Building Real-World Cloud Apps with Windows Azure

Tom Dykstra Rick Anderson Mike Wasson

Guide



Microsoft

www.dbooks.org

Building Real-World Cloud Apps with Windows Azure

Tom Dykstra Rick Anderson Mike Wasson

Summary: This e-book 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 originally delivered at the Norwegian Developers Conference (NDC) in June of 2013. Many others updated and augmented the content while transitioning it from video to written form.

Category: Guide
Applies to: Windows Azure Web Sites, ASP.NET, Visual Studio, Visual Studio Online, Windows Azure Active Directory, Windows Azure SQL Database,
Source: ASP.NET site (source content)
E-book publication date: January, 2014

Microsoft

Copyright © 2014 by Microsoft Corporation

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

Microsoft and the trademarks listed at http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

www.dbooks.org

Table of Contents

Building Real-World Cloud Apps with Windows Azure 1
Table of Contents 3
Introduction 7
Intended Audience 7
Cloud development patterns 7
The Fix it sample application 9
Windows Azure Web Sites 13
Summary 16
Resources 16
Automate Everything 17
DevOps Workflow 17
Windows Azure management scripts 18
Environment creation script 18
Deployment script 28
Summary 30
Resources 31
Source Control 32
Treat automation scripts as source code 32
Don’t check in secrets 32
Structure source branches to facilitate DevOps workflow 33
Add scripts to source control in Visual Studio 35
Store sensitive data in Windows Azure 39
Use Git in Visual Studio and Visual Studio Online 42
Summary 50
Resources 51
Continuous Integration and Continuous Delivery 52
Continuous Integration and Continuous Delivery workflow 52
How the cloud enables cost-effective CI and CD 53
Visual Studio Online 53
Resources 54
Web Development Best Practices 56

Stateless web tier behind a smart load balancer 56
Avoid session state 61
Use a CDN to cache static file assets 61
Use .NET 4.5's async support to avoid blocking calls 61
Summary 64
Resources 64
Single Sign-On 66
Introduction to WAAD 66
Set up a WAAD tenant 69
Create an ASP.NET app that uses WAAD for single sign-on 82
Summary 87
Resources 88
Data Storage Options 89
Data storage options on Windows Azure 89
Hadoop and MapReduce 91
Platform as a Service (PaaS) versus Infrastructure as a Service (IaaS) 95
Choosing a data storage option 98
Demo – using SQL Database in Windows Azure 100
Entity Framework versus direct database access using ADO.NET 108
SQL databases and the Entity Framework in the Fix It app 108
Choosing SQL Database (PaaS) versus SQL Server in a VM (IaaS) in Windows Azure 110
Summary 113
Resources 113
Data Partitioning Strategies 116
The three Vs of data storage 116
Vertical partitioning 116
Horizontal partitioning (sharding) 118
Hybrid partitioning 119
Partitioning a production application 119
Summary 119
Resources 120
Unstructured Blob Storage 121
What is Blob storage? 121

www.dbooks.org

Creating a Storage account 121
Using Blob storage in the Fix It app 123
Summary 129
Resources 130
Design to Survive Failures 131
Types of failures 131
Failure scope 131
SLAs 132
Summary 135
Resources 136
Monitoring and Telemetry 137
Buy or rent a telemetry solution 137
Log for insight 155
Logging in the Fix It app 158
Dependency Injection in the Fix It app 162
Built-in logging support in Windows Azure 163
Summary 167
Resources 167
Transient Fault Handling 169
Causes of transient failures 169
Use smart retry/back-off logic to mitigate the effect of transient failures 169
Circuit breakers 170
Summary 172
Resources 172
Distributed Caching 174
What is distributed caching 174
When to use distributed caching 174
Popular cache population strategies 174
Sample cache-aside code for Fix It app 175
Popular caching frameworks 176
ASP.NET session state using a cache provider 177
Summary 177
Resources 177

Queue-Centric Work Pattern 179
Reduced Latency 179
Increased Reliability 179
Rate Leveling and Independent Scaling 181
Adding Queues to the Fix It Application 182
Creating Queue Messages 182
Processing Queue Messages 184
Summary 189
Resources 189
More Patterns and Guidance 191
Resources 191
Acknowledgments 192
Appendix: The Fix It Sample Application 194
Known issues 194
Best practices 196
How to Run the App from Visual Studio on Your Local Computer 202
How to deploy the base app to a Windows Azure Web Site by using the Windows PowerShell scripts 204
Troubleshooting the Windows PowerShell scripts 207
How to deploy the app with queue processing to a Windows Azure Web Site and a Windows Azure Cloud Service 208

www.dbooks.org

Introduction

Download Sample Application: Fix It Project

This e-book 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.

Intended Audience

Developers who are curious about developing for the cloud, 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 links to other resources for 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.

Cloud development patterns

This e-book 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 which 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: Windows Azure management scripts.
- Source control.
  - Set up branching structure in source control to facilitate 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 auto-scaling in Windows Azure Web Sites.
  - Avoid session state.
  - Use a CDN.
  - Use asynchronous programming model.
  - Demo: async in ASP.NET MVC and Entity Framework.
- Single sign-on.
  - Introduction to Windows Azure Active Directory.
  - Demo: create an ASP.NET app that uses Windows Azure Active Directory.
- Data storage options.
  - Types of data stores.
  - How to choose the right data store.
  - Demo: Windows 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 Windows Azure
  - Demo: logging code in the Fix It app.
  - Demo: built-in logging support in Windows 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: Windows 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 Windows Azure.

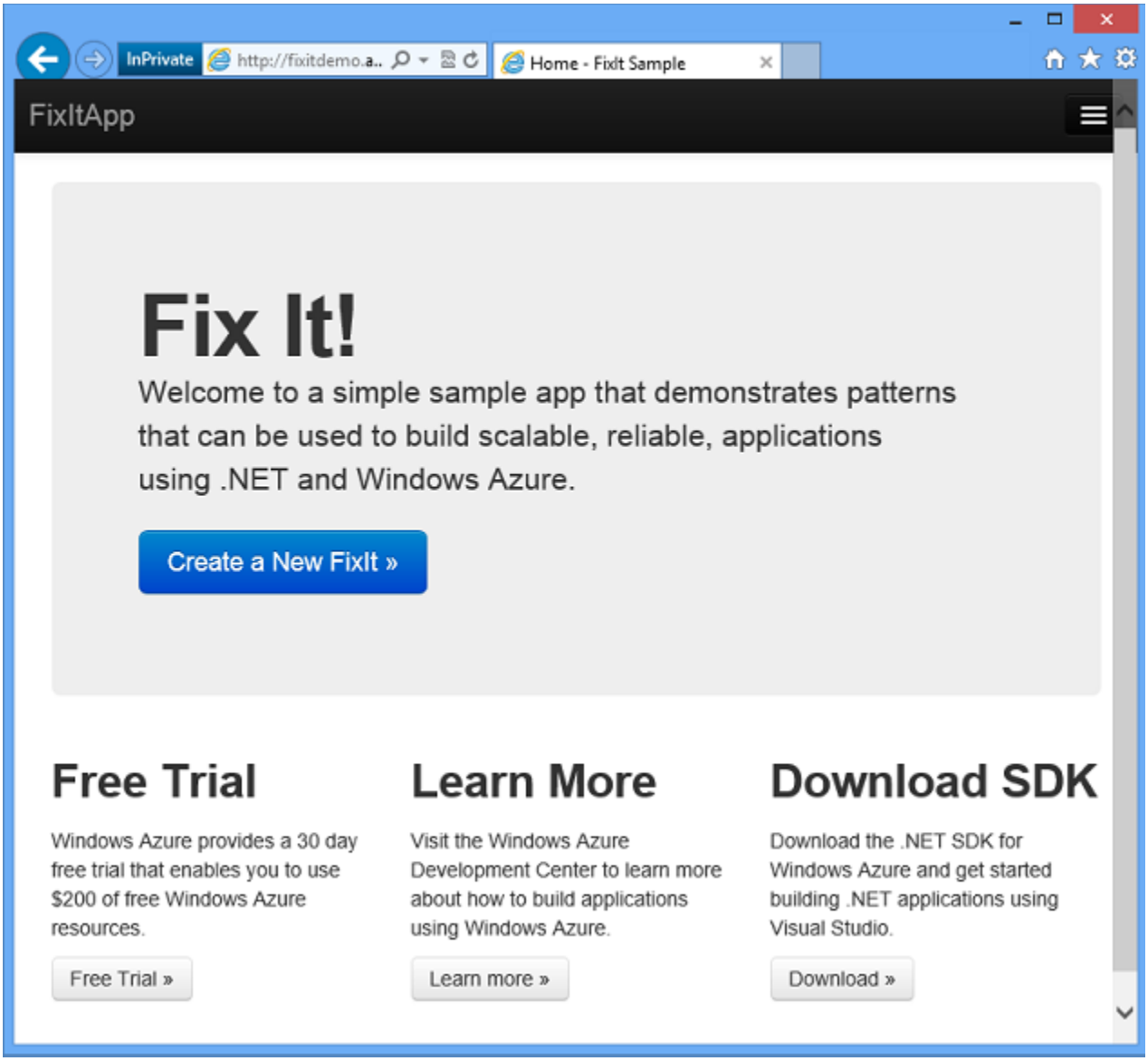
www.dbooks.org

This remainder of this chapter introduces the Fix It sample application and the Windows Azure Web Sites cloud environment that the Fix It app runs in:

- The Fix It sample application
- Introduction to Windows Azure Web Sites
- Getting started

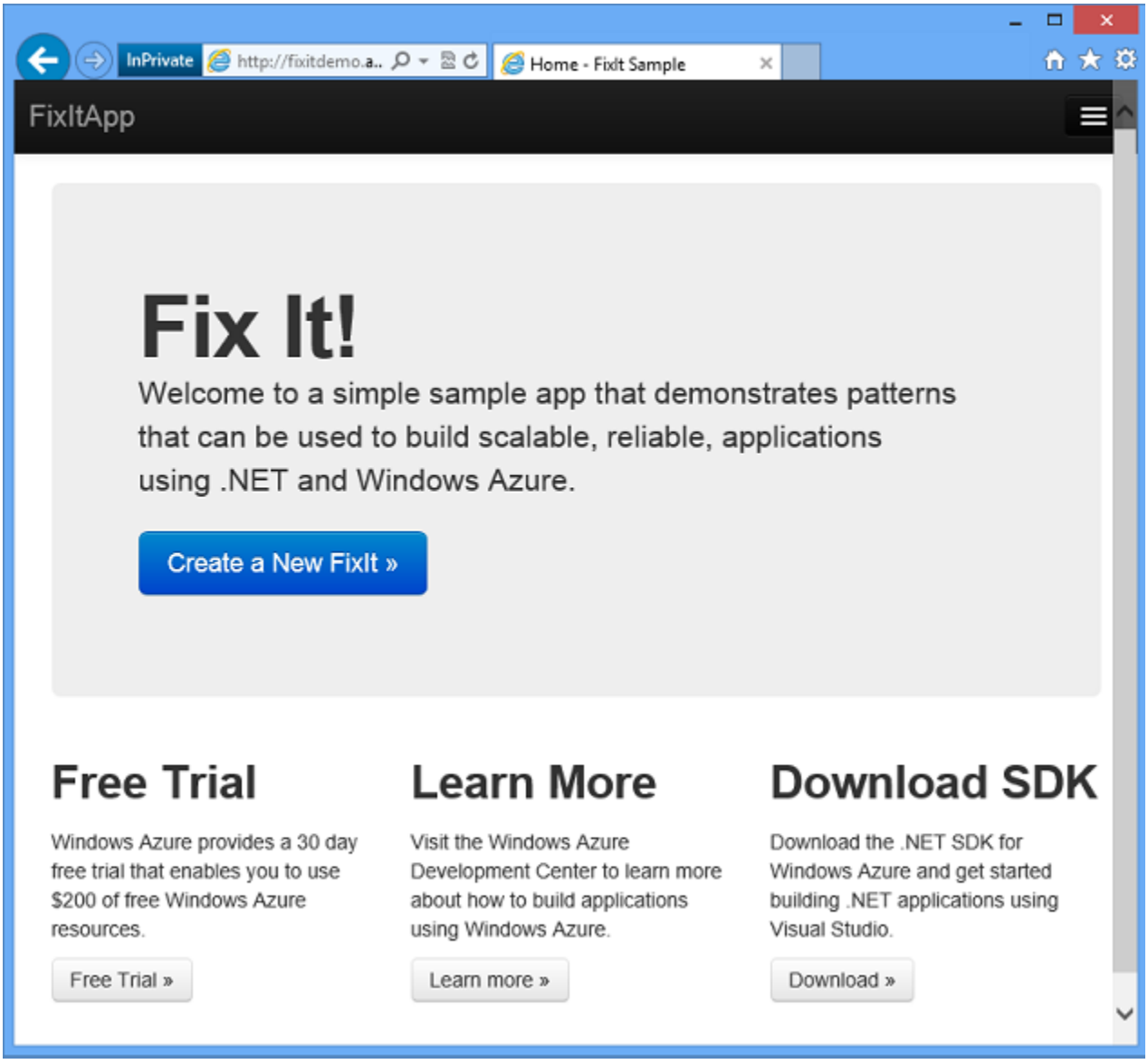
The Fix it sample application

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



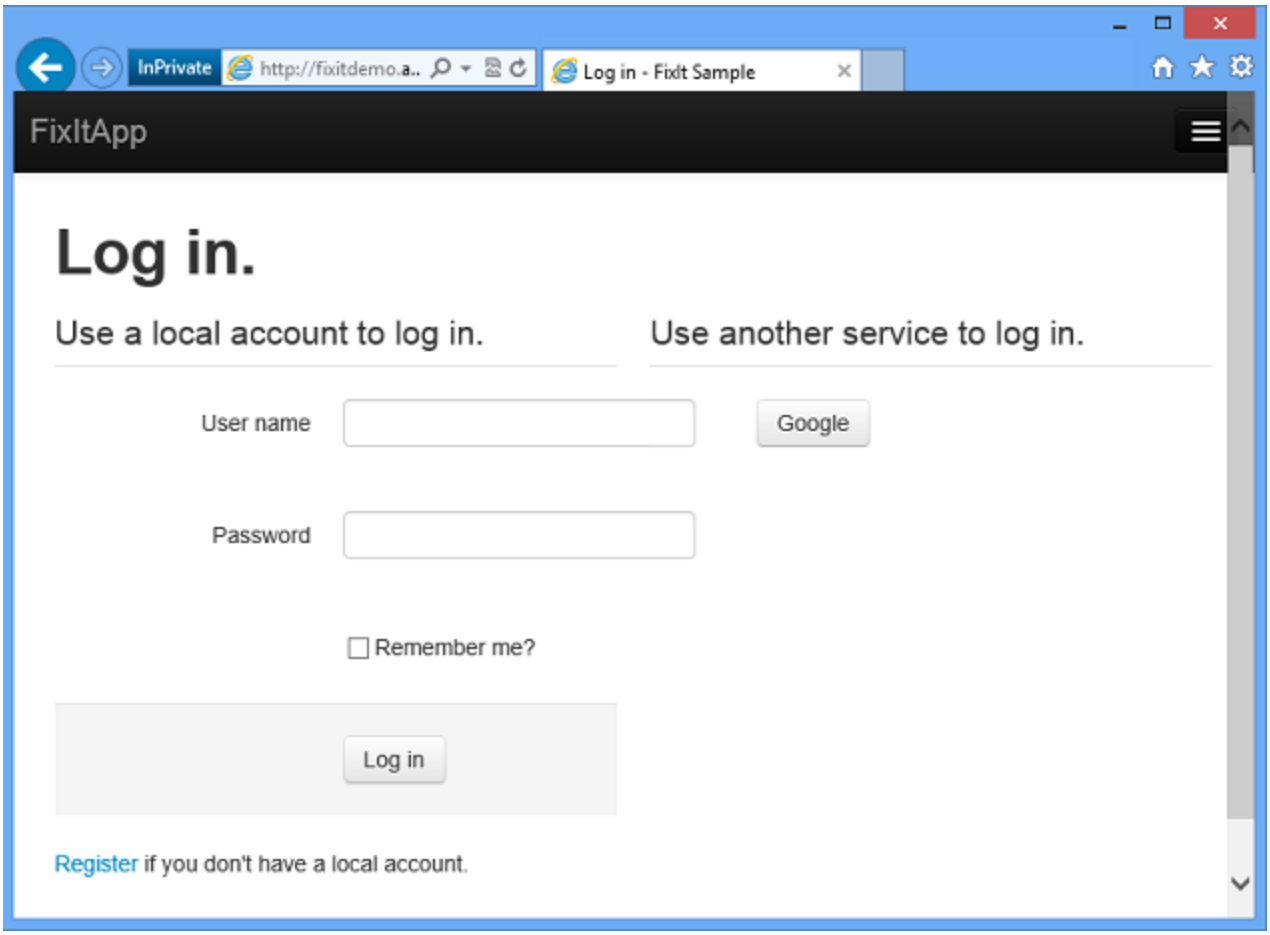
The sample app is a simple work item ticketing system. When you need something fixed, you create a ticket and assign it to someone, and others can log in and see the tickets assigned to them and mark tickets as completed when the work is done.

It's 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 a Windows Azure Web Site to run in the cloud.

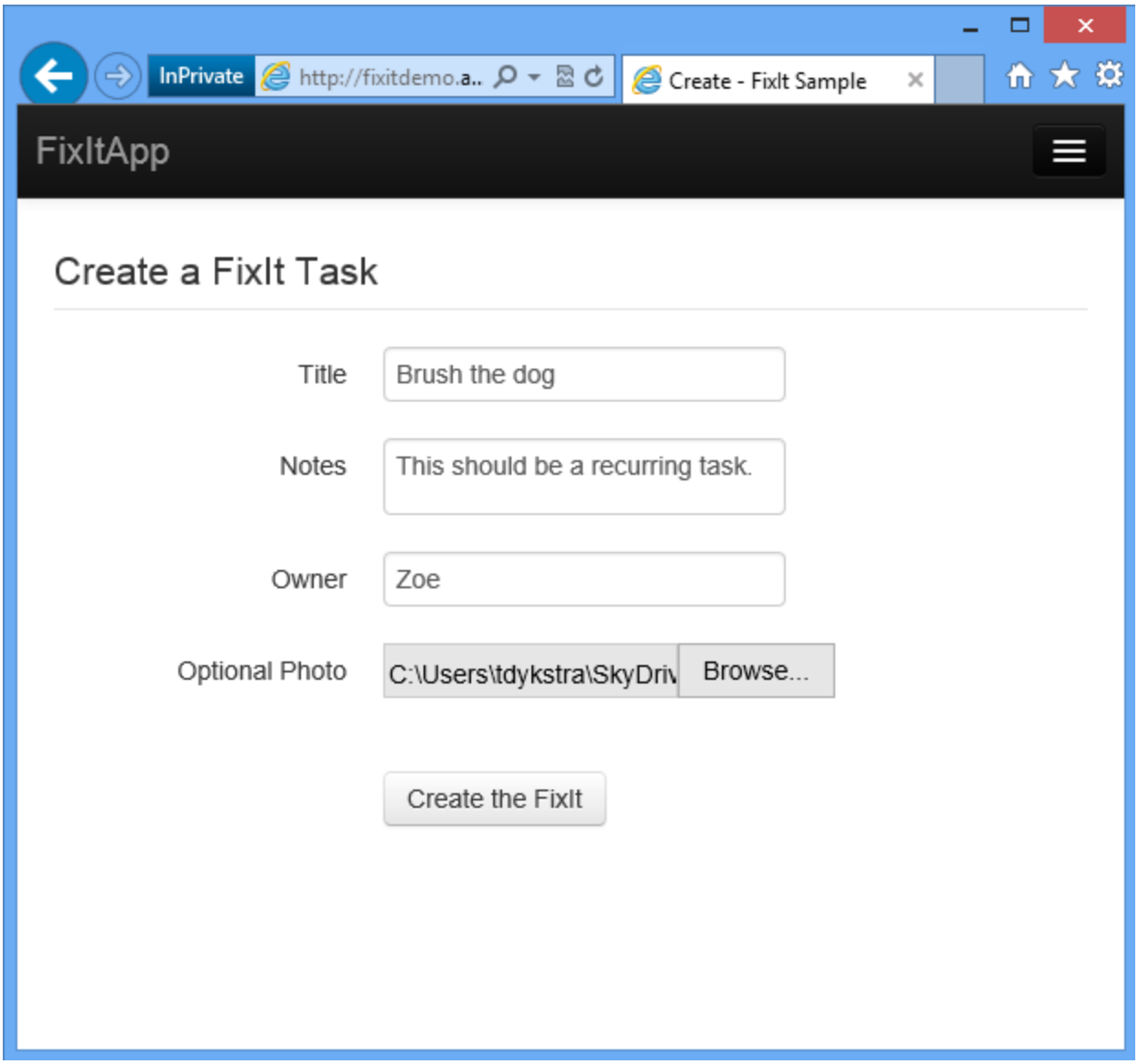


You can log in using forms authentication and a local database or by using a social provider such as Google. (Later we'll also show how to log in with an Active Directory organizational account.)

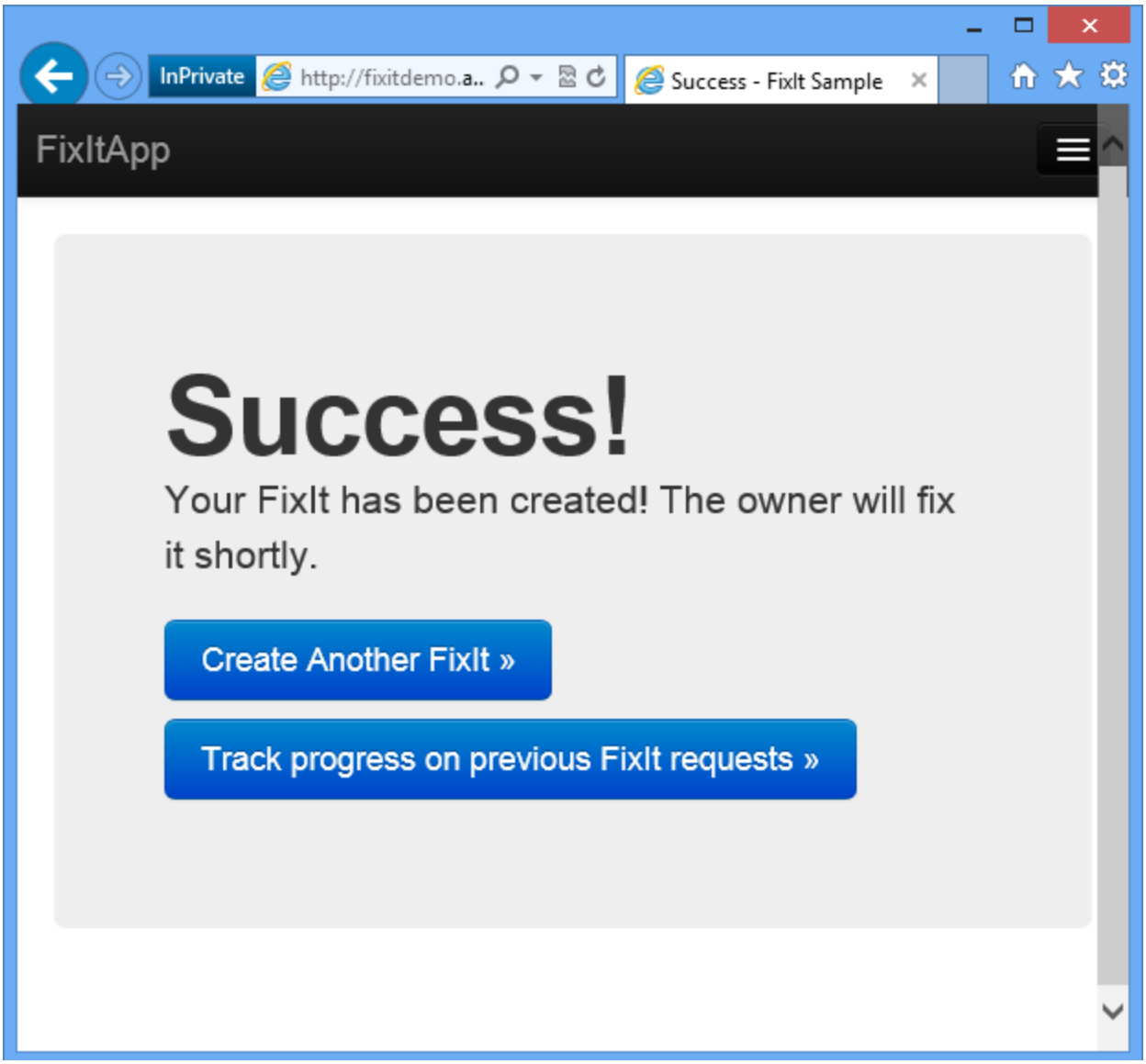
www.dbooks.org



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.



www.dbooks.org



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

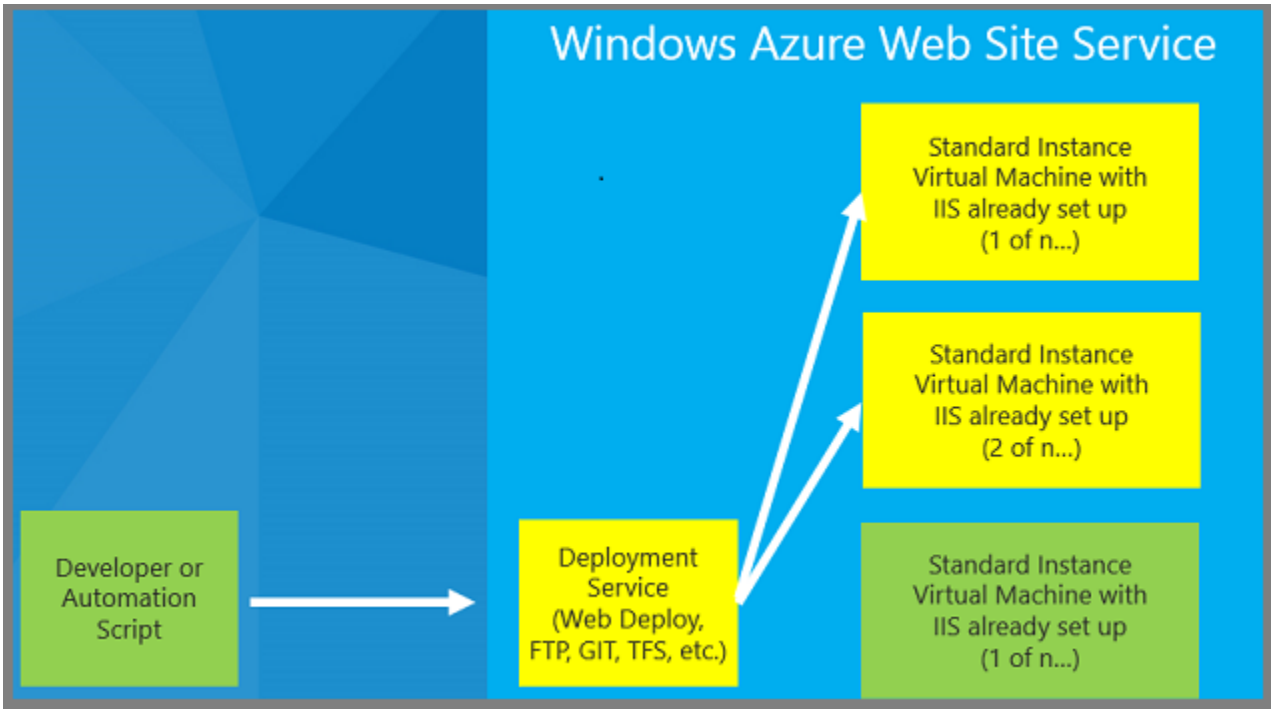
This is a very simple app from a feature perspective, but you'll see how to build it so that it can scale to millions of users and will be resilient to things like database failures and connection terminations. You'll also see how to create an automated and agile development workflow, which enables you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

Windows Azure Web Sites

The cloud environment used for the Fix It application is a service of Windows Azure that we call Web Sites. This service is a way that you can host your own web app in Windows Azure without having to create VMs and keep them updated, install and configure IIS, etc. We host your site on

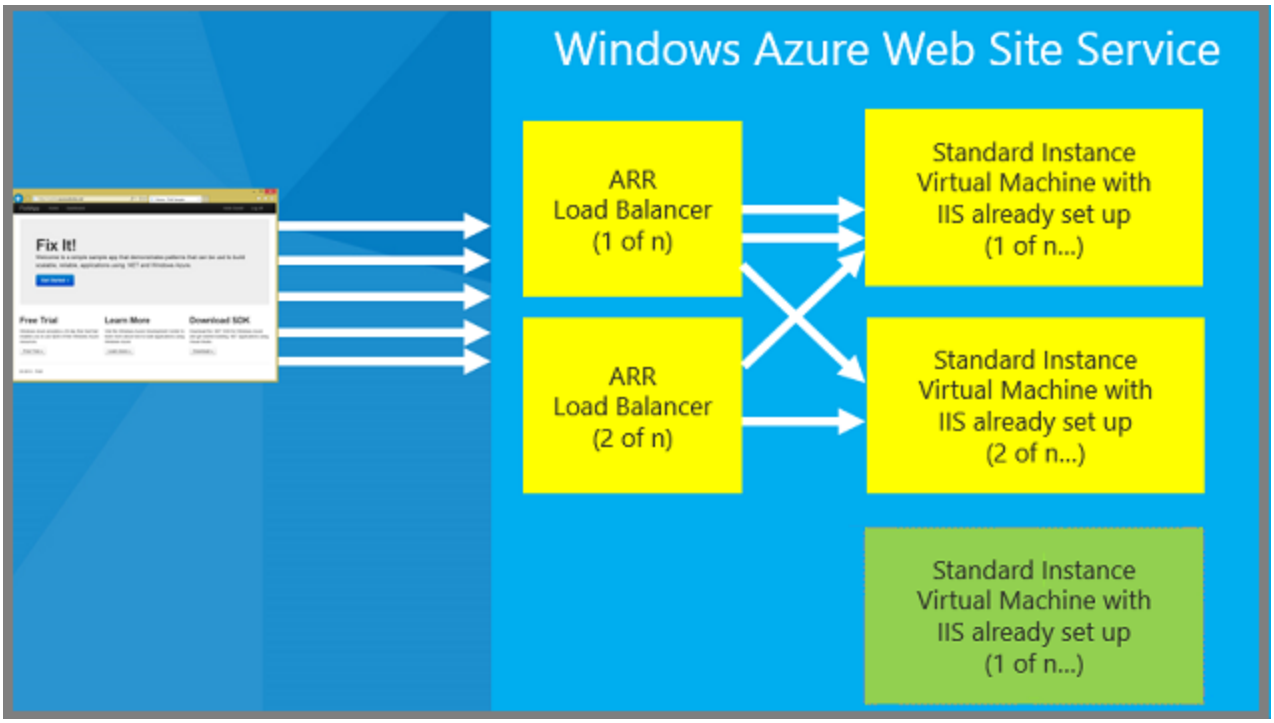
our VMs and automatically provide backup and recovery and other services for you. The Web Sites service works with ASP.NET, Node.js, PHP, and Python. It enables you to deploy very quickly using Visual Studio, Web Deploy, FTP, Git, or TFS. It's usually just a few seconds between the time you start a deployment 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 Windows Azure Web Sites service provides a lot of architectural components and features that you'd have to build yourself if you were going to host a web site 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.

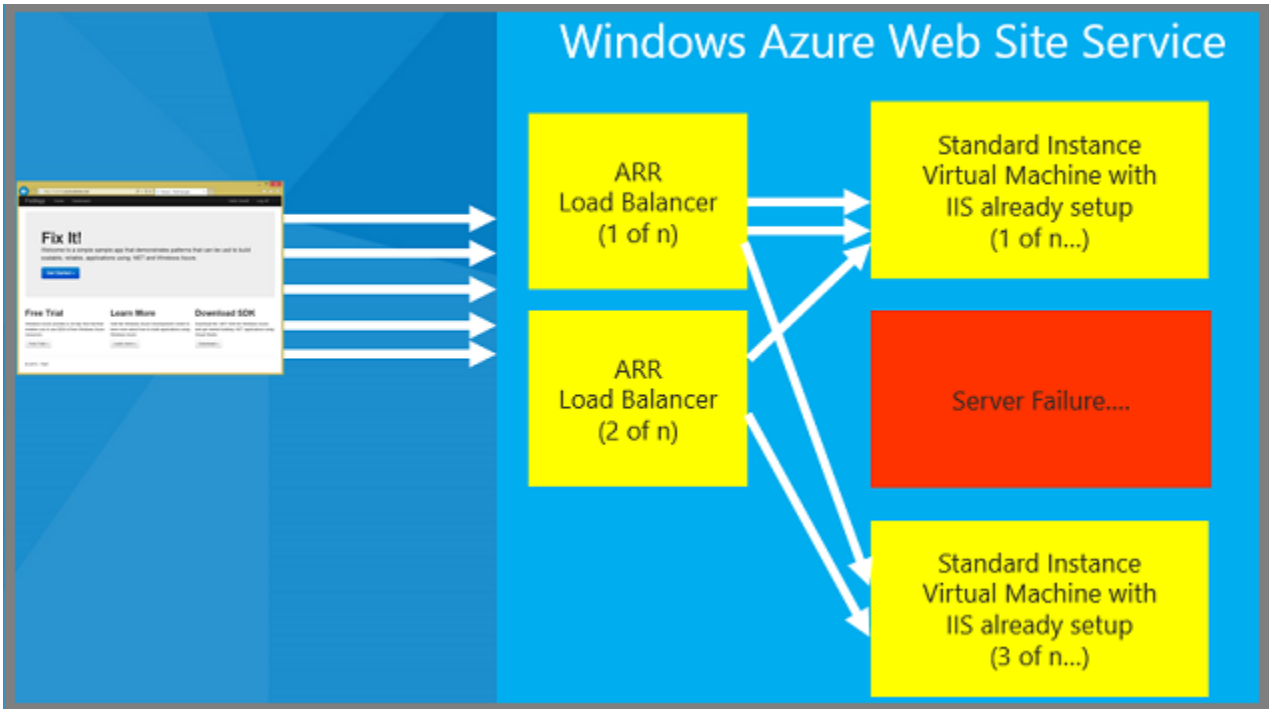


When a user hits the web site, they don't hit the IIS VMs directly, they go through Application Request Routing (ARR) load balancers. You can use these with your own servers, but the advantage here is that they're set up for you automatically. They use a smart heuristic that takes into account factors such as session affinity, queue depth in IIS, and CPU usage on each machine to direct traffic to the VMs that host your web site.

www.dbooks.org



If a machine goes down, Windows 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 web site and deploy your application to it, using Windows PowerShell, Visual Studio, or the Windows 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 a Windows Azure Web Site, see Get started with Windows Azure and ASP.NET.

Summary

This introduction has provided a list of topics the book will cover, screenshots of the sample application, and a brief overview of the Windows Azure Web Sites cloud environment. One of the great advantages of developing apps in and for the cloud is that it's easy to automate repetitive development tasks such as creating a test environment and deploying your code to it. How to do that is the subject of the next chapter.

Resources

For more information about the topics covered in this chapter, see the following resources.

Documentation:

- Windows Azure Web Sites. Portal page for WindowsAzure.com documentation about Windows Azure Web Sites (WAWS).
- Windows Azure Web Sites, Cloud Services, and VMs: When to use which? WAWS as shown in this chapter is just one of three ways you can run web apps in Windows Azure. This article explains the differences between the three ways and gives guidance on how to choose which one is right for your scenario. Like Web Sites, Cloud Services is a PaaS feature of Windows Azure. VMs are an IaaS feature. For an explanation of PaaS versus IaaS, see the Data Options chapter.

Videos:

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

www.dbooks.org

Automate Everything

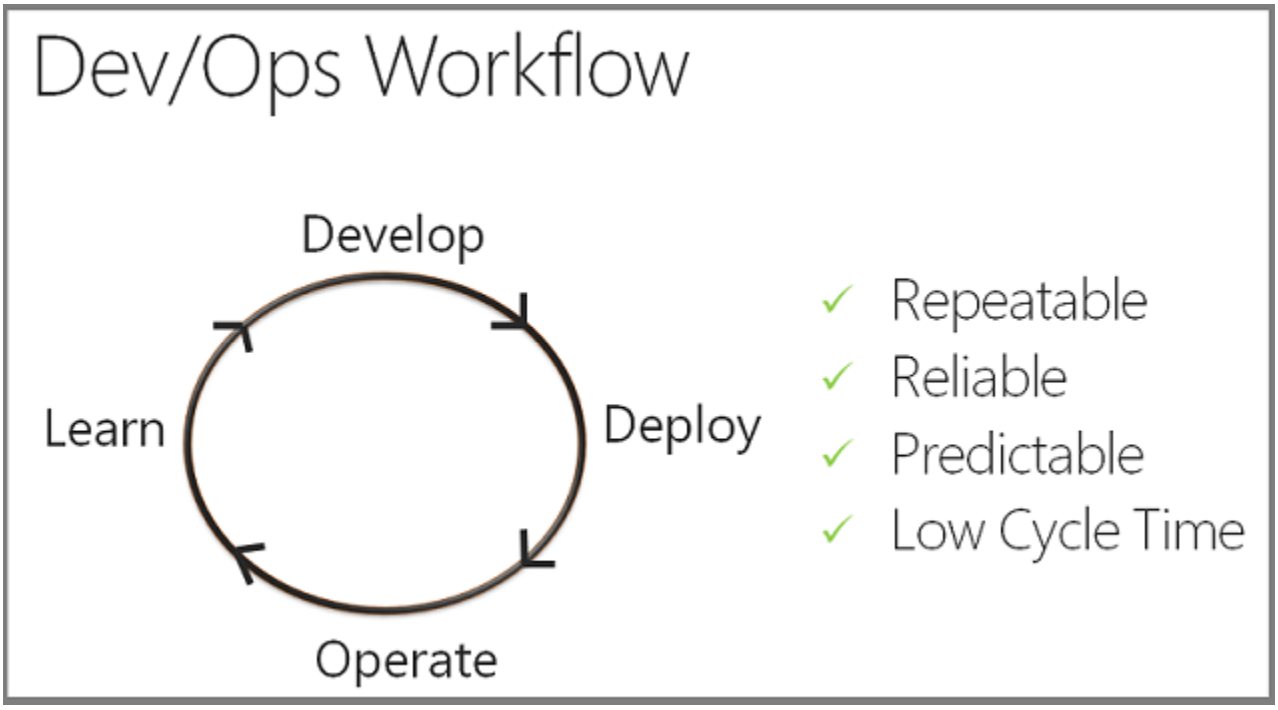
The first three patterns we'll look at actually apply to any software development project, but especially to cloud projects. This pattern is about automating development tasks. It's an important topic because manual processes are slow and error-prone; automating as many of them as possible helps set 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 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 in order 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've learned, and repeat the cycle quickly and reliably.

Some successful cloud development teams deploy multiple times a day to a live environment. The Windows Azure team used to deploy a major update every 2-3 months, but now it releases minor updates every 2-3 days and major releases every 2-3 weeks. Getting into that cadence really helps you be responsive to customer feedback.

In order 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 the 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 in order to enable that kind of process.

Windows Azure management scripts

In the introduction to this e-book, you saw the web-based console, the Windows Azure Management Portal. The management portal enables you to monitor and manage all of the resources that you have deployed on Windows Azure. It's an easy way to create and delete services such as web sites 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 in order to learn and explore Windows 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. Windows Azure has scripting APIs for all those different environments, and it has a .NET management API in case you want to write code instead of script.

For the Fix It app we've 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.

Environment creation script

The first script we'll look at is named *New-AzureWebsiteEnv.ps1*. It creates a Windows 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 web site.
- Create a storage account. (Required for blobs and queues, as you'll see in later chapters.)
- Create a SQL Database server and two databases: an application database, and a membership database.
- Store settings in Windows 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 in order to run them. This a demo and doesn't provide everything you need to know in order to run

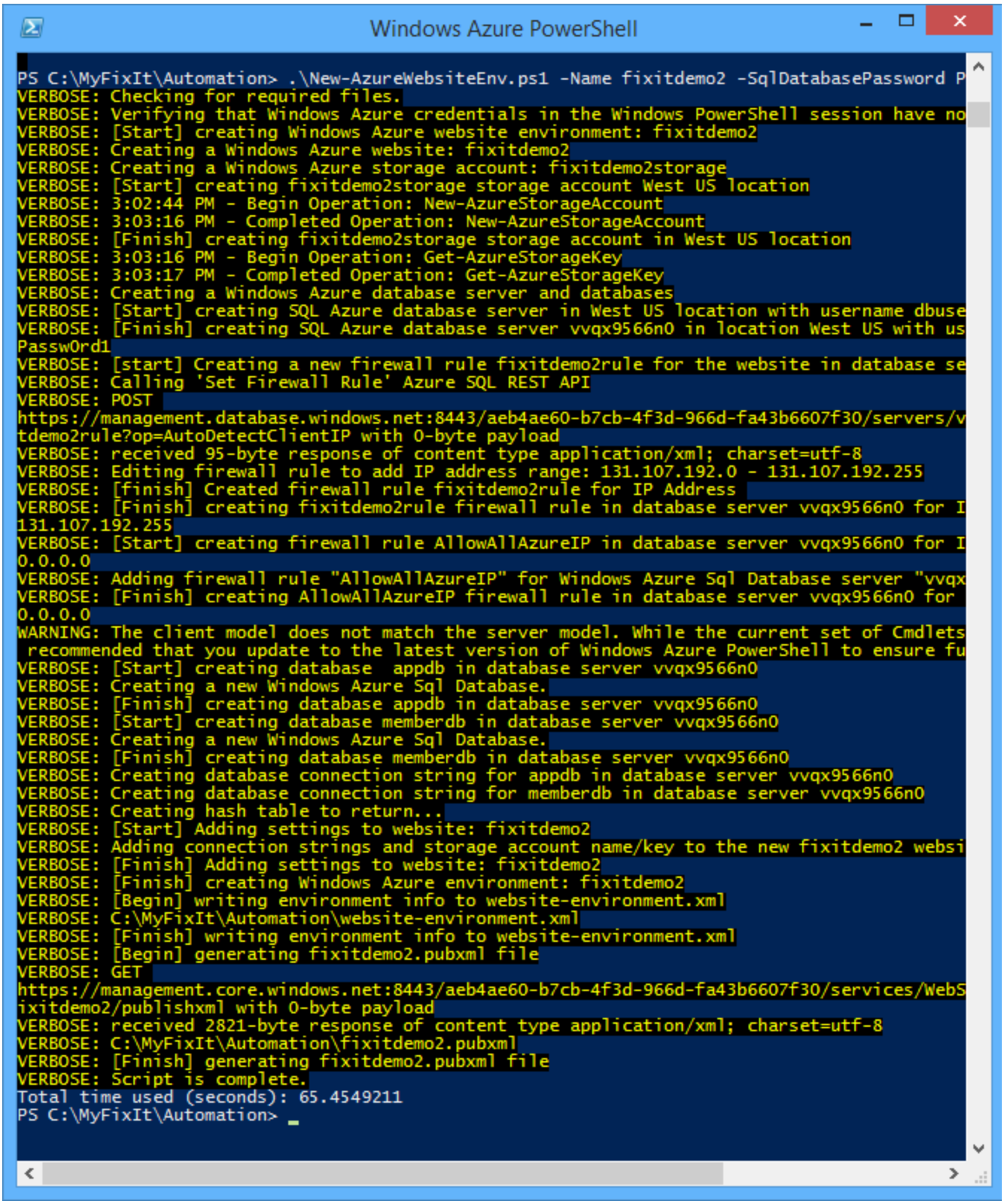
www.dbooks.org

the scripts. For step-by-step how-to-do-it instructions, see Appendix: The Fix It Sample Application.

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

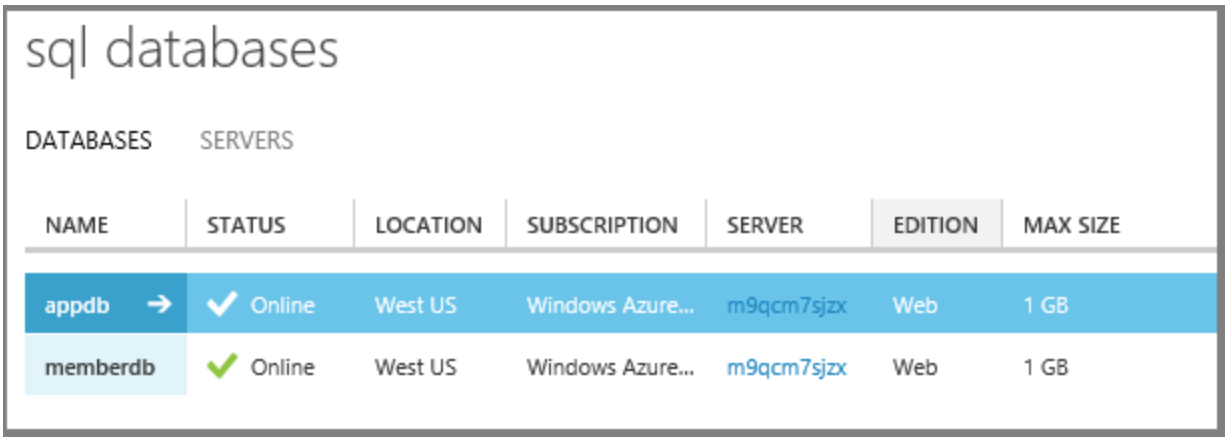
```
.\New-AzureWebsiteEnv.ps1 -Name <websitename> -SqlDatabasePassword <password>
```

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 SQL Database. There are other parameters you can use that we'll look at later.

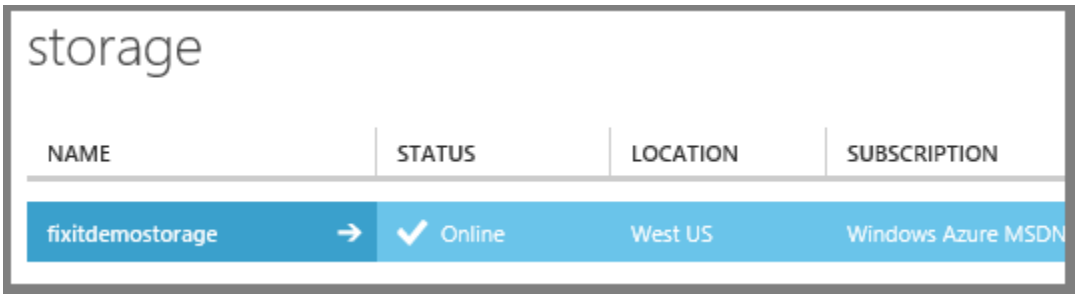


After the script finishes you can see in the management portal what was created. You'll find two databases:

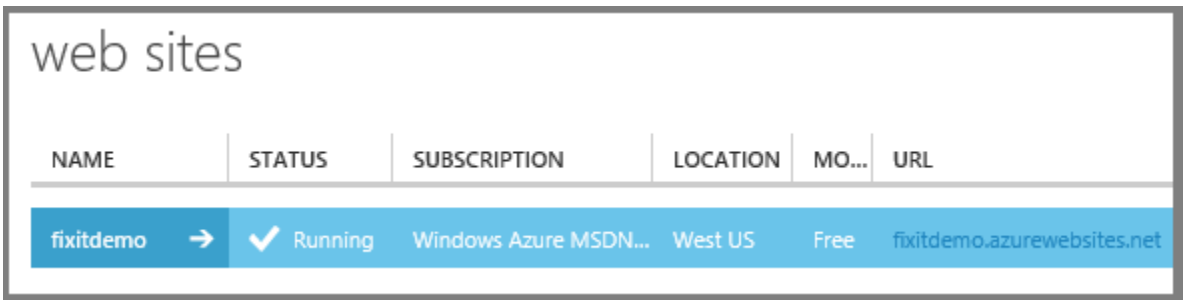
www.dbooks.org



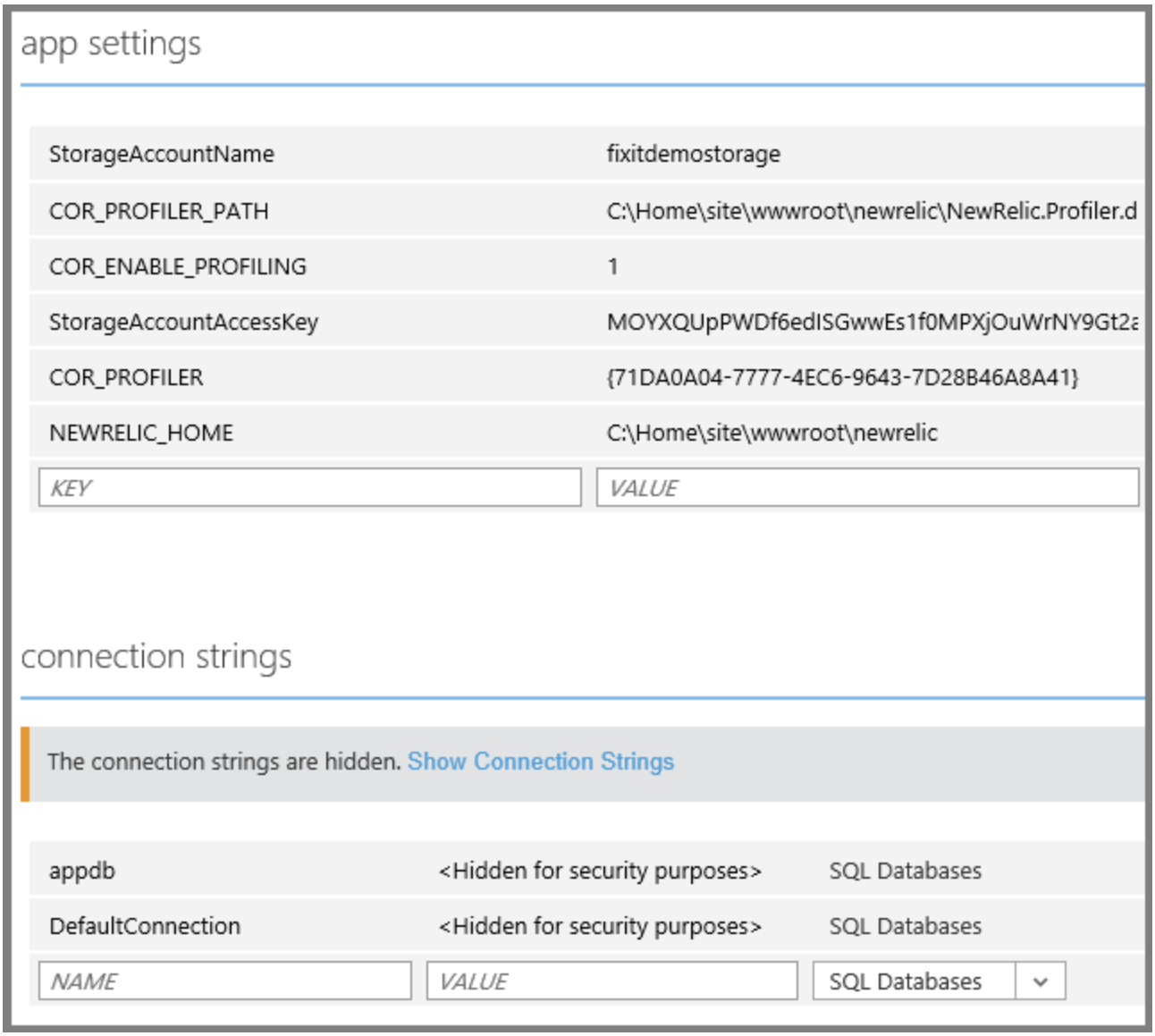
A storage account:



And a web site:



On the **Configure** tab for the web site, 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 Windows 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</SiteUrlToLau
nchAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <MSDeployServiceURL>waws-prod-bay-
003.publish.azurewebsites.windows.net:443</MSDeployServiceURL>
    <DeployIisAppPath>fixitdemo</DeployIisAppPath>
```


www.dbooks.org

```
    <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 also 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 web site 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 web site

The first thing the script does is create the web site by calling the `New-AzureWebsite` cmdlet, passing in to it the web site 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 web site.

```
$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 in order 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:

www.dbooks.org

```
$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 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 web site 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 whitelist all the Azure IP 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 Windows 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} `
}
```


www.dbooks.org

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 a Windows Azure Web Site.

Store app settings and connection strings

Windows Azure has a feature that enables you to store settings and connection strings that automatically override what is returned to the application when it tries to read 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 Windows Azure later in this e-book.

The environment creation script stores in Windows Azure all of the `appSettings` and `connectionStrings` values that the application needs to access the storage account and databases when it runs in Windows 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 the Monitoring and Telemetry chapter. The environment creation script also restarts the web site 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 Windows 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 non-terminating 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

www.dbooks.org

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 web site 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
```

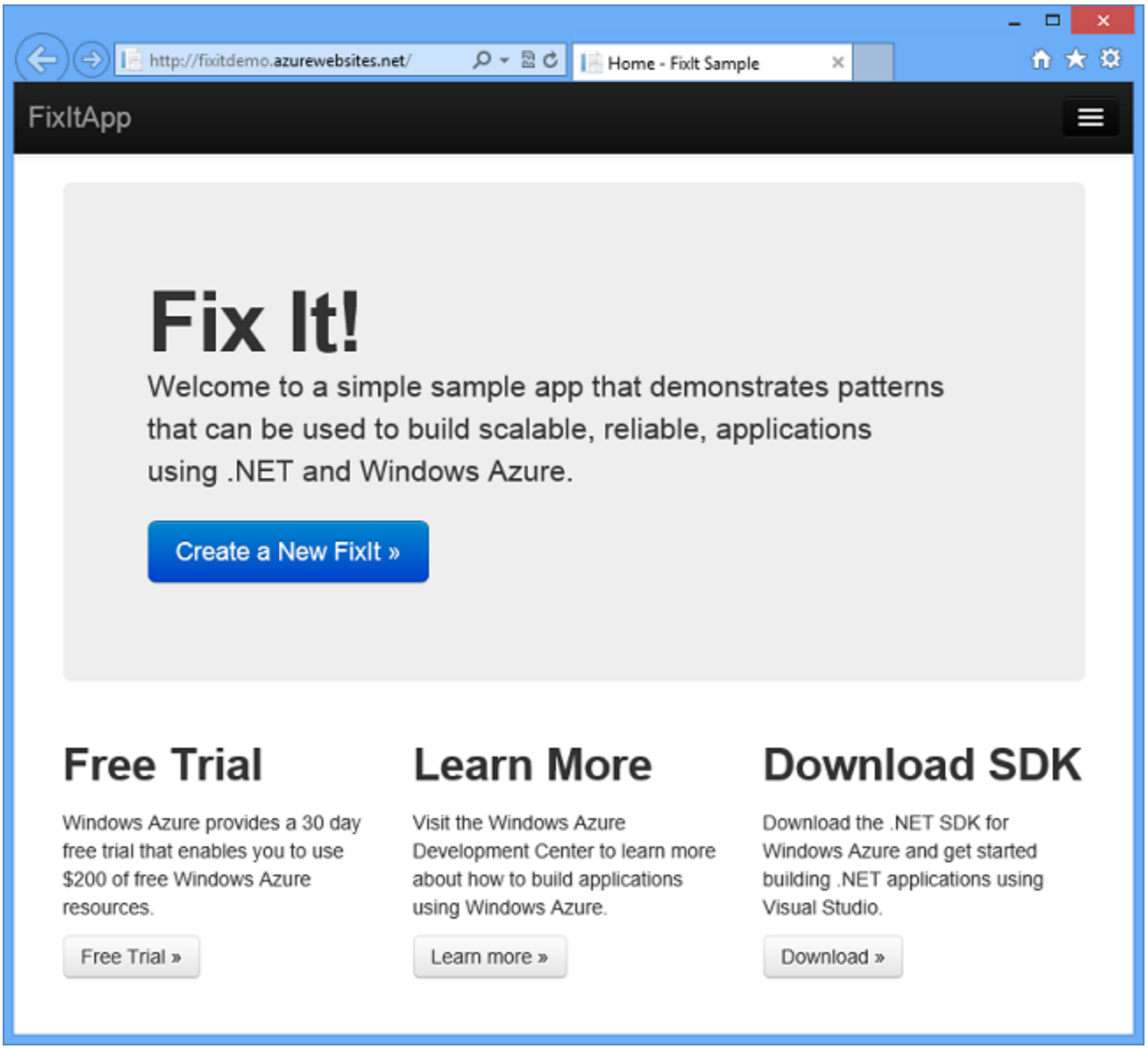
And if you've specified the `Launch` parameter on the command line, it calls the `Show-AzureWebsite` cmdlet to open your default browser to the website URL.

```
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 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 Windows Azure management functions that you can do 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.

www.dbooks.org

Resources

- Install and Configure Windows PowerShell for Windows Azure. Explains how to install the Windows Azure PowerShell cmdlets and how to install the certificate that you need on your computer in order to manage your Windows Azure account. This is a great place to get started because it also has links to resources for learning PowerShell itself.
- Windows Azure Script Center. WindowsAzure.com portal to resources for developing scripts that manage Windows 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 Windows Azure management functions.
- Install and Configure the Windows Azure Cross-Platform Command-Line Interface. Getting-started tutorial for a Windows Azure scripting framework that works on Mac and Linux as well as Windows systems.
- Windows Azure Command Line Tools. WindowsAzure.com portal page for documentation and downloads related to command line tools for Windows Azure.
- Automating everything with the Windows Azure Management Libraries and .NET. Scott Hanselman introduces the .NET management API for Windows Azure.

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 set-up, and you can use Visual Studio Online source control free for up to 5 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) into 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, Windows Azure, and Visual Studio Online:

- Add scripts to source control in Visual Studio
- Store sensitive data in Windows 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're changing 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 the Automate Everything pattern. All of the scripts that you use to create your environment, deploy to it, scale it, etc., 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 which 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 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,

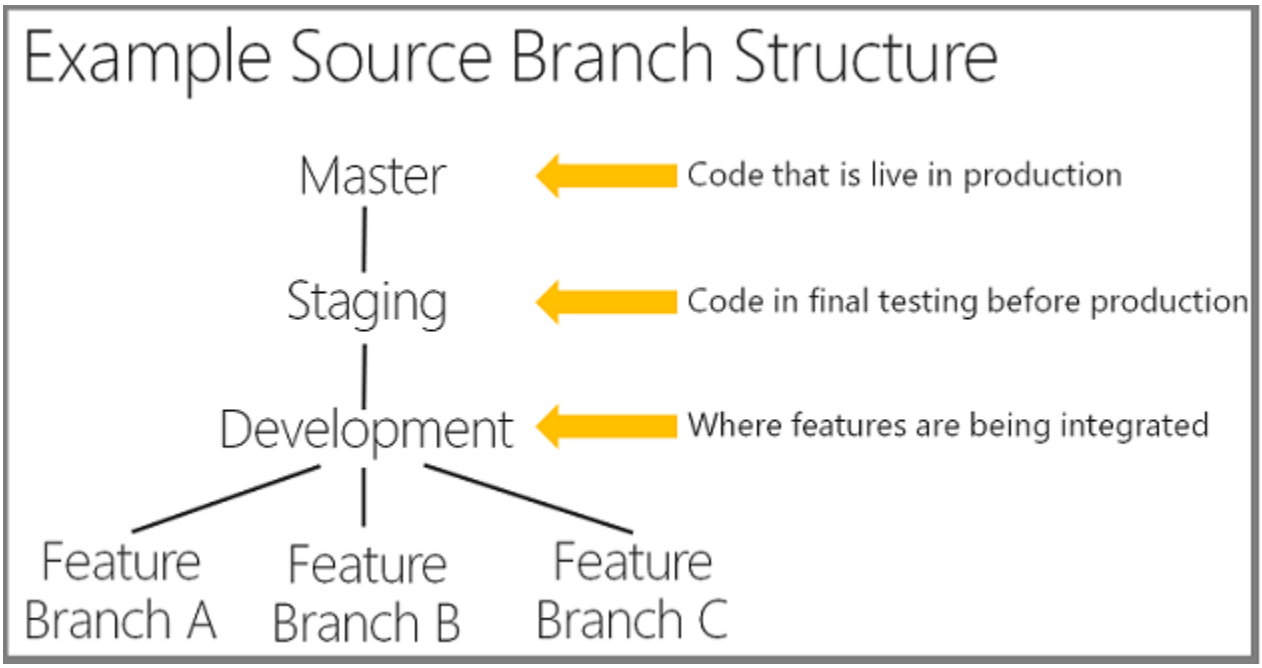
www.dbooks.org

parameterize those settings so that they don't get saved in source code, and store your secrets somewhere else.

For example, Windows 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 Windows 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 just have 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.

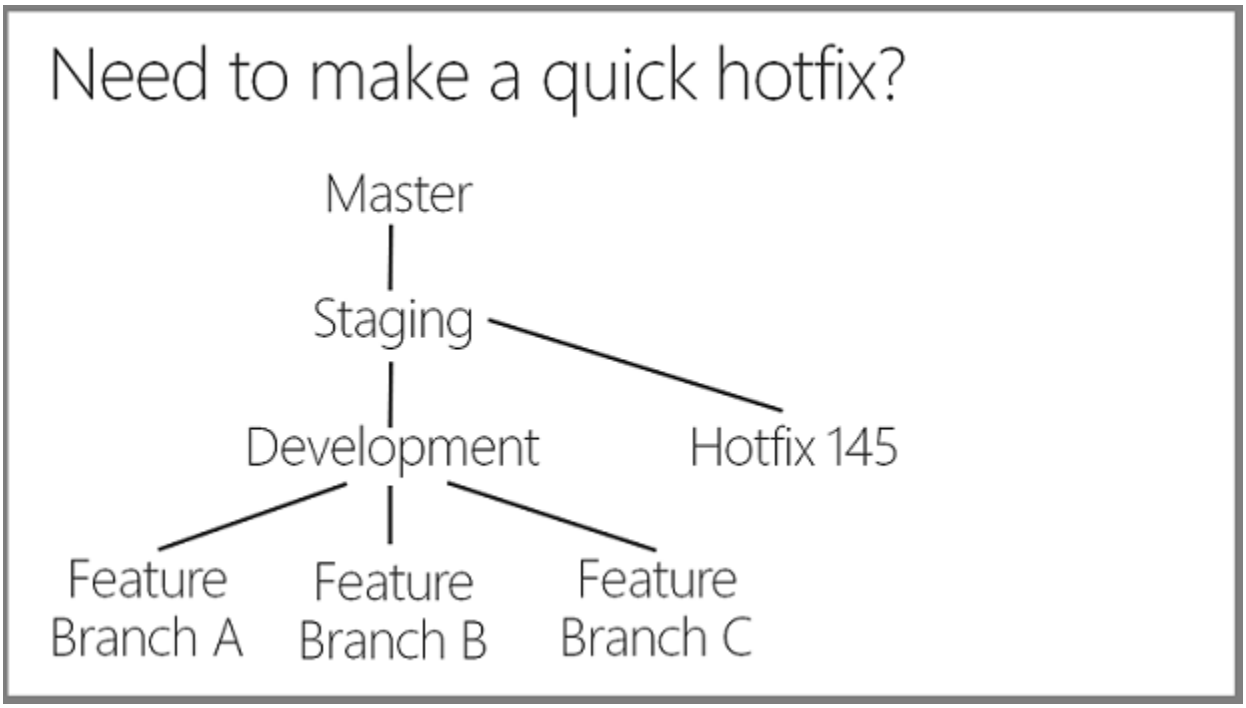
For big teams there may be separate branches for each new feature; for a smaller team you might have everyone checking in to the development branch.

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 only display the new field when a variable is set to true, and it will only use caching when a different variable is set to true. If Feature A isn't ready to be promoted but the 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 it's ready merge it up into master and down into development and 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.

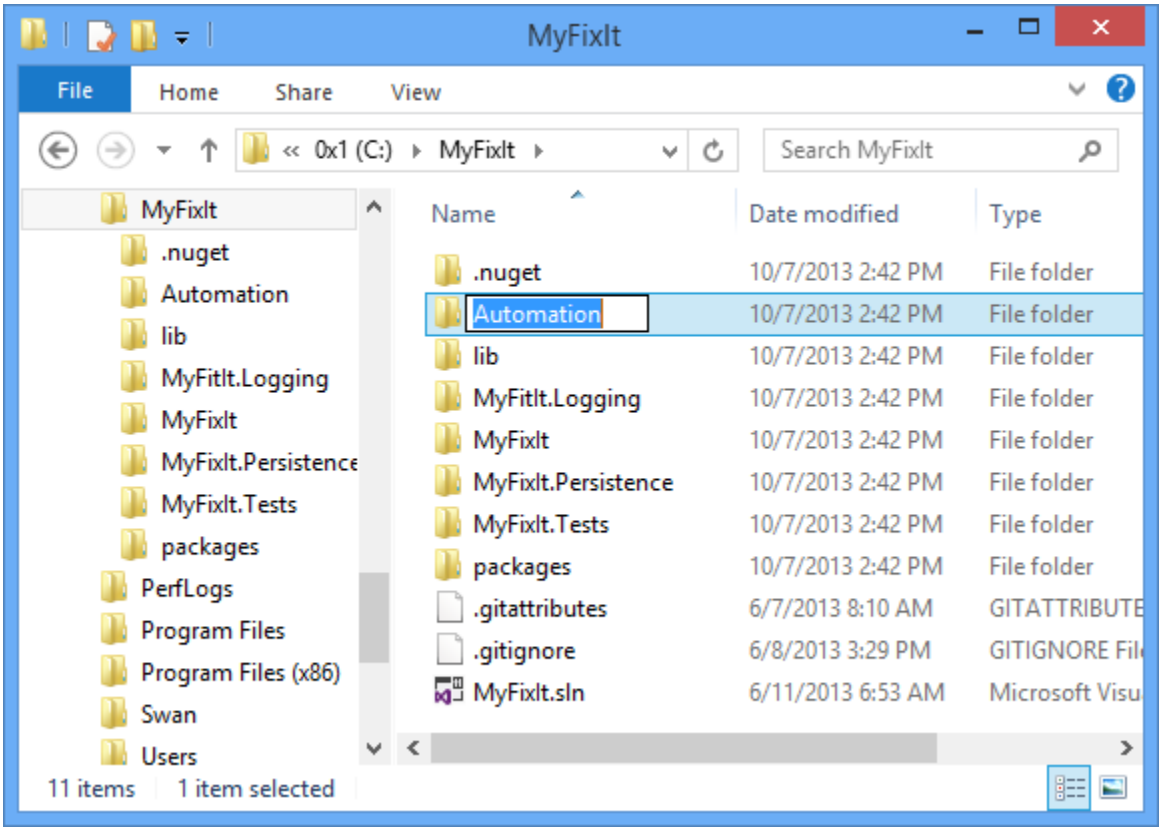
www.dbooks.org

Next you'll see examples of how to implement these three patterns in Visual Studio, Windows 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 the chapter.

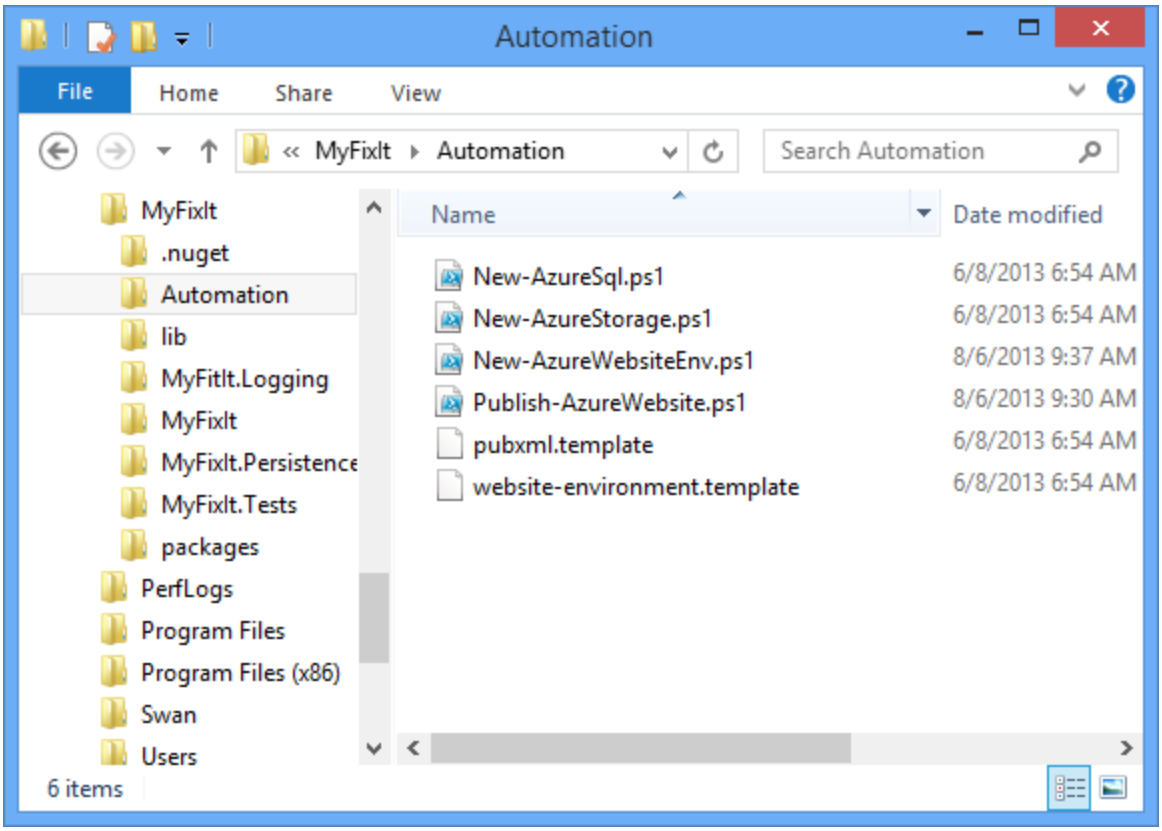
Add scripts to source control in Visual Studio

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).

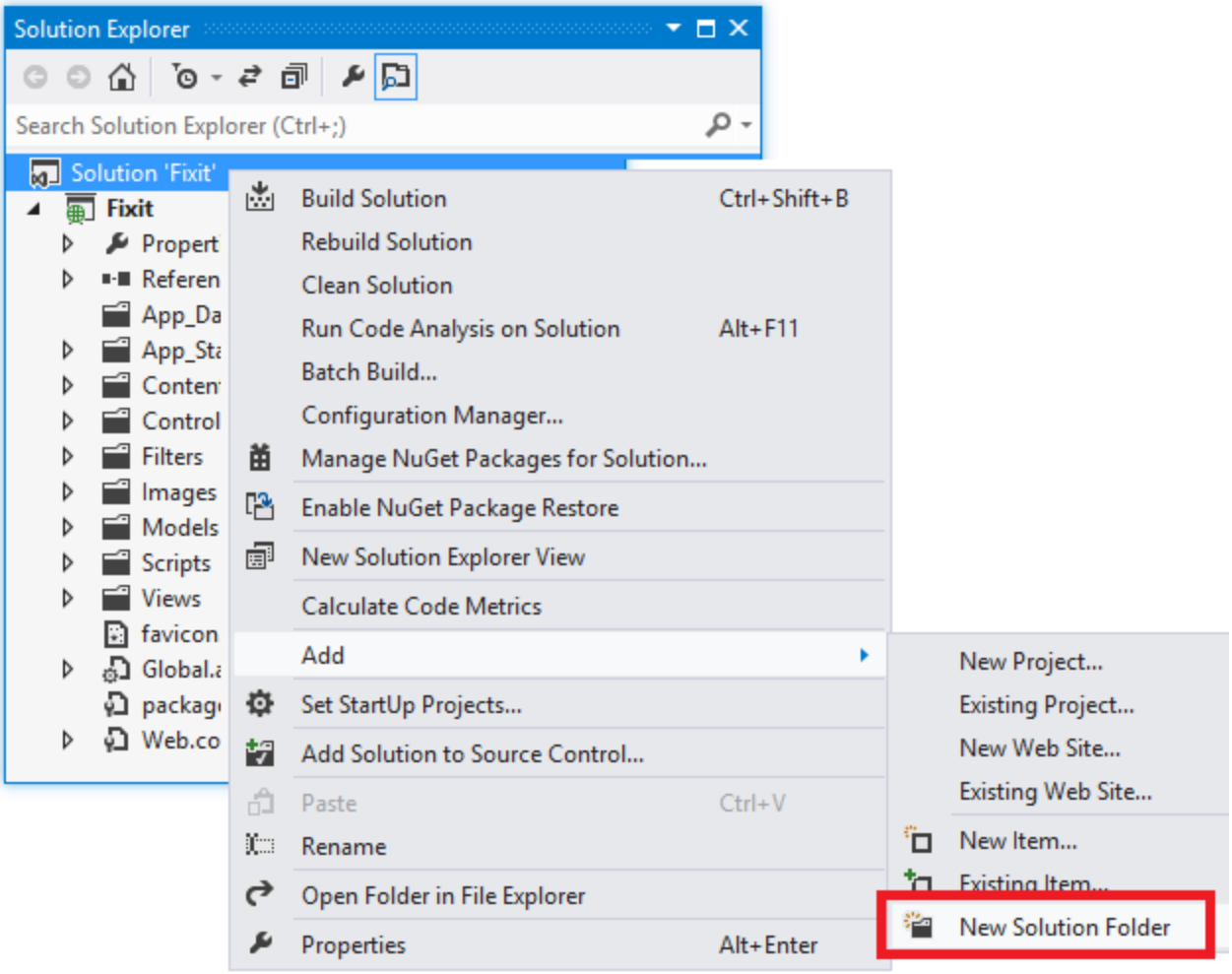


Copy the script files into the folder.

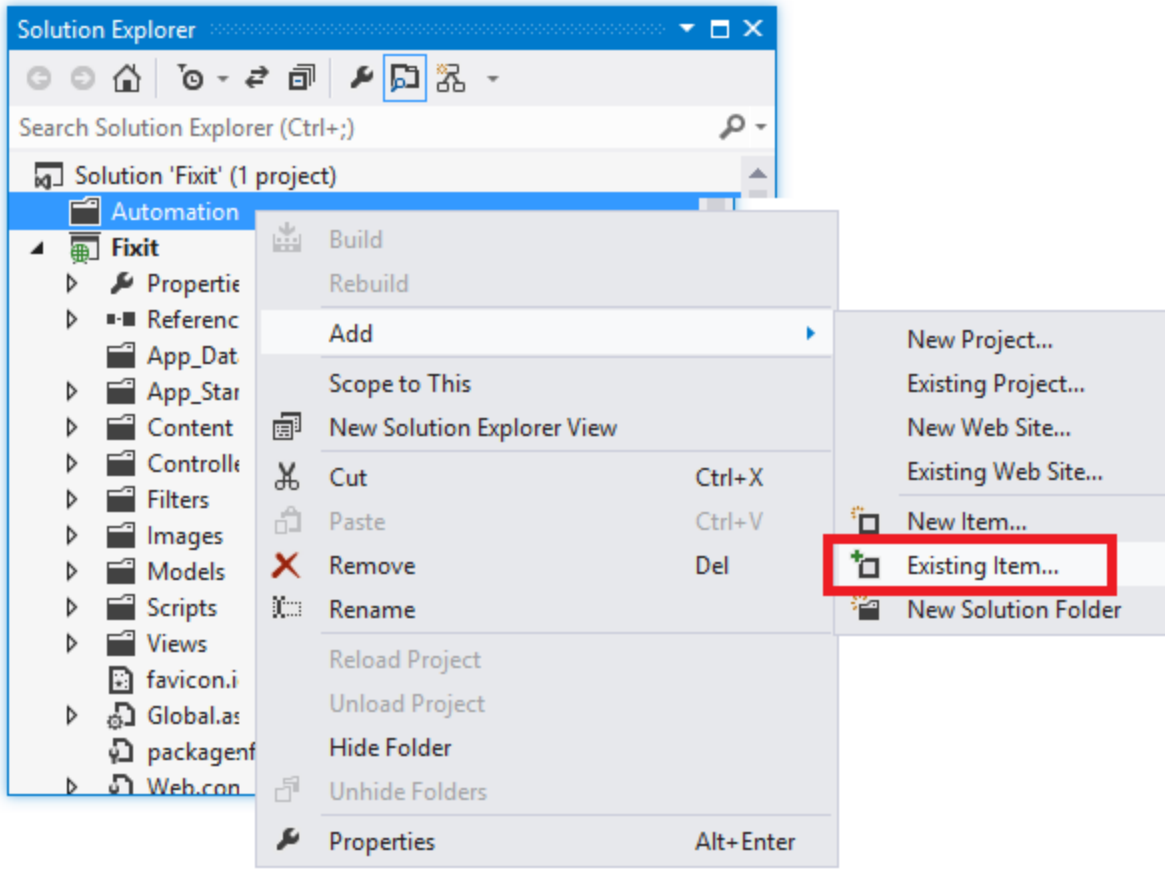


In Visual Studio, add a solution folder to the project.

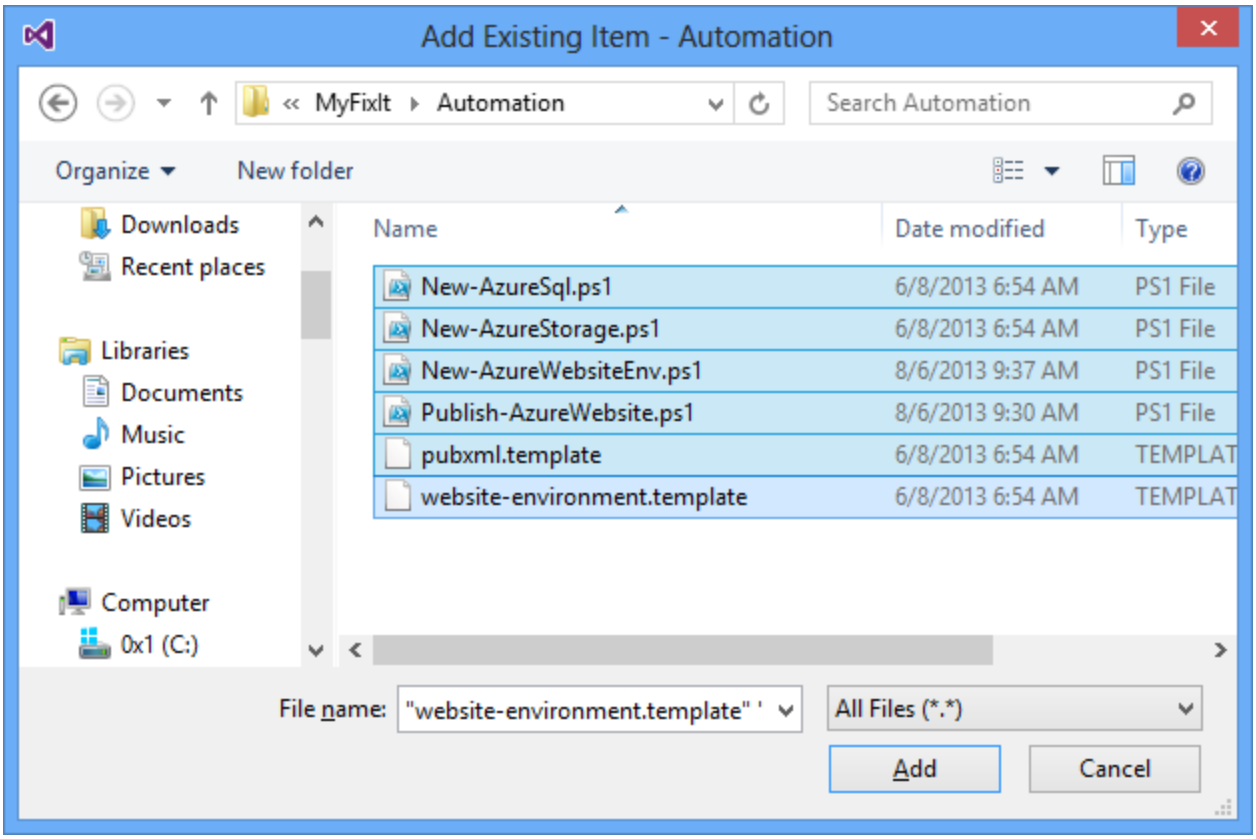
www.dbooks.org



And add the script files to the solution folder.



www.dbooks.org



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 Windows Azure

If you run your application in a Windows Azure Web Site, one way to avoid storing credentials in source control is to store them in Windows 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 your Windows Azure storage account.

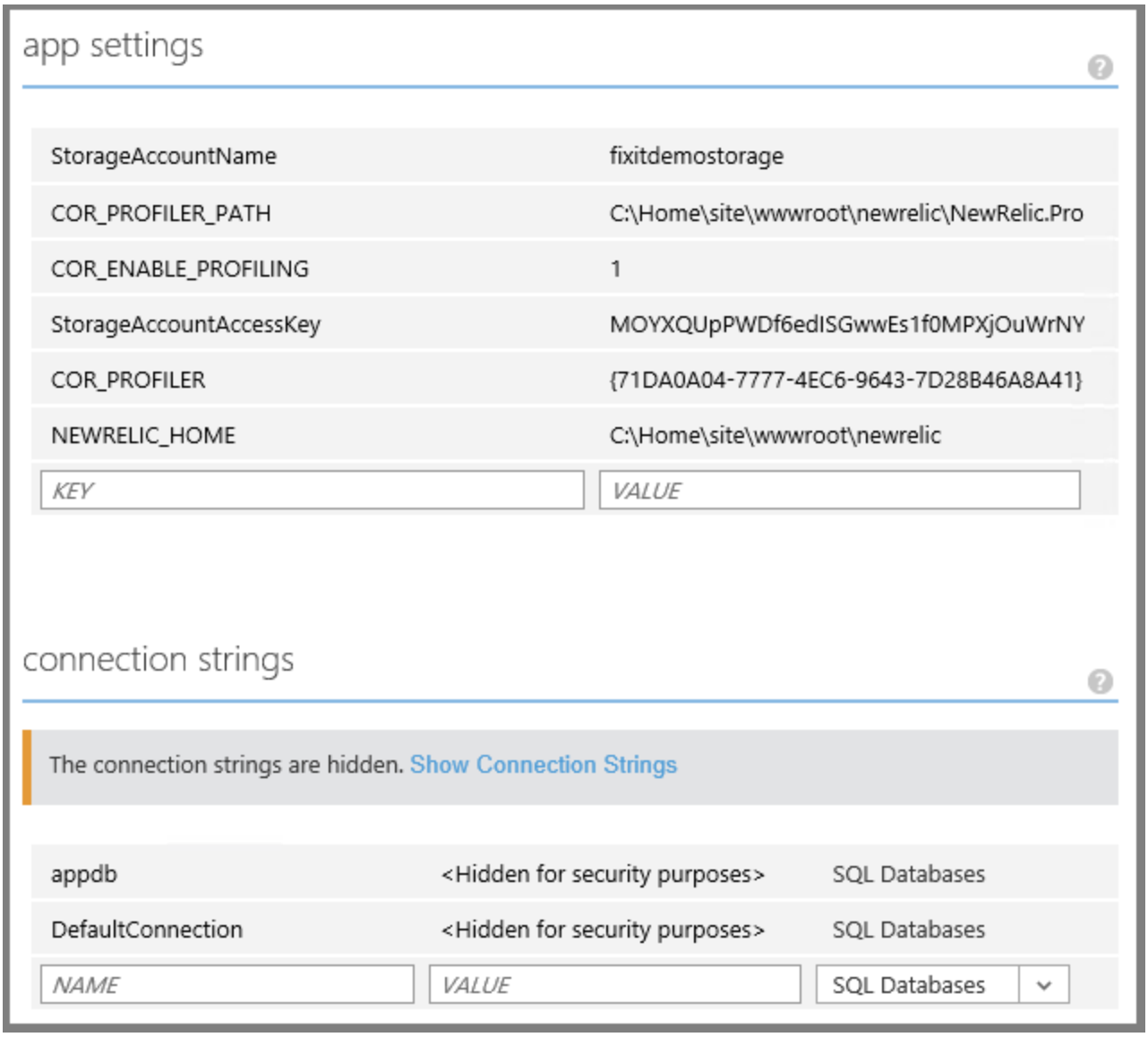
```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\MyFixItMembership.mdf
;Initial Catalog=MyFixItMembership;Integrated Security=True"
providerName="System.Data.SqlClient" />
  <add name="appdb" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\MyFixItTasks.mdf;Init
ial 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.)

Windows 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 web site in the Windows Azure management portal:

www.dbooks.org



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

You can set these values in Windows Azure by using either the management portal or scripts. The environment creation automation script you saw in the Automate Everything chapter creates a Windows Azure SQL Database, gets the storage and SQL Database connection strings, and stores these secrets in the settings for your web site.

```
# 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 Windows 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 Web Site, not what's actually in 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.

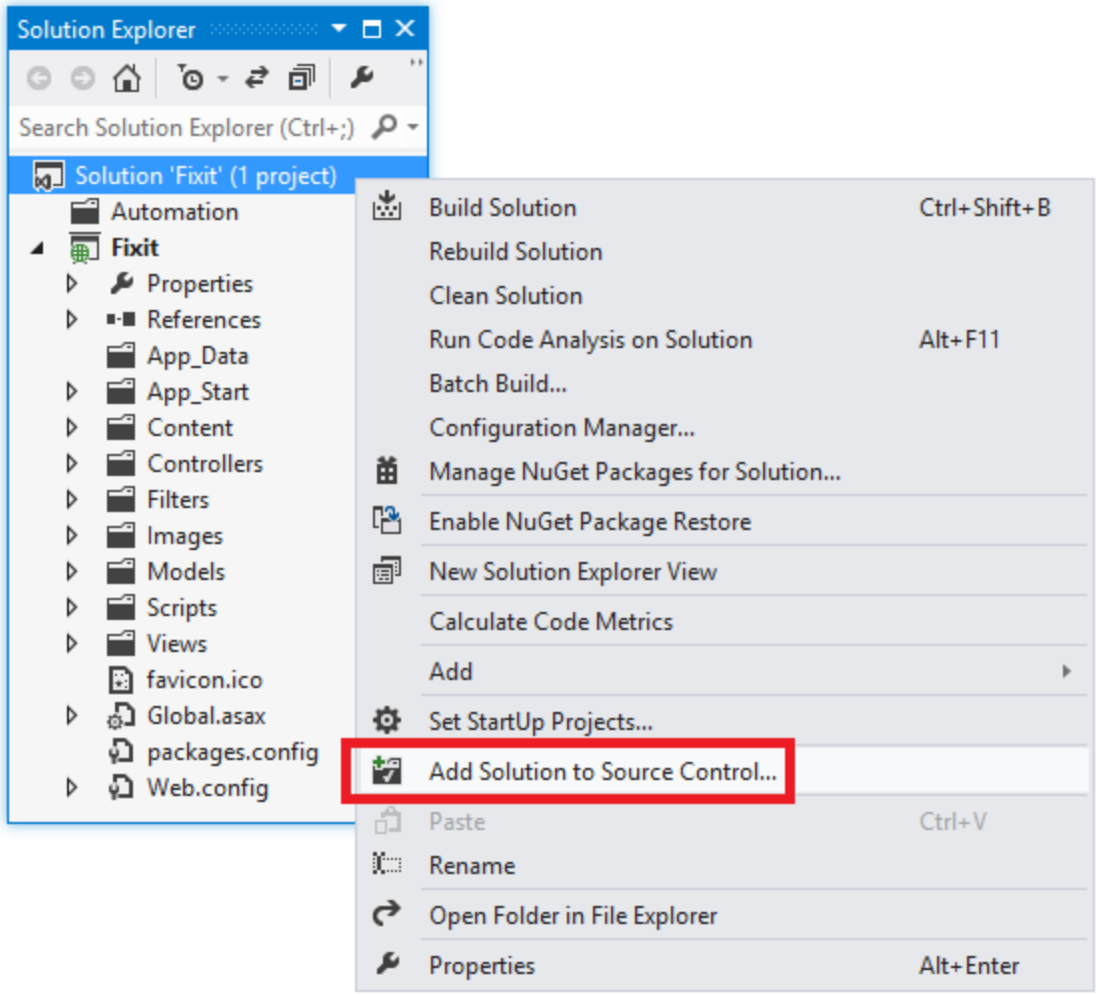
Git is a DVCS that is has become very popular. When you use Git for source control, you have a complete copy of the repository with all of its history on your local computer. Many people prefer that because it's easier to continue working when you're not connected to the network -- you can continue to 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 (VSO), formerly known as Team Foundation Service, offers both Git and Team Foundation Version Control (TFVC; centralized source control). Here at Microsoft in the Windows 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 VSO service is free for up to 5 users. You can sign up for a free plan here.

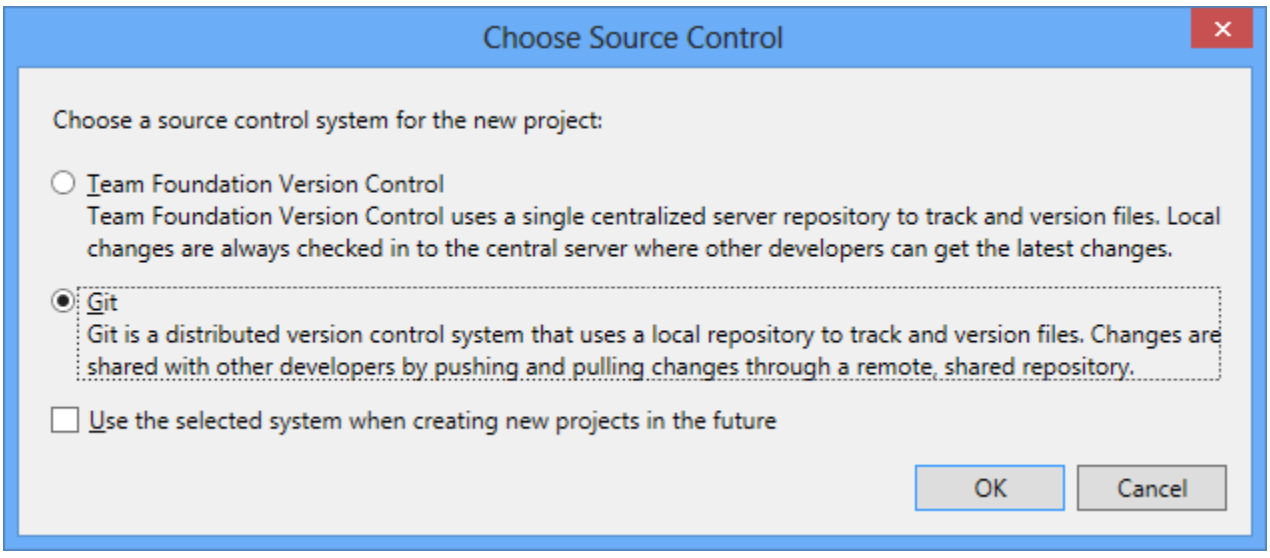
Visual Studio 2013 includes built-in first-class Git support; here's a quick demo of how that works.

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

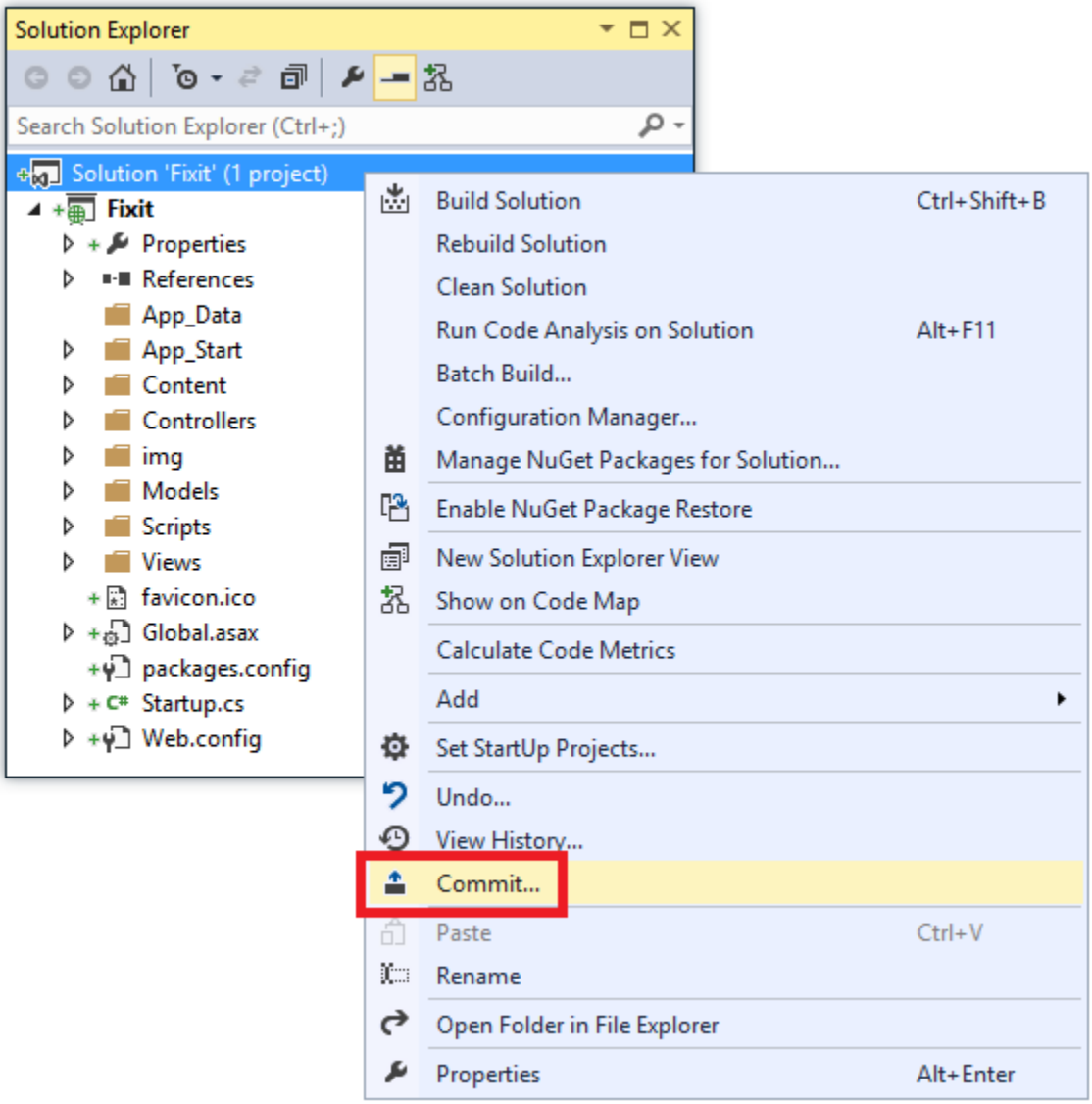
www.dbooks.org



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

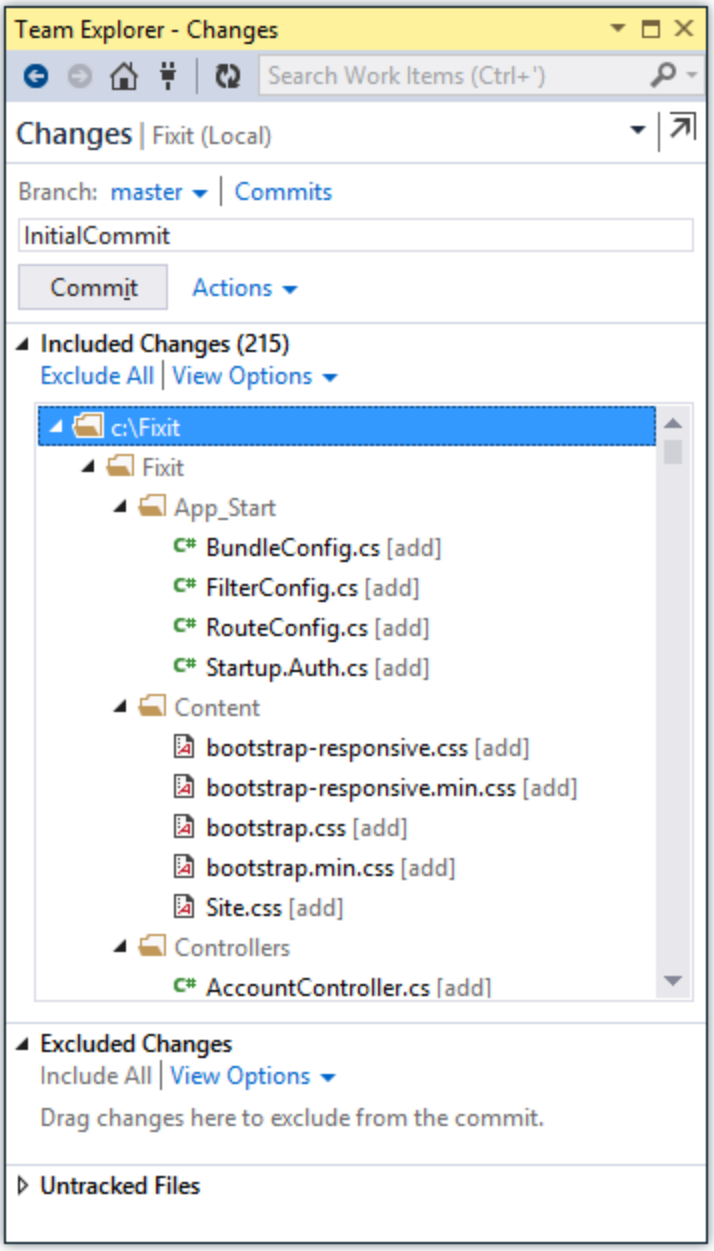


When you select Git and click **OK**, Visual Studio creates a new local Git repository in your solution folder. The new repository has no files yet; you have to add them to the repository by doing a Git commit. Right-click the solution in **Solution Explorer**, and then click **Commit**.

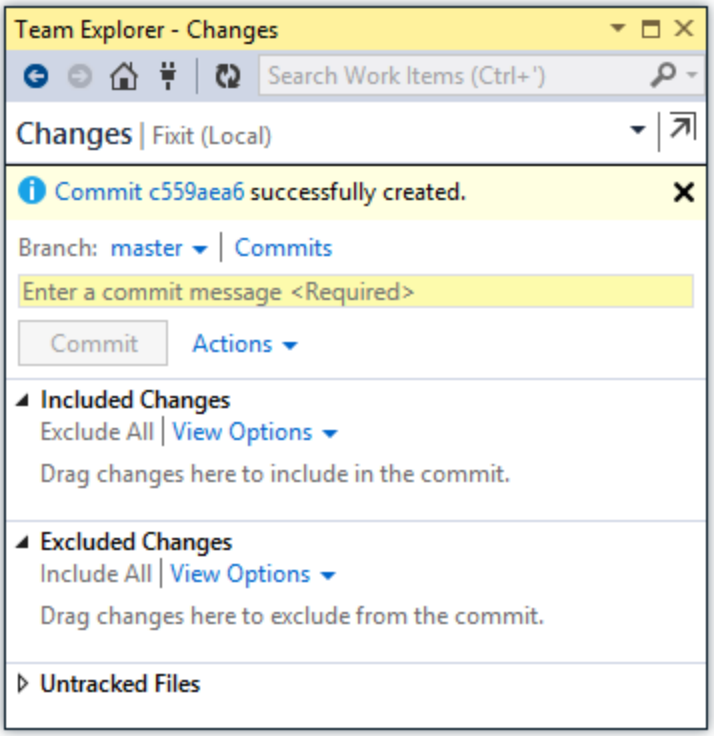


Visual Studio automatically stages all of the project files for the commit and lists them in **Team Explorer** in the **Included Changes** pane. (If there were some you didn't want to include in the commit, you could select them, right-click, and click **Exclude**.)

www.dbooks.org

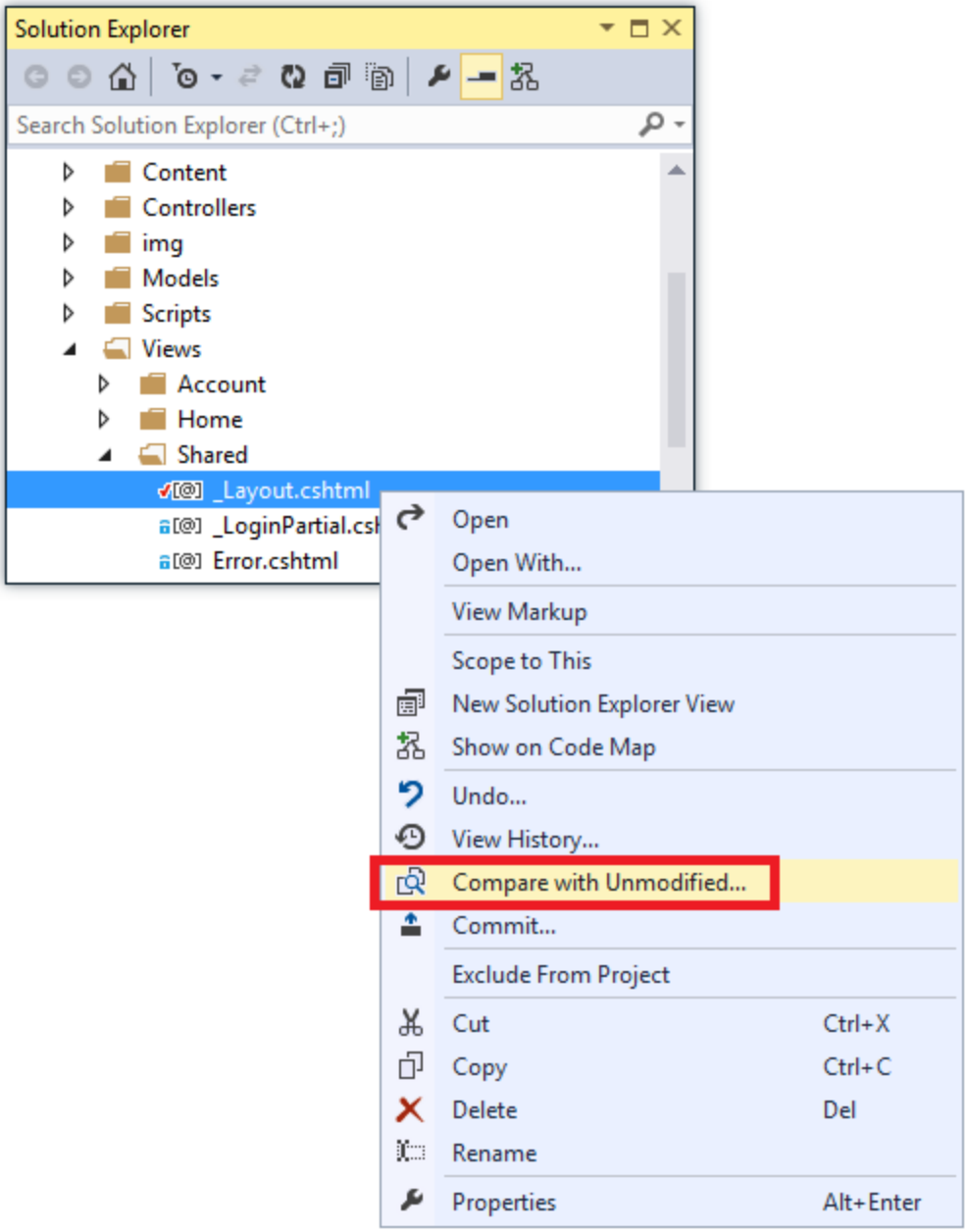


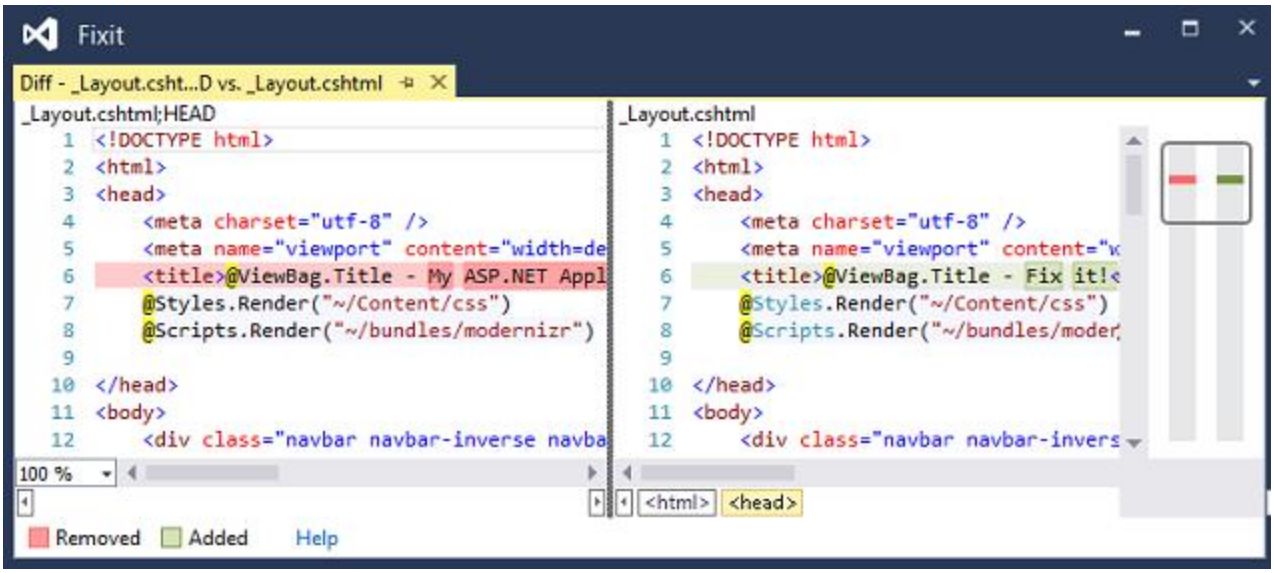
Enter a commit comment and click **Commit**, and Visual Studio executes the commit and displays the commit ID.



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.

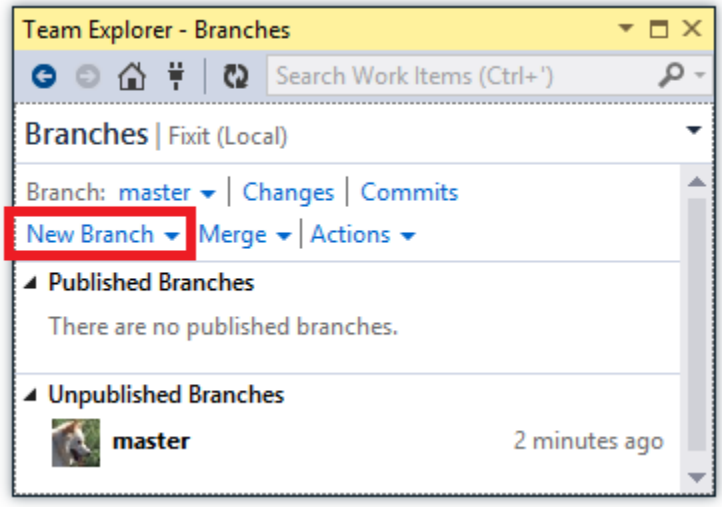
www.dbooks.org


Solution Explorer
Search Solution Explorer (Ctrl+;)
Content
Controllers
img
Models
Scripts
Views
Account
Home
Shared
_Layout.cshtml
_LoginPartial.csh
Error.cshtml
Open
Open With...
View Markup
Scope to This
New Solution Explorer View
Show on Code Map
Undo...
View History...
Compare with Unmodified...
Commit...
Exclude From Project
Cut Ctrl+X
Copy Ctrl+C
Delete Del
Rename
Properties Alt+Enter



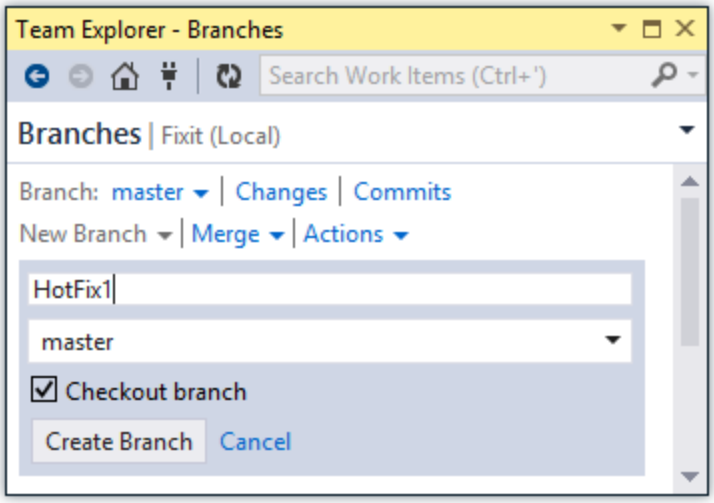
You can easily see what changes you're making and check them in.

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

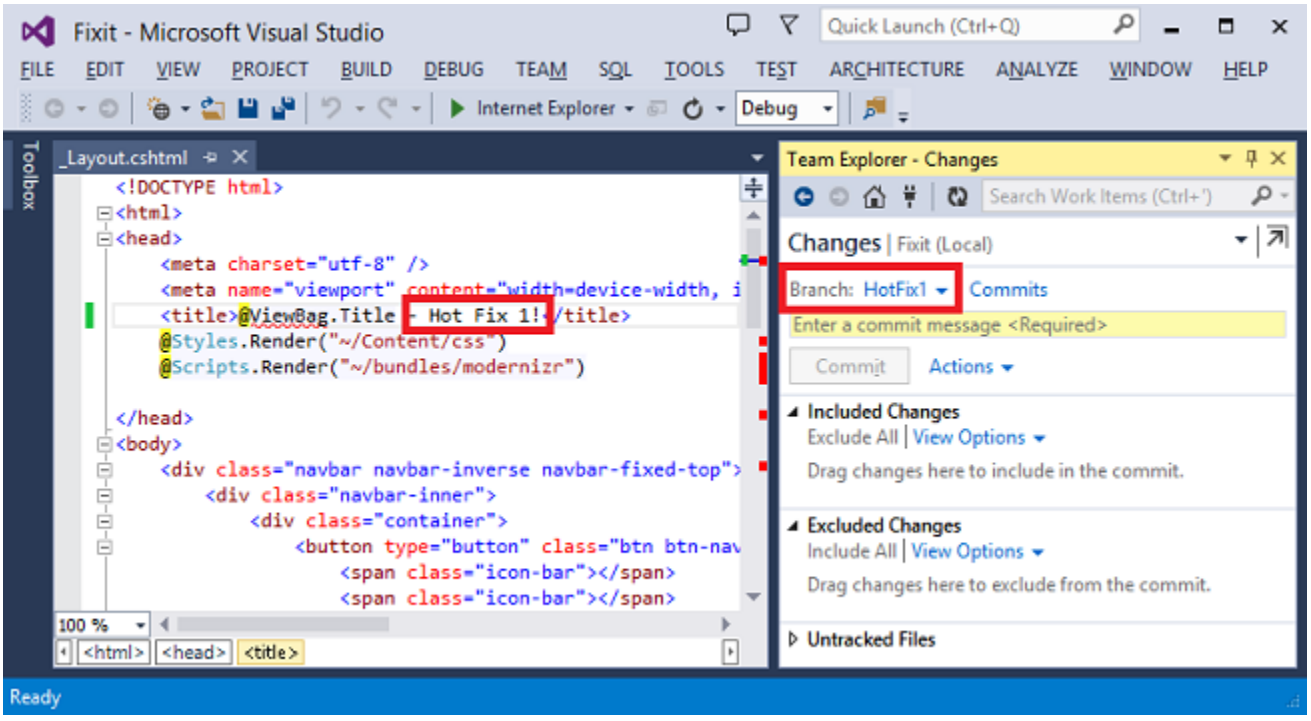


Enter a branch name, click **Create Branch**, and if you selected **Checkout branch**, Visual Studio automatically checks out the new branch.

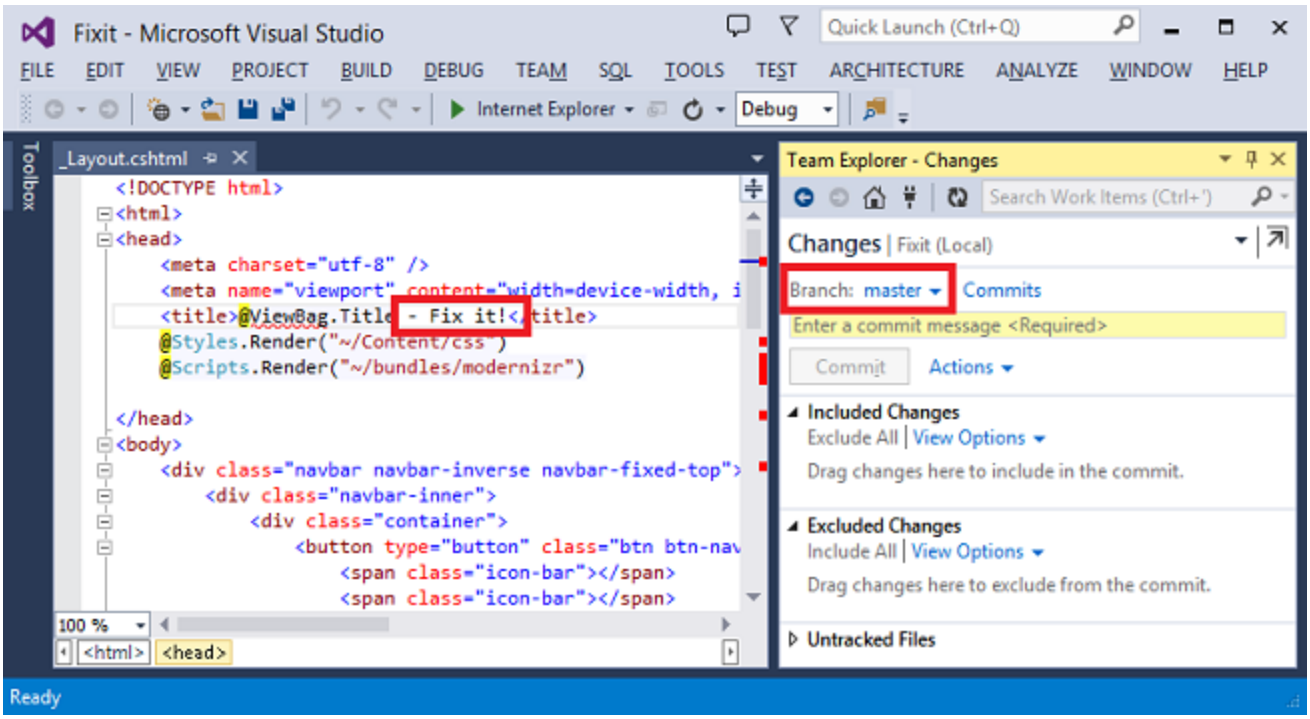
www.dbooks.org



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 web page title in *_Layout.cshtml* has been changed to "Hot Fix 1" in 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 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 the Automate Everything chapter. 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 we're adding Git to Visual Studio Online so that you can use Git integrated with all the other Visual Studio Online capabilities 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 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.

www.dbooks.org

Resources

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 and 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).
- 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 storing settings in Windows Azure, see the following resources:

- Windows Azure Web Sites: How Application Strings and Connection Strings Work. Explains the Windows 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.

Continuous Integration and Continuous Delivery

The first two recommended development process patterns were Automate Everything and Source Control, and the third process pattern 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.

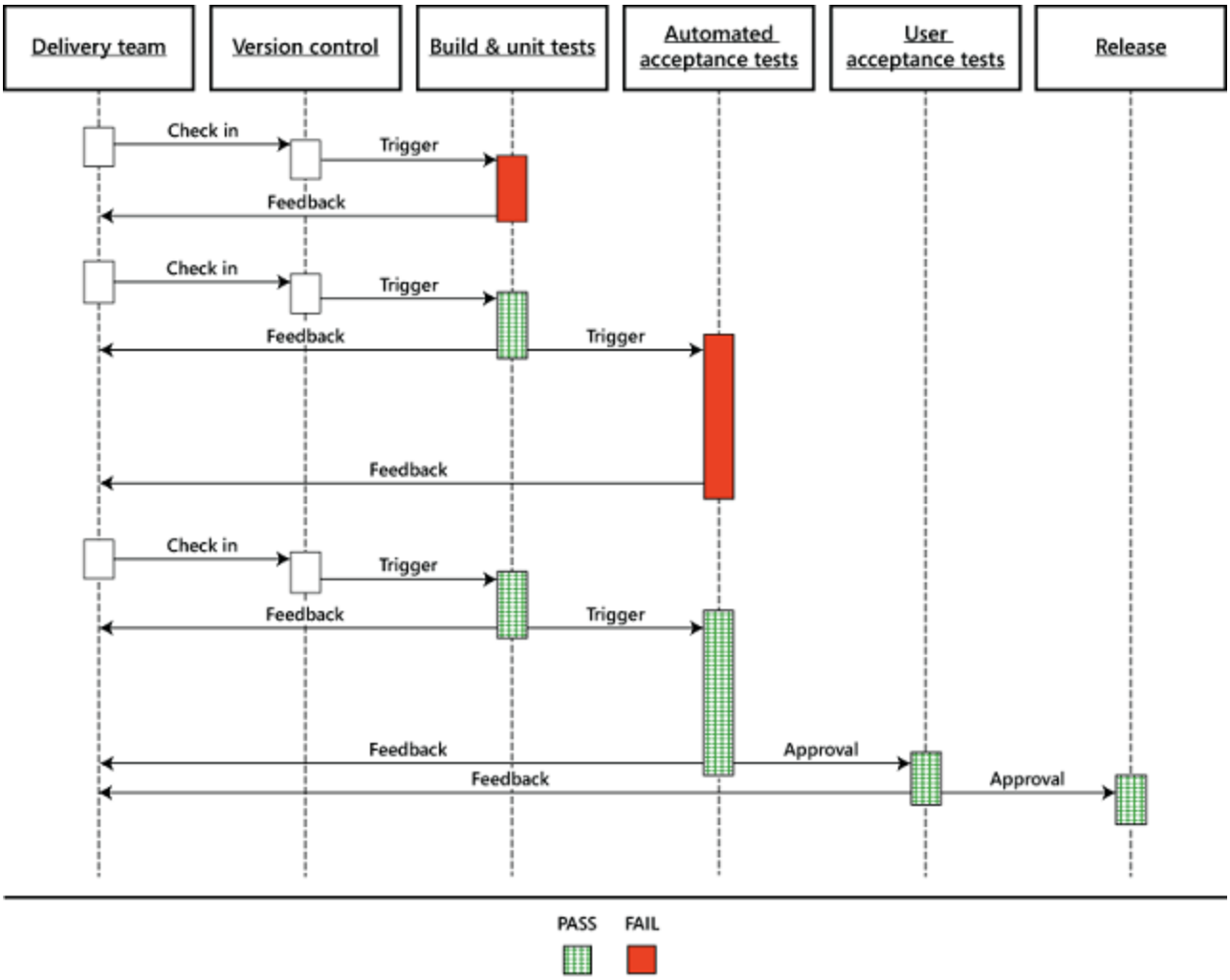
The cloud enables you to minimize the cost of maintaining a test environment because you only pay for the environment resources as long as you're using them. Your CD process can set up the test environment when you need it, and you can take down the environment when you're done testing.

Continuous Integration and Continuous Delivery workflow

Generally we recommend that you do continuous delivery to 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 make sure it 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 e-book about continuous delivery illustrates a typical workflow. Click the image to see it full size in its original context.

www.dbooks.org



How the cloud enables cost-effective CI and CD

Automating these processes in Windows 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. With every build that you do, you could spin up a test environment in Windows Azure using your automation script, run acceptance tests or more in-depth tests against it, and then when you're done just tear it down. And if you only run that server for 2 hours or 8 hours or a day, the amount of money that you have to pay for it is minimal, because you're only paying for the time that a machine is actually running. For example, the environment required for the Fix it application basically costs about 1 cent per hour if you go one tier up from the free level. Over the course of a month, if you only ran the environment 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 (VSO), formerly known as Team Foundation Service, is to make it work really well with continuous integration and delivery. Here are some key features of VSO:

- 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 have 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 Windows 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 VSO, 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 VSO. The service is free for up to 5 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 the following resources; however, if you're new to Windows Azure it might be best to save these for later, after you've learned enough to develop and deploy your application manually.

Getting started tutorials:

- Continuous delivery to Windows Azure by using Visual Studio Online. Step-by-step tutorial that shows how to get started with VSO and set up a Visual Studio project for continuous delivery. The tutorial shows source control and deployment to a Windows Azure Cloud Service. (TFVC is the centralized source control option in VSO, as opposed to Git, which is the distributed source control option.)

www.dbooks.org

- Deliver to Azure Continuously. Step by step tutorial that shows how to set up continuous delivery from VSO to a Windows Azure Web Site, using TFVC. See also the blog post Announcing Continuous Deployment to Azure with Visual Studio Online.
- Windows Azure: Continuous Delivery. This blog by Scott Guthrie announces a number of new Windows Azure features, and includes a section on continuous delivery support for Git in VSO, for both Web Sites and Cloud Services.
- Publishing from Source Control to Windows Azure Web Sites. Step-by-step tutorial that shows how to set up continuous deployment from a local Git repository or from a public Git repository such as BitBucket, CodePlex, or GitHub.

The following white papers and tools use Team Foundation Server rather than Visual Studio Online, but they explain concepts and procedures that apply more generally to Continuous Delivery:

- Building a Release Pipeline with Team Foundation Server 2012. E-book, hands-on labs, and sample code by Microsoft Patterns and 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 TFS 2012*, as a great way to start learning the concepts of DevOps & 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. E-book by Microsoft Patterns and 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 in order to enable operations to "roll back" to a previously deployed version. The tool has no external dependencies and can function stand-alone 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 a Windows Azure Web Site.

Hard-copy books:

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

Web Development Best Practices

The first three patterns were about setting up an agile development process; the rest are about architecture and code. This one 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, 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 only pay for server resources 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 to scale out the application. That too 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; and if the web tier is stateless, re-routing 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 using a cache provider. You can store files in Windows Azure Blob storage instead of the local file system.

As an example of how easy it is to scale an application in Windows Azure Web Sites if your web tier is stateless, see the **Scale** tab for a Windows Azure Web Site in the management portal:

www.dbooks.org

fixitdemo

DASHBOARD MONITOR CONFIGURE SCALE PREVIEW LINKED RESOURCES

general

WEB SITE MODE FREE SHARED STANDARD

CHOOSE SITES 3 sites selected

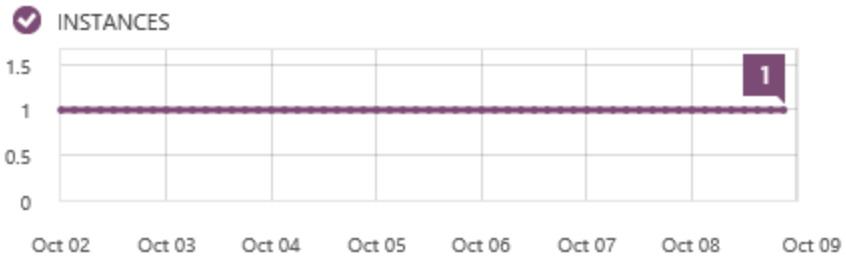
capacity

INSTANCE SIZE Small (1 core, 1.75 GB Memory)

EDIT SCALE SETTINGS FOR SCHEDULE No scheduled times

set up schedule times

SCALE BY METRIC NONE CPU



INSTANCE COUNT 1 instances

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 5 web servers in Windows Azure handling your web site's traffic.



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

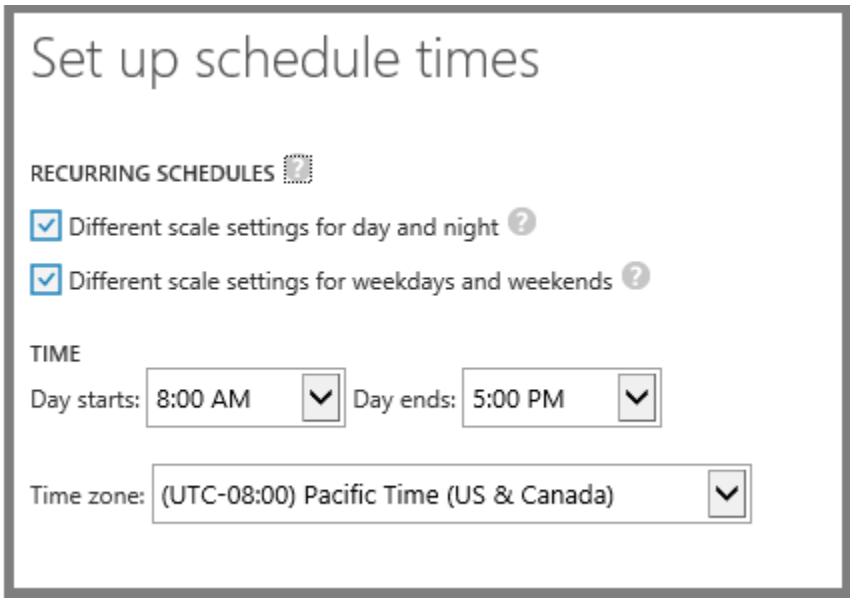
You can also tell Windows Azure to automatically increase or decrease the number of web servers based on CPU usage. In the following example, when CPU usage goes below 60%, the number of web servers will decrease to a minimum of 2, and if CPU usage goes above 80%, the number of web servers will be increased up to a maximum of 4.

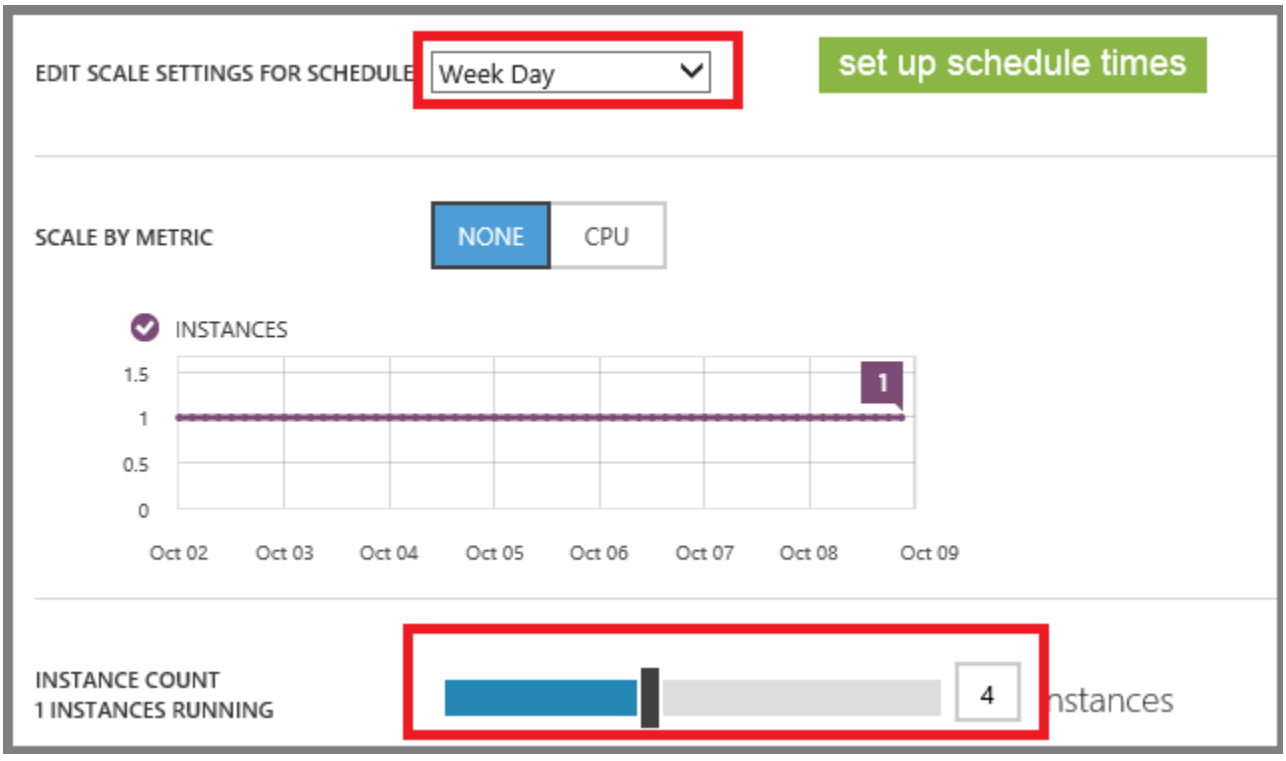


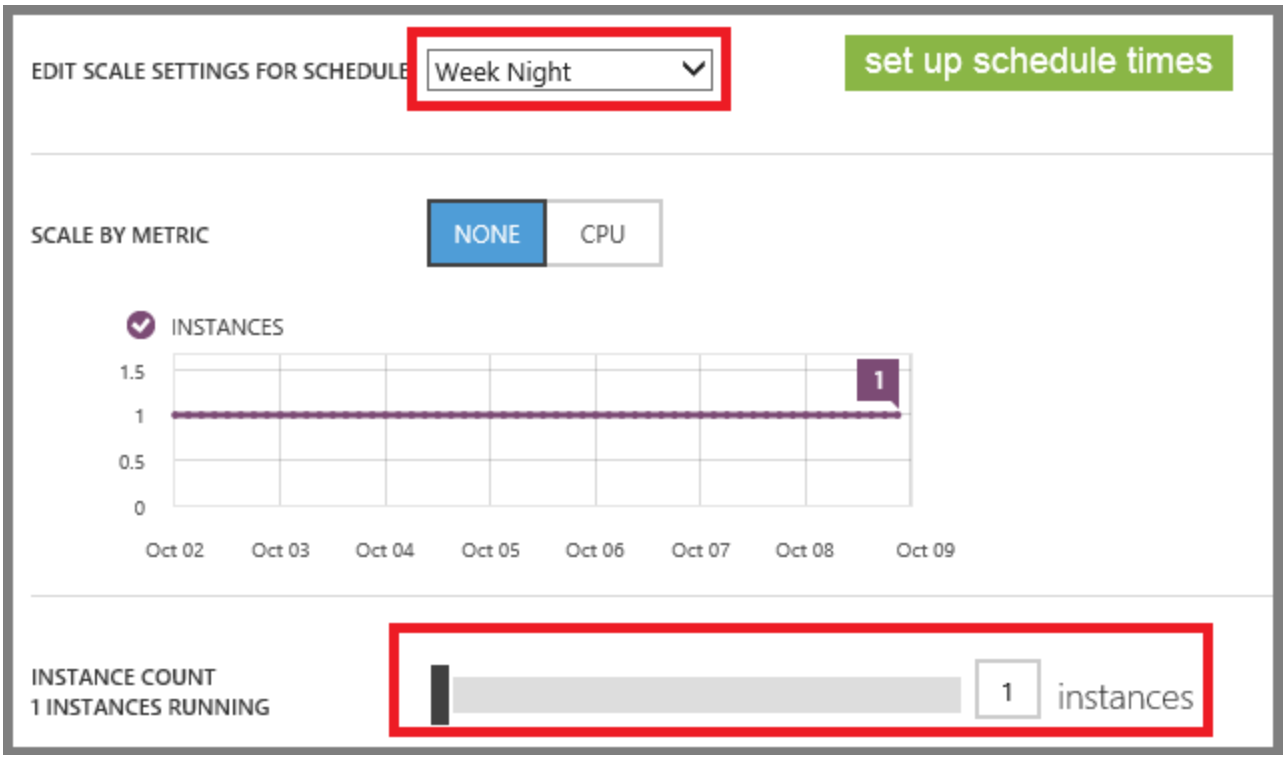
Or what if you know that your site will only be busy during working hours? You can tell Windows Azure to run multiple servers during the daytime and decrease to a single server evenings, nights, and weekends. The following series of screen shots shows how to set up the web site to run one server in off hours and 4 servers during work hours from 8 AM to 5 PM.

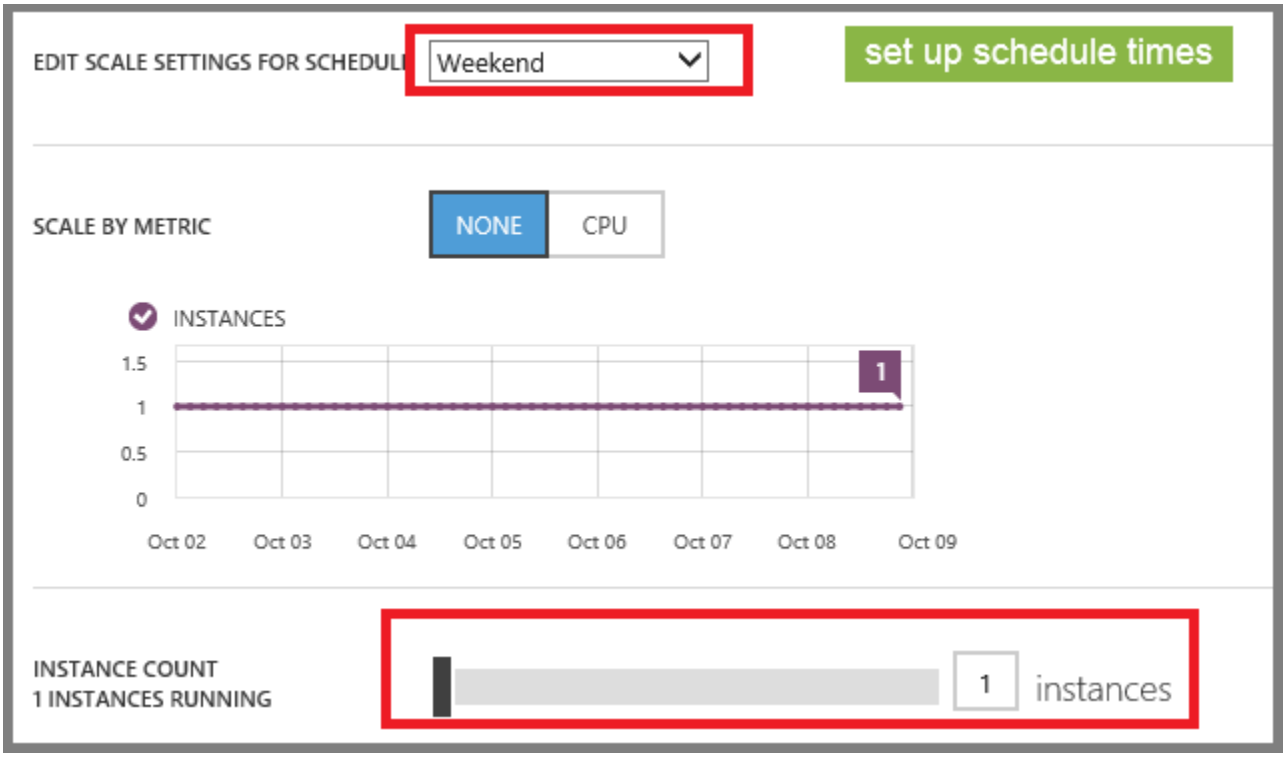
www.dbooks.org


EDIT SCALE SETTINGS FOR SCHEDULE
No scheduled times
set up schedule times


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





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

www.dbooks.org

The ability of your application to scale out is almost unlimited in Windows 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. 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.

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

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

.NET 4.5 enhanced the C# and VB programming languages in order to make it much simpler to handle tasks asynchronously. The benefit of asynchronous programming is not just for 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 reliable under high load conditions. A web server only has 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 the is received. In a busy web server, hundreds or thousands of requests can then be processed promptly which would otherwise be waiting for threads to be freed up.

As you saw earlier, it's as easy to decrease the number of web servers handling your web site 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 Windows 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 in order to display it in a web page or to save data entered in a web page. 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);
}
```

And here is the same method that 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.

www.dbooks.org

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

Async support in Entity Framework 6

As part of async support in 4.5 we shipped 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 our data libraries didn't support async. Now Entity Framework 6 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 we have tools in ASP.NET and Windows Azure to 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 and Practices - Windows Azure Guidance. See Autoscaling guidance.
- Disabling ARR's Instance Affinity in Windows Azure Web Sites. Blog post by Erez Benari, explains session affinity in Windows Azure Web Sites.

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 and Practices Cloud Design Patterns (See Static Content Hosting pattern.)
- CDN Reviews. Overview of many CDNs.

Asynchronous programming:

- Using Asynchronous Methods in ASP.NET MVC 4. Tutorial by Rick Anderson.

www.dbooks.org

- 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 graphic 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 e-book lists a number of best practices that were implemented in the Fix It application.
- Web Developer Checklist

Single Sign-On

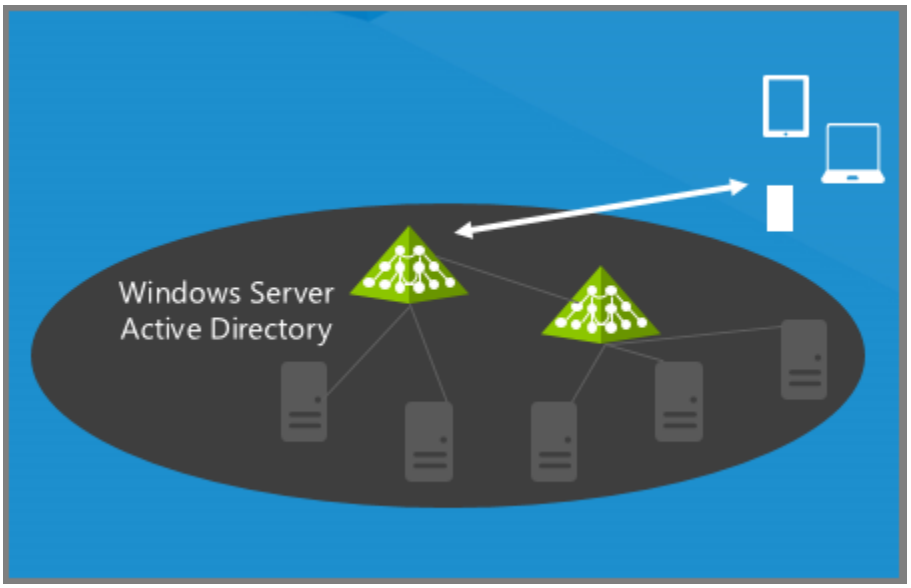
There are many security issues to think about when you're developing a cloud app, but for this series we'll focus on just one: single sign-on. A question people often ask is this: "I'm primarily building apps for the employees of my company; how do I host these apps in the cloud and still enable them 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 Windows Azure Active Directory (WAAD). WAAD 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 WAAD

WAAD 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 Enterprise Graph REST API.

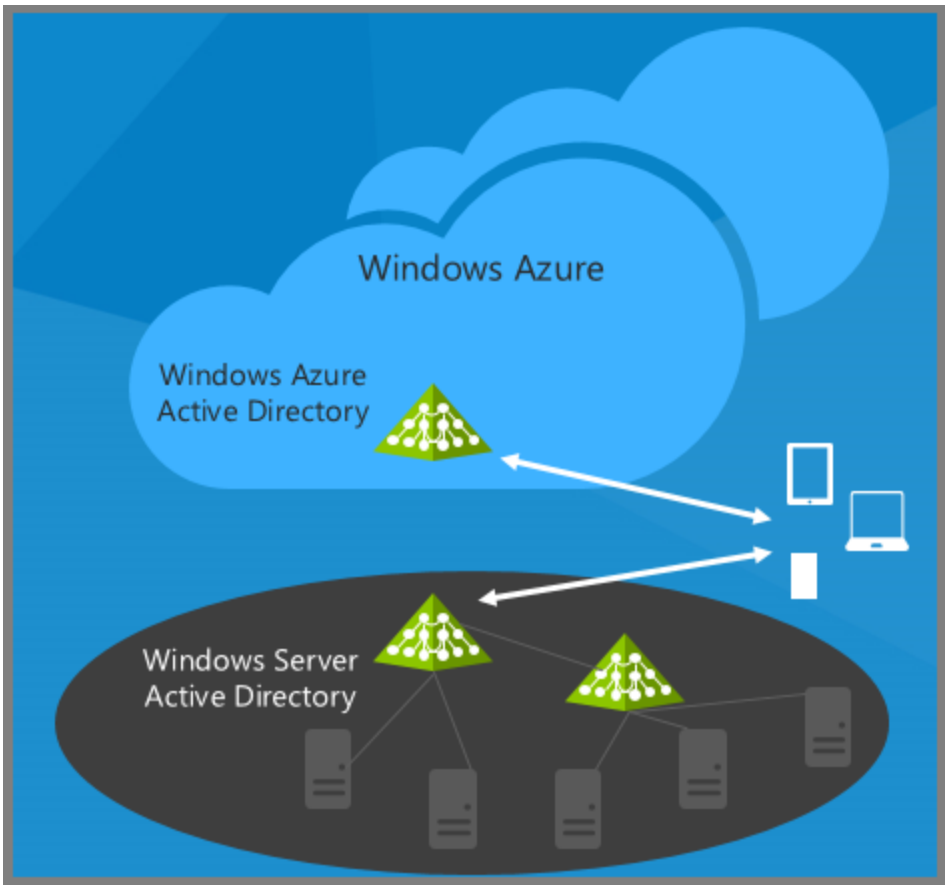
Suppose you have an on-premises Windows Server Active Directory environment that you use to enable employees to sign on to Intranet apps:



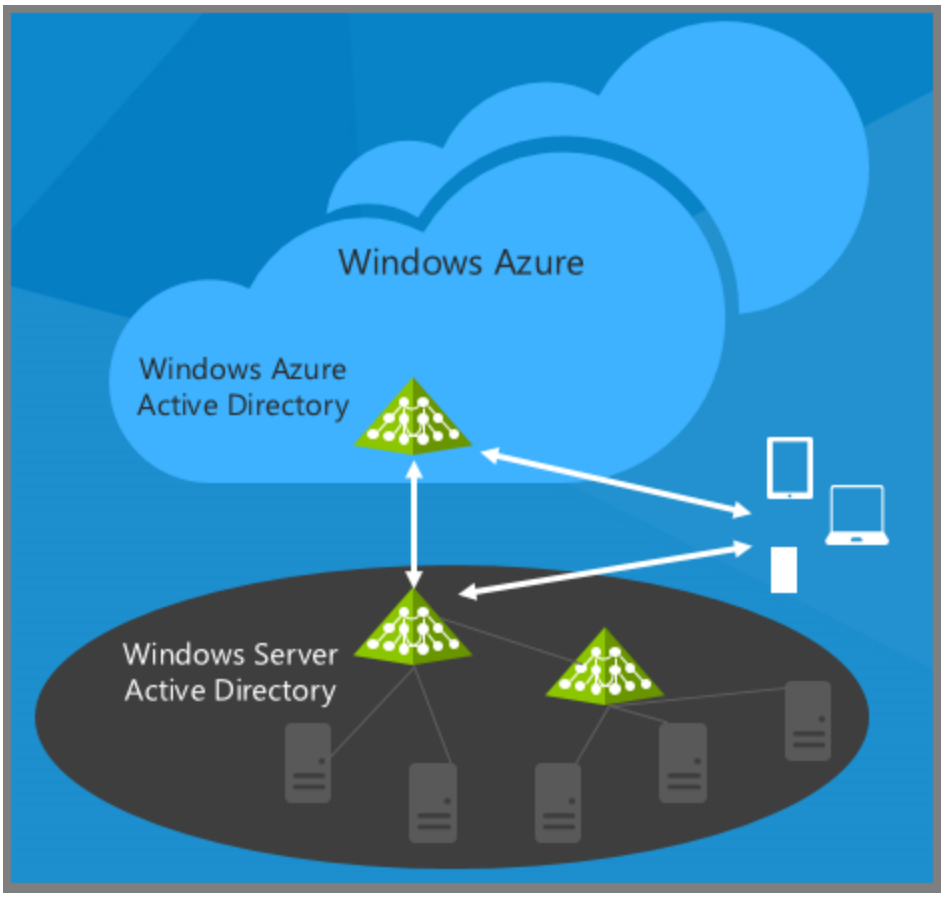
What WAAD enables you to do is create a directory in the cloud. It's a free feature and easy to set up.

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

www.dbooks.org



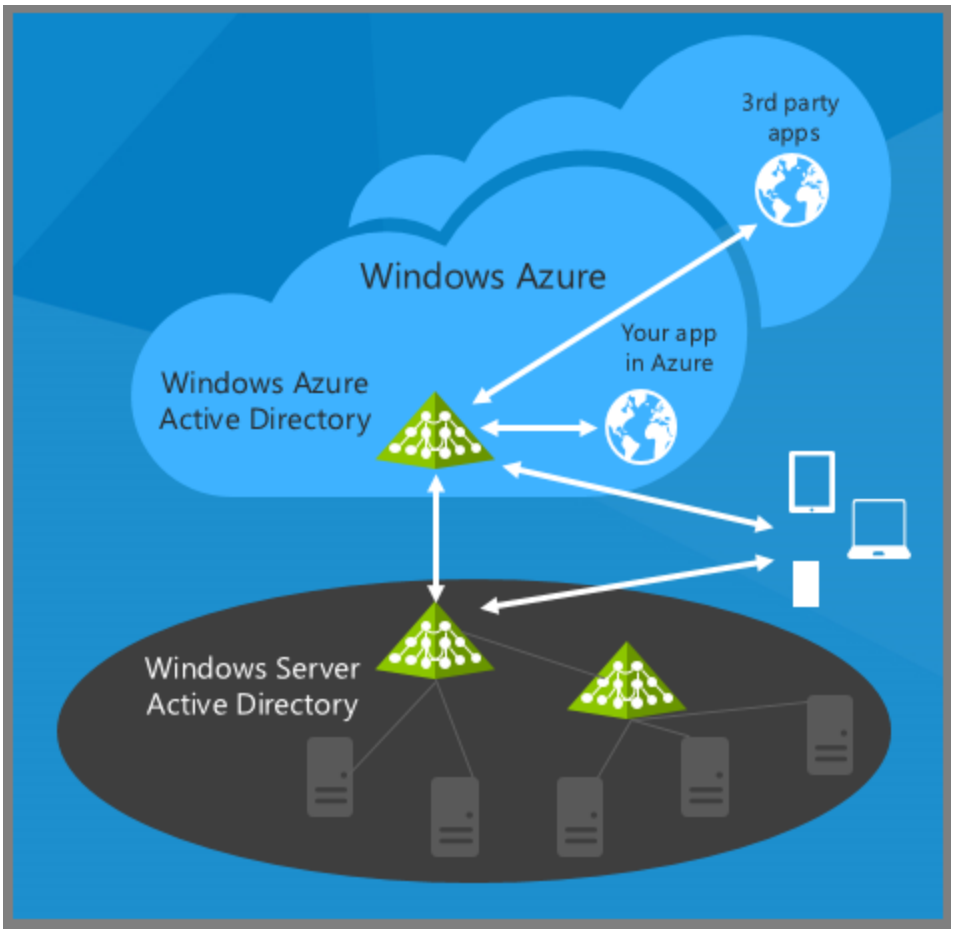
Or you can integrate it with your on-premises AD.



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 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 WAAD, 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 WAAD because Office 365 uses WAAD for authentication and authorization.

www.dbooks.org



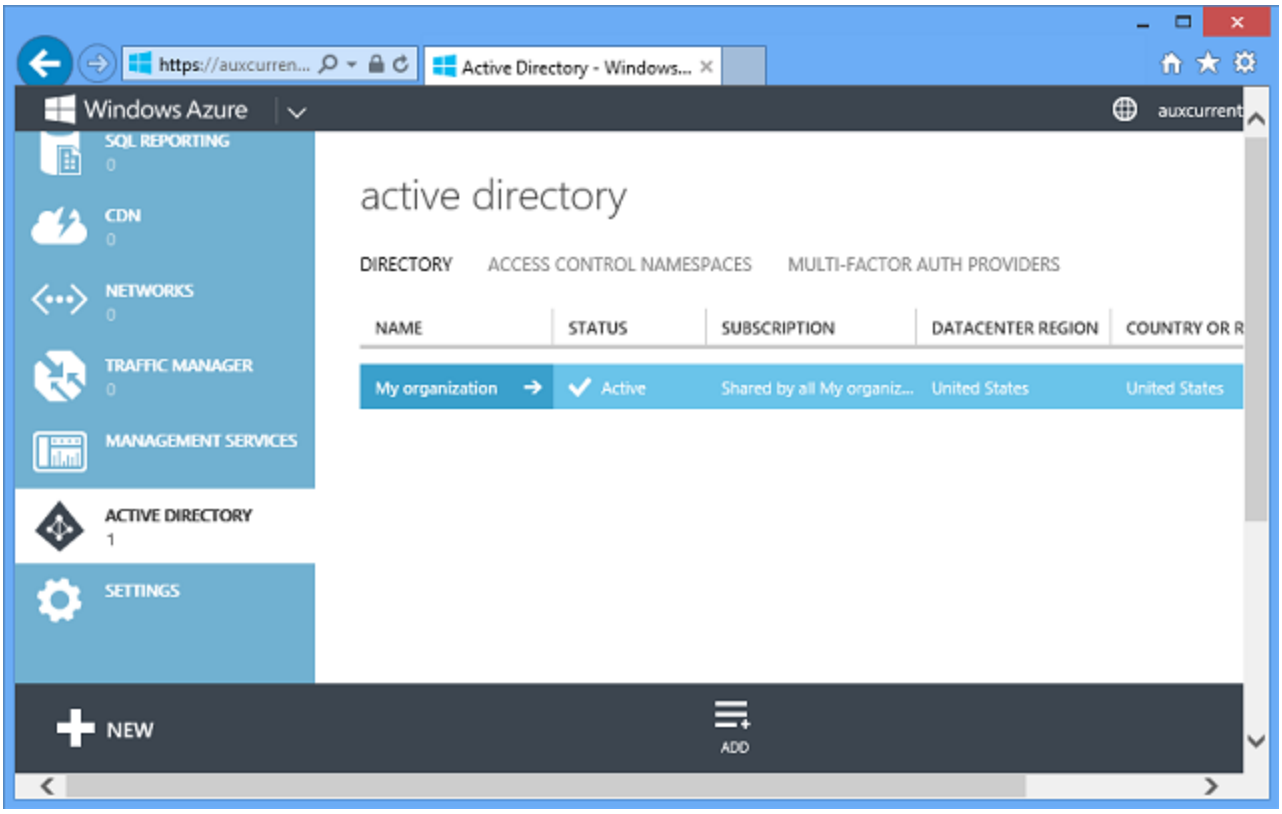
The beauty of this approach is that any time your organization adds or deletes a user, or a user changes a password, you use the same process that you use today in your on-premises environment. All of your on-premises AD changes are automatically propagated to the cloud environment.

If your company is using or moving to Office 365, the good news is that you'll have WAAD set up automatically because Office 365 uses WAAD for authentication. So you can easily use in your own apps the same authentication that Office 365 uses.

Set up a WAAD tenant

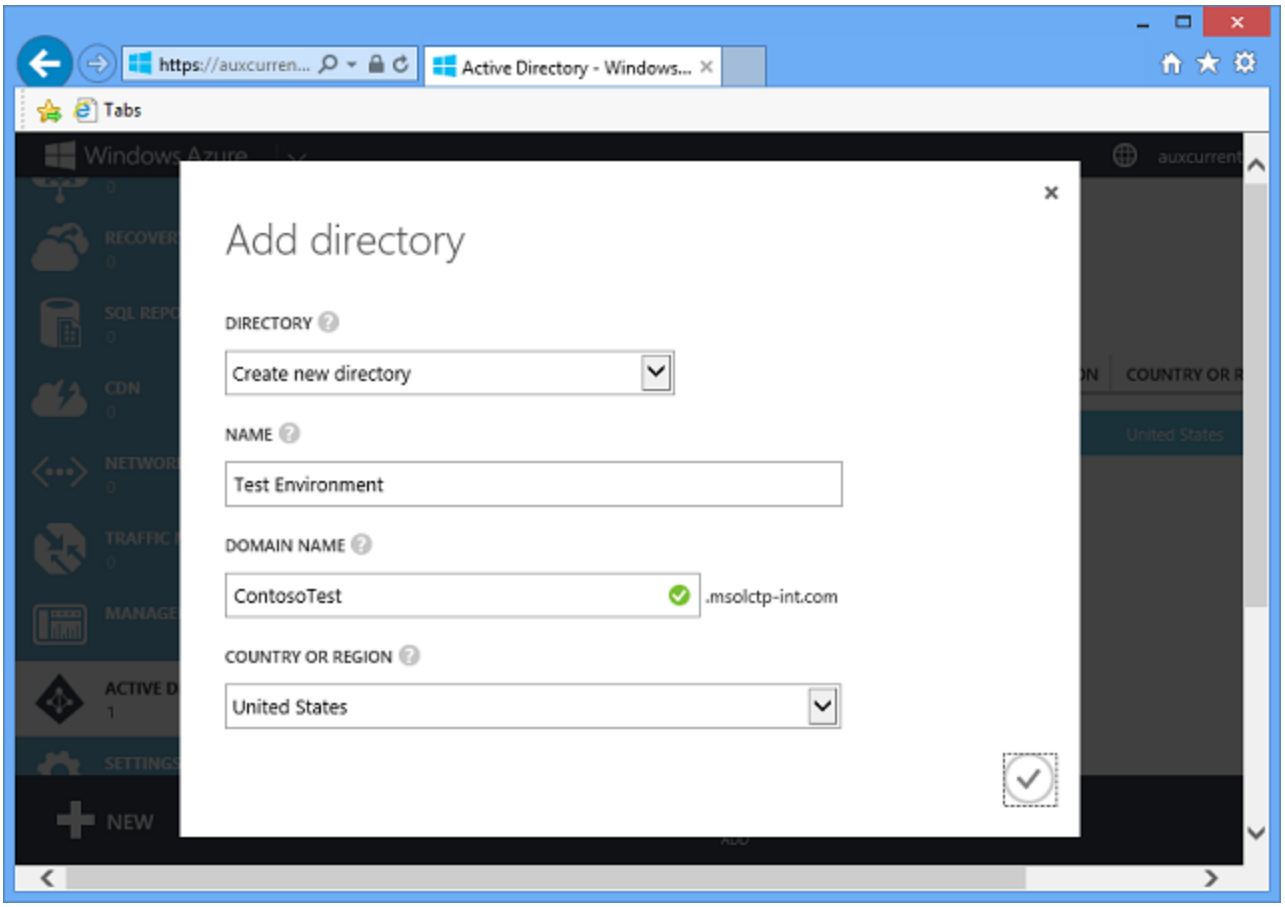
A WAAD directory is called a WAAD tenant, and setting up a tenant is pretty easy. We'll show you how it's done in the Windows Azure Management Portal in order to illustrate the concepts, but of course like the other portal functions you can also do it by using a script or management API.

In the management portal click the Active Directory tab.

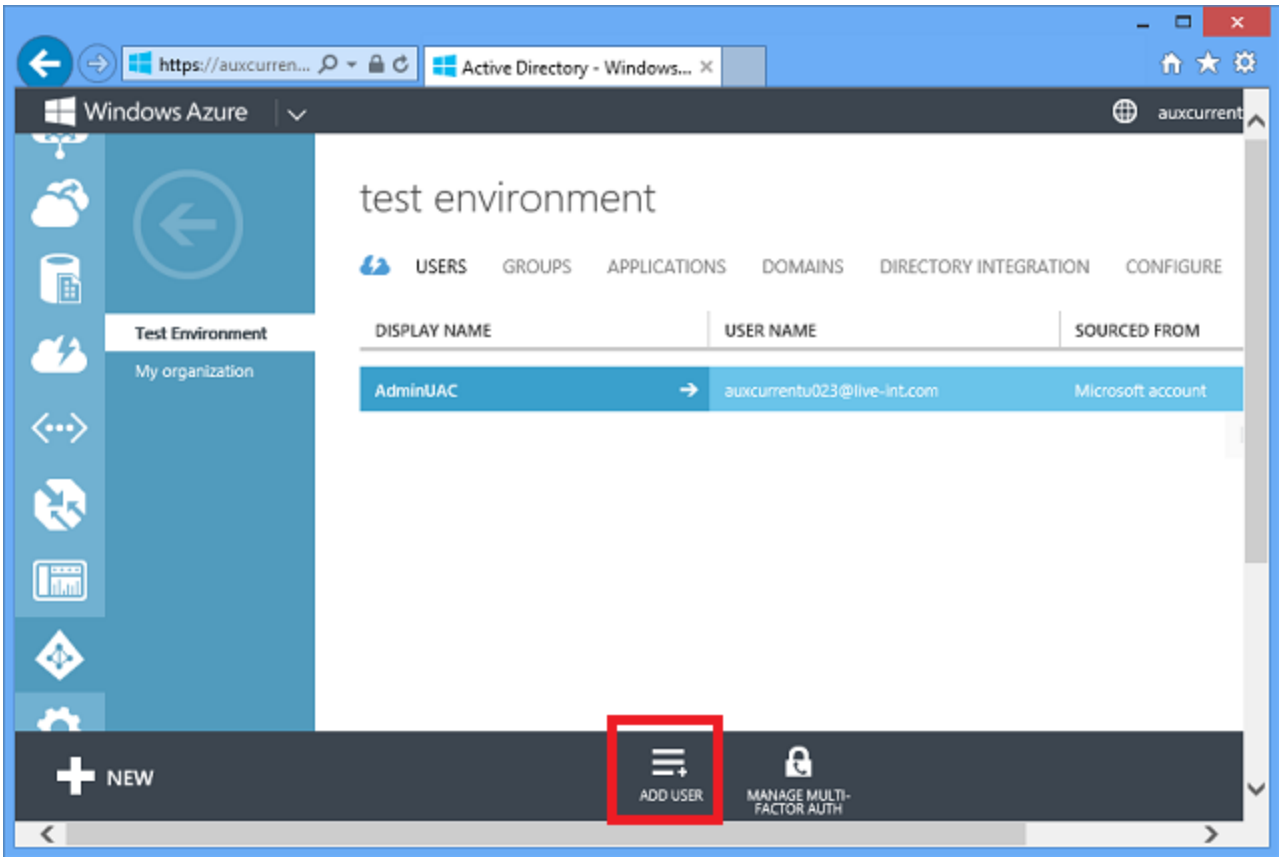


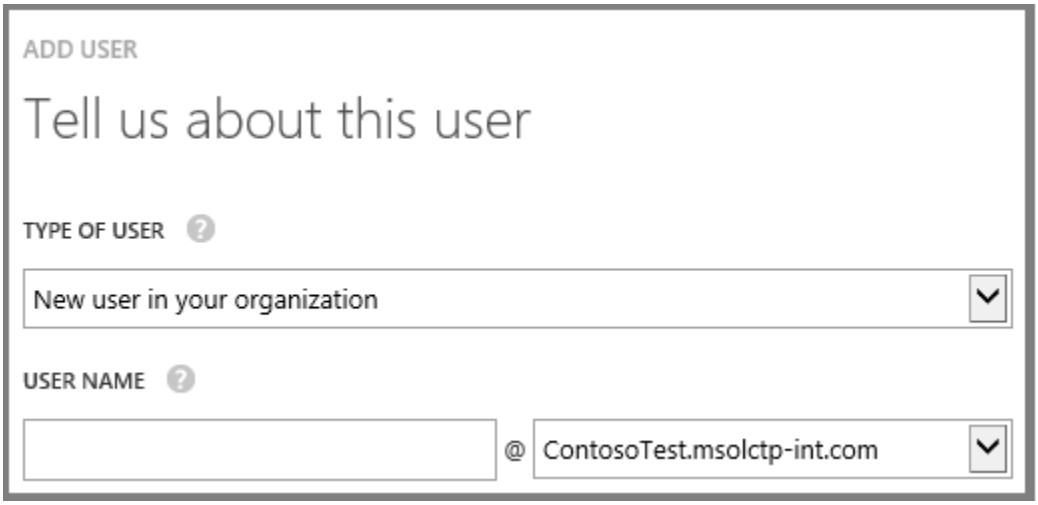
You automatically have one WAAD tenant for your Windows Azure account, and you can click the **Add** button at the bottom of the page to create additional directories. You might want one for a test environment and one for production, for example. Think carefully about what you name a new directory. If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

www.dbooks.org



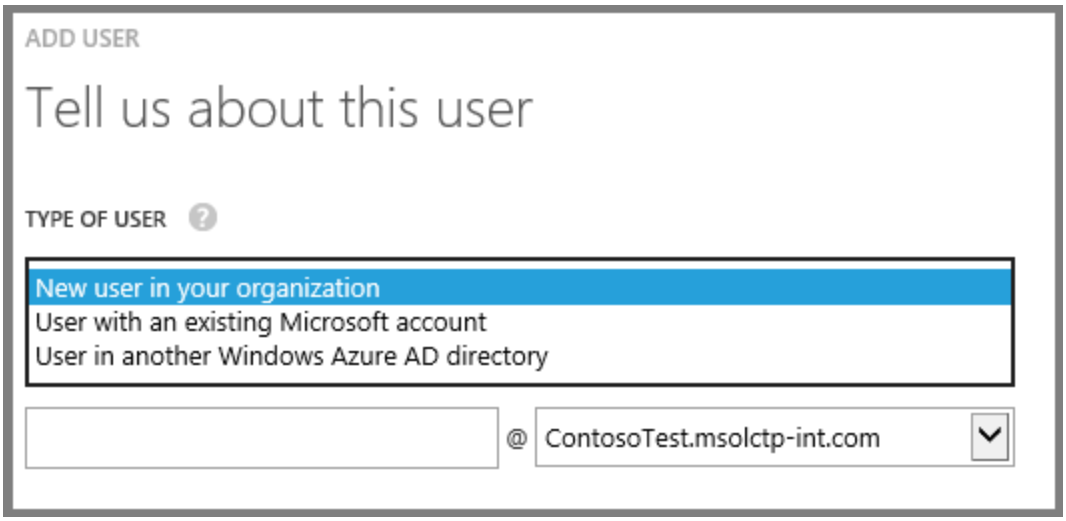
The portal has full support for creating, deleting, and managing users within this environment. For example, to add a user go to the **Users** tab and click the **Add User** button.

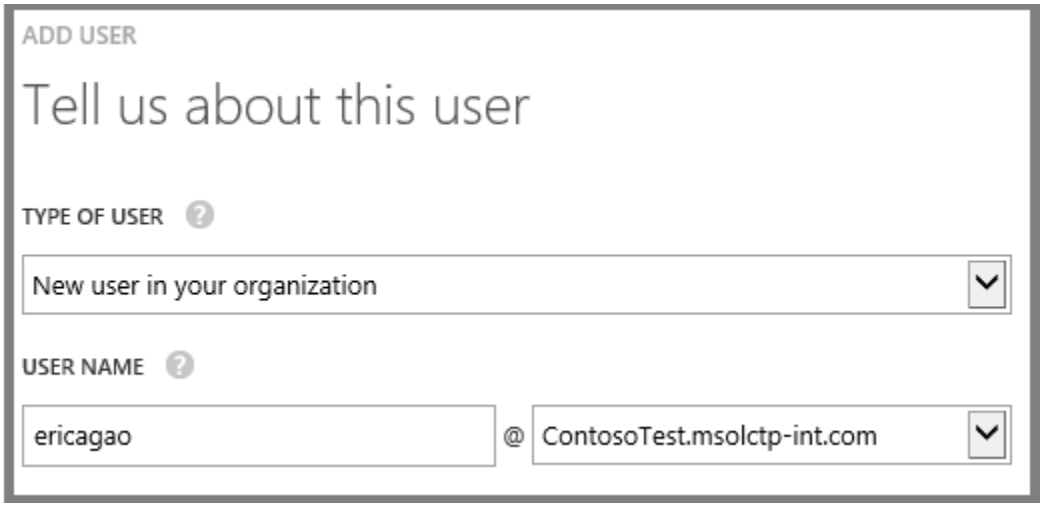




You can create a new user who exists only in this directory, or you can register a Microsoft Account as a user in this directory, or register or a user from another WAAD directory as a user in this directory. (In a real directory, the default domain would be ContosoTest.onmicrosoft.com. You can also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

www.dbooks.org

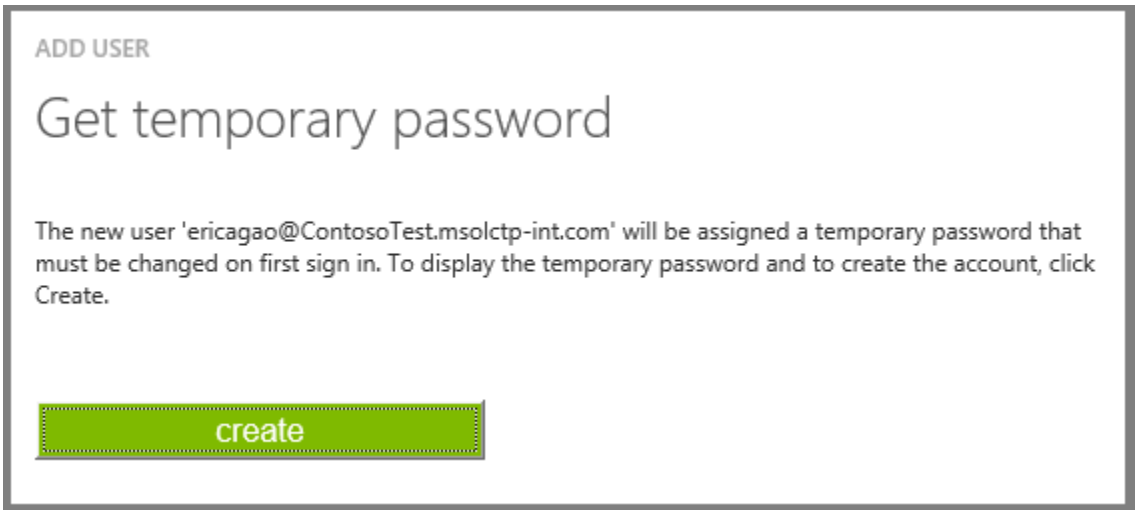




You can assign the user to a role.



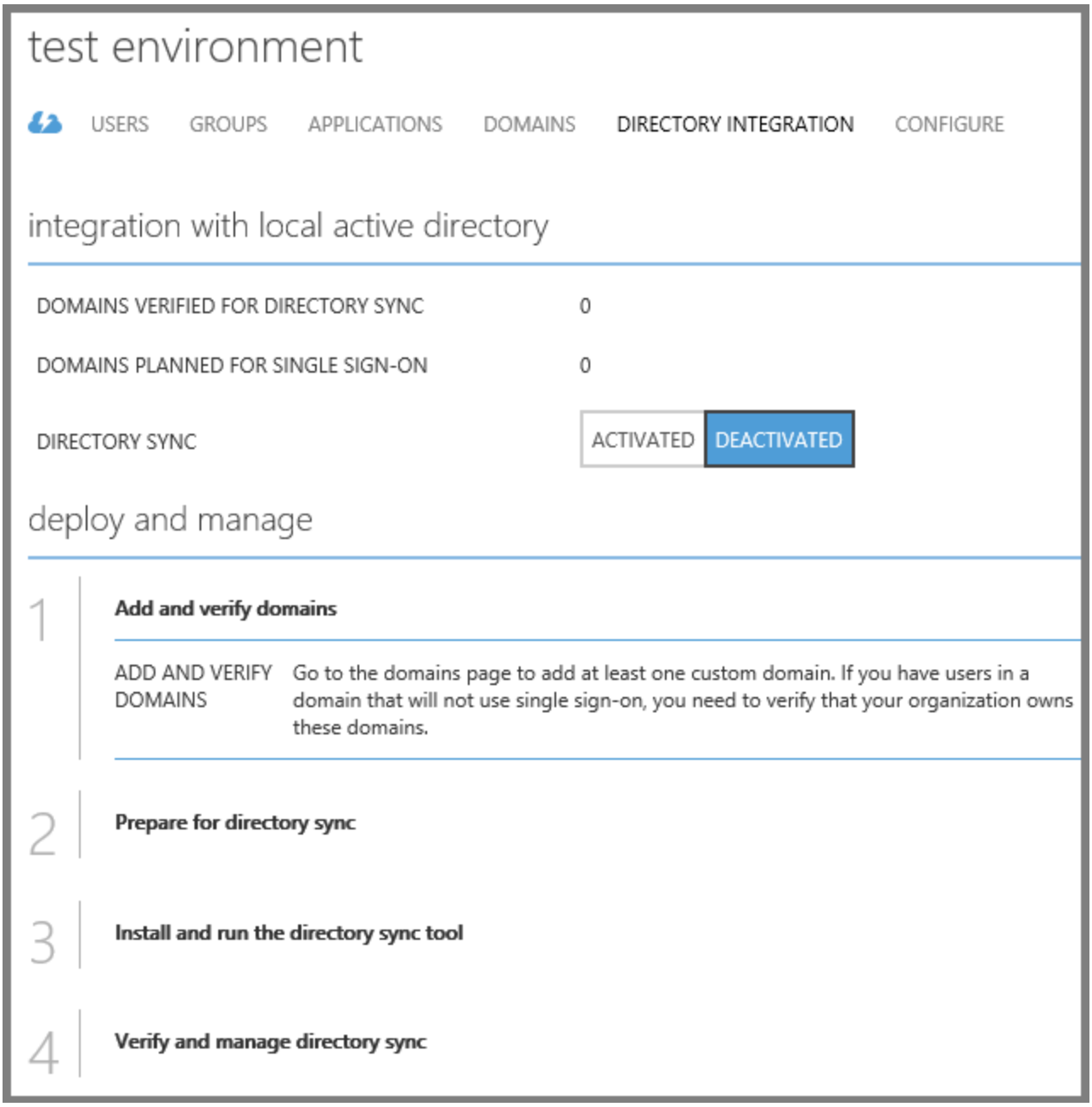
And the account is created with a temporary password.



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

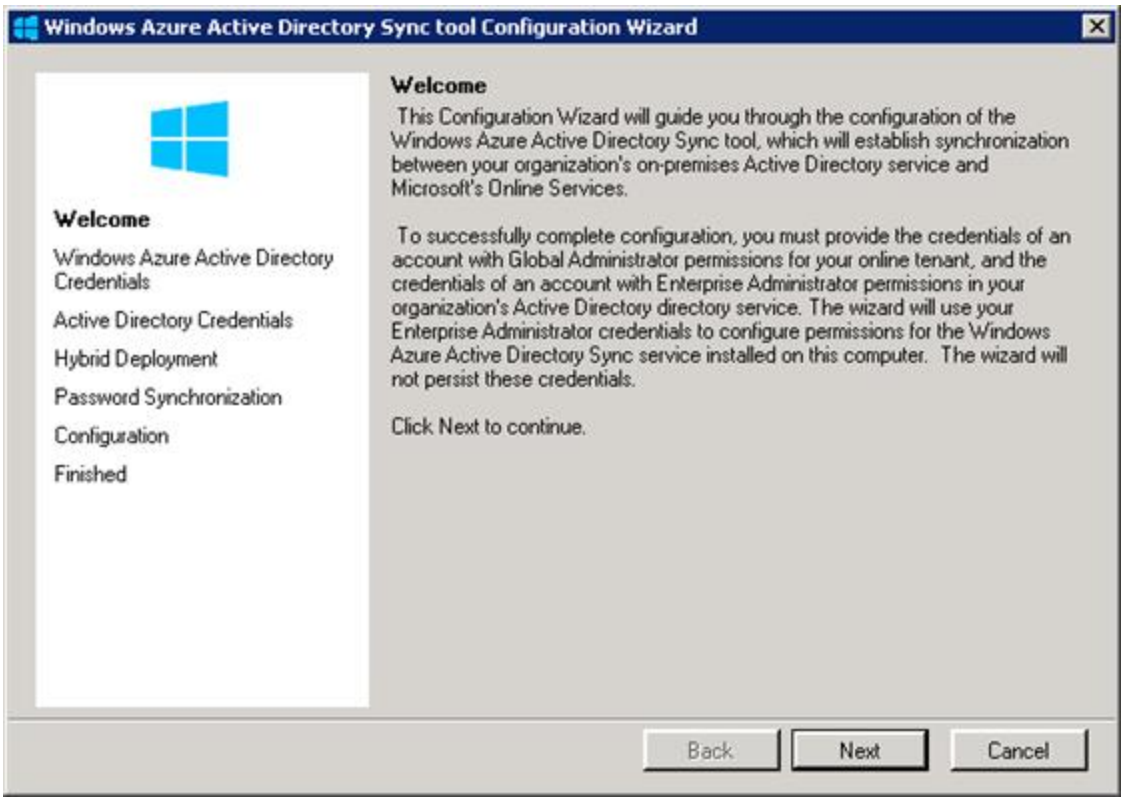
What's great for enterprise single sign-on, though, is the **Directory Integration** tab:

www.dbooks.org

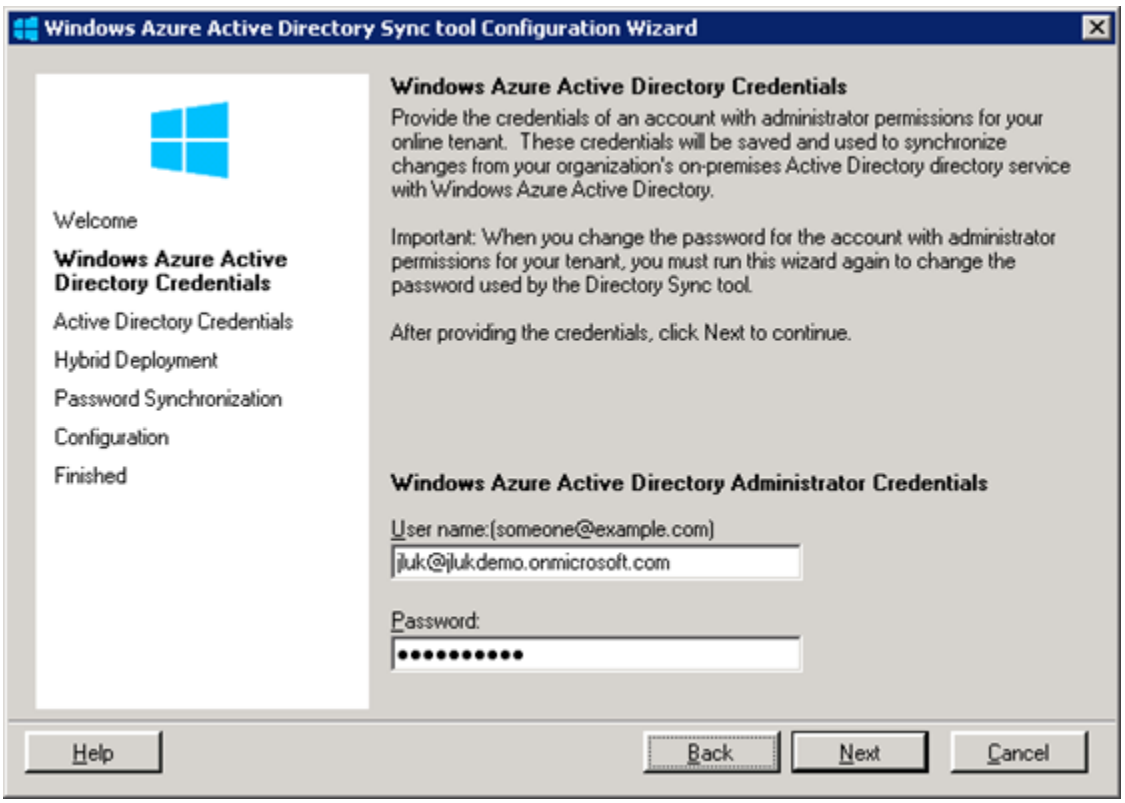


If you enable directory integration, and download a tool, you can sync this cloud directory with your existing on-premises Active Directory that you're already using inside your organization. Then all of the users stored in your directory will show up in this cloud directory. Your cloud apps can now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

The tool is a wizard that is easy to use, as you can see from these 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.

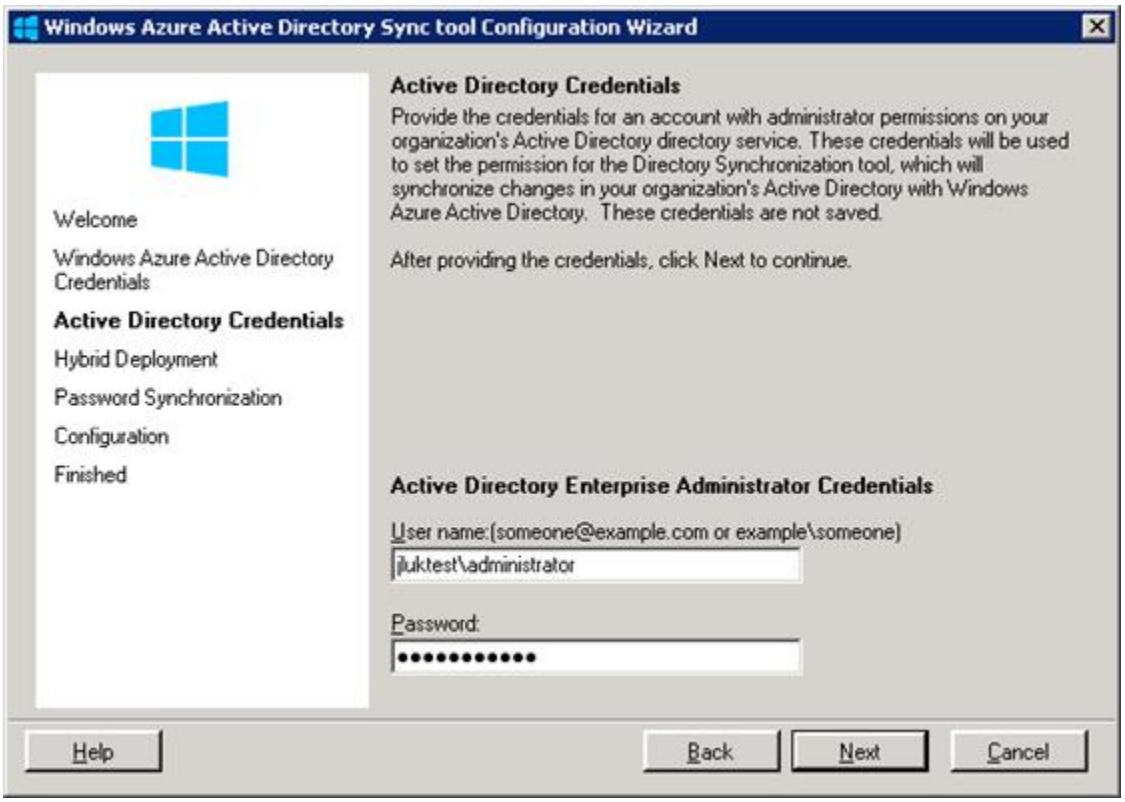


Click **Next**, and then enter your Windows Azure Active Directory credentials.

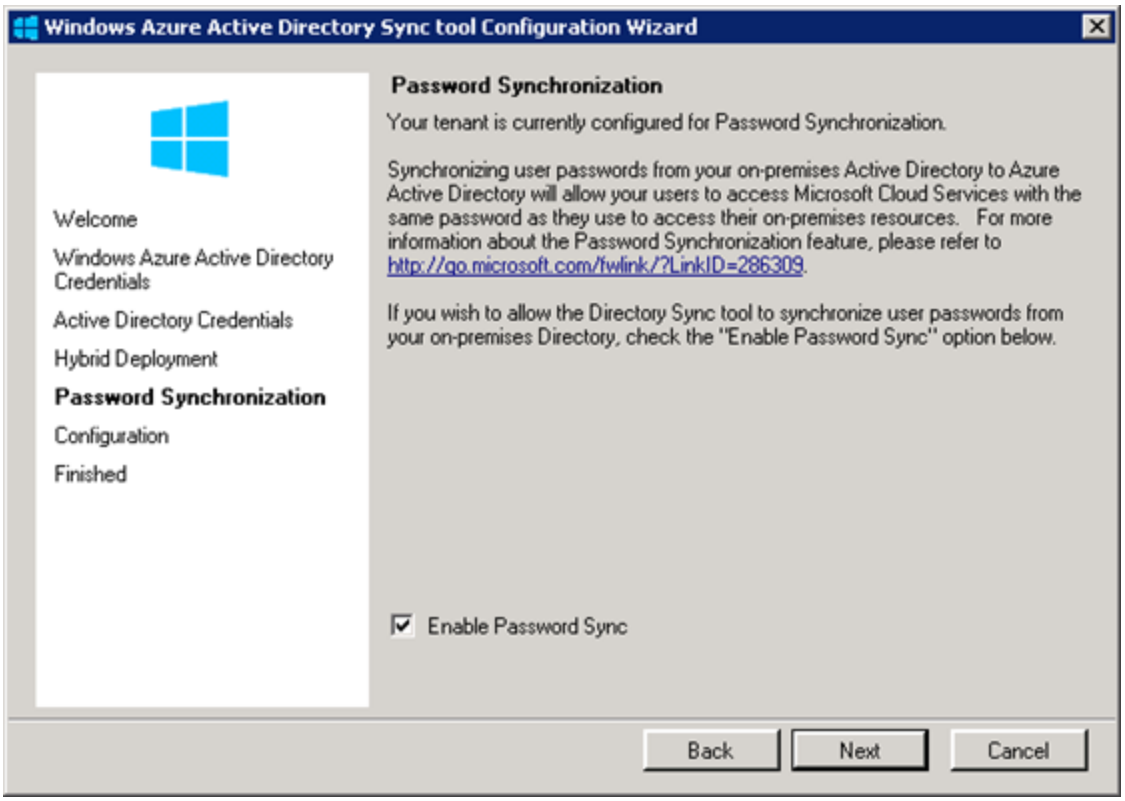


www.dbooks.org

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



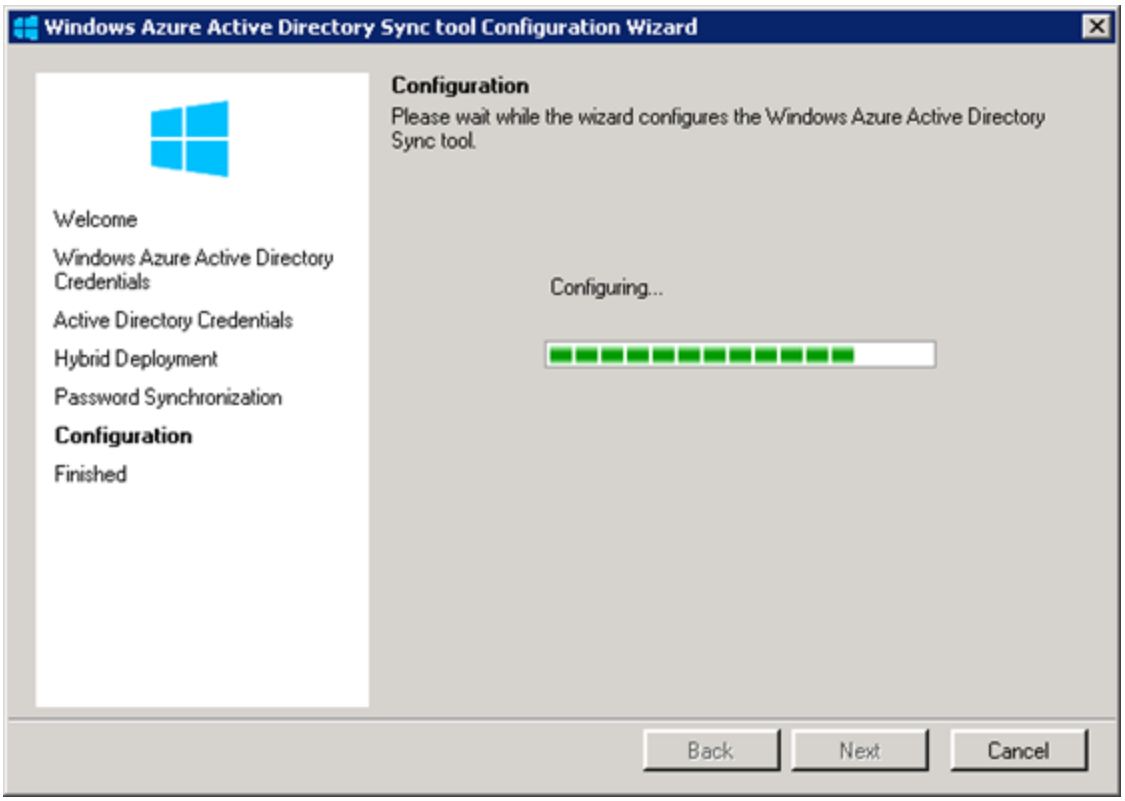
Click **Next**, and then indicate if you want to store a hash of your AD passwords in the cloud.



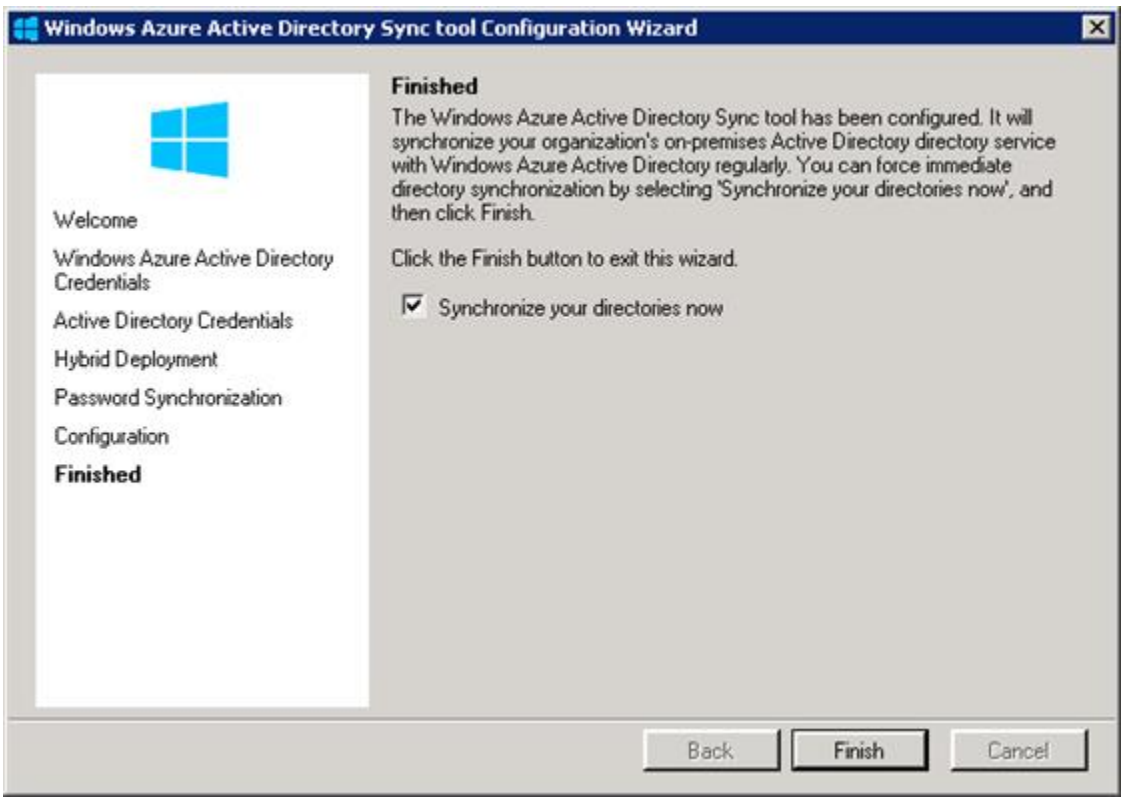
The password hash that you can store in the cloud is a one-way hash; actual passwords are never stored in WAAD. If you decide against storing hashes in the cloud, you'll have to use Active Directory Federation Services (ADFS). There are also other factors to consider when choosing whether or not to use 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**.

www.dbooks.org

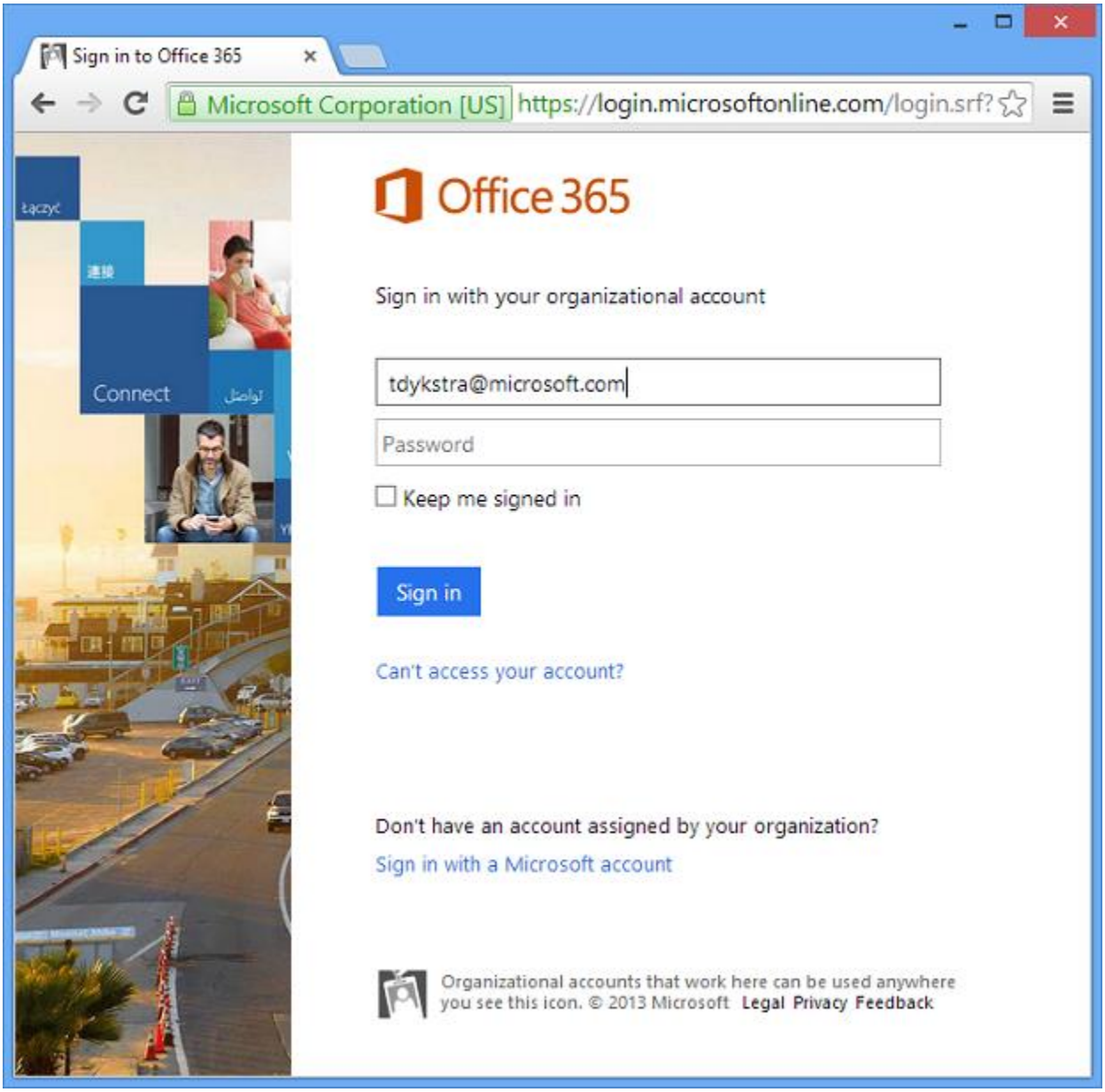


And in a few minutes you're done.



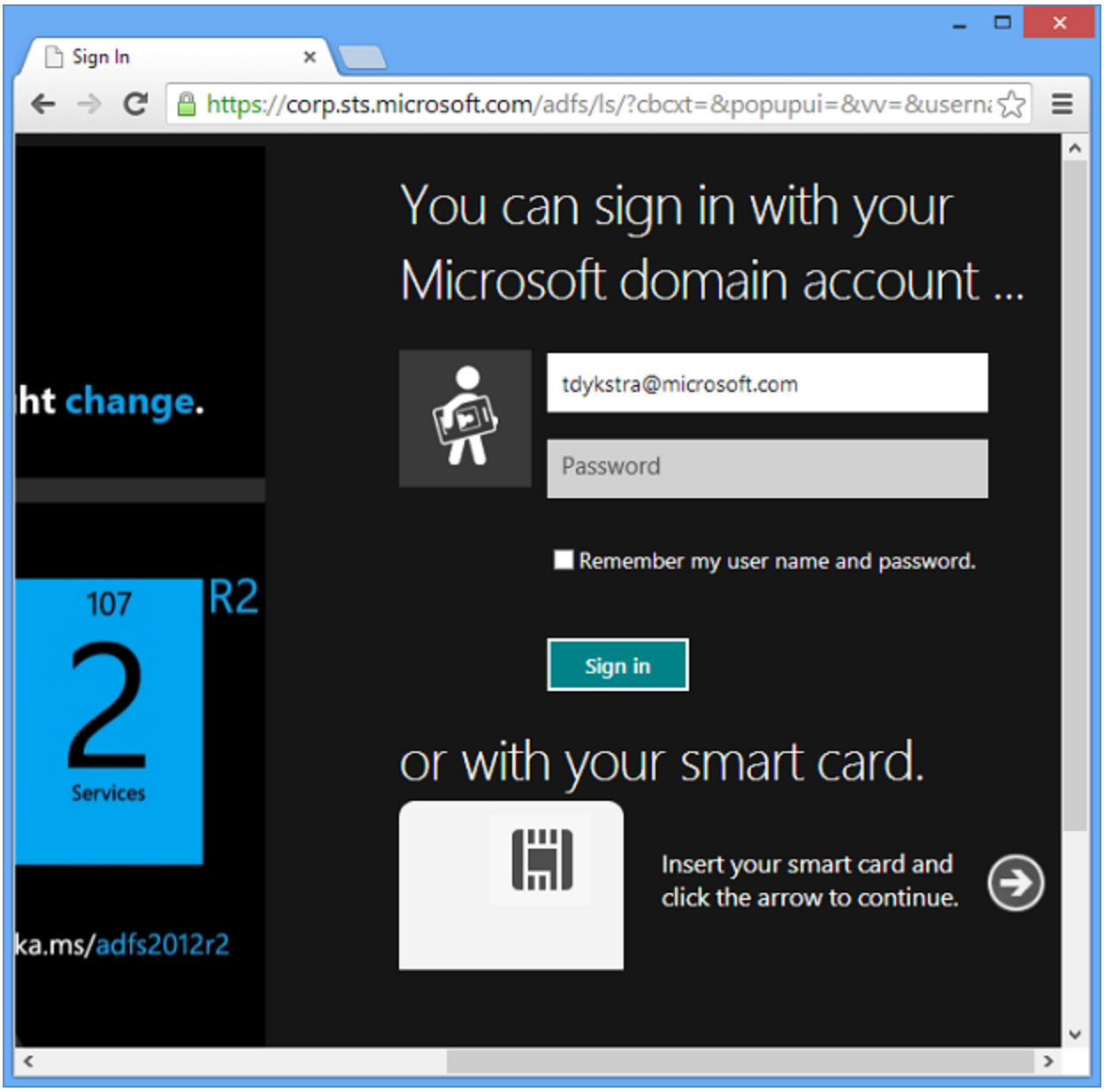
You only have to run this on one domain controller in the organization, on Windows 2003 or higher. And no need to reboot. When you're done, all of your users are in the cloud and you can do single sign-on from any web or mobile application, using SAML, OAuth, or WS-Fed.

Sometimes we get asked about how secure this is – does Microsoft use it for their own sensitive business data? And the answer is yes we do. For example, if you go to the internal Microsoft SharePoint site at http://microsoft.sharepoint.com, you get prompted to log in.

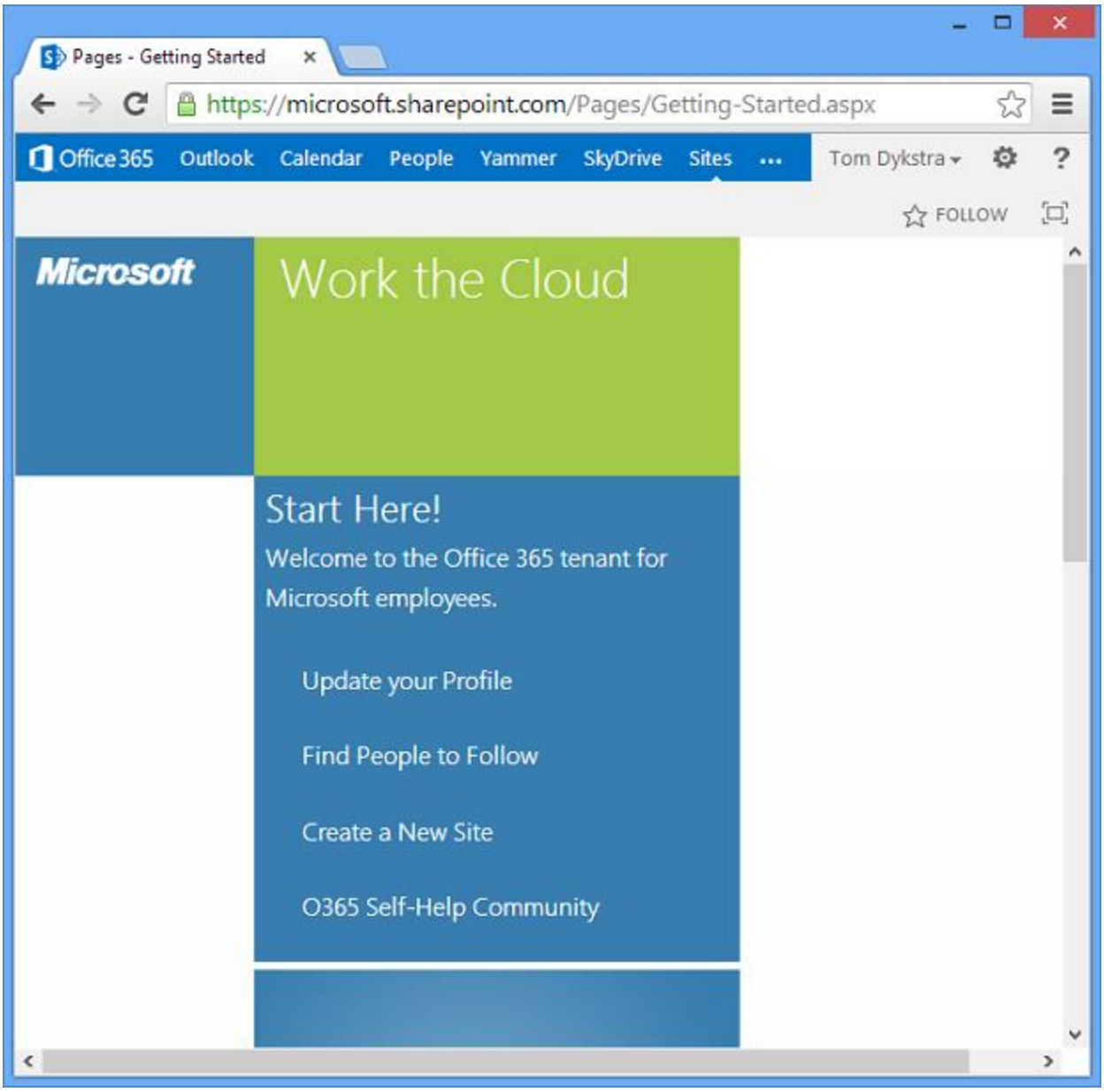


Microsoft has enabled ADFS, so when you enter a Microsoft ID, you get redirected to an ADFS log-in page.

www.dbooks.org



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



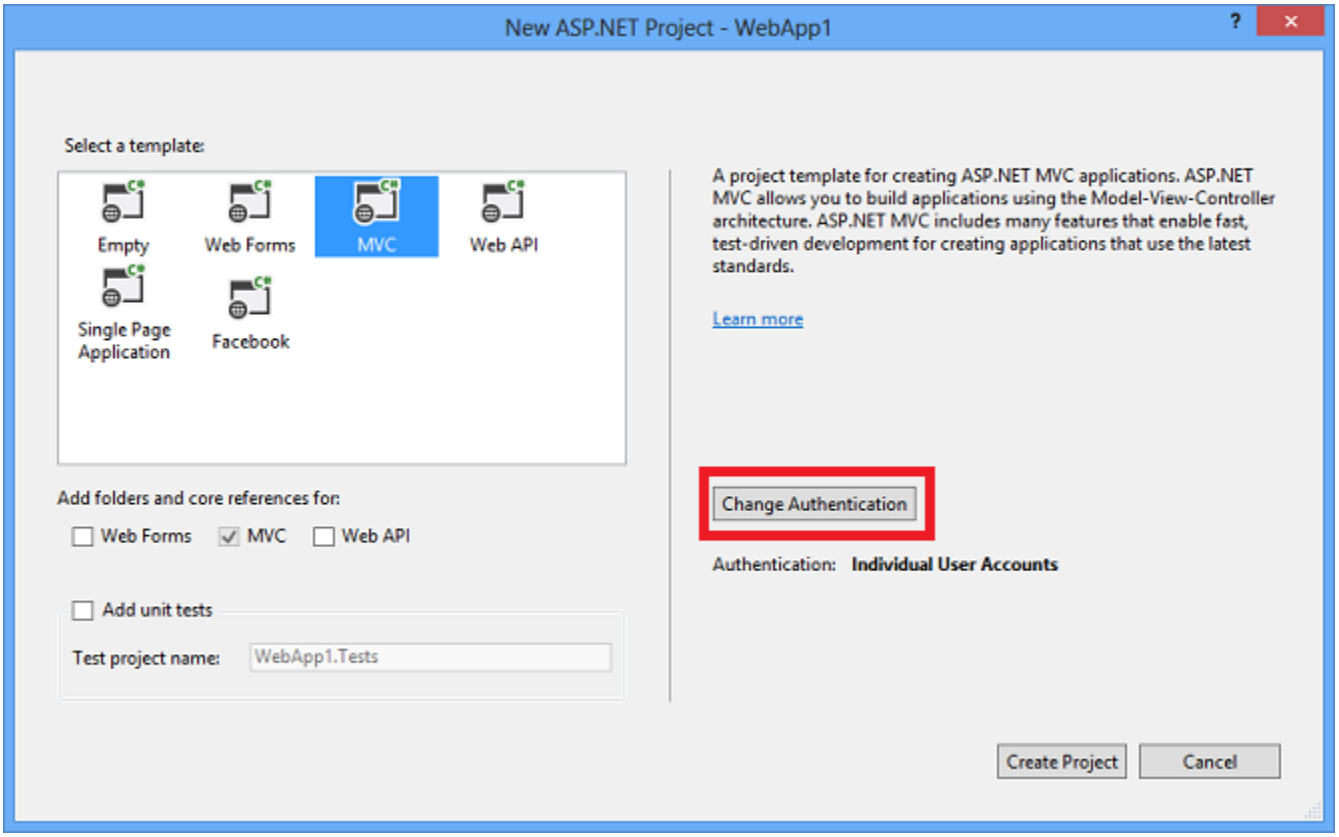
We're using an AD sign-in server mainly because we already had ADFS set up before WAAD became available, but the log-in process is going through a WAAD directory in the cloud. We put our important documents, source control, performance management files, sales reports, and more, in the cloud and are using this exact same solution to secure them.

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

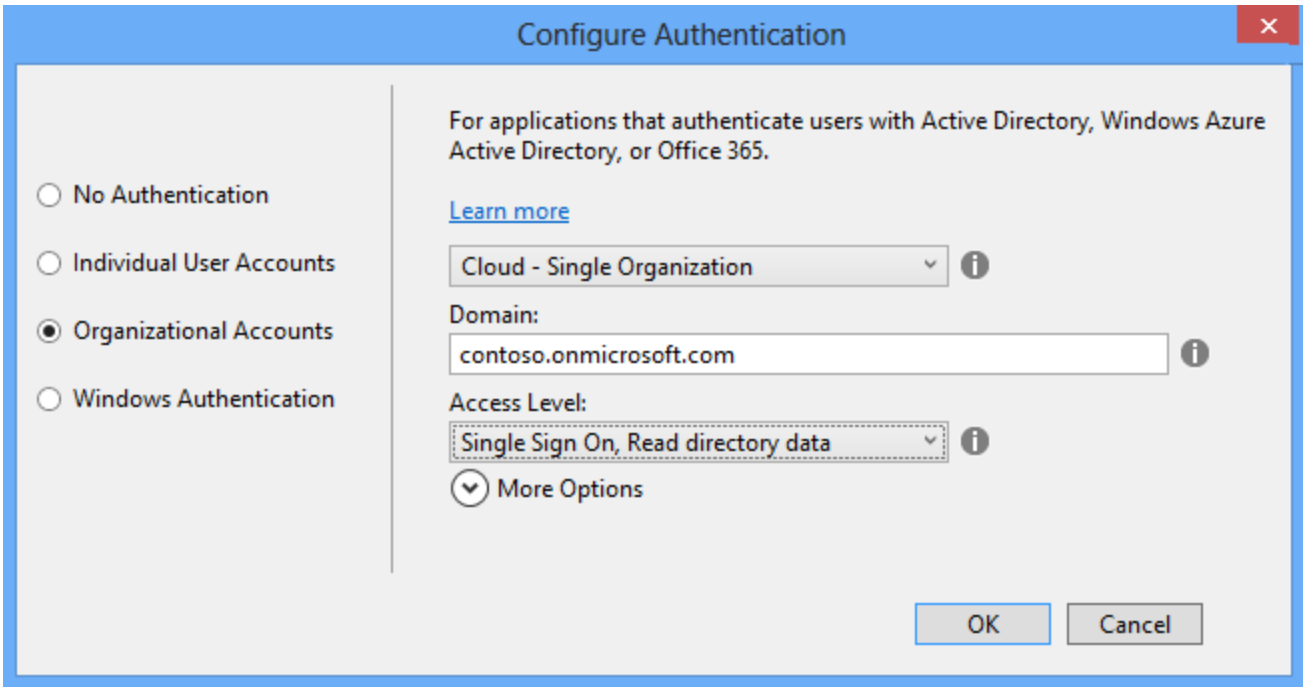
Visual Studio makes it really easy to create an app that uses WAAD for single sign-on, as you can see from a few screen shots.

www.dbooks.org

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 WAAD, you click a **Change Authentication** button.



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

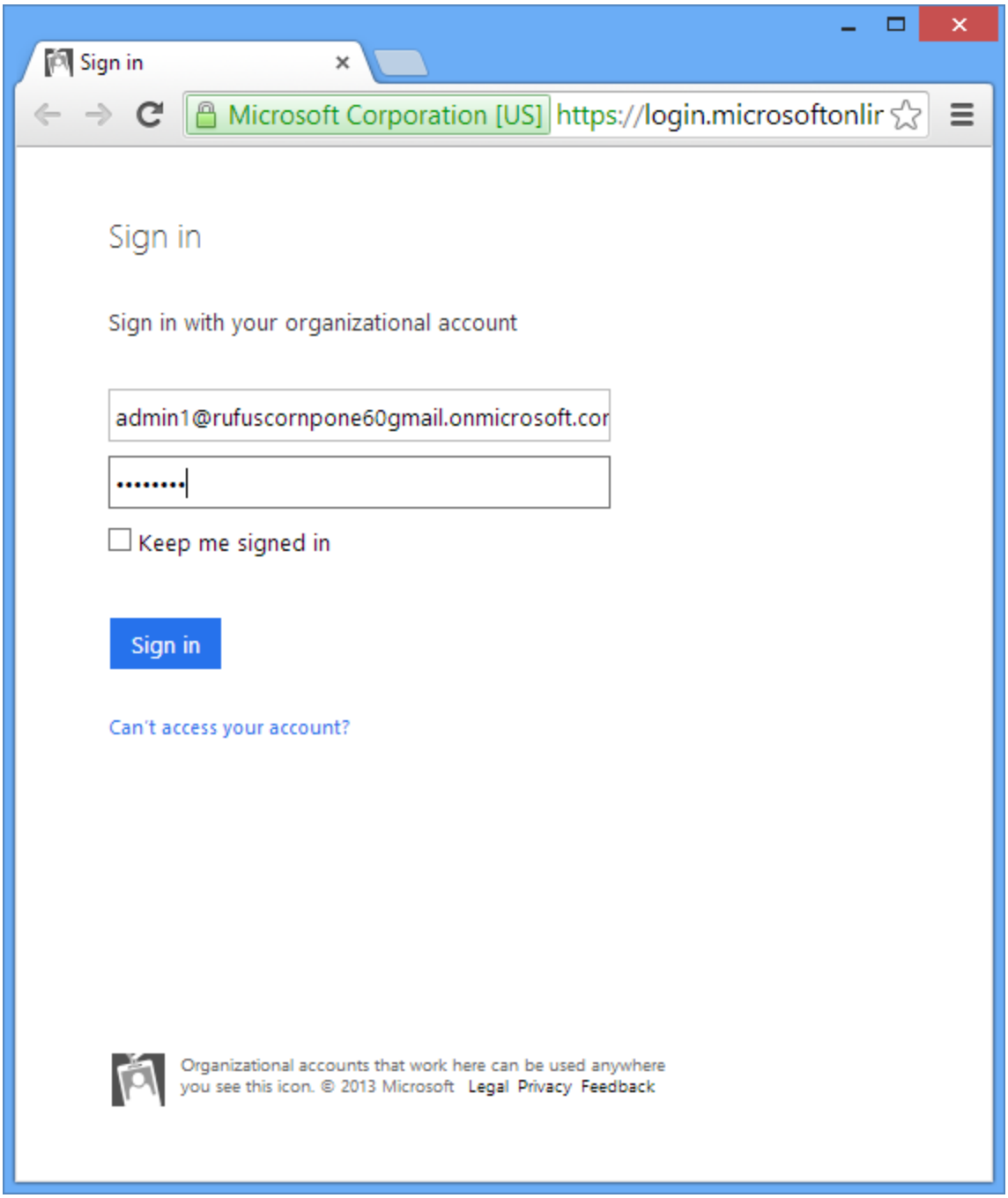


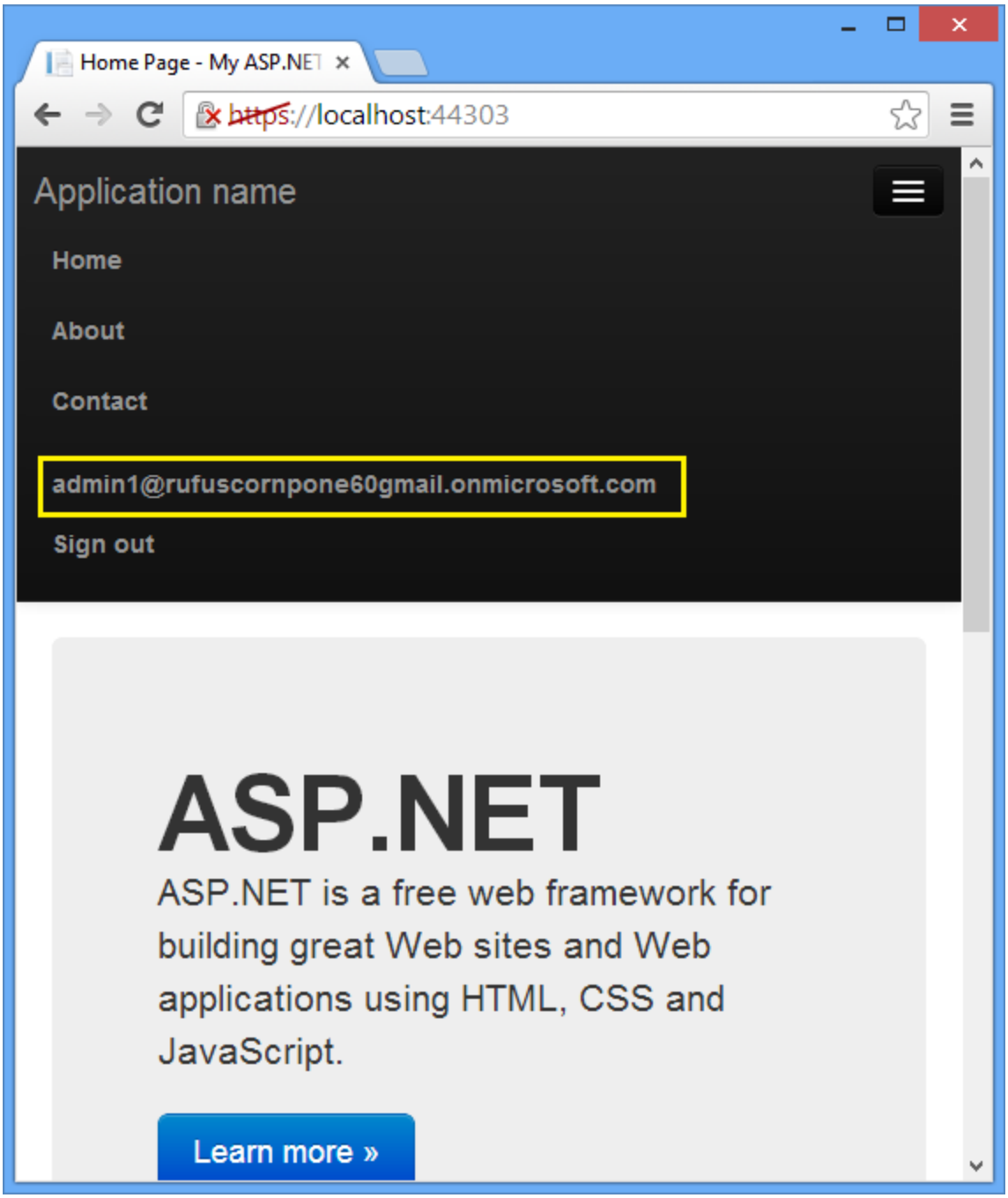
You can also give the app read or read/write permission for directory data. If you do that, it can use the Windows Azure Graph REST API to look up users' phone number, find out if they're in the office, when they last logged on, etc.

That's all you have to do - Visual Studio asks for the credentials for an administrator for your WAAD tenant, and then it configures both your project and your WAAD tenant for the new application.

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

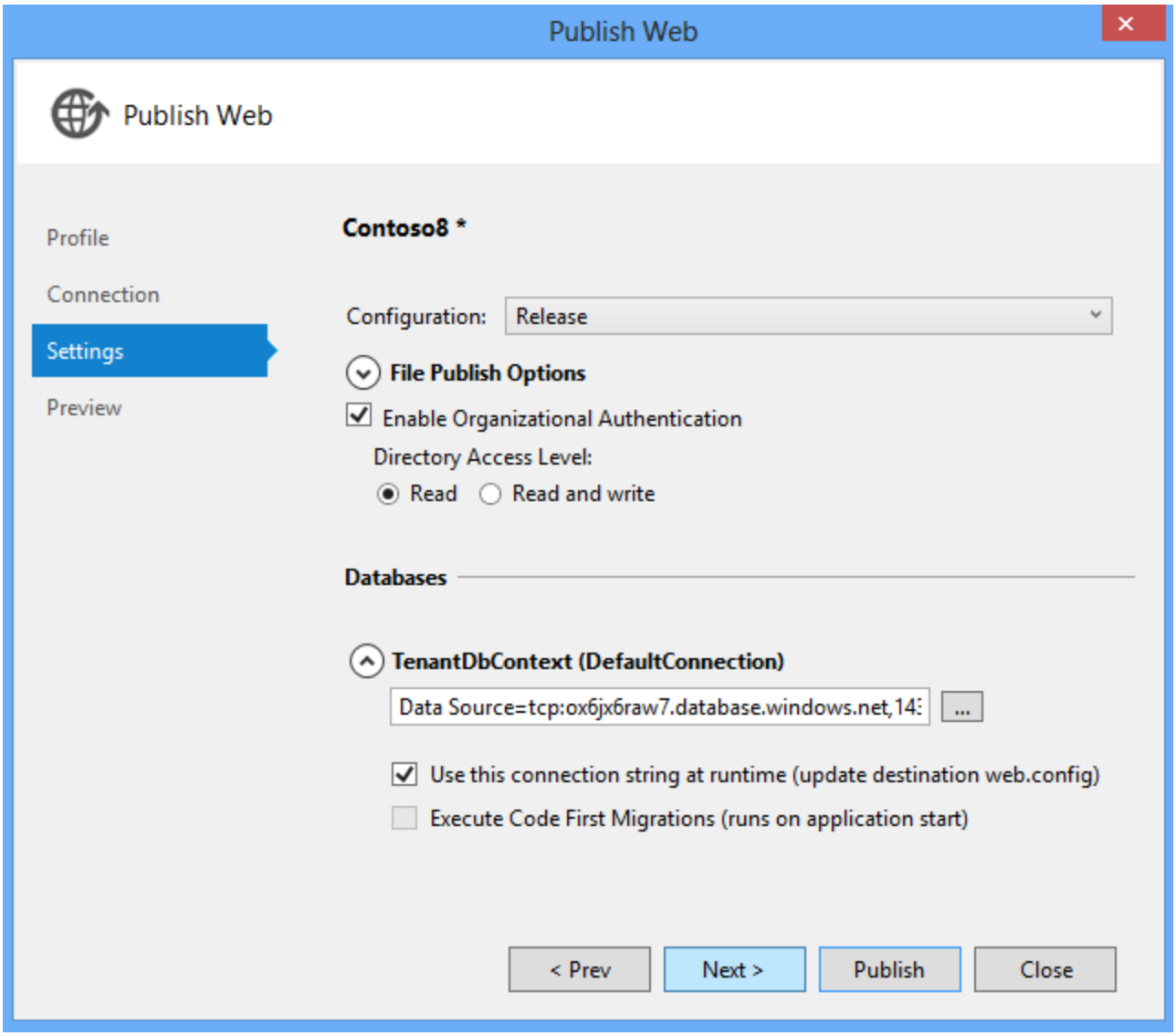
www.dbooks.org


Sign in
Microsoft Corporation [US]
https://login.microsoftonlir
Sign in
Sign in with your organizational account
admin1@rufuscornpone60gmail.onmicrosoft.cor
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 Windows Azure, all you have to do is select an **Enable Organizational Authentication** check box, and once again Visual Studio takes care of all the configuration for you.

www.dbooks.org



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

Summary

In this chapter you saw that Windows 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:

- Windows Azure Active Directory Documentation. Portal page for WAAD documentation on the windowsazure.com site. For step by step tutorials, see the **Develop** section.
- Windows Azure Multi-Factor Authentication. Portal page for documentation about multi-factor authentication in Windows Azure.
- Organizational account authentication options. Explanation of the WAAD authentication options in the Visual Studio 2013 new-project dialog.
- Microsoft Patterns and Practices - Windows Azure Guidance. See 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.

www.dbooks.org

Data Storage Options

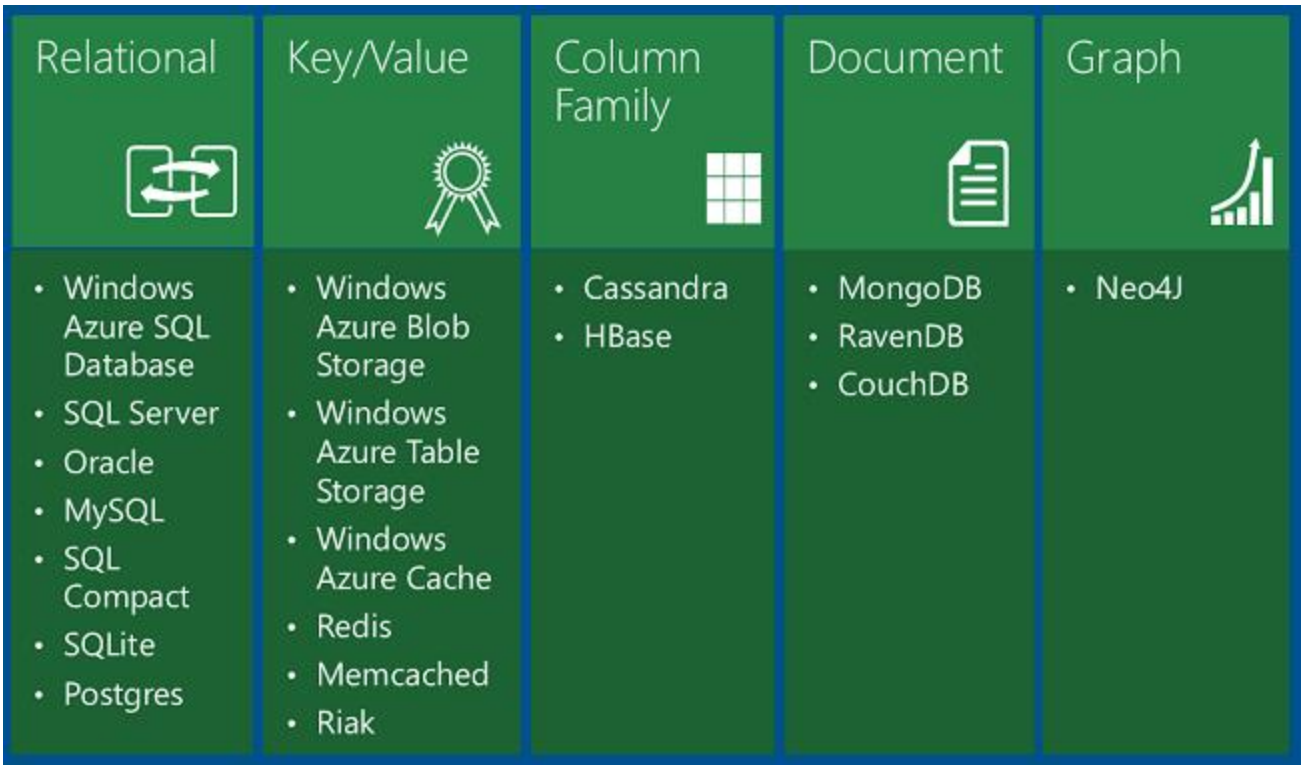
Most people are used to relational databases, and they tend to overlook other data storage options when they're designing a cloud app. The result can be suboptimal performance, high expenses, or worse, because NoSQL (non-relational) databases can handle some tasks more efficiently than relational databases. When customers ask us for help resolving a critical data storage problem, it's often because they have a relational database where one of the NoSQL options would have worked better. In those situations the customer would have been better off if they had implemented the NoSQL solution before deploying the app to production.

On the other hand, it would also be 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 multiple 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 ones that fit your scenario. It's best to be aware of the options available to you and 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 Windows 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 Windows Azure.



The table 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 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.

  Windows 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.

  Windows 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 Excel document but means a collection of named fields and values, any of which could be a child document. For

www.dbooks.org

example, in an order history table an order document might have order number, order date, and customer fields; and the customer field could 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 key/value databases is the ability to query on non-key fields and define secondary indexes to make querying more efficient. This ability makes a document database more suitable for applications that need to retrieve data based on 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 (DOB, gender, etc.). 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 to relational databases, the NoSQL options offer far greater scalability and cost-effectiveness for storage and analysis of unstructured data. The tradeoff is that they don't provide the rich queryability and robust data integrity capabilities of relational databases. NoSQL would work well for IIS log data, which involves high volume with no need for join queries. NoSQL would not work so well for banking transactions, which requires absolute data integrity and involves many relationships to other account-related data.

There is also a newer category of database platform called NewSQL that combines the scalability of a NoSQL database with the queryability and transactional integrity of a relational database. NewSQL databases are designed for distributed storage and query processing, which is often hard to implement in "OldSQL" databases. NuoDB is an example of a NewSQL database that can be used on Windows Azure.

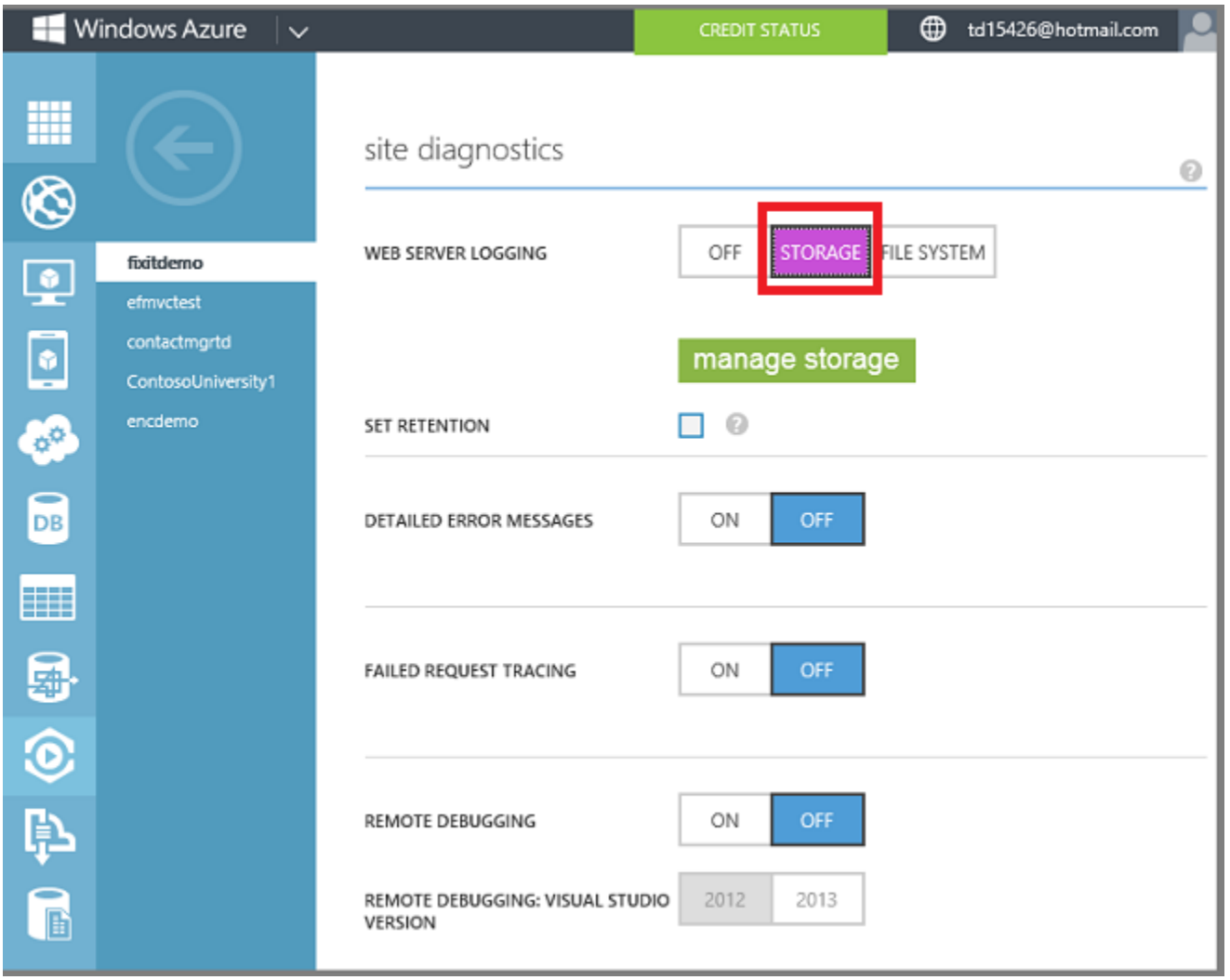
Hadoop and MapReduce

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

- Limit 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, you want to know the makeup of your user base by birth year, so you select only birth years out of your user profile data store.
- Break down the data into parts and send them to different computers for processing. Computer A calculates the number of people with 1950-1959 dates, computer B does 1960-1969, etc. This group of computers is called a *Hadoop cluster*.
- Put the results of each part back together after the processing on the parts is done. You now have a relatively short list of how many people for each birth year and the task of calculating percentages in this overall list is manageable.

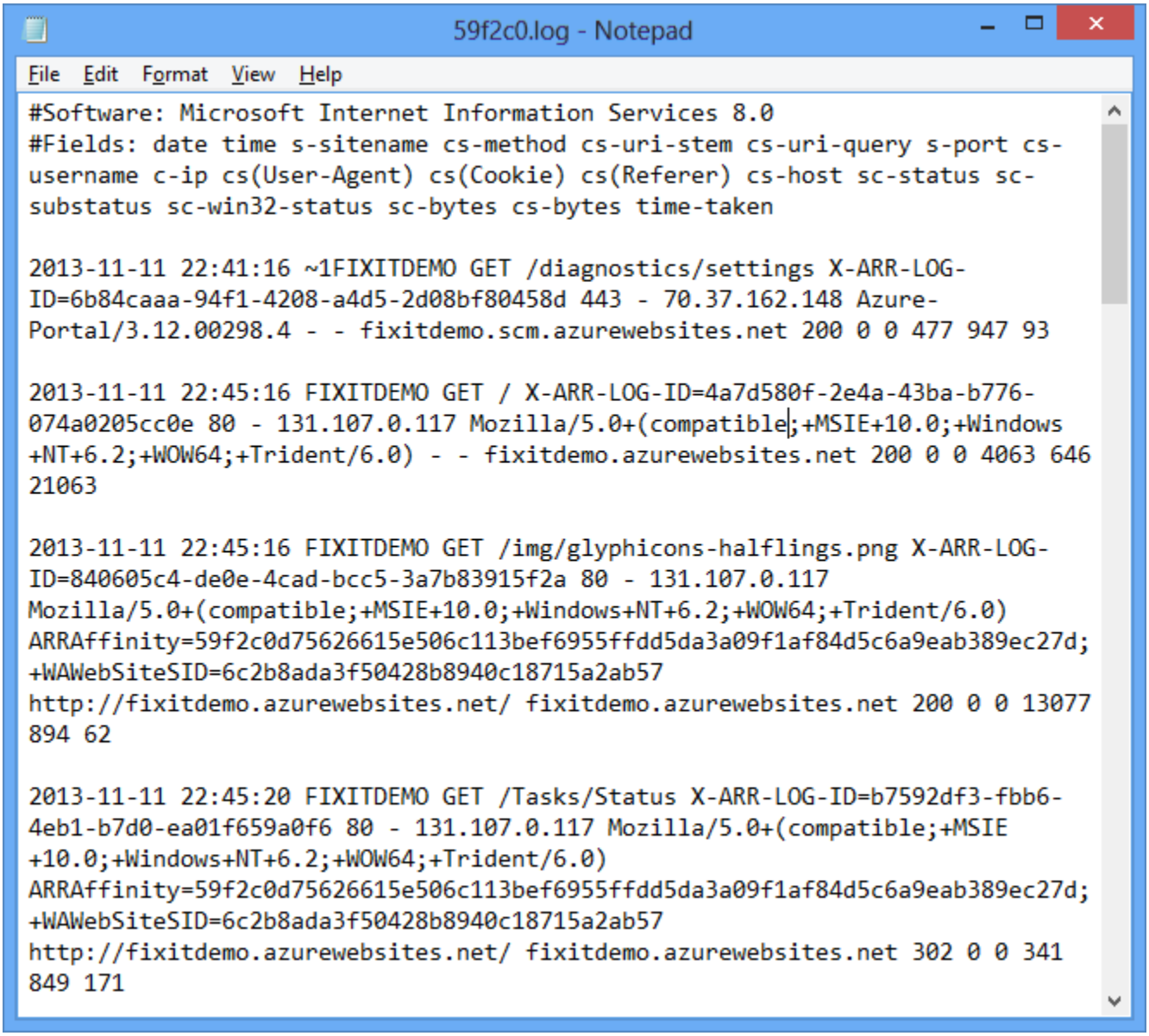
On Windows Azure, HDInsight enables you to process, analyze, and gain new insights from big data using the power of Hadoop. For example, you could use it to analyze web server logs:

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

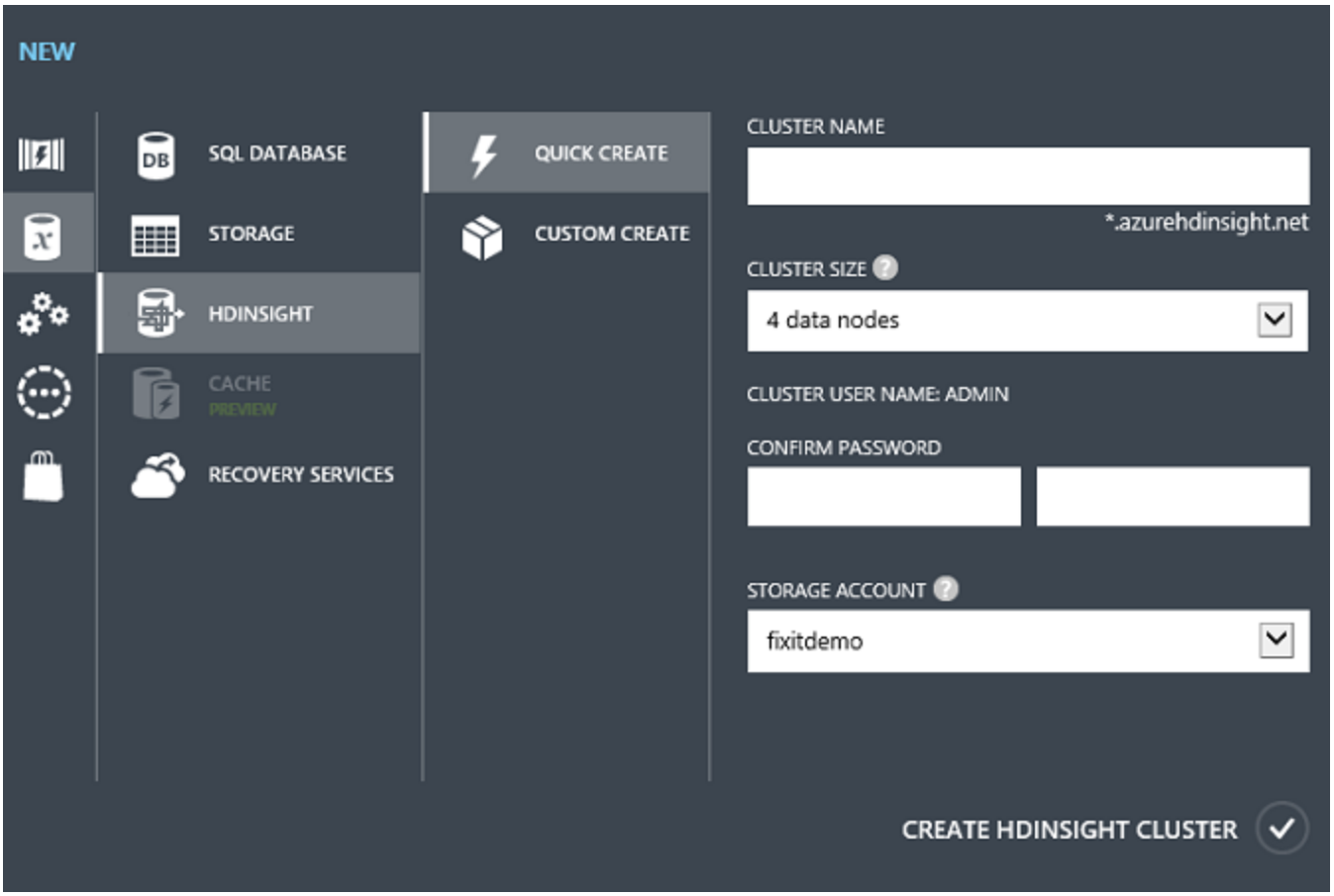


www.dbooks.org

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



- In the portal, click **New** - **Data Services** - **HDInsight** - **Quick Create**, and 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 gives you neighborhood income by IP address, and you can match that against IP address 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 like these to target ads based on the likelihood a customer would be interested in or likely to buy a particular product.

As explained in the Automate Everything chapter, most functions that you can do in the 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 that stores the output data to Blob storage.
- Wait for the job to complete.
- Delete the HDInsight cluster.

www.dbooks.org

- Access the output from Blob storage.

By running a script that does all this, 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.

PaaS means that we manage the hardware and software infrastructure and you just use the service. SQL Database is a PaaS feature of Windows Azure. You ask for databases, and behind the scenes Windows Azure sets up and configures the VMs and sets up the databases on them. You don't have direct access to the VMs and don't have to manage them.

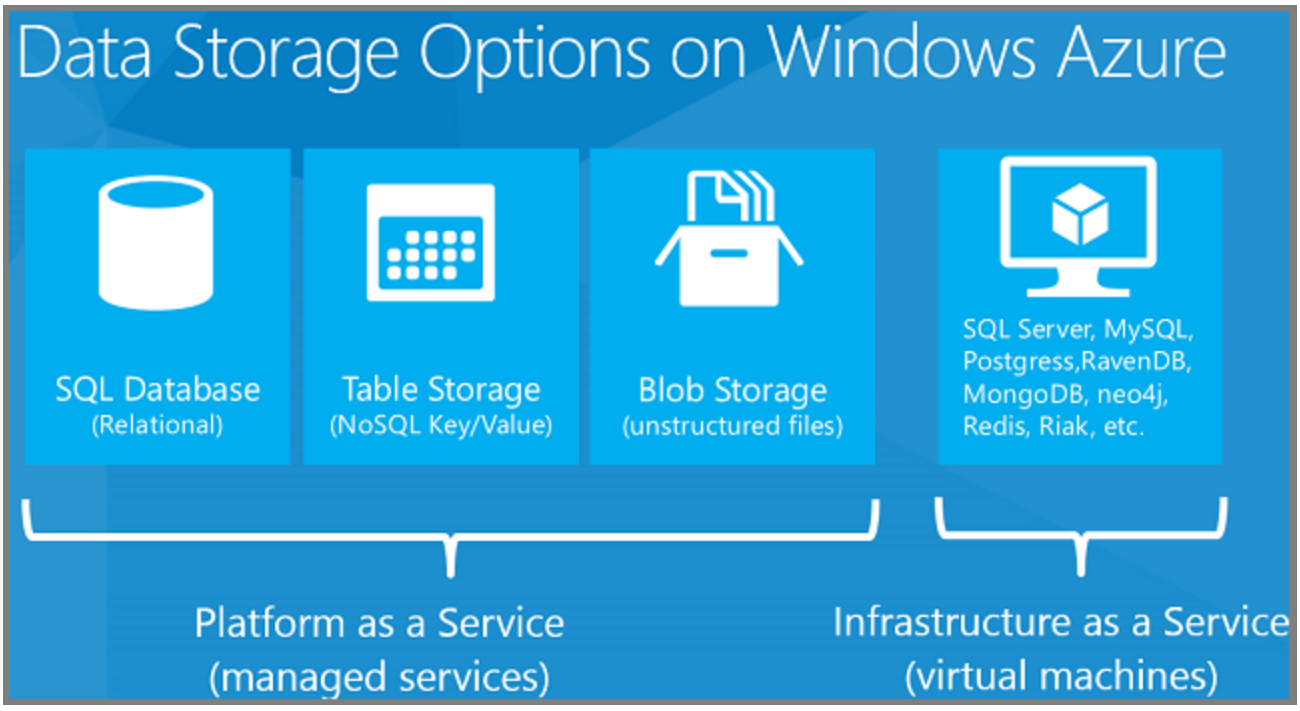
IaaS means that you set up, configure, and manage VMs that run in our data center infrastructure, and you put whatever you want on them. We provide a gallery of pre-configured VM images for common VM configurations. For example, you can install pre-configured VM images for Windows Server 2008, Windows Server 2012, BizTalk Server, Oracle WebLogic Server, Oracle Database, etc.

PaaS data solutions that Windows Azure offers include:

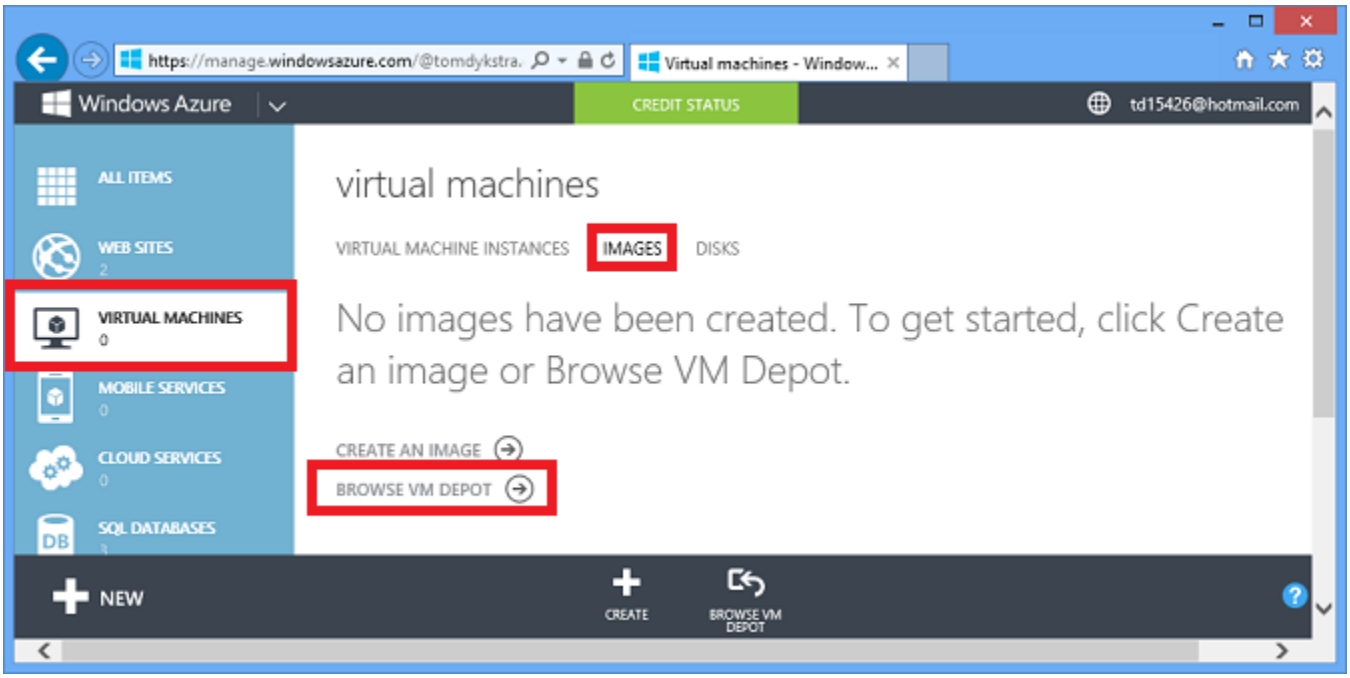
- Windows Azure SQL Database (formerly known as SQL Azure). A cloud relational database based on SQL Server.
- Windows Azure Table storage. A column-oriented NoSQL database.
- Windows Azure Blob storage. File storage in the cloud.

For IaaS, you can run anything 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.

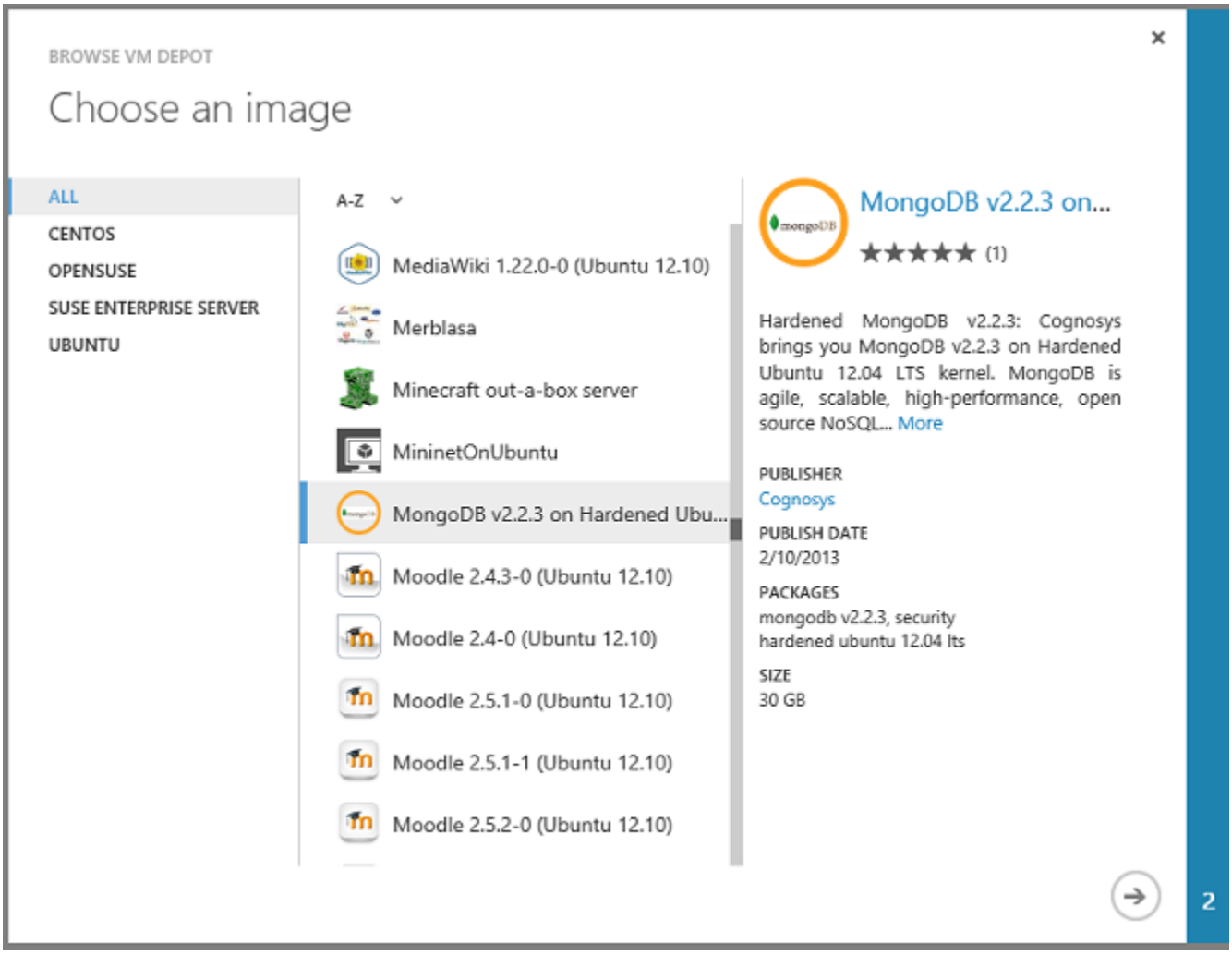


The IaaS option gives you almost unlimited data storage options, and many of them are especially easy to use because you can create VMs using preconfigured images. For example, in the management portal go to **Virtual Machines**, click the **Images** tab, and click **Browse VM Depot**.



www.dbooks.org

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 preinstalled, such as MongoDB, Neo4J, Redis, Cassandra, or CouchDB:



Windows 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 can 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 only pay for what you use.

PaaS data storage options in Windows Azure include offerings by third-party providers. For example, you can choose the MongoLab Add-On from the Windows 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 this technology is the answer, the first thing to ask is "What is the question?", because different solutions are optimized for different things. There are definite advantages to the relational model; that's why it's been around for so long. But there are also down-sides to SQL that can be addressed with a NoSQL solution.

Often what we see work best is a compositional approach, where you use SQL and NoSQL in a single solution. Even when people say they're embracing NoSQL, if you drill into what they're doing you often find that they're using several different NoSQL frameworks: they're using CouchDB, and Redis, and Riak for different things. Even Facebook, which uses NoSQL extensively, uses different NoSQL frameworks for different parts of the service. The flexibility to mix and match data storage approaches is one of the things that's nice about the cloud, because 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 given a key value but not so good for complex queries. For a user profile data store where you are always getting the data for one particular user, a key/value data store could work well; for a product catalog where 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, etc., be executed server-side? If I run SELECT COUNT(*) from a table in SQL, it will very efficiently do

www.dbooks.org

	all the work on the server and return the number I'm looking for. If I want the same calculation from a NoSQL data store that doesn't support aggregation, this is an inefficient "unbounded query" and will probably time out. Even if the query succeeds I have to retrieve all of the data from the server to the client and count the rows on the client. • What languages or types of expressions can be used? With a relational database I can use SQL. With some NoSQL databases such as Windows Azure Table storage, I'll be using OData, and all I can do is filter on 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/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 like Windows 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 Windows Azure SQL Database is 150 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 will need to keep 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? Table Storage and SQL Database are easy to set up on Windows Azure. Platforms that aren't built-in Windows Azure PaaS solutions require more effort.
API Support	• Is an API available that makes it easy to work with the platform? For the Windows Azure Table Service there's an SDK with a .NET API that supports the .NET 4.5 asynchronous programming model. If you're writing a .NET app, it'll be much easier to write and test code for the Windows Azure Table Service compared to another 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 in order 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 Windows 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 get 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 Windows Azure Table Service or Windows Azure SQL Database. If you don't need the rich querying facilities of SQL Database, you might choose Windows Azure Tables in part because it costs much less for a given amount of storage.

What we generally recommend is know the answer to the questions in each of these categories before you choose your data storage solutions.

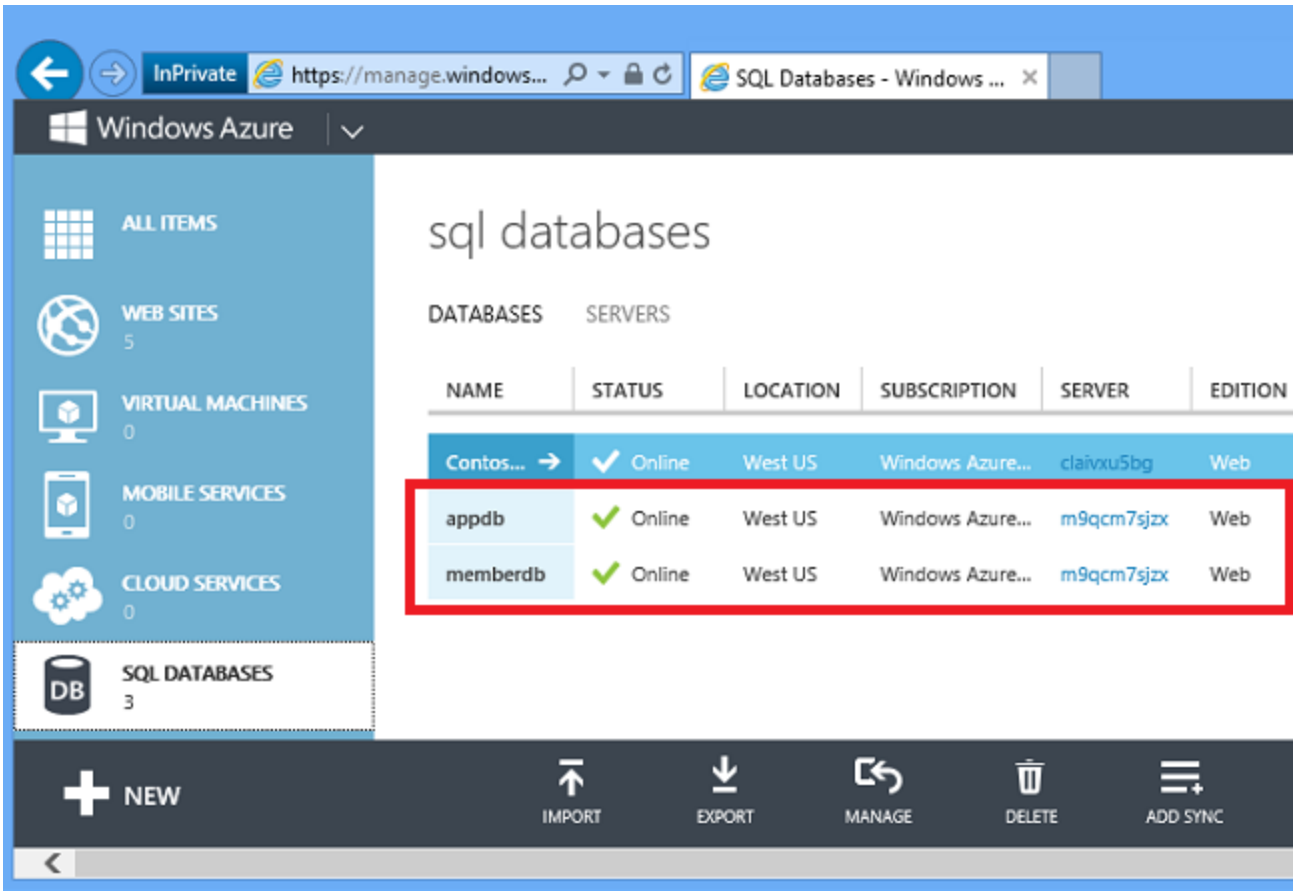
In addition, your workload might have specific requirements that some platforms can support better than others. For example:

- 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, the data includes PII (personally identifiable information) but you have to be able to make 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 integrating with your on-premises storage.

Demo – using SQL Database in Windows Azure

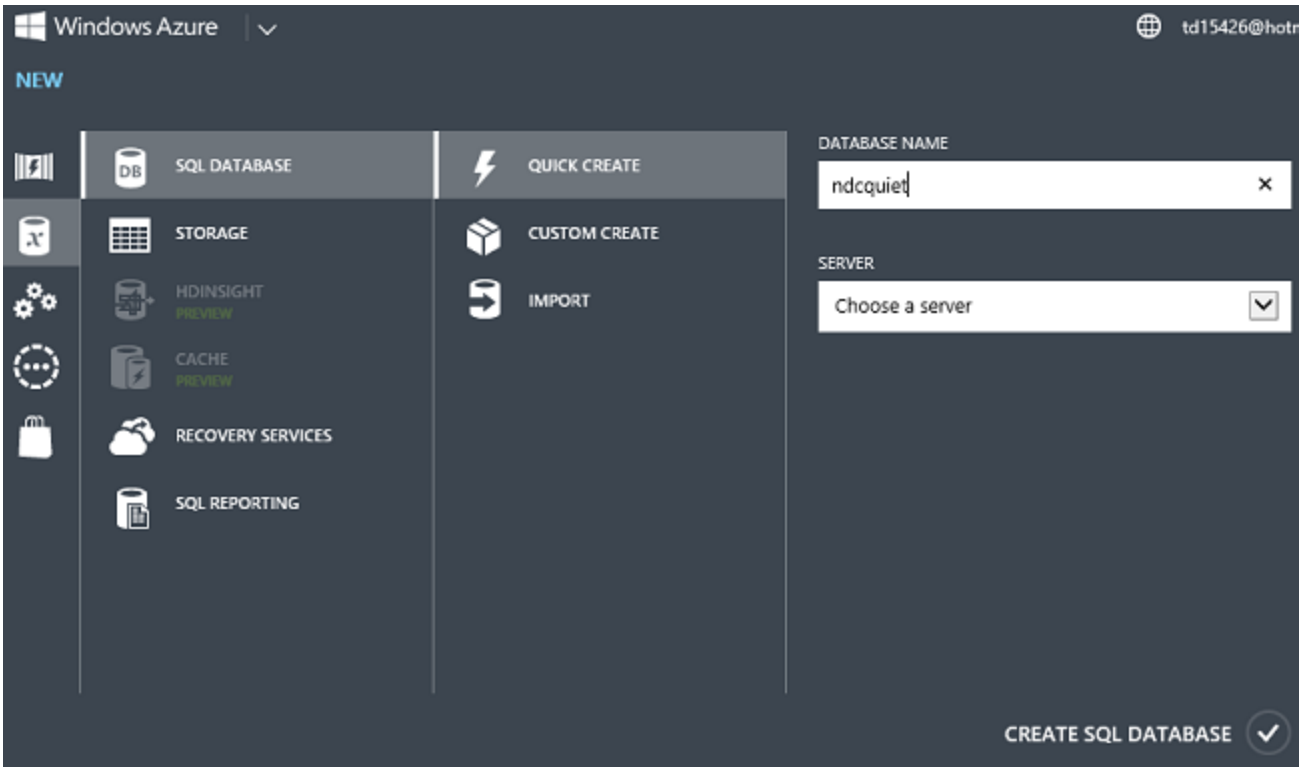
The Fix It app uses a relational database to store tasks. The environment creation Windows PowerShell script shown in the Automate Everything chapter creates two SQL Database instances. You can see these in the portal by clicking the **SQL Databases** tab.

www.dbooks.org



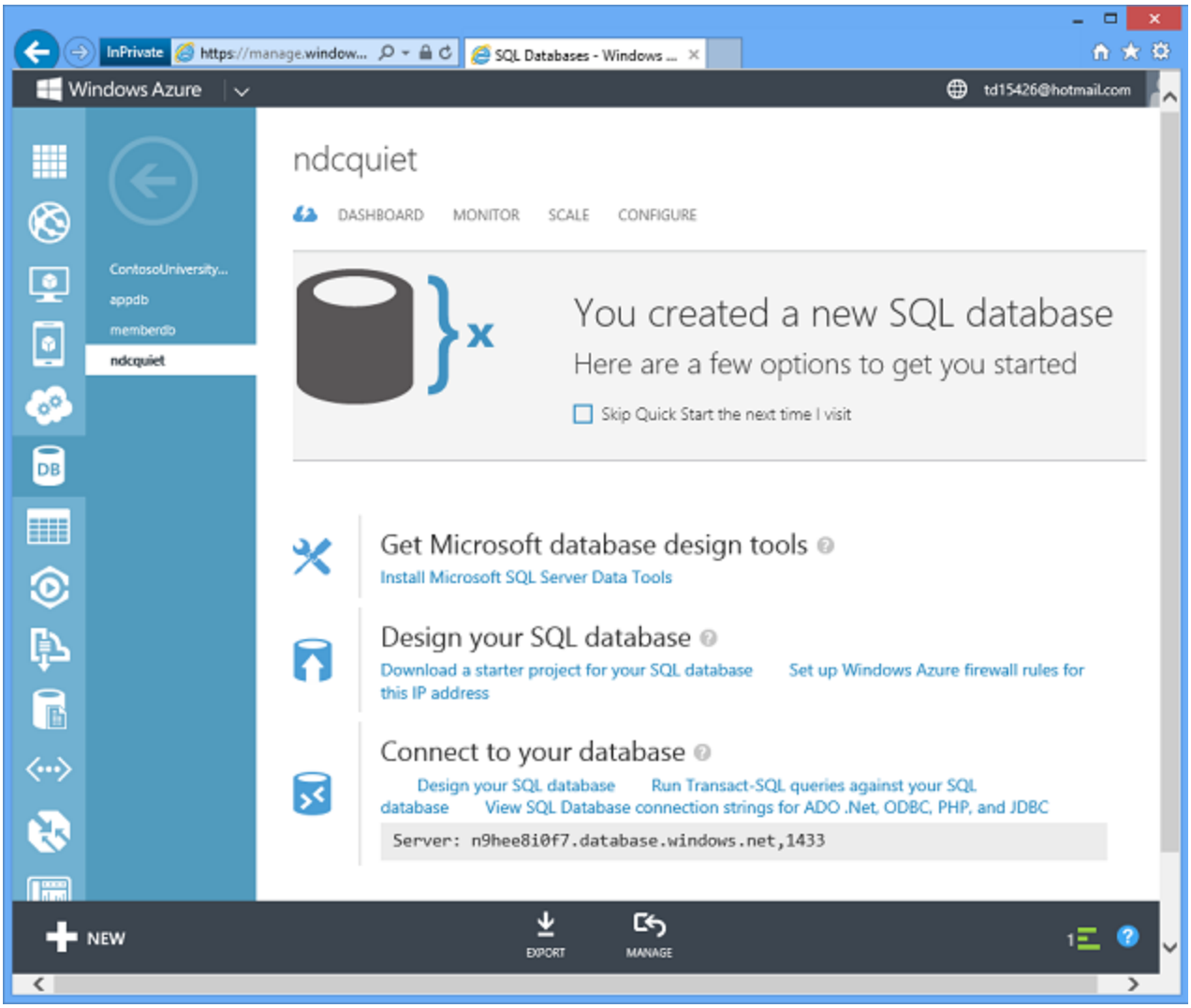
It's also easy to create databases by using the portal.

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 click **Create SQL Database**.



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

www.dbooks.org



So Windows Azure does in a few seconds what it may take you a day or a week or longer to accomplish in the on-premises environment. And since you can just as easily create databases automatically in 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.

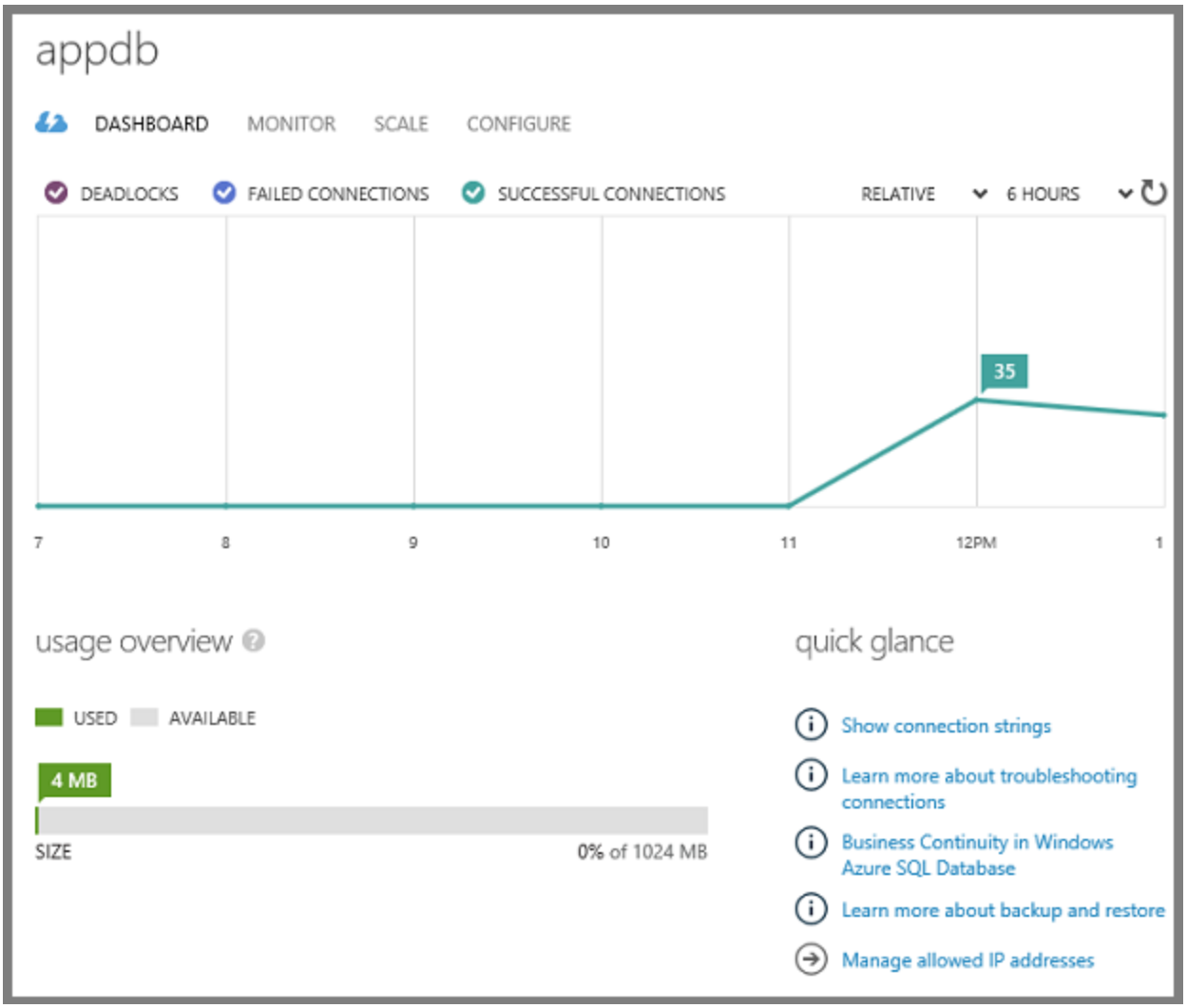
This is an example of our Platform-as-a-Service model. You don't have to manage the servers, we do it. You don't have to worry about backups, we do it. It's running in high availability – the data in the database is replicated across three servers automatically. If a machine dies, we automatically fail over and you lose no data. 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 can immediately start using the new database.

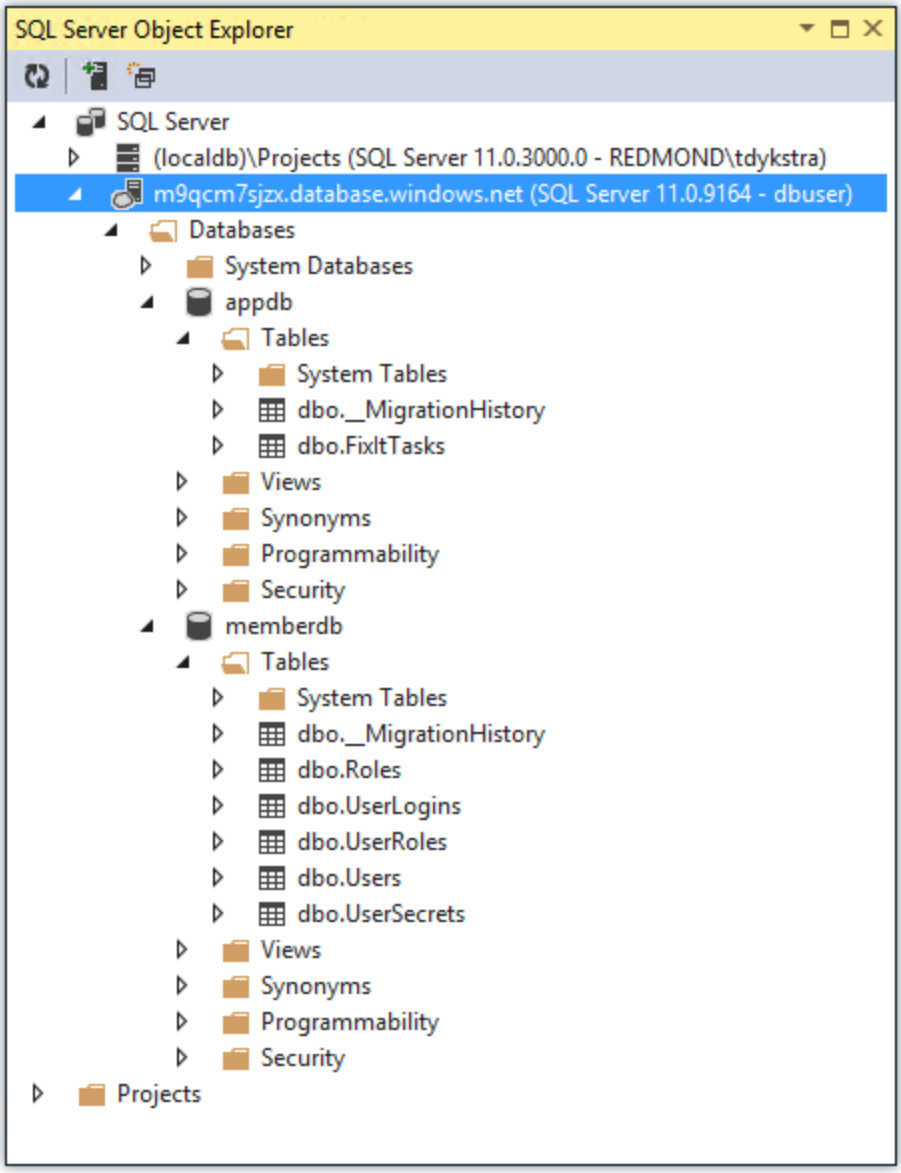


The Dashboard shows you connection history and amount of storage used.

www.dbooks.org



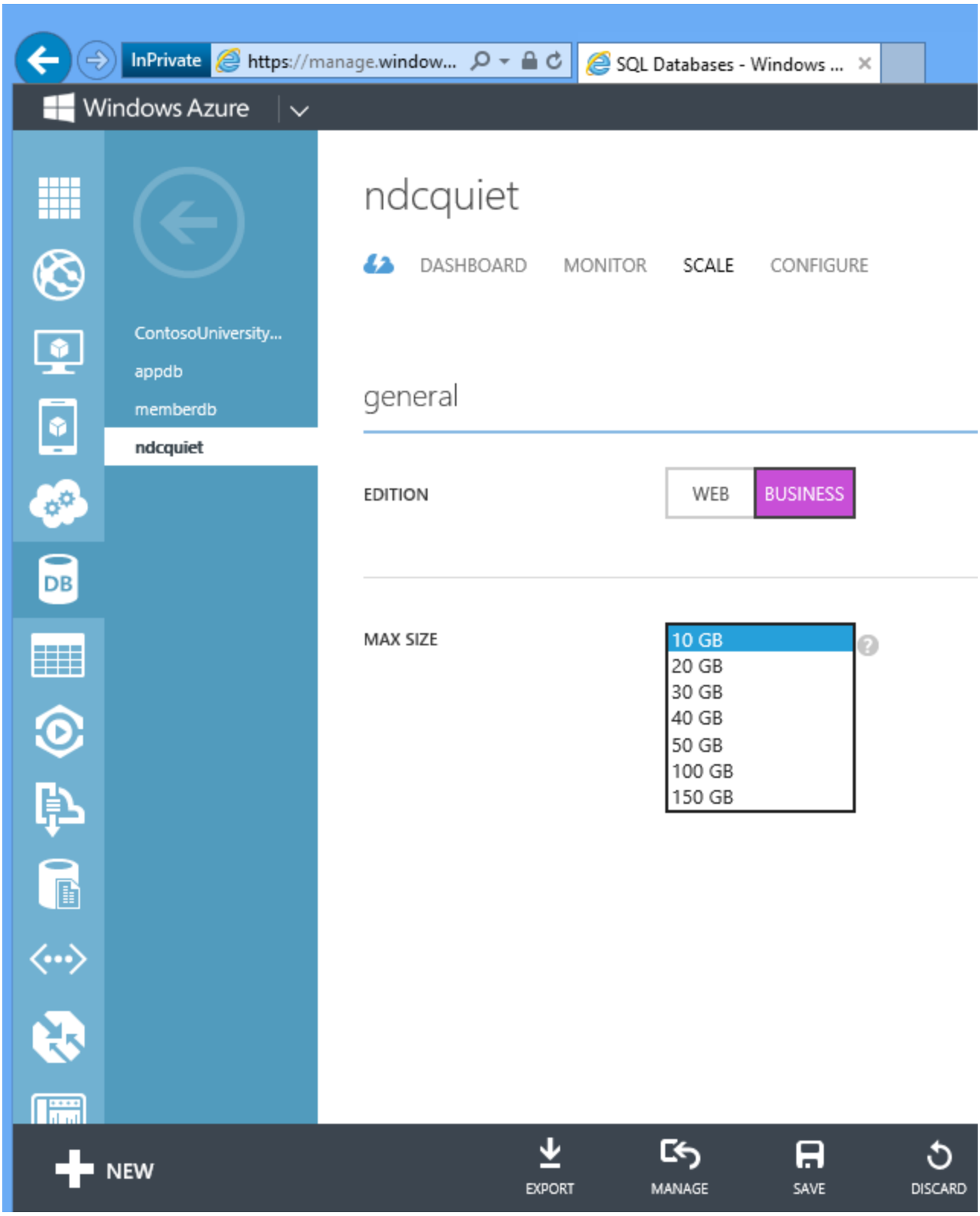
You can manage databases in the portal or by using SQL Server tools you're already familiar with, including SQL Server Management Studio (SSMS) and the Visual Studio tools SQL Server Object Explorer (SSOX) and Server Explorer.



Another nice thing 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 gig. If you want to scale it up to 150 gig, you can just go into the portal and change that setting, or execute a REST API command, and in seconds you have a 150 gig database that you can deploy data into.

www.dbooks.org



That's the power of the cloud to stand up infrastructure quickly and easily and 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 it and scale it. You saw earlier how to provision the databases through Windows PowerShell scripts, and now you've also seen how easy it is to do in the portal.

Entity Framework versus direct database access using ADO.NET

The Fix It app accesses these databases by using the Entity Framework, 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 rather than 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):

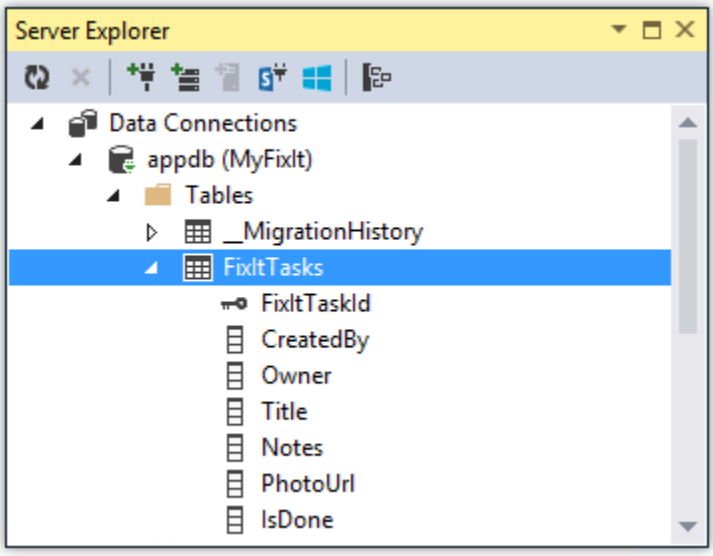
```
  <connectionStrings>
```


www.dbooks.org

```
    <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 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 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 there's also some timing and error logging code here, we'll look at that later in the Monitoring and Telemetry chapter.

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

A nice thing about SQL Server and Windows Azure SQL Database is that the core programming model for both of them 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 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

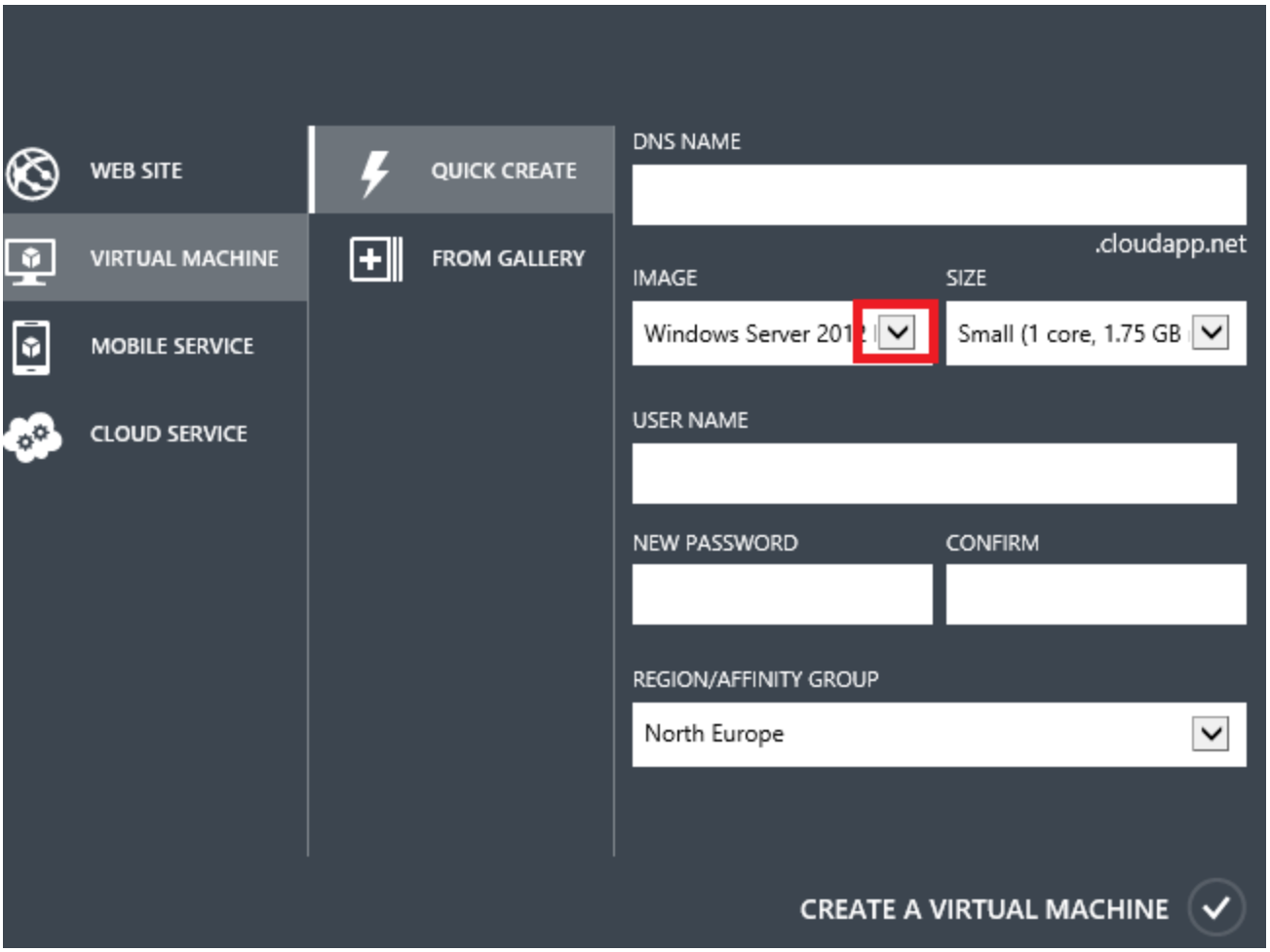
www.dbooks.org

better solution. Because a SQL Server database runs on a dedicated VM, it has more resources available to it than 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).

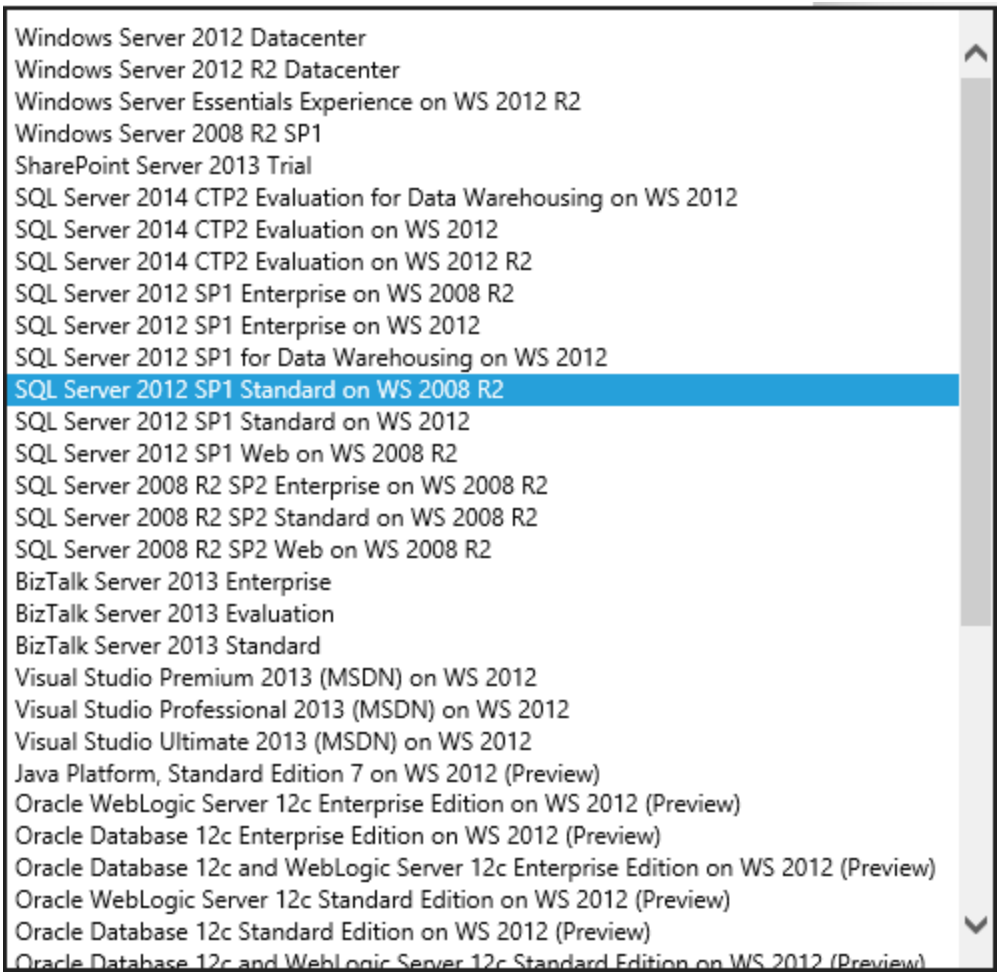
Here are some guidelines on how to choose between the two models.

Windows Azure SQL Database (PaaS)	SQL Server in a Virtual Machine (IaaS)
Pros • You don't have to create or manage VMs, update or patch OS or SQL; Windows 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 (<=150 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 2+ VMs, with VM-level SLA. • You have complete control over how SQL is managed. • Can re-use SQL licenses you already own, or pay by the hour for one. • Good for handling fewer but larger (1 TB+) databases.
Cons • Some feature gaps compared to on-premises SQL Server (lack of CLR integration, TDE, compression support, SQL Server Reporting Services, etc.) • Database size limit of 150GB.	**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 8000 (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.



www.dbooks.org



When you create a VM with a SQL Server image, we pro-rate the SQL Server license cost by the hour based on your usage of the VM. If you have a project that's only going to run 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 here in order to pick approaches that will continue to work well when your application grows. The next chapter will explain 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. E-book by Microsoft Patterns and Practices that goes in depth into the different kinds of data stores available for cloud applications.
- Microsoft Patterns and Practices - Windows Azure Guidance. See Data Consistency Primer, Data Replication and Synchronization Guidance, Index Table pattern, Materialized View pattern.
- BASE: An Acid Alternative. Article about tradeoffs 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:

- Choosing between SQL Server in Windows Azure VM & Windows Azure SQL Database. A comprehensive treatment of the subject, but was published in early 2013 before the introduction of SQL Database Premium level.
- Premium Preview for SQL Database Guidance. An introduction to SQL Database Premium, and guidance on when to choose it over the SQL Database Web and Business editions.
- Guidelines and Limitations (Windows Azure SQL Database). 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 Windows Azure Virtual Machines. Portal page that links to documentation about running SQL Server in Windows Azure.
- Scott Guthrie explains SQL Databases in Azure. 6-minute video introduction to SQL Database by Scott Guthrie.

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

- Getting Started with EF 6 using MVC 5. Nine-part tutorial series that walks you through building an MVC app that uses EF and deploys the database to Windows Azure and SQL Database.
- ASP.NET Web Deployment using Visual Studio. Twelve-part tutorial series 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 a Windows Azure Web Site. Step-by-step tutorial that walks you through creating a web app that uses authentication, stores application tables in the membership database, modifies the database schema, and deploys the app to Windows Azure.
- ASP.NET Data Access Content Map. Links to resources for working with EF and SQL Database.

Using MongoDB on Windows Azure:

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

HDInsight (Hadoop on Windows Azure):

www.dbooks.org

- HDInsight. Portal to HDInsight documentation on the http://windowsazure.com site.
- Hadoop and HDInsight: Big Data in Windows Azure. MSDN Magazine article by Bruno Terkaly and Ricardo Villalobos, introducing Hadoop on Windows Azure.
- Microsoft Patterns and Practices - Windows Azure Guidance. See MapReduce pattern.

Data Partitioning Strategies

Earlier we saw how easy it is to scale the web tier of a cloud application, by adding and removing web servers. But if they're all hitting the same data store, your application's bottleneck moves from the front-end to the back-end, 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 hard up front 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

In order 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 that customers will be uploading images and videos into?
- 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 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.

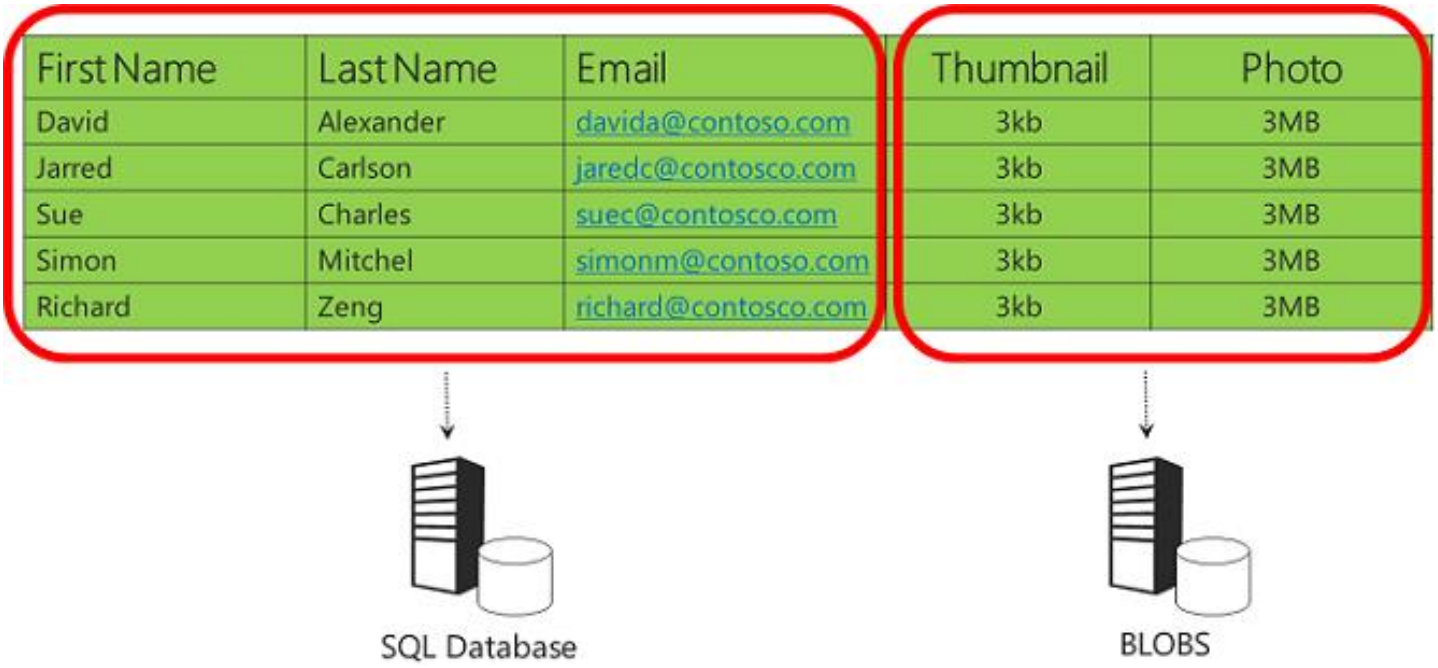
For example, suppose my app stores data about people, including images:

www.dbooks.org

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contosco.com	3kb	3MB
Sue	Charles	suec@contosco.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contosco.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, while 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 back-up 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 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.



Storing images in Blob storage instead of 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

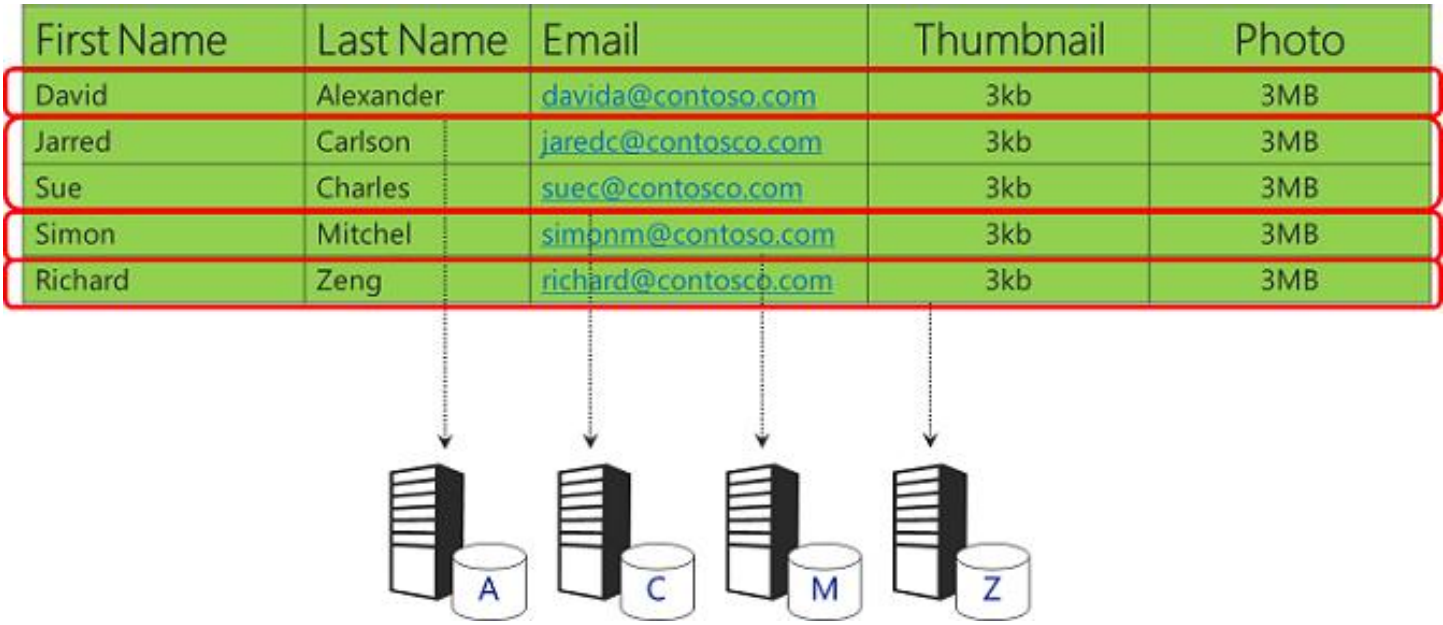
back-up and restore of data stored outside of the relational database: all that is handled for you automatically by the Blob storage service.

This is the partitioning approach we implemented in the Fix It app, and we'll look at the code for that in the Blob storage chapter. Without this partitioning scheme, and assuming an average image size of 3 megabytes, the Fix It app would only be able to store about 40,000 tasks before hitting the maximum database size of 150 gigabytes. After removing the images, the database can store 10 times as many tasks; you can go much longer 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.



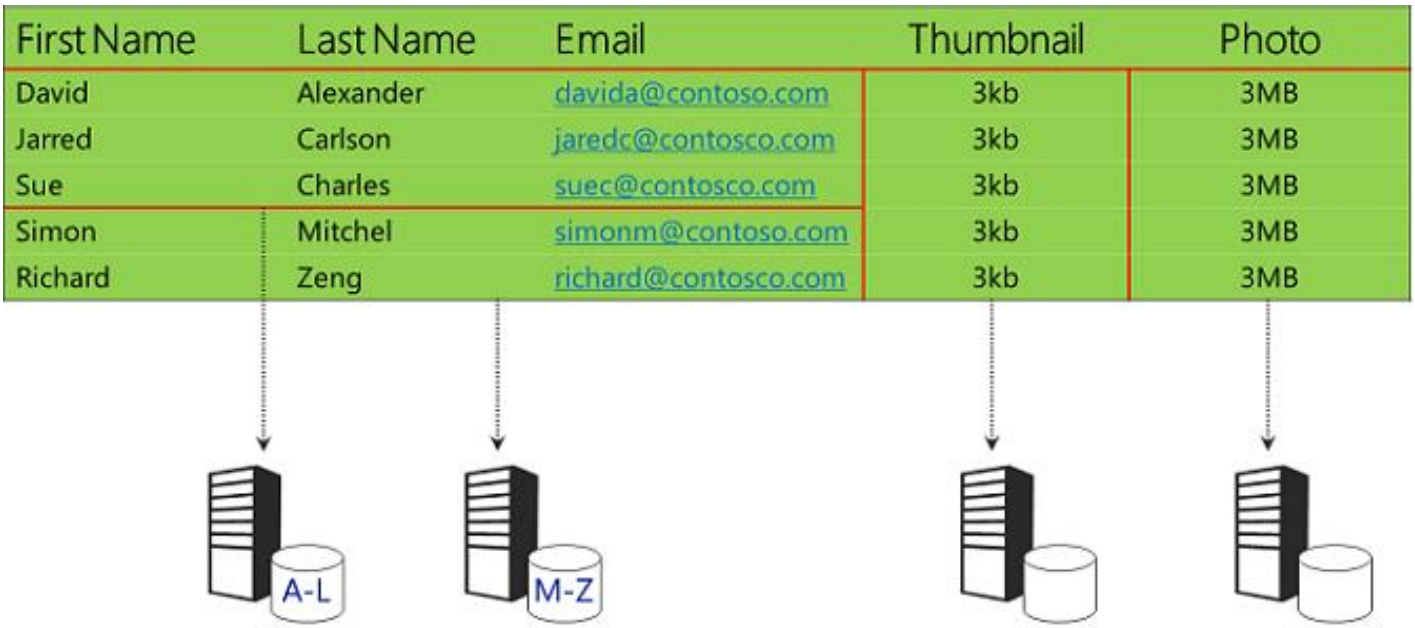
You want to be very careful about your sharding scheme to make sure that data is evenly distributed in order to avoid hot spots. This simple example using the first letter of the last name doesn't meet that requirement, because a lot of people have last names that start with certain common letters. You'd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

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

www.dbooks.org

Hybrid partitioning

You can combine vertical and horizontal partitioning. For example, in the example data 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're planning for them before you go to production. Many people who didn't do that wish they had later. On average our Customer Advisory Team (CAT) team gets panicked phone calls about once a month from customers whose apps are taking off in a really big way, and they didn't do this planning. And 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. We end up going through herculean efforts to help the customer partition their data on the fly with no down time. It's very exciting and very scary, and not something you want to be involved in if you can avoid it! Thinking about this 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 in the cloud 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 you can incrementally add capacity as you need it, and you're only paying for as much as you're using when you use it.

In the next chapter we'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 Windows Azure Cloud Services. White paper by Mark Simms and Michael Thomassy.
- Microsoft Patterns and Practices - Windows Azure Guidance. 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 Microsoft Customer Advisory Team (CAT) 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 Windows Azure blog.

www.dbooks.org

Unstructured Blob Storage

In the previous chapter we looked at partitioning schemes and explained how the Fix It app stores images in the Windows Azure Storage Blob service, and other task data in Windows 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?

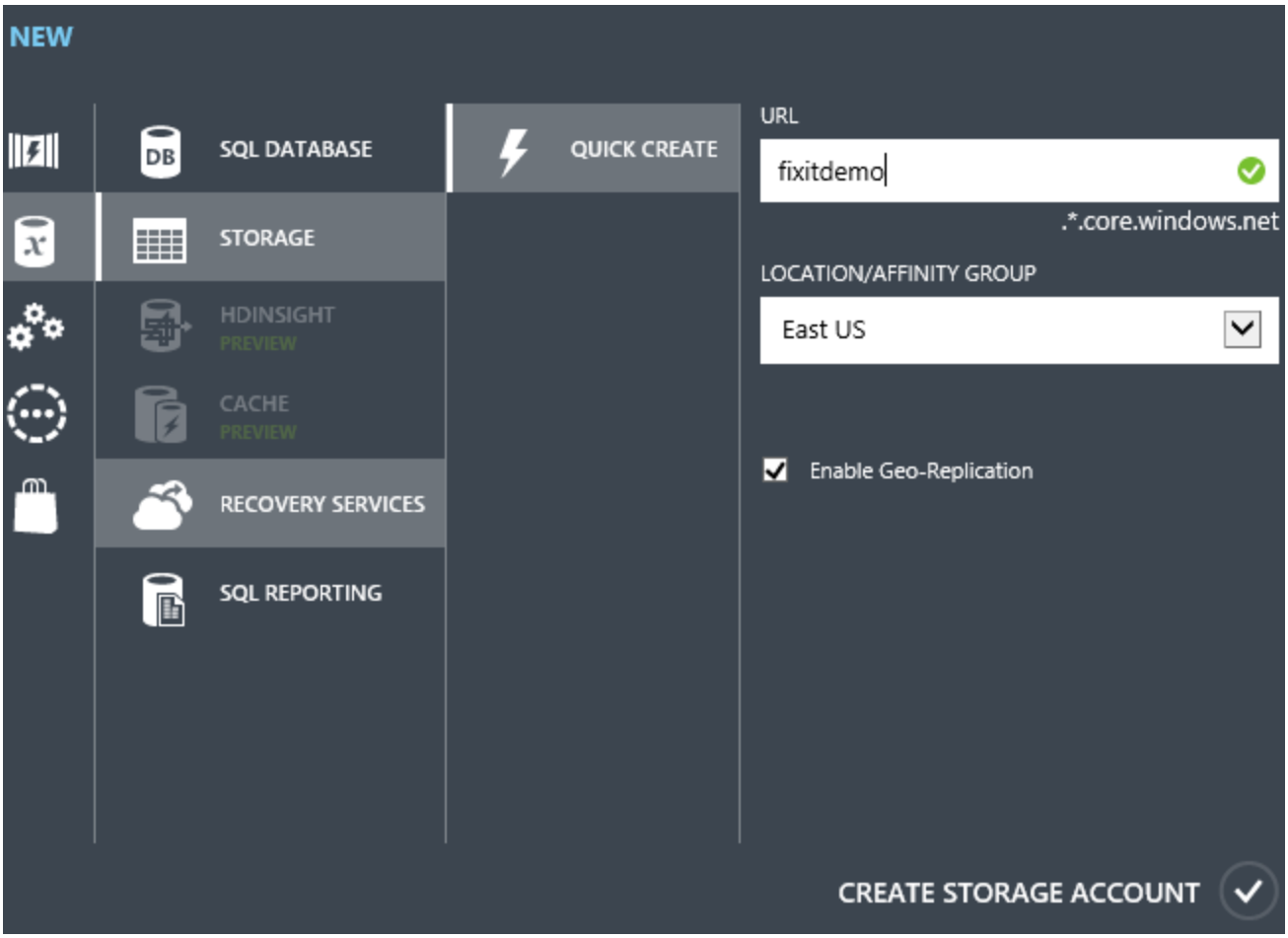
The Windows Azure Storage Blob service provides a way to store files in the cloud. The Blob service has a number of advantages over storing files in a local network file system:

- It's highly scalable. A single Storage account can store 100 terabytes, and you can have multiple Storage accounts. Some of the biggest Windows Azure customers store hundreds of petabytes. Microsoft SkyDrive 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% uptime.
- It's a platform-as-a-service (PaaS) feature of Windows Azure, which means you just store and retrieve files, paying only for the actual amount of storage you use, and Windows 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 others.
- 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 they can accessed only by authorized users, or you can provide temporary access tokens that makes them available to someone only for a limited period of time.

Anytime you're building an app for Windows 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, etc. -- consider the Blob service.

Creating a Storage account

To get started with the Blob service you create a Storage account in Windows 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 web site.



Notice the **Enable Geo-Replication** check box under the location drop-down list. You pick the primary region where you want to store the content, and if this option is selected, Windows Azure creates backups of all your data in a different data center in another region of the country. For example, if you choose the Western US data center, when you store a file it goes to the Western US data center, but in the background Windows 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.

Windows Azure won't replicate data across geo-political boundaries: if your primary location is in the U.S., your files are only replicated to another region within the U.S.; if your primary location is Australia, your files are only replicated to another data center in Australia.

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

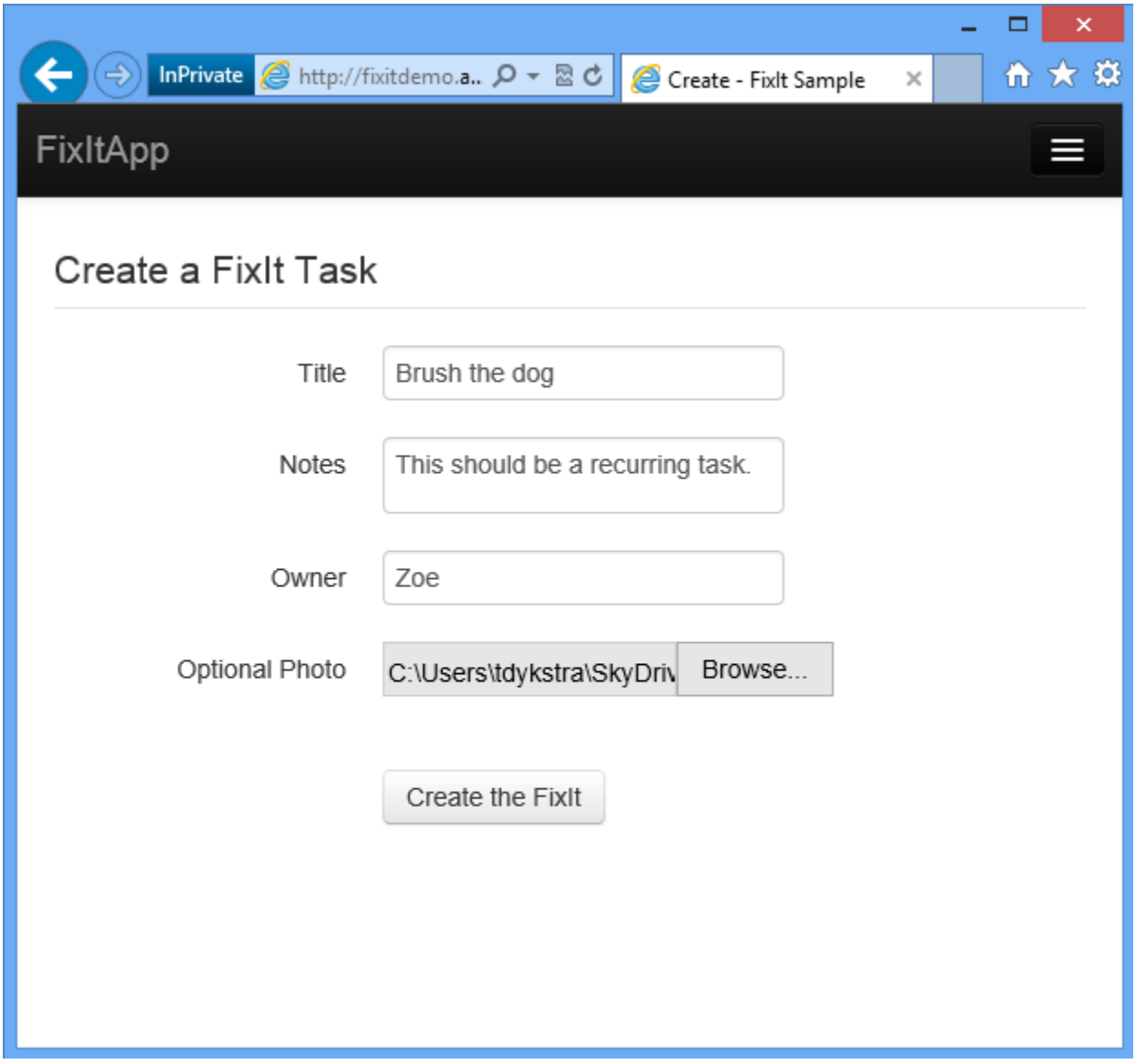
```
# Create a new storage account
New-AzureStorageAccount -StorageAccountName $Name -Location $Location -
Verbose
```

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

www.dbooks.org

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

In order 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 the Automate Everything chapter 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");
```


www.dbooks.org

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));
```


www.dbooks.org

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 web pages to display the uploaded image.

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

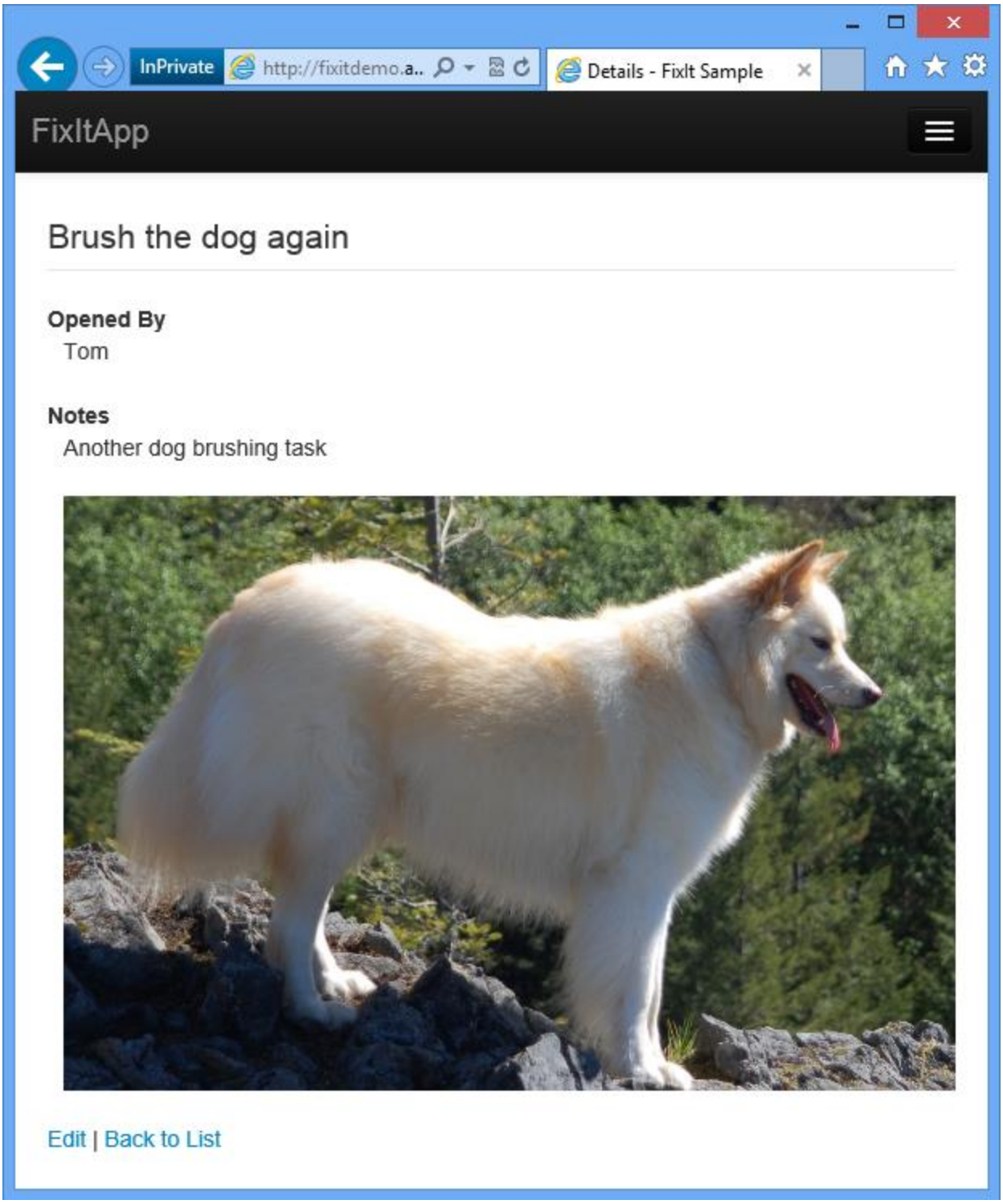
This URL is stored in the database as one of the columns of the FixItTask table.

```
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 9 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.

www.dbooks.org

```
<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 displays, 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 otherwise would be, 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 up to 100 TB in a storage account, and the storage cost is much less expensive than SQL Database storage, at about 9 cents per gigabyte 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 Windows Azure BLOB Storage. Blog by Mike Wood.
- How to use the Windows Azure Blob Storage Service in .NET. Official documentation on the WindowsAzure.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, etc.
- 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 Microsoft Customer Advisory Team (CAT) experience with actual customers. For a discussion of Windows Azure Storage service and blobs, see episode 5 starting at 35:13.
- Microsoft Patterns and Practices - Windows Azure Guidance. See Valet Key pattern.

www.dbooks.org

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 get and how much time you have to spend analyzing and fixing problems.

Types of failures

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

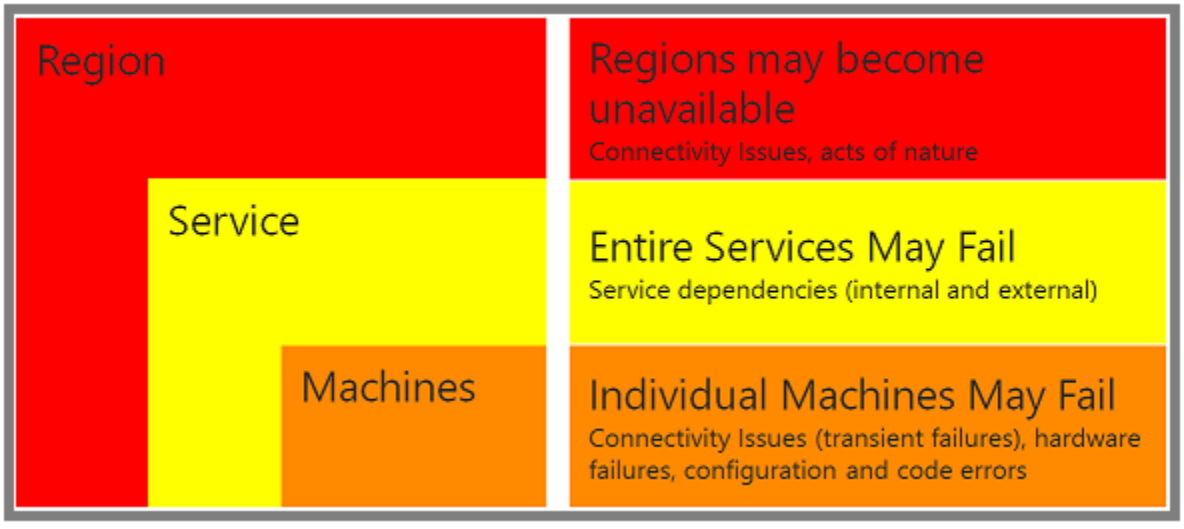
- Transient, self-healing failures such as intermittent network connectivity issues.
- Enduring failures that require intervention.

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 the Transient Fault Handling chapter.

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 the Monitoring and Telemetry chapter.

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 Windows 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 Web Sites. 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, the Storage service, and it's deployed to the Web Site 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.

The Queue-centric Work Pattern chapter 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 we'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 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 get 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 datacenter what do you do? It's possible to set up your app in Windows Azure to run in multiple regions simultaneously so that if there's a disaster in one, you continue 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 Windows Azure is to make handling all of these kinds of failures a lot easier, and you'll see some examples of how we're doing 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% SLA means

www.dbooks.org

you should expect the service to be working correctly 99.9% 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% 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

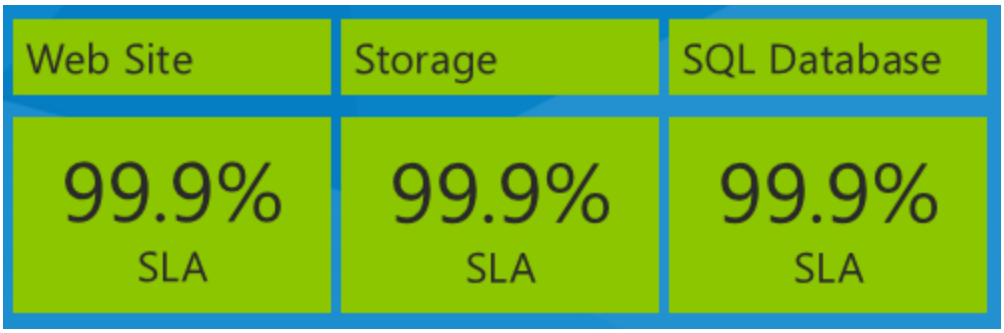
So a 99.9% SLA means your service could be down 8.76 hours a year or 43.2 minutes a month. That's more down time than most people realize. So as a developer you want to be aware that a certain amount of down time is possible and handle it in a graceful way. At some point someone is going to be using 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: does the clock get reset every week, every month, or every year? In Windows Azure we reset the clock 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.

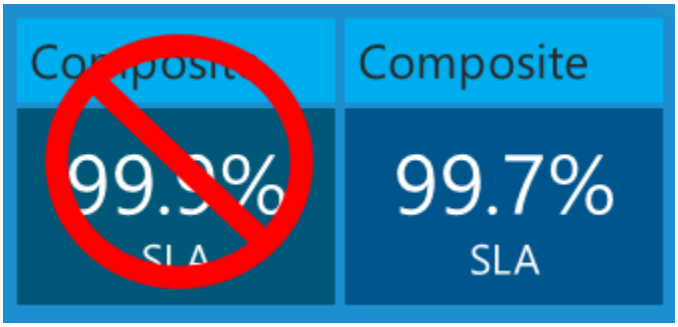
Of course we always aspire to do better than the SLA; usually you'll be down much less than that. The promise is that if we're ever down for longer than the maximum down time you can ask for money back. The amount of money you get back probably wouldn't fully compensate you for the business impact of the excess down time, but that aspect of the SLA acts as an enforcement policy and lets you know that we do take it 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 Web Site, Storage, and SQL Database services. Here are their SLA numbers as of the date this e-book is being written in December, 2013:



What is the maximum down time you would expect for the app based on these service SLAs? You might think that your down time would be equal to the worst SLA percentage, or 99.9% 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.



So your app could be down not just 43.2 minutes a month but 3 times that amount, 108 minutes a month – and still be within the Windows Azure SLA limits.

This issue is not unique to Windows Azure. We actually provide 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 to enterprise down-time experience

People sometimes say, "In my enterprise app I never have these problems." If you ask how much down time a month they actually have, 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 down time. Most enterprise apps unless they are especially mission-critical are actually down for more than the amount of time allowed by our 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 get more worried about it.

www.dbooks.org

When an enterprise achieves a greater up-time percentage than you get from a cloud SLA, they do it 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 up-time guarantee, you could be down far longer than you might imagine. For example, if you enable log-in to your site using a social provider such as Facebook or Twitter, check with the service provider to find out if there is an SLA, and you might find out 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 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 you 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 per second, you want to make sure automatic retries don't happen so fast that they cause the throttling to continue. We'll have more to say about throttling in the Transient Fault Handling chapter.

Summary

This chapter has tried to help you realize why a real world cloud app has to be designed to survive failures gracefully. Starting with the next chapter, the remaining patterns in this series 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 in order to minimize throughput, latency, and connection problems with database access.
- Implement loose coupling via the queue-centric work pattern, so that your app front end can continue to work when the back end is down.

Resources

For more information, see later chapters in this e-book and the following resources.

Documentation:

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

www.dbooks.org

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

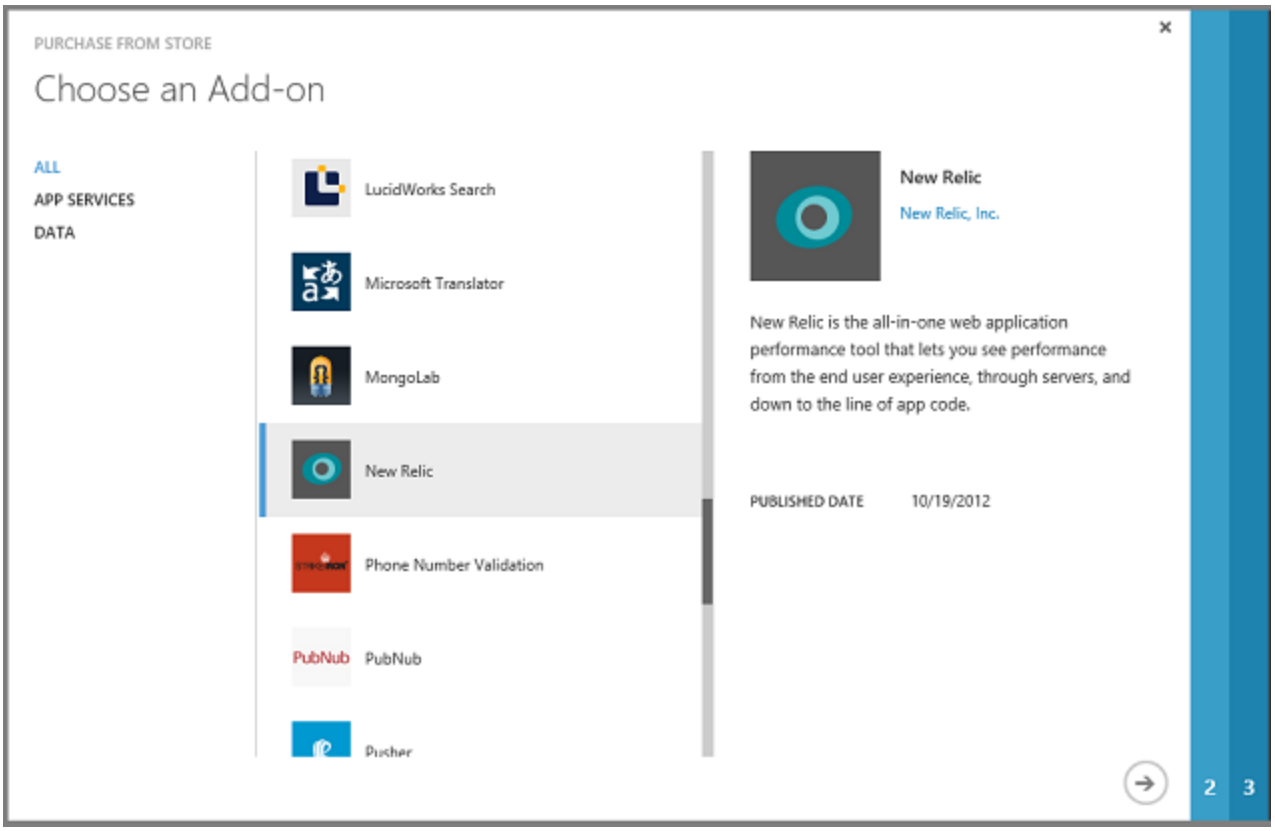
One of the things that's great about the cloud environment is that it's really easy to buy or rent your way to victory. Telemetry is an example. Without a lot of effort you can get a really good telemetry system up and running, very cost-effectively. There are a bunch of great partners that integrate with Windows 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 Windows Azure:

- New Relic
- AppDynamics
- MetricsHub
- Dynatrace

As this e-book is being written in December, 2013, Microsoft Application Insights for Visual Studio Online is not released yet but is available in preview to try out. Microsoft System Center also includes monitoring features.

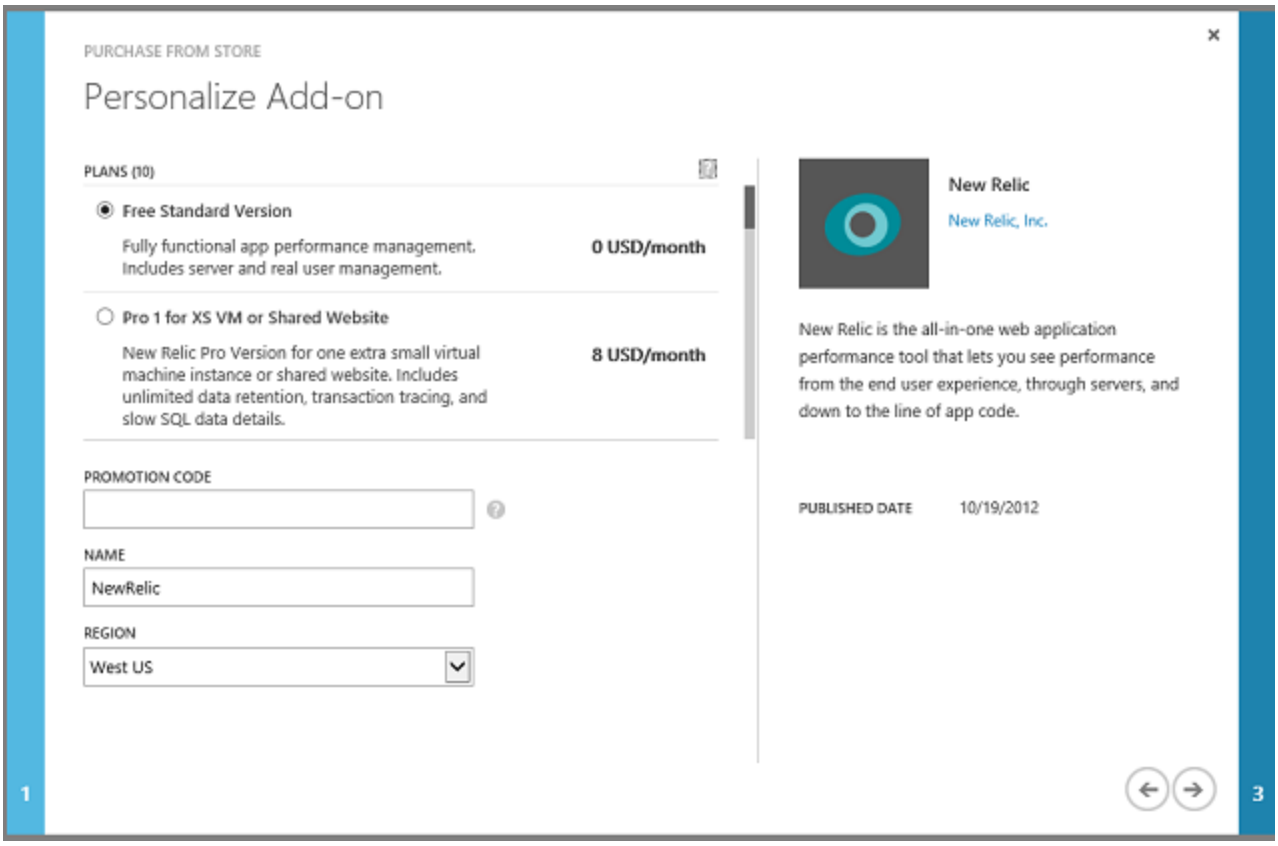
We'll quickly walk through setting up New Relic to show how easy it can be to use a telemetry system.

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

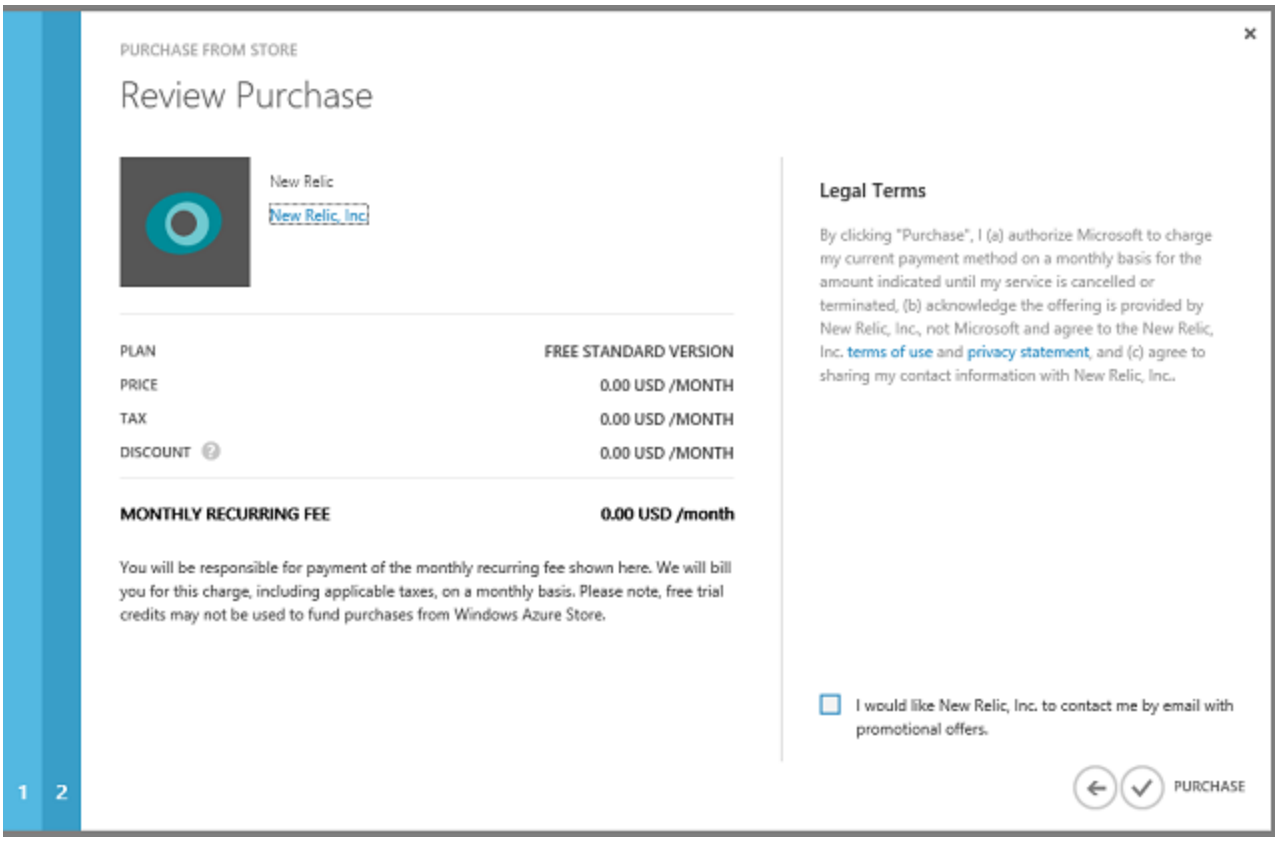


Click the right arrow and choose the service tier you want. For this demo we'll use the free tier.

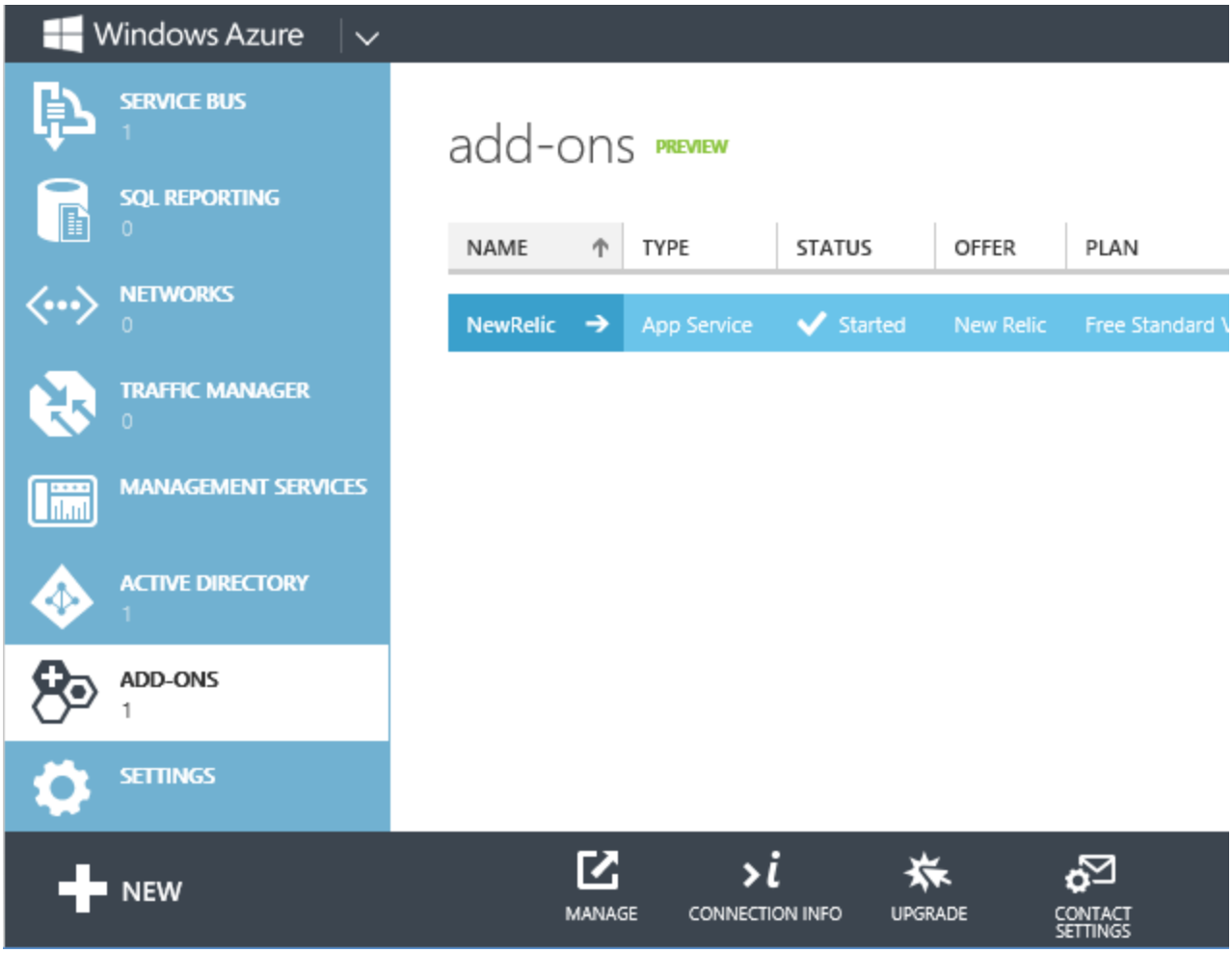
www.dbooks.org



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



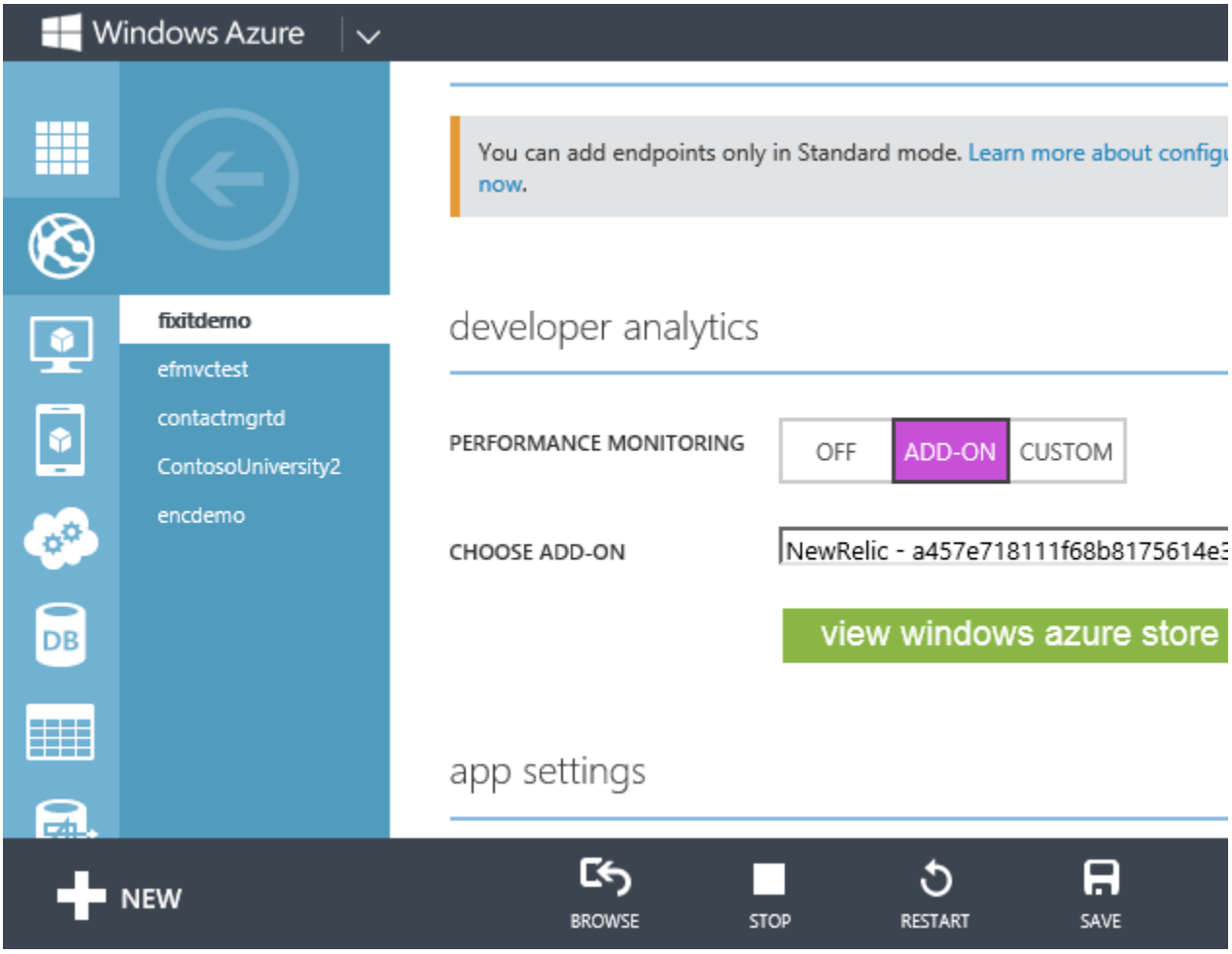
www.dbooks.org



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

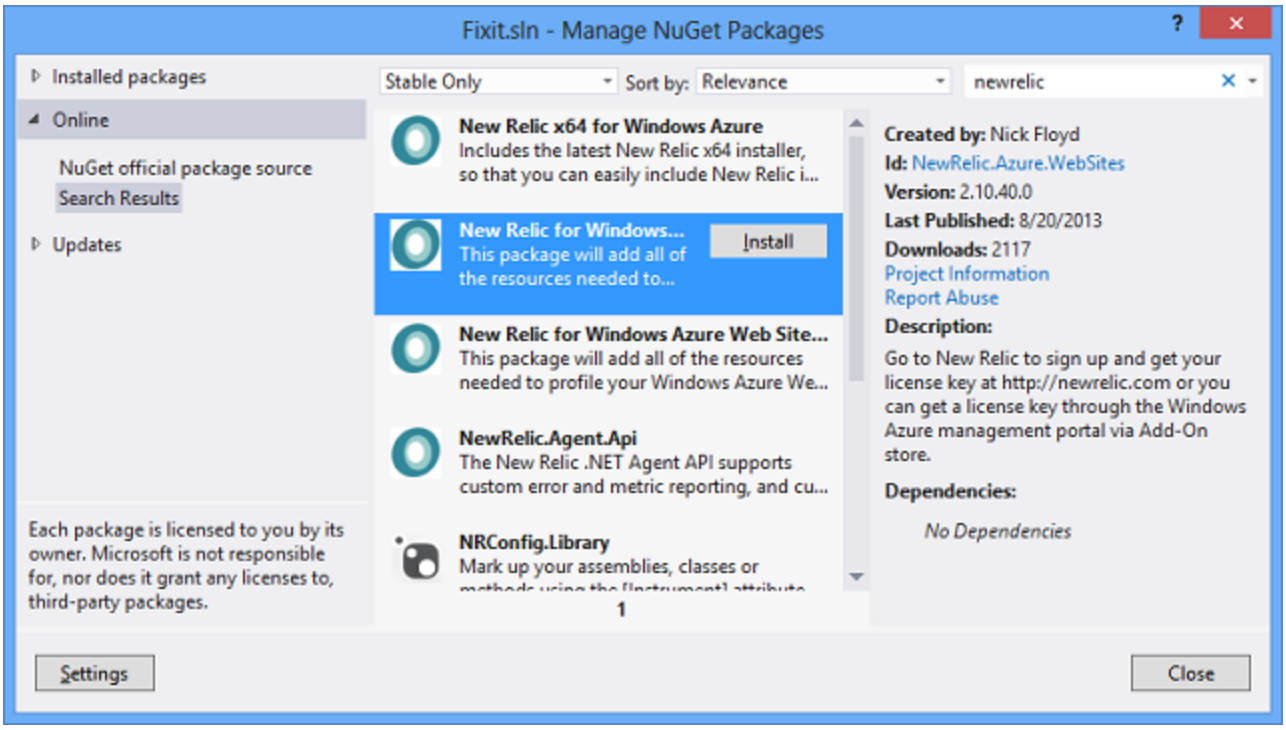


Go to the **Configure** tab for your Web Site in the portal, set **Performance Monitoring** to **Add-On**, and set the **Choose Add-On** drop-down list to **New Relic**. Then click **Save**.



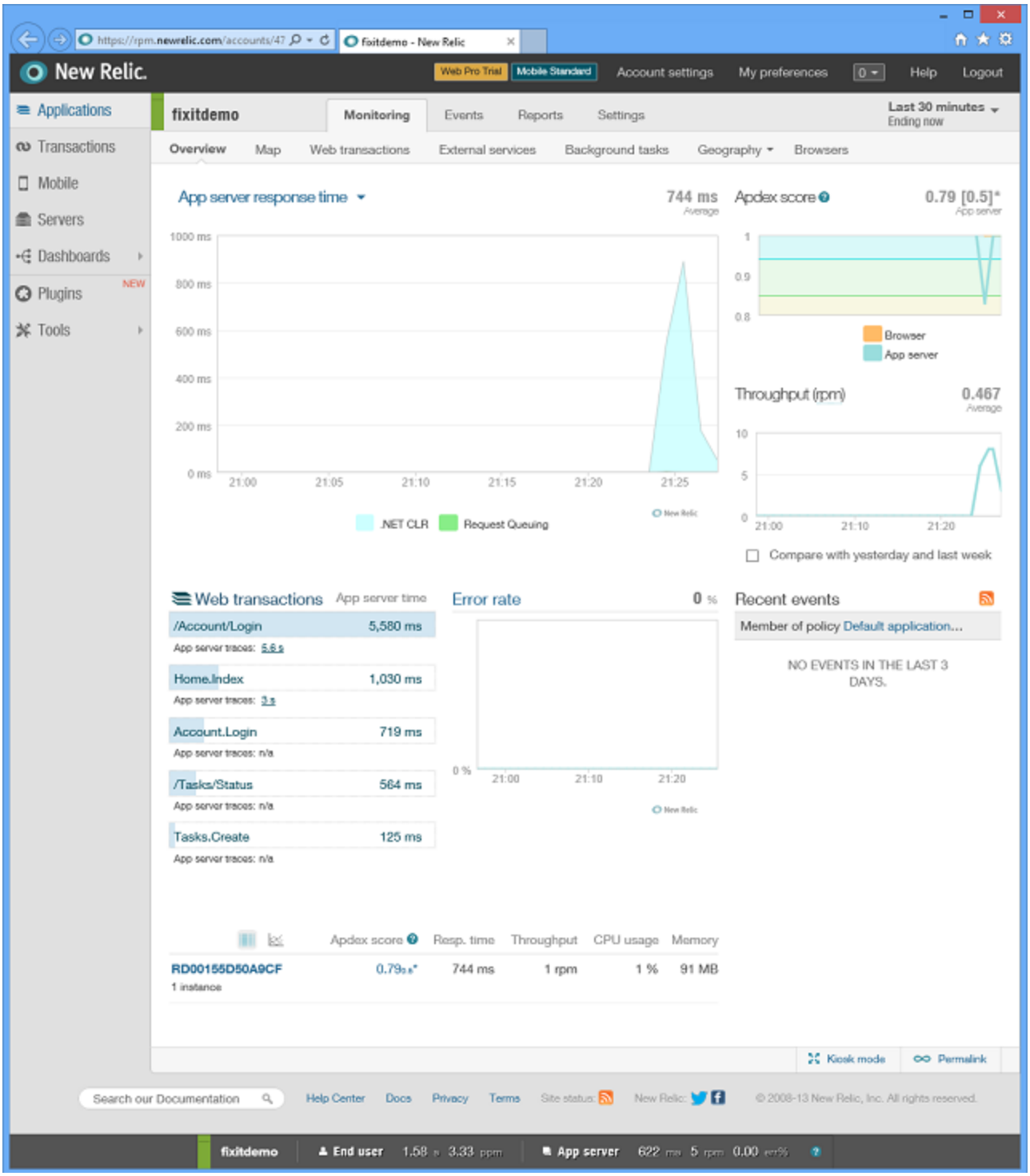
In Visual Studio, install the New Relic NuGet package in your app.

www.dbooks.org



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

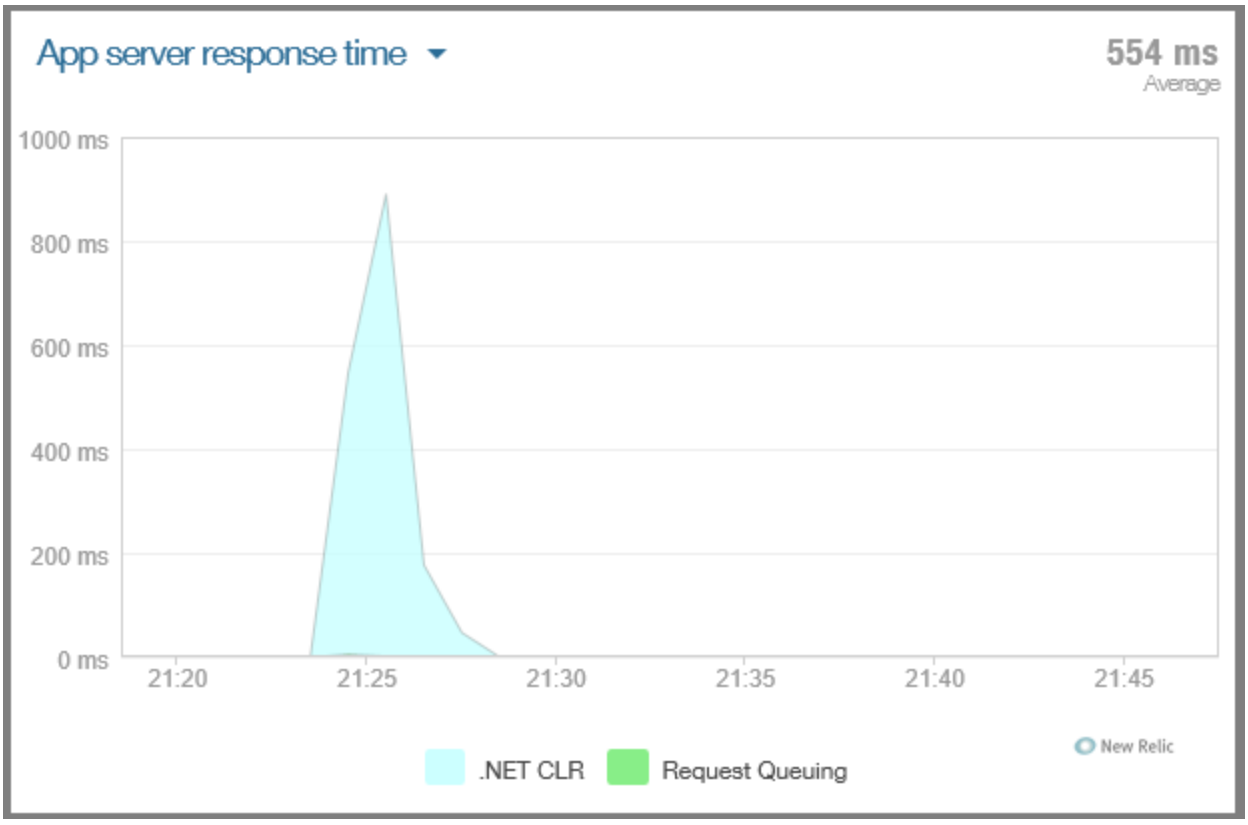
Then go back to the **New Relic** page in the **Add-ons** tab of the portal and click **Manage**. The portal sends you to the New Relic management portal, using single sign-on for authentication so you don't have to enter your credentials again. The Overview page presents a variety of performance statistics.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



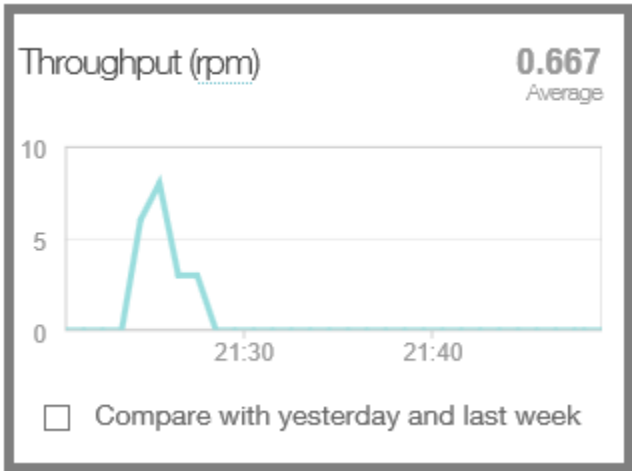
Here are just a few of the statistics you can see:

- Average response time at different times of day.

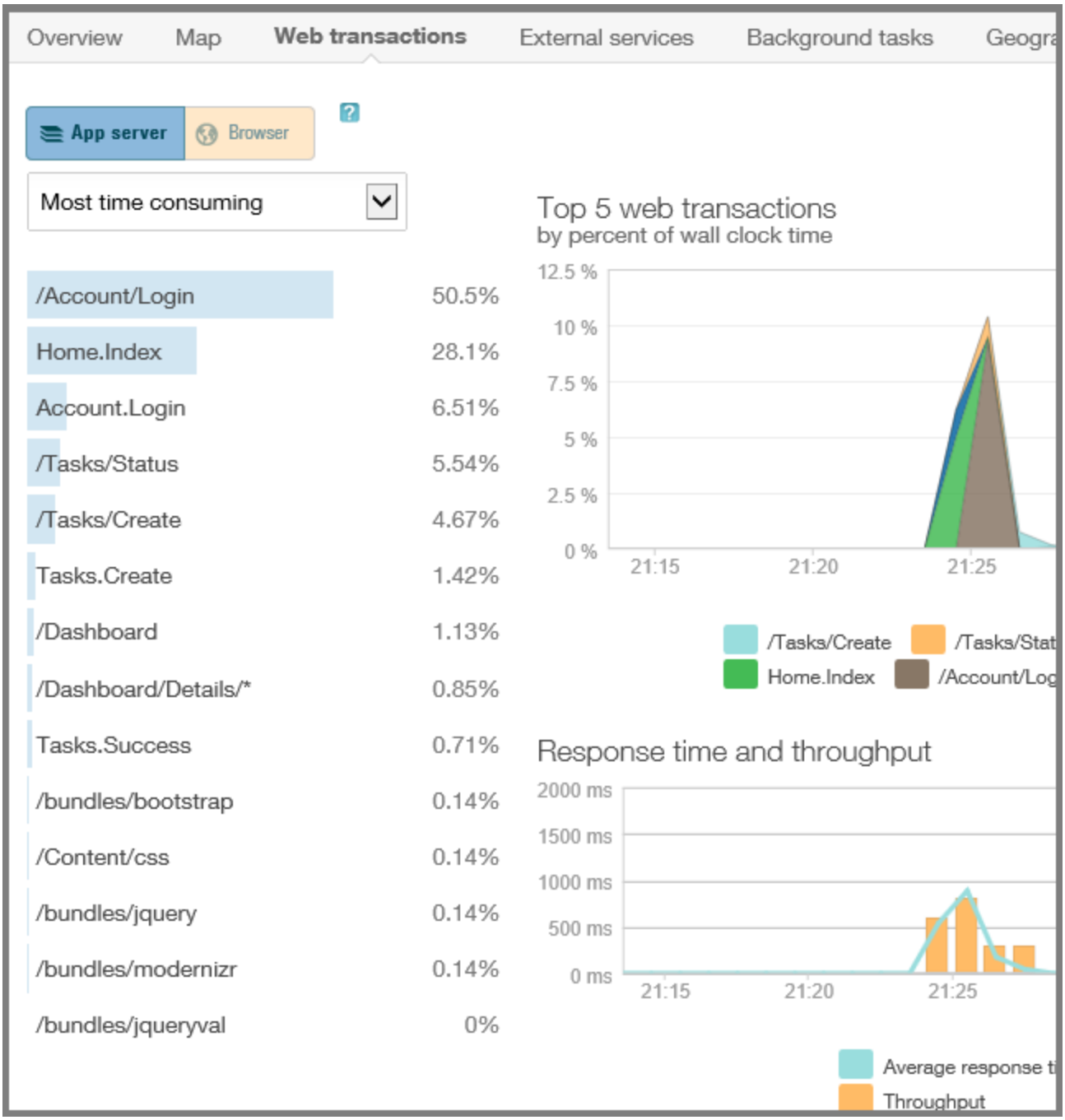
www.dbooks.org



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

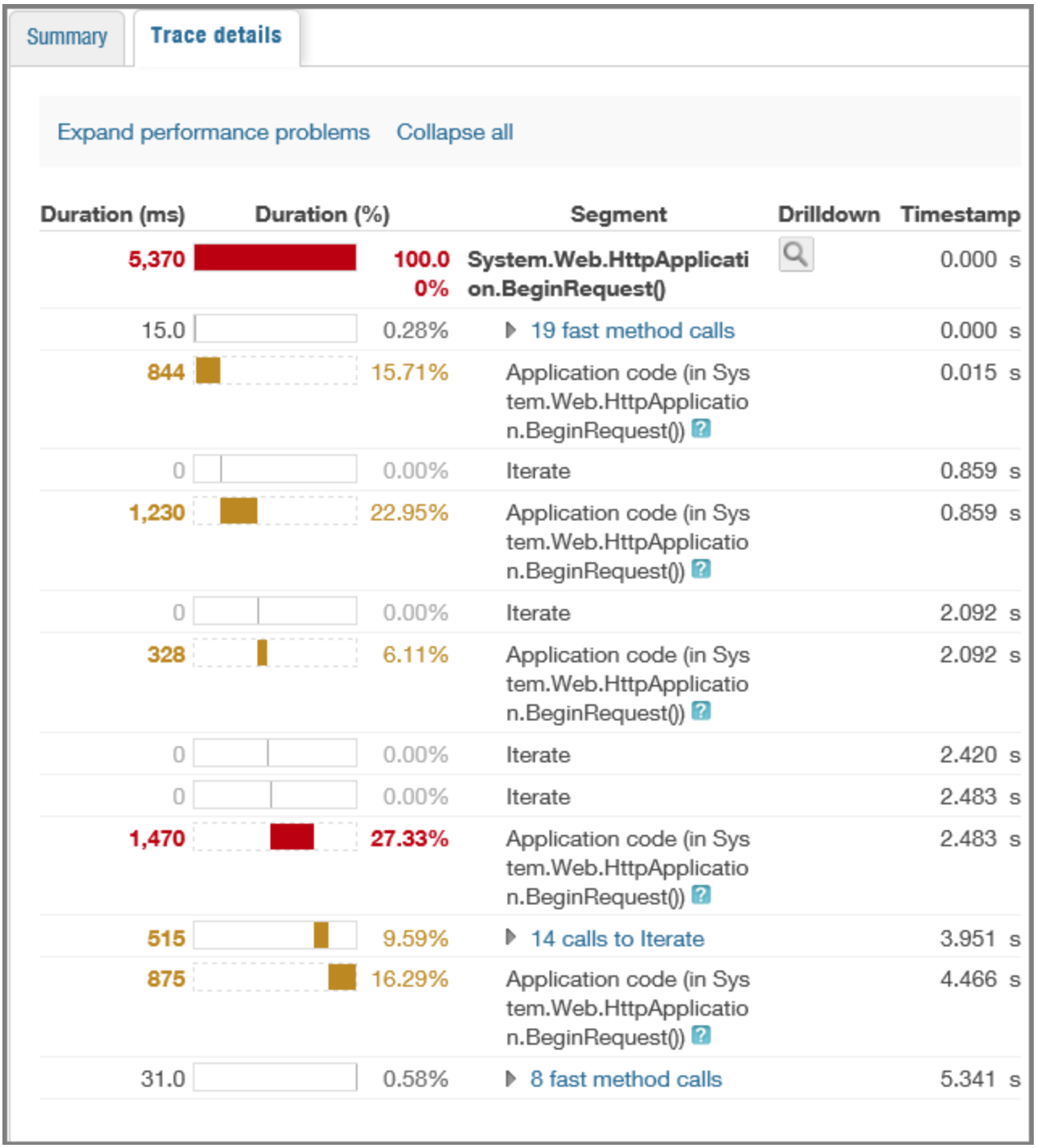


- Server CPU time spent handling different HTTP requests.



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

www.dbooks.org

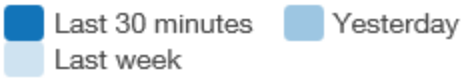


- Historical performance statistics.

/Account/Login Track as Key Transaction

App performance | **Historical performance**

App server historical performance



Response time

Last 30 minutes:
5,580 ms

Yesterday:
0 ms

Last week:
0 ms

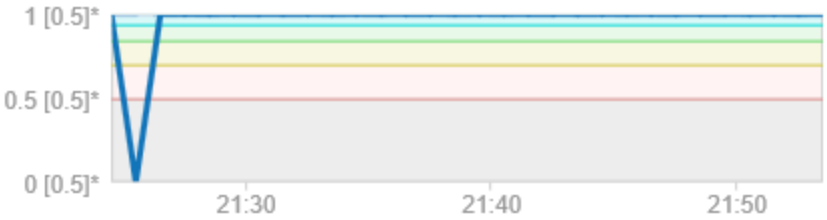


Apdex

Last 30 minutes:
$0.00_{0.5}$*

Yesterday:
$NS_{0.5}$

Last week:
$NS_{0.5}$



Throughput

Last 30 minutes:
0.0334 rpm

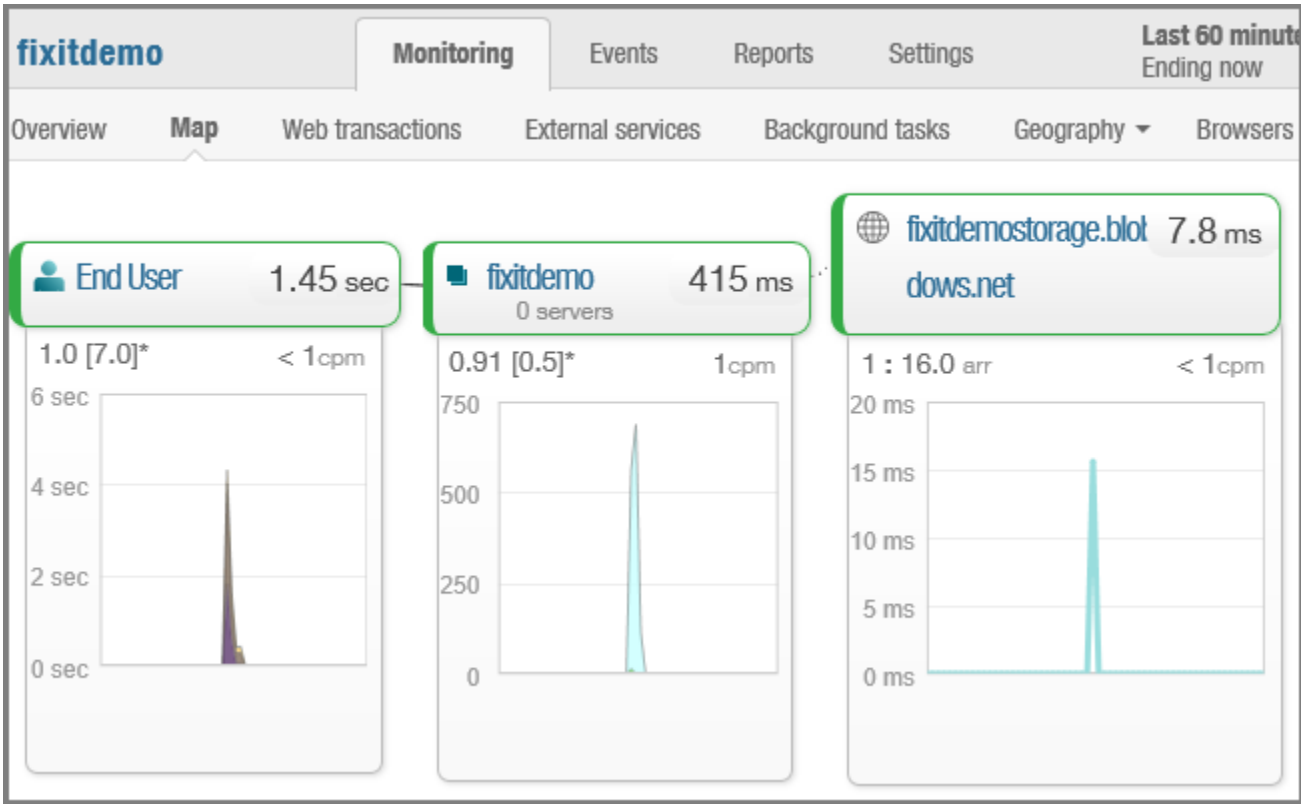
Yesterday:
0 rpm

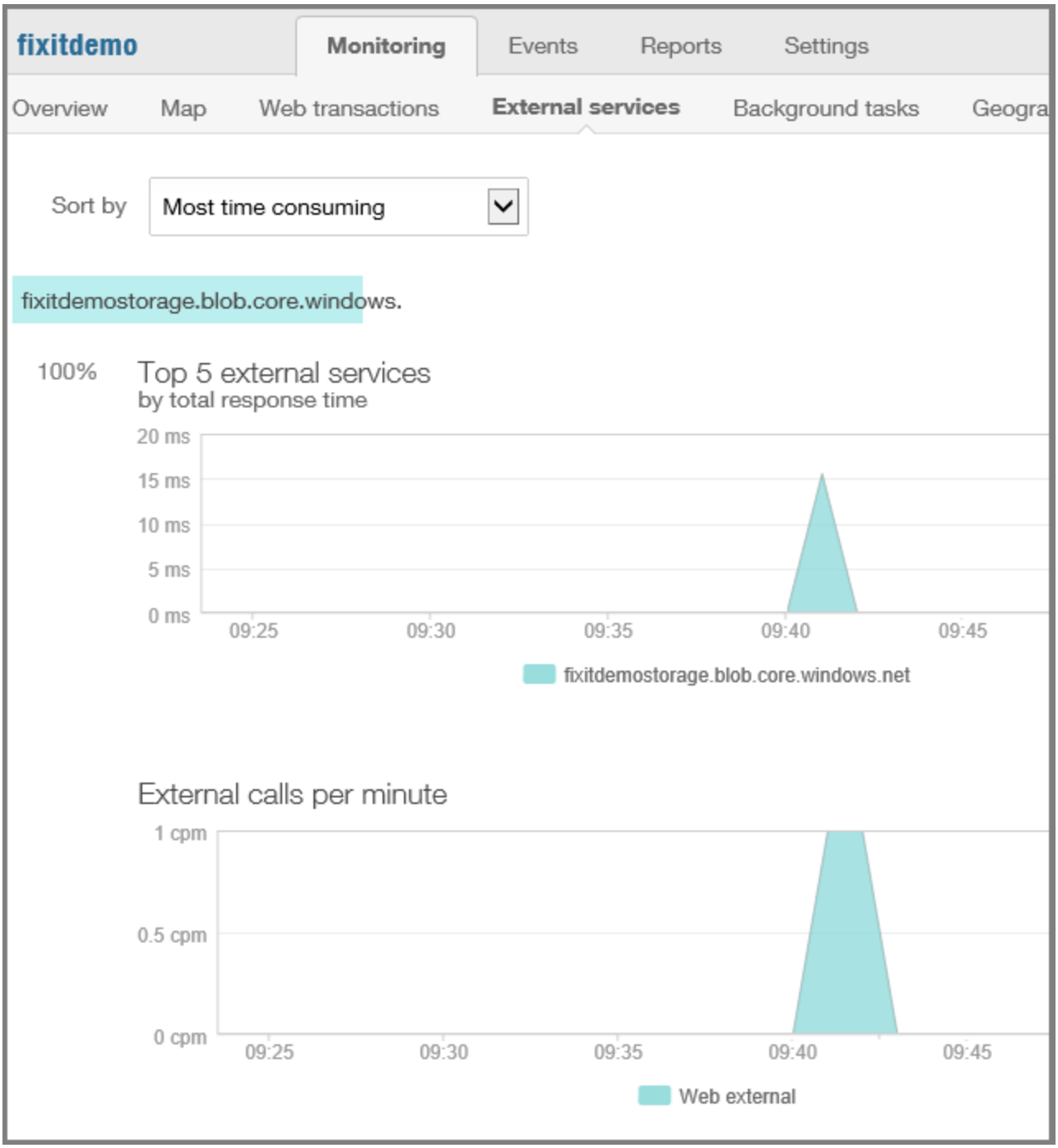
Last week:
0 rpm



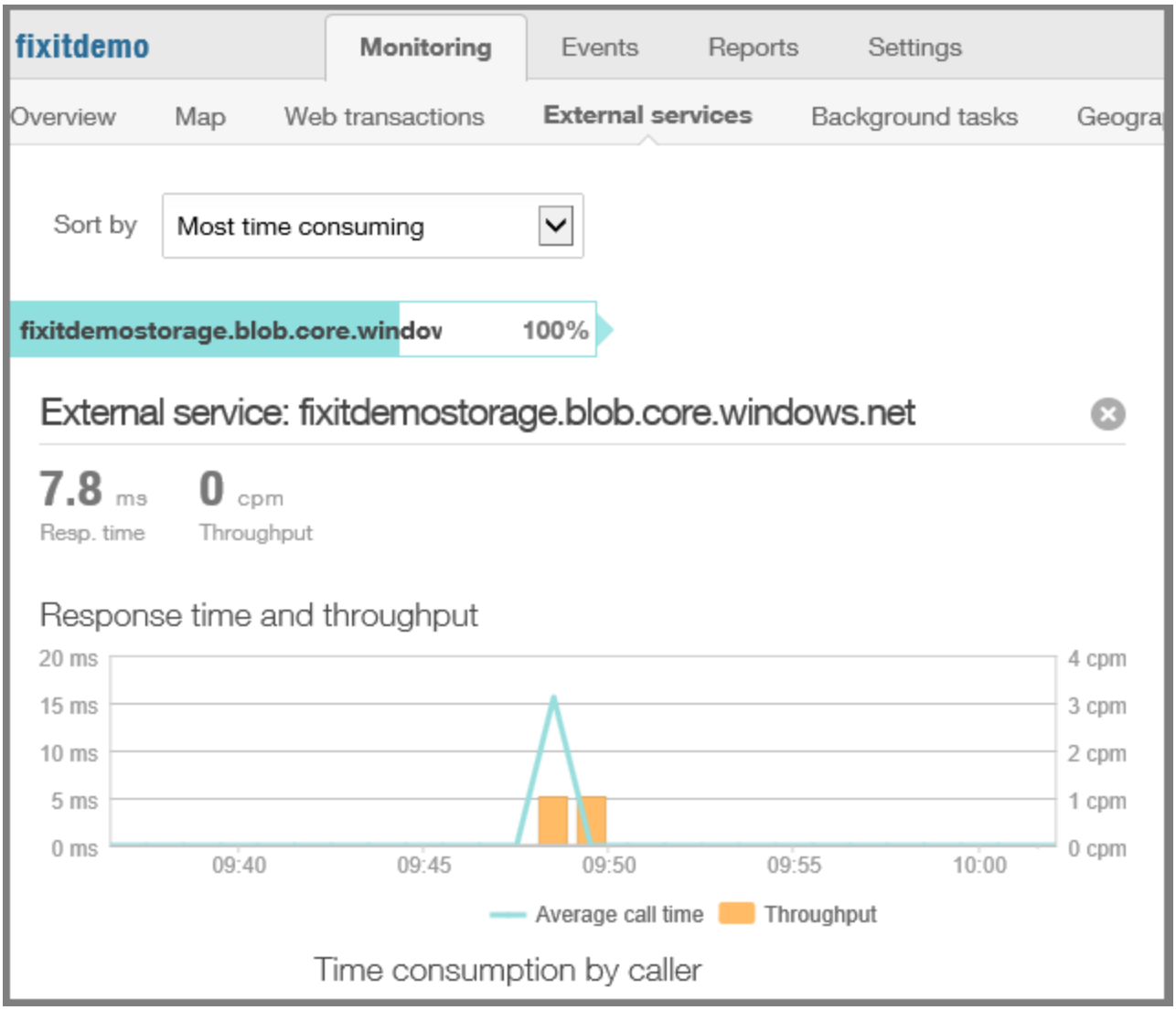
www.dbooks.org

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

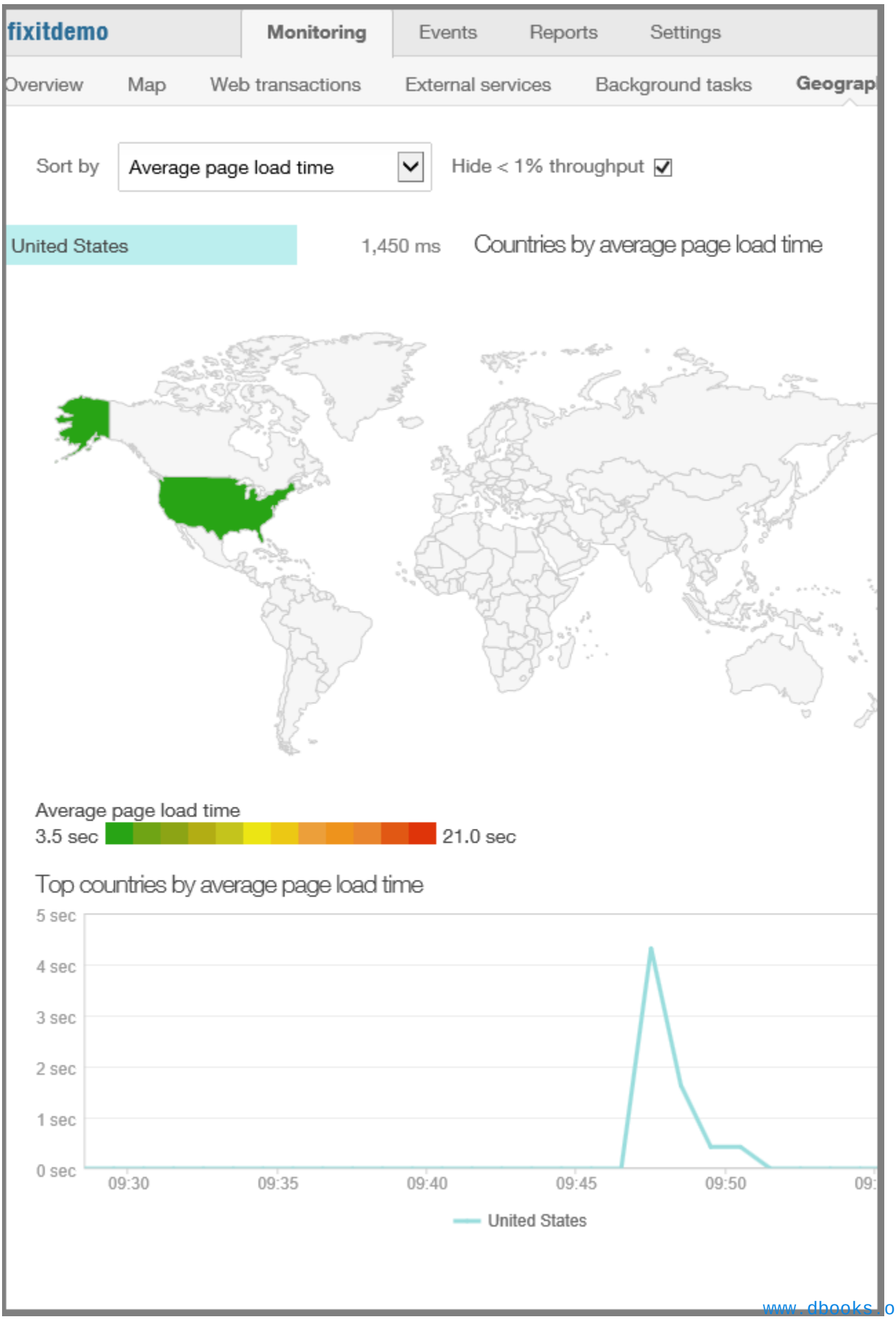




www.dbooks.org



- Information about where in the world or where in the U.S. web site traffic came from.

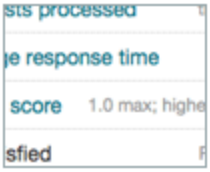


www.dbooks.org

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

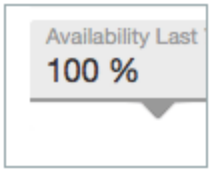
fixitdemo | Monitoring | Events | **Reports** | Settings

SLA | Availability | Scalability | Web transactions | Database | Background jobs | Speed index



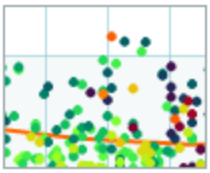
SLA report

Is your application meeting long-term SLAs and overall performance objectives? How are the changes, fixes, updates that you are making today affecting application performance over time? The long-term performance trends report gives you week-over-week views of performance including number of Requests Processed, Average Response Time, and Apdex Score to help you understand exactly where application performance is heading.



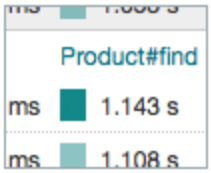
Availability report

Web site availability is an important measurement that goes hand-in-hand with application response time and throughput monitoring. New Relic's pinging service allows you to get instant feedback on site uptime and correlate with overall application performance.



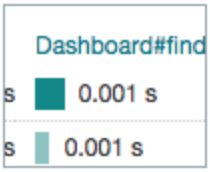
Scalability analysis

Knowing how your application scales allows you to predict performance as your application grows. Do you need new hardware? See your application response time, CPU, and database utilization plotted against application load.



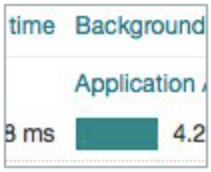
Web transaction analysis

Learn where your web transactions spend their time—so you know where to tune. Which web transactions consume the most time? Which are called the most? Which have the greatest standard deviations?



Database analysis

How does your app look from the database's point of view? Know how much time your application spends waiting for the database to respond. See the top-15 database consumers of wall clock time. And more.



Background job analysis

How are background jobs performing? See a complete list of jobs run in production using any of the background job frameworks instrumented by New Relic, including DelayedJob and Resque.

www.dbooks.org

New Relic is just one example of a telemetry system; you can get all of this 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 the 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've got 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 can introduce a substantial delay between the time you're aware of a problem and the time you get useful troubleshooting information about it. And the information you get might not be helpful for intermittent errors.

What we recommend in 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 you 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 to the amount of troubleshooting time and money you can save by having all of 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 -- in order to avoid performance bottlenecks, make 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 only write ACT logs 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 it all to find genuine issues. And if your ACT logs automatically trigger

some action such as sending email to support staff, avoid letting thousands of such actions be triggered by a single issue.

In .NET System.Diagnostics tracing, logs can be assigned Error, Warning, Info, and 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 have logging always 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 you always log Error, Warning, and Info logs in production, and you'll want to be able to dynamically add Debug/Verbose logging for troubleshooting on a case-by-case basis.

Windows Azure Web Sites have 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. The Blob storage support makes it easier to run HDInsight analysis jobs on your application logs, because HDInsight knows how to work with Blob storage directly.

Log Exceptions

www.dbooks.org

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 make sure that you are providing everything that will be 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!).

If you rely on each developer to do the right thing with exception logging, some of them won't. To ensure that it gets done the right way every time, build exception handling right 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 it's to a database or 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 slow-downs 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 whatever telemetry service you have and validate their experience, and then you want to be able to look are your own logs to dive into the details of why it's slow.

Use an ILogger interface

What we recommend doing when you create a production application is to create a simple *ILogger* interface and stick some methods in it. This makes it easy to change the logging implementation later and not have to go through all your code to do it. We could be using 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.

That way, 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 do asynchronous logging.)

One possible reason for using a framework such as NLog is to facilitate dividing up logging output into separate high-volume and high-value data stores. That helps you to 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 a key piece of information is the error number, so you could include an error number parameter in the method 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 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.

www.dbooks.org

```
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 details. And every time it makes a call to the database, 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 make sure that log messages identify what 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;
    }

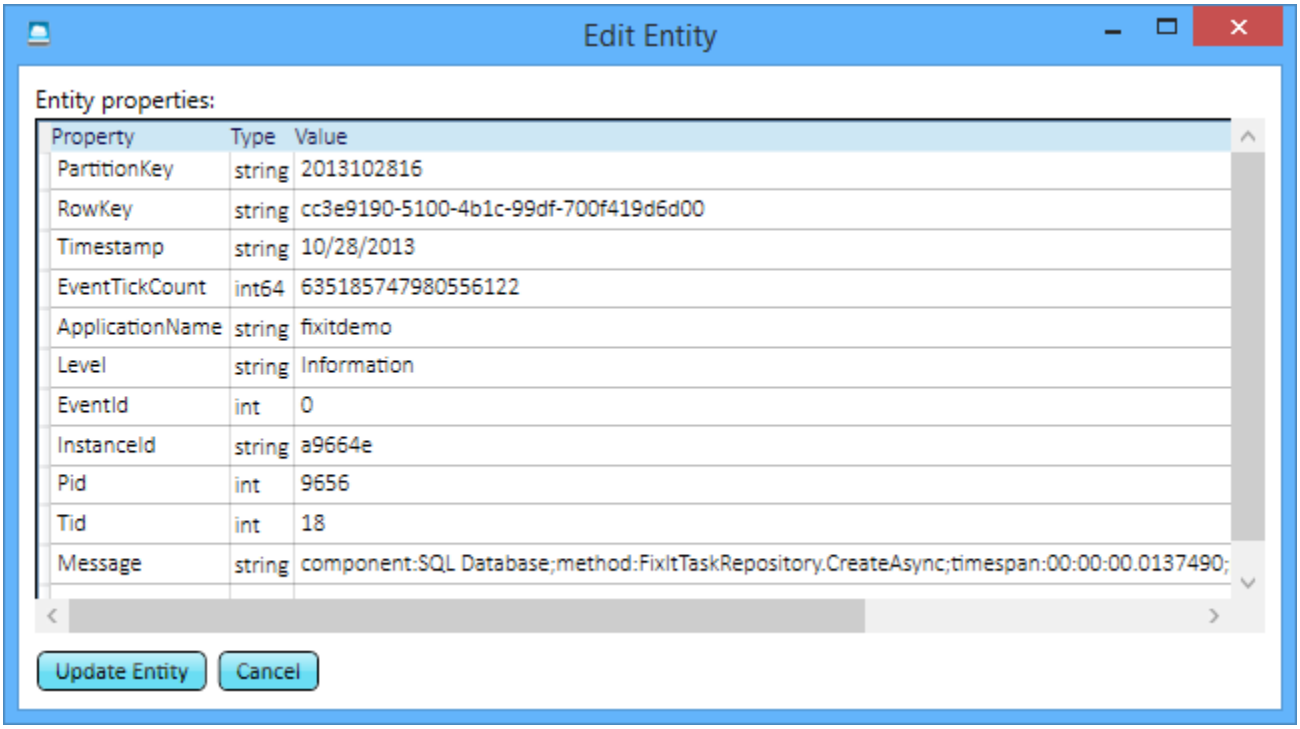
    public async Task<FixItTask> FindTaskByIdAsync(int id)
    {
        FixItTask fixItTask = null;
        Stopwatch timespan = Stopwatch.StartNew();

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

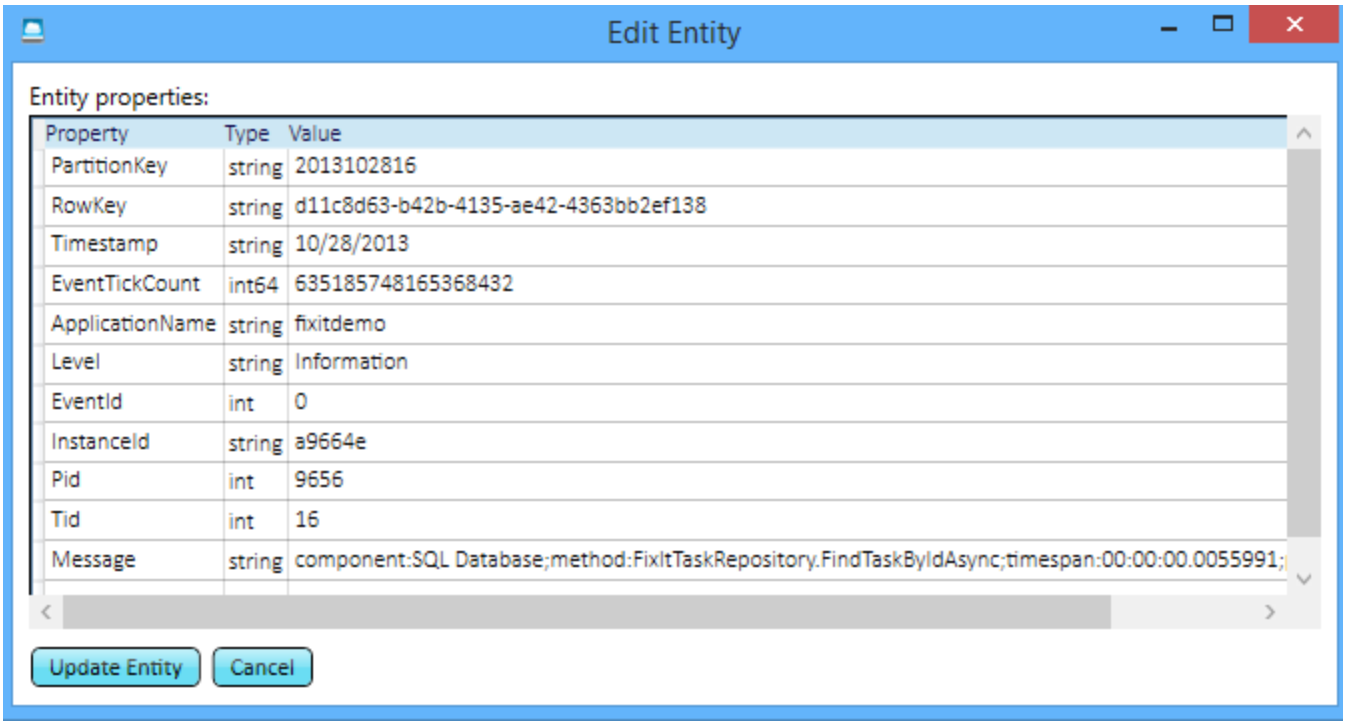
            timespan.Stop();
            log.TraceApi("SQL Database",
"FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed, "id={0}", id);
        }
        catch(Exception e)
        {
            log.Error(e, "Error in
FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
        }

        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.

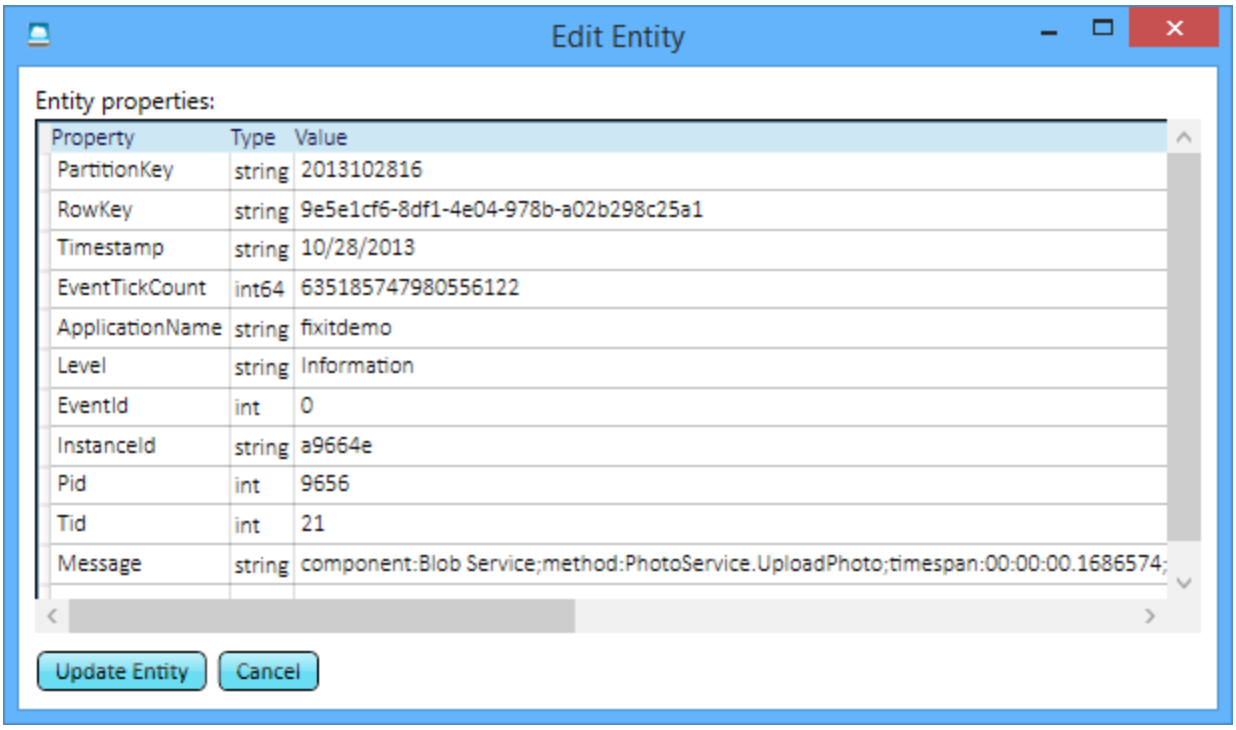


www.dbooks.org



If you go browsing through the logs, you can see that the time database calls take is variable. That information could be useful: because the app logs all this 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 do the same thing for the Blob service – for every time the app uploads a new file, there's a log, and you can see exactly how long it took to upload each file.



It's just a couple extra lines of code to write every time you call a service, and now whenever someone says they ran into an issue, you know exactly what the issue was, whether it was an error, or even if it 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 above 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 and only have to specify in one place the implementation that gets 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.

www.dbooks.org

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 it needs an *IFixItTaskRepository* interface, pass in an instance of the *FixItTaskRepository* class.

Built-in logging support in Windows Azure

Windows Azure supports the following kinds of logging in Web Sites:

- 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).

Windows Azure supports the following kinds of logging in Cloud Services:

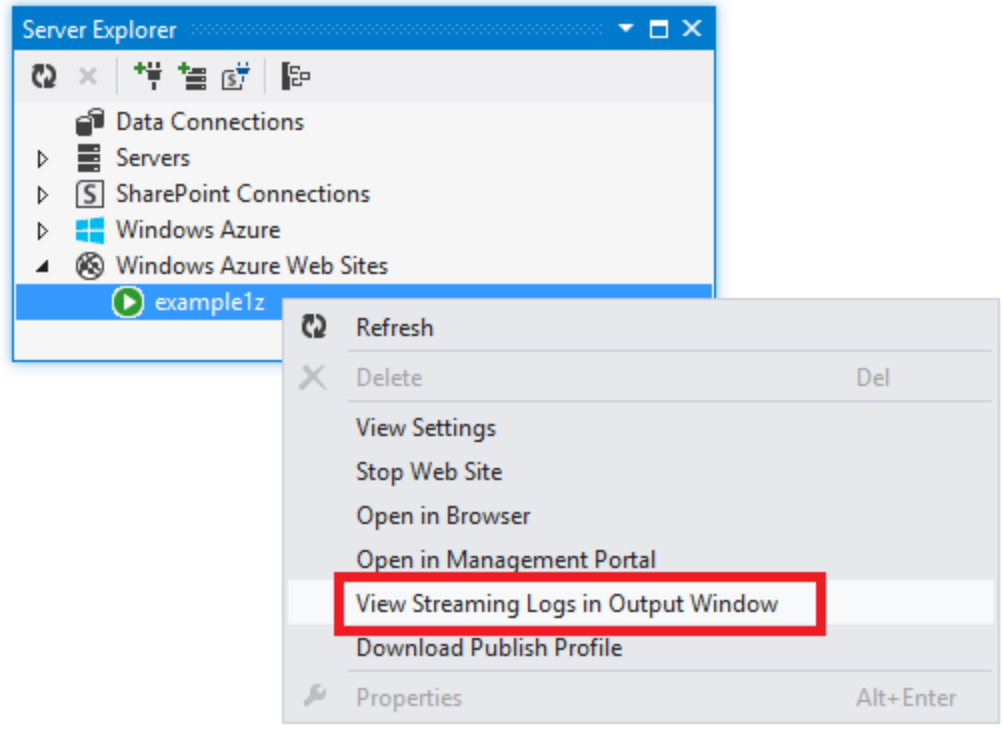
- System.Diagnostics tracing.
- Performance counters.
- Windows Events.
- IIS Logs (HTTP/FREB).
- Custom directory monitoring.

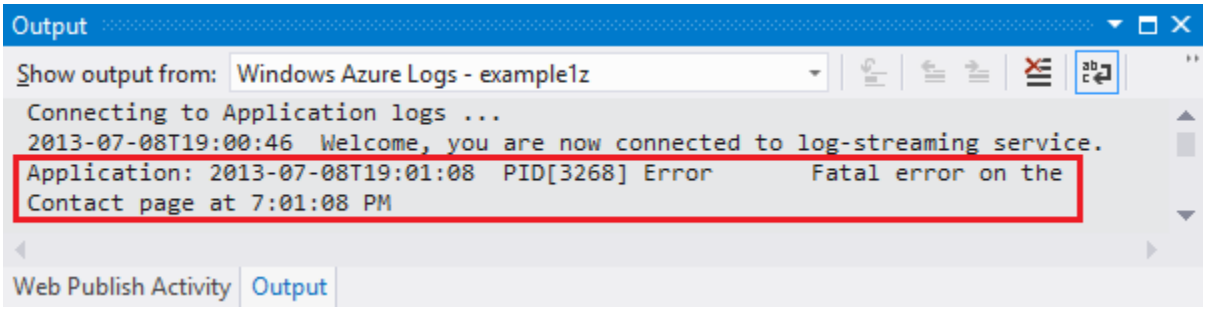
The Fix It app uses System.Diagnostics tracing. All you need to do to enable System.Diagnostics logging in a Windows Azure Web Site is flip a switch in the portal or call the REST API. In the portal, click the **Configuration** tab for your site, and scroll down to see the **Application Diagnostics**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

www.dbooks.org

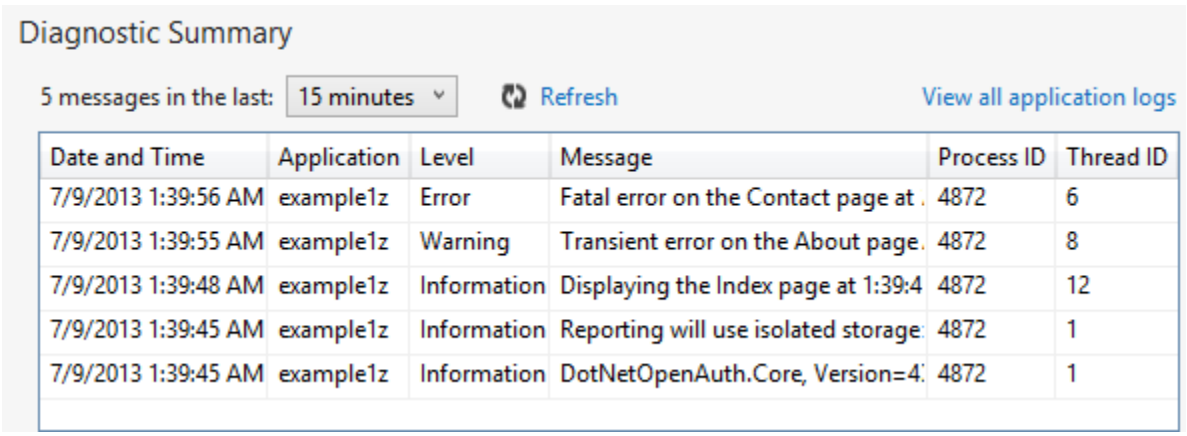


After you enable logging in Windows Azure, you can see logs in the Visual Studio Output window as they are created.





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



www.dbooks.org

Summary

It’s really simple to implement an out-of-the-box telemetry system, instrument logging in your own code, and configure logging in Windows 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 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 and Practices - Windows 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 Windows Azure Cloud Services. White paper by Mark Simms and Michael Thomassy. See the Telemetry and Diagnostics section.

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:

- Windows 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 a Windows 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 Windows 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 Microsoft Customer Advisory Team (CAT) 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 Microsoft Windows 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.

www.dbooks.org

Transient Fault Handling

When you're designing a real world cloud app, one of the things you have to think about is how to handle temporary service interruptions. This issue is uniquely important in cloud apps because you're so dependent on network connections and external services. You can frequently get little glitches that are typically 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 you'll find that failed and dropped database connections happen periodically. That's partly because you're going through more load balancers compared to the on-premises environment where your web server and database server have a direct physical connection. Also, sometimes when you're dependent on a multi-tenant service you'll see calls to the service get slower or time out because someone else who uses the service is hitting it heavily. In other cases you might be the user who is hitting the service too frequently, and the service deliberately throttles you – denies connections – in order to prevent you 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 the customer 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 (not through Entity Framework). You just 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.
        }
          }
```

TFH also supports Windows 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 this Patterns and 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 3 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 it will wait for a longer period of time before trying again. If three tries in a row fail, it will throw an exception. The following section about circuit breakers explains why you want exponential back-off and a limited number of retries.

  You can have similar issues when you're using the Windows Azure Storage service, as the Fix It app does for Blobs, and the .NET storage client API already implements the same kind of logic. You just specify the retry policy, or 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 be tying up IIS dispatch queues and preventing your app from servicing requests that it otherwise could handle successfully.

www.dbooks.org

- If everyone is retrying due to a service failure, there could be so many requests queued up that the service gets flooded when it starts to recover.
- If the error is due to 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 web page 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 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 silent. 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. 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 which 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
Windows 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, Entity Framework, or the Windows 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 Windows 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. Web page version of the FailSafe video series.
- Microsoft Patterns and Practices - Windows 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 Microsoft Customer Advisory Team (CAT) 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 Microsoft Windows Azure Customer Advisory Team that demonstrates how to use the Enterprise Library Transient Fault Handling Block (TFH). For more information, see Cloud Service Fundamentals

www.dbooks.org

Data Access Layer – Transient Fault Handling. TFH is recommended for database access using ADO.NET directly (without using Entity Framework).

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 Windows 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 distributed cache, which means the data is not stored on 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). That way, when the application scales by adding or removing servers, or when servers are replaced due to 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. It's 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, as 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 other reasons than performance as well. For data that doesn't have to be perfectly up-to-date when shown to the user, cache access can serve as a circuit breaker for when the persistent data store is unresponsive or unavailable.

Popular cache population strategies

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

www.dbooks.org

- 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 on 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 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.

In order to keep data in the cache current, you can delete related cache entries when your application creates, updates, or deletes data (we'll show how this might be done in Fix It app code). If it's alright 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 (amount of time since the cache item was created) or sliding expiration (amount of time since the last time a cache item was accessed). Absolute expiration is used when you are depending on the cache expiration mechanism to prevent the data from becoming too stale. In the Fix It app, we'll manually evict stale cache items and we'll use sliding expiration to keep the most current data in cache. 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 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 not found, we get it from the database and store it in the cache. The changes you'd make to add caching to the `FindTaskByIdAsync` method are highlighted.

```
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.

Popular caching frameworks

www.dbooks.org

The most popular caching frameworks for Windows Azure are Windows Azure In-Role Cache, Redis, and Memcached.

In order to use caching in a web app in Windows Azure, you have to deploy the app to a Windows Azure Cloud Service rather than a Windows Azure Web Site. For information about the differences between Cloud Services and Web Sites, see Windows Azure Web Sites, Cloud Services, and VMs: When to use which?

ASP.NET session state using a cache provider

As mentioned in the web development best practices chapter, 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 that doesn't enable scale out (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 to 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 Windows Azure Cache.

Summary

You've seen how the Fix It app could implement caching in order 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

- Windows Azure Cache. Official MSDN documentation on caching in Windows Azure.
- Microsoft Patterns and Practices - Windows 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 Windows Azure Cloud Services. W. 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 Windows Azure Web Sites 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.

www.dbooks.org

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 Web Sites, Storage, and SQL Database. If any one of these services fails, the app will return 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 it can be processed.

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

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

The queue-centric work pattern is useful for:

- Work that is time consuming (high latency).
- Work that requires an external service that might not always be available.
- Work that is resource-intensive (high CPU).
- 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 web site confirms your order immediately. But that doesn't mean your stuff is already in a truck being delivered. They put a task in a queue, and in the background they are doing the credit check, preparing your items for shipping, and so forth.

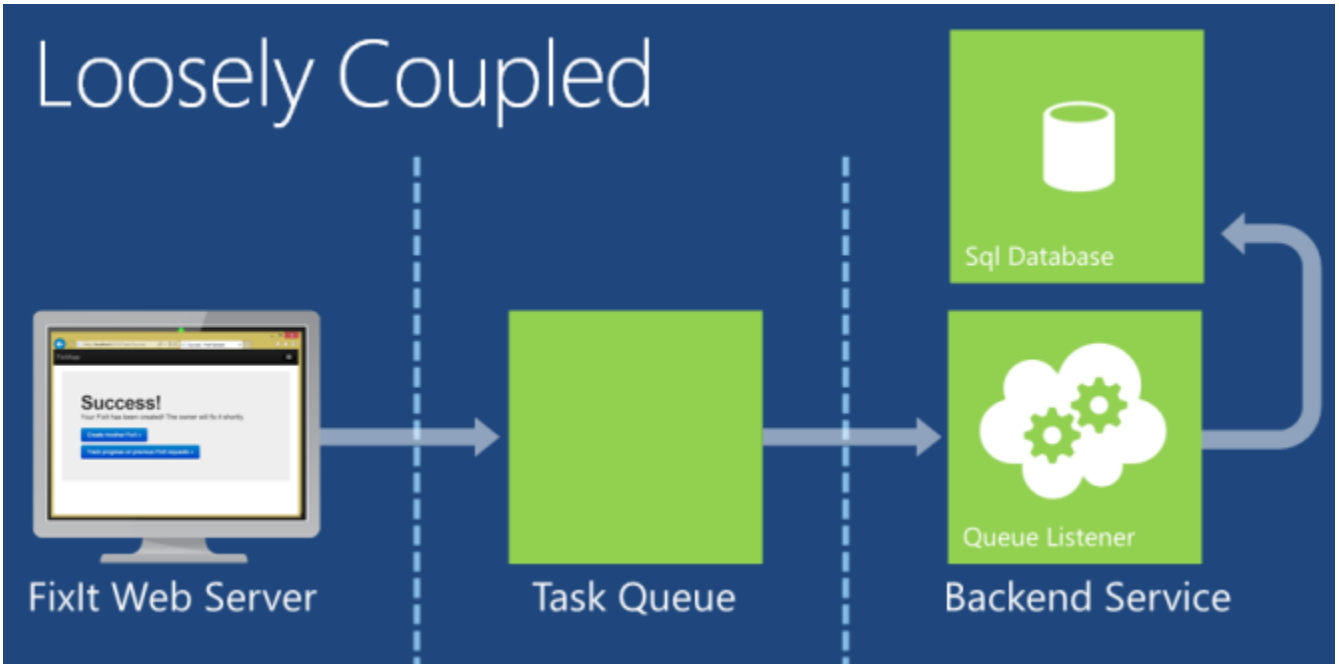
For scenarios with short latency, the total end-to-end time might be longer using a queue, compared with doing the task synchronously. But even then, the other benefits can outweigh that disadvantage.

Increased Reliability

In the version of Fix It that we've been looking at so far, 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.

Using queues, 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 queue is stored in persistent storage, so the Fix It tasks stay in the queue until they are processed. If any of the backend 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 backend services are available again. At that point, the backend services will catch up on the backlog.



Moreover, now you can add more backend 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 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 Web Sites × Storage × SQL Database = 99.7%. (See Design to Survive Failures.)

With a queue, the web front end depends on Web Sites and Storage, for a composite SLA of 99.8%. (Note that queues are built on Windows Azure storage, so they share the SLA with blob storage.)

www.dbooks.org

If you need even better than 99.8%, you can create two queues in two different regions. Designate one as the primary, 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.

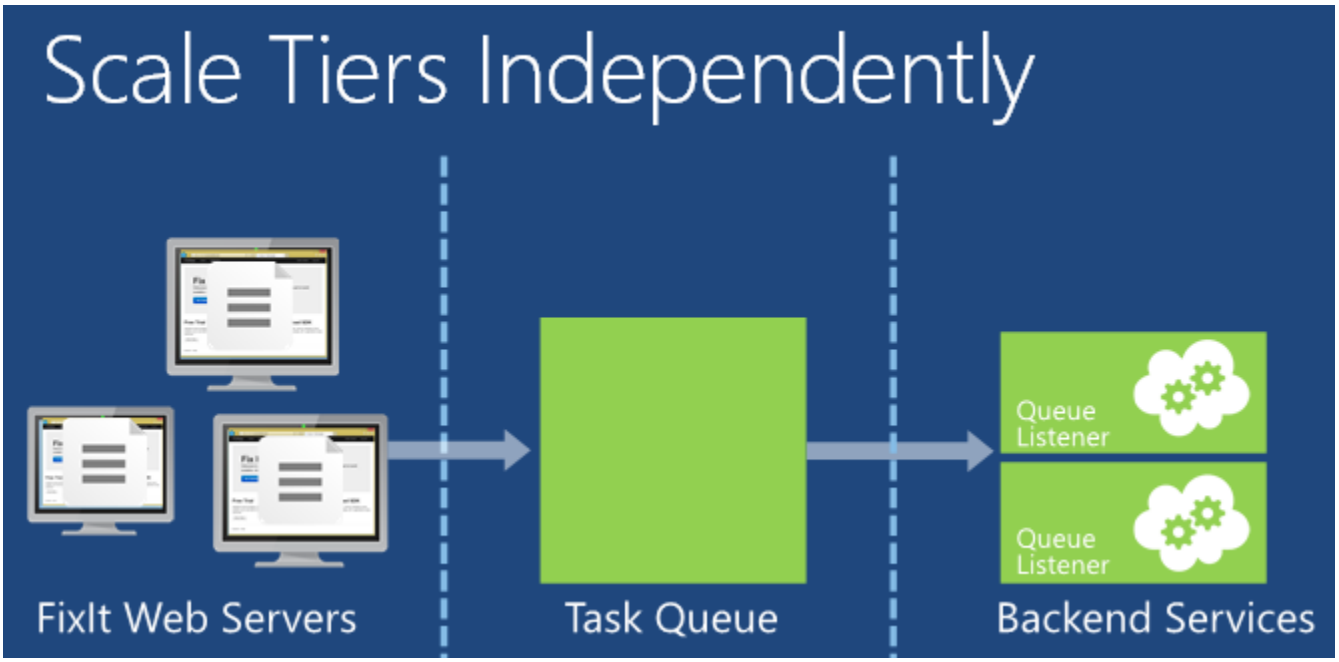
Rate Leveling and Independent Scaling

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 backend 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 off-loaded to a backend 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 backend 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 backend servers.



Autoscaling works with backend 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 backend VMs. Or, you can autoscale based on how many items are in a queue. For example,

you can tell autoscale to try to keep no more than 10 items in the queue. If the queue has more than 10 items, autoscale will add VMs. When they catch up, autoscale 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 Windows Azure Queue Storage Service , which is a persistent FIFO queue that runs on top of the Windows Azure storage service. Another option is to use Windows 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 Windows 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 can only be processed by 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 Windows 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. Windows Azure Service Bus is a separate service with its own SLA. If we used 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, make 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. Create a CloudQueueClient instance. The `CloudQueueClient` instance is used to execute requests against the Queue Service.
2. Create the queue, if it doesn't exist yet.
3. Serialize the Fix It task.

www.dbooks.org

4. Call 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()
    {
        CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
        await queue.CreateIfNotExistsAsync();

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

Here we are using the Json.NET library to serialize the fixit to JSON format. You can use whatever serialization approach you prefer. JSON has the advantage of being human-readable, while being 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 start-up, 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 backend service. The backend service will run an infinite loop that performs the following steps:

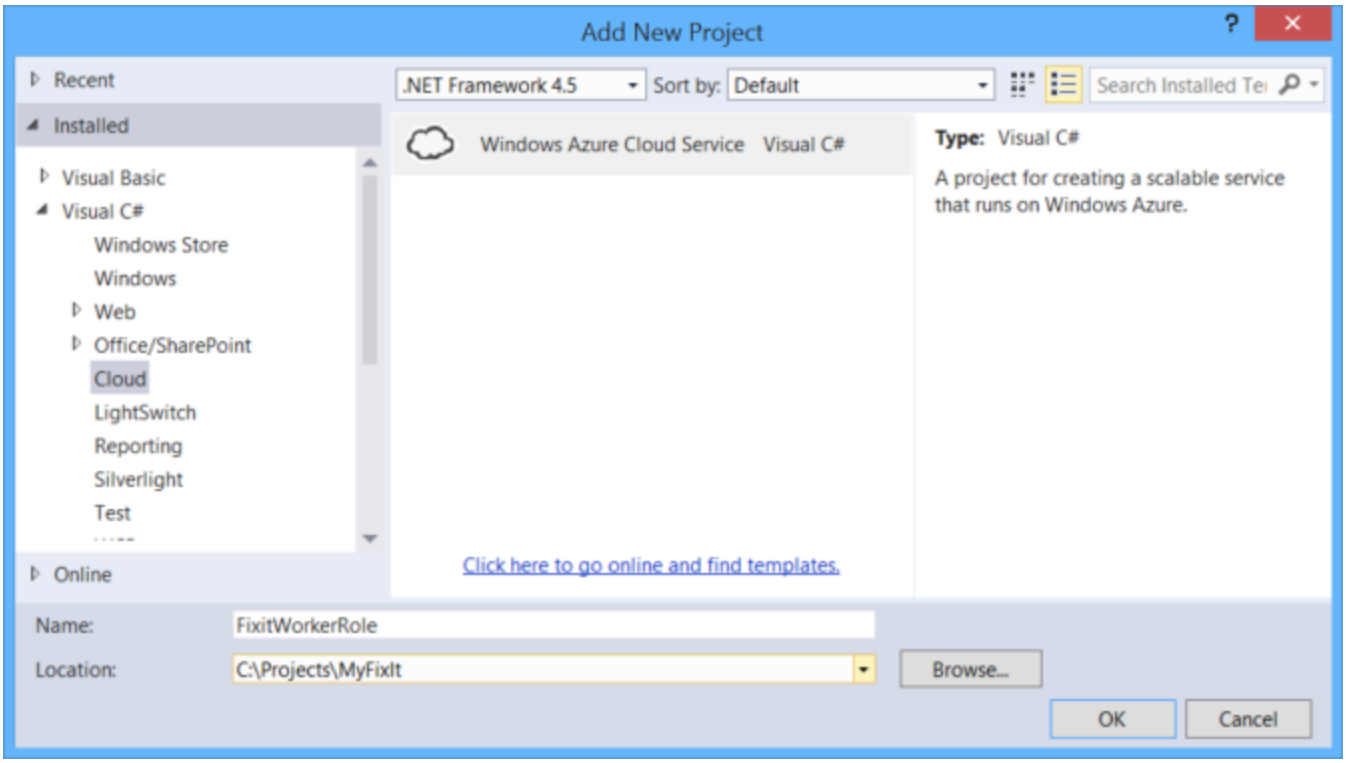
1. Get the next message from the queue.
2. Deserialize the message to a Fix It task.
3. Write the Fix It task to the database.

To host the backend service, we'll create a Windows Azure Cloud Service that contains a *worker role*. A worker role consists of one or more VMs that can do backend 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.

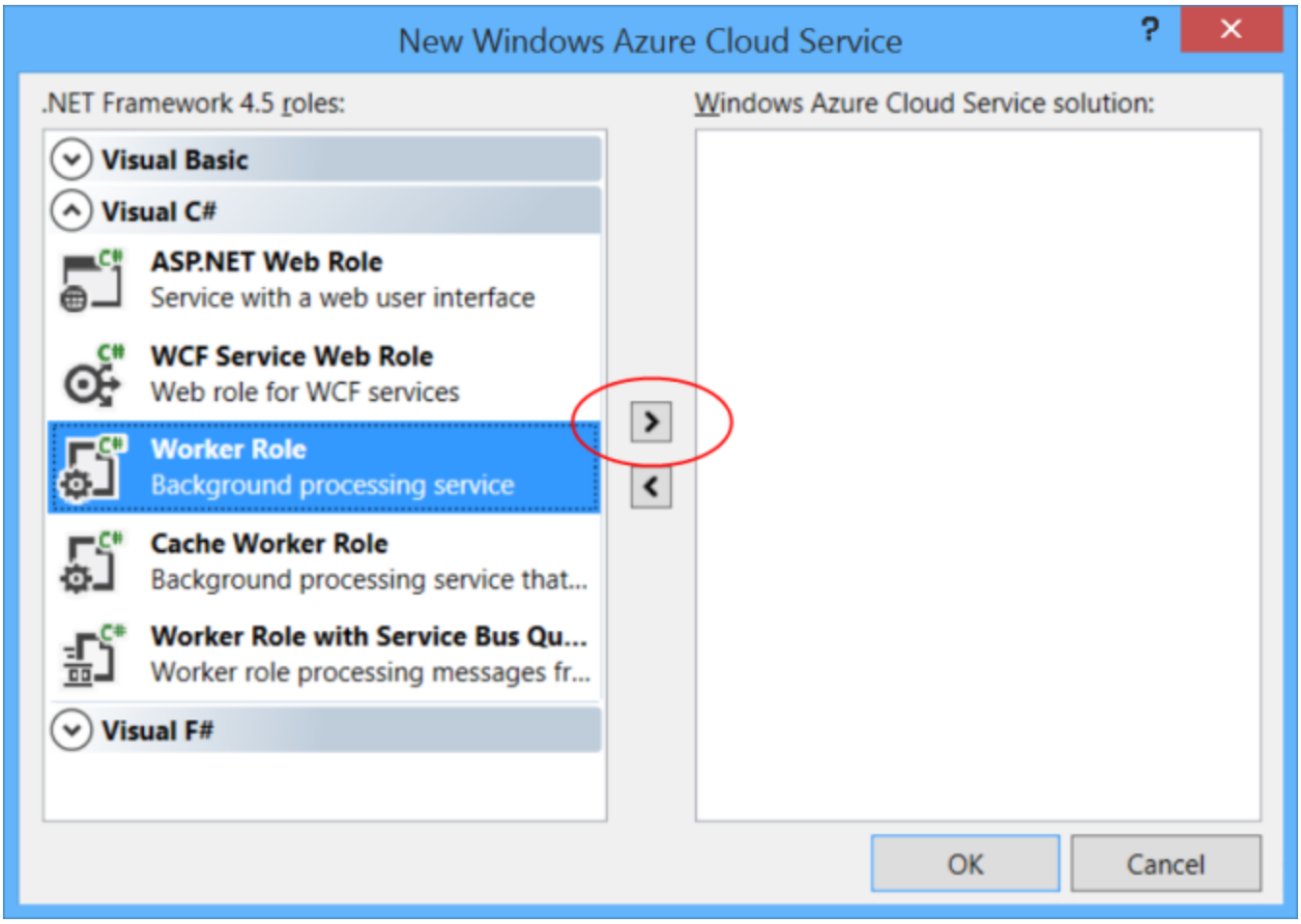
The following steps show how to add a worker role project to a solution that has a standard web project. These steps have already been done in the Fix It project that you can download.

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**.

www.dbooks.org



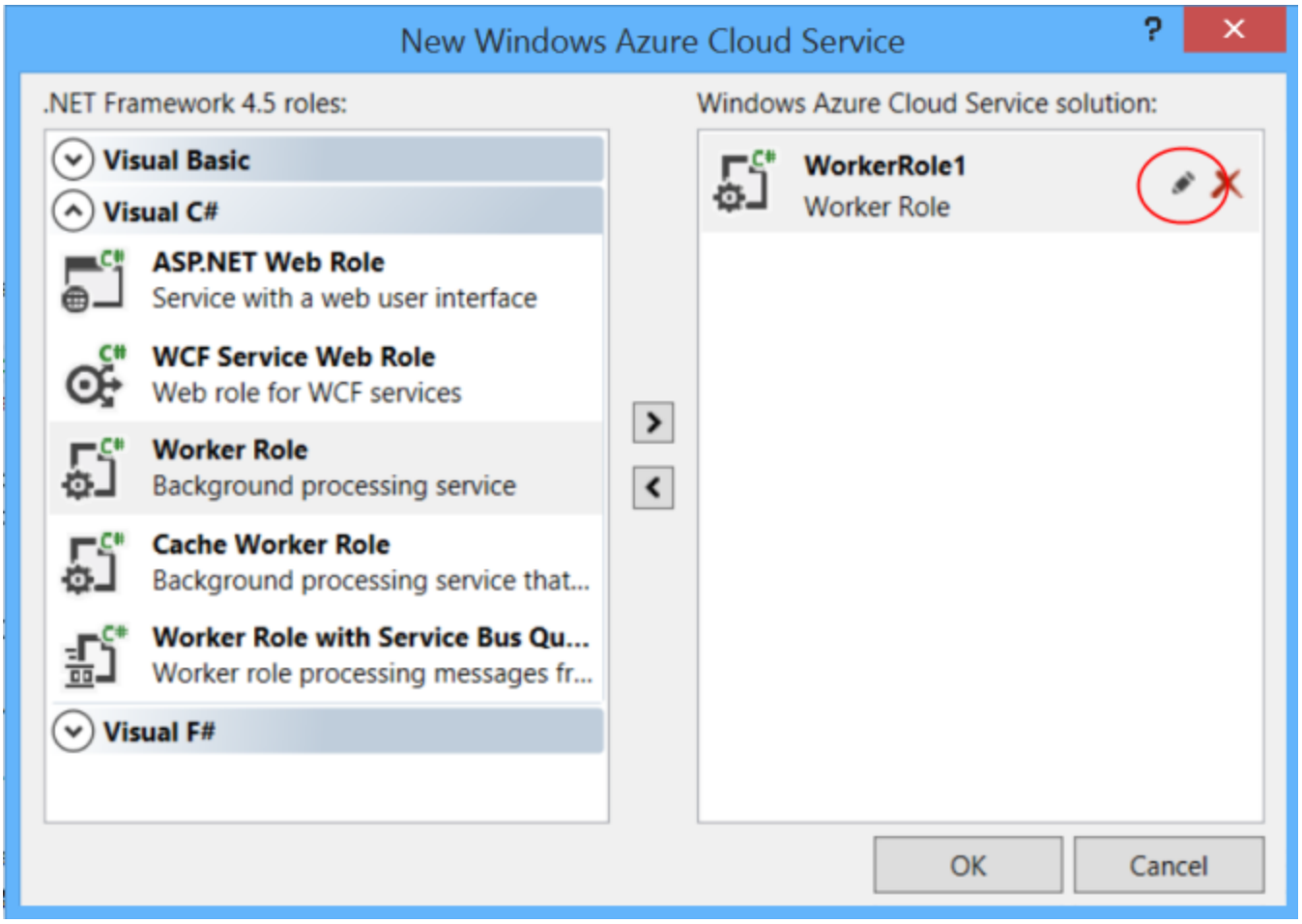
In the **New Windows Azure Cloud Service** dialog, expand the **Visual C#** node on the left pane. Select **Worker Role** and click the right-arrow icon.



(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 a Windows Azure Web Site. That has some advantages in making connections between front-end and back-end easier to coordinate. However, to keep this demo simple, we're keeping the front-end in a Web Site and only running the back-end in a Cloud Service.)

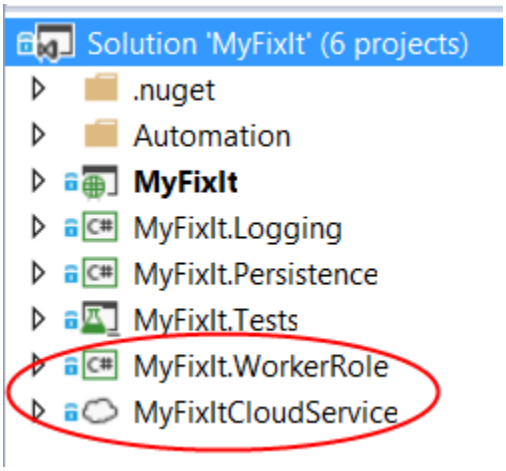
A default name is assigned to the worker role. To change the name, hover the mouse over the worker role in the right pane, then click the pencil icon.

www.dbooks.org



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

- A Windows Azure project that defines the cloud service, including configuration information.
- A worker role project that defines the worker role.



For more information, see [Creating a Windows Azure Project with Visual Studio.]

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>();
            while (!token.IsCancellationRequested)
            {
                try
                {
                    await queueManager.ProcessMessagesAsync();
                }
                catch (Exception ex)
                {
                    logger.Error(ex, "Exception in worker role Run loop.");
                }
                await Task.Delay(1000);
            }
        }
    }
    // Other code not shown.
}
```

The `ProcessMessagesAsync` method checks if there's a message waiting. If there is one, it deserializes the message into a `FixItTask` entity and saves the entity in the database. It loops until the queue is empty.

```
public async Task ProcessMessagesAsync()
{
    CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
    await queue.CreateIfNotExistsAsync();
    while (true)
    {
        CloudQueueMessage message = await queue.GetMessageAsync();
        if (message == null)
        {
            break;
```


www.dbooks.org

```
        }
        FixItTask fixit =
JsonConvert.DeserializeObject<FixItTask>(message.AsString);
        await _repository.CreateAsync(fixit);
        await queue.DeleteMessageAsync(message);
    }
}
```

Polling for queue messages incurs a small transaction charge, so when there's no message waiting to be processed, the worker role's `RunAsync` method waits a second before polling again by calling `Task.Delay(1000)`.

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 e-book, 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:

- Executing Background Tasks, chapter 5 of Moving Applications to the Cloud, 3rd Edition from Microsoft Patterns and Practices. (In particular, the section "Using Windows 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 Windows 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 and Practices - Windows Azure Guidance. See Asynchronous Messaging primer, Pipes and Filters pattern, Compensating Transaction pattern, Competing Consumers pattern, CQRS pattern.
- CQRS Journey. E-book about CQRS by Microsoft Patterns and Practices.

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 Microsoft Customer Advisory Team

(CAT) experience with actual customers. For an introduction to the Windows Azure Storage service and queues, see episode 5 starting at 35:13.

www.dbooks.org

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. E-book by Microsoft Patterns and Practices.
  - 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 Windows Azure Web Sites.
  - Moving Apps to the Azure: What changes? Short video by Stefan Schackow, explains file system access in Windows Azure Web Sites.
- Security, authentication, and authorization issues unique to cloud applications
  - Windows Azure Security Guidance
  - Windows Azure Network Security
  - Microsoft Patterns and Practices - Windows Azure Guidance. See Gatekeeper pattern, Federated Identity pattern.

See also additional cloud computing patterns and guidance at Microsoft Patterns and Practices - Windows Azure Guidance.

Resources

Each of the chapters in this e-book 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 Windows Azure.

Documentation

- Best Practices for the Design of Large-Scale Services on Windows 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. Web page version of the FailSafe video series.
- Windows Azure Architecture. Portal page on WindowsAzure.com for official documentation related to designing applications for Windows Azure.

Videos

- Building Real World Cloud Apps with Windows Azure - Part 1 and Part 2. Video of the presentation by Scott Guthrie that this e-book 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 e-book.
- 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 still actively being developed, making it a better choice for information on many topics than similar but older white papers.

Hard copy books

- Cloud Architecture Patterns: Using Microsoft Azure. Book by Bill Wilder.
- Release It! Design and Deploy Production-Ready Software. Book by Michael T. Nygard.

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

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.

www.dbooks.org

- 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 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
- 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
- Pawel Wilkosz

Appendix: The Fix It Sample Application

Download Sample Application: The Fix It Project

This appendix to the Building Real World Cloud Apps with Windows Azure e-book contains the following sections that provide additional information about the Fix It sample application that you can download:

- Known issues
- Best practices
- How to run the app from Visual Studio on your local computer
- How to deploy the base app to a Windows Azure Web Site by using the Windows PowerShell scripts
- Troubleshooting the Windows PowerShell scripts
- How to deploy the app with queue processing to a Windows Azure Web Site and a Windows Azure Cloud Service

Known issues

The Fix It app was originally developed in order to illustrate as simply as possible some of the patterns presented in this e-book. However, since the e-book 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 in the initial release of the Fix It sample application.

Security

- Ensure that you can't assign a task to a non-existent owner.
- Ensure that you can only view and modify 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. For example, the image size / image file size allowed for upload should be limited.

Administrator functionality

www.dbooks.org

An administrator should be able to change ownership on 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 in order 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 get processed twice, resulting in a duplicate task in the database. For more information about this issue, see Using Windows Azure Storage Queues.
- The queue polling logic could be 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 's'), which gets multiple messages in a single transaction. The transaction costs for Windows 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 multi-core 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 (e.g., 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 that runs entirely in a Windows Azure Web Site. We haven't provided scripts for setting up and deploying to the Web Site 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 there's something we learned 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)
            {
```


www.dbooks.org

```
                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

Since 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, and in order 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 only shows 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);
```


www.dbooks.org

```
    return null;
}
```

This would make it look to the user as if the query succeeded but just didn't return any rows. Solution is to re-throw 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

In order 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 web page 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 you 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 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

www.dbooks.org

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 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 than 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 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 backend service to process work items. The queue pattern is described in the chapter 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 Windows 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, then 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 Windows 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.

www.dbooks.org

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 will run the MyFixItCloudService project inside the Windows Azure compute emulator.

Using Visual Studio 2013:

1. Start the Windows Azure storage emulator.
2. Start Visual Studio with administrator privileges. (The Windows 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 Windows 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** dropdown list, select **Start**.
6. Click **OK**.
7. Press F5 to debug both projects.

When you debug MyFixItCloudService, Visual Studio will start the Windows 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 a Windows Azure Web Site 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 Windows Azure and deploy the project to the new environment. The following instructions explain how to use the scripts.

If you want to run in Windows Azure without using queues, and you made the changes to run locally with queues, make sure 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 a Windows Azure account or have a Windows Azure subscription that you are authorized to manage.

1. Install the **Windows Azure PowerShell** console. For instructions, see How to install and configure Windows Azure PowerShell.

   This customized console is configured to work with your Windows Azure subscription. The Windows Azure module is installed in the *Program Files* directory and is automatically imported on every use of the Windows 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 Windows Azure module or use a command in the Windows Azure module to trigger automatic importing of the module.

2. Start Windows Azure PowerShell with the **Run as administrator** option.
3. Run the Set-ExecutionPolicy cmdlet to set the Windows 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 unblock 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
   ```


www.dbooks.org

These credentials expire after a period of time and you have to re-run the `Add-AzureAccount` cmdlet. As this e-book 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 Windows 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 in order 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 Windows 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 in order 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 quotes 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 Windows Azure Web Site 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 lower-case 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 Windows Azure.

For example, if you want to create a Web Site 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 web site 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 Windows Azure PowerShell.

After the script finishes, you can use the Windows Azure Management Portal to see the resources that were created, as shown in the Automate Everything chapter.

10. To deploy the Fix It project to the new Windows Azure environment, use the *AzureWebsite.ps1* script. For example:

www.dbooks.org

```
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 Windows Azure.

Troubleshooting the Windows PowerShell scripts

The most common errors encountered when running these scripts are related to permissions. Make sure that `Add-AzureAccount` and `Import-AzurePublishSettingsFile` were successful and that you used them for the same Windows 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 web site name, the script will fail because the web site 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 a Windows Azure Web Site and a Windows Azure Cloud Service

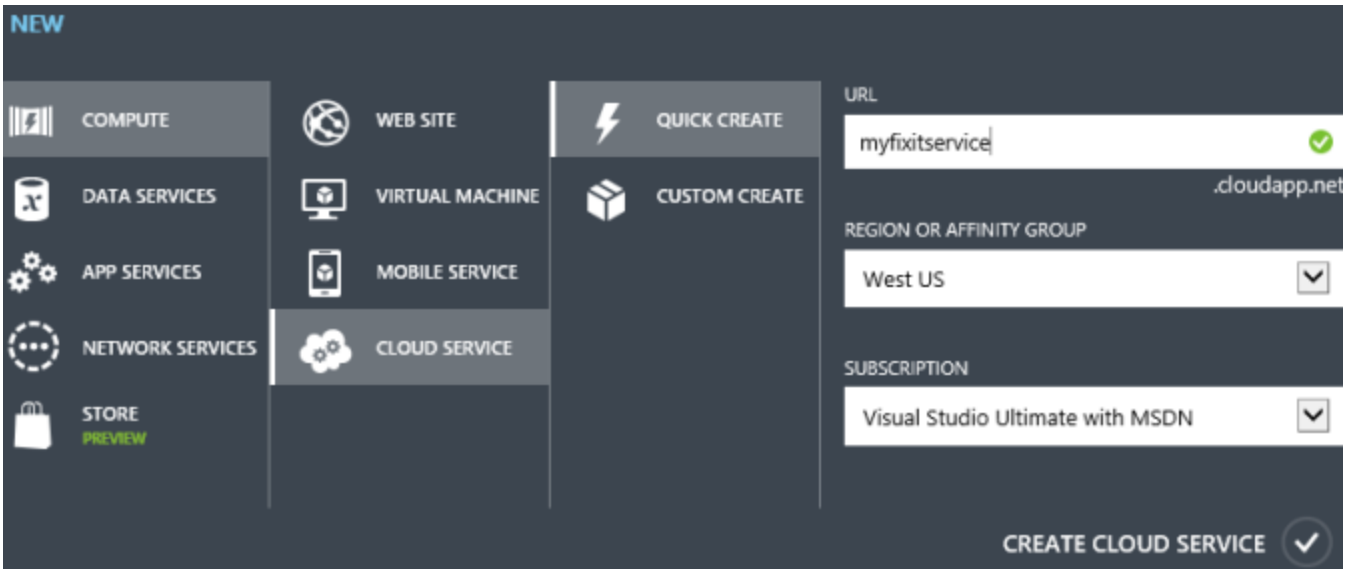
To enable queues, make the following change in the MyFixIt\Web.config file. Under `appSettings`, change the value of `UseQueues` to “true”:

```
<appSettings>
    <!-- Other settings not shown -->
    <add key="UseQueues" value="true"/>
</appSettings>
```

Then deploy the MVC application to a Windows Azure Web Site, as described in "How to deploy the base app to a Windows Azure Web Site by using the included Windows PowerShell scripts".

Next, create a new Windows Azure cloud service. The scripts included with the Fix It app do not create or deploy the cloud service, so you must use Azure 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 web site.

www.dbooks.org



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};Accoun
tKey={StorageAccountAccessKey}" />
  </ConfigurationSettings>
```

Now you are ready to deploy the cloud service. In Solution Explore, right-click the MyFixItCloudService project and select **Publish**. For more information, see "Deploy the Application to Windows Azure", which is in part 2 of this tutorial.

www.dbooks.org