToUpload.ContentLength == 0)
    {
        return null;
    }

    string fullPath = null;
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        CloudStorageAccount storageAccount = StorageUtils.StorageAccount;

        // Create the blob client and reference the container
        CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
```

```
        CloudBlobContainer container = blobClient.GetContainerReference("images");

        // Create a unique name for the images we are about to upload
        string imageName = String.Format("task-photo-{0}{1}",
            Guid.NewGuid().ToString(), Path.GetExtension(photoToUpload.FileName));

        // Upload image to Blob Storage
        CloudBlockBlob blockBlob = container.GetBlockBlobReference(imageName);
        blockBlob.Properties.ContentType = photoToUpload.ContentType;
        await blockBlob.UploadFromStreamAsync(photoToUpload.InputStream);

        // Convert to be HTTP based URI (default storage path is HTTPS)
        var uriBuilder = new UriBuilder(blockBlob.Uri);
        uriBuilder.Scheme = "http";
        fullPath = uriBuilder.ToString();

        timespan.Stop();
        log.TraceApi("Blob Service", "PhotoService.UploadPhoto", timespan.Elapsed,
            "imagepath={0}", fullPath);
    }
    catch (Exception ex)
    {
        log.Error(ex, "Error upload photo blob to storage");
    }

    return fullPath;
}
```

As in the `CreateAndConfigure` method, the code connects to the storage account and creates an object that represents the "images" blob container, except here it assumes the container already exists.

Then it creates a unique identifier for the image about to be uploaded, by concatenating a new GUID value with the file extension:

```
string imageName = String.Format("task-photo-{0}{1}",
    Guid.NewGuid().ToString(), Path.GetExtension(photoToUpload.FileName));
```

The code then uses the blob container object and the new unique identifier to create a blob object, sets an attribute on that object indicating what kind of file it is, and then uses the blob object to store the file in Blob storage.

```
CloudBlockBlob blockBlob = container.GetBlockBlobReference(imageName);
blockBlob.Properties.ContentType = photoToUpload.ContentType;
blockBlob.UploadFromStream(photoToUpload.InputStream);
```

Finally, it gets a URL that references the blob. This URL will be stored in the database and can be used in Fix It webpages to display the uploaded image.

```
fullPath = String.Format("http://{0}{1}", blockBlob.Uri.DnsSafeHost,
blockBlob.Uri.AbsolutePath);
```

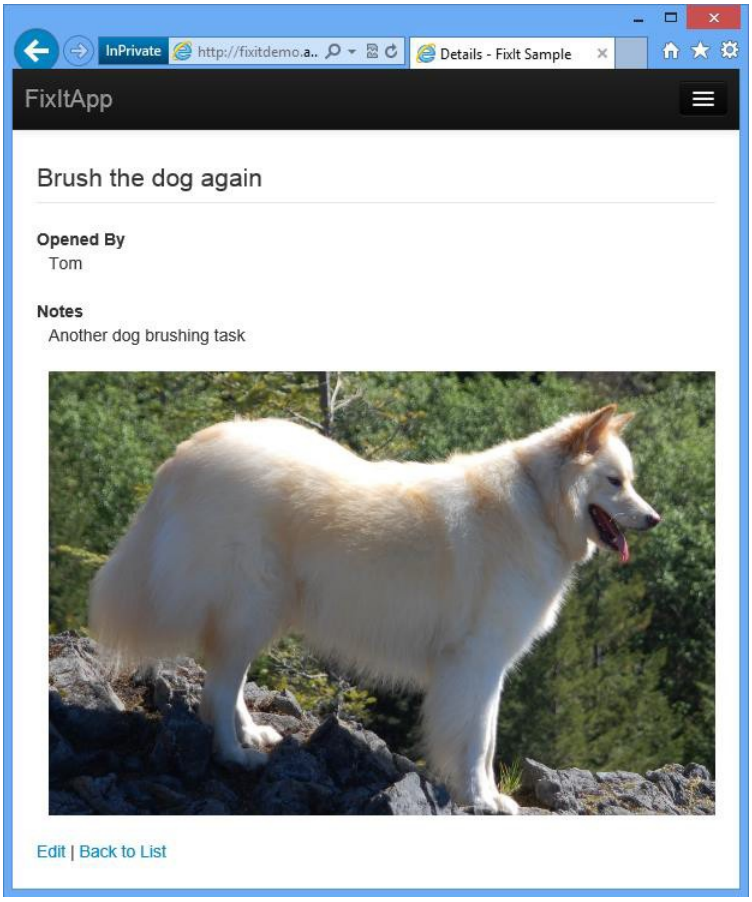
This URL is stored in the database as one of the columns of the FixItTask table.

```
public class FixItTask
{
    public int FixItTaskId { get; set; }
    public string CreatedBy { get; set; }
    [Required]
    public string Owner { get; set; }
    [Required]
    public string Title { get; set; }
    public string Notes { get; set; }
    public string PhotoUrl { get; set; }
    public bool IsDone     { get; set; }
}
```

With only the URL in the database, and images in Blob storage, the Fix It app keeps the database small, scalable, and inexpensive, while the images are stored where storage is cheap and capable of handling terabytes or petabytes. One storage account can store 100 terabytes of Fix It photos, and you only pay for what you use. So, you can start off small, paying nine cents for the first gigabyte, and add more images for pennies per additional gigabyte.

Display the uploaded file

The Fix It application displays the uploaded image file when it displays details for a task.



To display the image, all the MVC view has to do is include the `PhotoUrl` value in the HTML sent to the browser. The web server and the database are not using cycles to display the image, they are only serving up a few bytes to the image URL. In the following Razor code, `Model` refers to an instance of the `FixItTask` entity class.

```
<fieldset>
    <legend>@Html.DisplayFor(model => model.Title)</legend>
    <dl>
        <dt>Opened By</dt>
        <dd>@Html.DisplayFor(model => model.CreatedBy)</dd>
        <br />
        <dt>@Html.DisplayNameFor(model => model.Notes)</dt>
        <dd>@Html.DisplayFor(model => model.Notes)</dd>
        <br />
        @if(Model.PhotoUrl != null) {
            <dd><img src="@Model.PhotoUrl" title="@Model.Title"/></dd>
        }
    </dl>
```

```
</fieldset>
```

If you look at the HTML of the page that's displayed, you see the URL pointing directly to the image in Blob storage, something like this:

```
<fieldset>
    <legend>Brush the dog again</legend>
    <dl>
        <dt>Opened By</dt>
        <dd>Tom</dd>
        <br />
        <dt>Notes</dt>
        <dd>Another dog brushing task</dd>
        <br />
        <dd>
<img src="http://storageaccountname.blob.core.windows.net/images/task-photo-
312dd635-ba87-4542-8b15-767032c55f4e.jpg" title="Brush the dog again" />
        </dd>
    </dl>
</fieldset>
```

Summary

You've seen how the Fix It app stores images in the Blob service and only image URLs in the SQL database. Using the Blob service keeps the SQL database much smaller than it would be otherwise, makes it possible to scale up to an almost unlimited number of tasks, and can be done without writing a lot of code.

You can have hundreds of terabytes in a storage account, and the storage cost is much less expensive than SQL Database storage, starting at about three cents per gigabyte per month plus a small transaction charge. And keep in mind that you're not paying for the maximum capacity but only for the amount you actually store, so your app is ready to scale but you're not paying for all that extra capacity.

In the next chapter we'll talk about the importance of making a cloud app capable of gracefully handling failures.

Resources

For more information see the following resources:

- An Introduction to Azure BLOB Storage Blog by Mike Wood.
- How to use Blob Storage from .NET Official documentation on the azure.microsoft.com site. A brief introduction to Blob storage followed by code examples showing how to connect to Blob

storage, create containers, upload and download blobs, and so on.

- FailSafe: Building Scalable, Resilient Cloud Services Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from the Microsoft Customer Advisory Team's experience with actual customers. For a discussion of the Azure Storage service and blobs, see episode 5 starting at 35:13.
- Microsoft Patterns & Practices—Azure Guidance See Valet Key pattern.

Chapter 9

Design to survive failures

One of the things you have to think about when you build any type of application, but especially one that will run in the cloud, where lots of people will be using it, is how to design the app so that it can gracefully handle failures and continue to deliver value as much as possible. Given enough time, things are going to go wrong in any environment or any software system. How your app handles those situations determines how upset your customers will be and how much time you have to spend analyzing and fixing problems.

Types of failures

There are two basic categories of failures, which you'll want to handle differently:

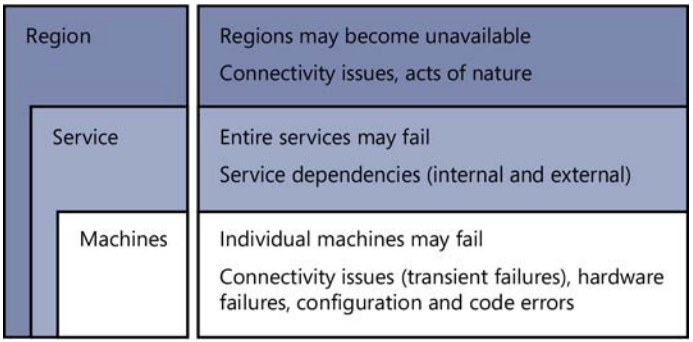
- Transient, self-healing failures such as intermittent network connectivity issues.
- Enduring failures that require intervention.

For transient failures, you can implement a retry policy to ensure that most of the time the app recovers quickly and automatically. Your customers might notice slightly longer response time, but otherwise they won't be affected. We'll show some ways to handle these errors in Chapter 11, "Transient fault handling."

For enduring failures, you can implement monitoring and logging functionality that notifies you promptly when issues arise and that facilitates root-cause analysis. We'll show some ways to help you stay on top of these kinds of errors in Chapter 10, "Monitoring and telemetry."

Failure scope

You also have to think about failure scope—whether a single machine is affected, a whole service such as SQL Database or Storage, or an entire region.



Machine failures

In Azure, a failed server is automatically replaced by a new one, and a well-designed cloud app recovers from this kind of failure automatically and quickly. Earlier, we stressed the scalability benefits of a stateless web tier, and ease of recovery from a failed server is another benefit of statelessness. Ease of recovery is also one of the benefits of platform-as-a-service (PaaS) features such as SQL Database and Websites. Hardware failures are rare, but when they occur, these services handle them automatically; you don't even have to write code to handle machine failures when you're using one of these services.

Service failures

Cloud apps typically use multiple services. For example, the Fix It app uses the SQL Database service and the Storage service, and it's deployed to the Websites service. What will your app do if one of the services you depend on fails? For some service failures a friendly "Sorry, try again later" message might be the best you can do. But in many scenarios you can do better. For example, when your back-end data store is down, you can accept user input, display "Your request has been received," and store the input someplace else temporarily. Then, when the service you need is operational again, you can retrieve the input and process it.

Chapter 13, "Queue-centric work pattern," shows one way to handle this scenario. The Fix It app stores tasks in SQL Database, but it doesn't have to quit working when SQL Database is down. In that chapter you'll see how to store user input for a task in a queue and use a worker process to read the queue and update the task. If SQL Database is down, the ability to create Fix It tasks is unaffected; the worker process can wait and process new tasks when SQL Database is available.

Region failures

Entire regions may fail. A natural disaster might destroy a data center—it might be flattened by a meteor, the trunk line into the datacenter could be cut by a farmer burying a cow with a backhoe, etc. If your app is hosted in the stricken data center, what do you do? It's possible to set up your app in

Azure to run in multiple regions simultaneously so that if a disaster occurs in one, your app continues running in another region. Such failures are extremely rare occurrences, and most apps don't jump through the hoops necessary to ensure uninterrupted service through failures of this sort. See the Resources section at the end of the chapter for information about how to keep your app available even through a region failure.

A goal of Azure is to make handling these kinds of failures a lot easier, and you'll see some examples of how Azure does that in the following chapters.

SLAs

People often hear about service-level agreements (SLAs) in the cloud environment. Basically, these are promises that companies make about how reliable their service is. A 99.9 percent SLA means you should expect the service to be working correctly 99.9 percent of the time. That's a fairly typical value for an SLA, and it sounds like a very high number, but you might not realize how much down time .1 percent actually amounts to. Here's a table that shows how much downtime various SLA percentages amount to over a year, a month, and a week.

Availability %	Downtime per year	Downtime per month*	Downtime per week
90% ("one nine")	36.5 days	72 hours	16.8 hours
99% ("two nines")	3.65 days	7.20 hours	1.68 hours
99.9% ("three nines")	8.76 hours	43.2 minutes	10.1 minutes
99.99% ("four nines")	52.56 minutes	4.32 minutes	1.01 minutes
99.999% ("five nines")	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% ("six nines")	31.5 seconds	2.59 seconds	0.605 seconds

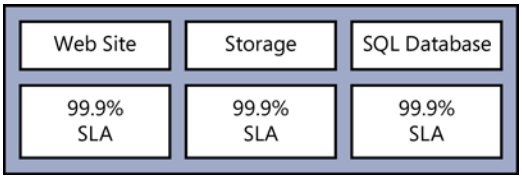
So a 99.9 percent SLA means your service could be down 8.76 hours a year or 43.2 minutes a month. That's more downtime than most people realize. As a developer, you want to be aware that a certain amount of downtime is possible and handle it in a graceful way. At some point someone is going to be using your app, and a service is going to be down, and you want to minimize the negative impact of that on the customer.

One thing you should know about an SLA is what time frame it refers to: is the clock reset every week, every month, or every year? In Azure, the clock is reset every month, which is better for you than a yearly SLA, since a yearly SLA could hide bad months by offsetting them with a series of good months.

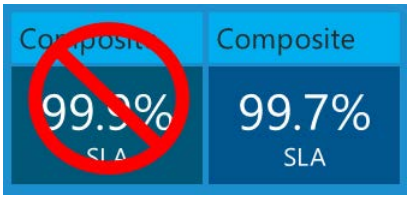
Of course, Microsoft aspires to do better than the SLA; usually, your app will be down much less time than what's shown in the previous table.. The promise is that if Azure's services are ever down for longer than the maximum downtime, you can ask for money back. The amount of money you get back probably won't fully compensate you for the business impact of the excess downtime, but that aspect of the SLA acts as an enforcement policy and lets you know that Microsoft does take its SLA levels very seriously.

Composite SLAs

An important thing to think about when you're looking at SLAs is the impact of using multiple services in an app, with each service having a separate SLA. For example, the Fix It app uses the Website, Storage, and SQL Database services. Here are their SLA numbers as of June 2014 (note that a 99.99% SLA is available for Storage at extra cost):



What is the maximum downtime you would expect for the app on the basis of these service SLAs? You might think that your downtime would be equal to the worst SLA percentage, or 99.9 percent in this case. That would be true if all three services always failed at the same time, but that isn't necessarily what actually happens. Each service may fail independently at different times, so you have to calculate the composite SLA by multiplying the individual SLA numbers.



This calculation means that your app could be down not just 43.2 minutes a month but three times that amount—108 minutes a month—and still be within the Azure SLA limits.

This issue is not unique to Azure. Microsoft actually offers the best cloud SLAs of any cloud service available, and you'll have similar issues to deal with if you use any vendor's cloud services. What this highlights is the importance of thinking about how you can design your app to handle the inevitable service failures gracefully, because they might happen often enough to impact your customers or users.

Cloud SLAs compared with enterprise downtime experience

People sometimes say, "In my enterprise app I never have these problems." If you ask how much downtime they actually have per month, they usually say, "Well, it happens occasionally." And if you ask how often, they admit that, "Sometimes we do need to back up or install a new server or update software." Of course, that counts as downtime. Most enterprise apps, unless they are especially mission-critical, are actually down for more than the amount of time allowed by Microsoft's service SLAs. But when it's your server and your infrastructure and you're responsible for it and in control of it, you tend to feel less angst about down times. In a cloud environment, you're dependent on someone

else, and you don't know what's going on, so you might tend to be more worried about it.

When an enterprise achieves a greater uptime percentage than comes with a cloud SLA, it does so by spending a lot more money on hardware. A cloud service could do that but would have to charge much more for its services. Instead, you take advantage of a cost-effective service and design your software so that the inevitable failures cause minimum disruption to your customers. Your job as a cloud app designer is not so much to avoid failure as to avoid catastrophe, and you do that by focusing on software, not on hardware. Whereas enterprise apps strive to maximize mean time between failures, cloud apps strive to minimize mean time to recover.

Not all cloud services have SLAs

Be aware also that not every cloud service even has an SLA. If your app is dependent on a service with no uptime guarantee, your app could be down far longer than you might imagine. For example, if you enable login to your site using a social provider such as Facebook or Twitter, check with the service provider to find out whether there is an SLA, and you might find there isn't one. But if the authentication service goes down or is unable to support the volume of requests you throw at it, your customers are locked out of your app. You could be down for days or longer. The creators of one new app expected hundreds of millions of downloads and took a dependency on Facebook authentication—but they didn't talk to Facebook before going live and discovered too late that there was no SLA for that service.

Not all downtime counts toward SLAs

Some cloud services may deliberately deny service if your app over uses them. This is called throttling. If a service has an SLA, it should state the conditions under which your app might be throttled, and your app design should avoid those conditions and react appropriately to the throttling if it happens. For example, if requests to a service start to fail when you exceed a certain number of requests per second, you want to be sure that automatic retries don't happen so fast that they cause the throttling to continue. We'll have more to say about throttling in Chapter 11.

Summary

This chapter has tried to help you realize why a real-world cloud app has to be designed to gracefully survive failures. Starting with the next chapter, the remaining patterns in this book go into more detail about some strategies you can use to do that:

- Have good monitoring and telemetry so that you find out quickly about failures that require intervention, and you have sufficient information to resolve them.
- Handle transient faults by implementing intelligent retry logic so that your app recovers automatically when it can and falls back to circuit-breaker logic when it can't.

- Use distributed caching to minimize throughput, latency, and connection problems with database access.
- Implement loose coupling via the queue-centric work pattern so that your app's front end can continue to work when the back end is down.

Resources

For more information, see later chapters in this ebook and the following resources.

Documentation:

- Failsafe: Guidance for Resilient Cloud Architectures White paper by Marc Mercuri, Ulrich Homann, and Andrew Townhill. Webpage version of the FailSafe video series.
- Best Practices for the Design of Large-Scale Services on Azure Cloud Services White paper by Mark Simms and Michael Thomassy.
- Azure Business Continuity Technical Guidance White paper by Patrick Wickline and Jason Roth.
- Disaster Recovery and High Availability for Azure Applications White paper by Michael McKeown, Hanu Kommalapati, and Jason Roth.
- Microsoft Patterns & Practices—Azure Guidance See Multi Data Center Deployment guidance, Circuit breaker pattern.
- Azure Support—Service Level Agreements.
- Azure SQL Database Business Continuity Documentation about SQL Database high-availability and disaster-recovery features.
- High Availability and Disaster Recovery for SQL Server in Azure Virtual Machines.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from the Microsoft Customer Advisory Team's experience with actual customers. Episodes 1 and 8 go in depth into the reasons for designing cloud apps to survive failures. See also the follow-up discussion of throttling in episode 2 starting at 49:57, the discussion of failure points and failure modes in episode 2 starting at 56:05, and the discussion of circuit breakers in episode 3 starting at 40:55.

- Building Big: Lessons learned from Windows Azure customers—Part II Mark Simms talks about designing for failure and instrumenting everything. Similar to the Failsafe series but goes into more how-to details.

Chapter 10

Monitoring and telemetry

A lot of people rely on customers to let them know when their application is down. That's not really a best practice anywhere, and especially not in the cloud. There's no guarantee of quick notification, and when you do get notified, you often get minimal or misleading data about what happened. With good telemetry and logging systems, you can be aware of what's going on with your app, and when something does go wrong, you find out right away and have helpful troubleshooting information to work with.

Buy or rent a telemetry solution

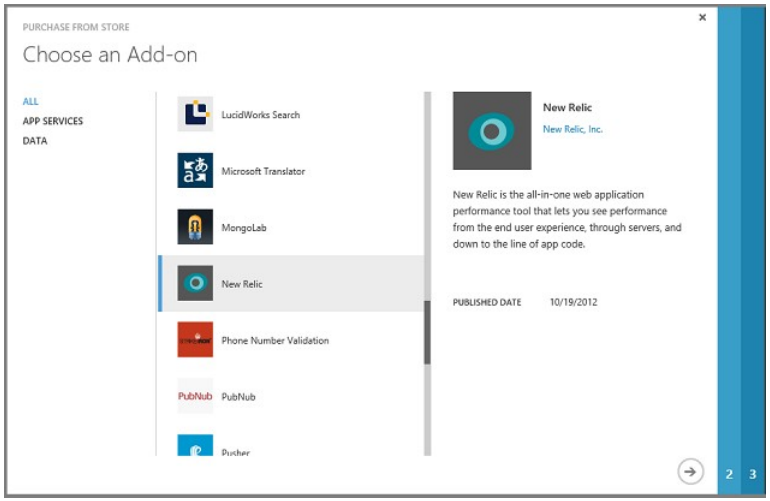
One of the things that's great about the cloud environment is that it's really easy to buy or rent your way to victory. Telemetry is an example. Without a lot of effort, you can get a really good telemetry system up and running, very cost-effectively. There are a bunch of great Microsoft partners that integrate with Azure, and some of them have free tiers—so you can get basic telemetry for nothing. Here are just a few of the ones currently available on Azure:

- New Relic
- AppDynamics
- MetricsHub
- Dynatrace

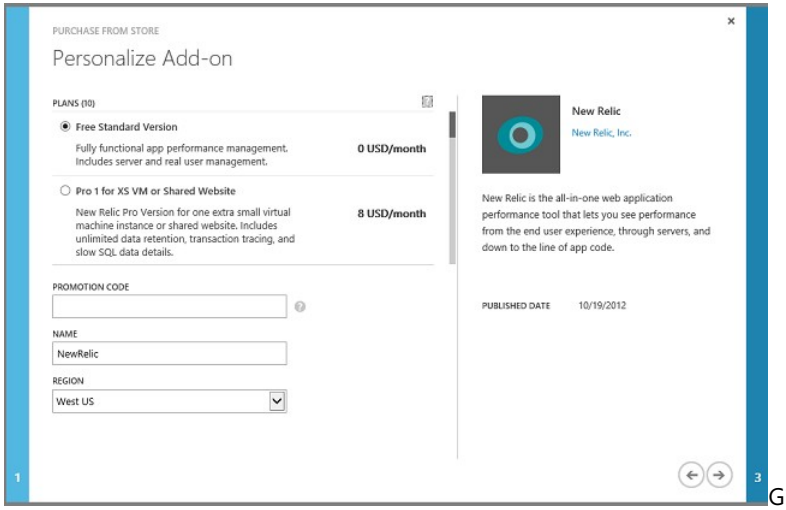
As June 2014, Microsoft Application Insights for Visual Studio Online is not released but is available in preview. Microsoft System Center also includes monitoring features.

In this section, we'll quickly walk through setting up New Relic to show how easy it can be to use a telemetry system.

In the Azure management portal, sign up for the service. Click New, **Store**. The **Choose an Add-on** dialog box appears. Scroll down and click **New Relic**.

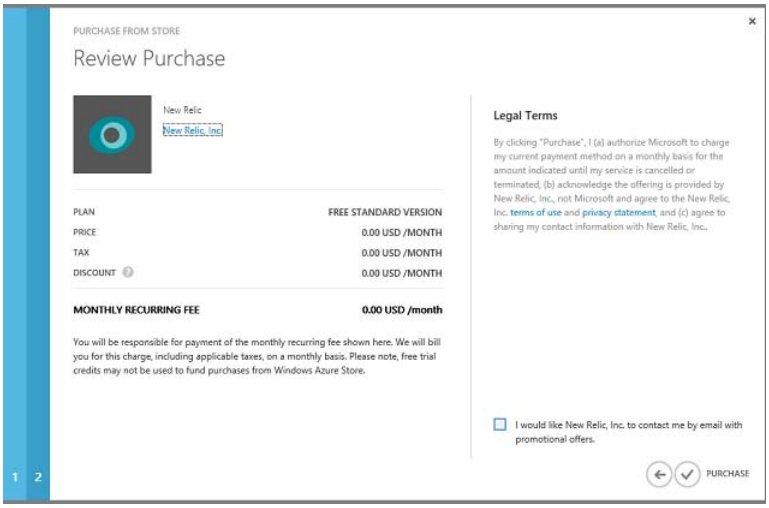


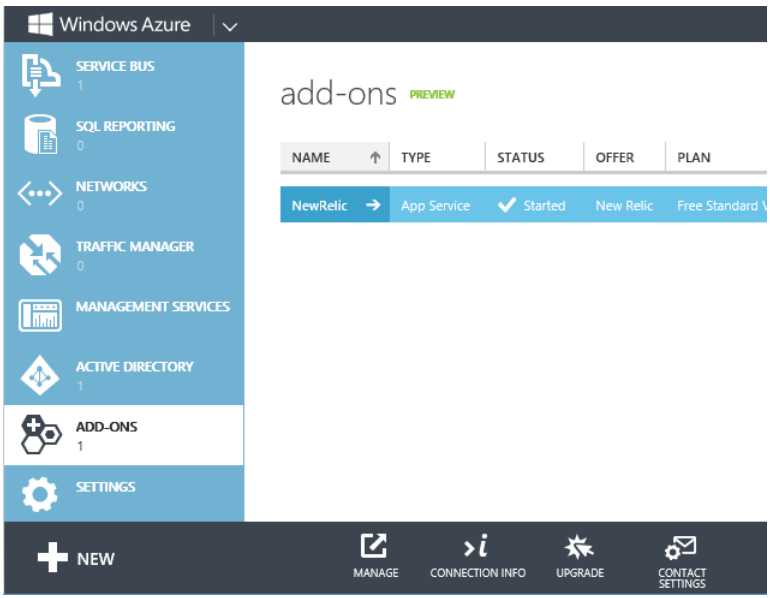
Click the right arrow and choose the service tier you want. For this demo we'll use the free tier.



G10xx02

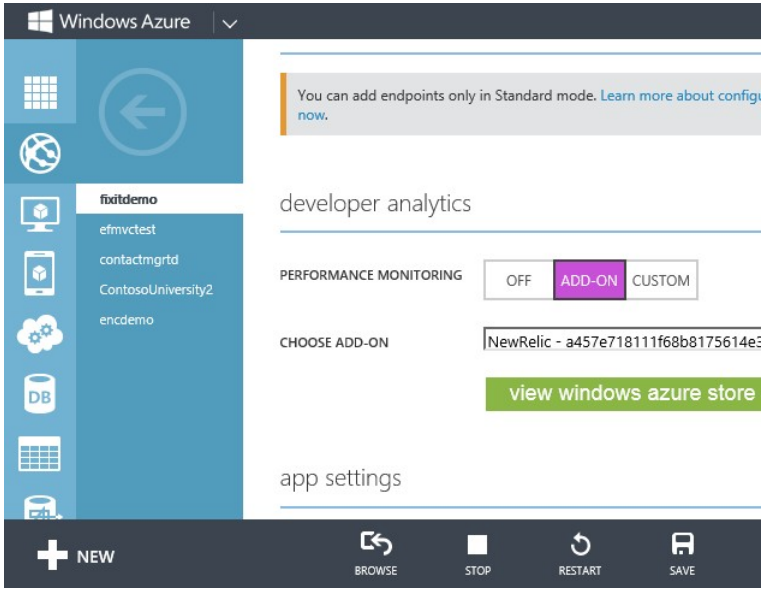
Click the right arrow, confirm the "purchase," and New Relic now shows up as an add-on in the portal.



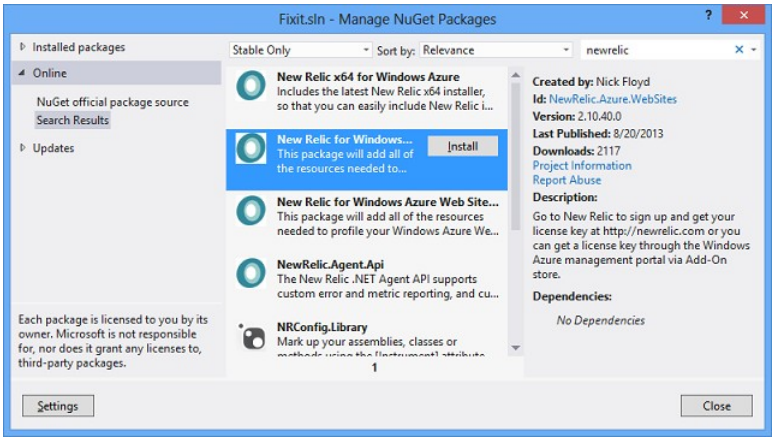


Click **Connection Info**, and copy the license key.

Go to the **Configure** tab for your website in the portal, set **Performance Monitoring** to **Add-On**, and set the **Choose Add-On** drop-down list to **New Relic**. Then click **Save**.

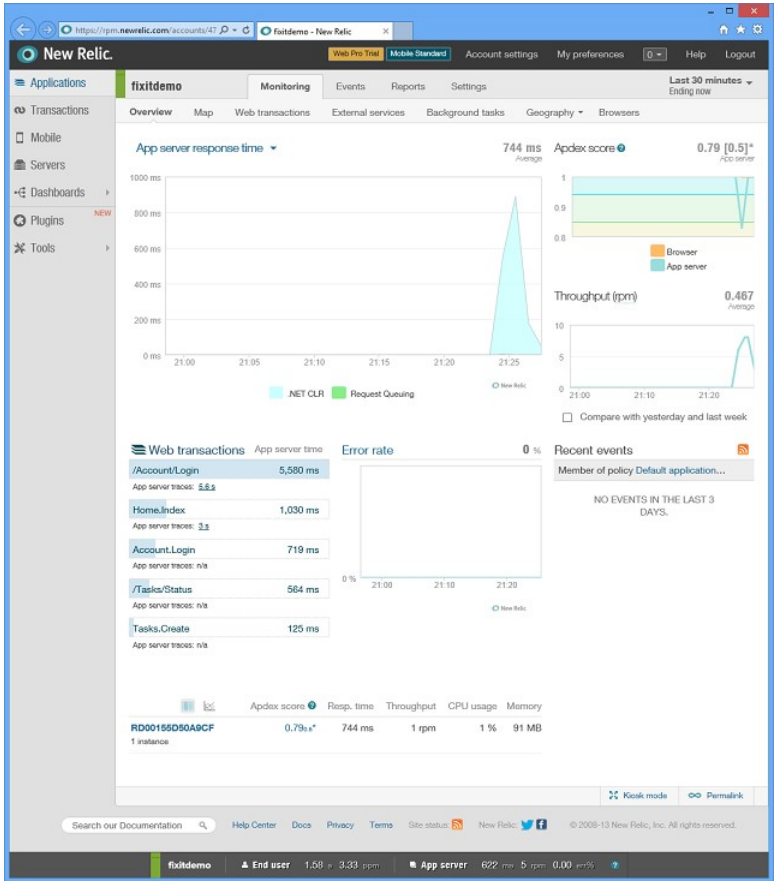


In Visual Studio, install the New Relic NuGet package in your app.



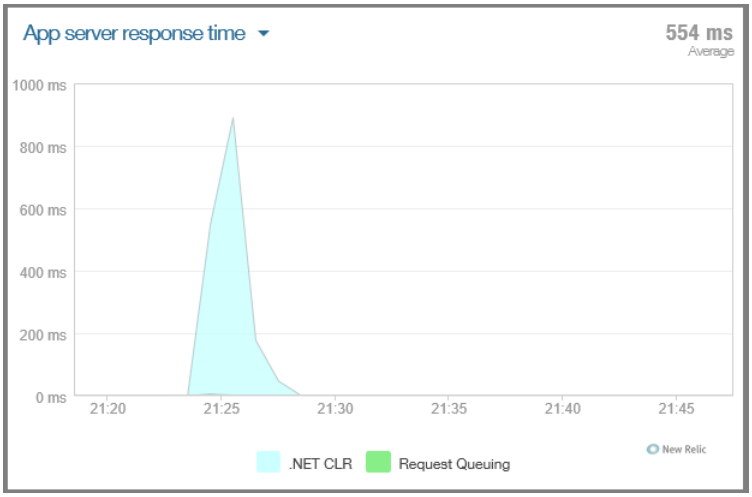
Deploy the app to Azure and start using it. Create a few Fix It tasks to provide some activity for New Relic to monitor.

Then go back to the **New Relic** page on the **Add-ons** tab of the portal and click **Manage**. The portal sends you to the New Relic management portal, using single sign-on for authentication, so you don't have to enter your credentials again. The Overview page presents a variety of performance statistics. (Click the image to see the overview page full size.)

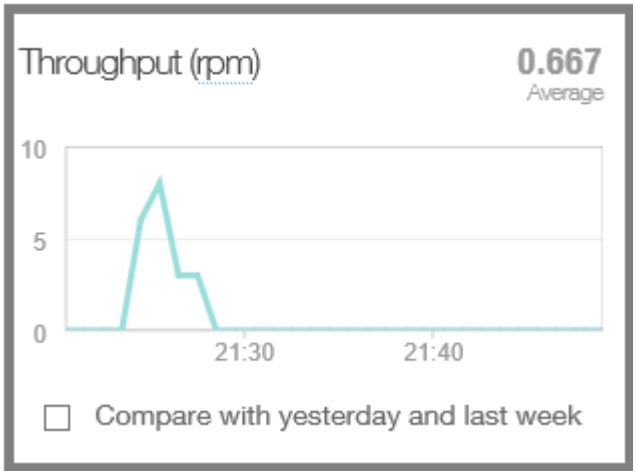


Here are just a few of the statistics you can see:

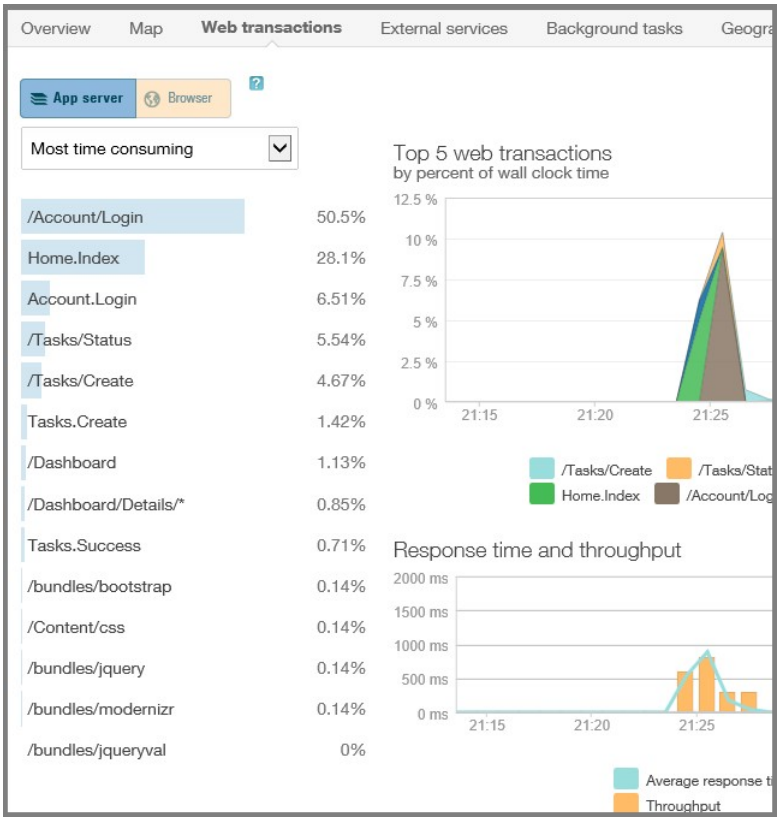
- Average response time at different times of day.



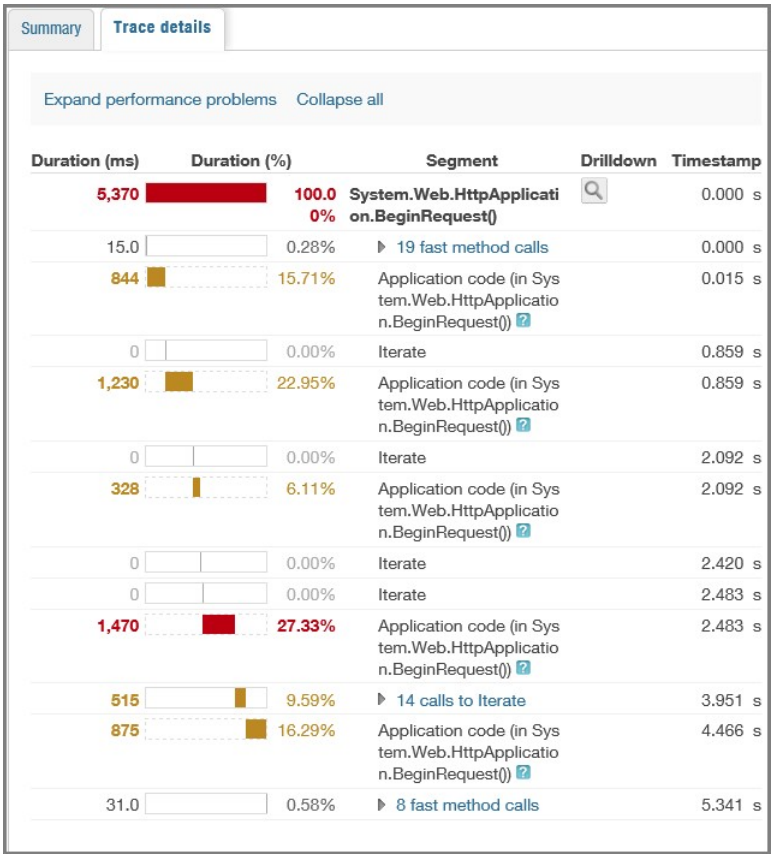
- Throughput rates (in requests per minute) at different times of day.



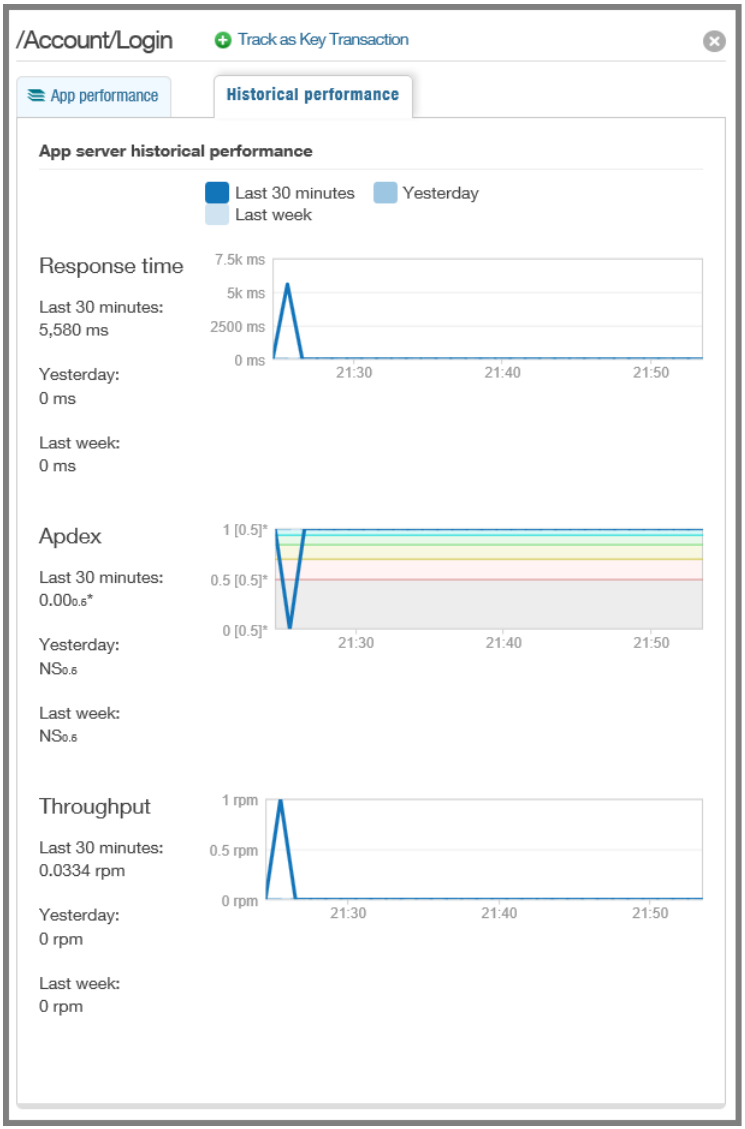
- Server CPU time spent handling different HTTP requests.



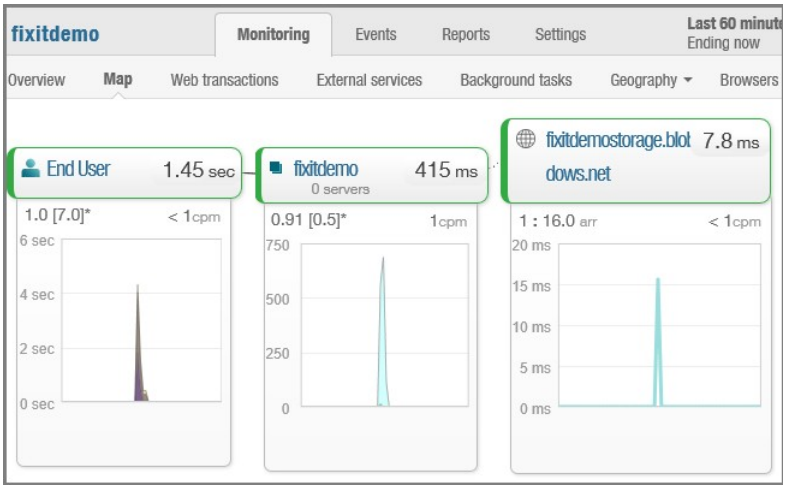
- CPU time spent in different parts of the application code.



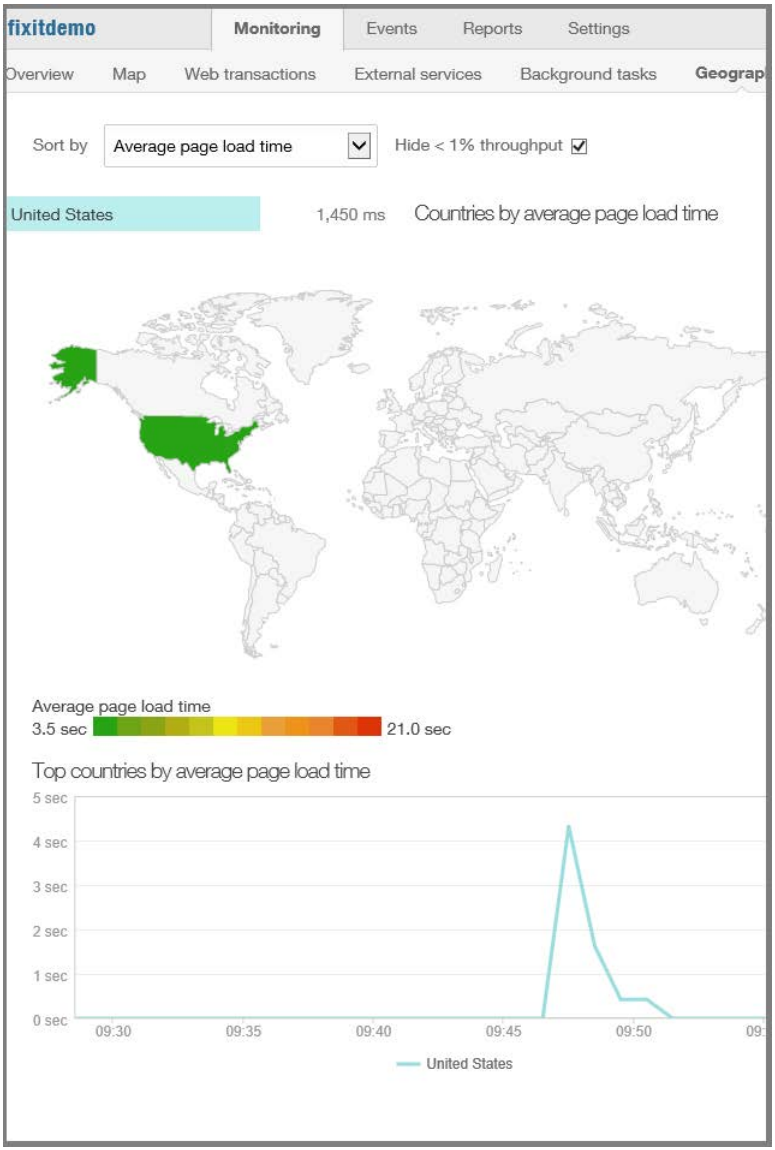
- Historical performance statistics.



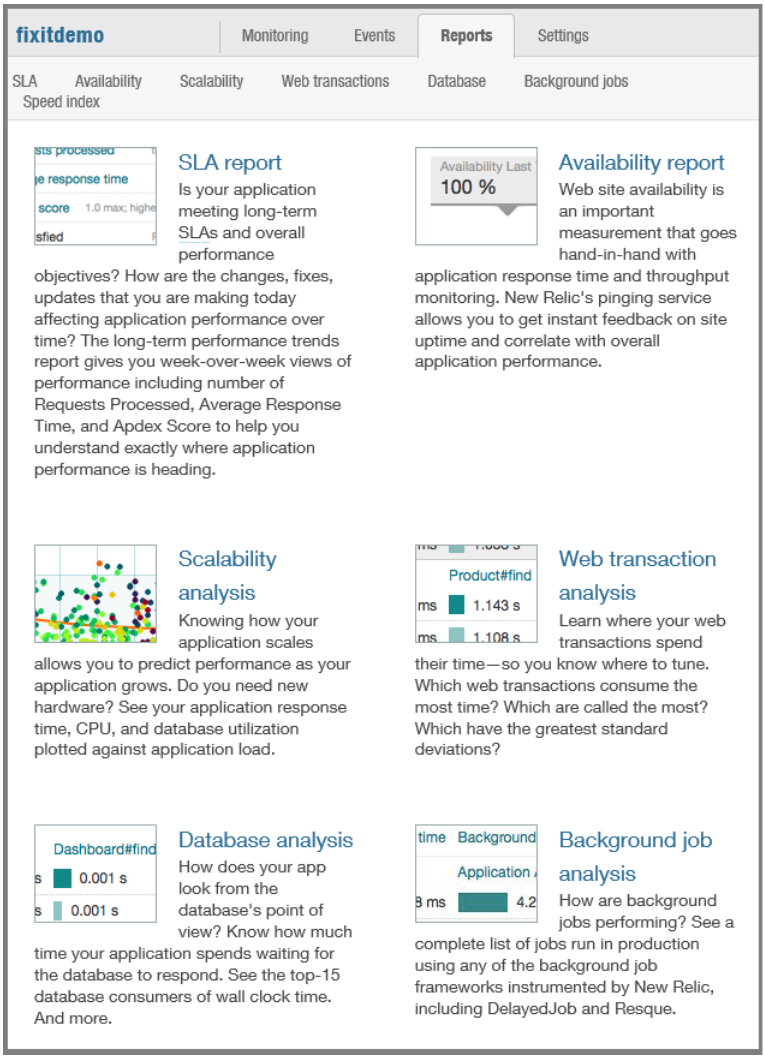
- Calls to external services such as the Blob service and statistics about how reliable and responsive the service has been.



- Information about where in the world or where in the United States website traffic came from.



You can also set up reports and events. For example, you can say that any time you start seeing errors, send an email to alert support staff to the problem.



New Relic is just one example of a telemetry system; you can get all of this functionality from other services as well. The beauty of the cloud is that without having to write any code, and for minimal or no expense, you suddenly can get a lot more information about how your application is being used and what your customers are actually experiencing.

Log for insight

A telemetry package is a good first step, but you still have to instrument your own code. The telemetry service tells you when there's a problem and tells you what customers are experiencing, but it may not

give you a lot of insight into what's going on in your code.

You don't want to have to remote into a production server to see what your app is doing. That might be practical when you have one server, but what about when you've scaled to hundreds of servers and you don't know which ones you need to remote into? Your logging should provide enough information that you never have to remote into production servers to analyze and debug problems. You should be logging enough information so that you can isolate issues solely through the logs.

Log in production

A lot of people turn on tracing in production only when there's a problem and they want to debug. This approach can introduce a substantial delay between the time you become aware of a problem and the time you obtain useful troubleshooting information about it. And the information you get might not be helpful for intermittent errors.

What we recommend for the cloud environment, where storage is cheap, is that you always leave logging on in production. That way, when errors happen, you already have them logged and have historical data that can help you analyze issues that develop over time or happen regularly at different times. You could automate a purge process to delete old logs, but you might find that it's more expensive to set up such a process than it is to keep the logs.

The added expense of logging is trivial compared with the amount of troubleshooting time and money you can save by having all the information you need already available when something goes wrong. Then, when someone tells you they had a random error sometime around 8:00 last night, but they don't remember the error, you can readily find out what the problem was.

For less than $4 a month, you can keep 50 gigabytes of logs on hand, and the performance impact of logging is trivial so long as you keep one thing in mind— be sure your logging library is asynchronous.

Differentiate logs that inform from logs that require action

Logs are meant to INFORM (I want you to know something) or ACT (I want you to do something). Be careful to write ACT logs only for issues that genuinely require a person or an automated process to take action. Too many ACT logs will create noise, requiring too much work to sift through all the log records to find genuine issues. And if your ACT logs trigger some action, such as sending email to support staff, avoid having a single issue trigger thousands of such actions.

In `.NET System.Diagnostics` tracing, logs can be assigned to the Error, Warning, Info, or Debug/Verbose level. You can differentiate ACT from INFORM logs by reserving the Error level for ACT logs and using the lower levels for INFORM logs.

Level	Context
Error	Always on in production. Any errors will trigger ACTION to resolve (automated or human). • Configuration issues • Application failure (cascading failure or critical service down)
Warning	Always on in production. Warnings will INFORM, and may signal potential ACTION • Timeouts or throttling in external service
Info	Always on in production. Info messages INFORM during diagnostics and troubleshooting
Debug (Verbose)	On during active debugging and troubleshooting on a case by case basis

Configure logging levels at run time

While it's worthwhile to always have logging on in production, another best practice is to implement a logging framework that enables you to adjust at run time the level of detail that you're logging, without redeploying or restarting your application. For example, when you use the tracing facility in `System.Diagnostics`, you can create Error, Warning, Info, and Debug/Verbose logs. We recommend that you always log Error, Warning, and Info logs in production and be able to dynamically add Debug/Verbose logging for troubleshooting on a case-by-case basis.

The Azure Websites service has built-in support for writing `System.Diagnostics` logs to the file system, Table storage, or Blob storage. You can select different logging levels for each storage destination, and you can change the logging level on the fly without restarting your application. Logging support in Blob storage makes it easier to run HDInsight analysis jobs on your application logs because HDInsight knows how to work with Blob storage directly.

Log exceptions

Don't just put `exception.ToString()` in your logging code. That leaves out inner exceptions and contextual information. In the case of SQL errors, it leaves out the SQL error number. For all exceptions, include context information, the exception itself, and inner exceptions to be sure that you provide everything that's needed for troubleshooting. For example, context information might include the server name, a transaction identifier, and a user name (but not the password or any secrets!).

Not every developer will do the right thing with exception logging if you rely on them to do so individually. To ensure that logging is done the right way every time, build exception handling into your logger interface: pass the exception object itself to the logger class and log the exception data properly in the logger class.

Log calls to services

We highly recommend that you write a log every time your app calls out to a service, whether to a database, a REST API, or any external service. Include in your logs not only an indication of success or failure but how long each request took. In the cloud environment you'll often see problems related to slowdowns rather than complete outages. Something that normally takes 10 milliseconds might suddenly start taking a second. When someone tells you your app is slow, you want to be able to look at New Relic or whichever telemetry service you have and validate the user's experience, and then you want to be able to look at your own logs to dive into the details of why your app is slow.

Use an ILogger interface

What Microsoft recommends doing when you create a production application is to create a simple `ILogger` interface and stick some methods in it. This makes changing the logging implementation later much easier, and you don't have to go through all your code to do it. We could use the `System.Diagnostics.Trace` class throughout the Fix It app, but instead we're using it under the covers in a logging class that implements `ILogger`, and we make `ILogger` method calls throughout the app.

With an approach such as this, if you ever want to make your logging richer, you can replace System.Diagnostics.Trace with whatever logging mechanism you want. For example, as your app grows, you might decide that you want to use a more comprehensive logging package, such as NLog or Enterprise Library Logging Application Block. (Log4Net is another popular logging framework, but it doesn't perform asynchronous logging.)

One reason for using a framework such as NLog is to divide logging output into separate high-volume and high-value data stores. Doing that helps you efficiently store large volumes of INFORM data that you don't need to execute fast queries against, while maintaining quick access to ACT data.

Semantic logging

For a relatively new way to do logging that can produce more useful diagnostic information, see Enterprise Library Semantic Logging Application Block (SLAB). SLAB uses Event Tracing for Windows (ETW) and EventSource support in .NET 4.5 to enable you to create more structured and queryable logs. You define a different method for each type of event that you log, which enables you to customize the information you write. For example, to log a SQL Database error you might call a `LogSQLDatabaseError` method. For that kind of exception, you know that a key piece of information is the error number, so you could include an error number parameter in the method's signature and record the error number as a separate field in the log record you write. Because the number is in a separate field, you can more easily and reliably get reports based on SQL error numbers than you could if you were just concatenating the error number into a message string.

Logging in the Fix It app

The ILogger interface

Here is the `ILogger` interface in the Fix It app.

```
public interface ILogger
{
    void Information(string message);
    void Information(string fmt, params object[] vars);
    void Information(Exception exception, string fmt, params object[] vars);

    void Warning(string message);
    void Warning(string fmt, params object[] vars);
    void Warning(Exception exception, string fmt, params object[] vars);

    void Error(string message);
    void Error(string fmt, params object[] vars);
    void Error(Exception exception, string fmt, params object[] vars);

    void TraceApi(string componentName, string method, TimeSpan timespan);
    void TraceApi(string componentName, string method, TimeSpan timespan, string properties);
    void TraceApi(string componentName, string method, TimeSpan timespan,
        string fmt, params object[] vars);
}
```

These methods enable you to write logs at the same four levels supported by `System.Diagnostics`. The `TraceApi` methods are for logging external service calls with information about latency. You could also add a set of methods for the Debug/Verbose level.

The Logger implementation of the ILogger interface

The implementation of the interface is really simple. It basically just calls into the standard `System.Diagnostics` methods. The following snippet shows all three of the Information methods and one each of the others.

```
public class Logger : ILogger
{
    public void Information(string message)
    {
        Trace.TraceInformation(message);
    }

    public void Information(string fmt, params object[] vars)
    {
        Trace.TraceInformation(fmt, vars);
    }

    public void Information(Exception exception, string fmt, params object[] vars)
```

```
    {
        var msg = String.Format(fmt, vars);
        Trace.TraceInformation(string.Format(fmt, vars) + ";Exception Details={0}",
            exception.ToString());
    }

    public void Warning(string message)
    {
        Trace.TraceWarning(message);
    }

    public void Error(string message)
    {
        Trace.TraceError(message);
    }

    public void TraceApi(string componentName, string method, TimeSpan timespan, string
properties)
    {
        string message = String.Concat("component:", componentName, ";method:", method,
                ";timespan:", timespan.ToString(), ";properties:", properties);
        Trace.TraceInformation(message);
    }
}
```

Calling the ILogger methods

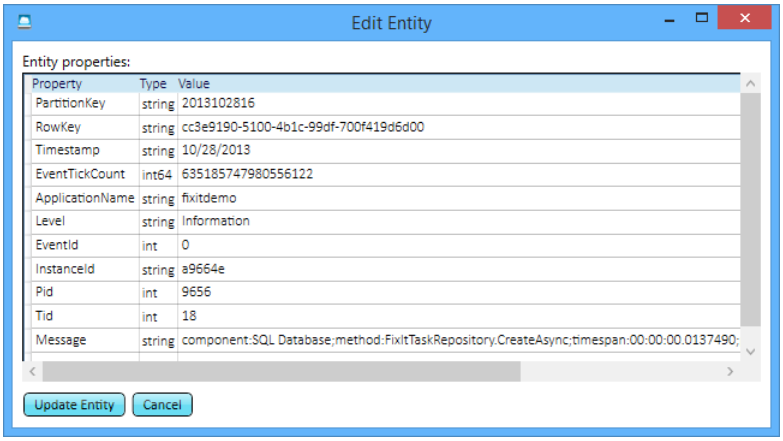
Every time code in the Fix It app catches an exception, it calls an `ILogger` method to log the exception's details. And every time Fix It makes a call to the database, the Blob service, or a REST API, it starts a stopwatch before the call, stops the stopwatch when the service returns, and logs the elapsed time along with information about success or failure.

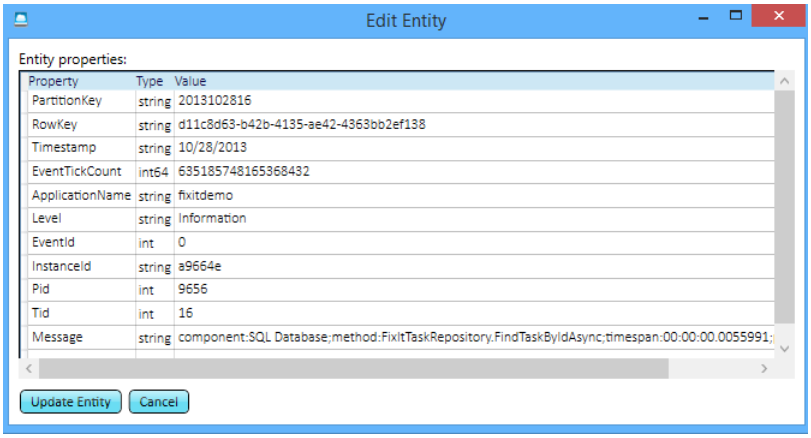
Notice that the log message includes the class name and method name. It's a good practice to be sure that log messages identify which part of the application code wrote them.

```
public class FixItTaskRepository : IFixItTaskRepository
{
    private MyFixItContext db = new MyFixItContext();
    private ILogger log = null;
    public FixItTaskRepository(ILogger logger)
    {
        log = logger;
    }

    public async Task<FixItTask> FindTaskByIdAsync(int id)
    {
        FixItTask fixItTask = null;
        Stopwatch timespan = Stopwatch.StartNew();

        try
        {
```

```
            fixItTask = await db.FixItTasks.FindAsync(id);

            timespan.Stop();
            log.TraceApi("SQL Database",
                "FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed, "id={0}", id);
        }
        catch(Exception e)
        {
            log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
        }

        return fixItTask;
    }
```

So now, for every time the Fix It app has made a call to SQL Database, you can see the call, the method that called it, and exactly how much time it took.

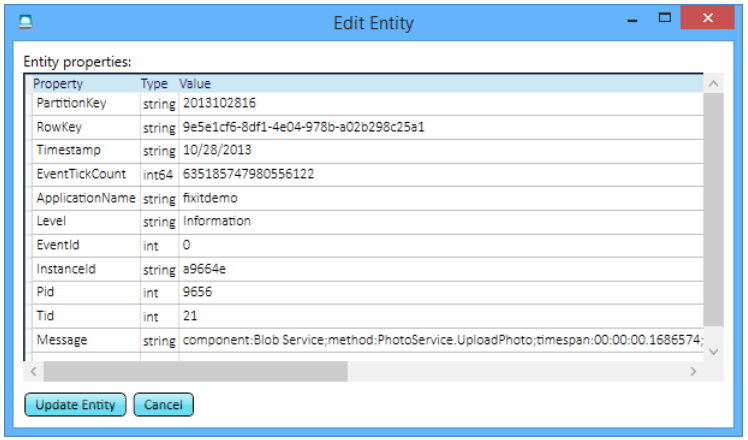




If you go browsing through the logs, you can see that the time that database calls take is variable.

That information could be useful: because the app logs all this information, you can analyze historical trends in how the database service is performing over time. For instance, a service might be fast most of the time, but requests might fail or responses might slow down at certain times of day.

You can take the same approach for the Blob service—for every time the app uploads a new file, there's a log, and you can see exactly how long each file took to upload.



It's only a couple of extra lines of code to write every time you call a service, and now whenever a user says he or she ran into an issue, you know exactly what the issue was, whether it was an error, or whether the app was just running slow. You can pinpoint the source of the problem without having to remote into a server or turn on logging after the error happens and hope to re-create it.

Dependency injection in the Fix It app

You might wonder how the repository constructor in the example shown earlier gets the logger interface implementation:

```
public class FixItTaskRepository : IFixItTaskRepository
{
    private MyFixItContext db = new MyFixItContext();
    private ILogger log = null;

    public FixItTaskRepository(ILogger logger)
    {
        log = logger;
    }
```

To wire up the interface to the implementation, the app uses dependency injection (DI) with AutoFac. DI enables you to use an object based on an interface in many places throughout your code, but you have to specify only in one place the implementation that is used when the interface is

instantiated. This makes it easier to change the implementation: for example, you might want to replace the `System.Diagnostics` logger with an NLog logger. Or, for automated testing, you might want to substitute a mock version of the logger.

The Fix It application uses DI in all of the repositories and all of the controllers. The constructors of the controller classes get an `ITaskRepository` interface, the same way the repository gets a logger interface:

```
public class DashboardController : Controller
{
    private IFixItTaskRepository fixItRepository = null;

    public DashboardController(IFixItTaskRepository repository)
    {
        fixItRepository = repository;
    }
```

The app uses the AutoFac DI library to automatically provide `TaskRepository` and `Logger` instances for these constructors.

```
public class DependenciesConfig
{
    public static void RegisterDependencies()
    {
        var builder = new ContainerBuilder();

        builder.RegisterControllers(typeof(MvcApplication).Assembly);
        builder.RegisterType<Logger>().As<ILogger>().SingleInstance();

        builder.RegisterType<FixItTaskRepository>().As<IFixItTaskRepository>();

        builder.RegisterType<PhotoService>().As<IPhotoService>().SingleInstance();
        builder.RegisterType<FixItQueueManager>().As<IFixItQueueManager>();

        var container = builder.Build();
        DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
    }
}
```

This code basically says that anywhere a constructor needs an `ILogger` interface, pass in an instance of the `Logger` class, and whenever the constructor needs an `IFixItTaskRepository` interface, pass in an instance of the `FixItTaskRepository` class.

Built-in logging support in Azure

Azure supports the following kinds of logging in the Websites service:

- System.Diagnostics tracing (you can turn on and off and set levels on the fly without restarting

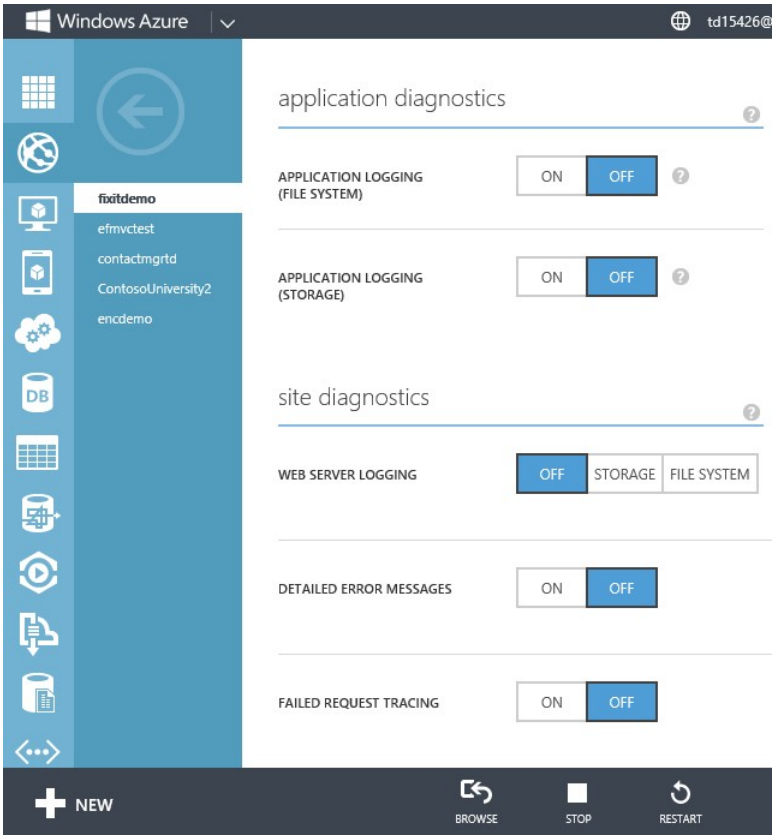
the site)

- Windows events
- IIS logs (HTTP/FREB)

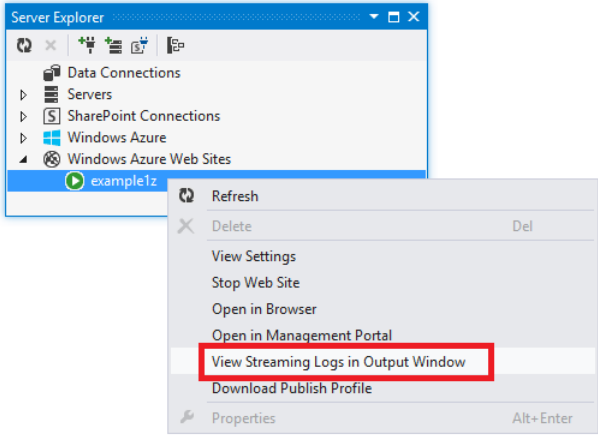
Azure supports the following kinds of logging in Cloud Services:

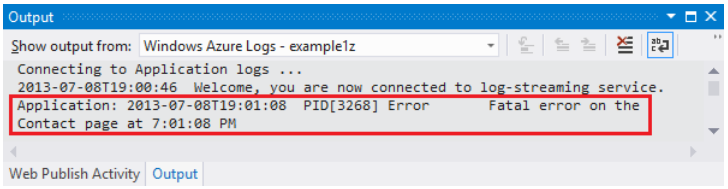
- System.Diagnostics tracing
- Performance counters
- Windows events
- IIS logs (HTTP/FREB)
- Custom directory monitoring

The Fix It app uses System.Diagnostics tracing. All you need to do to enable System.Diagnostics logging in an Azure website is flip a switch in the portal or call the REST API. In the portal, click the Configuration tab for your site and scroll down to see the Application Diagnostics section. You can turn logging on or off and select the logging level you want. You can have Azure write the logs to the file system or to a storage account.

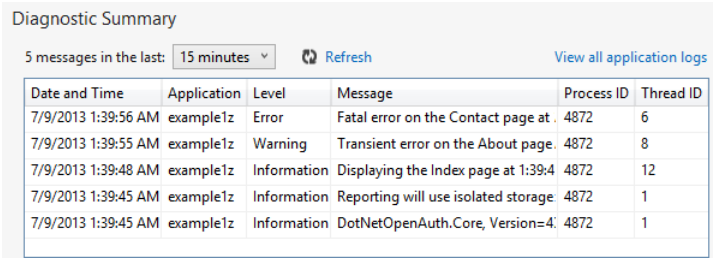


After you enable logging in Azure, you can see logs in the Visual Studio Output window as they are created.





You can also have logs written to your storage account and view them with any tool that can access the Azure Storage Table service, such as Server Explorer in Visual Studio or Azure Storage Explorer.



Summary

It's really simple to implement an out-of-the-box telemetry system, instrument logging in your own code, and configure logging in Azure. And when you have production issues, the combination of a telemetry system and custom logs will help you resolve problems quickly, before they become major issues for your customers.

In the next chapter we'll look at how to handle transient errors so that they don't become production issues that you have to investigate.

Resources

For more information, see the following resources.

Documentation mainly about telemetry:

- Microsoft Patterns & Practices—Azure Guidance See Instrumentation and Telemetry guidance, Service Metering guidance, Health Endpoint Monitoring pattern, and Runtime Reconfiguration pattern.
- Penny Pinching in the Cloud: Enabling New Relic Performance Monitoring on Windows Azure Websites.
- Best Practices for the Design of Large-Scale Services on Azure Cloud Services White paper by

Mark Simms and Michael Thomassy. See the Telemetry and Diagnostics section.

- Next-Generation Development with Application Insights *MSDN Magazine* article.

Documentation mainly about logging:

- Semantic Logging Application Block (SLAB) Neil Mackenzie presents the case for semantic logging with SLAB.
- Creating Structured and Meaningful Logs with Semantic Logging (Video) Julian Dominguez presents the case for semantic logging with SLAB.
- EF6 SQL Logging–Part 1: Simple Logging Arthur Vickers shows how to log queries executed by Entity Framework in EF 6.
- Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application Fourth in a nine-part tutorial series, shows how to use the EF 6 command interception feature to log SQL commands sent to the database by Entity Framework.

Documentation mainly about troubleshooting:

- Azure Troubleshooting & Debugging blog.
- AzureTools—The Diagnostic Utility used by the Windows Azure Developer Support Team Introduces and provides a download link for a tool that can be used on an Azure VM to download and run a wide variety of diagnostic and monitoring tools. Useful when you need to diagnose an issue on a particular VM.
- Troubleshooting Azure Web Sites in Visual Studio A step-by-step tutorial for getting started with System.Diagnostics tracing and remote debugging.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from the Microsoft Customer Advisory Team's experience with actual customers. Episodes 4 and 9 are about monitoring and telemetry. Episode 9 includes an overview of monitoring services MetricsHub, AppDynamics, New Relic, and PagerDuty.
- Building Big: Lessons learned from Windows Azure customers—Part II Mark Simms talks about designing for failure and instrumenting everything. Similar to the Failsafe series but goes into more how-to details.

Code sample:

- Cloud Service Fundamentals in Windows Azure Sample application created by the Azure Customer Advisory Team. Demonstrates both telemetry and logging practices, as explained in

the following articles. The sample implements application logging by using NLog. For related documentation, see the series of four TechNet wiki articles about telemetry and logging.

Chapter 11

Transient fault handling

When you design a real-world cloud app, you need to think about how to handle temporary service interruptions. This issue is uniquely important in cloud apps because the apps are so dependent on network connections and external services. You will frequently run into little glitches that are usually self-healing, and if you aren't prepared to handle them intelligently, they'll result in a bad experience for your customers.

Causes of transient failures

In the cloud environment, failed and dropped database connections happen periodically. That's partly because your app goes through more load balancers than in an on-premises environment, where your web server and database server have a direct physical connection. Also, sometimes when your app is dependent on a multitenant service, you'll see calls to the service become slower or time out because another tenant of the service is hitting it heavily. In other cases your app might be the user that is hitting the service too frequently, and the service deliberately throttles your app—denies connections—to prevent your app from adversely affecting other tenants of the service.

Use smart retry/back-off logic to mitigate the effect of transient failures

Instead of throwing an exception and displaying a not-available or error page to your customer, you can recognize errors that are typically transient and automatically retry the operation that resulted in the error—in hopes that before long you'll be successful. Most of the time the operation will succeed on the second try, and you'll recover from the error without your customers ever having been aware that there was a problem.

There are several ways you can implement smart retry logic:

- The Microsoft Patterns & Practices group has a Transient Fault Handling Application Block that does everything for you if you're using ADO.NET for SQL Database access (the block doesn't work with the Entity Framework). You simply set a policy for retries—how many times to retry a query or command and how long to wait between tries—and wrap your SQL code in a using block.

```
public void HandleTransients()
```

```
{
    var connStr = "some database";
    var _policy = RetryPolicy.Create < SqlAzureTransientErrorDetectionStrategy(
        retryCount: 3,
        retryInterval: TimeSpan.FromSeconds(5));

    using (var conn = new ReliableSqlConnection(connStr, _policy))
    {
        // Do SQL stuff here.
    }
}
```

Transient fault handling (TFH) also supports Azure In-Role Cache and Service Bus.

- When you use the Entity Framework, you typically aren't working directly with SQL connections, so you can't use the Patterns & Practices package, but Entity Framework 6 builds this kind of retry logic right into the framework. In a similar way, you specify the retry strategy, and then EF uses that strategy whenever it accesses the database.

  To use this feature in the Fix It app, all we have to do is add a class that derives from `DbConfiguration` and turn on the retry logic.

  ```
  // EF follows a Code based Configuration model and will look for a class that
  // derives from DbConfiguration for executing any Connection Resiliency strategies

  public class EFConfiguration : DbConfiguration
  {
      public EFConfiguration()
      {
          AddExecutionStrategy(() => new SqlAzureExecutionStrategy());
      }
  }
  ```

  For SQL Database exceptions that the framework identifies as typically transient errors, the code shown instructs EF to retry the operation up to three times, with an exponential back-off delay between retries, and a maximum delay of 5 seconds. Exponential back-off means that after each failed retry, the app will wait for a longer period of time before trying again. If three tries in a row fail, the app will throw an exception. The next section is about circuit breakers and explains why you want to implement exponential back-off and use a limited number of retries.

  You can run into issues when you're using the Azure Storage service, and the .NET storage client API already implements the same kind of logic. You just specify the retry policy—and you don't even have to do that if you're happy with the default settings.

Circuit breakers

There are several reasons why you don't want to retry too many times over too long a period:

- Too many users persistently retrying failed requests might degrade other users' experience. If millions of people are all making repeated retry requests, you could tie up IIS dispatch queues and prevent your app from servicing requests that it otherwise could handle successfully.
- If everyone is retrying an operation because of a service failure, so many requests could be queued up that the service gets flooded when it starts to recover.
- If the error is the result of throttling and there's a window of time the service uses for throttling, continued retries could move that window out and cause the throttling to continue.
- You might have a user waiting for a webpage to render. Making people wait too long might be more annoying that relatively quickly advising them to try again later.

Exponential back-off addresses some of these issue by limiting the frequency of retries that a service can get from your application. But you also need to have circuit breakers: this means that at a certain retry threshold your app stops retrying and takes some other action, such as one of the following:

- Custom fallback. If you can't get a stock price from Reuters, maybe you can get it from Bloomberg; or if you can't get data from the database, maybe you can get it from cache.
- Fail silently. If what you need from a service isn't all-or-nothing for your app, just return null when you can't get the data. For example, if you're displaying a Fix It task and the Blob service isn't responding, you could display the task details without the image.
- Fail fast. Error out the user to avoid flooding the service with retry requests that could cause service disruption for other users or extend a throttling window. You can display a friendly "try again later" message.

There is no one-size-fits-all retry policy. You can retry more times and wait longer in an asynchronous background worker process than you would in a synchronous web app where a user is waiting for a response. You can wait longer between retries for a relational database service than you would for a cache service. Here are some sample recommended retry policies to give you an idea of how the numbers might vary. ("Fast First" means no delay before the first retry.)

Platform	Context	Sample Target e2e latency max	Fast First	Retry Count	Delay	Backoff
SQL Database	Synchronous (e.g. render web page)	200 ms	Yes	3	50 ms	Linear
	Asynchronous (e.g. process queue item)	60 seconds	No	4	5 s	Exponential
Azure Table Storage	Synchronous (e.g. render web page)	100 ms	Yes	3	25 ms	Linear
	Asynchronous (e.g. process queue item)	500 ms	Yes	3	100 ms	Exponential

Summary

A retry/back-off strategy can help make temporary errors invisible to the customer most of the time, and Microsoft provides frameworks that you can use to minimize your work implementing a strategy, whether you're using ADO.NET, the Entity Framework, or the Azure Storage service.

In the next chapter, we'll look at how to improve performance and reliability by using distributed caching.

Resources

For more information, see the following resources:

Documentation:

- Best Practices for the Design of Large-Scale Services on Azure Cloud Services White paper by Mark Simms and Michael Thomassy. Similar to the Failsafe series but goes into more how-to details. See the Telemetry and Diagnostics section.
- Failsafe: Guidance for Resilient Cloud Architectures White paper by Marc Mercuri, Ulrich Homann, and Andrew Townhill. Webpage version of the FailSafe video series.
- Microsoft Patterns & Practices—Azure Guidance See Retry pattern, Scheduler Agent Supervisor pattern.
- Fault-tolerance in Windows Azure SQL Database Blog post by Tony Petrossian.
- Entity Framework—Connection Resiliency / Retry Logic How to use and customize the transient fault handling feature of Entity Framework 6.

- Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application Fourth in a nine-part tutorial series, shows how to set up the EF 6 connection resilience feature for SQL Database.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from the Microsoft Customer Advisory Team's experience with actual customers. See the discussion of circuit breakers in episode 3 starting at 40:55.
- Building Big: Lessons learned from Windows Azure customers—Part II Mark Simms talks about designing for failure, transient fault handling, and instrumenting everything.

Code sample:

- Cloud Service Fundamentals in Windows Azure Sample application created by the Azure Customer Advisory Team that demonstrates how to use the Enterprise Library Transient Fault Handling Block (TFH). For more information, see Cloud Service Fundamentals
- Data Access Layer –Transient Fault Handling TFH is recommended for database access using ADO.NET directly (without using Entity Framework).

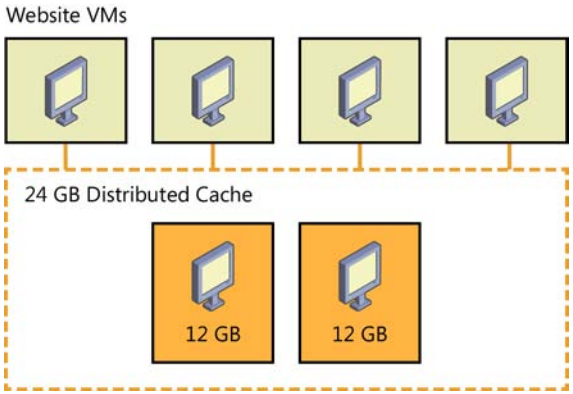
Chapter 12

Distributed caching

The previous chapter looked at transient fault handling and mentioned caching as a circuit breaker strategy. This chapter gives more background about caching, including when to use it, common patterns for using it, and how to implement it in Azure.

What is distributed caching?

A cache provides high throughput, low-latency access to commonly accessed application data by storing the data in memory. For a cloud app, the most useful type of cache is a distributed cache, which means that the data is not stored in the individual web server's memory but on other cloud resources, and the cached data is made available to all of an application's web servers (or other cloud VMs that are used by the application).



When the application scales by adding or removing servers, or when servers are replaced because of upgrades or faults, the cached data remains accessible to every server that runs the application.

By avoiding the high-latency data access of a persistent data store, caching can dramatically improve application responsiveness. For example, retrieving data from cache is much faster than retrieving it from a relational database.

A side benefit of caching is reduced traffic to the persistent data store, which may result in lower costs when there are data egress charges for the persistent data store.

When to use distributed caching

Caching works best for application workloads that do more reading than writing of data and when the data model supports the key/value organization that you use to store and retrieve data in cache. Caching is also more useful when application users share a lot of common data; for example, cache would not provide as many benefits if each user typically retrieves data unique to that user. An example where caching could be very beneficial is a product catalog, because the data does not change frequently and all customers are looking at the same data.

The benefit of caching becomes increasingly measurable the more an application scales, because the throughput limits and latency delays of the persistent data store become more of a limit on overall application performance. However, you might implement caching for reasons other than performance as well. For data that doesn't have to be perfectly up to date when shown to a user, cache access can serve as a circuit breaker for when the persistent data store is unresponsive or unavailable.

Popular cache population strategies

To be able to retrieve data from cache, you have to store it there first. There are several strategies for getting the data you need in a cache into the cache:

- **On demand/cache aside** The application tries to retrieve data from cache, and when the cache doesn't have the data (a “miss”), the application stores the data in the cache so that it will be available the next time. The next time the application tries to get the same data, it finds what it's looking for in the cache (a “hit”). To prevent fetching cached data that has changed in the database, you invalidate the cache when making changes to the data store.
- **Background data push** Background services push data into the cache on a regular schedule, and the app always pulls from the cache. This approach works great with high-latency data sources that don't require that you always return the latest data.
- **Circuit breaker** The application normally communicates directly with the persistent data store, but when the persistent data store has availability problems, the application retrieves data from cache. Data may have been put in cache using either the cache aside or background data push strategy. This is a fault-handling strategy rather than a performance-enhancing strategy.

To keep data in the cache current, you can delete related cache entries when your application creates, updates, or deletes data. If it's all right for your application to sometimes get data that is slightly out of date, you can rely on a configurable expiration time to set a limit on how old cache data can be.

You can configure absolute expiration (the amount of time since the cache item was created) or sliding expiration (the amount of time since a cache item was last accessed). Absolute expiration is used when you depend on the cache expiration mechanism to prevent data from becoming too stale. Regardless of the expiration policy you choose, the cache will automatically evict the oldest (least recently used, or LRU) items when the cache's memory limit is reached.

Sample cache-aside code for the Fix It app

In the following sample code, we check the cache first when retrieving a Fix It task. If the task is found in cache, we return it; if the task is not found, we get it from the database and store it in the cache. The changes you would make to add caching to the `FindTaskByIdAsync` method are shown in bold.

```
public async Task<FixItTask> FindTaskByIdAsync(int id)
{
    FixItTask fixItTask = null;
    Stopwatch timespan = Stopwatch.StartNew();
    string hitMiss = "Hit";
    try
    {
        fixItTask = (FixItTask)cache.Get(id.ToString());
        if (fixItTask == null)
        {
            fixItTask = await db.FixItTasks.FindAsync(id);
            cache.Put(id.ToString(), fixItTask);
            hitMiss = "Miss";
        }

        timespan.Stop();
        log.TraceApi("SQL Database", "FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed,
            "cache {0}, id={1}", mark>hitMiss, id);
    }
    catch (Exception e)
    {
        log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
    }
    return fixItTask;
}
```

When you update or delete a Fix It task, you have to invalidate (remove) the cached task; otherwise, future attempts to read that task will continue to get the old data from the cache.

```
public async Task UpdateAsync(FixItTask taskToSave)
{
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        cache.Remove(taskToSave.FixItTaskId.ToString());
```

```
            db.Entry(taskToSave).State = EntityState.Modified;
            await db.SaveChangesAsync();

            timespan.Stop();
            log.TraceApi("SQL Database", "FixItTaskRepository.UpdateAsync",
              timespan.Elapsed, "taskToSave={0}", taskToSave);
        }
        catch (Exception e)
        {
            log.Error(e, "Error in FixItTaskRepository.UpdateAsync(taskToSave={0})", taskToSave);
        }
    }
```

These are samples to illustrate simple caching code; caching has not been implemented in the downloadable Fix It project.

Azure caching services

Azure offers several caching services: Azure Redis Cache, Azure Managed Cache, and Azure In-Role Cache. Azure Redis Cache is based on the popular open source Redis Cache and is the newest offering. It is currently in preview, but when it is released for general availability, it will be the first choice for most caching scenarios.

ASP.NET session state using a cache provider

As mentioned in Chapter 4, "Web development best practices," a best practice is to avoid using session state. If your application requires session state, the next best practice is to avoid the default in-memory provider because it doesn't enable scale out (using multiple instances of the web server). The ASP.NET SQL Server session state provider enables a site that runs on multiple web servers to use session state, but it incurs a high latency cost compared with an in-memory provider. The best solution if you have to use session state is to use a cache provider such as the Session State Provider for Azure Cache.

Summary

You've seen how the Fix It app could implement caching to improve response time and scalability and to enable the app to continue to be responsive for read operations when the database is unavailable. In the next chapter we'll show how to further improve scalability and make the app continue to be responsive for write operations.

Resources

For more information about caching, see the following resources.

Documentation:

- Azure Cache Official MSDN documentation on caching in Azure.
- Microsoft Patterns & Practices—Azure Guidance See Caching guidance and Cache-Aside pattern.
- Failsafe: Guidance for Resilient Cloud Architectures White paper by Marc Mercuri, Ulrich Homann, and Andrew Townhill. See the section on caching.
- Best Practices for the Design of Large-Scale Services on Azure Cloud Services White paper by Mark Simms and Michael Thomassy. See the section on distributed caching.
- Distributed Caching on the Path to Scalability An older (2009) *MSDN Magazine* article, but a clearly written introduction to distributed caching in general; goes into more depth than the caching sections of the FailSafe and Best Practices white papers.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents a 400-level view of how to architect cloud apps. This series focuses on theory and reasons why; for more how-to details, see the Building Big series by Mark Simms. See the caching discussion in episode 3 starting at 1:24:14.
- Building Big: Lessons learned from Windows Azure customers—Part I Simon Davies discusses distributed caching starting at 46:00. Similar to the Failsafe series but goes into more how-to details. The presentation was given October 31, 2012, so it does not cover the Azure Websites caching service that was introduced in 2013.

Code sample:

- Cloud Service Fundamentals in Windows Azure Sample application that implements distributed caching. See the accompanying blog post Cloud Service Fundamentals – Caching Basics.

Chapter 13

Queue-centric work pattern

Earlier, we saw that using multiple services can result in a composite SLA, where the app's effective SLA is the product of the individual SLAs. For example, the Fix It app uses the Azure Websites, Storage, and SQL Database services. If any one of these services fails, the app returns an error to the user.

Caching is a good way to handle transient failures for read-only content. But what if your application needs to do work? For example, when the user submits a new Fix It task, the app can't just put the task into the cache. The app needs to write the Fix It task into a persistent data store so that the task can be processed.

That's where the queue-centric work pattern comes in. This pattern enables loose coupling between a web tier and a back-end service.

Here's how the pattern works. When the application receives a request, it puts a work item onto a queue and immediately returns the response. Then a separate back-end process pulls work items from the queue and does the work.

The queue-centric work pattern is useful for:

- Work that is time-consuming (has a high latency).
- Work that requires an external service that might not always be available.
- Work that is resource intensive (high CPU demand).
- Work that would benefit from rate leveling (subject to sudden load bursts).

Reduced latency

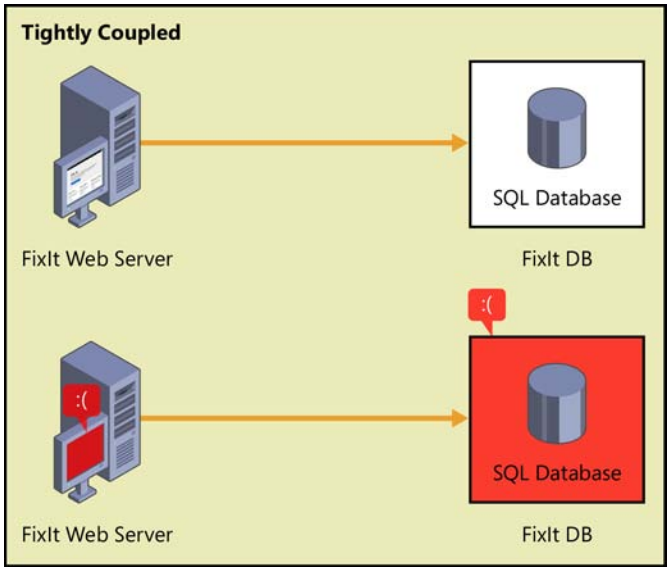
Queues are useful any time you are doing time-consuming work. If a task takes a few seconds or longer, instead of blocking the end user, put the work item into a queue. Tell the user "We're working on it," and then use a queue listener to process the task in the background.

For example, when you purchase something at an online retailer, the website confirms your order immediately. But that doesn't mean your stuff is already in a truck being delivered. The retailer puts a task in a queue, and in the background their system does the credit check, prepares your items for shipping, and so forth.

For scenarios with short latency, the total end-to-end time might be longer using a queue than doing the task synchronously. But even then, the other benefits can outweigh that disadvantage.

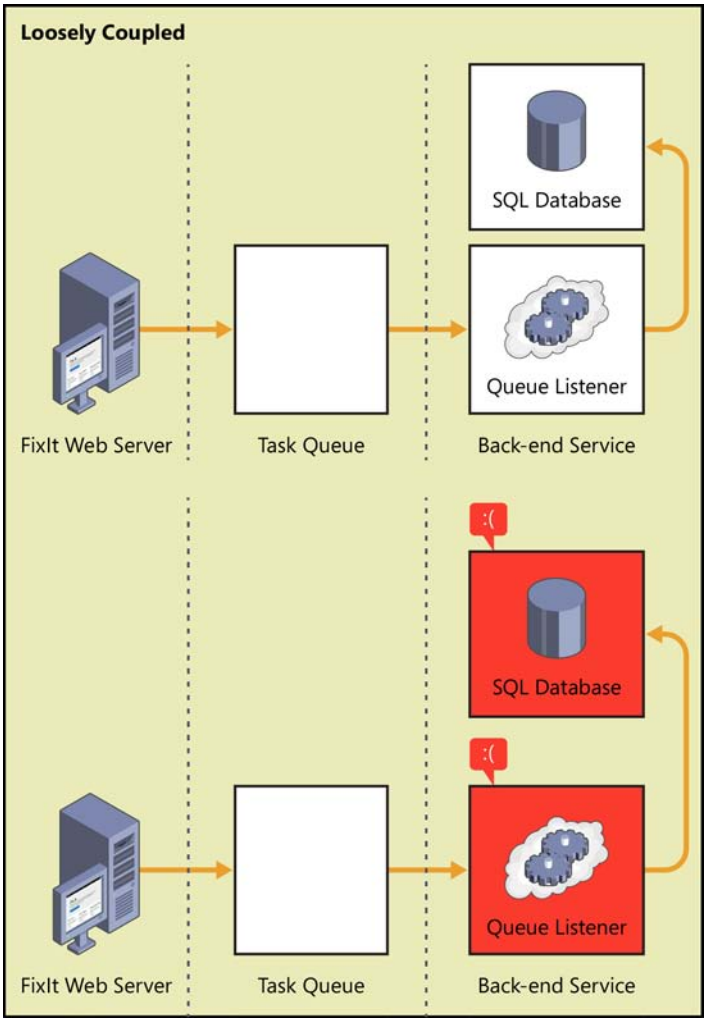
Increased reliability

In the version of Fix It that we've been looking at so far, the web front-end is tightly coupled with the back-end SQL Database. If the SQL Database service is unavailable, the user gets an error. If retries don't work (that is, the failure is more than transient), the only thing you can do is show an error and ask the user to try again later.



With queues in place, when a user submits a Fix It task, the app writes a message to the queue. The message payload is a JSON representation of the task. As soon as the message is written to the queue, the app returns and immediately shows a success message to the user.

The Fix It tasks stay in the queue until they are processed. If any of the back-end services—such as the SQL database or the queue listener—go offline, users can still submit new Fix It tasks. The messages will just queue up until the back-end services are available again. At that point, the back-end services will catch up on the backlog.



Moreover, now you can add more back-end logic without worrying about the resiliency of the front end. For example, you might want to send an email or SMS message to the owner whenever a new Fix It task is assigned. If the email or SMS service becomes unavailable, you can process everything else and then put a message into a separate queue for sending email/SMS messages.

Previously, our effective SLA was Websites × Storage × SQL Database = 99.7 percent. (See the section "Composite SLAs" in Chapter 9, "Design to survive failures.")

With a queue, the web front end depends on Websites and Storage, for a composite SLA of 99.8 percent. (Note that queues are built on Azure Storage, so they share the SLA with Blob storage.)

If you need even better than 99.8 percent, you can create two queues in two different regions. Designate one as the primary queue and the other as the secondary. In your app, fail over to the secondary queue if the primary queue is not available. The chance of both being unavailable at the same time is very small.

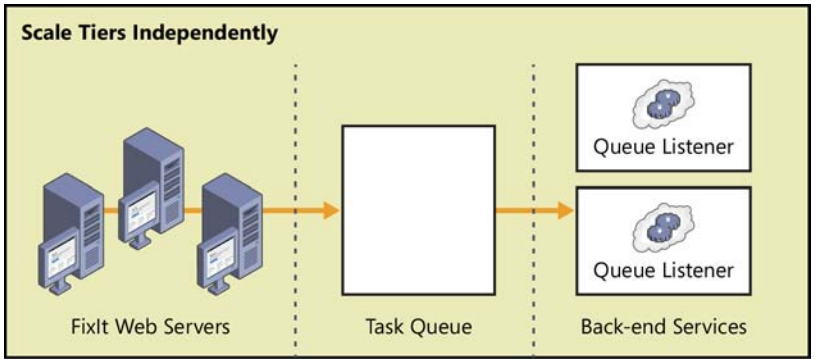
Rate leveling and independent scaling

Queues are also useful for something called rate leveling or load leveling.

Web apps are often susceptible to sudden bursts in traffic. While you can use autoscaling to automatically add web servers to handle increased web traffic, autoscaling might not be able to react quickly enough to handle abrupt spikes in load. If the web servers can offload some of the work they have to do by writing a message to a queue, they can handle more traffic. A back-end service can then read messages from the queue and process them. The depth of the queue will grow or shrink as the incoming load varies.

With much of its time-consuming work offloaded to a back-end service, the web tier can more easily respond to sudden spikes in traffic. And you save money because any given amount of traffic can be handled by fewer web servers.

You can scale the web tier and back-end service independently. For example, you might need three web servers but only one server processing queue messages. Or, if you're running a compute-intensive task in the background, you might need more back-end servers than web servers.



Autoscaling works with back-end services as well as with the web tier. You can scale up or scale down the number of VMs that are processing the tasks in the queue, based on the CPU usage of the back-end VMs. Or, you can autoscale based on how many items are in a queue. For example, you can configure autoscaling to try to keep no more than 10 items in the queue. If the queue has more than 10 items, autoscaling will add VMs. When the app catches up, autoscaling will tear down the extra VMs.

Adding queues to the Fix It application

To implement the queue pattern, we need to make two changes to the Fix It app.

- When a user submits a new Fix It task, put the task in the queue instead of writing it to the database.
- Create a back-end service that processes messages in the queue.

For the queue, we'll use the Azure Queue Storage Service, which is a persistent FIFO queue that runs on top of the Azure Storage service. Another option is to use Azure Service Bus.

To decide which queue service to use, consider how your app needs to send and receive the messages in the queue:

- If you have cooperating producers and competing consumers, consider using Azure Queue Storage Service. "Cooperating producers" means multiple processes are adding messages to a queue. "Competing consumers" means multiple processes are pulling messages off the queue to process them, but any given message is processed by only one consumer. If you need more throughput than you can get with a single queue, use additional queues and/or additional storage accounts.
- If you need a publish/subscribe model, consider using Azure Service Bus queues.

The Fix It app fits the cooperating producers and competing consumers model.

Another consideration is application availability. The Queue Storage Service is part of the same service that we're using for Blob storage, so using it has no effect on our SLA. Azure Service Bus is a separate service with its own SLA. If we were to use Service Bus queues, we would have to factor in an additional SLA percentage, and our composite SLA would be lower. When you're choosing a queue service, be sure you understand the impact of your choice on application availability. For more information, see the Resources section.

Creating queue messages

To put a Fix It task on the queue, the web front end performs the following steps:

1. Creates a CloudQueueClient instance. The `CloudQueueClient` instance is used to execute requests against the Queue service.
2. Creates the queue, if it doesn't exist yet.
3. Serializes the Fix It task.

4. Calls CloudQueue.AddMessageAsync to put the message onto the queue.

We'll do this work in the constructor and `SendMessageAsync` method of a new `FixItQueueManager` class.

```
public class FixItQueueManager : IFixItQueueManager
{
    private CloudQueueClient _queueClient;
    private IFixItTaskRepository _repository;

    private static readonly string fixitQueueName = "fixits";

    public FixItQueueManager(IFixItTaskRepository repository)
    {
        _repository = repository;
        CloudStorageAccount storageAccount = StorageUtils.StorageAccount;
        _queueClient = storageAccount.CreateCloudQueueClient();
    }

    // Puts a serialized fixit onto the queue.
    public async Task SendMessageAsync(FixItTask fixIt)
    {
        CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
        await queue.CreateIfNotExistsAsync();

        var fixitJson = JsonConvert.SerializeObject(fixIt);
        CloudQueueMessage message = new CloudQueueMessage(fixitJson);

        await queue.AddMessageAsync(message);
    }

    // Processes any messages on the queue.
    public async Task ProcessMessagesAsync(CancellationToken token)
    {
        CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
        await queue.CreateIfNotExistsAsync();
        while (!token.IsCancellationRequested)
        {
            // The default timeout is 90 seconds, so we won't continuously poll the queue if
            // there are no messages.
            // Pass in a cancellation token, because the operation can be long-running.
            CloudQueueMessage message = await queue.GetMessageAsync(token);
            if (message != null)
            {
                FixItTask fixit = JsonConvert.DeserializeObject<FixItTask>(message.AsString);
                await _repository.CreateAsync(fixit);
                await queue.DeleteMessageAsync(message);
            }
        }
    }
}
```

Here we use the Json.NET library to serialize the FixIt task to JSON format. You can use whichever

serialization approach you prefer. JSON has the advantage of being human-readable and is less verbose than XML.

Production-quality code would add error-handling logic, pause if the database became unavailable, handle recovery more cleanly, create the queue on application startup, and manage "poison" messages. (A poison message is a message that cannot be processed for some reason. You don't want poison messages to sit in the queue, where the worker role will continually try to process them, fail, try again, fail, and so on.)

In the front-end MVC application, we need to update the code that creates a new task. Instead of putting the task into the repository, call the `SendMessageAsync` method shown above.

```
public async Task<ActionResult> Create(FixItTask fixittask, HttpPostedFileBase photo)
{
    if (ModelState.IsValid)
    {
        fixittask.CreatedBy = User.Identity.Name;
        fixittask.PhotoUrl = await photoService.UploadPhotoAsync(photo);
        //previous code:
        //await fixItRepository.CreateAsync(fixittask);
        //new code:
        await queueManager.SendMessageAsync(fixittask);
        return RedirectToAction("Success");
    }
    return View(fixittask);
}
```

Processing queue messages

To process messages in the queue, we'll create a back-end service. The back-end service will run an infinite loop that performs the following steps:

1. Get the next message from the queue.
2. Deserialize the message to a Fix It task.
3. Write the Fix It task to the database.

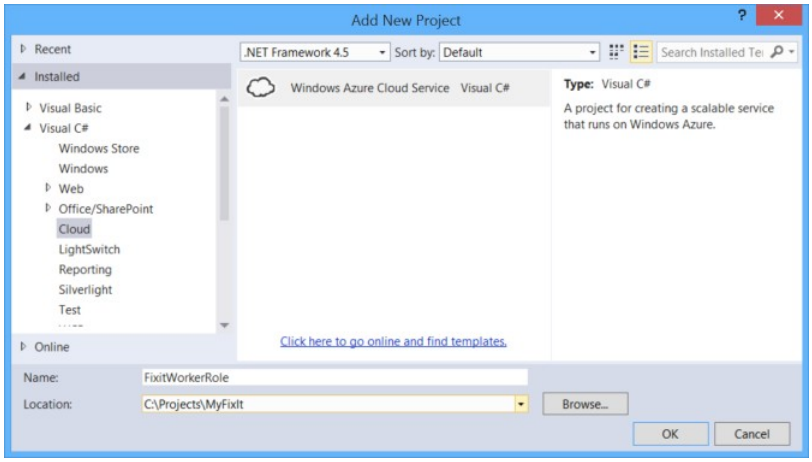
To host the back-end service, we'll create an Azure Cloud Service that contains a worker role. A worker role consists of one or more VMs that can do back-end processing. The code that runs in these VMs will pull messages from the queue as they become available. For each message, we'll deserialize the JSON payload and write an instance of the Fix It Task entity to the database, using the same repository that we used earlier in the web tier.

Note A Cloud Service is not the only option for creating the back-end. Other options include Virtual Machines and the WebJobs feature of Azure Websites. For more information about the trade-offs, see

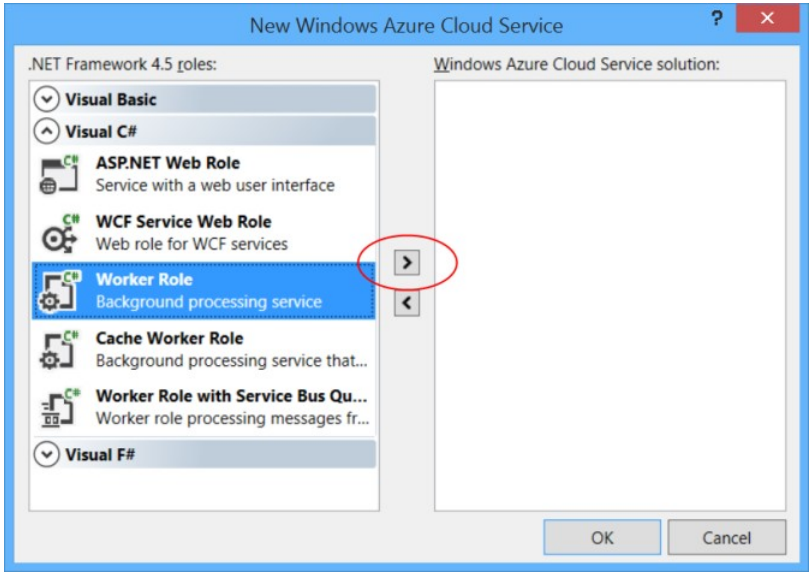
[Azure Web Sites, Cloud Services and Virtual Machines comparison](#).

The following steps show how to add a worker role project to a solution that has a standard web project. These steps have already been completed in the Fix It project that you can download.

First, add a Cloud Service project to the Visual Studio solution. Right-click the solution and select **Add**, then **New Project**. In the left pane, expand **Visual C#** and select **Cloud**.

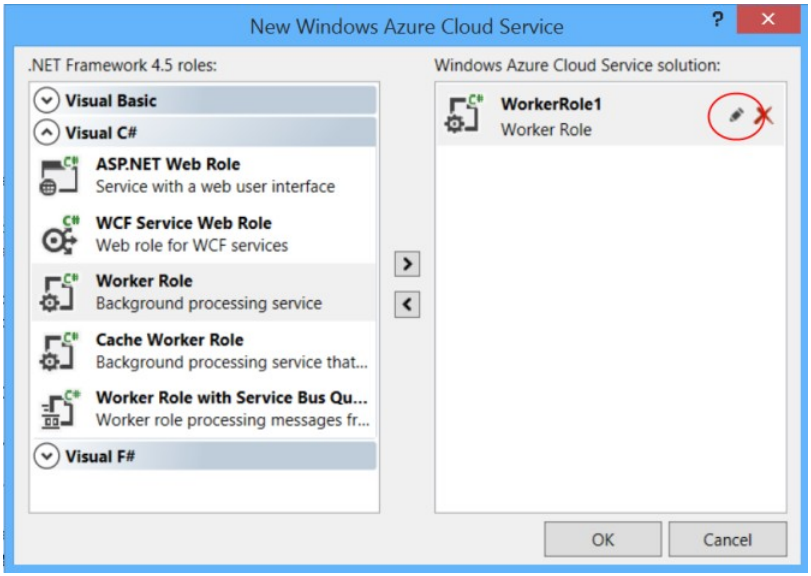


In the **New Windows Azure Cloud Service** dialog, expand the **Visual C#** node in the left pane. Select **Worker Role**, and then click the right arrow icon.



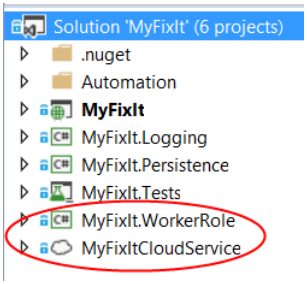
(Notice that you can also add a web role. We could run the Fix It front end in the same Cloud Service instead of running it in an Azure website. That has some advantages in making connections between the front end and back end easier to coordinate. However, to keep this demo simple, we're keeping the front end in a Website and running only the back end in a Cloud Service.)

A default name is assigned to the worker role. To change the name, point to the worker role in the right pane, and then click the pencil icon.



Click **OK** to complete the dialog. This adds two projects to the Visual Studio solution.

- An Azure project that defines the Cloud service, including configuration information.
- A worker role project that defines the worker role.



Inside the worker role, we poll for messages by calling the `ProcessMessageAsync` method of the `FixItQueueManager` class that we saw earlier.

```
public class WorkerRole : RoleEntryPoint
{
```

```
    public override void Run()
    {
        Task task = RunAsync(tokenSource.Token);
        try
        {
            task.Wait();
        }
        catch (Exception ex)
        {
            logger.Error(ex, "Unhandled exception in FixIt worker role.");
        }
    }

    private async Task RunAsync(CancellationToken token)
    {
        using (var scope = container.BeginLifetimeScope())
        {
            IFixItQueueManager queueManager = scope.Resolve<IFixItQueueManager>();
            try
            {
                await queueManager.ProcessMessagesAsync(token);
            }
            catch (Exception ex)
            {
                logger.Error(ex, "Exception in worker role Run loop.");
            }
        }
    }
    // Other code not shown.
}
```

The `ProcessMessagesAsync` method checks whether a message is waiting. If one is, the method deserializes the message into a `FixItTask` entity and saves the entity in the database. It loops until the worker role is stopped.

```
public async Task ProcessMessagesAsync(CancellationToken token)
{
    CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
    await queue.CreateIfNotExistsAsync();

    while (!token.IsCancellationRequested)
    {
        CloudQueueMessage message = await queue.GetMessageAsync(token);
        if (message != null)
        {
            FixItTask fixit = JsonConvert.DeserializeObject<FixItTask>(message.AsString);
            await _repository.CreateAsync(fixit);
            await queue.DeleteMessageAsync(message);
        }
    }
}
```

In a web project, adding asynchronous code can automatically improve performance because IIS manages a limited thread pool. That is not the case in a worker role project. To improve scalability of the worker role, you can write multithreaded code or use asynchronous code to implement parallel programming. The sample doesn't implement parallel programming but shows how to make the code asynchronous so that you can implement parallel programming.

Summary

In this chapter you've seen how to improve application responsiveness, reliability, and scalability by implementing the queue-centric work pattern.

This is the last of the 13 patterns covered in this ebook, but there are, of course, many other patterns and practices that can help you build successful cloud apps. The final chapter provides links to resources for topics that haven't been covered in these 13 patterns.

Resources

For more information about queues, see the following resources.

Documentation:

- Microsoft Azure Storage Queues Part 1: Getting Started Article by Roman Schacherl.
- Executing Background Tasks Chapter 5 of Moving Applications to the Cloud, 3rd Edition from Microsoft Patterns & Practices. (In particular, see the section Using Azure Storage Queues.)
- Best Practices for Maximizing Scalability and Cost Effectiveness of Queue-Based Messaging Solutions on Windows Azure White paper by Valery Mizonov.
- Comparing Microsoft Azure Queues and Service Bus Queues *MSDN Magazine* article, provides additional information that can help you choose which queue service to use. The article mentions that Service Bus is dependent on ACS for authentication, which means your SB queues would be unavailable when ACS is unavailable. However, since the article was written, SB was changed to enable you to use SAS tokens as an alternative to ACS.
- Microsoft Patterns & Practices—Azure Guidance See Asynchronous Messaging primer, Pipes and Filters pattern, Compensating Transaction pattern, Competing Consumers pattern, CQRS pattern.
- CQRS Journey Ebook about CQRS by Microsoft Patterns & Practices.
- Creating an Azure Project with Visual Studio How to create Azure Cloud Service solutions with worker roles and web roles.

Video:

- FailSafe: Building Scalable, Resilient Cloud Services Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from the Microsoft Customer Advisory Team's experience with actual customers. For an introduction to the Azure Storage service and queues, see episode 5 starting at 35:13.

Chapter 14

More patterns and guidance

You've now seen 13 patterns that provide guidance on how to be successful in cloud computing. These are just a few of the patterns that apply to cloud apps. Here are some more cloud computing topics and resources to help with them:

- Migrating existing on-premises applications to the cloud.
  - Moving Applications to the Cloud, 3rd edition Ebook by Microsoft Patterns & Practices.
  - Migrating Microsoft's ASP.NET and IIS.NET Case study by Robert McMurray.
  - Moving 4th & Mayor to Windows Azure Web Sites Blog post by Jeff Wilcox chronicling his experience moving a web app from Amazon Web Services to Azure Websites.
  - Moving Apps to the Azure: What Changes? Short video by Stefan Schackow, explains file system access in Azure Websites.
- Security, authentication, and authorization issues unique to cloud applications
  - Azure Security Guidance
  - Azure Network Security
  - Microsoft Patterns & Practices—Azure Guidance See Gatekeeper pattern, Federated Identity pattern.

See also additional cloud computing patterns and guidance at Microsoft Patterns & Practices—Azure Guidance.

Resources

Each of the chapters in this ebook provides links to resources for more information about that specific topic. The following list provides links to overviews of best practices and recommended patterns for successful cloud development with Azure.

Documentation:

- Best Practices for the Design of Large-Scale Services on Azure Cloud Services White paper by Mark Simms and Michael Thomassy.
- Failsafe: Guidance for Resilient Cloud Architectures White paper by Marc Mercuri, Ulrich

Homann, and Andrew Townhill. Webpage version of the FailSafe video series.

- Azure Guidance Portal page on azure.microsoft.com for official documentation related to designing applications for Azure.

Videos:

- Building Real World Cloud Apps with Windows Azure—Part 1 and Part 2 Video of the presentation by Scott Guthrie that this ebook is based on. Presented at Tech Ed Australia in September 2013. An earlier version of the same presentation was delivered at Norwegian Developers Conference (NDC) in June 2013: NDC part 1, NDC part 2.
- FailSafe: Building Scalable, Resilient Cloud Services Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents a 400-level view of how to architect cloud apps. This series focuses on theory and reasons behind recommended patterns; for more how-to details, see the Building Big series by Mark Simms.
- Building Big: Lessons learned from Windows Azure customers- Part 1 and Part 2 Two-part video series by Simon Davies and Mark Simms, similar to the FailSafe series but oriented more toward practical implementation.

Code sample:

- The Fix It application that accompanies this ebook.
- Cloud Service Fundamentals in Windows Azure in C# for Visual Studio 2012 Downloadable project in the Microsoft Code Gallery site, includes both code and documentation developed by the Microsoft Customer Advisory Team (CAT). Demonstrates many of the best practices advocated in the FailSafe and Building Big video series and the FailSafe white paper. The Code Gallery page also links to extensive documentation by the authors of the project—see especially the Cloud Service Fundamentals wiki collection link in the blue box near the top of the project description. This project and the documentation for it is still actively being developed, making it a better choice for information on many topics than similar but older white papers.

Hard-copy books:

- Cloud Computing Bible By Barrie Sosinsky.
- Release It! Design and Deploy Production-Ready Software By Michael T. Nygard.
- Cloud Architecture Patterns By Bill Wilder.
- Windows Azure Platform By Tejaswi Redkar.
- Windows Azure Programming Patterns for Start-ups By Riccardo Becker.
- Microsoft Windows Azure Development Cookbook By Neil Mackenzie.

Finally, when you get started building real-world apps and running them in Azure, sooner or later you'll probably need assistance from experts. You can ask questions on community sites such as Azure forums or StackOverflow, or you can contact Microsoft directly for Azure support. Microsoft offers several levels of technical support Azure: for a summary and comparison of the options, see Azure Support Plans.

Appendix

The Fix It sample application

You can download the sample application here: The Fix It Project.

This appendix contains the following sections, which provide additional information about the Fix It sample application:

- Known issues
- Best practices
- How to run the app from Visual Studio on your local computer
- How to deploy the base app to an Azure website by using the Windows PowerShell scripts
- Troubleshooting the Windows PowerShell scripts
- How to deploy the app with queue processing to an Azure website and an Azure cloud service

Known issues

The Fix It app was originally developed to illustrate as simply as possible some of the patterns presented in this ebook. However, since the ebook is about building real-world apps, we subjected the Fix It code to a review and testing process similar to what we'd do for released software. We found a number of issues, and as with any real-world application, some of them we fixed and some of them we deferred to a later release.

The following list includes issues that should be addressed in a production application, but for one reason or another we decided not to address them in the initial release of the Fix It sample application.

Security

- Ensure that you can't assign a task to a nonexistent owner.
- Ensure that you can view and modify only tasks that you created or are assigned to you.
- Use HTTPS for sign-in pages and authentication cookies.
- Specify a time limit for authentication cookies.

Input validation

In general, a production app would do more input validation than the Fix It app has. For example, the image size and image file size allowed for upload should be limited.

Administrator functionality

An administrator should be able to change ownership of existing tasks. For example, the creator of a task might leave the company, leaving no one with authority to maintain the task unless administrative access is enabled.

Queue message processing

Queue message processing in the Fix It app was designed to be simple to illustrate the queue-centric work pattern with a minimum amount of code. This simple code would not be adequate for an actual production application.

- The code does not guarantee that each queue message will be processed at most once. When you get a message from the queue, there is a timeout period, during which the message is invisible to other queue listeners. If the timeout expires before the message is deleted, the message becomes visible again. Therefore, if a worker role instance spends a long time processing a message, it is theoretically possible for the same message to be processed twice, resulting in a duplicate task in the database. For more information about this issue, see Using Azure Storage Queues.

- The queue polling logic could be made more cost-effective by batching message retrieval. Every time you call CloudQueue.GetMessageAsync, there is a transaction cost. Instead, you can call CloudQueue.GetMessagesAsync (note the plural "messages"), which gets multiple messages in a single transaction. The transaction costs for Azure Storage Queues are very low, so the impact on costs is not substantial in most scenarios.

- The tight loop in the queue-message-processing code causes CPU affinity, which does not utilize multicore VMs efficiently. A better design would use task parallelism to run several async tasks in parallel.

- Queue message processing has only rudimentary exception handling. For example, the code doesn't handle poison messages. (When message processing causes an exception, you have to log the error and delete the message, or the worker role will try to process it again, and the loop will continue indefinitely.)

SQL queries are unbounded

Current Fix It code places no limit on how many rows the queries for Index pages might return. If a large volume of tasks is entered into the database, the size of the resulting lists received could cause

performance issues. The solution is to implement paging. For an example, see Sorting, Filtering, and Paging with the Entity Framework in an ASP.NET MVC Application.

View models recommended

The Fix It app uses the FixItTask entity class to pass information between the controller and the view. A best practice is to use view models. The domain model (for example, the FixItTask entity class) is designed around what is needed for data persistence, while a view model can be designed for data presentation. For more information, see 12 ASP.NET MVC Best Practices.

Secure image blob recommended

The Fix It app stores uploaded images as public, meaning that anyone who finds the URL can access the images. The images could be secured instead of public.

No PowerShell automation scripts for queues

Sample PowerShell automation scripts were written only for the base version of Fix It, which runs entirely in an Azure website. We haven't provided scripts for setting up and deploying to the Website plus Cloud Service environment required for queue processing.

Special handling for HTML codes in user input

ASP.NET automatically prevents many ways in which malicious users might attempt cross-site scripting attacks by entering script in user input text boxes. And the MVC `DisplayFor` helper used to display task titles and notes automatically HTML-encodes values that it sends to the browser. But in a production app you might want to take additional measures. For more information, see Request Validation in ASP.NET.

Best practices

Following are some issues that were fixed after being discovered in code review and testing of the original version of the Fix It app. Some were caused by the original coder not being aware of a particular best practice, some simply because the code was written quickly and wasn't intended for released software. We're listing the issues here in case we learned something from this review and testing that might be helpful to others who are also developing web apps.

Dispose the database repository

The `FixItTaskRepository` class must dispose the Entity Framework `DbContext` instance. We did this by implementing `IDisposable` in the `FixItTaskRepository` class:

```
public class FixItTaskRepository : IFixItTaskRepository, IDisposable
{
    private MyFixItContext db = new MyFixItContext();

    // other code not shown
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Free managed resources.
            if (db != null)
            {
                db.Dispose();
                db = null;
            }
        }
    }
}
```

Note that AutoFac will automatically dispose the `FixItTaskRepository` instance, so we don't need to explicitly dispose it.

Another option is to remove the `DbContext` member variable from `FixItTaskRepository` and instead create a local `DbContext` variable within each repository method, inside a `using` statement. For example:

```
// Alternate way to dispose the DbContext
using (var db = new MyFixItContext())
{
    fixItTask = await db.FixItTasks.FindAsync(id);
}
```

Register singletons as such with DI

Because only one instance of the `PhotoService` class and `Logger` class is needed, these classes should be registered as single instances for dependency injection in DependenciesConfig.cs:

```
builder.RegisterType<Logger>().As<ILogger>().SingleInstance();
builder.RegisterType<FixItTaskRepository>().As<IFixItTaskRepository>();
builder.RegisterType<PhotoService>().As<IPhotoService>().SingleInstance();
```

Security: Don't show error details to users

The original Fix It app didn't have a generic error page and just let all exceptions bubble up to the UI,

so some exceptions, such as database connection errors, could result in a full stack trace being displayed to the browser. Detailed error information can sometimes facilitate attacks by malicious users. The solution is to log the exception details and display an error page to the user that doesn't include error details. The Fix It app was already logging; to display an error page, we added `<customErrors mode=On>` in the Web.config file.

```
<system.web>
    <customErrors mode="On"/>
    <authentication mode="None" />
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
</system.web>
```

By default, this causes Views\Shared\Error.cshtml to be displayed for errors. You can customize Error.cshtml or create your own error page view and add a `defaultRedirect` attribute. You can also specify different error pages for specific errors.

Security: Only allow a task to be edited by its creator

The Dashboard Index page shows only tasks created by the logged-on user, but a malicious user could create a URL with an ID to another user's task. We added code in DashboardController.cs to return a 404 in that case:

```
public async Task<ActionResult> Edit(int id)
{
    FixItTask fixittask = await fixItRepository.FindTaskByIdAsync(id);
    if (fixittask == null)
    {
        return HttpNotFound();
    }

    // Verify logged in user owns this FixIt task.
    if (User.Identity.Name != fixittask.Owner)
    {
        return HttpNotFound();
    }

    return View(fixittask);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(int id, [Bind(Include =
"CreatedBy,Owner,Title,Notes,PhotoUrl,IsDone")]FormCollection form)
{
    FixItTask fixittask = await fixItRepository.FindTaskByIdAsync(id);

    // Verify logged in user owns this FixIt task.
    if (User.Identity.Name != fixittask.Owner)
    {
```

```
            return HttpNotFound();
        }

        if (TryUpdateModel(fixittask, form))
        {
            await fixItRepository.UpdateAsync(fixittask);
            return RedirectToAction("Index");
        }

        return View(fixittask);
    }
```

Don't swallow exceptions

The original Fix It app just returned null after logging an exception that resulted from a SQL query:

```
catch (Exception e)
{
    log.Error(e, "Error in FixItTaskRepository.FindTasksByOwnerAsync(userName={0})", userName);
    return null;
}
```

This would make it look to the user as though the query succeeded but just didn't return any rows. The solution is to rethrow the exception after catching and logging:

```
catch (Exception e)
{
    log.Error(e, "Error in FixItTaskRepository.FindTasksByCreatorAsync(creater={0})", creator);
    throw;
}
```

Catch all exceptions in worker roles

Any unhandled exceptions in a worker role will cause the VM to be recycled, so you want to wrap everything you do in a try-catch block and handle all exceptions.

Specify length for string properties in entity classes

To display simple code, the original version of the Fix It app didn't specify lengths for the fields of the FixItTask entity, and as a result they were defined as varchar(max) in the database. As a result, the UI would accept almost any amount of input. Specifying lengths sets limits that apply both to user input in the webpage and column size in the database:

```
public class FixItTask
{
    public int FixItTaskId   { get; set; }
    [StringLength(80)]
    public string CreatedBy { get; set; }
    [Required]
    [StringLength(80)]
```

```
    public string Owner { get; set; }
    [Required]
    [StringLength(80)]
    public string Title { get; set; }
    [StringLength(1000)]
    public string Notes { get; set; }
    [StringLength(200)]
    public string PhotoUrl { get; set; }
    public bool IsDone { get; set; }
}
```

Mark private members as readonly when they aren't expected to change

For example, in the `DashboardController` class, an instance of `FixItTaskRepository` is created and isn't expected to change, so we defined it as readonly.

```
public class DashboardController : Controller
{
    private readonly IFixItTaskRepository fixItRepository = null;
```

Use list.Any() instead of list.Count() > 0

If all you care about is whether one or more items in a list fit the specified criteria, use the Any method because it returns as soon as an item fitting the criteria is found, whereas the `Count` method always has to iterate through every item. The Dashboard Index.cshtml file originally had this code:

```
@if (Model.Count() == 0) {
    <br />
    <div>You don't have anything currently assigned to you!!!</div>
}
```

We changed it to this:

```
@if (!Model.Any()) {
    <br />
    <div>You don't have anything currently assigned to you!!!</div>
}
```

Generate URLs in MVC views using MVC helpers

For the Create a Fix It button on the home page, the Fix It app hard coded an anchor element:

```
<a href="/Tasks/Create" class="btn btn-primary btn-large">Create a New FixIt
&raquo;</a>
```

For View/Action links like this it's better to use the Url.Action HTML helper; for example:

```
@Url.Action("Create","Tasks")
```

Use Task.Delay instead of Thread.Sleep in a worker role

The new-project template puts `Thread.Sleep` in the sample code for a worker role, but causing the thread to sleep can cause the thread pool to spawn additional unnecessary threads. You can avoid that by using Task.Delay instead.

```
while (true)
{
    try
    {
        await queueManager.ProcessMessagesAsync();
    }
    catch (Exception ex)
    {
        logger.Error(ex, "Exception in worker role Run loop.");
    }
    await Task.Delay(1000);
}
```

Avoid async void

If an async method doesn't need to return a value, return a `Task` type rather than *void*. This example is from the `FixItQueueManager` class:

```
// Correct
public async Task SendMessageAsync(FixItTask fixIt) { ... }

// Incorrect
public async void SendMessageAsync(FixItTask fixIt) { ... }
```

You should use `async void` only for top-level event handlers. If you define a method as `async void`, the caller cannot *await* the method or catch any exceptions the method throws. For more information, see Best Practices in Asynchronous Programming.

Use a cancellation token to break from a worker role loop

Typically, the `Run` method on a worker role contains an infinite loop. When the worker role is stopping, the RoleEntryPoint.OnStop method is called. You should use this method to cancel the work that is being done inside the `Run` method and exit gracefully. Otherwise, the process might be terminated in the middle of an operation.

Opt out of Automatic MIME Sniffing Procedure

In some cases, Internet Explorer reports a MIME type different from the type specified by the web server. For instance, if Internet Explorer finds HTML content in a file delivered with the HTTP response header Content-Type: text/plain, Internet Explorer determines that the content should be rendered as HTML. Unfortunately, this "MIME-sniffing" can also lead to security problems for servers hosting

untrusted content. To combat this problem, Internet Explorer (starting with Internet Explorer 8) has made a number of changes to MIME-type determination code and allows application developers to opt out of MIME-sniffing. The following code was added to the Web.config file.

```
<system.webServer>
    <httpProtocol>
        <customHeaders>
            <add name="X-Content-Type-Options" value="nosniff"/>
        </customHeaders>
    </httpProtocol>
    <modules>
        <remove name="FormsAuthenticationModule" />
    </modules>
</system.webServer>
```

Enable bundling and minification

When Visual Studio creates a new web project, bundling and minification of JavaScript files is not enabled by default. We added a line of code in BundleConfig.cs:

```
// For more information on bundling, visit
http://go.microsoft.com/fwlink/?LinkId=301862
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jquery").Include( "~/Scripts/jquery-{version}.js"));

    // Code removed for brevity/

    BundleTable.EnableOptimizations = true;
}
```

Set an expiration time-out for authentication cookies

By default, authentication cookies never expire. You have to manually specify an expiration time limit, as shown in the following code in StartupAuth.cs:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
   LoginPath = new PathString("/Account/Login"),
    ExpireTimeSpan = System.TimeSpan.FromMinutes(20)
});
```

How to run the app from Visual Studio on your local computer

There are two ways to run the Fix It app:

- Run the base MVC application.

- Run the application using a queue plus a back-end service to process work items. The queue pattern is described in Chapter 13, "Queue-centric work pattern."

The following instructions explain how to download the Fix It app and run the base version locally.

1. Install Visual Studio 2013 or Visual Studio 2013 Express for Web.
2. Install the Azure SDK for .NET for Visual Studio 2013.
3. Download the .zip file from the MSDN Code Gallery.
4. In File Explorer, right-click the .zip file and click Properties. In the Properties window, click Unblock.
5. Unzip the file.
6. Double-click the .sln file to launch Visual Studio.
7. From the Tools menu, click Library Package Manager, then Package Manager Console.
8. In the Package Manager Console (PMC), click Restore.
9. Exit Visual Studio.
10. Start the Azure storage emulator.
11. Restart Visual Studio, opening the solution file you closed in the previous step.
12. Make sure the Fix It project is set as the startup project, and then press CTRL+F5 to run the project.

To enable queues, make the following change in the MyFixIt\Web.config file. Under `appSettings`, change the value of `UseQueues` to "true":

```
<appSettings>
    <!-- Other settings not shown -->
    <add key="UseQueues" value="true"/>
</appSettings>
```

Next, modify the connection string in MyFixIt.WorkerRoler\app.config. In the value for connectionString, replace "{path}" with the full path to the MyFixIt\App_Data folder.

```
    <connectionStrings>
      <clear />
      <add name="appdb" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename={path}\MyFixIt\App_Data\MyFixItTasks.
mdf;Initial Catalog=MyFixItTasks;Integrated Security=True"
providerName="System.Data.SqlClient" />
    </connectionStrings>
```

For example:

```
    <connectionStrings>
      <clear />
      <add name="appdb" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=C:\Projects\MyFixIt\MyFixIt\App_Data\
MyFixItTasks.mdf;Initial Catalog=MyFixItTasks;Integrated Security=True"
providerName="System.Data.SqlClient" />
    </connectionStrings>
```

Next, you must run the Fix It project and the MyFixItCloudService project simultaneously. You run the MyFixItCloudService project inside the Azure compute emulator.

Using Visual Studio 2013:

1. Start the Azure storage emulator.
2. Start Visual Studio with administrator privileges. (The Azure compute emulator requires administrator privileges.)
3. Press F5 to run the Fix It project.
4. In Solution Explorer, right-click the MyFixItCloudService project.
5. Select **Debug**, and then select **Start New Instance.**

Using Visual Studio 2013 Express for Web:

1. Start the Azure storage emulator.
2. Start Visual Studio with administrator privileges.
3. In Solution Explorer, right-click the Fix It solution and select Properties.
4. Select **Multiple Startup Projects.**
5. Under MyFixIt and MyFixItCloudService, in the Action drop-down list, select Start.
6. Click **OK.**
7. Press F5 to debug both projects.

When you debug MyFixItCloudService, Visual Studio will start the Azure compute emulator. Depending on your firewall configuration, you might need to allow the emulator through the firewall.

How to deploy the base app to an Azure website by using the Windows PowerShell scripts

To illustrate the automate everything pattern, the Fix It app is supplied with scripts that set up an environment in Azure and deploy the project to the new environment. The following instructions

explain how to use the scripts.

If you want to run in Azure without using queues, and you made the changes to run locally with queues, be sure that you set the UseQueues `appSetting` value back to "false" before proceeding with the following instructions.

These instructions assume you have already downloaded and run the Fix It solution locally and that you have an Azure account or have an Azure subscription that you are authorized to manage.

1. Install the Azure PowerShell console. For instructions, see How to install and configure Azure PowerShell.

   This customized console is configured to work with your Azure subscription. The Azure module is installed in the Program Files directory and is automatically imported on every use of the Azure PowerShell console.

   If you prefer to work in a different host program, such as Windows PowerShell ISE, be sure to use the Import-Module cmdlet to import the Azure module or use a command in the Azure module to trigger automatic importing of the module.

2. Start Azure PowerShell with the **Run as administrator** option.

3. Run the Set-ExecutionPolicy cmdlet to set the Azure PowerShell execution policy to `RemoteSigned`. Enter Y (for Yes) to complete the policy change.

   ```
   PS C:\> Set-ExecutionPolicy RemoteSigned
   ```

   This setting enables you to run local scripts that aren't digitally signed. (You can also set the execution policy to `Unrestricted`, which would eliminate the need for the unblocking step later, but this is not recommended for security reasons.)

4. Run the `Add-AzureAccount` cmdlet to set up PowerShell with credentials for your account.

   ```
   PS C:\> Add-AzureAccount
   ```

   These credentials expire after a period of time, and you have to rerun the `Add-AzureAccount` cmdlet. As this ebook is being written, the time limit before credentials expire is 12 hours.

5. If you have multiple subscriptions, use the Set-AzureSubscription cmdlet to specify the subscription you want to create the test environment in.

6. Import a management certificate for the same Azure subscription by using the `Get-AzurePublishSettingsFile` and `Import-AzurePublishSettingsFile` cmdlets. The first of these cmdlets downloads a certificate file, and in the second one you specify the location of that file to import it.

   > Important: Keep the downloaded file in a safe location or delete it when you're done with it because it contains a certificate that can be used to manage your Azure services.

```
PS C:\Users\username\Documents\Visual Studio
2013\Projects\MyFixIt\Automation> Get-AzurePublishSettingsFile
PS C:\Users\username\Documents\Visual Studio
2013\Projects\MyFixIt\Automation> Import-AzurePublishSettingsFile
"C:\Users \username\Downloads\Windows Azure MSDN - Visual Studio
Ultimate-12-14-2013-credentials.publishsettings"
```

The certificate is used for a REST API call that detects the development machine's IP address to set a firewall rule on the SQL Database server.

7. Run the Set-Location cmdlet (aliases are `cd`, `chdir`, and `sl`) to navigate to the directory that contains the scripts. (They're located in the Automation folder in the Fix It solution folder.) Put the path in quotation marks if any of the directory names contain spaces. For example, to navigate to the *c:\Sample Apps\FixIt\Automation* directory, you could enter the following command:

   ```
   PS C:\> cd "c:\Sample Apps\MyFixIt\Automation"
   ```

8. To allow Windows PowerShell to run these scripts, use the Unblock-File cmdlet. (The scripts are blocked because they were downloaded from the Internet.)

   > Security Note: Before running `Unblock-File` on any script or executable file, open the file in Notepad, examine the commands, and verify that they do not contain any malicious code.

   For example, the following command runs the `Unblock-File` cmdlet on all scripts in the current directory.

   ```
   PS C:\Sample Apps\FixIt\Automation> Unblock-File -Path .\*.ps1
   ```

9. To create the Azure Website environment for the base (no queues processing) Fix It app, run the environment creation script.

   The required `Name` parameter specifies the name of the database and is also used for the storage account that the script creates. The name must be globally unique within the azurewebsites.net domain. If you specify a name that is not unique, like Fixit or Test (or, even as in the example, fixitdemo), the `New-AzureWebsite` cmdlet fails with an internal error that reports a conflict. The script converts the name to all lowercase to comply with name requirements for websites, storage accounts, and databases.

   The required `SqlDatabasePassword` parameter specifies the password for the admin account that will be created for SQL Database. Don't include special XML characters in the password (& < > ;). This is a limitation of the way the scripts were written, not a limitation of Azure.

   For example, if you want to create a Website named *fixitdemo* and use a SQL Server administrator password of *Passw0rd1*, you could enter the following command:

```
PS C:\Sample Apps\FixIt\Automation> .\New-AzureWebsiteEnv.ps1 -Name fixitdemo -
SqlDatabasePassword Passw0rd1
```

The website name must be unique in the azurewebsites.net domain, and the password must meet SQL Database requirements for password complexity. (The example Passw0rd1 does meet the requirements.)

Note that the command begins with ".\". To help prevent malicious execution of scripts, Windows PowerShell requires that you provide the fully qualified path to the script file when you run a script. You can use a dot to indicate the current directory (".\") or provide the fully qualified path, such as:

```
PS C:\Temp\FixIt\Automation> C:\Temp\FixIt\Automation\New-
AzureWebsiteEnv.ps1 -Name fixitdemo -SqlDatabasePassword Pas$w0rd
```

For more information about the script, use the `Get-Help` cmdlet.

```
PS C:\Sample Apps\FixIt\Automation> Get-Help -Full .\New- AzureWebsiteEnv.ps1
```

You can use the `Detailed`, `Full`, `Parameters`, and `Examples` parameters of the `Get-Help` cmdlet to filter the help that is returned.

If the script fails or generates errors, such as "New-AzureWebsite : Call Set-AzureSubscription and Select-AzureSubscription first," you might not have completed the configuration of Azure PowerShell.

After the script finishes, you can use the Azure management portal to see the resources that were created, as shown in Chapter 1, "Automate everything."

10. To deploy the Fix It project to the new Azure environment, use the AzureWebsite.ps1 script. For example:

```
PS C:\Sample Apps\FixIt\Automation> .\Publish-AzureWebsite.ps1
..\MyFixIt\MyFixIt.csproj -Launch
```

When deployment is done, the browser opens with Fix It running in Azure.

Troubleshooting the Windows PowerShell scripts

The most common errors encountered when running these scripts are related to permissions. Be sure that `Add-AzureAccount` and `Import-AzurePublishSettingsFile` were successful and that you used them for the same Azure subscription. Even if `Add-AzureAccount` was successful, you might have to run it again. The permissions added by `Add-AzureAccount` expire in 12 hours.

Object reference not set to an instance of an object.

If the script returns errors, such as "Object reference not set to an instance of an object," which means that Windows PowerShell can't find an object to process (this is a null reference exception), run the `Add-AzureAccount` cmdlet and try the script again.

```
New-AzureSqlDatabaseServer : Object reference not set to an instance of an object.
At C:\ps-test\azure-powershell-samples-master\WebSite\create-azure-sql.ps1:80 char:19
+ $databaseServer = New-AzureSqlDatabaseServer -AdministratorLogin $UserName
-Admi ...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~
+ CategoryInfo:          NotSpecified: (:) [New-AzureSqlDatabaseServer],
NullReferenceException
    + FullyQualifiedErrorId :
Microsoft.WindowsAzure.Commands.SqlDatabase.Server.Cmdlet.NewAzureSqlDatabase Server
```

InternalError: The server encountered an internal error.

The `New-AzureWebsite` cmdlet returns an internal error when the website name is not unique in the azurewebsites.net domain. To resolve the error, use a different value for the website name, which is in the `Name` parameter of New-AzureWebsiteEnv.ps1.

```
New-AzureWebsite : InternalError: The server encountered an internal error.
Please retry the request.
At line:1 char:1 + New-AzureWebsite -Name fixitdemo
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : CloseError: (:) [New-AzureWebsite], Exception
+ FullyQualifiedErrorId :
Microsoft.WindowsAzure.Commands.Websites.NewAzureWebsiteCommand
```

Restarting the script

If you need to restart the New-AzureWebsiteEnv.ps1 script because it failed before it printed the "Script is complete" message, you might want to delete resources that the script created before it stopped. For example, if the script already created the ContosoFixItDemo website and you run the script again with the same website name, the script will fail because the website name is in use.

To determine which resources the script created before it stopped, use the following cmdlets:

- `Get-AzureWebsite`
- `Get-AzureSqlDatabaseServer`
- `Get-AzureSqlDatabase`—to run this cmdlet, pipe the database server name to `Get-AzureSqlDatabase`:
- `Get-AzureSqlDatabaseServer | Get-AzureSqlDatabase.`

To delete these resources, use the following commands. Note that if you delete the database server, you automatically delete the databases associated with the server.

- `Get-AzureWebsite -Name <WebsiteName> | Remove-AzureWebsite`
- `Get-AzureSqlDatabase -Name <DatabaseName> -ServerName <DatabaseServerName> | Remove-SqlAzureDatabase`
- `Get-AzureSqlDatabaseServer | Remove-AzureSqlDatabaseServer`

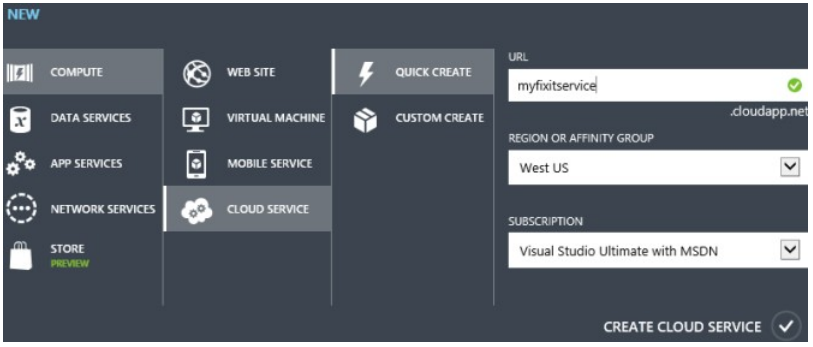
How to deploy the app with queue processing to an Azure website and an Azure cloud service

To enable queues, make the following change in the MyFixIt\Web.config file. Under `appSettings`, change the value of `UseQueues` to "true":

```
<appSettings>
    <!-- Other settings not shown -->
    <add key="UseQueues" value="true"/>
</appSettings>
```

Then deploy the MVC application to an Azure website as described in "How to deploy the base app to an Azure website by using the Windows PowerShell scripts" earlier in this appendix.

Next, create a new Azure cloud service. The scripts included with the Fix It app do not create or deploy the cloud service, so you must use the Azure management portal for this. In the portal, click New, **Compute, Cloud Service, Quick Create**, and then enter a URL and a data center location. Use the same data center where you deployed the website.



Before you can deploy the cloud service, you need to update some of the configuration files.

In MyFixIt.WorkerRoler\app.config, under `connectionStrings`, replace the value of the `appdb` connection string with the actual connection string for the SQL Database. You can get the connection

string from the portal. In the portal, click SQL Databases, appdb, View SQL Database connection strings for ADO .Net, ODBC, PHP, and JDBC. Copy the ADO.NET connection string and paste the value into the app.config file. Replace "{your_password_here}" with your database password. (Assuming you used the scripts to deploy the MVC app, you specified the database password in the `SqlDatabasePassword` script parameter.)

The result should look like the following:

```
<add name="appdb"
connectionString="Server=tcp:####.database.windows.net,1433;Database=appdb;Us
er ID=####;Password=####;Trusted_Connection=False;Encrypt=True;Connection
Timeout=30;" providerName="System.Data.SqlClient" />
```

In the same MyFixIt.WorkerRoler\app.config file, under `appSettings`, replace the two placeholder values for the Azure Storage account.

```
<appSettings>
    <add key="StorageAccountName" value="{StorageAccountName}" />
    <add key="StorageAccountAccessKey" value="{StorageAccountAccessKey}" />
</appSettings>
```

You can get the access key from the portal. See How to Manage Storage Accounts.

In MyFixItCloudService\ServiceConfiguration.Cloud.cscfg, replace the same two placeholders' values for the Azure Storage account.

```
    <ConfigurationSettings>
    <Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"

value="DefaultEndpointsProtocol=https;AccountName={StorageAccountName};AccountKey={key}"/>
    </ConfigurationSettings>
```

Now you are ready to deploy the cloud service. In Solution Explorer, right-click the MyFixItCloudService project and select Publish. For more information, see Get Started with Azure Cloud Services and ASP.NET.

About the authors

As executive vice president of the Microsoft Cloud and Enterprise group, **Scott Guthrie** is responsible for the company's cloud infrastructure, server, database, management, and development tools businesses. His engineering team builds Microsoft Azure, Windows Server, SQL Server, Active Directory, System Center, Visual Studio, and .NET. Prior to leading the Cloud and Enterprise group, Guthrie helped lead Microsoft Azure, Microsoft's public cloud platform. Since joining the company in 1997, he has made critical contributions to many of Microsoft's key cloud, server, and development technologies and was one of the original founders of the .NET project. Guthrie graduated with a bachelor's degree in computer science from Duke University. He lives in Seattle with his wife and two children.

Mark Simms is a principal group program manager on the AzureCAT (Azure Customer Advisory Team), working on large-scale Azure architecture and implementation. He is usually found nerding out on challenging customer problems and is currently focused on architecture and application patterns at scale, making big things work on Azure. Mark's specialties are cloud architecture and implementation, real-time analytics, distributed architecture and applications, hybrid cloud applications, and big data approaches.

Tom Dykstra is a senior programming writer on Microsoft's Web Platform team. Tom has a PhD from the University of Washington. He writes about Azure, ASP.NET, web deployment, Entity Framework, and other topics related to the Microsoft web stack. Tom's blog can be found at http://tomdykstra.wordpress.com, and he is on Twitter at http://www.twitter.com/tdykstra.

Rick Anderson is a senior programming writer on Microsoft's Web Platform team. Rick has an MS in applied math from Montana State University. He writes about the Microsoft web stack on Azure, including ASP.NET MVC, security, and ASP.NET Identity. You can follow Rick on Twitter at http://twitter.com/RickAndMSFT.

Mike Wasson is a senior programming writer on Microsoft's Web Platform team. He writes about ASP.NET, Web API, Entity Framework, and other topics related to the Microsoft web stack. Mike has an MS degree in computer science from the University of Washington.

From technical overviews to drilldowns on special topics, get *free* ebooks from Microsoft Press at:

www.microsoftvirtualacademy.com/ebooks

Download your free ebooks in PDF, EPUB, and/or Mobi for Kindle formats.

Look for other great resources at Microsoft Virtual Academy, where you can learn new skills and help advance your career with free Microsoft training delivered by experts.

Microsoft Press

Now that you've read the book...

Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at http://aka.ms/tellpress

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