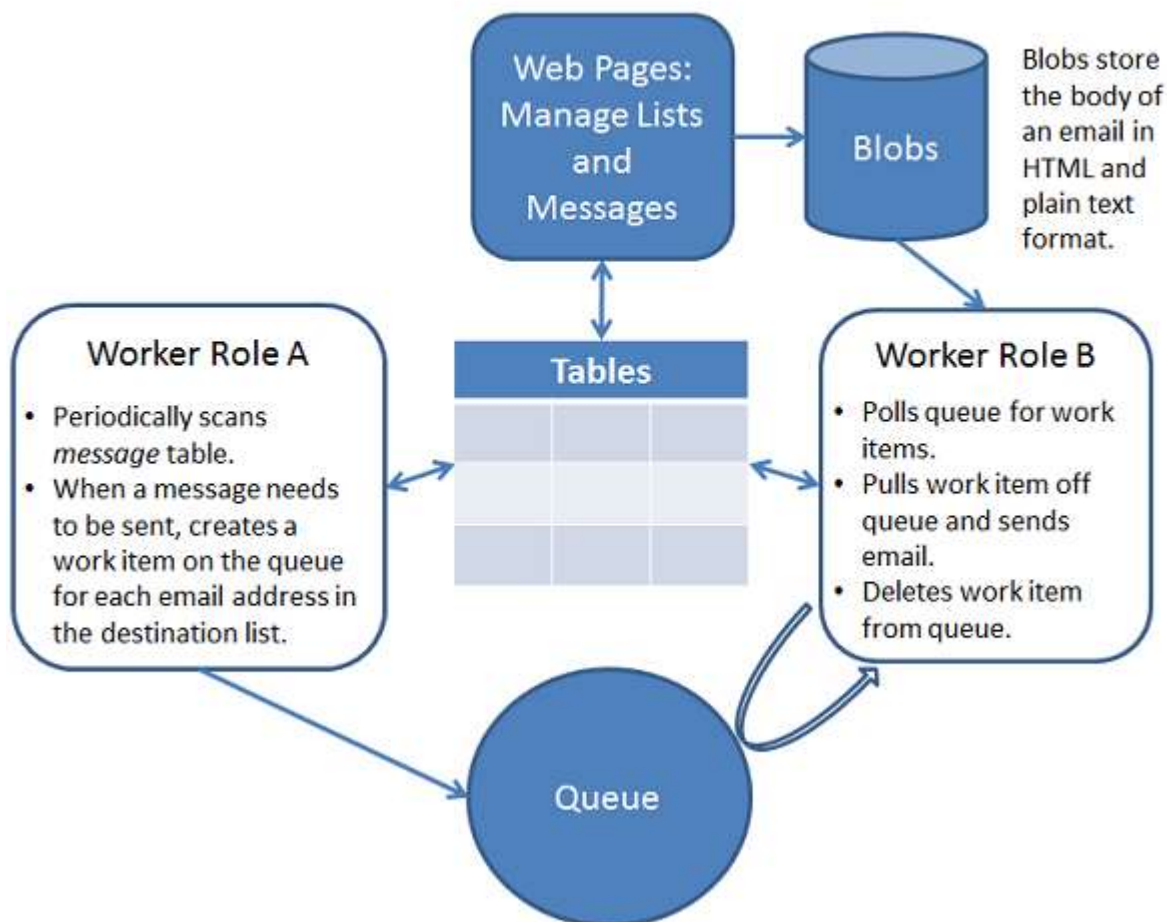


Azure Cloud Service Tutorial: ASP.NET MVC Web Role, Worker Role, and Azure Storage Tables, Queues, and Blobs - 1 of 5

This tutorial series shows how to create and deploy a multi-tier ASP.NET MVC web application that runs in an Azure cloud service and uses Azure Storage tables, queues, and blobs. You can download [the completed application](#) from the MSDN Code Gallery or an [e-book](#) of an earlier version from the TechNet E-Book Gallery.

Here's a diagram that shows how the parts of the application interact:

Email Message Processing



This is a 5-part tutorial series. If you want a quicker, simpler introduction to cloud services, queues, and blobs, see [Get Started with Azure Cloud Services and ASP.NET](#). As an alternative you can run a multi-tier application in Websites and WebJobs; for more information, see [Get Started with the Azure WebJobs SDK](#).

In this tutorial series you'll learn:

How to enable your machine for Azure development by installing the Azure SDK.

How to create a Visual Studio cloud project with an ASP.NET MVC web role and two worker roles.

How to publish the cloud project to an Azure Cloud Service.

How to use the Azure Queue storage service for communication between tiers or between worker roles.

How to use the Azure Table storage service as a highly scalable data store for structured, non-relational data.

How to use the Azure Blob service to store files in the cloud.

How to view and edit Azure tables, queues, and blobs by using Visual Studio Server Explorer.

How to use SendGrid to send emails.

How to configure tracing and view trace data.

How to scale an application by increasing the number of worker role instances.

Tutorials in the Series

There are five tutorials in the series:

1. Introduction to the Azure Email Service application (this tutorial). An in-depth look at the application and its architecture. You can skip this if you just want to see how to deploy or you want to see the code, and you can come back here later to better understand the architecture.
2. [Configuring and Deploying the Azure Email Service application](#). How to download the sample application, configure it, test it locally, deploy it, and test it in the cloud.
3. [Building the web role for the Azure Email Service application](#). How to build the MVC components of the application and test them locally.
4. [Building worker role A \(email scheduler\) for the Azure Email Service application](#). How to build the back-end component that creates queue work items for sending emails, and test it locally.
5. [Building worker role B \(email sender\) for the Azure Email Service application](#). How to build the back-end component that processes queue work items for sending emails, and test it locally.

Prerequisites

The instructions in these tutorials work for the following products:

Visual Studio 2013 with Update 2

Visual Studio 2013 Express for Web with Update 2

You also need an Azure subscription. You can create a [free trial account](#) or [activate MSDN subscriber benefits](#).

Front-end overview

The application is an email list service. The front-end includes web pages that administrators of the service use to manage email lists.

(The screen shots show Visual Studio 2012 template style; the content is the same for Visual Studio 2013 but the style is different.)

←

→

http://127.0.0.1/

Mailing Lists - Azure Email ...

★

⚙

Windows Azure Email Service

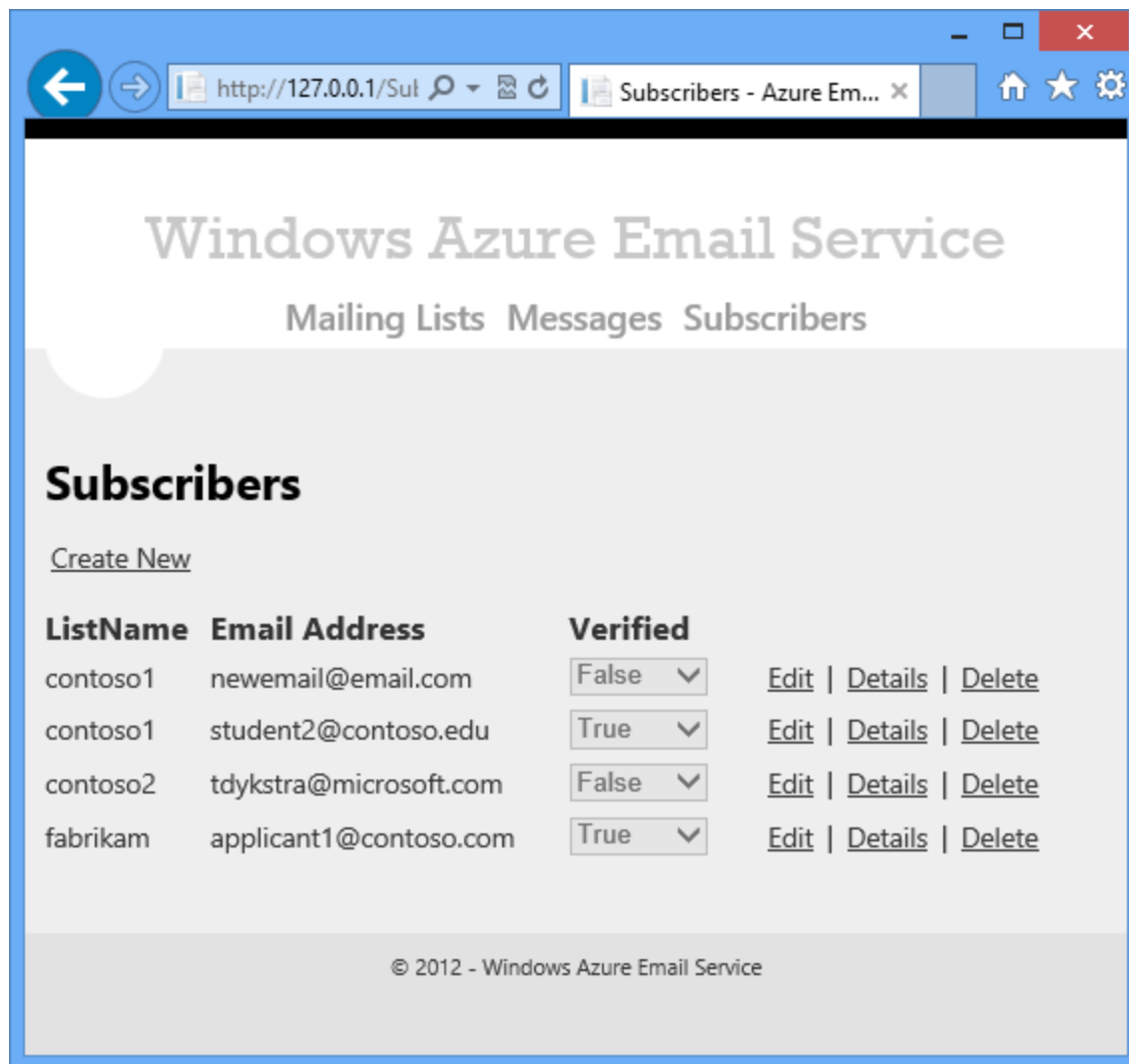
Mailing ListsMessagesSubscribers

Mailing Lists

Create New

ListName	Description	'From' Email Address	
contoso1	Contoso University History Department announcements	donotreply@contoso.com	Edit Delete
contoso2	Contoso University Art Department announcements	donotreply@contoso.com	Edit Delete
fabrikam	Fabrikam Engineering job postings	donotreply@fabrikam.com	Edit Delete
wingtip1	Wingtip Toys sale notifications	donotreply@wingtip.com	Edit Delete

© 2012 - Windows Azure Email Service



There is also a set of pages that administrators use to create messages to be sent to an email list.

http://127.0.0.1/Me

Messages - Azure Email Ser... x

Windows Azure Email Service

Mailing ListsMessagesSubscribers

Messages

Create New

List Name	Subject Line	Scheduled Date	Status	
contoso1	Guest lecture by Nino Olivetto	10/8/2012	Complete	Details Delete
fabrikam	Now hiring programming writers!	10/31/2012	Pending	Edit Details Delete
contoso1	Olivetto lecture canceled	10/8/2012	Processing	Details Delete

© 2012 - Windows Azure Email Service

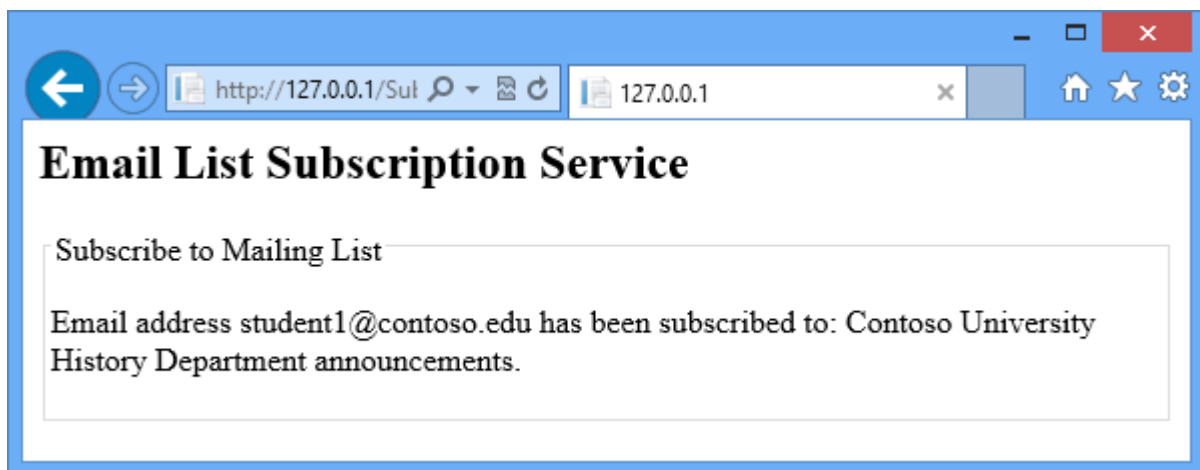
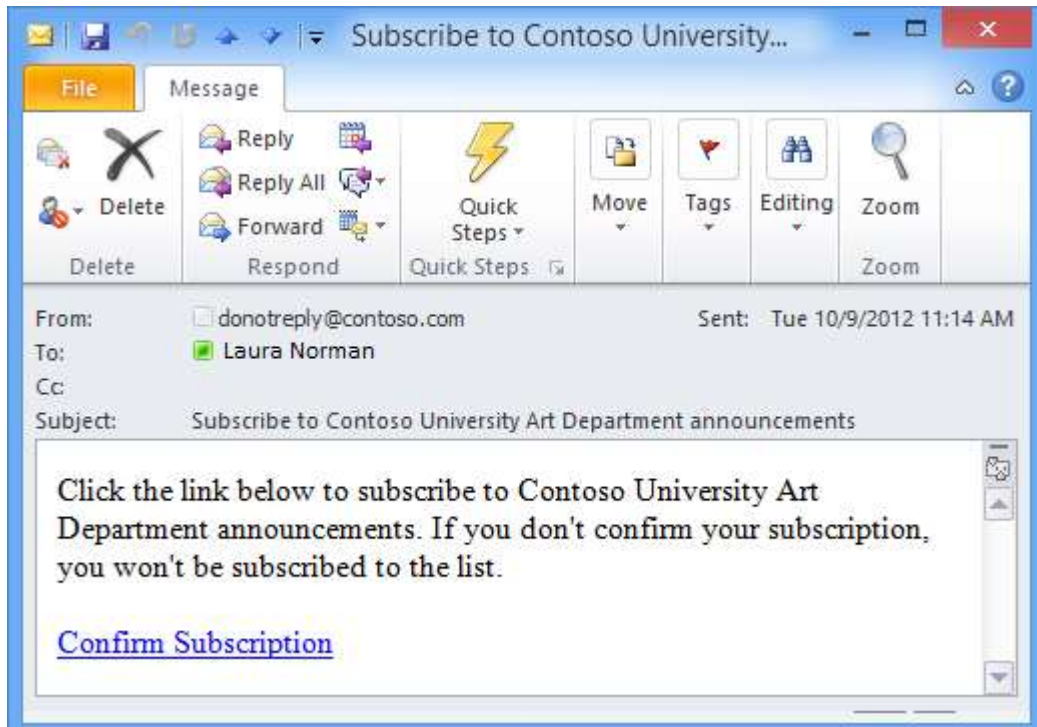
The screenshot shows a web browser window with the address bar displaying `http://127.0.0.1/Me`. The page title is "Create Message - Azure Em...". The main heading is "Windows Azure Email Service" with sub-navigation links for "Mailing Lists", "Messages", and "Subscribers". The page content is titled "Create Message" and includes the following fields:

- List Name:** A dropdown menu showing "Contoso University History Department announcements".
- Subject Line:** A text input field containing "Guest lecture by Nino Olivetto".
- HTML Path:** A text input field containing "C:\rm\olivettolecture.html" and a "Browse..." button.
- Text Path:** A text input field containing "C:\rm\olivettolecture.txt" and a "Browse..." button.
- Scheduled Date:** A text input field containing "10/8/2012".

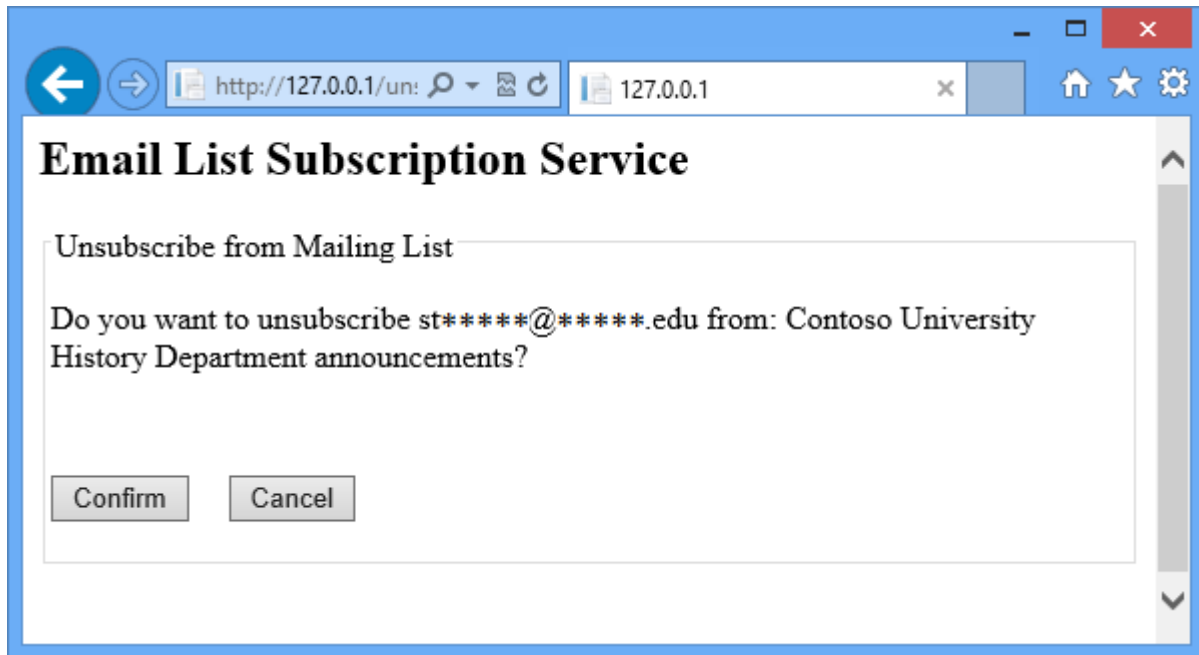
At the bottom of the form is a "Create" button. Below the form is a link labeled "Back to List". The footer of the page reads "© 2012 - Windows Azure Email Service".

Clients of the service are companies that give their customers an opportunity to sign up for a mailing list on the client website. For example, an administrator sets up a list for Contoso University History Department announcements. When a student interested in History Department announcements clicks a link on the Contoso University website, Contoso University makes a web service call to the Azure Email Service application. The service method causes an email to be sent to the customer. That email contains a hyperlink, and when the

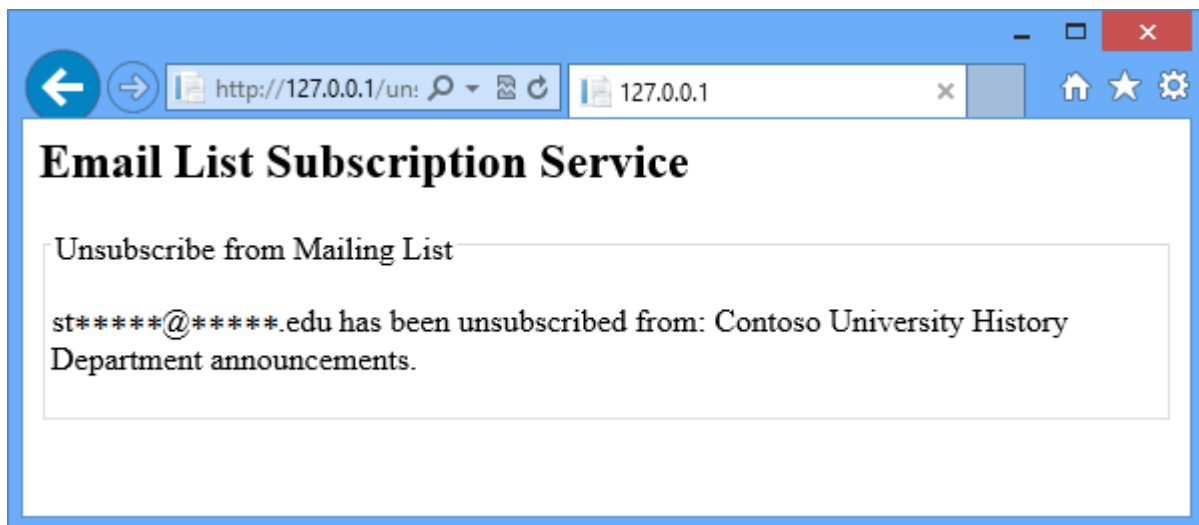
recipient clicks the link, a page welcoming the customer to the History Department Announcements list is displayed.



Every email sent (except the subscribe confirmation) includes an unsubscribe hyperlink. If a recipient clicks the link, a web page asks for confirmation of intent to unsubscribe.



If the recipient clicks the Confirm button, a page is displayed confirming that the person has been removed from the list.



Back-end overview

The front-end stores email lists and messages to be sent to them in Azure tables. When an administrator schedules a message to be sent, a table row containing the scheduled date and other data such as the subject line is added to the message table. A worker role periodically scans the message table looking for messages that need to be sent (we'll call this worker role A).

When worker role A finds a message needing to be sent, it does the following tasks:

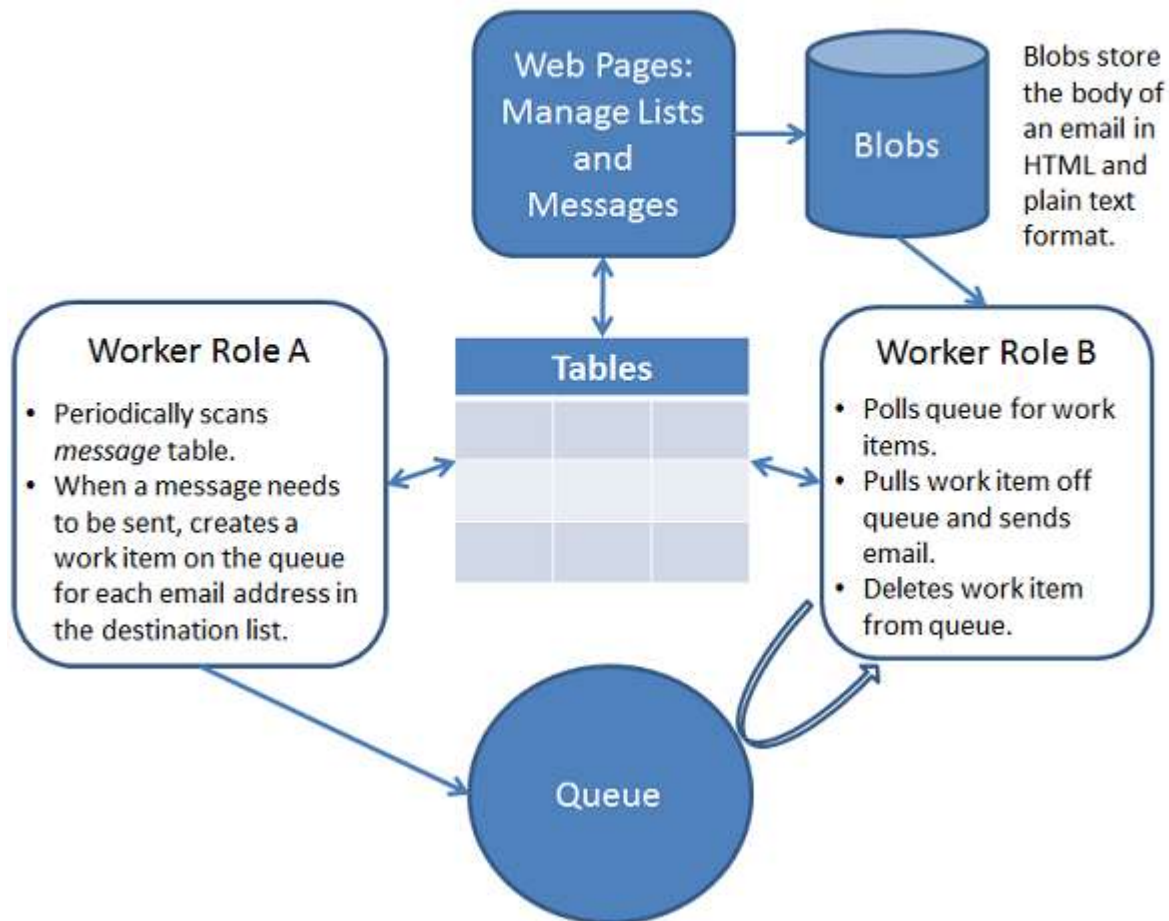
Gets all the email addresses in the destination email list.

Puts the information needed to send each email in the `message` table.

Creates a queue work item for each email that needs to be sent.

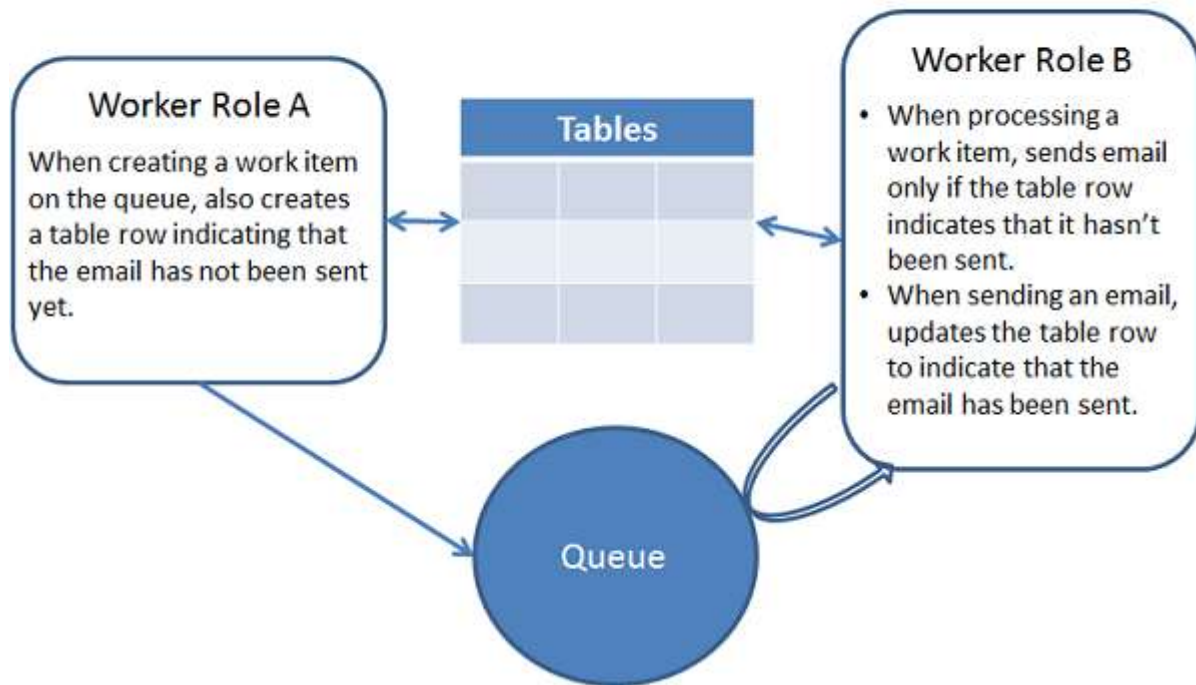
A second worker role (worker role B) polls the queue for work items. When worker role B finds a work item, it processes the item by sending the email, and then it deletes the work item from the queue. The following diagram shows these relationships.

Email Message Processing



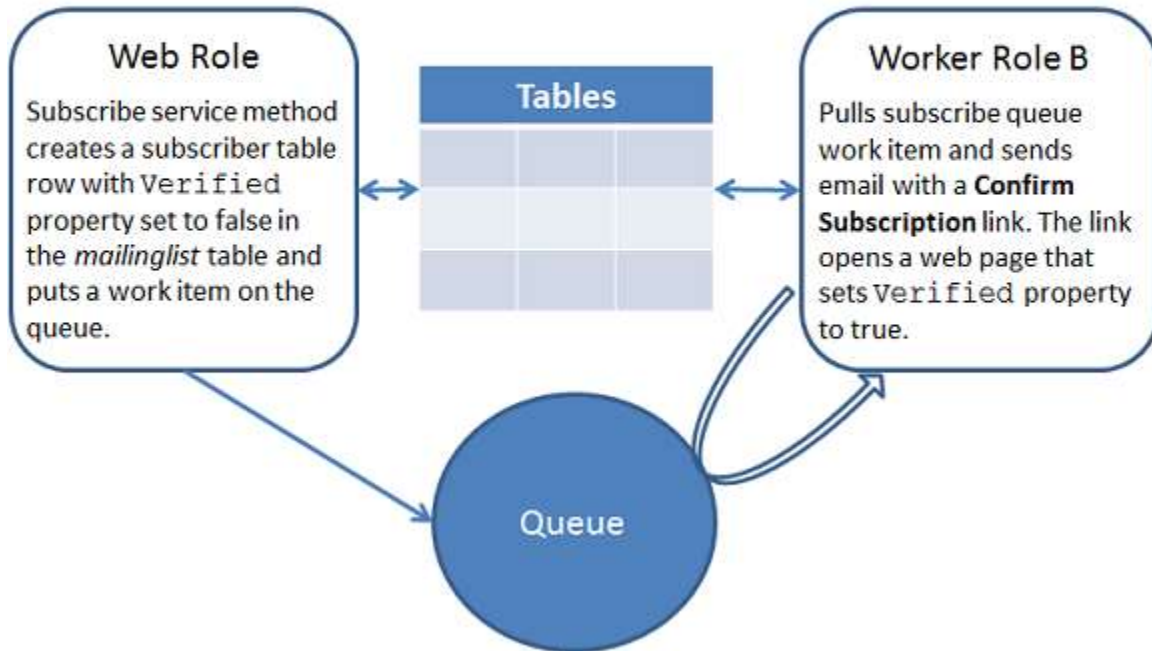
No emails are missed if worker role B goes down and has to be restarted, because a queue work item for an email isn't deleted until after the email has been sent. The application also prevents multiple emails from getting sent in case worker role A goes down and has to be restarted.

Preventing Duplicate Emails



Worker role B polls a subscription queue for work items put there by the Web API service method for new subscriptions. When it finds one, it sends the confirmation email.

Subscription Process



Azure Tables

The Azure Email Service application stores data in Azure Storage tables. Azure tables are a NoSQL data store, not a relational database like [Azure SQL Database](#). Azure tables are a good choice when efficiency and scalability are more important than data normalization and relational integrity. For example, in this application, one worker role creates a row every time a queue work item is created, and another one retrieves and updates a row every time an email is sent, which might become a performance bottleneck if a relational database were used. Additionally, Azure tables are cheaper than Azure SQL. For more information about Azure tables, see [the last tutorial in this series](#).

The following sections describe the contents of the Azure tables that are used by the Azure Email Service application. For a diagram that shows the tables and their relationships, see the [Azure Email Service data diagram](#) later in this page.

mailinglist table

The `mailinglist` table stores information about mailing lists and the subscribers to mailing lists. (An Azure table naming convention best practice is to use all lower-case letters.)

In Azure tables, different rows can have different schemas, and this flexibility is commonly used to make one table store data that would require multiple tables in a relational database. For example, to store mailing list data in SQL Database you could use three tables: a `mailinglist` table that stores information about the list, a `subscriber` table that stores information about subscribers, and a `mailinglistsubscriber` table that associates mailing lists with subscribers and vice versa. In the NoSQL table in this application, all of those functions are rolled into one table named `mailinglist`.

In an Azure table, every row has a *partition key* and a *row key* that uniquely identifies the row. The partition key divides the table up logically into partitions. Within a partition, the row key uniquely identifies a row. There are no secondary indexes; therefore to make sure that the application will be scalable, it is important to design your tables so that you can always query by partition key and row key.

The partition key for the `mailinglist` table is the name of the mailing list.

The row key for the `mailinglist` table can be one of two things: the constant "mailinglist" or the email address of the subscriber. Rows that have row key "mailinglist" include information about the mailing list. Rows that have the email address as the row key have information about the subscribers to the list.

In other words, rows with row key "mailinglist" are equivalent to a `mailinglist` table in a relational database. Rows with row key = email address are equivalent to a `subscriber` table and a `mailinglistsubscriber` association table in a relational database.

Making one table serve multiple purposes in this way facilitates better performance. In a relational database three tables would have to be read, and three sets of rows would have to be sorted and matched up against each other, which takes time. Here just one table is read and its rows are automatically returned in partition key and row key order.

The following grid shows row properties for the rows that contain mailing list information (row key = "mailinglist").

PROPERTY	DATA TYPE	DESCRIPTION
PartitionKey	String	ListName: A unique identifier for the mailing list, for example: contoso1. The typical use for the table is to retrieve all information for a specific mailing list, so using the list name is an efficient way to partition the table.
RowKey	String	The constant "mailinglist".
Description	String	Description of the mailing List, for example: "Contoso University History Department announcements".

FromEmailAddress	String	The "From" email address in the emails sent to this list, for example: donotreply@contoso.edu.
------------------	--------	--

The following grid shows row properties for the rows that contain subscriber information for the list (row key = email address).

PROPERTY	DATA TYPE	DESCRIPTION
PartitionKey	String	ListName: The name (unique identifier) of the mailing list, for example: contoso1.
RowKey	String	EmailAddress: The subscriber email address, for example: student1@contoso.edu.
SubscriberGUID	String	<p>Generated when the email address is added to a list. Used in subscribe and unsubscribe links so that it's difficult to subscribe or unsubscribe someone else's email address.</p> <p>Some queries for the Subscribe and Unsubscribe web pages specify only the PartitionKey and this property. Querying a partition without using the RowKey limits the scalability of the application, because queries will take longer as mailing list sizes increase. An option for improving scalability is to add lookup rows that have the SubscriberGUID in the RowKey property. For example, for each email address one row could have "email:student1@domain.com" in the RowKey and another row for the same subscriber could have "guid:6f32b03b-90ed-41a9-b8ac-c1310c67b66a" in the RowKey. This is simple to implement because atomic batch transactions on rows within a partition are easy to code. For more information, see Real World: Designing a Scalable Partitioning Strategy for Azure Table Storage</p>
Verified	Boolean	When the row is initially created for a new subscriber, the value is false. It changes to true only after the new subscriber clicks the Confirm hyperlink in the welcome email or an administrator sets it to true. If a message is sent to a list while the Verified value for one of its subscribers is false, no email is sent to that subscriber.

The following list shows an example of what data in the table might look like.

PARTITION KEY	contoso1
ROW KEY	mailinglist
DESCRIPTION	Contoso University History Department announcements
FROMEMAILADDRESS	donotreply@contoso.edu

PARTITION KEY	contoso1
ROW KEY	student1@domain.com
SUBSCRIBERGUID	6f32b03b-90ed-41a9-b8ac-c1310c67b66a
VERIFIED	true

PARTITION KEY	contoso1
ROW KEY	student2@domain.com
SUBSCRIBERGUID	01234567-90ed-41a9-b8ac-c1310c67b66a
VERIFIED	false

PARTITION KEY	fabrikam1
ROW KEY	mailinglist
DESCRIPTION	Fabrikam Engineering job postings
FROMEMAILADDRESS	donotreply@fabrikam.com

PARTITION KEY	fabrikam1
ROW KEY	applicant1@domain.com
SUBSCRIBERGUID	76543210-90ed-41a9-b8ac-c1310c67b66a
VERIFIED	true

message table

The `message` table stores information about messages that are scheduled to be sent to a mailing list. Administrators create and edit rows in this table using web pages, and the worker roles use it to pass information about each email from worker role A to worker role B.

The partition key for the `message` table is the date the email is scheduled to be sent, in `yyyy-mm-dd` format. This optimizes the table for the query that is executed most often against this table, which selects rows that have `ScheduledDate` of today or earlier. However, it does create a potential performance bottleneck, because Azure Storage tables have a maximum throughput of 500 entities per second for a partition. For each email to be sent, the application writes a `message` table row, reads a row, and deletes a row. Therefore the shortest possible time for processing 1,000,000 emails scheduled for a single day is almost two hours, regardless of how many worker roles are added in order to handle increased loads.

The row key for the `message` table can be one of two things: the constant "message" plus a unique key for the message called the `MessageRef`, or the `MessageRef` value plus the email address of the subscriber. Rows that have row key that begins with "message" include information about the message, such as the mailing list to send it to and when it should be sent. Rows that have the `MessageRef` and email address as the row key have all of the information needed to send an email to that email address.

In relational database terms, rows with row key that begins with "message" are equivalent to a `message` table. Rows with row key = `MessageRef` plus email address are equivalent to a join query view that contains `mailinglist`, `message`, and `subscriber` information.

The following grid shows row properties for the `message` table rows that have information about the message itself.

PROPERTY	DATA TYPE	DESCRIPTION
PartitionKey	String	The date the message is scheduled to be sent, in <code>yyyy-mm-dd</code> format.
RowKey	String	<p>The constant "message" concatenated with the <code>MessageRef</code> value. The <code>MessageRef</code> is a unique value created by getting the <code>Ticks</code> value from <code>DateTime.Now</code> when the row is created.</p> <p>Note: High volume multi-threaded, multi-instance applications should be prepared to handle duplicate <code>RowKey</code> exceptions when using <code>Ticks</code>. <code>Ticks</code> are not guaranteed to be unique.</p>
ScheduledDate	Date	The date the message is scheduled to be sent. (Same as <code>PartitionKey</code> but in <code>Date</code> format.)

SubjectLine	String	The subject line of the email.
ListName	String	The list that this message is to be sent to.
Status	String	<p>"Pending" -- Worker role A has not yet started to create queue messages to schedule emails.</p> <p>"Queuing" -- Worker role A has started to create queue messages to schedule emails.</p> <p>"Processing" -- Worker role A has created queue work items for all emails in the list, but not all emails have been sent yet.</p> <p>"Completed" -- Worker role B has finished processing all queue work items (all emails have been sent). Completed rows are archived in the <code>messagearchive</code> table, as explained later. We hope to make this property an <code>enum</code> in the next release.</p>

When worker role A creates a queue message for an email to be sent to a list, it creates an email row in the `message` table. When worker role B sends the email, it moves the email row to the `messagearchive` table and updates the `EmailSent` property to `true`. When all of the email rows for a message in `Processing` status have been archived, worker role A sets the status to `Completed` and moves the message row to the `messagearchive` table.

The following grid shows row properties for the email rows in the `message` table.

PROPERTY	DATA TYPE	DESCRIPTION
PartitionKey	String	The date the message is scheduled to be sent, in yyyy-mm-dd format.
RowKey	String	The <code>MessageRef</code> value and the destination email address from the <code>subscriber</code> row of the <code>mailinglist</code> table.
MessageRef	Long	Same as the <code>MessageRef</code> component of the <code>RowKey</code> .
ScheduledDate	Date	The scheduled date from the message row of the <code>message</code> table. (Same as <code>PartitionKey</code> but in <code>Date</code> format.)
SubjectLine	String	The email subject line from the message row of the <code>message</code> table.
ListName	String	The mailing list name from the <code>mailinglist</code> table.
From EmailAddress	String	The "from" email address from the <code>mailinglist</code> row of the <code>mailinglist</code> table.

EmailAddress	String	The email address from the subscriber row of the mailinglist table.
SubscriberGUID	String	The subscriber GUID from the subscriber row of the mailinglist table.
EmailSent	Boolean	False means the email has not been sent yet; true means the email has been sent.

There is redundant data in these rows, which you would typically avoid in a relational database. But in this case you are trading some of the disadvantages of redundant data for the benefit of greater processing efficiency and scalability. Because all of the data needed for an email is present in one of these rows, worker role B only needs to read one row in order to send an email when it pulls a work item off the queue.

You might wonder where the body of the email comes from. These rows don't have blob references for the files that contain the body of the email, because that value is derived from the MessageRef value. For example, if the MessageRef is 634852858215726983, the blobs are named 634852858215726983.htm and 634852858215726983.txt.

The following list shows an example of what data in the table might look like.

PARTITION KEY	2012-10-15
ROW KEY	message634852858215726983
MESSAGEREF	634852858215726983
SCHEDULEDDATE	2012-10-15
SUBJECTLINE	New lecture series
LISTNAME	contoso1
STATUS	Processing

PARTITION KEY	2012-10-15
ROW KEY	634852858215726983student1@contoso.edu
MESSAGEREF	634852858215726983

SCHEDULEDDATE	2012-10-15
SUBJECTLINE	New lecture series
LISTNAME	contoso1
FROMEMAILADDRESS	donotreply@contoso.edu
EMAILADDRESS	student1@contoso.edu
SUBSCRIBERGUID	76543210-90ed-41a9-b8ac-c1310c67b66a
EMAILSENT	true

PARTITION KEY	2012-10-15
ROW KEY	634852858215726983student2@contoso.edu
MESSAGEREF	634852858215726983
SCHEDULEDDATE	2012-10-15
SUBJECTLINE	New lecture series
LISTNAME	contoso1
FROMEMAILADDRESS	donotreply@contoso.edu
EMAILADDRESS	student2@contoso.edu
SUBSCRIBERGUID	12345678-90ed-41a9-b8ac-c1310c679876
EMAILSENT	true

PARTITION KEY	2012-11-15
---------------	------------

ROW KEY	message124852858215726999
MESSAGEREF	124852858215726999
SCHEDULEDDATE	2012-11-15
SUBJECTLINE	New job postings
LISTNAME	fabrikam
STATUS	Pending

messagearchive table

One strategy for making sure that queries execute efficiently, especially if you have to search on fields other than `PartitionKey` and `RowKey`, is to limit the size of the table. The query in worker role A that checks to see if all emails have been sent for a message needs to find email rows in the `message` table that have `EmailSent = false`. The `EmailSent` value is not in the `PartitionKey` or `RowKey`, so this would not be an efficient query for a message with a large number of email rows. Therefore, the application moves email rows to the `messagearchive` table as the emails are sent. As a result, the query to check if all emails for a message have been sent only has to query the `message` table on `PartitionKey` and `RowKey` because if it finds any email rows for a message at all, that means there are unsent messages and the message can't be marked `Complete`.

The schema of rows in the `messagearchive` table is identical to that of the `message` table. Depending on what you want to do with this archival data, you could limit its size and expense by reducing the number of properties stored for each row, and by deleting rows older than a certain age.

Azure Queues

Azure queues facilitate communication between tiers of this multi-tier application, and between worker roles in the back-end tier. Queues are used to communicate between worker role A and worker role B in order to make the application scalable. Worker role A could create a row in the `Message` table for each email, and worker role B could scan the table for rows representing emails that haven't been sent, but you wouldn't be able to add additional instances of worker role B in order to divide up the work. The problem with using table rows to coordinate the work between worker role A and worker role B is that you have no way of ensuring that only one worker role instance will pick up any given table row for processing. Queues give you that assurance.

When a worker role instance pulls a work item off a queue, the queue service makes sure that no other worker role instance can pull the same work item. This exclusive lease feature of Azure queues facilitates sharing a workload among multiple instances of a worker role.

Azure also provides the Service Bus queue service. For more information about Azure Storage queues and Service Bus queues, see [the last tutorial in this series](#).

The Azure Email Service application uses two queues, named `AzureMailQueue` and `AzureMailSubscribeQueue`.

AzureMailQueue

The `AzureMailQueue` queue coordinates the sending of emails to email lists. Worker role A places a work item on the queue for each email to be sent, and worker role B pulls a work item from the queue and sends the email.

A queue work item contains a comma-delimited string that consists of the scheduled date of the message (partition key to the message table) and the `MessageRef` and `EmailAddress` values (row key to the message table) values, plus a flag indicating whether the item is created after the worker role went down and restarted, for example:

<code>2012-10-15,634852858215726983,student1@contoso.edu,0</code>

Worker role B uses these values to look up the row in the message table that contains all of the information needed to send the email. If the restart flag indicates a restart, worker B makes sure the email has not already been sent before sending it.

When traffic spikes, the Cloud Service can be reconfigured so that multiple instances of worker role B are instantiated, and each of them can independently pull work items off the queue.

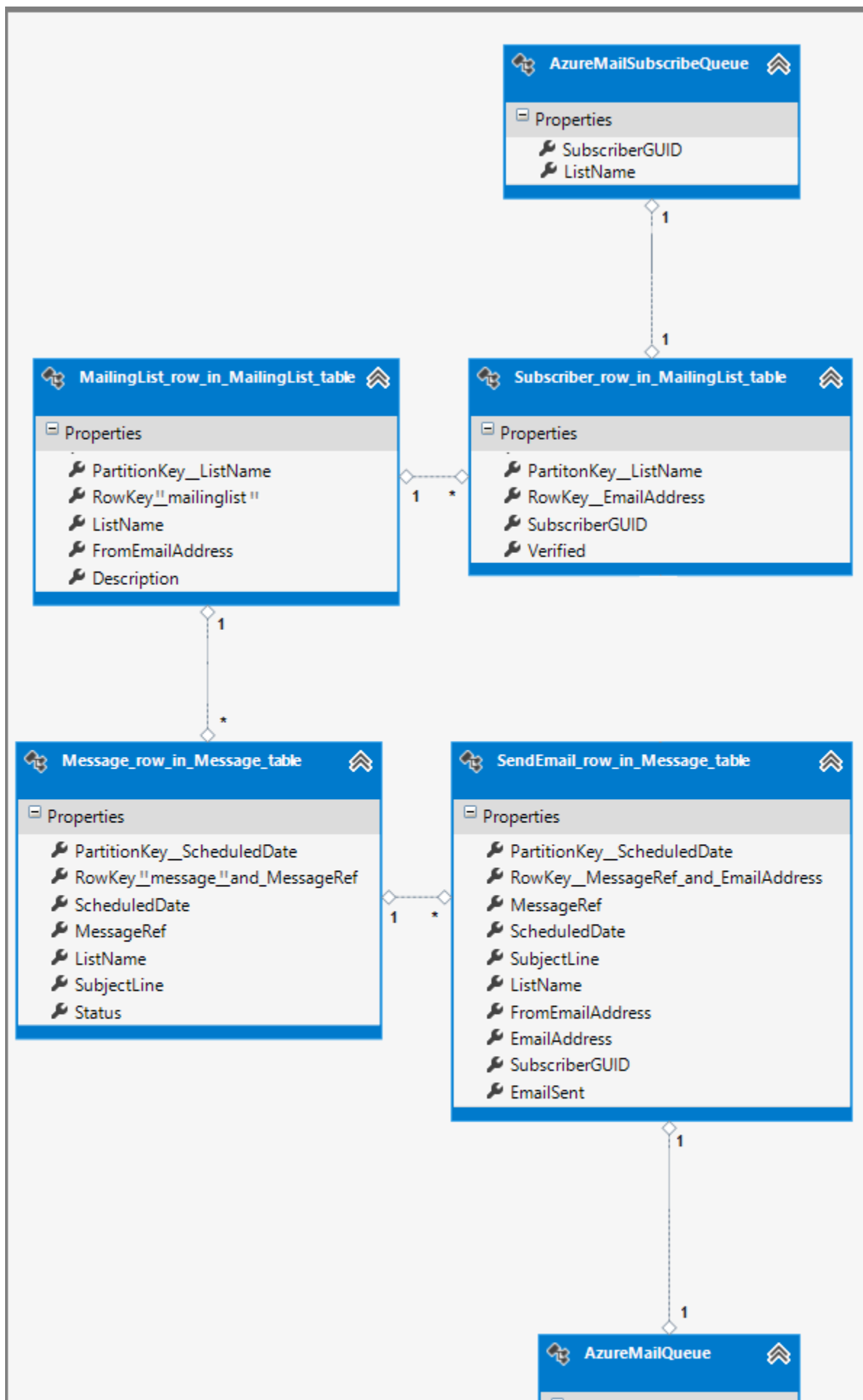
AzureMailSubscribeQueue

The `AzureMailSubscribeQueue` queue coordinates the sending of subscription confirmation emails. In response to a service method call, the service method places a work item on the queue. Worker role B pulls the work item from the queue and sends the subscription confirmation email.

A queue work item contains the subscriber GUID. This value uniquely identifies an email address and the list to subscribe it to, which is all that worker role B needs to send a confirmation email. As explained earlier, this requires a query on a field that is not in the `PartitionKey` or `RowKey`, which is inefficient. To make the application more scalable, the `mailinglist` table would have to be restructured to include the subscriber GUID in the `RowKey`.

Azure Email Service data diagram

The following diagram shows the tables and queues and their relationships.



Azure Blobs

Blobs are "binary large objects." The Azure Blob service provides a means for uploading and storing files in the cloud. For more information about Azure blobs, see [the last tutorial in this series](#).

Azure Mail Service administrators put the body of an email in HTML form in an *.htm* file and in plain text in a *.txt* file. When they schedule an email, they upload these files in the Create Message web page, and the ASP.NET MVC controller for the page stores the uploaded file in an Azure blob.

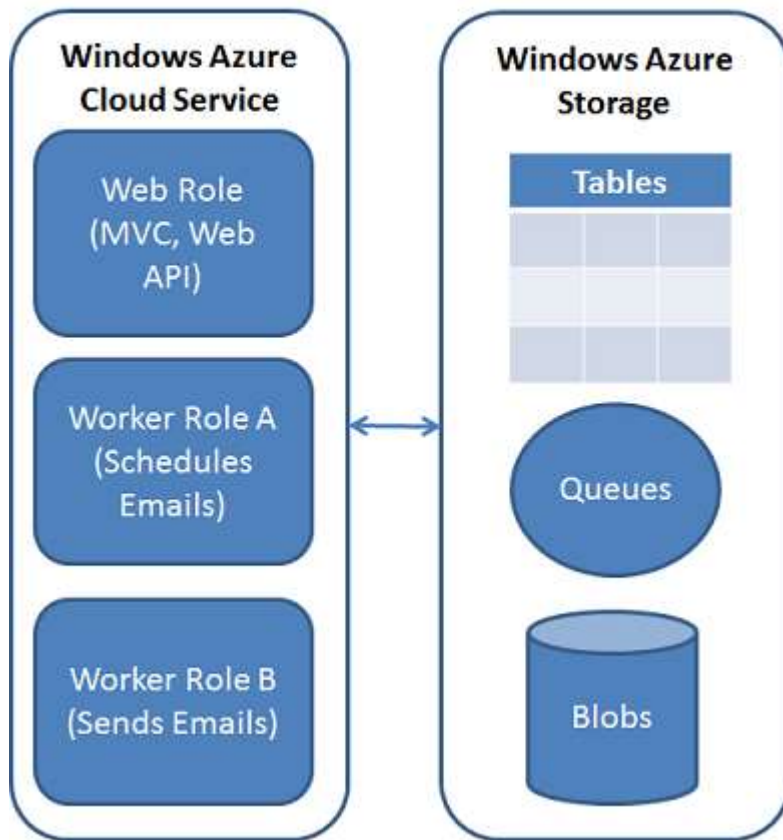
Blobs are stored in blob containers, much like files are stored in folders. The Azure Mail Service application uses a single blob container, named `azuremailblobcontainer`. The name of the blobs in the container is derived by concatenating the `MessageRef` value with the file extension, for example: `634852858215726983.htm` and `634852858215726983.txt`.

Since both HTML and plain text messages are essentially strings, we could have designed the application to store the email message body in string properties in the `Message` table instead of in blobs. However, there is a 64K limit on the size of a property in a table row, so using a blob avoids that limitation on email body size. (64K is the maximum total size of the property; after allowing for encoding overhead, the maximum string size you can store in a property is actually closer to 48k.)

Azure Cloud Service versus Azure Website

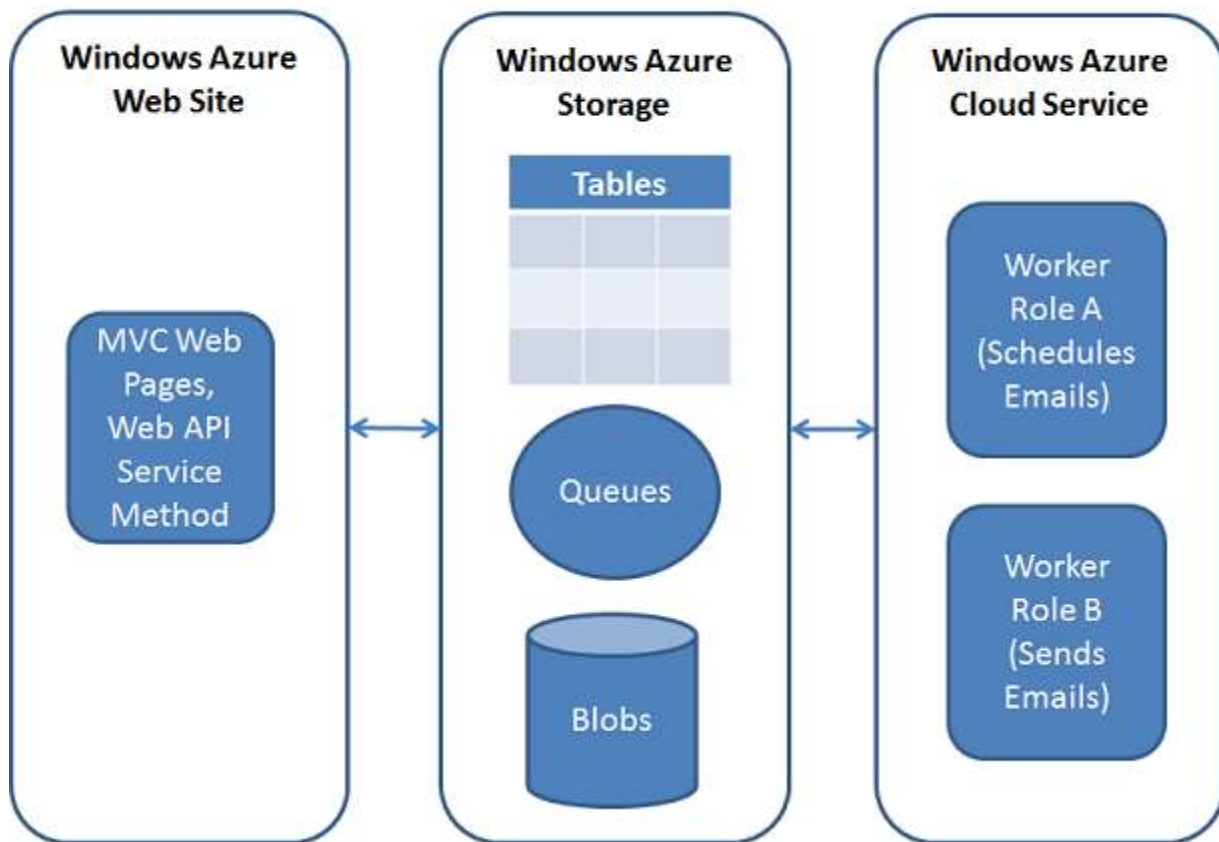
The Azure Email Service is configured so that the front-end and back-end all run in an Azure Cloud Service.

Default Architecture



An alternative architecture is to run the front-end in an Azure Website.

Alternative Architecture



Another alternative is to run the front-end in an Azure website and use the WebJobs feature to run the back-end in the same servers that run the front-end. For more information, see [Get Started with the Azure WebJobs SDK](#).

Cost

This section provides a brief overview of costs for running the sample application in Azure, given rates in effect when the tutorial was originally published in December of 2012. Before making any business decisions based on costs, be sure to check current rates on the following web pages:

[Azure Pricing Calculator](#)

[SendGrid Azure](#)

Costs are affected by the number of web and worker role instances you decide to maintain. In order to qualify for the [Azure Cloud Service 99.95% Service Level Agreement \(SLA\)](#), you must deploy two or more instances of each role. One of the reasons you must run at least two role instances is because the virtual machines that

run your application are restarted approximately twice per month for operating system upgrades. (For more information on OS Updates, see [Role Instance Restarts Due to OS Upgrades.](#))

The work performed by the two worker roles in this sample is not time critical and so does not need the 99.5% SLA. Therefore, running a single instance of each worker role is feasible so long as one instance can keep up with the work load. The web role instance is time sensitive, that is, users expect the website to not have any down time, so a production application should have at least two instances of the web role.

The following table shows the costs for the default architecture for the Azure Email Service sample application assuming a minimal workload. The costs shown are based on using an extra small (shared) virtual machine size. The default virtual machine size when you create a Visual Studio cloud project is small, which is about six times more expensive than the extra small size.

COMPONENT OR SERVICE	RATE	COST PER MONTH
Web role	2 instances at \$.02/hour for extra small instances	\$29.00
Worker role A (schedules emails to be sent)	1 instance at \$.02/hour for an extra small instance	\$14.50
Worker role B (sends emails)	1 instance at \$.02/hour for an extra small instance	\$14.50
Azure storage transactions	1 million transactions per month at \$0.10/million (Each query counts as a transaction; worker role A continuously queries tables for messages that need to be sent. The application is also configured to write diagnostic data to Azure Storage, and each time it does that is a transaction.)	\$0.10
Azure locally redundant storage	\$2.33 for 25 GB (Includes storage for application tables and diagnostic data.)	\$2.33
Bandwidth	5 GB egress is free	Free
SendGrid	Azure customers can send 25,000 emails per month for free	Free
Total		\$60.43

As you can see, role instances are a major component of the overall cost. Role instances incur a cost even if they are stopped; you must delete a role instance to not incur any charges. One cost saving approach would be

to move all the code from worker role A and worker role B into one worker role. For these tutorials we deliberately chose to implement two worker instances in order to simplify scale out. The work that worker role B does is coordinated by the Azure Queue service, which means that you can scale out worker role B simply by increasing the number of role instances. (Worker role B is the limiting factor for high load conditions.) The work performed by worker role A is not coordinated by queues, therefore you cannot run multiple instances of worker role A. If the two worker roles were combined and you wanted to enable scale out, you would need to implement a mechanism for ensuring that worker role A tasks run in only one instance. (One such mechanism is provided by [CloudFx](#). See the [WorkerRole.cs sample](#).)

It is also possible to move all of the code from the two worker roles into the web role so that everything runs in the web role. However, performing background tasks in ASP.NET is not supported or considered robust, and this architecture would complicate scalability. For more information see [The Dangers of Implementing Recurring Background Tasks In ASP.NET](#). See also [How to Combine a Worker and Web Role in Azure](#) and [Combining Multiple Azure Worker Roles into an Azure Web Role](#). If you wanted to go in this direction, a better solution would be to [run in an Azure web site and use the WebJobs feature for back-end tasks](#).

Another architecture alternative that would reduce cost is to use the [Autoscaling Application Block](#) to automatically deploy worker roles only during scheduled periods, and delete them when work is completed. For more information on autoscaling, see [the last tutorial in this series](#).

Azure in the future might provide a notification mechanism for scheduled reboots, which would allow you to only spin up an extra web role instance for the reboot time window. You wouldn't qualify for the 99.95 SLA, but you could reduce your costs by almost half and ensure your web application remains available during the reboot interval.

Authentication and authorization

In a production application you would implement an authentication and authorization mechanism like [ASP.NET Identity](#) for the ASP.NET MVC web front-end, including the ASP.NET Web API service method. There are also other options, such as using a shared secret, for securing the Web API service method. Authentication and authorization functionality has been omitted from the sample application to keep it simple to set up and deploy.

Next steps

In the **next tutorial**, you'll download the sample project, configure your development environment, configure the project for your environment, and test the project locally and in the cloud. In the tutorials 3 through 5 you'll see how to build the project from scratch.

Configuring and Deploying the Azure Email Service application - 2 of 5

This is the second tutorial in a series of five that show how to build and deploy the Azure Email Service sample application. For information about the application and the tutorial series, see the first tutorial in the series.

In this tutorial you'll learn:

How to set up your computer for Azure development by installing the Azure SDK.

How to configure and test the Azure Email Service application on your local machine.

How to publish the application to Azure.

How to view and edit Azure tables, queues, and blobs by using Visual Studio Server Explorer.

How to configure tracing and view trace data.

How to scale the application by increasing the number of worker role instances.

Set up the development environment

To start, set up your development environment by installing the [Azure SDK for Visual Studio 2013](#).

If you don't have Visual Studio installed, Visual Studio Express for Web will be installed along with the SDK.

NOTE:

Depending on how many of the SDK dependencies you already have on your machine, installing the SDK could take a long time, from several minutes to a half hour or more.

Download and runDownload and run the completed solution

1. Download and unzip the [completed solution](#).
2. Start Visual Studio.
3. From the File menu choose Open Project, navigate to where you downloaded the solution, and then open the solution file.
4. Press CTRL+SHIFT+B to build the solution.

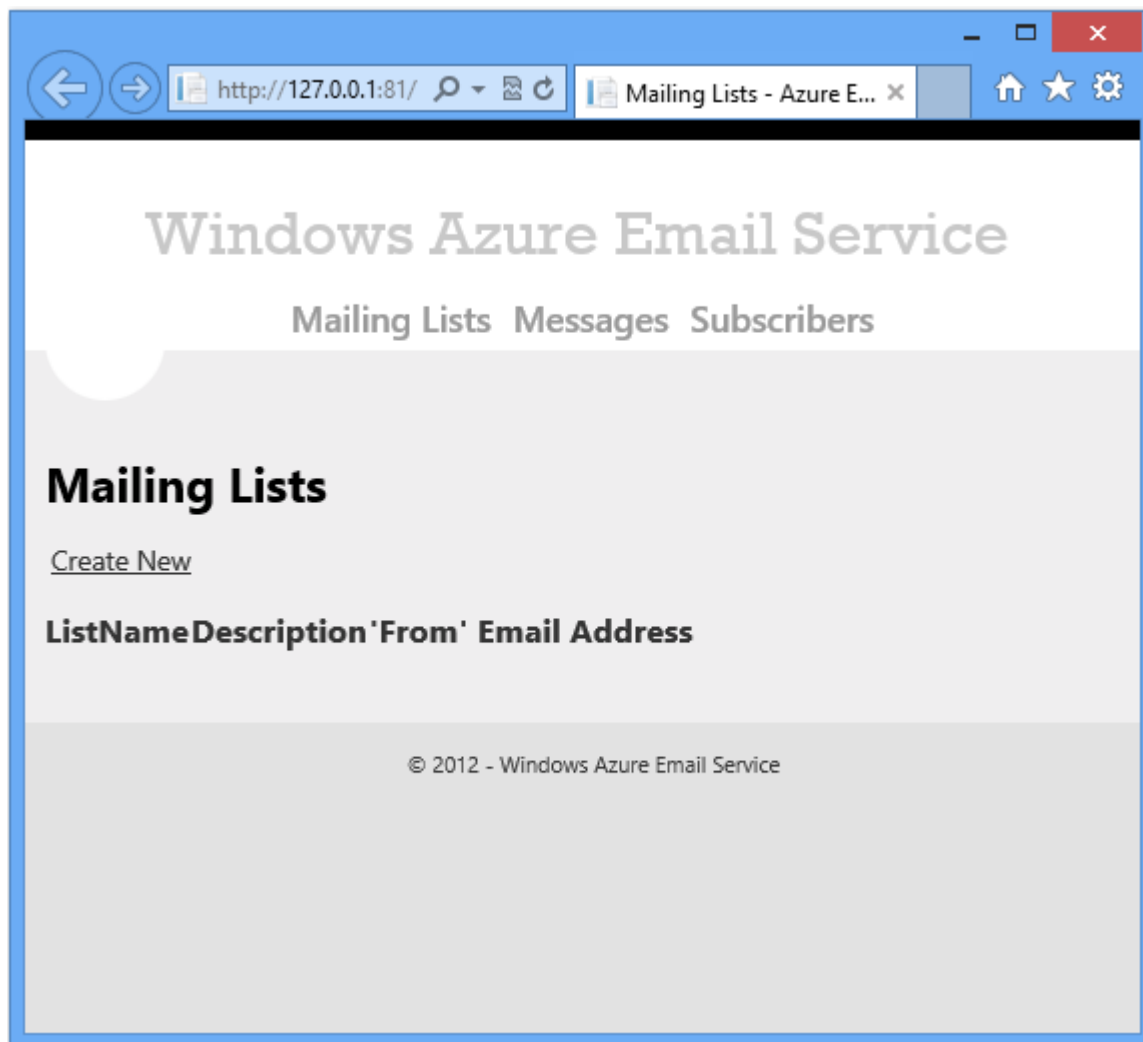
By default, Visual Studio automatically restores the NuGet package content, which was not included in the .zip file. If the packages don't restore, install them manually by going to the Manage NuGet Packages for Solution dialog and clicking the Restore button at the top right.

5. In Solution Explorer, make sure that AzureEmailService is selected as the startup project.
6. Press CTRL+F5 to run the application.

The application home page appears in your browser.

7. Click Mailing Lists in the menu bar.

(Screen shots show web page style from Visual Studio 2012 project templates but the content is the same for Visual Studio 2013.)



8. Click Create New.
9. Enter some test data, and then click Create.

The screenshot shows a web browser window with the URL `http://127.0.0.1:81/Mai`. The page title is "Create Mailing List - Azu...". The main heading is "Azure Email Service" with sub-links "Mailing Lists", "Messages", and "Subscribers". The form is titled "Create Mailing List" and contains three input fields: "ListName" with the value "wingtip1", "Description" with the value "Wingtip Toys sale notifications", and "'From' Email Address" with the value "donotreply@wingtip.com". A "Create" button is located below the fields, and a "Back to List" link is at the bottom left. The footer text is "© 2012 - AzureMail".

← → `http://127.0.0.1:81/Mai` Create Mailing List - Azu... x

Azure Email Service

Mailing Lists Messages Subscribers

Create Mailing List

ListName

Description

'From' Email Address

 x

10. Create a couple more mailing list entries.

Windows Azure Email Service

[Mailing Lists](#) [Messages](#) [Subscribers](#)

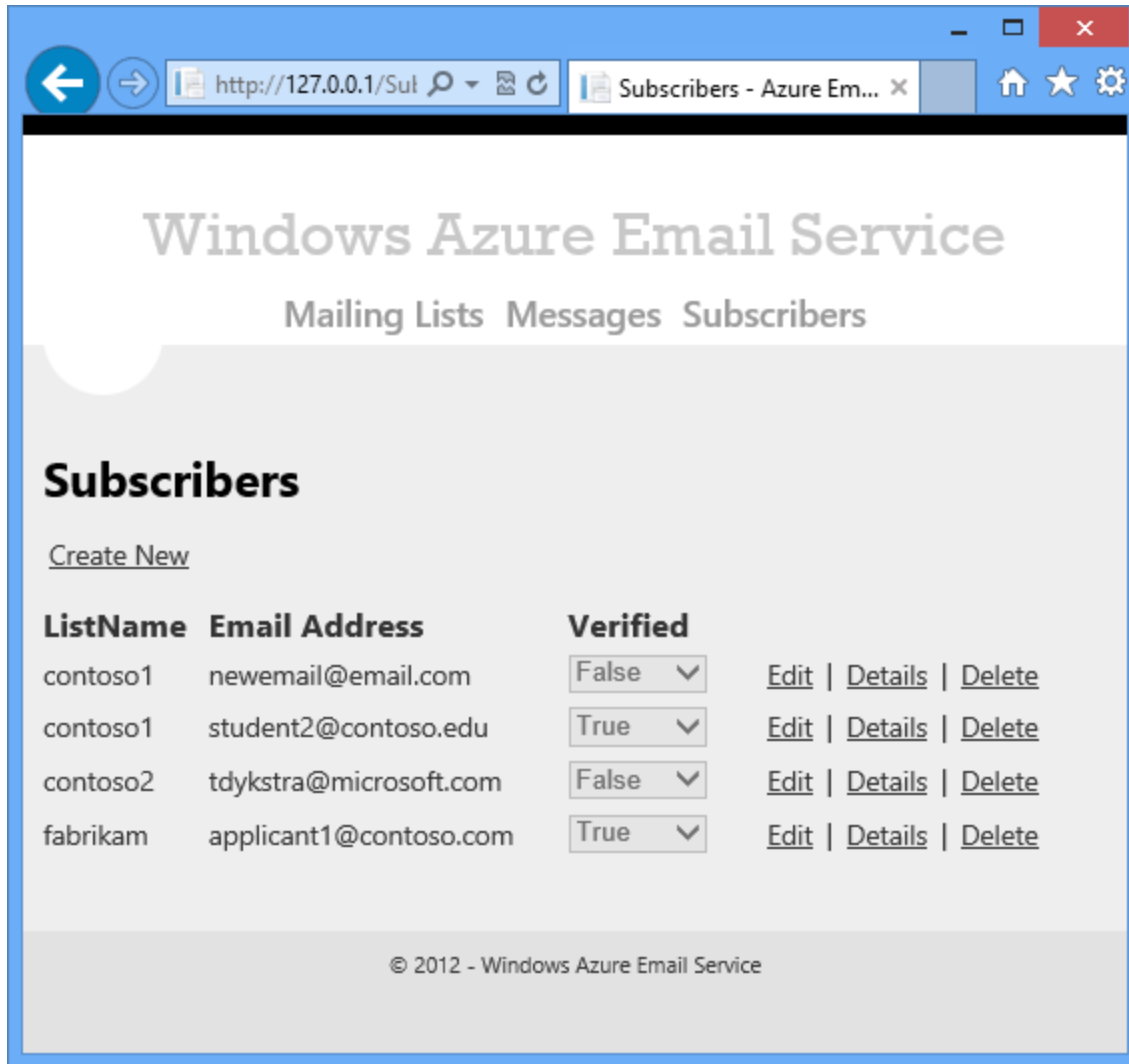
Mailing Lists

[Create New](#)

ListName	Description	'From' Email Address	
contoso1	Contoso University History Department announcements	donotreply@contoso.com	Edit Delete
contoso2	Contoso University Art Department announcements	donotreply@contoso.com	Edit Delete
fabrikam	Fabrikam Engineering job postings	donotreply@fabrikam.com	Edit Delete
wingtip1	Wingtip Toys sale notifications	donotreply@wingtip.com	Edit Delete

© 2012 - Windows Azure Email Service

11. Click Subscribers, and then add some subscribers. Set Verified to `true`.



12. Prepare to add messages by creating a `.txt` file that contains the body of an email that you want to send. Then create an `.htm` file that contains the same text but with some HTML (for example, make one of the words in the message bold or italicized). You'll use these files in the next step.
13. Click Messages, and then add some messages. Select the files that you created in the previous step. Don't change the scheduled date which defaults to one week in the future. The application can't send messages until you configure SendGrid.

←

→

http://127.0.0.1/Me

🔍

🔄

Create Message - Azure Em...

×

🏠

★

⚙️

Windows Azure Email Service

Mailing ListsMessagesSubscribers

Create Message

List Name

Contoso University History Department announcements ▾

Subject Line

Guest lecture by Nino Olivetto

HTML Path:

C:\rm\olivettolecture.html

Browse...

Text Path:

C:\rm\olivettolecture.txt

Browse...

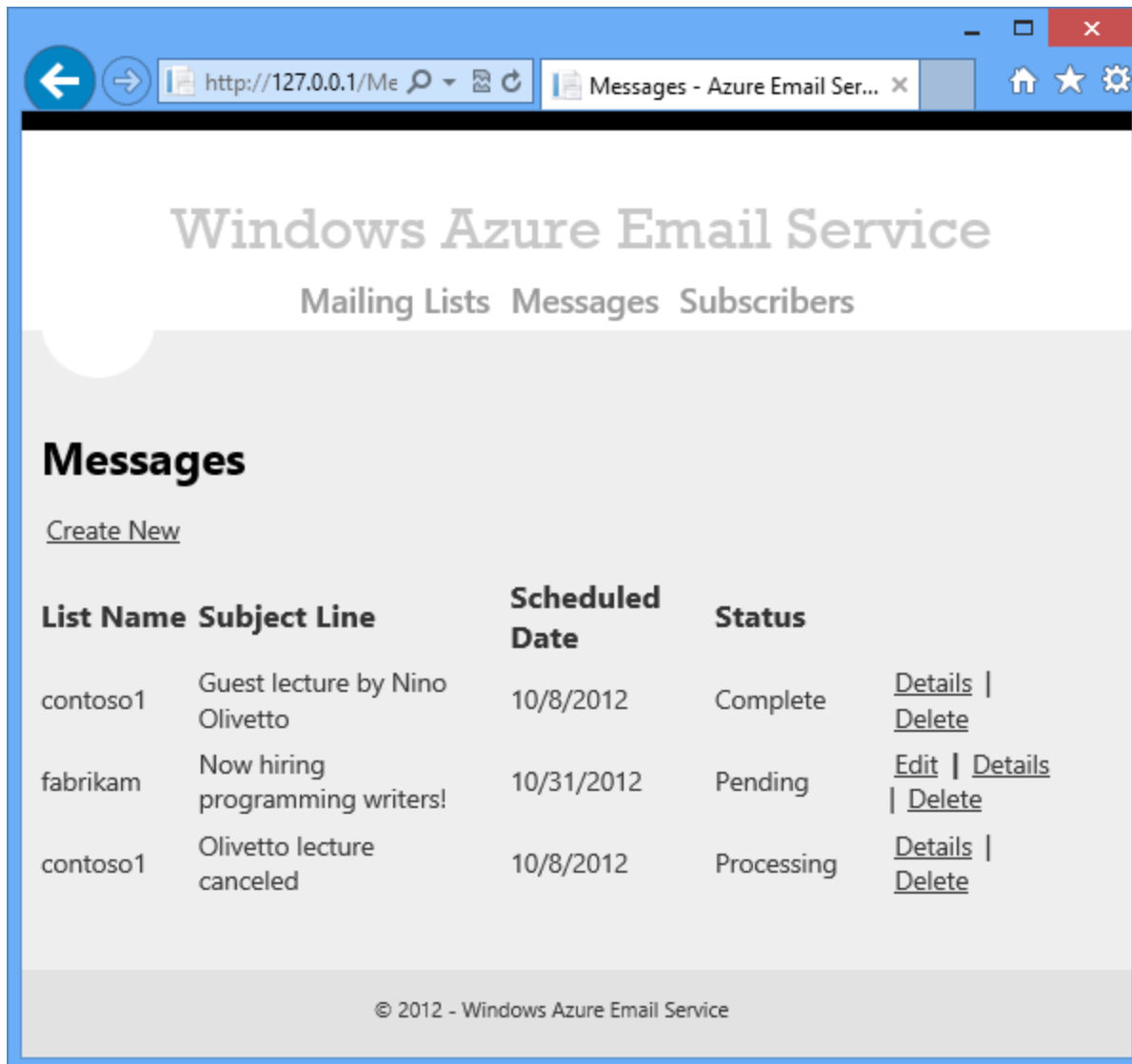
Scheduled Date

10/8/2012

Create

[Back to List](#)

© 2012 - Windows Azure Email Service



The data that you have been entering and viewing is being managed by the Azure storage emulator. The storage emulator uses a SQL Server Express LocalDB database to emulate the way Azure Storage works in the cloud. The application is using the storage emulator because that is what the project was configured to use when you downloaded it. This setting is stored in *.cscfg* files in the AzureEmailService project. The *ServiceConfiguration.Local.cscfg* file determines what is used when you run the application locally in Visual Studio, and the *ServiceConfiguration.Cloud.cscfg* file determines what is used when you deploy the application to the cloud. Later you'll see how to configure the application to use an Azure Storage account.

14. Close the browser.

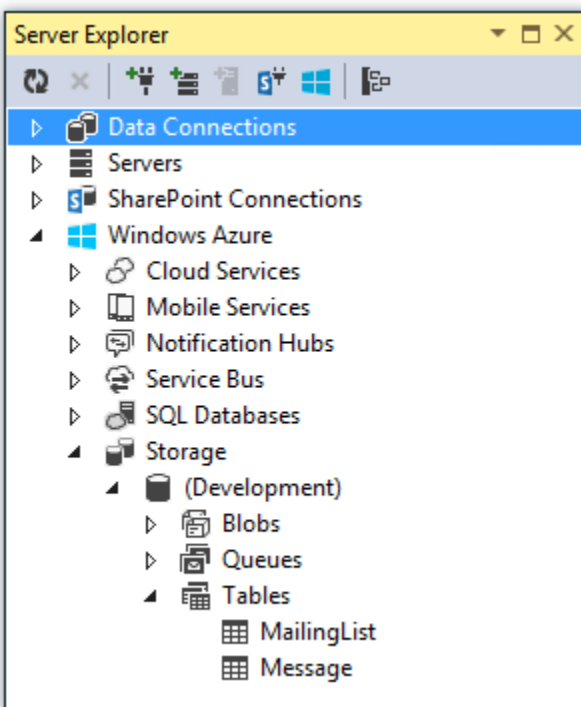
View development storage in Visual Studio

The Azure Storage browser in Server Explorer provides a convenient way to work directly with Azure Storage resources.

1. From the View menu in Visual Studio, choose Server Explorer.
2. Expand the Azure node, then expand the Storage node, and then expand the (Development) node underneath the Azure Storage node.

If you haven't already logged in to Azure in Visual Studio, you'll be prompted for your Azure credentials. Enter credentials for the account that has the subscription you want to run the cloud service in.

3. Expand Tables to see the tables that you created in the previous steps.



4. Double click the MailingList table.

AzureEmailService - MailingList [Table]

MailingList [Table] → ×

Enter a WCF Data Services filter to limit the entities returned

	PartitionKey	RowKey	Timestamp	Description	FromEmailAddress	Verified
▶	contoso1	MailingList	11/2/2012...	Contoso U...	donotreply@cont...	
	contoso1	student1@contoso.edu	11/2/2012...			False
	wingtip1	MailingList	11/2/2012...	Wingtip T...	donotreply@wing...	
	wingtip1	student2@contoso.edu	11/2/2012...			False

1 of 4

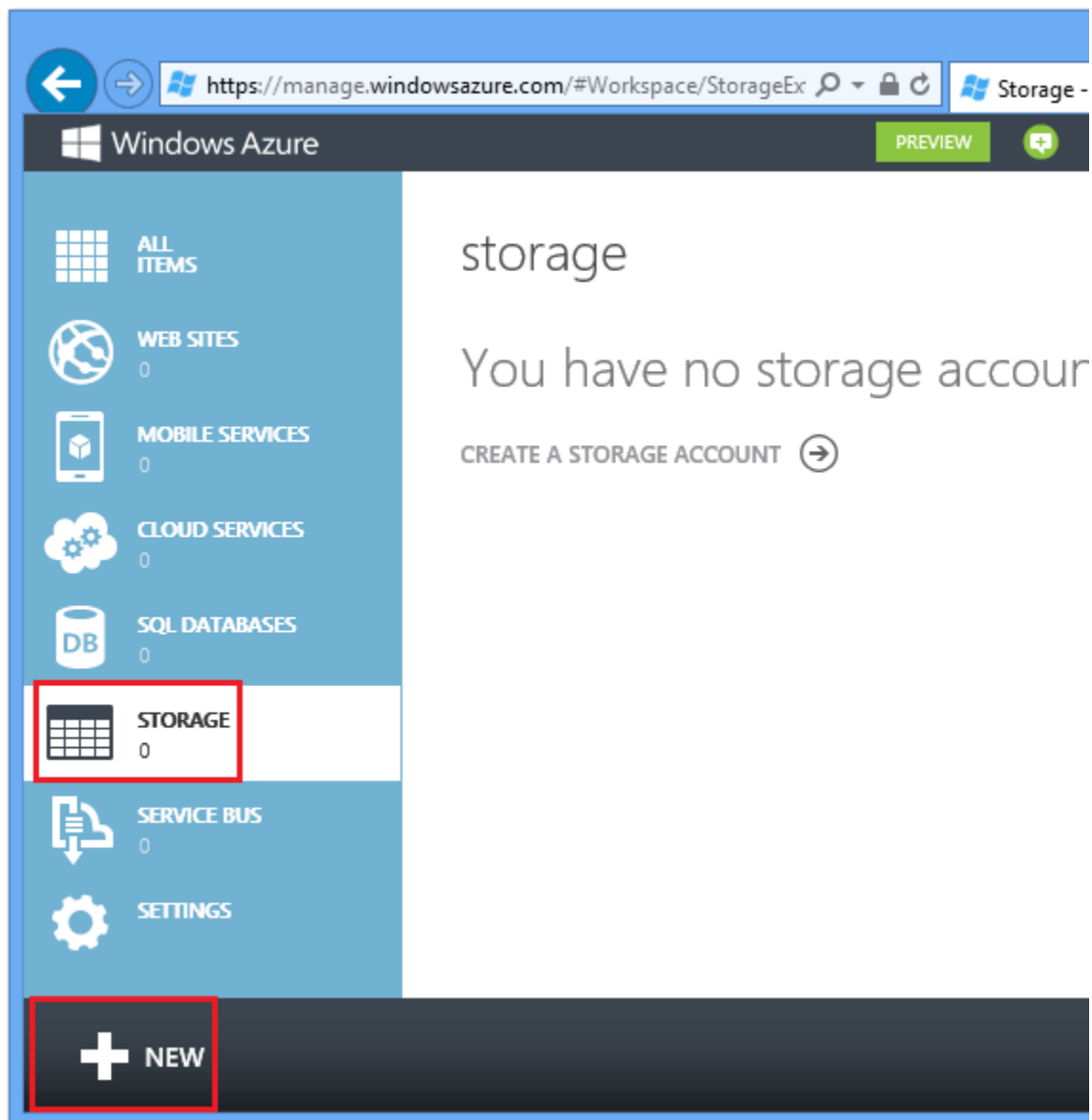
Notice how the window shows the different schemas in the table. `MailingList` entities have `Description` and `FromEmailAddress` property, and `Subscriber` entities have the `Verified` property (plus `SubscriberGUID` which isn't shown because the image isn't wide enough). The table has columns for all of the properties, and if a given table row is for an entity that doesn't have a given property, that cell is blank.

Another tool you can use to work with Azure Storage resources is [Azure Storage Explorer](#).

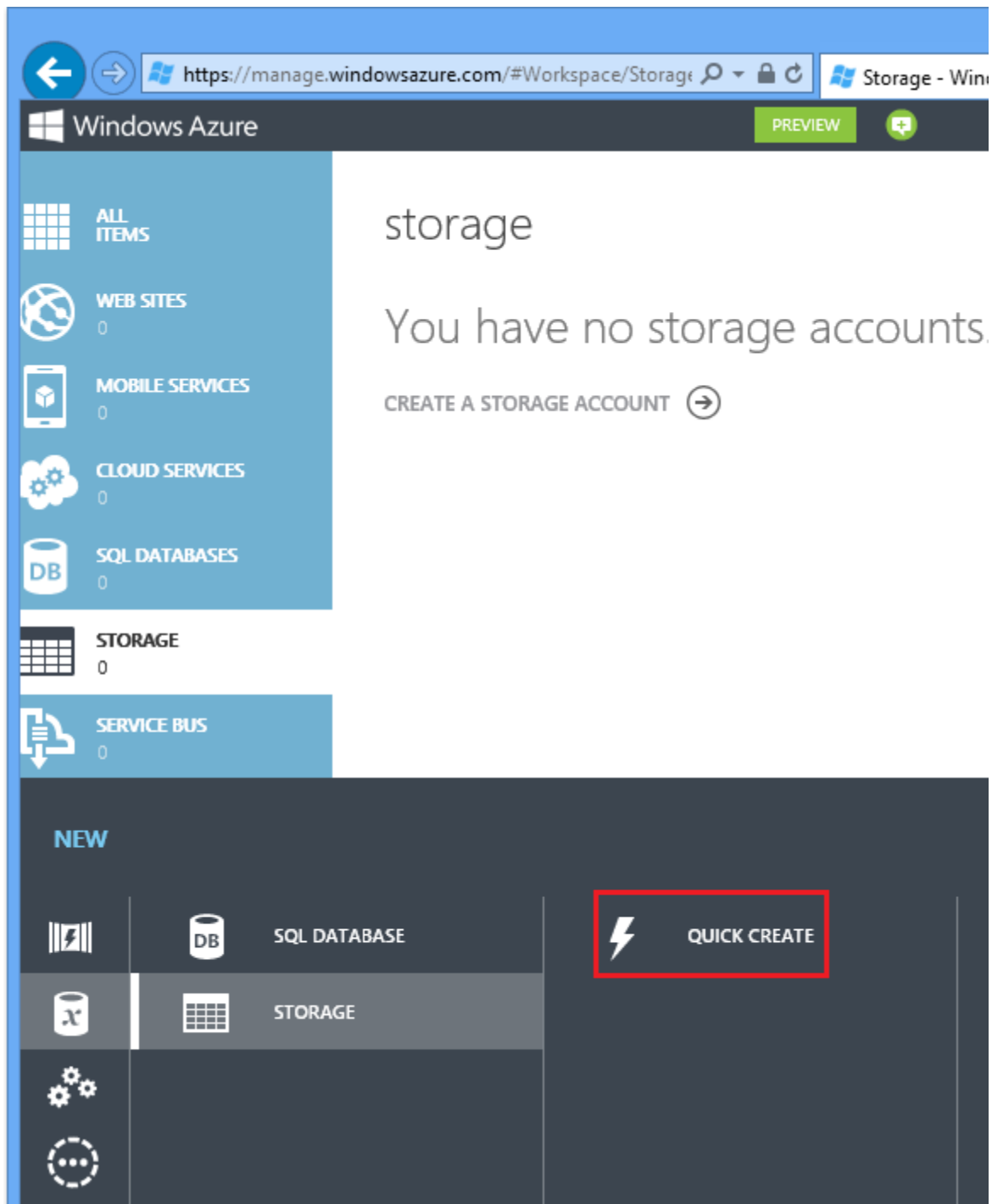
Create an Azure Storage account

When you run the sample application in Visual Studio, you can access tables, queues, and blobs in the Azure storage emulator or in an Azure Storage account in the cloud. In this section of the tutorial you create the Azure Storage account that you'll configure Visual Studio to use later in the tutorial.

1. In your browser, open the [Azure Management Portal](#).
2. In the [Azure Management Portal](#), click Storage, then click New.



1. Click Quick Create.



1. In the URL input box, enter a URL prefix.

This prefix plus the text you see under the box will be the unique URL to your storage account. If the prefix you enter has already been used by someone else, you'll see "The storage name is already in use" above the text box and you'll have to choose a different prefix.

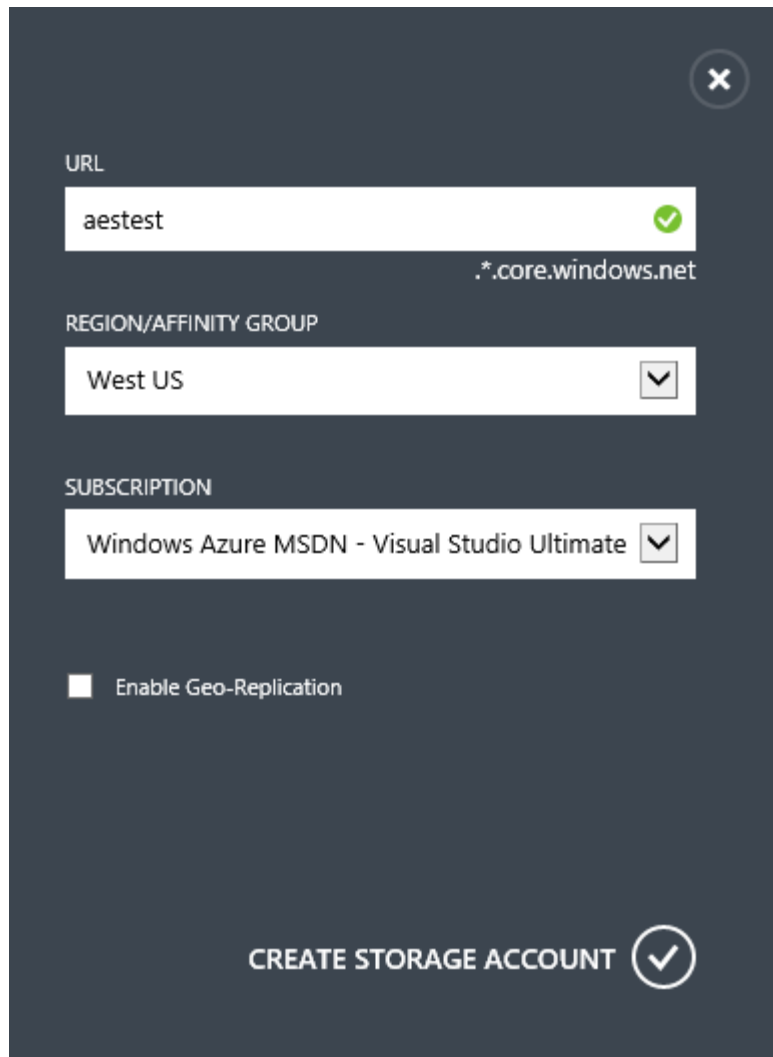
2. Set the region to the area where you want to deploy the application.

3. Set the Replication drop-down box to Locally redundant.

When geo-replication is enabled for a storage account, the stored content is replicated to a secondary location to enable failover to that location in case of a major disaster in the primary location. Geo-replication can incur additional costs. For test and development accounts, you generally don't want to pay for geo-replication. For more information, see [How To Manage Storage Accounts](#).

4. Click Create Storage Account.

In the image, a storage account is created with the URL `aetest3.core.windows.net`.



The image shows a dark-themed dialog box for creating a storage account. It has a close button (X) in the top right corner. The form contains the following fields:

- URL:** A text input field containing 'aetest' with a green checkmark icon to its right. Below the input field, the text '.*.core.windows.net' is displayed.
- REGION/AFFINITY GROUP:** A dropdown menu showing 'West US' with a downward arrow icon.
- SUBSCRIPTION:** A dropdown menu showing 'Windows Azure MSDN - Visual Studio Ultimate' with a downward arrow icon.
- Enable Geo-Replication:** A checkbox that is currently unchecked, followed by the text 'Enable Geo-Replication'.
- CREATE STORAGE ACCOUNT:** A large button with a white checkmark icon inside a circle.

This step can take several minutes to complete. While you are waiting, you can repeat these steps and create a production storage account. It's often convenient to have a test storage account to use for local development, another test storage account for testing in Azure, and a production storage account.

5. Click the test account that you created in the previous step, then click the Manage Access Keys icon.

← →

https://manage.windo... 🔍 🔒 ↻

Storage - Windows Azure


🏠 ★ ⚙️

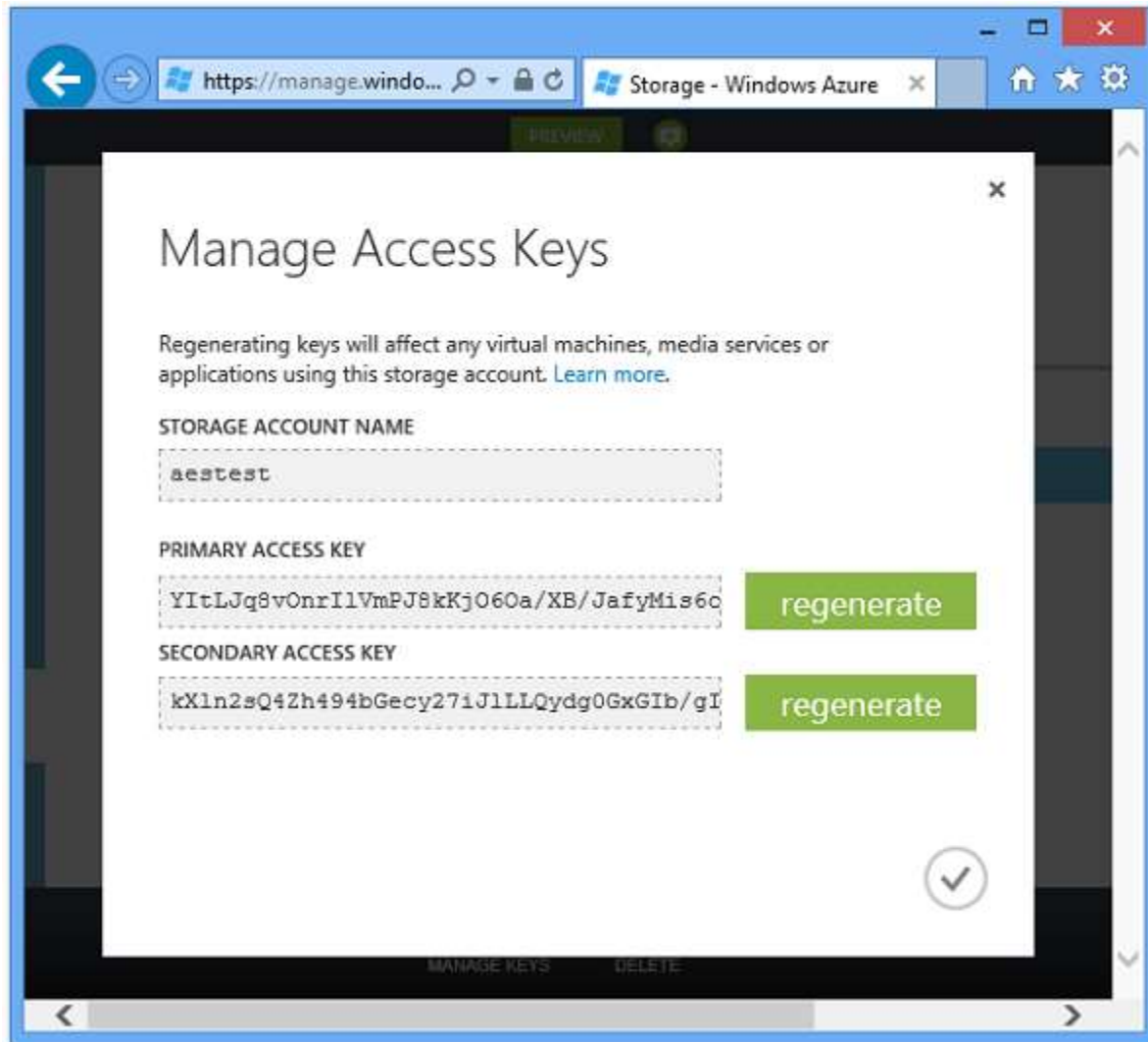
PREVIEW

storage

NAME	STATUS	LOCATION	SUBSCRIPTION
aesprod	✓ Online	West US	Windows Azure MSDN - Visual Studio Ultimate
aestest →	✓ Online	West US	Windows Azure MSDN - Visual Studio Ultimate

i
MANAGE KEYS


DELETE

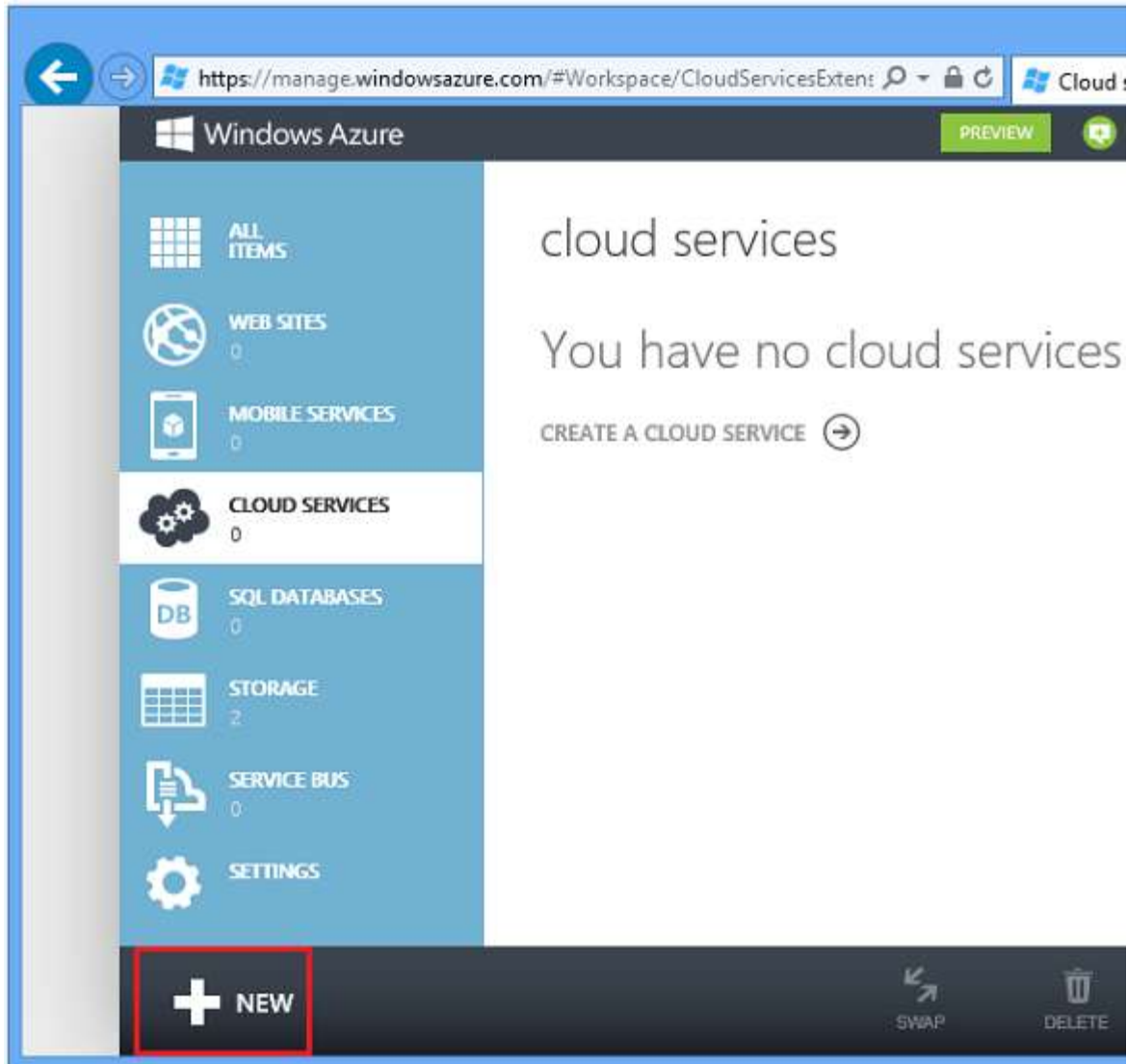


Visual Studio will automatically configure connection strings with one of these keys when you select the storage account. You can also update the connection strings manually.

There are two keys so that you can periodically change the key that you use without causing an interruption in service to a live application. You regenerate the key that you're not using, then you can change the connection string in your application to use the regenerated key. If there were only one key, the application would lose connectivity to the storage account when you regenerated the key. The keys that are shown in the image are no longer valid because they were regenerated after the image was captured.

Create Cloud Service

1. In the [Azure Management Portal](#), click Cloud Services then click the New icon.



2. Click Quick Create.
3. In the URL input box, enter a URL prefix.

Like the storage URL, this URL has to be unique, and you will get an error message if the prefix you choose is already in use by someone else.

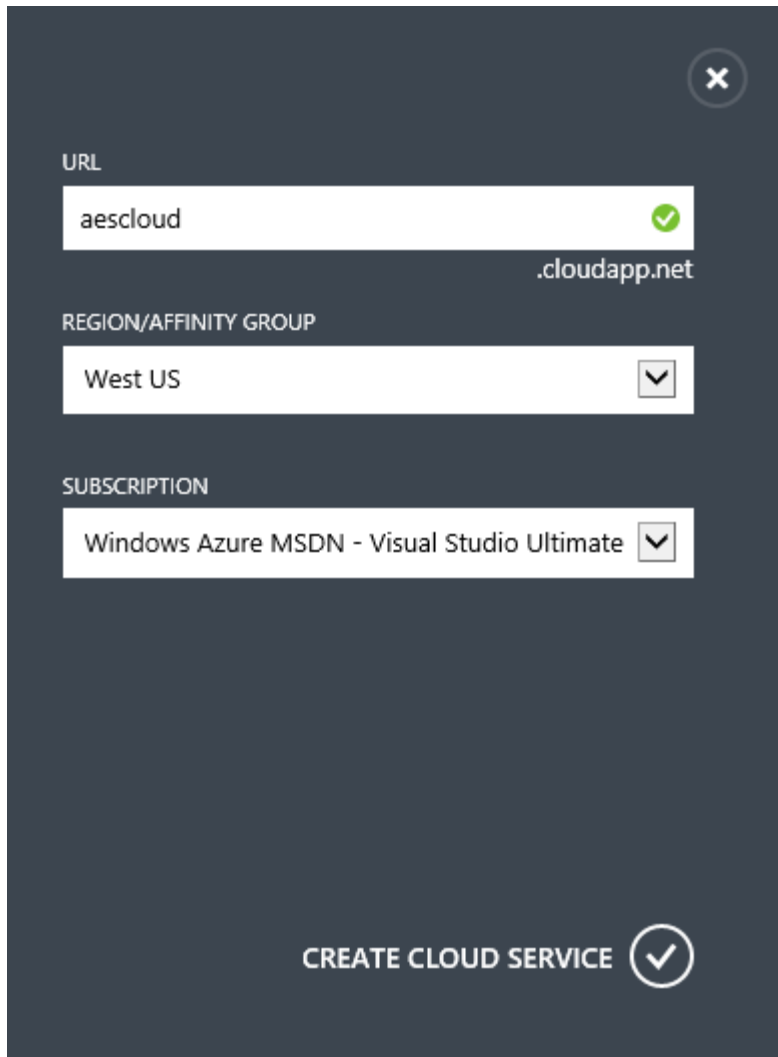
4. Set the region to the area where you want to deploy the application.

You should create the cloud service in the same region that you created the storage account. When the cloud service and storage account are in different datacenters (different regions), latency will increase and you will be charged for bandwidth outside the data center. Bandwidth within a data center is free.

Azure affinity groups provide a mechanism to minimize the distance between resources in a data center, which can reduce latency. This tutorial does not use affinity groups. For more information, see [How to Create an Affinity Group in Azure](#).

5. Click Create Cloud Service.

In the following image, a cloud service is created with the URL `aescloud.cloudapp.net`.



URL

aescloud ✓

.cloudapp.net

REGION/AFFINITY GROUP

West US ▼

SUBSCRIPTION

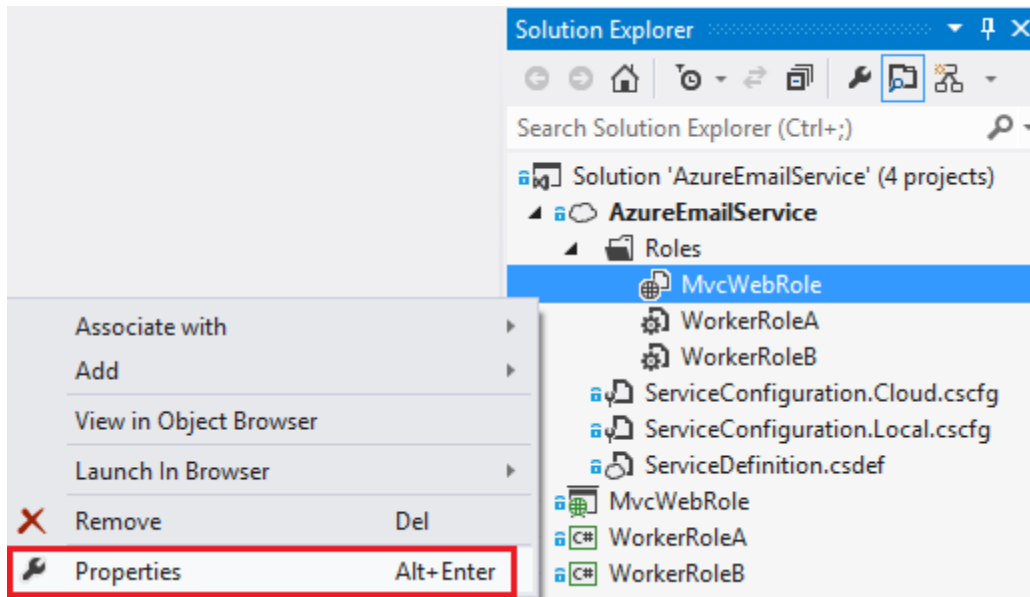
Windows Azure MSDN - Visual Studio Ultimate ▼

CREATE CLOUD SERVICE ✓

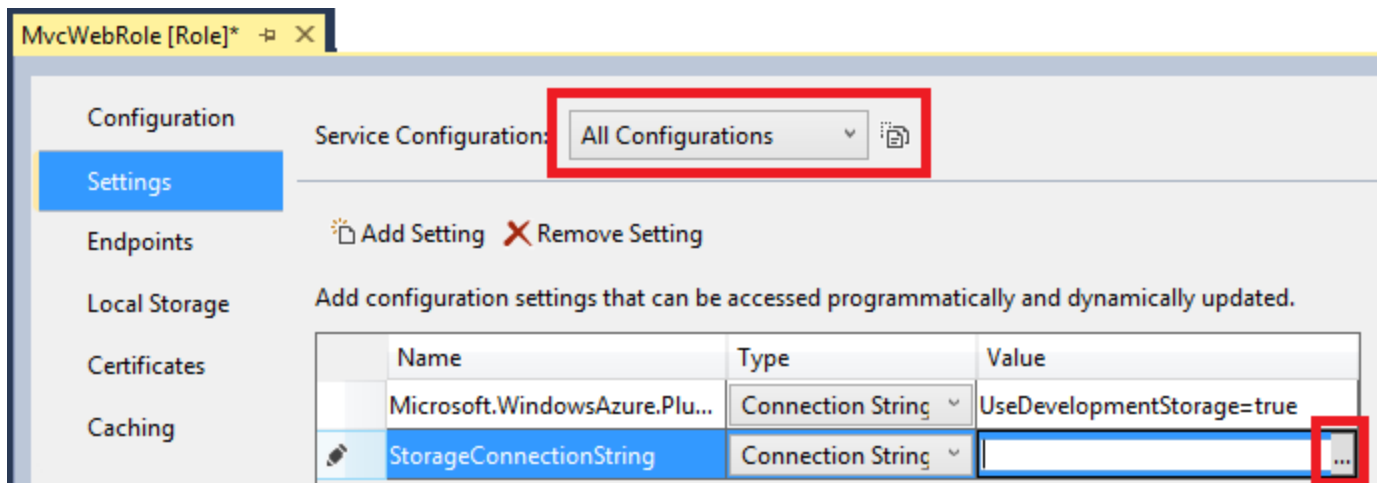
Use your storage accountConfigure the application to use your Azure Storage account

Next, you'll see how to configure the application so that it uses your Azure Storage account when it runs in Visual Studio, instead of development storage.

1. In Solution Explorer, right-click MvcWebRole under Roles in the AzureEmailService project, and click Properties.



2. In the MvcWebRole [Role] window, click the Settings tab.
3. In the Service Configuration drop down box, select Local.
4. Select the StorageConnectionString entry, and you'll see an ellipsis (...) button at the right end of the line. Click the ellipsis button to open the Storage Account Connection String dialog box.



5. In the Create Storage Connection String dialog, click Your subscription, and then choose your Subscription and your storage Account name.

Create Storage Connection String

Connect using:

- ☐ Windows Azure storage emulator
- ☒ **Your subscription**
- ☐ Manually entered credentials

Select a subscription and a storage account associated with it.

Signed in as: td15426@hotmail.com

Subscription:
Windows Azure MSDN - Visual Studio Ultimate (@hotmail.com)

Account name:
aetest3 (West US)

☒ Use HTTPS (Recommended)

Preview connection string:

DefaultEndpointsProtocol=https;AccountName=aetest3;AccountKey=LeYI9ntWlncP1/L4z4rsuJ+mudZthcV99gnthqd2jNG4MkteYohS7rE8APJtPxXqbky+2qbDO

[Online privacy statement](#) OK Cancel

- Follow the same procedure that you used for the `StorageConnectionString` connection string to set the `Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString` connection string.
- Follow the same procedure that you used for the two connection strings for the `MvcWebRole` role to set the connection strings for the `WorkerRoleA` role and the `workerRoleB` role.
- Open the `ServiceConfiguration.Local.cscfg` file that is located in the `AzureEmailService` project.

In the `Role` element for `MvcWebRole` you'll see a `ConfigurationSettings` element that has the settings that you updated by using the Visual Studio UI.

```
<Role name="MvcWebRole">
  <Instances count="1" />
  <ConfigurationSettings>
    <Setting
name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
```

```
value="DefaultEndpointsProtocol=https;AccountName=[name];AccountKey=[Key] "
/>
    <Setting name="StorageConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=aestest;AccountKey=[Key]
" />
    </ConfigurationSettings>
</Role>
```

In the `Role` elements for the two worker roles you'll see the same two connection strings.

You can edit these files directly instead of using the Visual Studio [Role] window. For more information on the configuration files, see [Configuring an Azure Project](#)

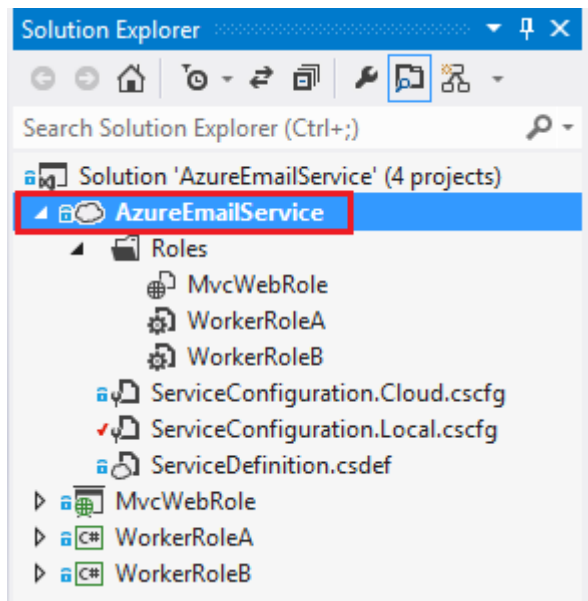
Test the application configured to use your storage account

1. Press CTRL+F5 to run the application. Enter some data by clicking the Mailing Lists, Subscribers, and Messages links as you did previously in this tutorial.
2. In Visual Studio, open Server Explorer.
3. Expand the Storage node under the Azure node, and then expand the node for the storage account that you configured the application to use.
4. Expand Tables, and double-click the `MailingList` table to see the data that you entered on the Mailing List and Subscriber pages of the application.

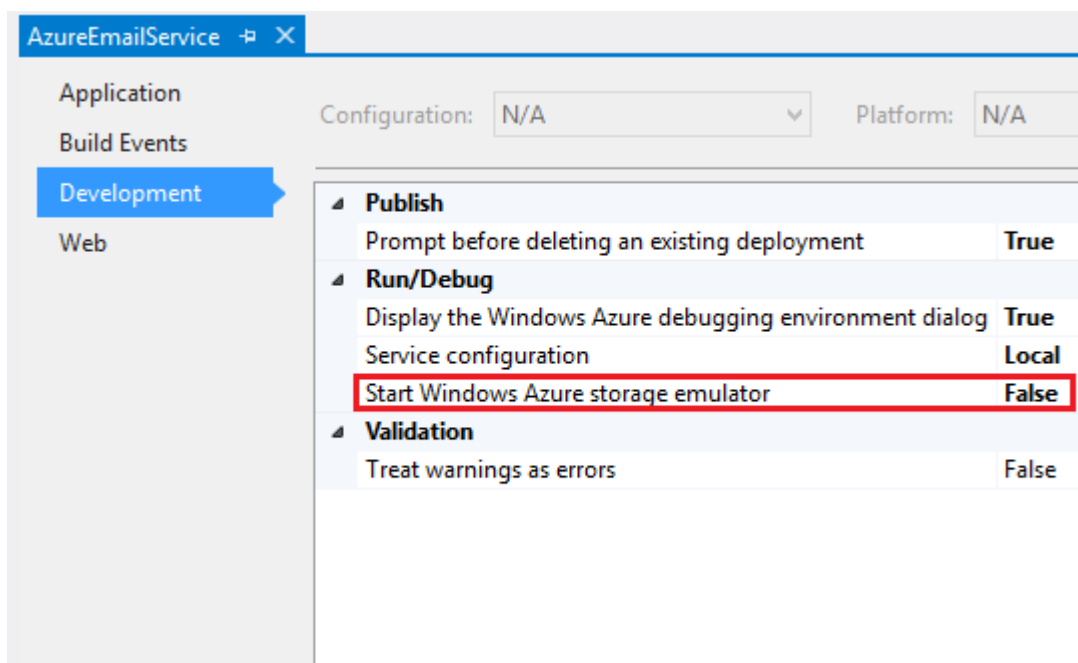
Optional steps to disable Azure Storage Emulator automatic startup

If you are not using the storage emulator, you can decrease project start-up time and use less local resources by disabling automatic startup for the Azure storage emulator.

1. In Solution Explorer, right click the `AzureEmailService` cloud project and select Properties.



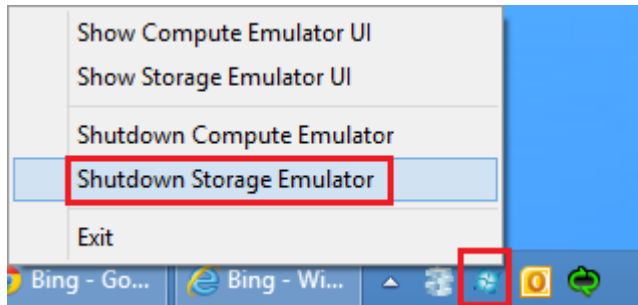
2. Select the Development tab.
3. Set Start Azure storage emulator to False.



Note: You should only set this to false if you are not using the storage emulator.

This window also provides a way to change the Service Configuration file that is used when you run the application locally from Local to Cloud (from *ServiceConfiguration.Local.cscfg* to *ServiceConfiguration.Cloud.cscfg*).

4. In the Windows system tray, right click on the compute emulator icon and click Shutdown Storage Emulator.



SendGridConfigure the application to use SendGrid

The sample application uses SendGrid to send emails. In order to send emails by using SendGrid, you have to set up a SendGrid account, and then you have to update a configuration file with your SendGrid credentials.

Note: If you don't want to use SendGrid, or can't use SendGrid, you can easily substitute your own email service. The code that uses SendGrid is isolated in two methods in worker role B. [Tutorial 5][tut5] explains what you have to change in order to implement a different method of sending emails. If you want to do that, you can skip this procedure and continue with this tutorial; everything else in the application will work (web pages, email scheduling, etc.) except for the actual sending of emails.

Create a SendGrid account

1. Follow the instructions in [How to Send Email Using SendGrid with Azure](#) to sign up for a free account.

Update SendGrid credentials in worker role properties

Earlier in the tutorial when you set the storage account credentials for the web role and the two worker roles, you may have noticed that worker role B had three settings that were not in the web role or worker role A. You can use that same UI now to configure those three settings (select Cloud in the Service Configuration drop-down list).

The following steps show an alternative method for setting the properties, by editing the configuration file.

1. Edit the *ServiceConfiguration.Cloud.cscfg* file in the `AzureEmailService` project and enter the SendGrid user name and password values that you obtained in the previous step into the `WorkerRoleB` element that has these settings. The following code shows the `WorkerRoleB` element.

```
<Role name="WorkerRoleB">
  <Instances count="1" />
  <ConfigurationSettings>
    <Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString" value=
    <Setting name="StorageConnectionString" value="DefaultEndpointsProtocol=https;Acc
    <Setting name="SendGridUserName" value="SendGrid User Name" />
    <Setting name="SendGridPassword" value="SendGrid Password" />
    <Setting name="AzureMailServiceURL" value="Azure Mail Service URL" />
  </ConfigurationSettings>
</Role>
```

2. There is also an AzureMailServiceURL setting. Set this value to the URL that you selected when you created your Azure Cloud Service, for example: "http://aescloud.cloudapp.net".

By updating the cloud configuration file, you are configuring settings that will be used when the application runs in the cloud. If you wanted the application to send emails while it runs locally, you would also have to update the *ServiceConfiguration.Local.cscfg* file.

Deploy to AzureDeploy the Application to Azure

To deploy the application you can create a package in Visual Studio and upload it by using the Azure Management Portal, or you can publish directly from Visual Studio. In this tutorial you'll use the publish method.

You'll publish the application to the staging environment first, and later you'll promote the staging deployment to production.

Implement IP restrictions

When you deploy to staging, the application will be publicly accessible to anyone who knows the URL. Therefore, your first step is to implement IP restrictions to ensure that no unauthorized persons can use it. In a production application you would implement an authentication and authorization mechanism like ASP.NET Identity, but these functions have been omitted from the sample application to keep it simple to set up, deploy, and test.

1. Open the *Web.Release.config* file that is located in the root folder of the MvcWebRole project, and replace the ipAddress attribute value 127.0.0.1 with your IP address. (To see the Web.Release.config file in Solution Explorer you have to expand the *Web.config* file.)

You can find your IP address by searching for "Find my IP" with [Bing](#) or another search engine.

When the application is published, the transformations specified in the *Web.release.config* file are applied, and the IP restriction elements are updated in the *web.config* file that is deployed to the cloud. You can view the transformed *web.config* file in the

AzureEmailService\MvcWebRole\obj\Release\TransformWebConfig\transformed folder after the package is created.

Configure the application to use your storage account when it runs in the cloud

Earlier in the tutorial when you set the storage account credentials for the web role and the two worker roles, you set the credentials to use when you run the application locally. Now you need to set the storage account credentials to use when you run the application in the cloud.

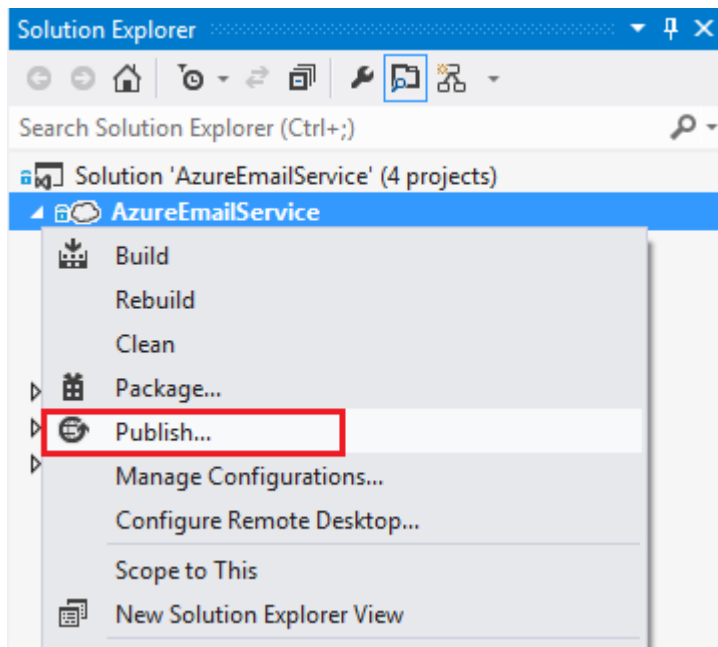
For this test run you'll use the same credentials for the cloud that you have been using for running locally. If you were deploying a production application, you would typically use a different account for production than you use for testing. Also a best practice for production would be to use a different account for the diagnostics connection string than the storage connection string, but for this test run you'll use the same account.

You can use the same UI to configure the connection strings (just make sure that you select Cloud in the Service Configuration drop-down list). As an alternative, you can edit the configuration file, as explained in the following steps.

1. Open the *ServiceConfiguration.Local.cscfg* file in the AzureEmailService project, and copy the *Setting* elements for *StorageConnectionString* and *Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString*.
2. Open the *ServiceConfiguration.Cloud.cscfg* file in the AzureEmailService project, and paste the copied elements into the *Configuration Settings* element for *MvcWebRole*, *WorkerRoleA*, and *WorkerRoleB*, replacing the *Setting* elements that are already there.
3. Verify that the web role and two worker role elements all define the same connection strings.

Publish the application

1. If it is not already open, launch Visual Studio and open the AzureEmailService solution.
2. Right-click the AzureEmailService cloud project and select Publish.



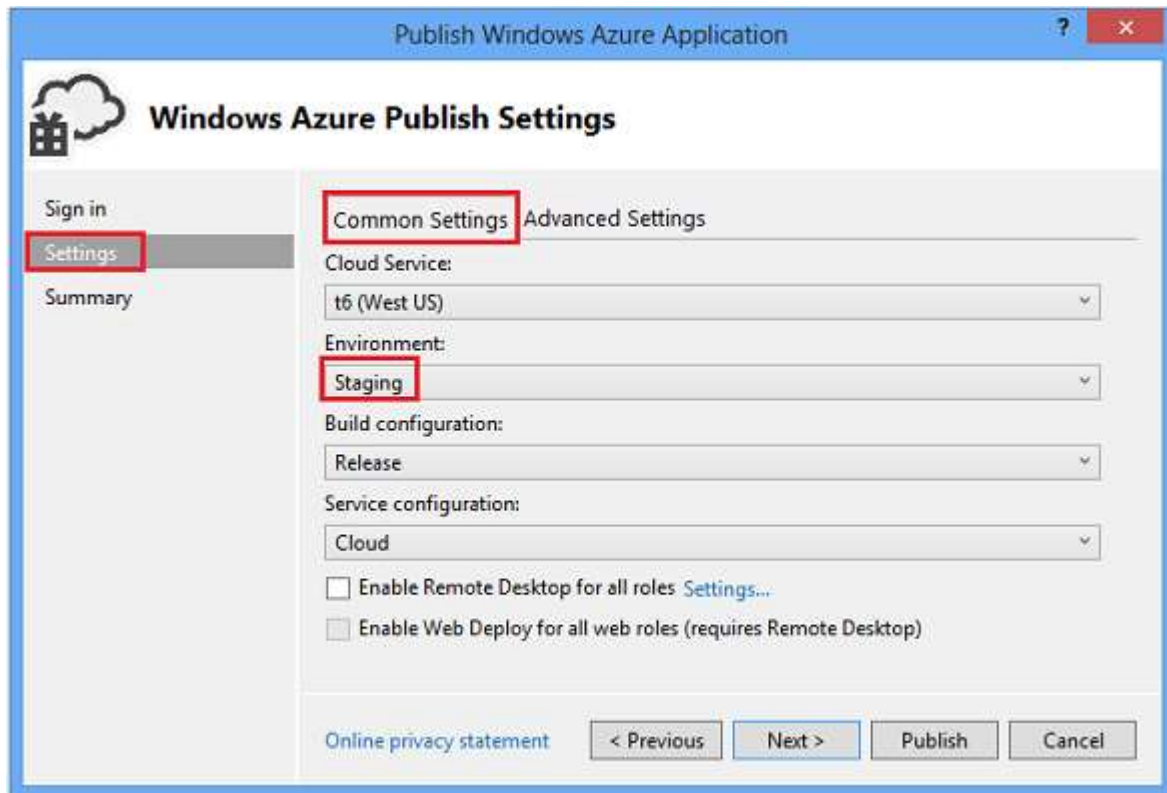
The Publish Azure Application dialog appears.



3. If you used the automatic method for importing storage account credentials earlier, your Azure subscription is in the drop-down list and you can select it and then click Next. Otherwise, click Sign in to download

credentials and follow the instructions in [Configure the application for Azure Storage](#) to download and import your publish settings.

4. In the Common Settings tab, verify that your cloud service is selected in the Cloud Service drop-down list.
5. In the Environment drop-down list, change Production to Staging.



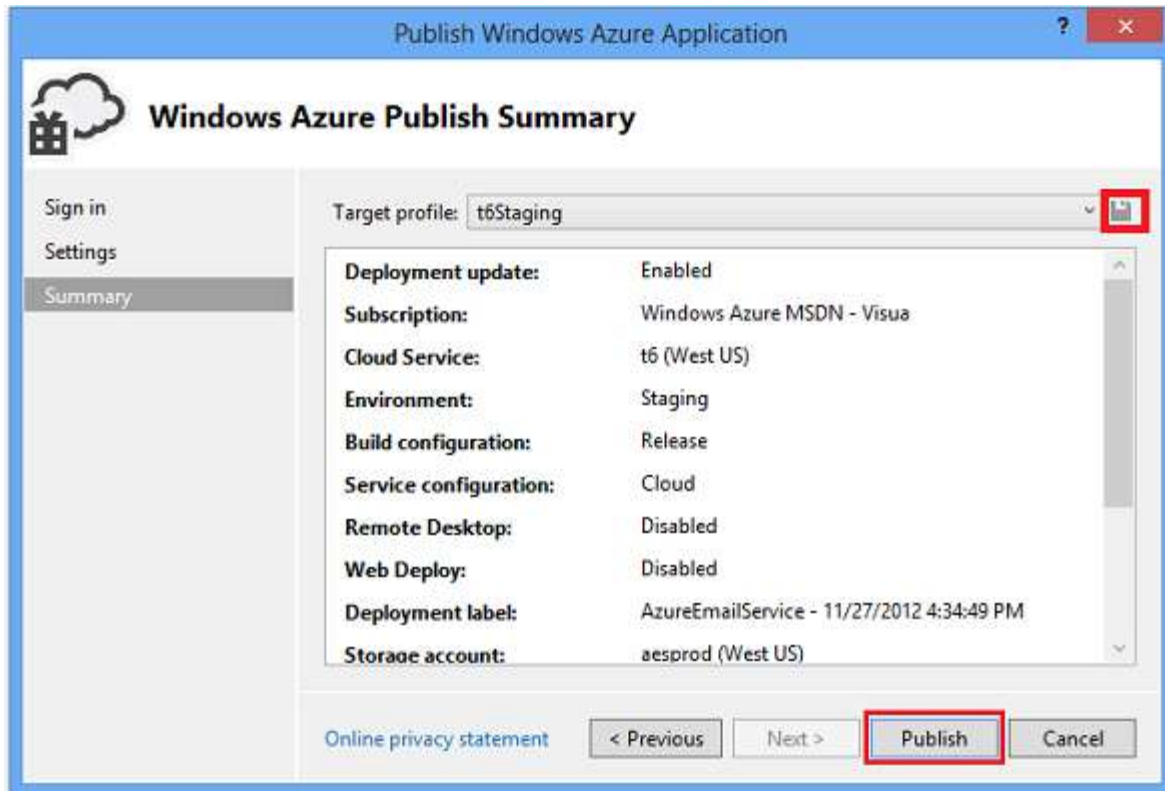
6. Keep the default Release setting for Build configuration and Cloud for Service configuration.

The default settings in the Advanced tab are fine for this tutorial. On the Advanced tab are a couple of settings that are useful for development and testing. For more information on the advanced tab, see [Publish Azure Application Wizard](#).

7. Click Next.
8. In the Summary step of the wizard, click the save icon (the diskette icon shown to the right of the Target profile drop-down list) to save the publish settings.

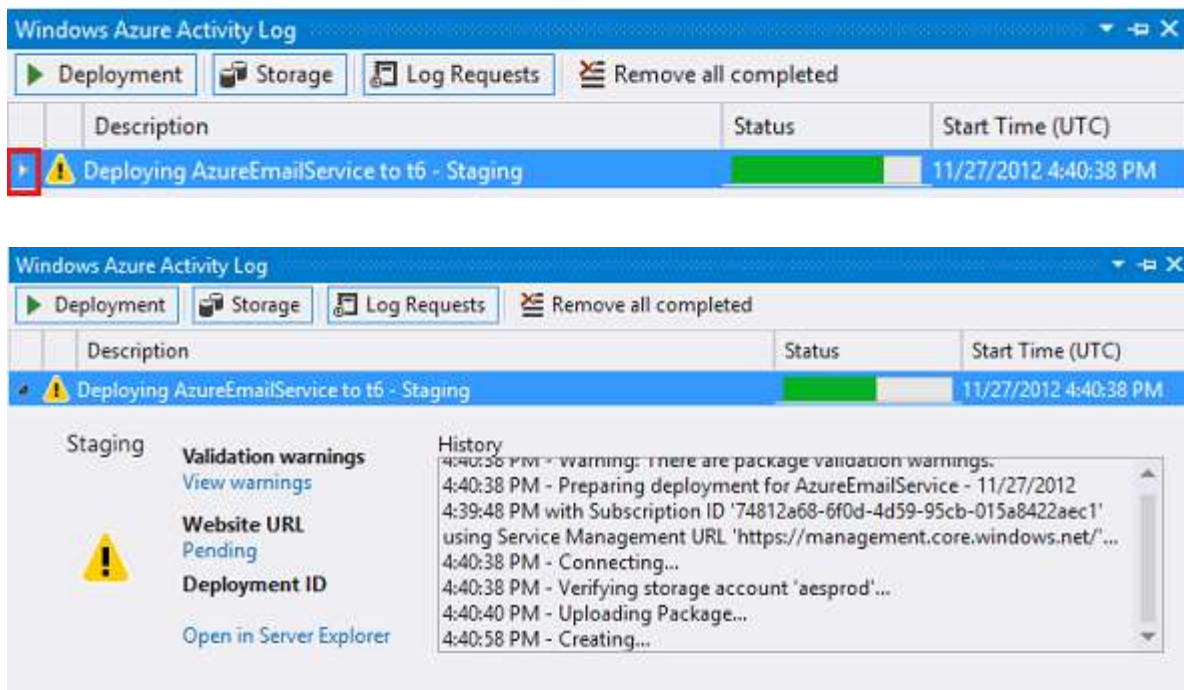
The next time you publish the application, the saved settings will be used and you won't need to go through the publish wizard again.

9. Review the settings, then click Publish.



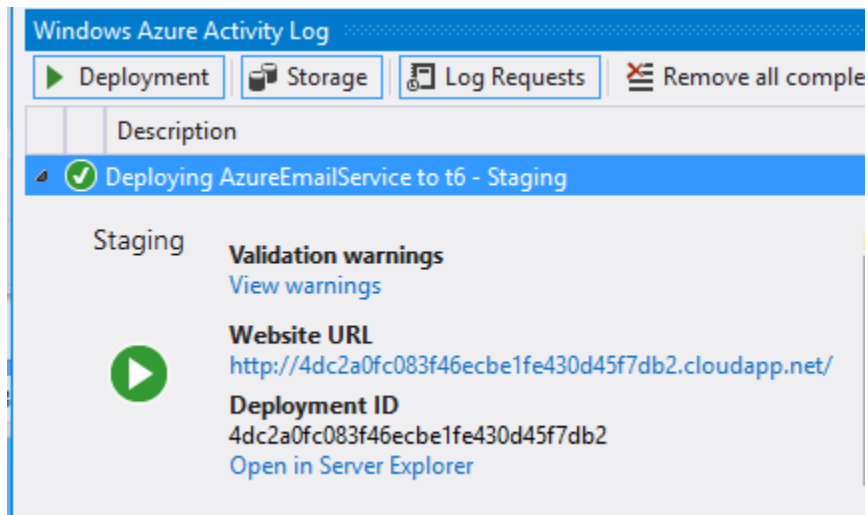
The Azure Activity Log window is opened in Visual Studio.

1. Click the right arrow icon to expand the deployment details.



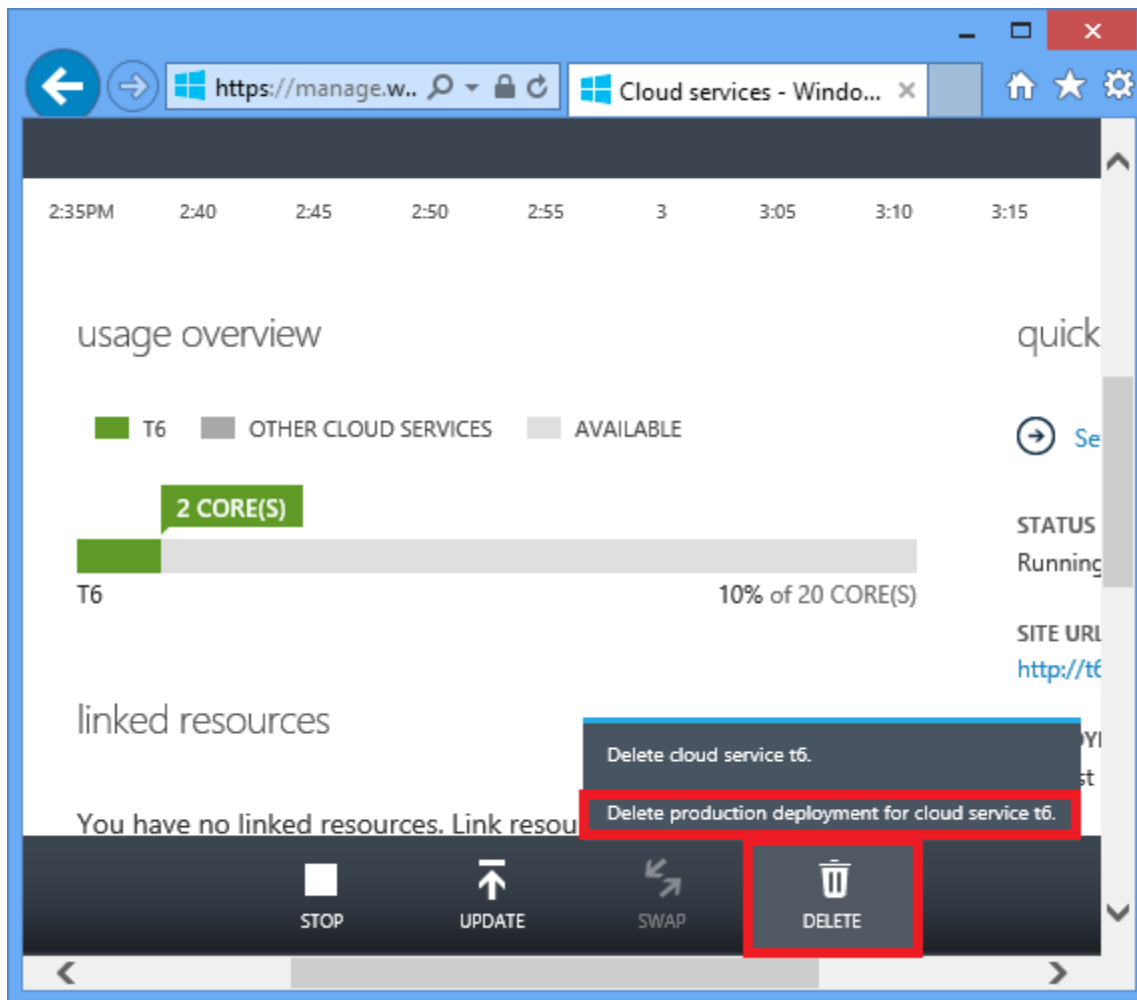
The deployment can take about 5 minutes or more to complete.

- When the deployment status is complete, click the Website URL to launch the application.



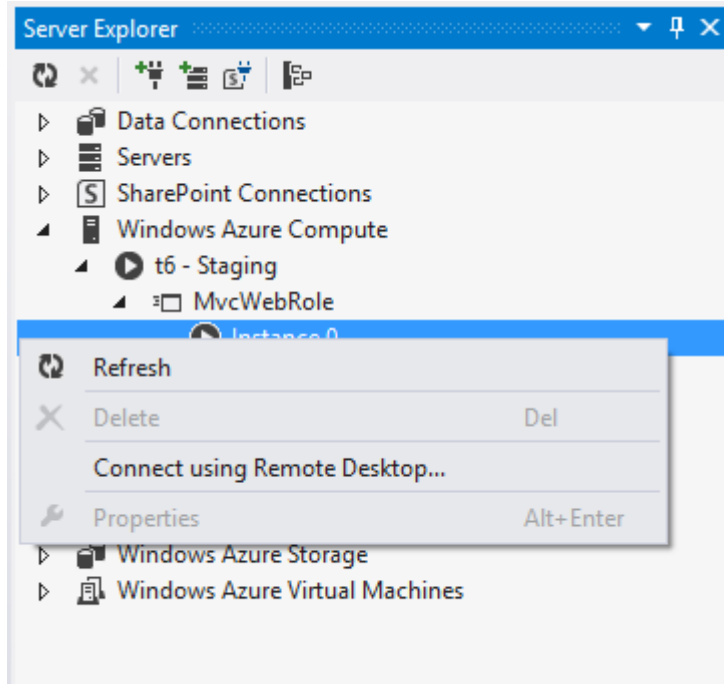
- Enter some data in the Mailing List, Subscriber, and Message web pages to test the application.

Note: Delete the application after you have finished testing it to avoid paying for resources that you aren't using. If you are using a [Azure free trial account](#), the three deployed roles will use up your monthly limit in a couple of weeks. To delete a deployment by using the Azure management portal, select the cloud service and click DELETE at the bottom of the page, and then select the production or staging deployment.



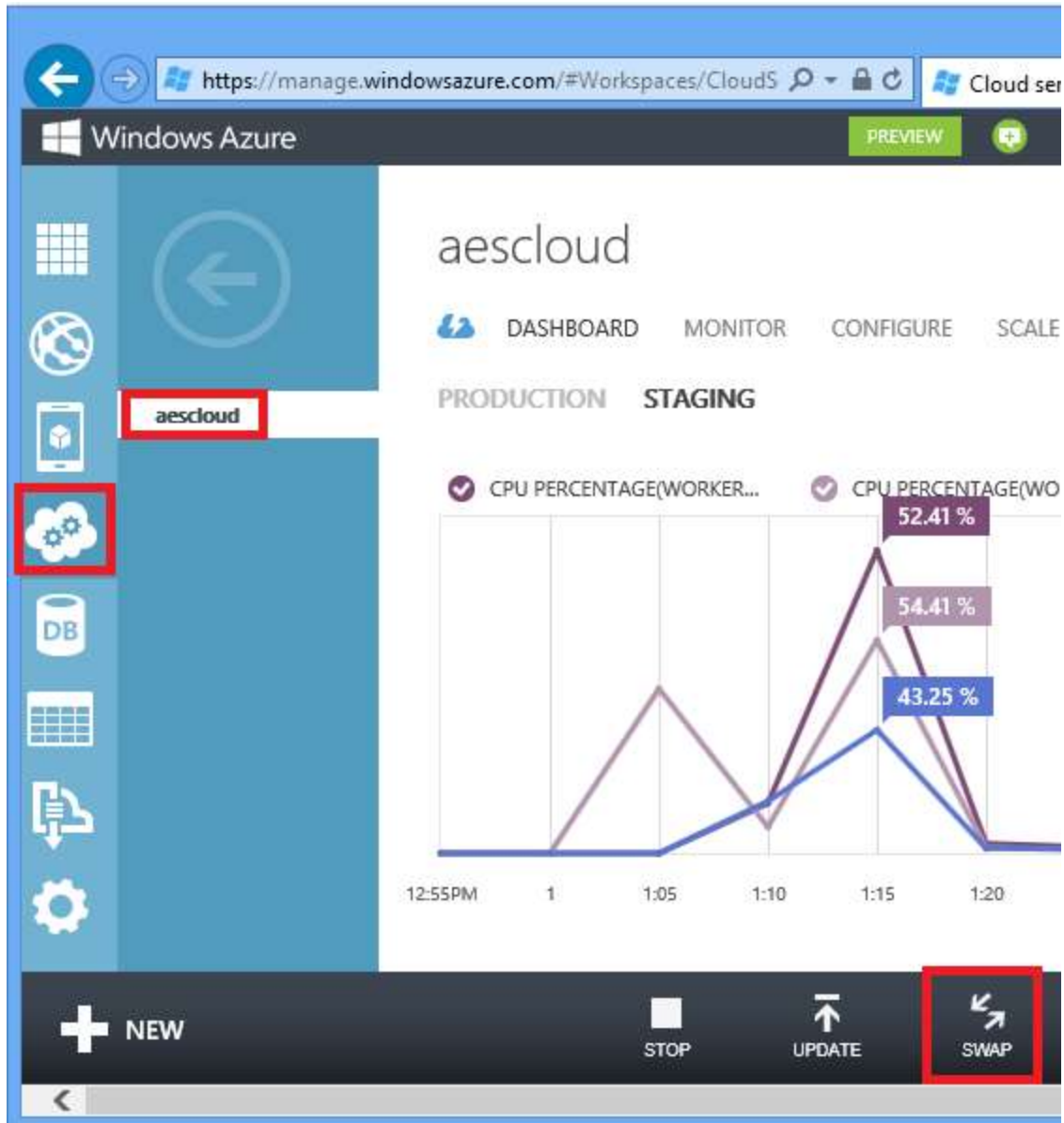
4. In the Azure Activity Log in Visual studio, select Open in Server Explorer.

Under Azure Compute in Server Explorer you can monitor the deployment. If you selected Enable Remote Desktop for all roles in the Publish Azure Application wizard, you can right click on a role instance and select Connect using Remote Desktop.



Promote the Application from Staging to Production

1. In the [Azure Management Portal](#), click the Cloud Services icon in the left pane, and then select your cloud service and click the Dashboard tab.
2. Click Swap.
3. Click Yes to complete the VIP (virtual IP) swap. This step can take several minutes to complete.



- When the swap has completed, scroll down the Dashboard tab for the Production deployment to the quick glance section on the lower right part of the page. Notice that the Site URL has changed from a GUID prefix to the name of your cloud service.

quick glance



STATUS
Running

SITE URL
<http://aescloud.cloudapp.net/>

DEPLOYMENT NAME
initial

PUBLIC VIRTUAL IP ADDRESS (VIP)
168.62.21.234

INPUT ENDPOINTS
MvcWebRole:168.62.21.234:80

5. Click the link under Site URL or copy and paste it to a browser to test the application in production.

If you haven't changed the storage account settings, the data you entered while testing the staged version of the application is shown when you run the application in the cloud.

Configure and View Tracing Data

Tracing is an invaluable tool for debugging a cloud application. In this section of the tutorial you'll see how to view tracing data.

1. Verify that the diagnostics connection string is configured to use your Azure Storage account and not development storage.

If you followed the instructions earlier in the tutorial, they will be the same. You can verify that they are the same either using the Visual Studio UI (the Settings tab in Properties for the roles), or by looking at the *ServiceConfiguration..cscfg** files.

Note: A best practice is to use a different storage account for tracing data than the storage account used for production data, but for simplicity in this tutorial you have been configuring the same account for tracing.

1. In Visual Studio, open *WorkerRoleA.cs* in the WorkerRoleA project, search for `ConfigureDiagnostics`, and examine the `ConfigureDiagnostics` method.

```
2. private void ConfigureDiagnostics()
```

```

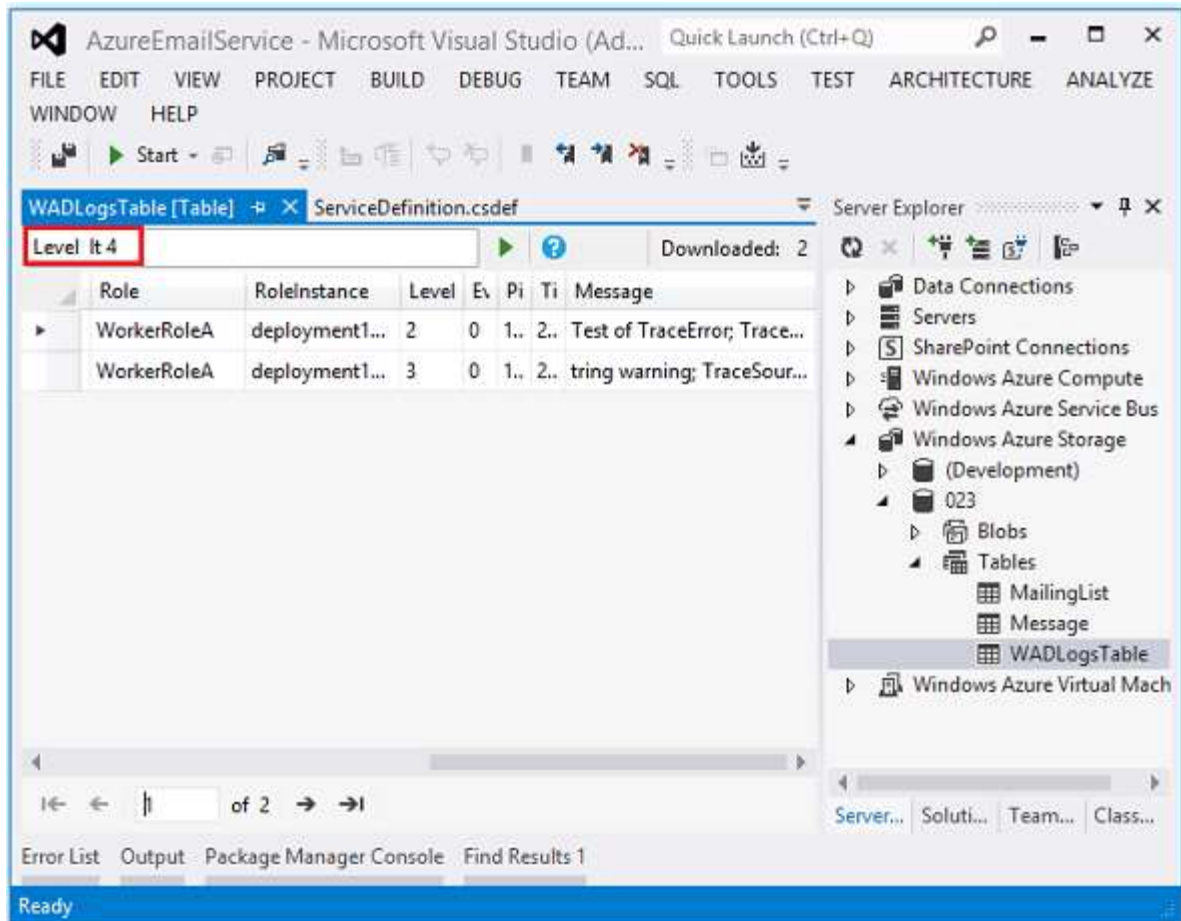
3. {
4.     DiagnosticMonitorConfiguration config =
        DiagnosticMonitor.GetDefaultInitialConfiguration();
5.     config.ConfigurationChangePollInterval = TimeSpan.FromMinutes(1d);
6.     config.Logs.BufferQuotaInMB = 500;
7.     config.Logs.ScheduledTransferLogLevelFilter = LogLevel.Verbose;
8.     config.Logs.ScheduledTransferPeriod = TimeSpan.FromMinutes(1d);
9.
10.         DiagnosticMonitor.Start(
11.             "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString",
12.             config);
    }

```

In this code, the `DiagnosticMonitor` is configured to store up to 500 MB of trace information (after 500 MB, the oldest data is overwritten) and to store all trace messages (`LogLevel.Verbose`). The `ScheduledTransferPeriod` transfers the trace data to storage every minute. You must set the `ScheduledTransferPeriod` to save trace data.

The `ConfigureDiagnostics` method in each of the worker and web roles configures the trace listener to record data when you call the Trace API. For more information, see [Using Trace in Windows Azure Cloud Applications](#)

13. In Server Explorer, double-click `WADLogsTable` (expand `Azure / Storage / yourstorageaccountname / Tables`) for the storage account that you added previously. You can enter a [WCF Data Services filter](#) to limit the entities displayed. In the following image, only warning and error messages are displayed.



Add another worker role instance to handle increased load

There are two approaches to scaling compute resources in Azure roles, by specifying the [virtual machine size](#) and/or by specifying the instance count of running virtual machines.

The virtual machine (VM) size is specified in the `vmSize` attribute of the `WebRole` or `WorkerRole` element in the `ServiceDefinition.csdef` file. The default setting is `Small` which provides you with one core and 1.75 GB of RAM. For applications that are multi-threaded and use lots of memory, disk, and bandwidth, you can increase the VM size for increased performance. For example, an `ExtraLarge` VM has 8 CPU cores and 14 GB of RAM. Increasing memory, cpu cores, disk, and bandwidth on a single machine is known as *scale up*. Good candidates for scale up include ASP.NET web applications that use [asynchronous methods](#). See [Virtual Machine Sizes](#) for a description of the resources provided by each VM size.

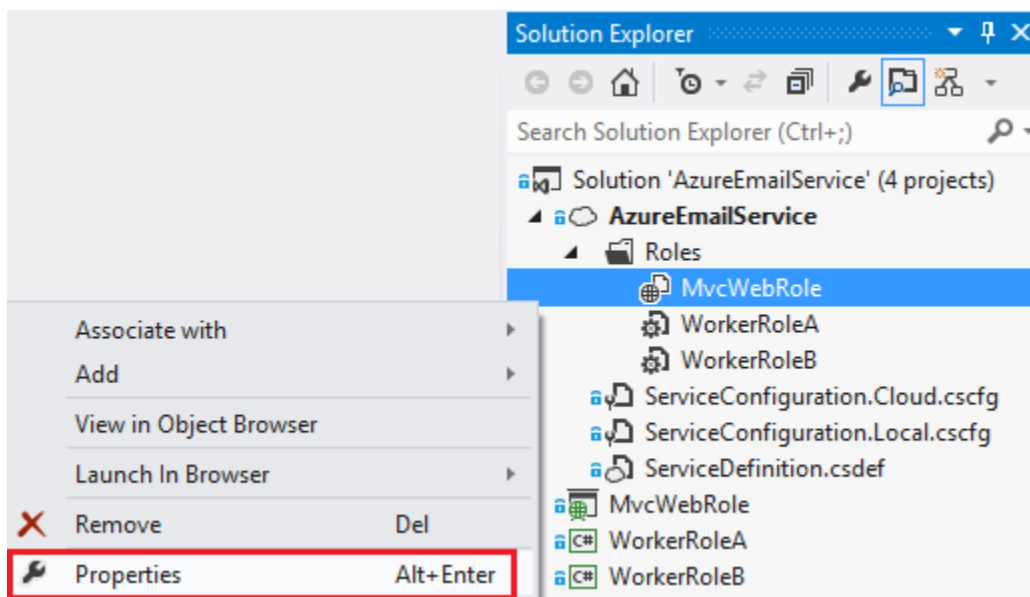
Worker role B in this application is the limiting component under high load because it does the work of sending emails. (Worker role A just creates queue messages, which is not resource-intensive.) Because worker role B is not multi-threaded and does not have a large memory footprint, it's not a good candidate for scale up.

Worker role B can scale linearly (that is, nearly double performance when you double the instances) by increasing the instance count. Increasing the number of compute instances is known as *scale out*. There is a cost for each instance, so you should only scale out when your application requires it.

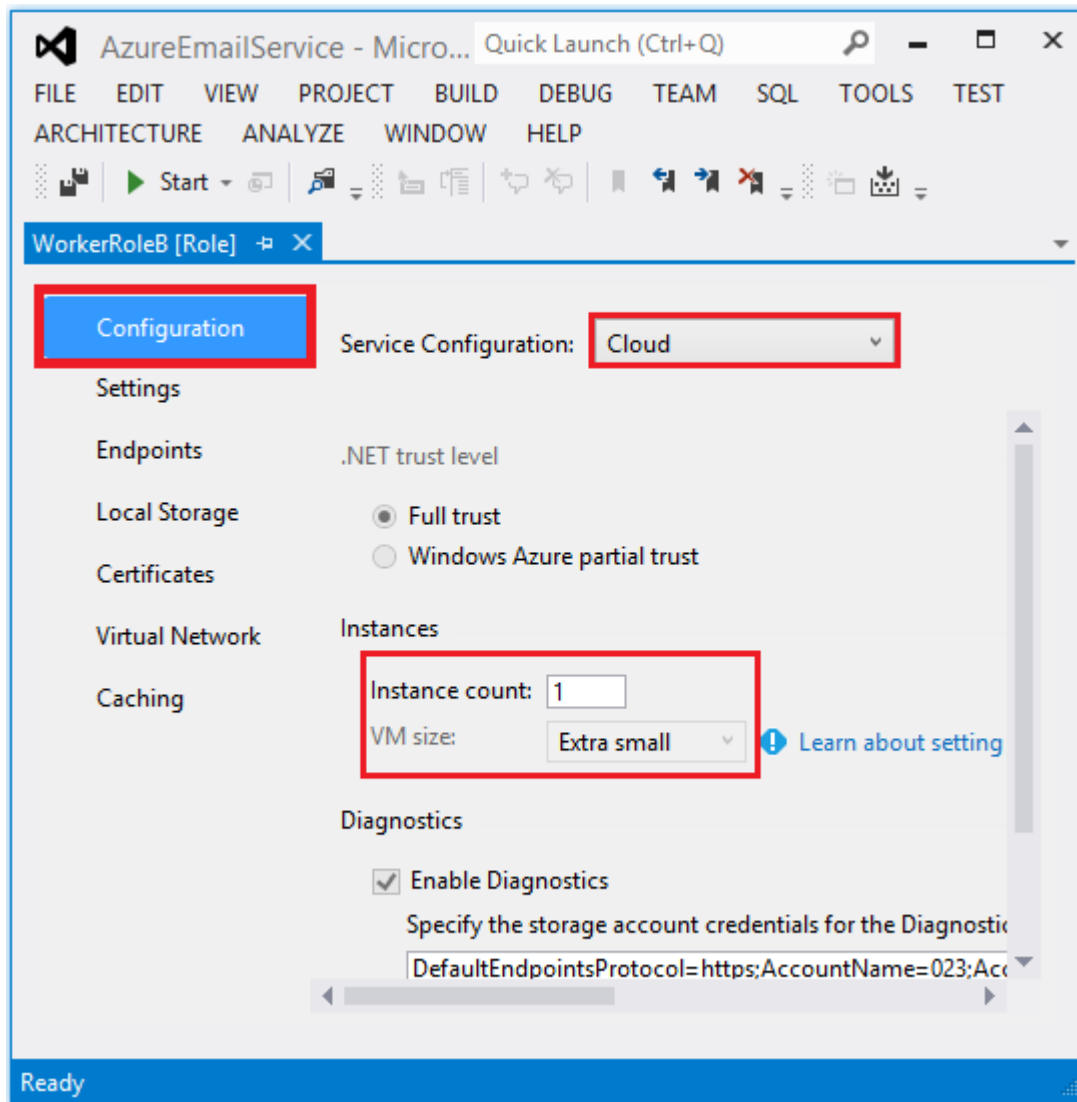
You can scale out a web or worker role by updating the setting in the Visual Studio UI or by editing the *ServiceConfiguration.cscfg** files directly. The instance count is specified in the Configuration tab of the role Properties window and in the *Instances* element in the *.cscfg* files. When you update the setting, you have to deploy the updated configuration file to make the change take effect. Alternatively, for transient increases in load, you can change the number of role instances manually or configure Azure to automatically change the number of instances based on criteria you specify. For more information on autoscaling, see [the last tutorial in this series](#).

In this section of the tutorial you'll scale out worker role B by using the management portal, but first you'll see how it's done in Visual Studio.

To do it in Visual Studio, you would right-click the role under Roles in the cloud project and select Properties.



You would then select the Configuration tab on the left, and select Cloud in the Service Configuration drop down.



Notice that you can also configure the VM size in this tab.

The following steps explain how to scale out by using the Azure Management Portal.

1. In the Azure Management Portal, select your cloud service, then click Scale.
2. Increase the number of instances for worker role B, and then click Save.

workerroleb

EDIT SCALE SETTINGS FOR SCHEDULE No scheduled times

set up schedule times

SCALE BY METRIC

NONE

CPU

QUEUE

INSTANCES



INSTANCE COUNT

A1 (1 CORE, 1.75 GB MEMORY)



3

instance(s)



SAVE



DISCARD

2



It can take a few minutes for the new VMs to be provisioned.

3. Select the Instances tab to see each role instance in your application.

aetest

 DASHBOARD MONITOR CONFIGURE SCALE **INSTANCES** LINKED RESOURCES CERTIFICATES

PRODUCTION **STAGING**

NAME	STATUS	ROLE	SIZE	UPDATE DOMAIN	FAULT DOMAIN
WorkerRoleB_IN_0	✓ Running	WorkerRoleB	Extra Small	0	0
WorkerRoleA_IN_0	✓ Running	WorkerRoleA	Small	0	0
MvcWebRole_IN_0	✓ Running	MvcWebRole	Small	0	0
WorkerRoleB_IN_1	✓ Running	WorkerRoleB	Extra Small	1	1

Next steps

You have now seen how to configure, deploy, and scale the completed application. The following tutorials show how to build the application from scratch. In the next tutorial you'll build the web role.

Building the web role for the Azure Email Service application - 3 of 5.

This is the third tutorial in a series of five that show how to build and deploy the Azure Email Service sample application. For information about the application and the tutorial series, see the first tutorial in the series.

In this tutorial you'll learn:

How to create a solution that contains a Cloud Service project with a web role and a worker role.

How to work with Azure tables, blobs, and queues in MVC 5 controllers and views.

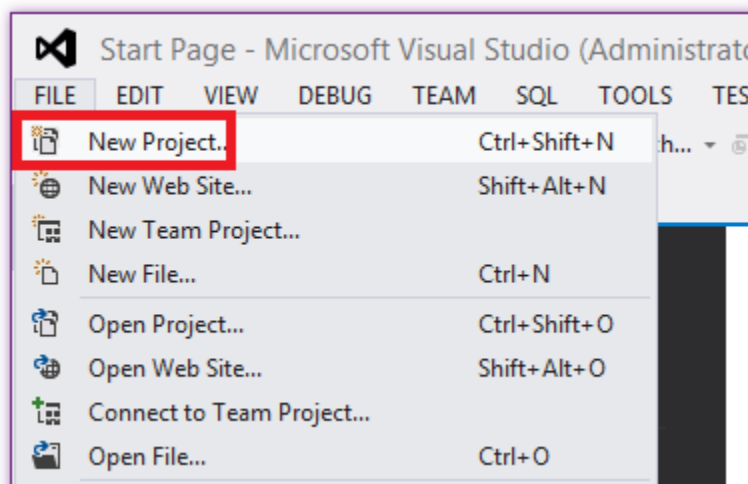
How to handle concurrency conflicts when you are working with Azure tables.

Create the Visual Studio solution

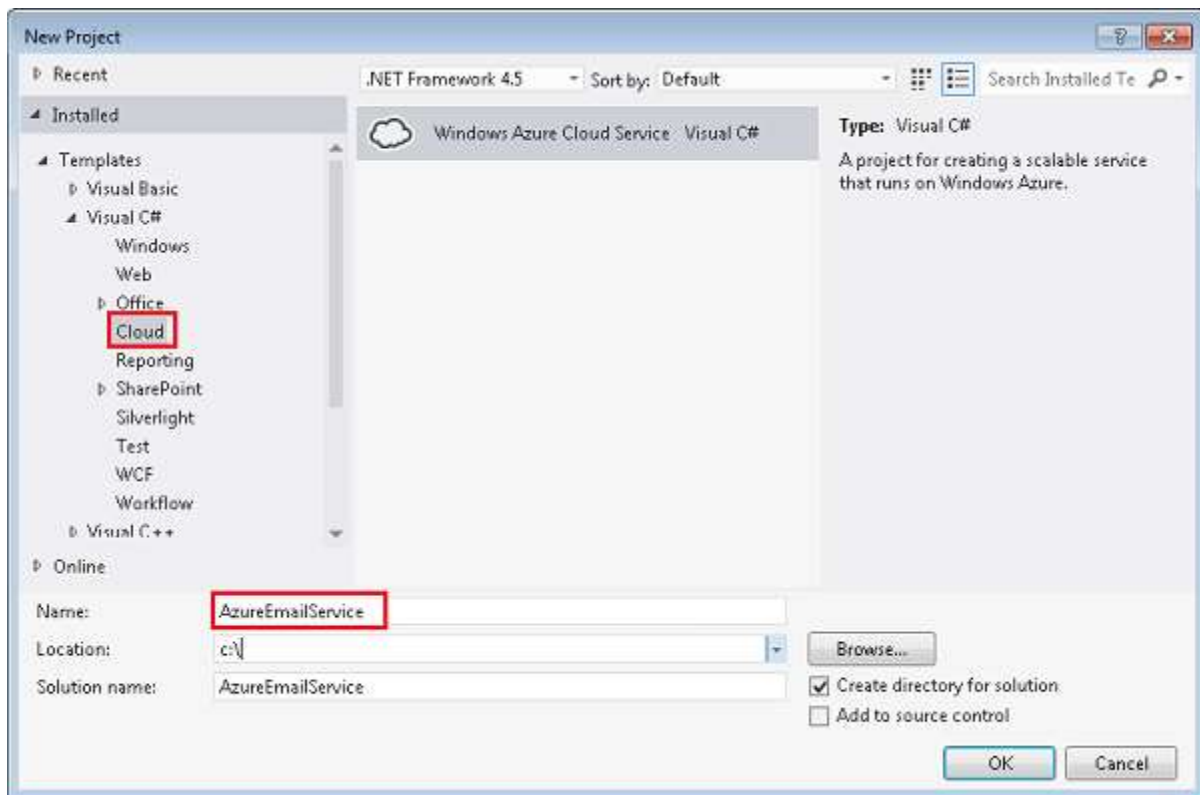
You begin by creating a Visual Studio solution with a project for the web front-end and a project for one of the back-end Azure worker roles. You'll add the second worker role later.

Create a cloud service project with a web role and a worker role

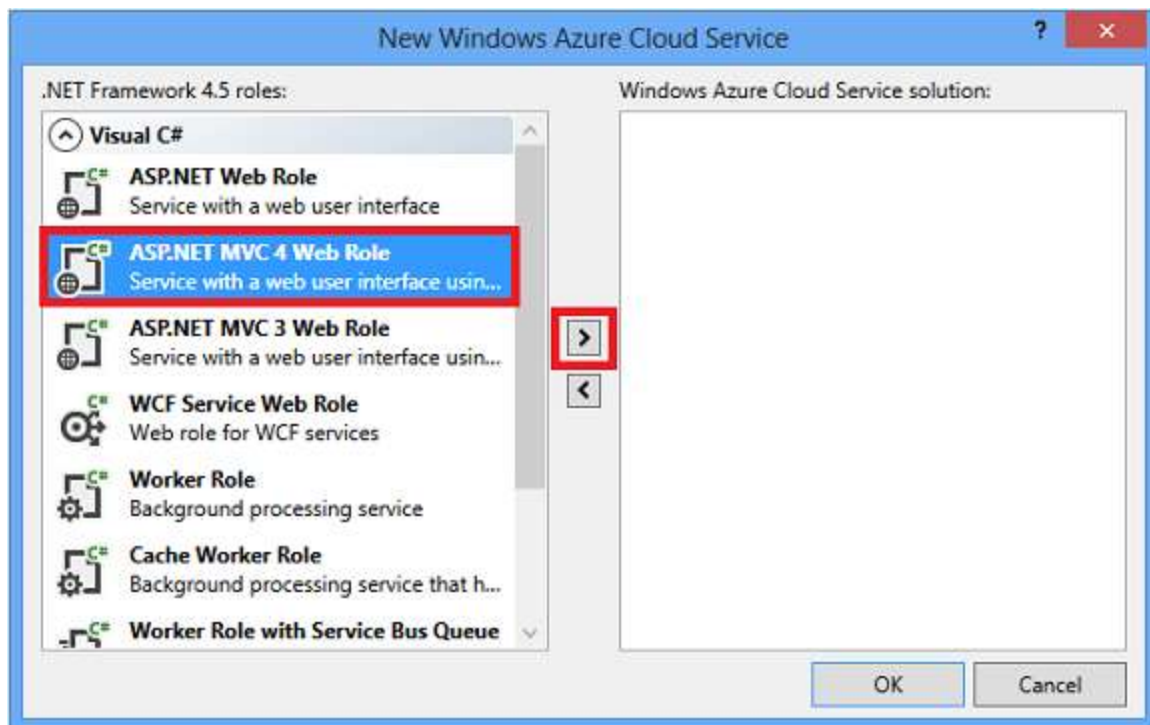
1. Start Visual Studio.
2. From the File menu select New Project.



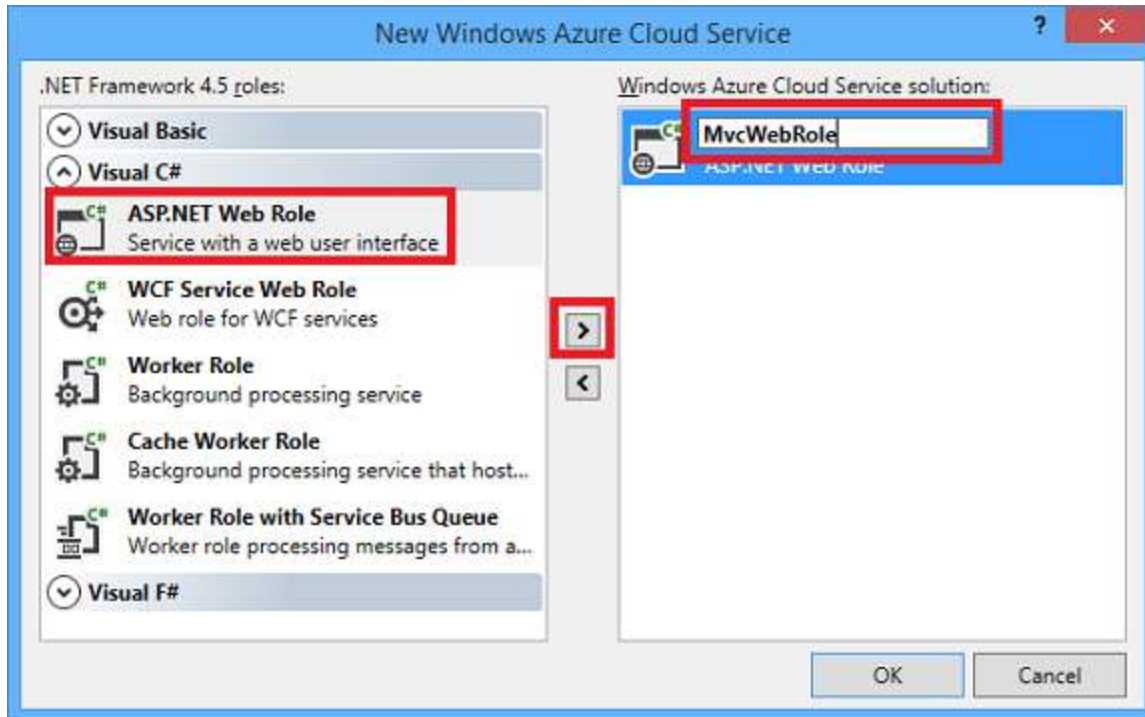
3. Expand C# and select Cloud under Installed Templates, and then select Azure Cloud Service.
4. Name the application AzureEmailService and click OK.



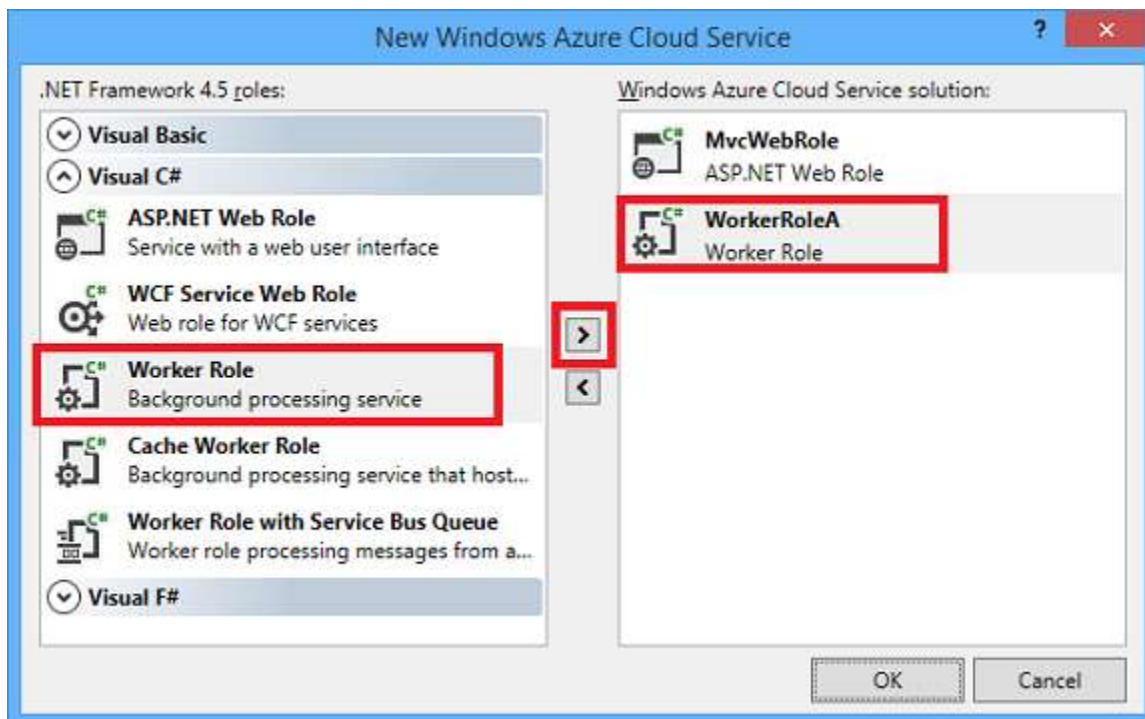
5. In the New Azure Cloud Service dialog box, select ASP.NET Web Role and click the arrow that points to the right.



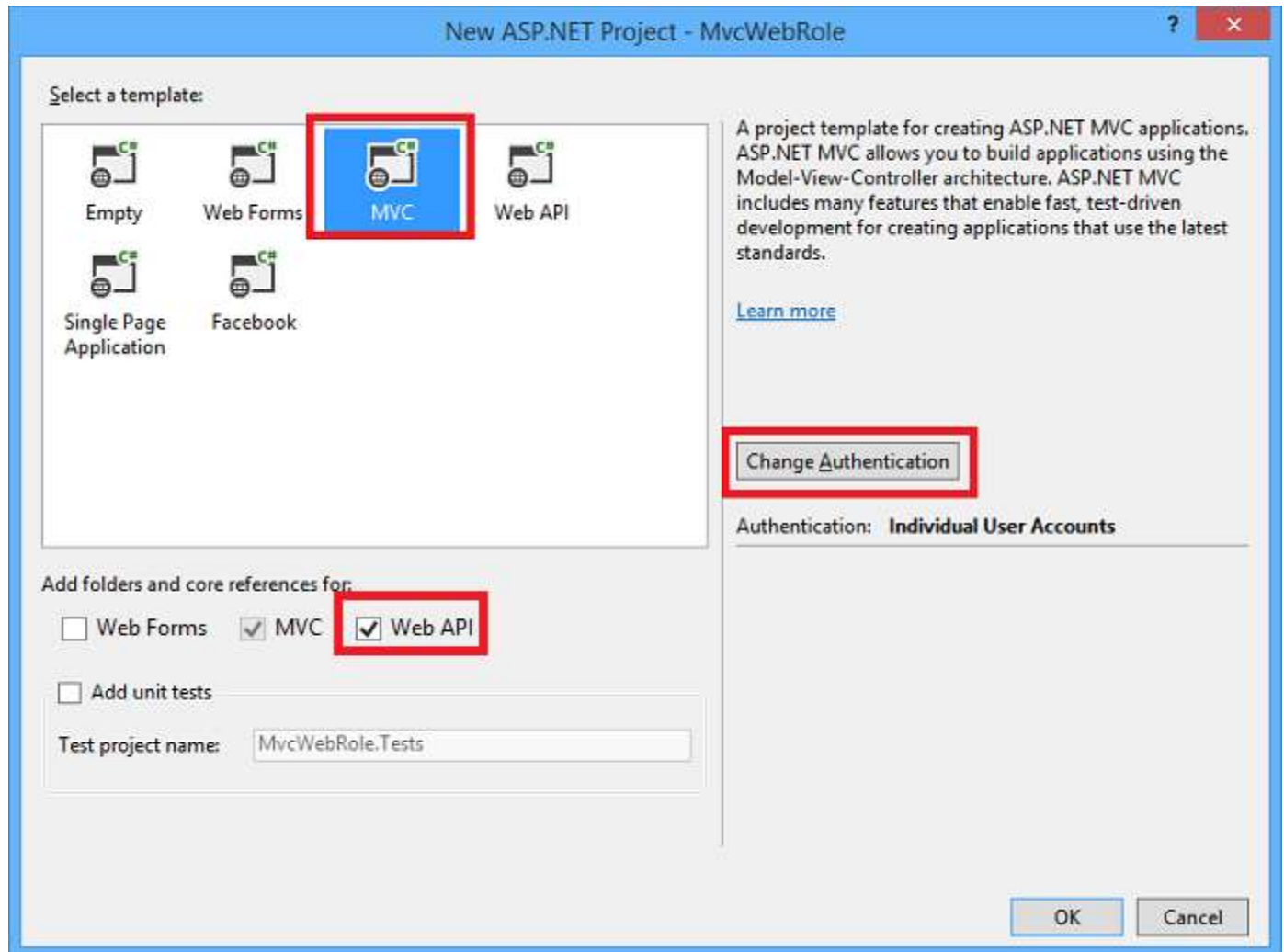
6. In the column on the right, hover the pointer over WebRole1, and then click the pencil icon to change the name of the web role.
7. Enter MvcWebRole as the new name, and then press Enter.



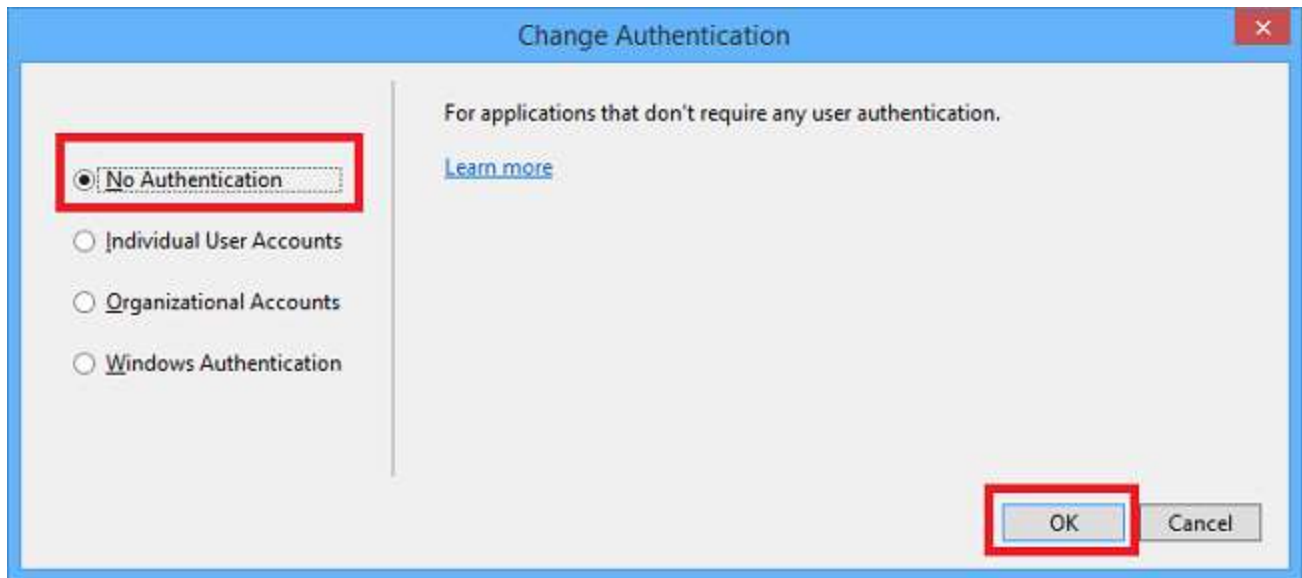
8. Follow the same procedure to add a Worker Role, name it WorkerRoleA, and then click OK.



9. In the New ASP.NET Project dialog box, select the MVC template, select the Web API check box, and then click Change Authentication.



10. In the Change Authentication dialog box, click No Authentication, and then click OK.



11. In the New ASP.NET Project dialog box, click OK.

Set the page header, menu, and footer

In this section you update the headers, footers, and menu items that are shown on every page for the administrator web UI. The application will have three sets of administrator web pages: one for Mailing Lists, one for Subscribers to mailing lists, and one for Messages.

1. If you haven't already downloaded the [completed solution](#), do that before continuing with the next step.

In the remainder of the tutorial, when you need to add code you'll copy files from the downloaded project into the new project, instead of copying and pasting snippets. The tutorial will show and explain key parts of the code that you're copying.

To add a file from the downloaded project, right-click the project you want to add the file to, or the folder you want to add it to, and choose Add - Existing Item from the context menu. Then navigate to where you downloaded the completed project, select the file(s) you want, and click Add. If the you get a Destination File Exists dialog box, click Yes.

2. In the MvcWebRole project, add the *Views\Shared_Layout.cshtml* file from the downloaded project (right-click the *Shared* folder under *Views* to add the file).

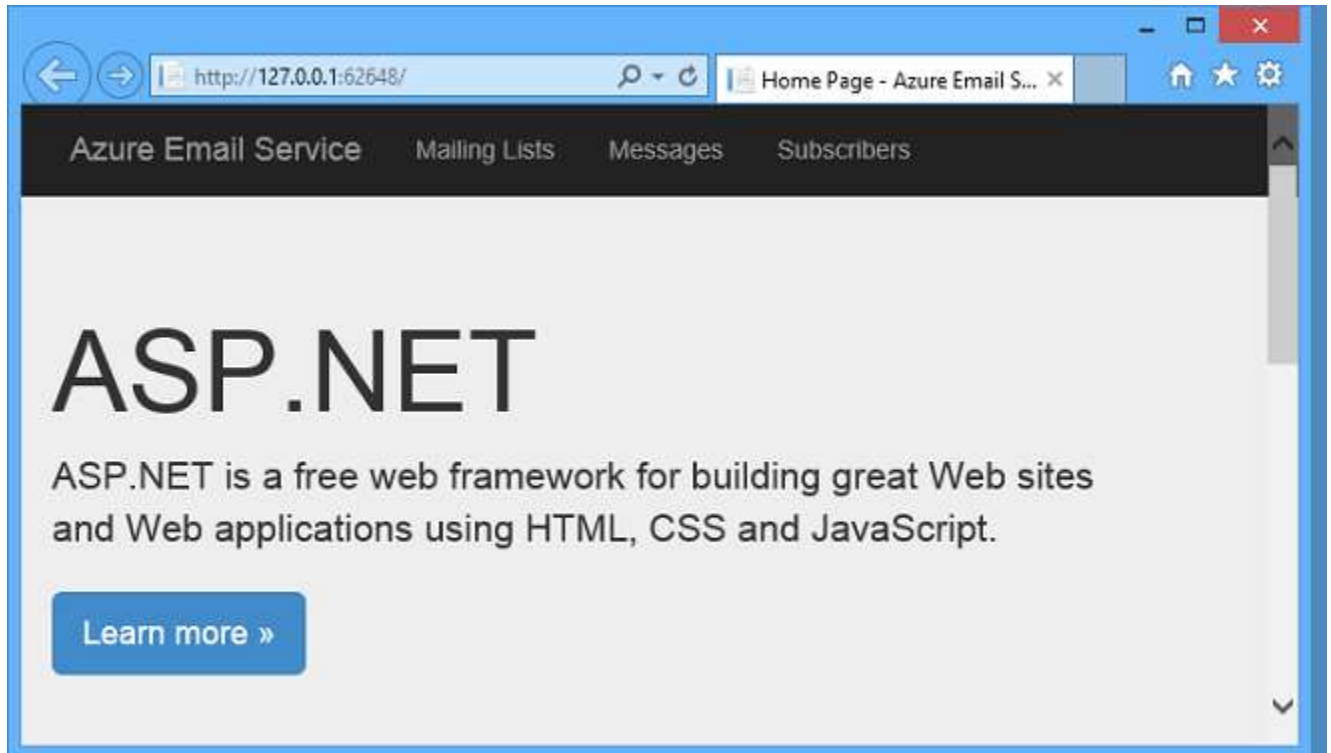
This adds header, footer, and menu entries for Mailing List, Message, and Subscriber pages:

```
<ul class="nav navbar-nav">
  <li>@Html.ActionLink("Mailing Lists", "Index", "MailingList")</li>
  <li>@Html.ActionLink("Messages", "Index", "Message")</li>
  <li>@Html.ActionLink("Subscribers", "Index", "Subscriber")</li>
```

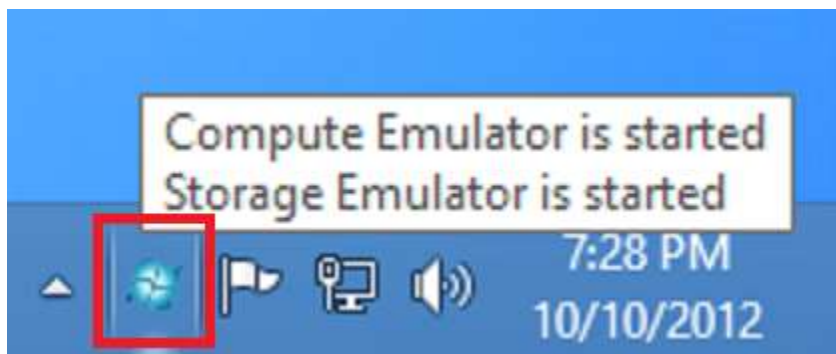

Run the application locally

1. Press CTRL+F5 to run the application.

If you're used to starting web projects that aren't Azure cloud service projects, you'll notice it takes longer than usual before you see the home page in the browser.



The delay is because Visual Studio starts the Azure compute emulator and Azure storage emulator. You can see the compute emulator icon in the Windows system tray:

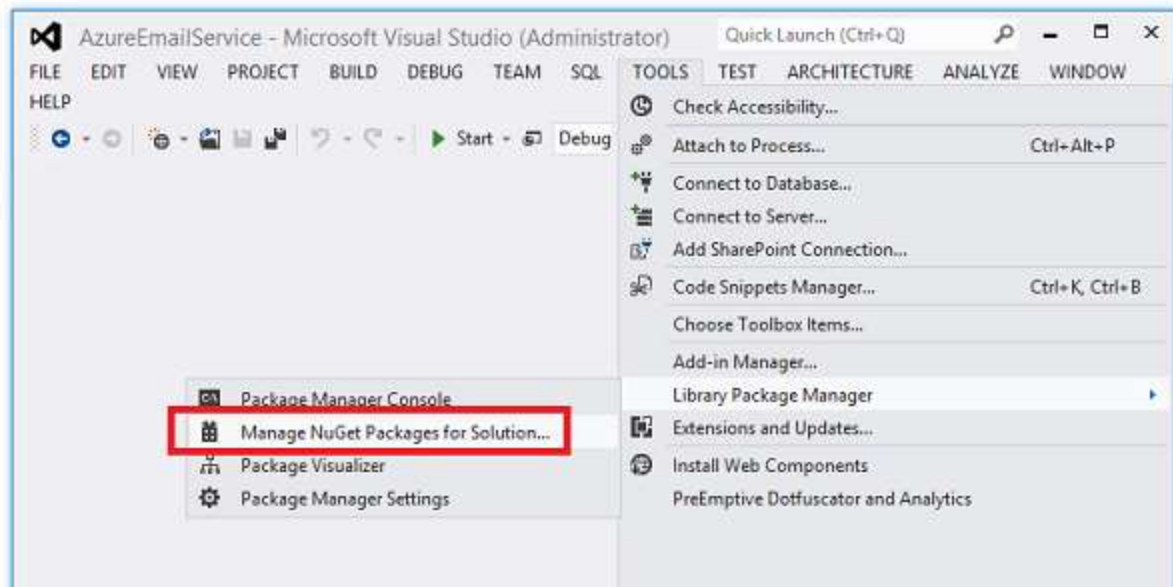


2. Close the browser.

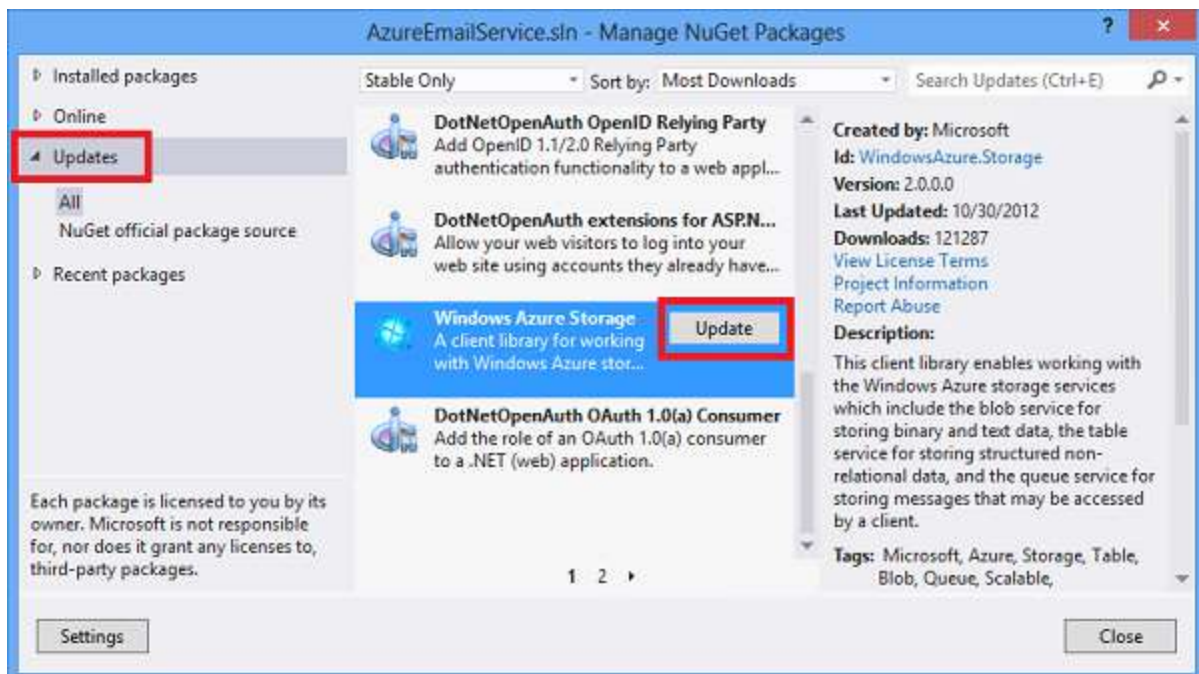
Update the Storage Client Library NuGet Package

The API framework that you use to work with Azure Storage tables, queues, and blobs is the Storage Client Library (SCL). This API is included in a NuGet package in the Cloud Service project template. However, updates to the SCL are often released after the project templates were created, so it's always a good idea to check if an update is available for your SCL NuGet package.

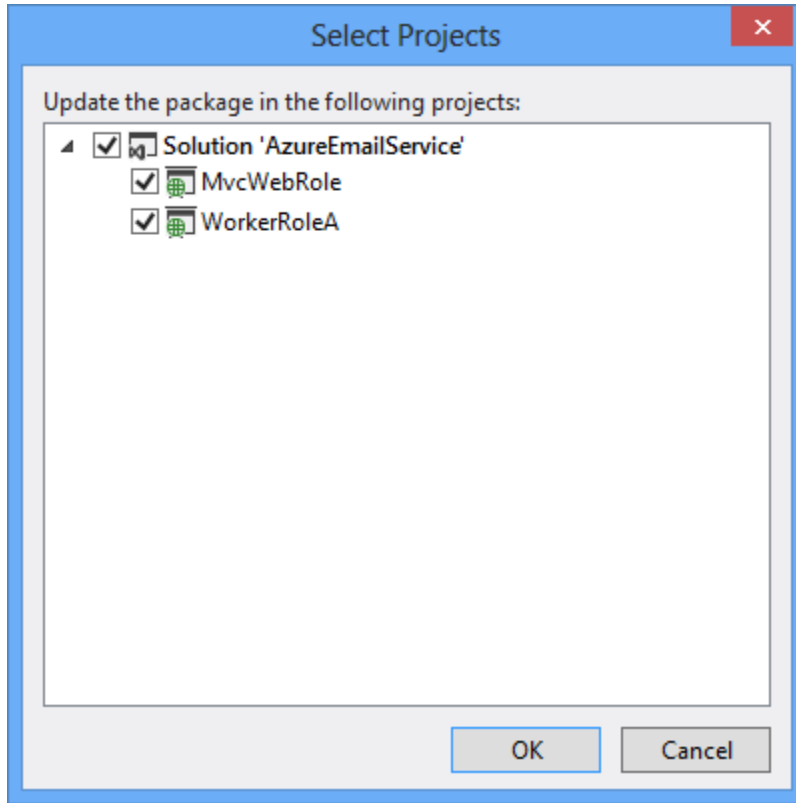
1. In the Visual Studio Tools menu, hover over Library Package Manager, and then click Manage NuGet Packages for Solution.



2. In the left pane of the Manage NuGet Packages dialog box, select Updates, then scroll down to the Azure Storage package and click Update.



3. In the Select Projects dialog box, make sure both projects are selected, and then click OK.

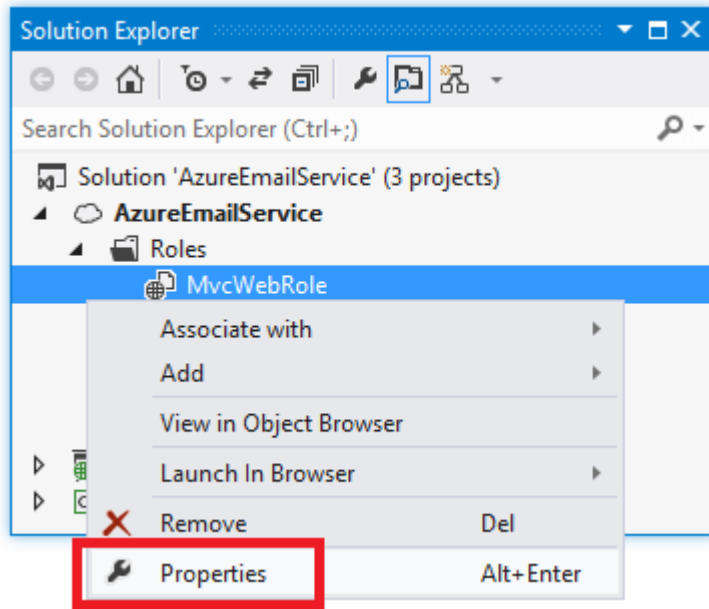


4. Accept the license terms to complete installation of the package, and then close the Manage NuGet Packages dialog box.

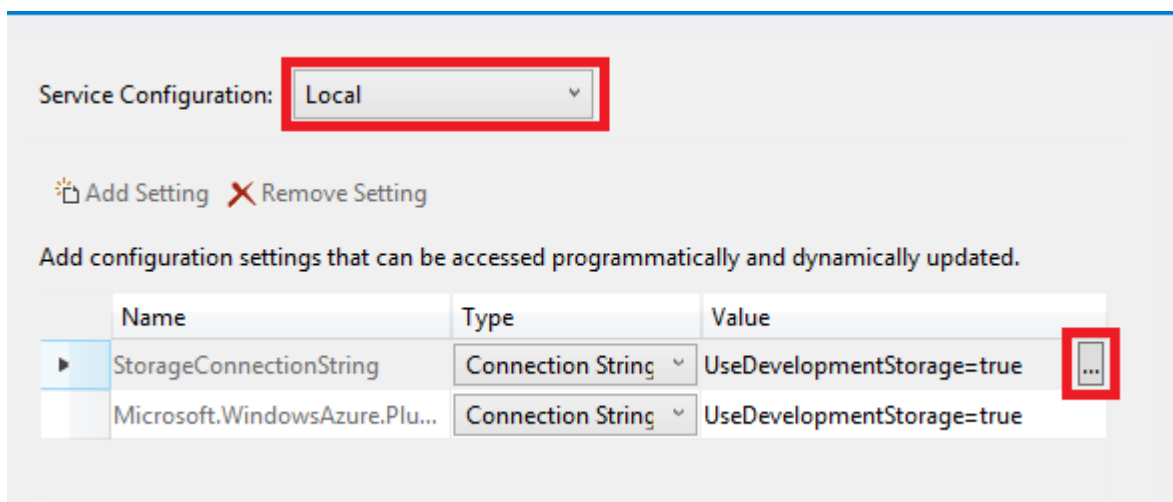
Configure the projects to use the storage emulator

The web role and worker role code that you'll add later will use a connection string named `StorageConnectionString` to connect to Azure Storage. In this section you'll add the setting to role properties and configure it to use the storage emulator. The second tutorial in the series shows how to configure the connection string to use an Azure storage account.

1. In Solution Explorer, right-click `MvcWebRole` under Roles in the `AzureEmailService` cloud project, and then choose Properties.



2. Make sure that All Configurations is selected in the Service Configuration drop-down list.
3. Select the Settings tab and then click Add Setting.
4. Enter "StorageConnectionString" in the Name column.
5. Select Connection String in the Type drop-down list.
6. Click the ellipsis (...) button at the right end of the line to open the Storage Account Connection String dialog box.



7. In the Create Storage Connection String dialog, click the Azure storage emulator radio button, and then click OK.

The `Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString` connection string is set to the storage emulator by default, so you don't have to change that.

8. Follow the same procedure that you used for the `MvcWebRole` to add the storage connection string for the `WorkerRoleA` role.

When you added a new setting with the Add Settings button, the new setting was added to the XML in the *ServiceDefinition.csdf* file and in each of the two *.cscfg* configuration files. The following XML is added by Visual Studio to the *ServiceDefinition.csdf* file.

```
<ConfigurationSettings>
  <Setting name="StorageConnectionString" />
</ConfigurationSettings>
```

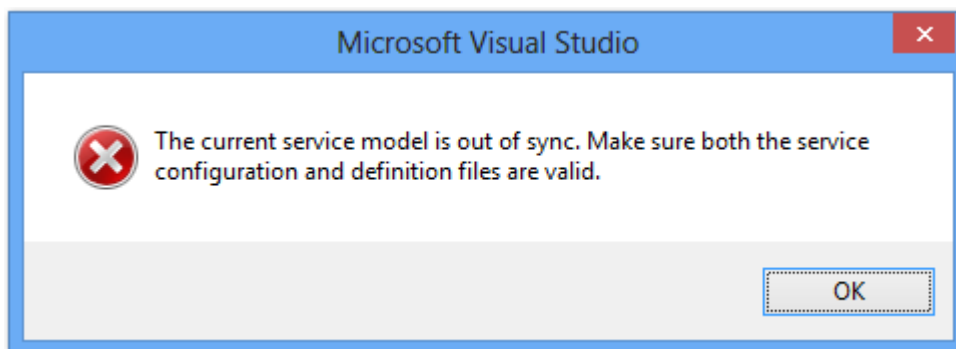
The following XML is added to each *.cscfg* configuration file.

```
<Setting name="StorageConnectionString"
value="UseDevelopmentStorage=true" />
```

You can manually add settings to the *ServiceDefinition.csdf* file and the two *.cscfg* configuration files, but using the properties editor has the following advantages for connection strings:

You only add the new setting in one place, and the correct setting XML is added to all three files.

The correct XML is generated for the three settings files. The *ServiceDefinition.csdf* file defines settings that must be in each *.cscfg* configuration file. If the *ServiceDefinition.csdf* file and the two *.cscfg* configuration files settings are inconsistent, you can get the following error message from Visual Studio: *The current service model is out of sync. Make sure both the service configuration and definition files are valid.*



If you get this error, the properties editor will not work until you resolve the inconsistency problem manually by editing the files.

Configure tracing and handle restarts

1. In the MvcWebRole project, add the *WebRole.cs* file from the downloaded project.

This adds a method that configures logging, and calls it from the `OnStart` method that executes when the web role starts. The code in the new `ConfigureDiagnostics` method is explained in [the second tutorial](#).

This also adds code that runs when the web role is notified that it's about to be shut down. Azure Cloud Service applications are restarted approximately twice per month for operating system updates. (For more information on OS updates, see [Role Instance Restarts Due to OS Upgrades](#).) When a web application is going to be shut down, an `OnStop` event is raised. The web role boiler plate created by Visual Studio does not override the `OnStop` method, so the application will have only a few seconds to finish processing HTTP requests before it is shut down. You can add code to override the `OnStop` method in order to ensure that shutdowns are handled gracefully.

The file you just added contains the following `OnStop` method override.

```
public override void OnStop()
{
    Trace.TraceInformation("OnStop called from WebRole");
    var rcCounter = new PerformanceCounter("ASP.NET", "Requests
Current", "");
    while (rcCounter.NextValue() > 0)
    {
        Trace.TraceInformation("ASP.NET Requests Current = " +
rcCounter.NextValue().ToString());
        System.Threading.Thread.Sleep(1000);
    }
}
```

The `OnStop` method has up to 5 minutes to exit before the application is shut down. You could add a sleep call for 5 minutes to the `OnStop` method to give your application the maximum amount of time to process the current requests, but if your application is scaled correctly, it should be able to process the remaining requests in much less than 5 minutes. It is best to stop as quickly as possible, so that the application can restart as quickly as possible and continue processing requests.

Once a role is taken off-line by Azure, the load balancer stops sending requests to the role instance, and after that the `OnStop` method is called. If you don't have another instance of your role, no requests will be processed until your role completes shutting down and is restarted (which typically takes several minutes). That is one reason why the Azure service level agreement requires you to have at least two instances of each role in order to take advantage of the up-time guarantee.

In the code shown for the `OnStop` method, an ASP.NET performance counter is created for `Requests Current`. The `Requests Current` counter value contains the current number of requests, including those that are queued, currently executing, or waiting to be written to the client. The `Requests Current` value is checked every second, and once it falls to zero, the `OnStop` method returns. Once `OnStop` returns, the role shuts down.

Trace data is not saved when called from the `OnStop` method without performing an [On-Demand Transfer](#). You can view the `OnStop` trace information in real time with the [dbgview](#) utility from a remote desktop connection.

Add code to create tables, queue, and blob container in the `Application_Start` method

The web application will use the `MailingList` table, the `Message` table, the `azuremailsubscribequeue` queue, and the `azuremailblobcontainer` blob container. You could create these manually by using a tool such as Azure Storage Explorer, but then you would have to do that manually every time you started to use the application with a new storage account. In this section you'll add code that runs when the application starts, checks if the required tables, queues, and blob containers exist, and creates them if they don't.

You could add this one-time startup code to the `OnStart` method in the `WebRole.cs` file, or to the `Global.asax` file. For this tutorial you'll initialize Azure Storage in the `Global.asax` file.

1. In Solution Explorer, right-click the `MvcWebRole` project, and add the `Global.asax.cs` file from the downloaded project.

You've added a new method that is called from the `Application_Start` method:

```
private static void CreateTablesQueuesBlobContainers()
{
    var storageAccount =
CloudStorageAccount.Parse(RoleEnvironment.GetConfigurationSettingValue("StorageConnectionString"));

    var tableClient = storageAccount.CreateCloudTableClient();
    var mailingListTable =
tableClient.GetTableReference("MailingList");
    mailingListTable.CreateIfNotExists();
}
```

```

        var messageTable = tableClient.GetTableReference("Message");
        messageTable.CreateIfNotExists();

        var blobClient = storageAccount.CreateCloudBlobClient();
        var blobContainer =
blobClient.GetContainerReference("azuremailblobcontainer");
        blobContainer.CreateIfNotExists();

        var queueClient = storageAccount.CreateCloudQueueClient();
        var subscribeQueue =
queueClient.GetQueueReference("azuremailsubscribequeue");
        subscribeQueue.CreateIfNotExists();
    }

```

In the following sections you build the components of the web application, and you can test them with development storage or your storage account without having to manually create tables, queues, or blob container first.

Create and test the Mailing List controller and views

The Mailing List web UI is used by administrators to create, edit and display mailing lists, such as "Contoso University History Department announcements" and "Fabrikam Engineering job postings".

Add the MailingList entity class to the Models folder

The `MailingList` entity class is used for the rows in the `MailingList` table that contain information about the list, such as its description and the "From" email address for emails sent to the list.

1. In the `Models` folder in the MVC project, add the *MailingList.cs* file from the downloaded project.

You'll use the `MailingList` entity class for reading and writing mailing list rows in the mailinglist table.

```

public class MailingList : TableEntity
{
    public MailingList()
    {
        this.RowKey = "mailinglist";
    }
}

```

```

    }

    [Required]
    [RegularExpression(@"[\w]+",
        ErrorMessage = @"Only alphanumeric characters and underscore (_) are
allowed.")]
    [Display(Name = "List Name")]
    public string ListName
    {
        get
        {
            return this.PartitionKey;
        }
        set
        {
            this.PartitionKey = value;
        }
    }

    [Required]
    [Display(Name = "'From' Email Address")]
    public string FromEmailAddress { get; set; }

    public string Description { get; set; }
}

```

The Azure Storage API requires that entity classes for table operations derive from [TableEntity](#).

TableEntity defines PartitionKey, RowKey, TimeStamp, and ETag fields. The TimeStamp and ETag properties are used by the system. You'll see how the ETag property is used for concurrency handling later in the tutorial.

(There is also a [DynamicTableEntity](#) class for use when you want to work with table rows as Dictionary collections of key value pairs instead of by using predefined model classes. For more information, see [Azure Storage Client Library 2.0 Tables Deep Dive](#).)

The mailinglist table partition key is the list name. In this entity class the partition key value can be accessed either by using the PartitionKey property (defined in the TableEntity class) or the

ListName property (defined in the MailingList class). The ListName property uses PartitionKey as its backing variable. Defining the ListName property enables you to use a more descriptive variable name in code and makes it easier to program the web UI, since formatting and validation DataAnnotations attributes can be added to the ListName property, but they can't be added directly to the PartitionKey property.

The RegularExpression attribute on the ListName property causes MVC to validate user input to ensure that the list name value entered only contains alphanumeric characters or underscores. This restriction was implemented in order to keep list names simple so that they can easily be used in query strings in URLs.

NOTE:

If you wanted the list name format to be less restrictive, you could allow other characters and URL-encode list names when they are used in query strings. However, certain characters are not allowed in Azure Table partition keys or row keys, and you would have to exclude at least those characters. For information about characters that are not allowed or cause problems in the partition key or row key fields, see [Understanding the Table Service Data Model](#) and [% Character in PartitionKey or RowKey](#).

The MailingList class defines a default constructor that sets RowKey to the hard-coded string "mailinglist", because all of the mailing list rows in this table have that value as their row key. (For an explanation of the table structure, see the [first tutorial in the series](#).) Any constant value could have been chosen for this purpose, as long as it could never be the same as an email address, which is the row key for the subscriber rows in this table.

The list name and the "from" email address must always be entered when a new MailingList entity is created, so they have Required attributes.

The Display attributes specify the default caption to be used for a field in the MVC UI.

Add the MailingList MVC controller

1. In the *Controllers* folder in the MVC project, add the *MailingListController.cs* file from the downloaded project.

The controller's default constructor creates a CloudTable object to use for working with the mailinglist table.

```
public class MailingListController : Controller
{
    private CloudTable mailingListTable;

    public MailingListController()
```



```

{
    var storageAccount =
Microsoft.WindowsAzure.Storage.CloudStorageAccount.Parse(RoleEnvironment.GetConfigurationSettingValue("StorageConnectionString"));

    var tableClient = storageAccount.CreateCloudTableClient();
    mailingListTable = tableClient.GetTableReference("mailinglist");
}

```

The code gets the credentials for your Azure Storage account from the Cloud Service project settings file in order to make a connection to the storage account. (You'll configure those settings later in this tutorial, before you test the controller.)

Next is a `FindRowAsync` method that is called whenever the controller needs to look up a specific mailing list entry of the `MailingList` table, for example to edit a mailing list entry. The code retrieves a single `MailingList` entity by using the partition key and row key values passed in to it. The rows that this controller edits are the ones that have "MailingList" as the row key, so "MailingList" could have been hard-coded for the row key, but specifying both partition key and row key is a pattern used for the `FindRow` methods in all of the controllers.

NOTE:

The application uses ASP.NET 4.5 async code for I/O operations in the web role in order to use server resources efficiently. For information about async code in web application, see [Use .NET 4.5's async support to avoid blocking calls](#).

```

private async Task<MailingList> FindRowAsync(string partitionKey, string
rowKey)
{
    var retrieveOperation =
TableOperation.Retrieve<MailingList>(partitionKey, rowKey);
    var retrievedResult = await
mailingListTable.ExecuteAsync(retrieveOperation);
    var mailingList = retrievedResult.Result as MailingList;
    if (mailingList == null)
    {
        throw new Exception("No mailing list found for: " + partitionKey);
    }
    return mailingList;
}

```

```
}
```

The code in this `FindRow` method returns a mailing list row. The code in the corresponding `FindRow` method in the `Subscriber` controller returns a subscriber row from the same `mailinglist` table. The code in the two methods is identical except for the the model type used with the [TableOperation.Retrieve](#) method.

```
private async Task<Subscriber> FindRowAsync(string partitionKey, string
rowKey)
{
    var retrieveOperation =
TableOperation.Retrieve<Subscriber>(partitionKey, rowKey);
    var retrievedResult = await
mailingListTable.ExecuteAsync(retrieveOperation);
    var subscriber = retrievedResult.Result as Subscriber;
    if (subscriber == null)
    {
        throw new Exception("No subscriber found for: " + partitionKey +
", " + rowKey);
    }
    return subscriber;
}
```

The `TableOperation` object returned by the `TableOperation.Retrieve` method specifies the schema (the properties) of the row or rows that you expect the query to return. A single table may have different schemas in different rows. Typically you specify the same model type when reading a row that was used to create the row.

The `Index` page displays all of the mailing list rows, so the query in the `Index` method returns all `MailingList` entities that have "mailinglist" as the row key (the other rows in the table have email address as the row key, and they contain subscriber information).

```
var query = new TableQuery<MailingList>()
    .Where(TableQuery.GenerateFilterCondition
        ("RowKey", QueryComparisons.Equal, "mailinglist"));
TableContinuationToken token = null;
OperationContext ctx = new OperationContext();
TableQuerySegment<MailingList> currentSegment = null;
```

```

while (currentSegment == null || currentSegment.ContinuationToken != null)
{
    currentSegment = await mailingListTable.ExecuteQuerySegmentedAsync
        (query, token, webUIRetryPolicy, ctx);
    lists.AddRange(currentSegment.Results);
    token = currentSegment.ContinuationToken;
}

```

The `ExecuteQuerySegmentedAsync` method breaks large result sets into segments. It returns up to 1,000 rows. When you execute a query that would retrieve more than 1,000 rows, you get 1,000 rows and a continuation token. You can use the continuation token to execute another query that starts where the previous one left off. The code shown is simplified for a sample application: it aggregates all segments into one list. For a production application you'd implement paging code. For more information about large result sets and continuation tokens, see [How to get most out of Azure Tables](#) and [Azure Tables: Expect Continuation Tokens, Seriously](#).

When you create the `OperationContext` object, you can set the `ClientID` property value in order to provide a unique identifier that will be included in logs written by Azure Storage. You can use this identifier to trace storage operation logs to the code that caused the storage service activity. For information about Azure storage logging, see [Azure Storage Logging: Using Logs to Track Storage Requests](#).

With the SCL 2.1 and later API you can also use LINQ for your table queries. For a code sample that shows how to use LINQ, see [PhluffyFotos](#).

If you don't specify a retry policy, the API automatically retries three times with exponentially increasing timeout limits. For a web interface with a user waiting for a page to appear, this could result in unacceptably long wait times. Therefore, this code specifies linear retries (so the timeout limit doesn't increase each time) and a timeout limit that is reasonable for the user to wait. The retry policy is specified in the `webUIRetryPolicy` object that is passed to the `ExecuteQuerySegmentedAsync` method. The `webUIRetryPolicy` object is defined in the controller constructor:

```

private TableRequestOptions webUIRetryPolicy;

public MailingListController()
{
    // Other constructor code not shown.

    webUIRetryPolicy = new TableRequestOptions()

```

```

    {
        MaximumExecutionTime = TimeSpan.FromSeconds(1.5),
        RetryPolicy = new LinearRetry(TimeSpan.FromSeconds(3), 3)
    };
}

```

The Index method includes a try-catch block that is designed to handle timeout conditions.

```

try
{
    // Code not shown for retrieving MailingList rows.
}
catch (StorageException se)
{
    ViewBag.errorMessage = "Timeout error, try again. ";
    Trace.TraceError(se.Message);
    return View("Error");
}

```

When the user clicks the Create button on the Create page, the MVC model binder creates a MailingList entity from input entered in the view, and the `HttpPost Create` method adds the entity to the table.

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create(MailingList mailingList)
{
    if (ModelState.IsValid)
    {
        var insertOperation = TableOperation.Insert(mailingList);
        await mailingListTable.ExecuteAsync(insertOperation);
        return RedirectToAction("Index");
    }

    return View(mailingList);
}

```

For the Edit page, the `HttpPost Edit` method updates the row.

```
var replaceOperation = TableOperation.Replace(mailingList);
await mailingListTable.ExecuteAsync(replaceOperation);
```

In the `HttpPost Edit` method a catch block handles concurrency errors.

```
catch (StorageException ex)
{
    if (ex.RequestInformation.HttpStatusCode == 412)
    {
        // Concurrency error
        var retrieveOperation =
TableOperation.Retrieve<MailingList>(partitionKey, rowKey);
        var retrievedResult = mailingListTable.Execute(retrieveOperation);
        var currentMailingList = retrievedResult.Result as MailingList;
        if (currentMailingList == null)
        {
            ModelState.AddModelError(string.Empty, "The record you
attempted to edit "
                + "was deleted by another user after you got the original
value. The "
                + "edit operation was canceled. Click the Back to List
hyperlink.");
        }
        if (currentMailingList.FromEmailAddress !=
editedMailingList.FromEmailAddress)
        {
            ModelState.AddModelError("FromEmailAddress", "Current value: "
+ currentMailingList.FromEmailAddress);
        }
        if (currentMailingList.Description !=
editedMailingList.Description)
        {
            ModelState.AddModelError("Description", "Current value: " +
currentMailingList.Description);
        }
    }
}
```

```

        ModelState.AddModelError(string.Empty, "The record you attempted
to edit "
        + "was modified by another user after you got the original
value. The "
        + "edit operation was canceled and the current values in the
database "
        + "have been displayed. If you still want to edit this record,
click "
        + "the Save button again. Otherwise click the Back to List
hyperlink.");
        ModelState.SetModelValue("ETag", new
ValueProviderResult(currentMailingList.ETag, currentMailingList.ETag,
null));
    }
    else
    {
        throw;
    }
}

```

A concurrency exception is raised if a user selects a mailing list for editing, then while the Edit page is displayed in the browser another user edits the same mailing list. When that happens, the code displays a warning message and indicates which fields were changed by the other user. The TSL API uses the ETag to check for concurrency conflicts. Every time a table row is updated, the ETag value is changed. When you get a row to edit, you save the ETag value, and when you execute an update or delete operation you pass in the ETag value that you saved. (The Edit view has a hidden field for the ETag value.) If the update operation finds that the ETag value on the record you are updating is different than the ETag value that you passed in to the update operation, it raises a concurrency exception. If you don't care about concurrency conflicts, you can set the ETag field to an asterisk ("*") in the entity that you pass in to the update operation, and conflicts are ignored.

Note: The HTTP 412 error is not unique to concurrency errors. It can be raised for other errors by the SCL API.

For the Delete page, the `HttpPost Delete` method deletes the `MailingList` row along with any `Subscriber` rows that are associated with it in the `MailingList` table.

```
[HttpPost, ActionName("Delete")]
```

```

[ValidateAntiForgeryToken]
public async Task<ActionResult> DeleteConfirmed(string partitionKey)
{
    // Delete all rows for this mailing list, that is,
    // Subscriber rows as well as MailingList rows.
    // Therefore, no need to specify row key.

    // Get all rows for this mailing list. For a production app where this
    // could return too many split the work up into segments.
    var query = new
TableQuery<MailingList>().Where(TableQuery.GenerateFilterCondition("Partit
ionKey", QueryComparisons.Equal, partitionKey));
    TableContinuationToken token = null;
    OperationContext ctx = new OperationContext();
    TableQuerySegment<MailingList> currentSegment = null;
    List<MailingList> listRows = new List<MailingList>();
    while (currentSegment == null || currentSegment.ContinuationToken !=
null)
    {
        currentSegment = await
mailingListTable.ExecuteQuerySegmentedAsync(query, token,
webUIRetryPolicy, ctx);
        listRows.AddRange(currentSegment.Results);
        token = currentSegment.ContinuationToken;
    }

    // Delete the rows in batches of 100.
    var batchOperation = new TableBatchOperation();
    int itemsInBatch = 0;
    foreach (MailingList listRow in listRows)
    {
        batchOperation.Delete(listRow);
        itemsInBatch++;
        if (itemsInBatch == 100)
        {

```

```

        await mailingListTable.ExecuteBatchAsync(batchOperation);
        itemsInBatch = 0;
        batchOperation = new TableBatchOperation();
    }
}
if (itemsInBatch > 0)
{
    await mailingListTable.ExecuteBatchAsync(batchOperation);
}
return RedirectToAction("Index");
}

```

In case a large number of subscribers need to be deleted, the code deletes the records in batches. The transaction cost of deleting one row is the same as deleting 100 rows in a batch. The maximum number of operations that you can perform in one batch is 100.

Although the loop processes both `MailingList` rows and `Subscriber` rows, it reads them all into the `MailingList` entity class because the only fields needed for the `Delete` operation are the `PartitionKey`, `RowKey`, and `ETag` fields.

Add the MailingList MVC views

1. In the *Views* folder in the MVC project, create a new folder and name it *MailingList*.
2. In the new *Views\MailingList* folder, add all four of the *.cshtml* files from the downloaded project.

In the *Edit.cshtml* file notice the hidden field that is included to preserve the `ETag` value which is used for handling concurrency conflicts.

```

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    @Html.HiddenFor(model => model.ETag)

```

Notice also that the `ListName` field has a `DisplayFor` helper instead of an `EditorFor` helper.

```

<div class="form-group">
    @Html.LabelFor(model => model.ListName, htmlAttributes: new {
@class = "control-label col-md-2" })

```



```

        <div class="col-md-10">
            @Html.DisplayFor(model => model.ListName, new { htmlAttributes
= new { @class = "form-control" } })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.Description, htmlAttributes: new {
@class = "control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.Description, new {
htmlAttributes = new { @class = "form-control" } })
            @Html.ValidationMessageFor(model => model.Description, "", new
{ @class = "text-danger" })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.FromEmailAddress, htmlAttributes:
new { @class = "control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.FromEmailAddress, new {
htmlAttributes = new { @class = "form-control" } })
            @Html.ValidationMessageFor(model => model.FromEmailAddress,
"", new { @class = "text-danger" })
        </div>
    </div>

```

We didn't enable the Edit page to change the list name, because that would have required complex code in the controller: the `HttpPost Edit` method would have had to delete the existing mailing list row and all associated subscriber rows, and re-insert them all with the new key value. In a production application you might decide that the additional complexity is worthwhile. As you'll see later, the `Subscriber` controller does allow list name changes, since only one row at a time is affected.

In the *Index.cshtml* file, the Edit and Delete hyperlinks specify partition key and row key query string parameters in order to identify a specific row. For `MailingList` entities only the partition key is actually

needed since row key is always "MailingList", but both are kept so that the MVC view code is consistent across all controllers and views.

```
<td>
    @Html.ActionLink("Edit", "Edit", new { PartitionKey =
item.PartitionKey, RowKey=item.RowKey }) |
    @Html.ActionLink("Delete", "Delete", new { PartitionKey =
item.PartitionKey, RowKey=item.RowKey })
</td>
```

Make MailingList the default controller

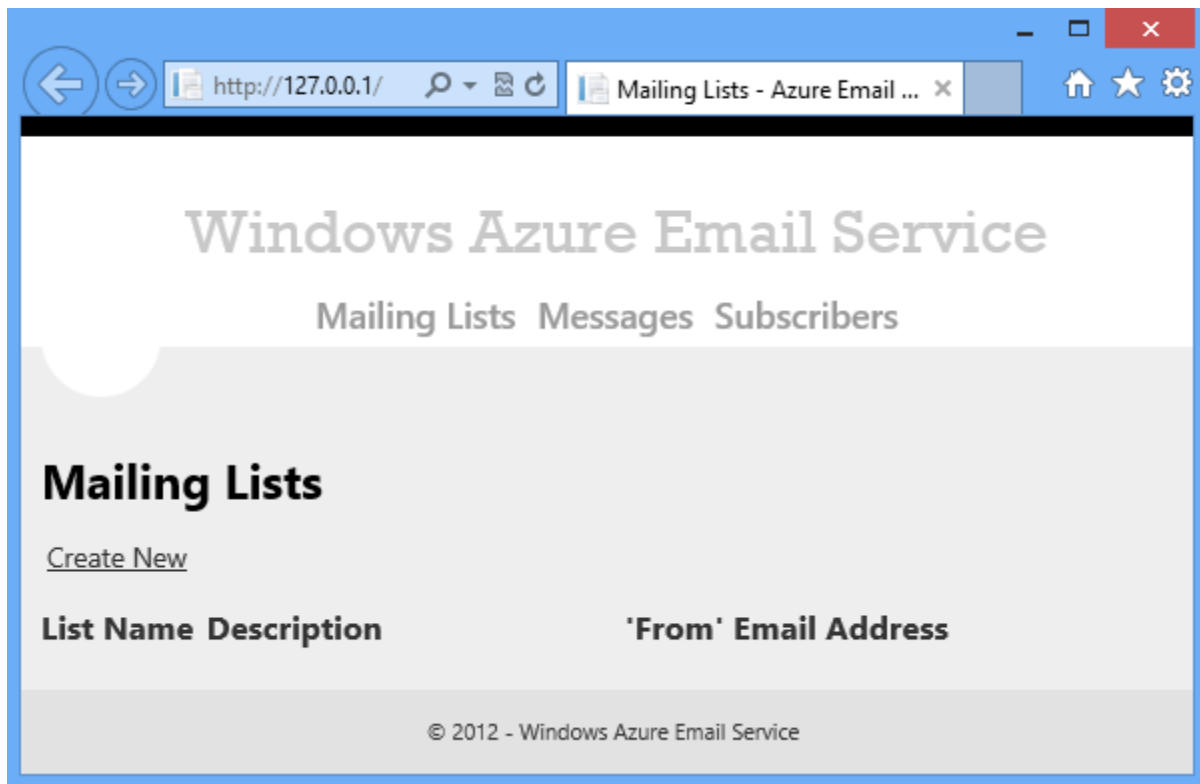
1. Open *Route.config.cs* in the *App_Start* folder.
2. In the line that specifies defaults, change the default controller from "Home" to "MailingList".

```
3. routes.MapRoute(
4.     name: "Default",
5.     url: "{controller}/{action}/{id}",

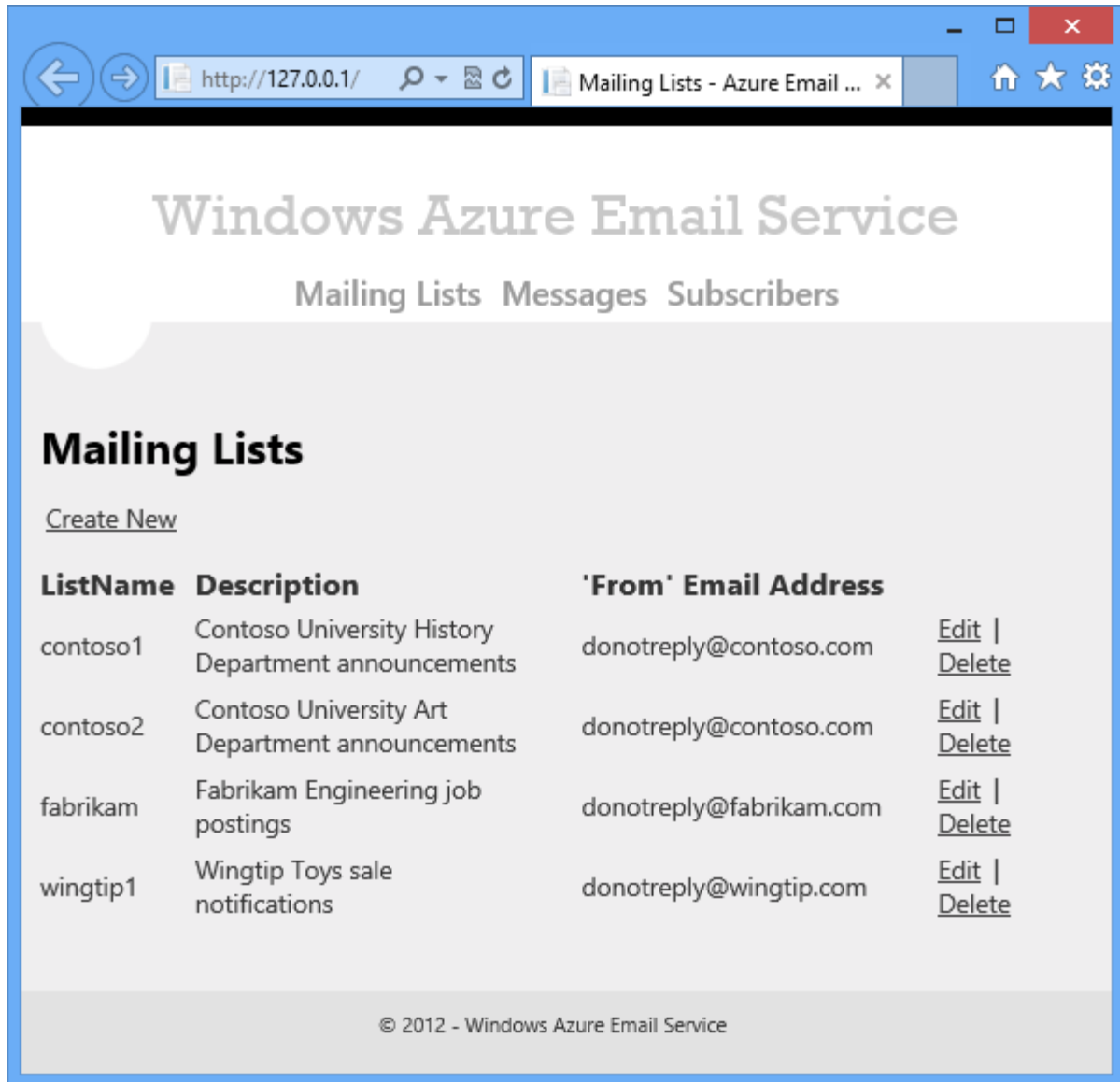
    defaults: new { controller = "MailingList", action = "Index", id =
UrlParameter.Optional }
```

Test the application

1. Run the project by pressing CTRL+F5.



2. Use the Create function to add some mailing lists, and try the Edit and Delete functions to make sure they work.



SubscriberCreate and test the Subscriber controller and views

The Subscriber web UI is used by administrators to add new subscribers to a mailing list, and to edit, display, and delete existing subscribers.

Add the Subscriber entity class to the Models folder

The `Subscriber` entity class is used for the rows in the `MailingList` table that contain information about subscribers to a list. These rows contain information such as the person's email address and whether the address is verified.

1. In the *Models* folder in the MVC project, add the *Subscriber.cs* file from the downloaded project.

Like the *MailingList* entity class, the *Subscriber* entity class is used to read and write rows in the *mailinglist* table.

```
public class Subscriber : TableEntity
{
    [Required]
    public string ListName
    {
        get
        {
            return this.PartitionKey;
        }
        set
        {
            this.PartitionKey = value;
        }
    }

    [Required]
    [Display(Name = "Email Address")]
    public string EmailAddress
    {
        get
        {
            return this.RowKey;
        }
        set
        {
            this.RowKey = value;
        }
    }

    public string SubscriberGUID { get; set; }

    public bool? Verified { get; set; }
```

```
}
```

`Subscriber` rows use the email address instead of the constant "mailinglist" for the row key. (For an explanation of the table structure, see the first tutorial in the series.) Therefore an `EmailAddress` property is defined that uses the `RowKey` property as its backing field, the same way that `ListName` uses `PartitionKey` as its backing field. As explained earlier, this enables you to put formatting and validation `DataAnnotations` attributes on the properties.

The `SubscriberGUID` value is generated when a `Subscriber` entity is created. It is used in subscribe and unsubscribe links to help ensure that only authorized persons can subscribe or unsubscribe email addresses.

When a row is initially created for a new subscriber, the `Verified` value is `false`. The `Verified` value changes to `true` only after the new subscriber clicks the Confirm hyperlink in the welcome email. If a message is sent to a list while a subscriber has `Verified = false`, no email is sent to that subscriber.

The `Verified` property in the `Subscriber` entity is defined as nullable. When you specify that a query should return `Subscriber` entities, it is possible that some of the retrieved rows might not have a `Verified` property. Therefore the `Subscriber` entity defines its `Verified` property as nullable so that it can more accurately reflect the actual content of a row if table rows that don't have a *Verified* property are returned by a query. You might be accustomed to working with SQL Server tables, in which every row of a table has the same schema. In an Azure Storage table, each row is just a collection of properties, and each row can have a different set of properties. For example, in the Azure Email Service sample application, rows that have "MailingList" as the row key don't have a `Verified` property. If a query returns a table row that doesn't have a `Verified` property, when the `Subscriber` entity class is instantiated, the `Verified` property in the entity object will be null. If the property were not nullable, you would get the same value of `false` for rows that have `Verified = false` and for rows that don't have a `Verified` property at all. Therefore, a best practice for working with Azure Tables is to make each property of an entity class nullable in order to accurately read rows that were created by using different entity classes or different versions of the current entity class.

Add the Subscriber MVC controller

1. In Solution Explorer, right-click the *Controllers* folder in the MVC project, and choose Add Existing Item.
2. Navigate to the folder where you downloaded the sample application, select the *SubscriberController.cs* file in the *Controllers* folder, and click Add. (Make sure that you get *Subscriber.cs* and not *Subscribe.cs*; you'll add *Subscribe.cs* later.)

Most of the code in this controller is similar to what you saw in the `MailingList` controller. Even the table name is the same because subscriber information is kept in the `MailingList` table.

In addition to the `FindRowAsync` method there's a `FindRow` method, since there is a need to call it from a catch block, and you can't call an async method from a catch block.

After the `FindRow` methods you see a `GetListNamesAsync` method. This method gets the data for a drop-down list on the Create and Edit pages, from which you can select the mailing list to subscribe an email address to.

```
private async Task<List<MailingList>> GetListNamesAsync()
{
    List<MailingList> lists = new List<MailingList>();
    var query = (new
TableQuery<MailingList>().Where(TableQuery.GenerateFilterCondition("RowKey",
QueryComparisons.Equal, "mailinglist")));
    TableContinuationToken token = null;
    OperationContext ctx = new OperationContext();
    TableQuerySegment<MailingList> currentSegment = null;
    while (currentSegment == null || currentSegment.ContinuationToken
!= null)
    {
        currentSegment = await
mailingListTable.ExecuteQuerySegmentedAsync(query, token,
webUIRetryPolicy, ctx);
        lists.AddRange(currentSegment.Results);
        token = currentSegment.ContinuationToken;
    }
    return lists;
}
```

This is the same query you saw in the `MailingList` controller. For the drop-down list you want rows that have information about mailing lists, so you select only those that have `RowKey = "mailinglist"`.

For the method that retrieves data for the Index page, you want rows that have subscriber information, so you select all rows that do not have `RowKey = "MailingList"`.

```
var query = (new TableQuery<Subscriber>()
    .Where(TableQuery.GenerateFilterCondition
        ("RowKey", QueryComparisons.NotEqual, "mailinglist")));
```

Notice that the query specifies that data will be read into `Subscriber` objects (by specifying `<Subscriber>`) but the data will be read from the `mailinglist` table.

The number of subscribers could grow to be too large to handle this way in a single query. As noted earlier, in a production application you would implement paging using continuation tokens.

In the `HttpGet Create` method, you set up data for the drop-down list; and in the `HttpPost` method, you set default values before saving the new entity.

```
public async Task<ActionResult> Create()
{
    var lists = await GetListNamesAsync();
    ViewBag.ListName = new SelectList(lists, "ListName",
    "Description");
    var model = new Subscriber() { Verified = false };
    return View(model);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create(Subscriber subscriber)
{
    if (ModelState.IsValid)
    {
        subscriber.SubscriberGUID = Guid.NewGuid().ToString();
        if (subscriber.Verified.HasValue == false)
        {
            subscriber.Verified = false;
        }

        var insertOperation = TableOperation.Insert(subscriber);
        await mailingListTable.ExecuteAsync(insertOperation);
        return RedirectToAction("Index");
    }

    var lists = await GetListNamesAsync();
    ViewBag.ListName = new SelectList(lists, "ListName",
    "Description", subscriber.ListName);
}
```



```
        return View(subscriber);  
    }  
}
```

The `HttpPost Edit` page is more complex than what you saw in the `MailingList` controller because the `Subscriber` page enables you to change the list name or email address, both of which are key fields. If the user changes one of these fields, you have to delete the existing record and add a new one instead of updating the existing record. The following code shows the part of the `edit` method that handles the different procedures for key versus non-key changes:

```
UpdateModel(editedSubscriber, string.Empty, null, excludeProperties);  
if (editedSubscriber.PartitionKey == partitionKey &&  
editedSubscriber.RowKey == rowKey)  
{  
    //Keys didn't change -- Update the row  
    var replaceOperation = TableOperation.Replace(editedSubscriber);  
    await mailingListTable.ExecuteAsync(replaceOperation);  
}  
else  
{  
    // Keys changed, delete the old record and insert the new one.  
    if (editedSubscriber.PartitionKey != partitionKey)  
    {  
        // PartitionKey changed, can't do delete/insert in a batch.  
        var deleteOperation = TableOperation.Delete(new Subscriber {  
PartitionKey = partitionKey, RowKey = rowKey, ETag = editedSubscriber.ETag  
});  
  
        await mailingListTable.ExecuteAsync(deleteOperation);  
        var insertOperation = TableOperation.Insert(editedSubscriber);  
        await mailingListTable.ExecuteAsync(insertOperation);  
    }  
    else  
    {  
        // RowKey changed, do delete/insert in a batch.  
        var batchOperation = new TableBatchOperation();  
        var deleteOperation = TableOperation.Delete(new Subscriber {  
PartitionKey = partitionKey, RowKey = rowKey, ETag = editedSubscriber.ETag  
});  
        var insertOperation = TableOperation.Insert(editedSubscriber);  
        batchOperation.Add(deleteOperation, insertOperation);  
        await mailingListTable.ExecuteAsync(batchOperation);  
    }  
}
```

```

        batchOperation.Delete(new Subscriber { PartitionKey =
partitionKey, RowKey = rowKey, ETag = editedSubscriber.ETag });
        batchOperation.Insert(editedSubscriber);
        await mailingListTable.ExecuteBatchAsync(batchOperation);
    }
}

```

The parameters that the MVC model binder passes to the `Edit` method include the original list name and email address values (in the `partitionKey` and `rowKey` parameters) and the values entered by the user (in the `listName` and `emailAddress` parameters):

```

public async Task<ActionResult> Edit(string partitionKey, string
rowKey, string listName, string emailAddress, Subscriber editedSubscriber)

```

The parameters passed to the `UpdateModel` method exclude `PartitionKey` and `RowKey` properties from model binding:

```

var excludeProperties = new string[] { "PartitionKey", "RowKey" };

```

The reason for this is that the `ListName` and `EmailAddress` properties use `PartitionKey` and `RowKey` as their backing properties, and the user might have changed one of these values. When the model binder updates the model by setting the `ListName` property, the `PartitionKey` property is automatically updated. If the model binder were to update the `PartitionKey` property with that property's original value after updating the `ListName` property, it would overwrite the new value that was set by the `ListName` property. The `EmailAddress` property automatically updates the `RowKey` property in the same way.

After updating the `editedSubscriber` model object, the code then determines whether the partition key or row key was changed. If either key value changed, the existing subscriber row has to be deleted and a new one inserted. If only the row key changed, the deletion and insertion can be done in an atomic batch transaction.

Notice that the code creates a new entity to pass in to the `Delete` operation:

```

// RowKey changed, do delete/insert in a batch.
var batchOperation = new TableBatchOperation();
batchOperation.Delete(new Subscriber { PartitionKey =
partitionKey, RowKey = rowKey, ETag = editedSubscriber.ETag });
batchOperation.Insert(editedSubscriber);
await mailingListTable.ExecuteBatchAsync(batchOperation);

```

Entities that you pass in to operations in a batch must be distinct entities. For example, you can't create a `Subscriber` entity, pass it in to a `Delete` operation, then change a value in the same `Subscriber` entity and pass it in to an `Insert` operation. If you did that, the state of the entity after the property change would be in effect for both the `Delete` and the `Insert` operation.

Operations in a batch must all be on the same partition. Because a change to the list name changes the partition key, it can't be done in a transaction.

Add the Subscriber MVC views

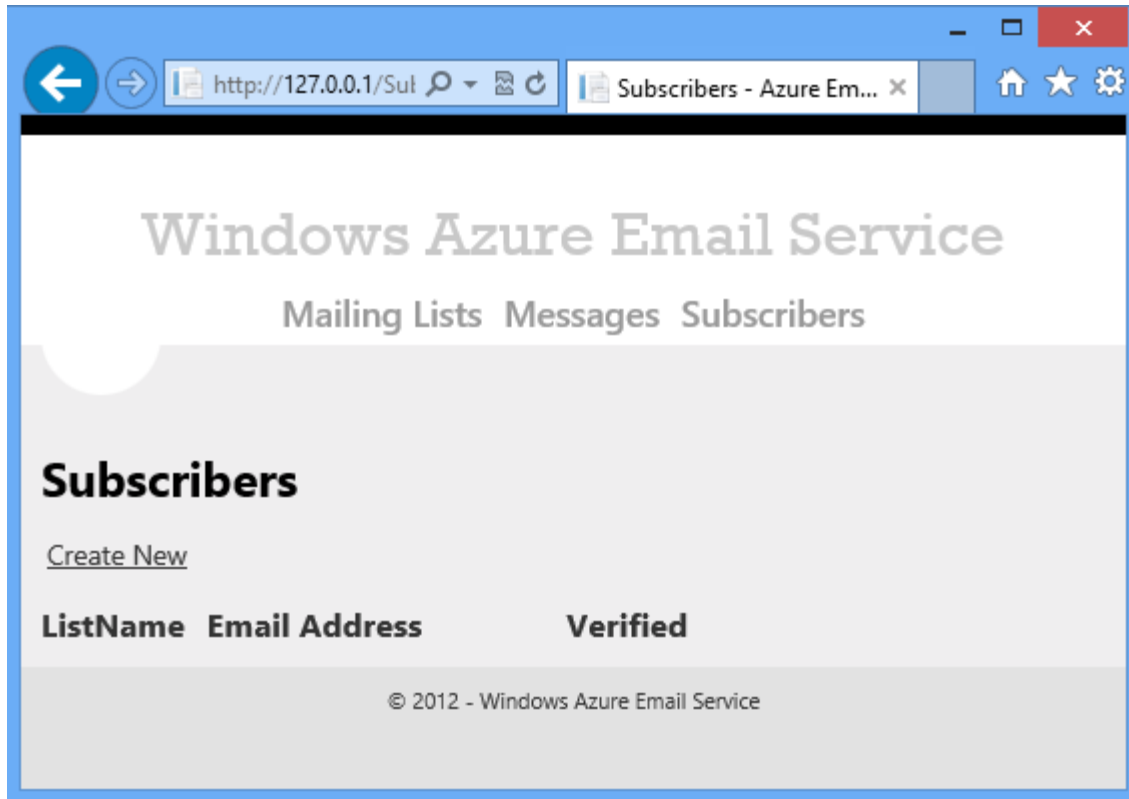
1. In Solution Explorer, create a new folder under the *Views* folder in the MVC project, and name it *Subscriber*.
2. Right-click the new *Views\Subscriber* folder, and choose Add Existing Item.
3. Navigate to the folder where you downloaded the sample application, select all five of the `.cshtml` files in the *Views\Subscriber* folder, and click Add.

In the *Edit.cshtml* file, a hidden field is included for the `SubscriberGUID` value, since that field is not shown and so is not automatically provided in a form field

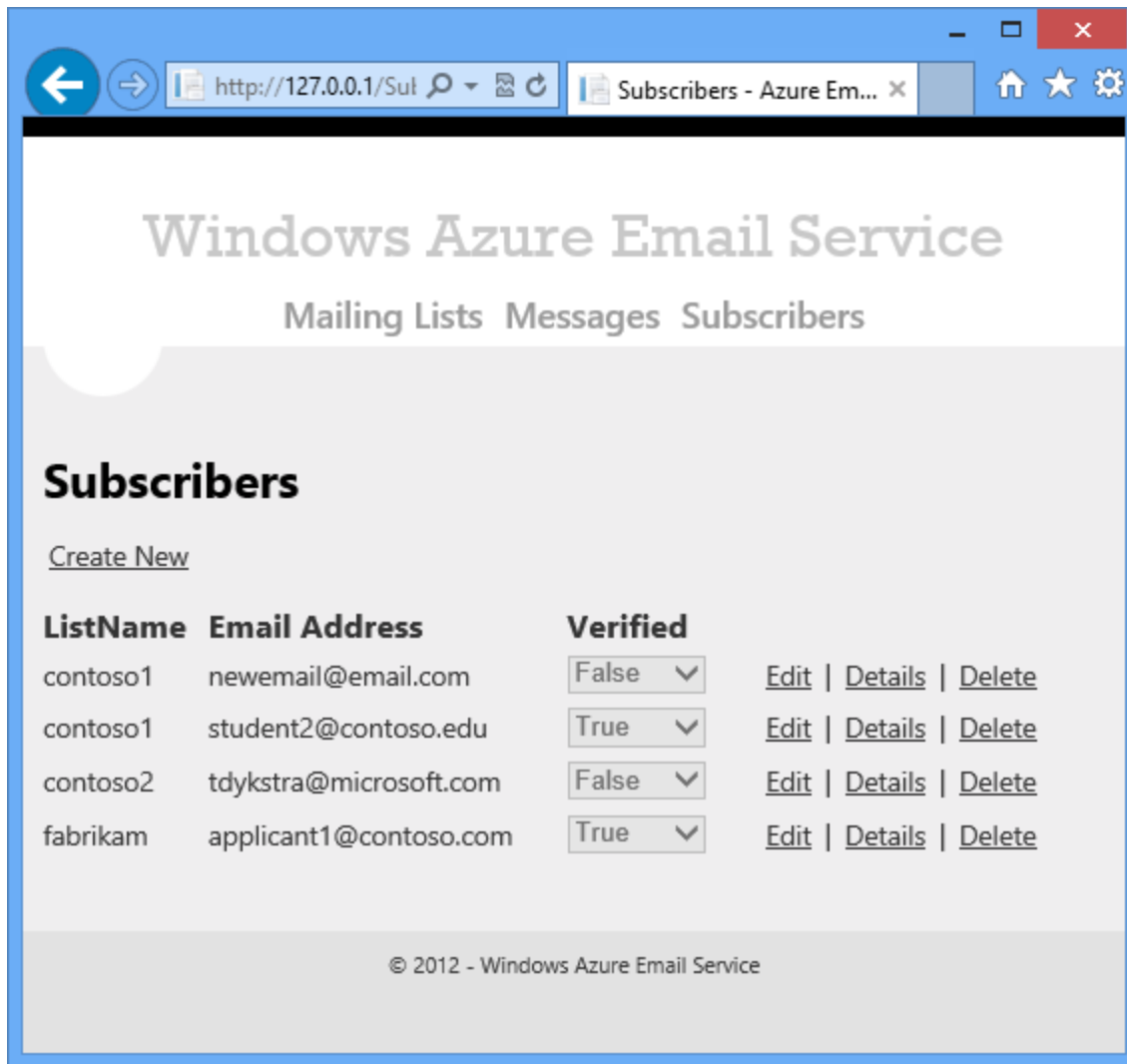
```
@using (Html.BeginForm()) {  
    @Html.AntiForgeryToken()  
    @Html.ValidationSummary(true)  
    @Html.HiddenFor(model => model.SubscriberGUID)  
    @Html.HiddenFor(model => model.ETag)
```

Test the application

1. Run the project by pressing CTRL+F5, and then click Subscribers.



2. Use the Create function to add some mailing lists, and try the Edit and Delete functions to make sure they work.



Create and test the Message controller and views

The Message web UI is used by administrators to create, edit, and display information about messages that are scheduled to be sent to mailing lists.

Add the Message entity class to the Models folder

1. In the **Models** folder in the MVC project, add the **Message.cs** file from the downloaded project.

The Message entity class is used for the rows in the Message table that contain information about a message that is scheduled to be sent to a list. These rows include information such as the subject line, the list to send a message to, and the scheduled date to send it.

```
public class Message : TableEntity
{
```

```

private DateTime? _scheduledDate;
private long _messageRef;

public Message()
{
    this.MessageRef = DateTime.Now.Ticks;
    this.Status = "Pending";
}

[Required]
[Display(Name = "Scheduled Date")]
// DataType.Date shows Date only (not time) and allows easy hook-
up of jQuery DatePicker
[DataType(DataType.Date)]
public DateTime? ScheduledDate
{
    get
    {
        return _scheduledDate;
    }
    set
    {
        _scheduledDate = value;
        this.PartitionKey = value.Value.ToString("yyyy-MM-dd");
    }
}

public long MessageRef
{
    get
    {
        return _messageRef;
    }
    set
    {

```

```

        _messageRef = value;
        this.RowKey = "message" + value.ToString();
    }
}

[Required]
[Display(Name = "List Name")]
public string ListName { get; set; }

[Required]
[Display(Name = "Subject Line")]
public string SubjectLine { get; set; }

// Pending, Queuing, Processing, Complete
public string Status { get; set; }
}

```

The `Message` class defines a default constructor that sets the `MessageRef` property to a unique value for the message. Since this value is part of the row key, the setter for the `MessageRef` property automatically sets the `RowKey` property also. The `MessageRef` property setter concatenates the "message" literal and the `MessageRef` value and puts that in the `RowKey` property.

The `MessageRef` value is created by getting the `Ticks` value from `DateTime.Now`. This ensures that by default when displaying messages in the web UI they will be displayed in the order in which they were created for a given scheduled date (`ScheduledDate` is the partition key). You could use a GUID to make message rows unique, but then the default retrieval order would be random.

The default constructor also sets default status of `Pending` for new message rows.

For more information about the `Message` table structure, see the first tutorial in the series.

Add the Message MVC controller

1. In the **Controllers** folder in the MVC project, add the *MessageController.cs* file from the downloaded project.

Most of the code in this controller is similar to what you saw in the `Subscriber` controller. What is new here is code for working with blobs. For each message, the HTML and plain text content of the email is uploaded in the form of `.htm` and `.txt` files and stored in blobs.

Blobs are stored in blob containers. The Azure Email Service application stores all of its blobs in a single blob container named "azuremailblobcontainer", and code in the controller constructor gets a reference to this blob container:

```
var blobClient = storageAccount.CreateCloudBlobClient();
blobContainer =
blobClient.GetContainerReference("azuremailblobcontainer");
```

For each file that a user selects to upload, the MVC view provides an `HttpPostedFile` object that contains information about the file. When the user creates a new message, the `HttpPostedFile` object is used to save the file to a blob. When the user edits a message, the user can choose to upload a replacement file or leave the blob unchanged.

The controller includes a method that the `HttpPost Create` and `HttpPost Edit` methods call to save a blob:

```
private async Task SaveBlobAsync(string blobName, HttpPostedFileBase
httpPostedFile)
{
    // Retrieve reference to a blob.
    var blob = blobContainer.GetBlockBlobReference(blobName);
    // Create the blob or overwrite the existing blob by uploading a
    local file.
    using (var fileStream = httpPostedFile.InputStream)
    {
        await blob.UploadFromStreamAsync(fileStream);
    }
}
```

The `HttpPost Create` method saves the two blobs and then adds the `Message` table row. Blobs are named by concatenating the `MessageRef` value with the file name extension ".htm" or ".txt".

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create(Message message,
HttpPostedFileBase file, HttpPostedFileBase txtFile)
{
    if (file == null)
```



```

        {
            ModelState.AddModelError(string.Empty, "Please provide an HTML
file path");
        }

        if (txtFile == null)
        {
            ModelState.AddModelError(string.Empty, "Please provide a Text
file path");
        }

        if (ModelState.IsValid)
        {
            await SaveBlobAsync(message.MessageRef + ".htm", file);
            await SaveBlobAsync(message.MessageRef + ".txt", txtFile);

            var insertOperation = TableOperation.Insert(message);
            await messageTable.ExecuteAsync(insertOperation);

            return RedirectToAction("Index");
        }

        var lists = await GetListNamesAsync();
        ViewBag.ListName = new SelectList(lists, "ListName",
"Description");
        return View(message);
    }

```

The `HttpGet Edit` method validates that the retrieved message is in `Pending` status so that the user can't change a message once worker role B has begun processing it. Similar code is in the `HttpPost Edit` method and the `Delete` and `DeleteConfirmed` methods.

```

    if (message.Status != "Pending")
    {
        throw new Exception("Message can't be edited because it isn't in
Pending status.");
    }

```

```
}
```

In the `HttpPost Edit` method, the code saves a new blob only if the user chose to upload a new file.

```
if (httpFile == null)
{
    // They didn't enter a path or navigate to a file, so don't update
the file.
    excludePropLst.Add("HtmlPath");
}
else
{
    // They DID enter a path or navigate to a file, assume it's
changed.
    await SaveBlobAsync(editedMsg.MessageRef + ".htm", httpFile);
}

if (txtFile == null)
{
    excludePropLst.Add("TextPath");
}
else
{
    await SaveBlobAsync(editedMsg.MessageRef + ".txt", txtFile);
}
```

If the scheduled date is changed, the partition key is changed, and a row has to be deleted and inserted. This can't be done in a single transaction because it affects more than one partition.

```
var deleteOperation = TableOperation.Delete(new Message { PartitionKey
= partitionKey, RowKey = rowKey, ETag = editedMsg.ETag });
await messageTable.ExecuteAsync(deleteOperation);
var insertOperation = TableOperation.Insert(editedMsg);
await messageTable.ExecuteAsync(insertOperation);
```

The `HttpPost Delete` method deletes the blobs when it deletes the row in the table:

```
[HttpPost, ActionName("Delete")]
```

```

    public async Task<ActionResult> DeleteConfirmed(String partitionKey,
string rowKey)
    {
        // Get the row again to make sure it's still in Pending status.
        var message = await FindRowAsync(partitionKey, rowKey);
        if (message.Status != "Pending")
        {
            throw new Exception("Message can't be deleted because it isn't
in Pending status.");
        }

        await DeleteBlobAsync(message.MessageRef + ".htm");
        await DeleteBlobAsync(message.MessageRef + ".txt");
        var deleteOperation = TableOperation.Delete(message);
        messageTable.Execute(deleteOperation);
        return RedirectToAction("Index");
    }

    private async Task DeleteBlobAsync(string blobName)
    {
        var blob = blobContainer.GetBlockBlobReference(blobName);
        await blob.DeleteAsync();
    }
}

```

Add the Message MVC views

1. In the MVC project, create a new folder under the *Views* folder, and name it *Message*.
2. In the new *Views\Message* folder, add all five of the *.cshtml* files from the downloaded project.

The `HttpPost Edit` method needs the partition key and row key, so the code in the *Edit.cshtml* file provides these in hidden fields.

```

@using (Html.BeginForm("Edit", "Message", FormMethod.Post, new {
enctype = "multipart/form-data" }))
{
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)
}

```

```

@Html.HiddenFor (model => model.ETag)
<fieldset>
    <legend>Message</legend>
    @Html.HiddenFor (model => model.MessageRef)
    @Html.HiddenFor (model => model.PartitionKey)
    @Html.HiddenFor (model => model.RowKey)

```

The hidden fields were not needed in the Subscriber controller because (a) the `ListName` and `EmailAddress` properties in the Subscriber model update the `PartitionKey` and `RowKey` properties, and (b) the `ListName` and `EmailAddress` properties were included with `EditorFor` helpers in the Edit view. When the MVC model binder for the Subscriber model updates the `ListName` property, the `PartitionKey` property is automatically updated, and when the MVC model binder updates the `EmailAddress` property in the Subscriber model, the `RowKey` property is automatically updated. In the Message model, the fields that map to partition key and row key are not editable fields, so they don't get set that way.

A hidden field is also included for the `MessageRef` property. This is the same value as the partition key, but it is included in order to enable better code clarity in the `HttpPost Edit` method. Including the `MessageRef` hidden field enables the code in the `HttpPost Edit` method to refer to the `MessageRef` value by that name when it constructs file names for the blobs.

The Message Index view in the *Index.cshtml* file is different from the other Index views in that the Edit and Delete links are shown only for messages that are in Pending status:

```

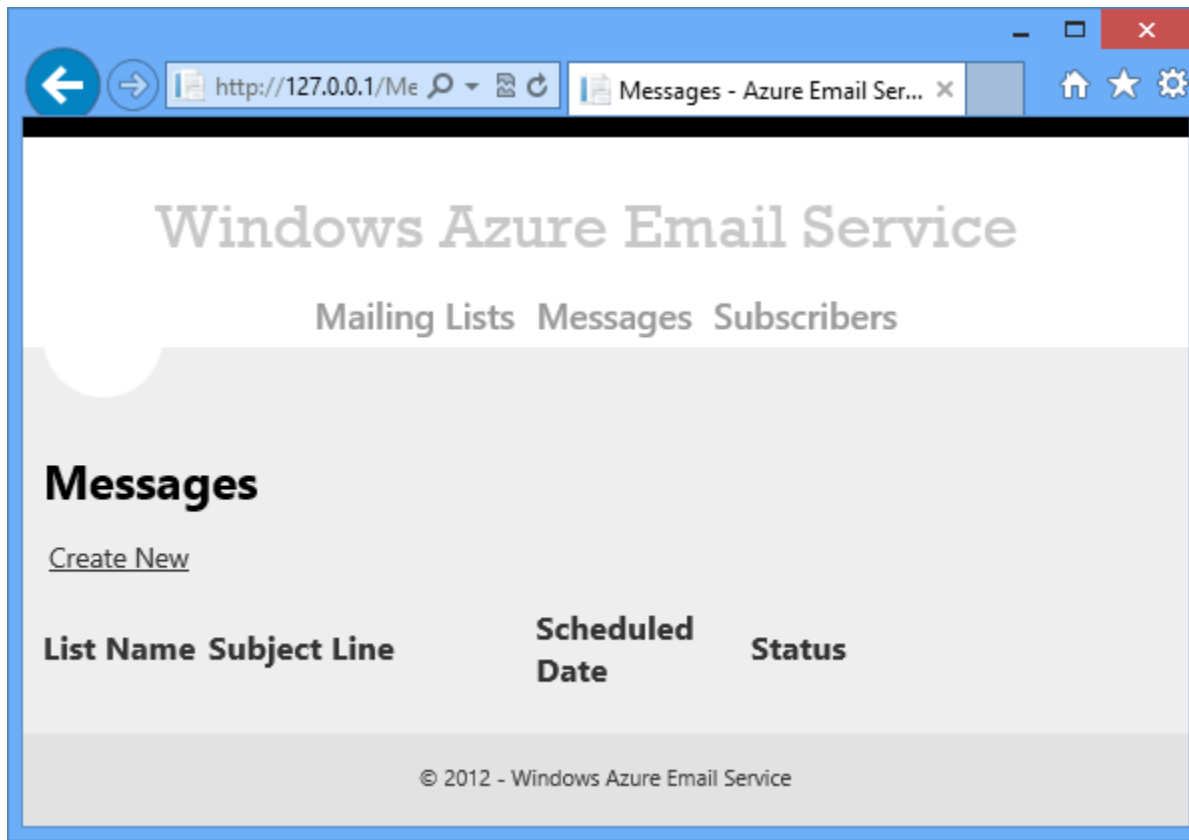
<td>
    @if (item.Status == "Pending")
    {
        @Html.ActionLink("Edit", "Edit", new { PartitionKey =
item.PartitionKey, RowKey = item.RowKey }) @: |
        @Html.ActionLink("Delete", "Delete", new { PartitionKey =
item.PartitionKey, RowKey = item.RowKey }) @: |
    }
    @Html.ActionLink("Details", "Details", new { PartitionKey =
item.PartitionKey, RowKey = item.RowKey })
</td>

```

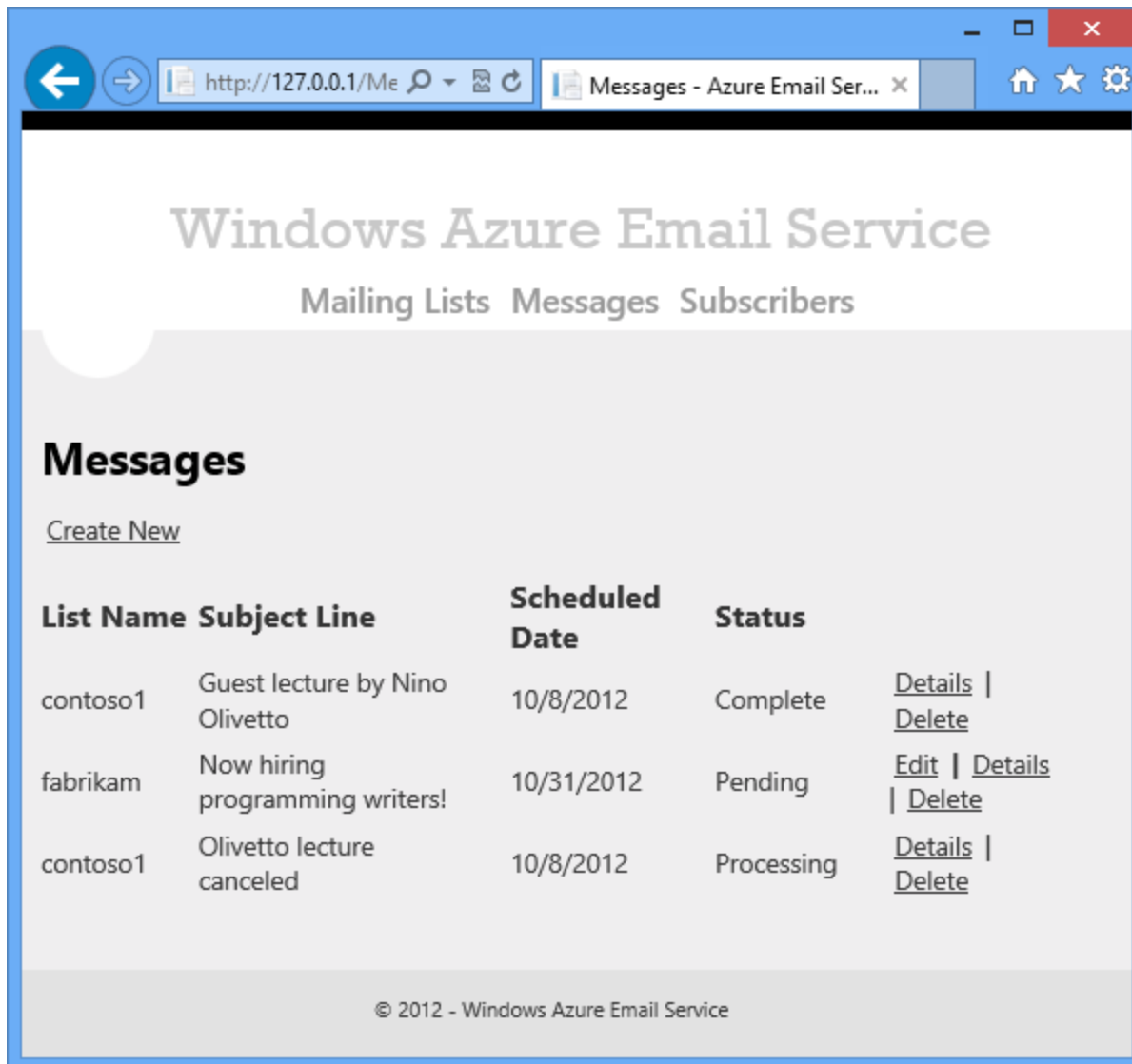
This helps prevent the user from making changes to a message after worker role A has begun to process it.

Test the application

1. Run the project by pressing CTRL+F5, then click Messages.



2. Use the Create function to add some mailing lists, and try the Edit and Delete functions to make sure they work.



Create and test the Unsubscribe controller and view

Next, you'll implement the UI for the unsubscribe process.

This tutorial only includes instructions for building the controller for the unsubscribe process, not the subscribe process. As the first tutorial explained, the UI and service method for the subscription process are included in the download but have been left out of tutorial instructions because the sample application doesn't implement security for the service method. For testing purposes you can use the Subscriber administrator pages to subscribe email addresses to lists.

Add the Unsubscribe view model to the Models folder

1. In the `Models` folder in the MVC project, add the `UnsubscribeVM.cs` file from the downloaded project.

The UnsubscribeVM view model is used to pass data between the Unsubscribe controller and its view.

```
public class UnsubscribeVM
{
    public string EmailAddress { get; set; }
    public string ListName { get; set; }
    public string ListDescription { get; set; }
    public string SubscriberGUID { get; set; }
    public bool? Confirmed { get; set; }
}
```

Unsubscribe links contain the SubscriberGUID. That value is used to get the email address, list name, and list description from the MailingList table. The view displays the email address and the description of the list that is to be unsubscribed from, and it displays a Confirm button that the user must click to complete the unsubscription process.

Add the Unsubscribe controller

1. In the Controllers folder in the MVC project, add the *UnsubscribeController.cs* file from the downloaded project.

This controller has an `HttpGet Index` method that displays the initial unsubscribe page, and an `HttpPost Index` method that processes the Confirm or Cancel button.

The `HttpGet Index` method uses the GUID and list name in the query string to get the MailingList table row for the subscriber. Then it puts all the information needed by the view into the view model and displays the Unsubscribe page. It sets the `Confirmed` property to null in order to tell the view to display the initial version of the Unsubscribe page.

```
public async Task<ActionResult> Index(string id, string listName)
{
    if (string.IsNullOrEmpty(id) == true ||
        string.IsNullOrEmpty(listName))
    {
        ViewBag.errorMessage = "Empty subscriber ID or list name.";
        return View("Error");
    }

    TableRequestOptions reqOptions = new TableRequestOptions()
    {
```

```

        MaximumExecutionTime = TimeSpan.FromSeconds(1.5),
        RetryPolicy = new LinearRetry(TimeSpan.FromSeconds(3), 3)
    };
    string filter = TableQuery.CombineFilters(
        TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, listName),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("SubscriberGUID",
QueryComparisons.Equal, id));
    var query = new TableQuery<Subscriber>().Where(filter);
    TableContinuationToken token = null;
    OperationContext ctx = new OperationContext() { ClientRequestID =
"" };
    TableQuerySegment<Subscriber> currentSegment = null;
    currentSegment = await
mailingListTable.ExecuteQuerySegmentedAsync(query, token, reqOptions,
ctx);

    var subscriber = currentSegment.Results.ToList().Single();

    if (subscriber == null)
    {
        ViewBag.Message = "You are already unsubscribed";
        return View("Message");
    }

    var unsubscribeVM = new UnsubscribeVM();
    unsubscribeVM.EmailAddress = MaskEmail(subscriber.EmailAddress);
    var mailingList = await FindRowAsync(subscriber.ListName,
"mailinglist");
    unsubscribeVM.ListDescription = mailingList.Description;
    unsubscribeVM.SubscriberGUID = id;
    unsubscribeVM.Confirmed = null;
    return View(unsubscribeVM);
}

```


Note: The `SubscriberGUID` is not in the partition key or row key, so the performance of this query will degrade as partition size (the number of email addresses in a mailing list) increases. For information about alternatives to make this query more scalable, see the first tutorial in the series.

The `HttpPost Index` method again uses the `GUID` and list name to get the subscriber information and populates the view model properties. Then, if the `Confirm` button was clicked, it deletes the subscriber row in the `MailingList` table. If the `Confirm` button was pressed it also sets the `Confirm` property to `true`, otherwise it sets the `Confirm` property to `false`. The value of the `Confirm` property is what tells the view to display the confirmed or canceled version of the `Unsubscribe` page.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Index(string subscriberGUID, string
listName, string action)
{
    TableRequestOptions reqOptions = new TableRequestOptions()
    {
        MaximumExecutionTime = TimeSpan.FromSeconds(1.5),
        RetryPolicy = new LinearRetry(TimeSpan.FromSeconds(3), 3)
    };
    string filter = TableQuery.CombineFilters(
        TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, listName),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("SubscriberGUID",
QueryComparisons.Equal, subscriberGUID));
    var query = new TableQuery<Subscriber>().Where(filter);
    TableContinuationToken token = null;
    OperationContext ctx = new OperationContext() { ClientRequestID =
"" };
    TableQuerySegment<Subscriber> currentSegment = null;
    currentSegment = await
mailingListTable.ExecuteQuerySegmentedAsync(query, token, reqOptions,
ctx);
    var subscriber = currentSegment.Results.ToList().Single();

    var unsubscribeVM = new UnsubscribeVM();
```

```

        unsubscribeVM.EmailAddress = MaskEmail(subscriber.EmailAddress);
        var mailingList = await FindRowAsync(subscriber.ListName,
"mailinglist");
        unsubscribeVM.ListDescription = mailingList.Description;
        unsubscribeVM.SubscriberGUID = subscriberGUID;
        unsubscribeVM.Confirmed = false;

        if (action == "Confirm")
        {
            unsubscribeVM.Confirmed = true;
            var deleteOperation = TableOperation.Delete(subscriber);
            mailingListTable.Execute(deleteOperation);
        }

        return View(unsubscribeVM);
    }

```

Create the MVC views

1. In the *Views* folder in the MVC project, create a new folder and name it *Unsubscribe*.
2. In the new *Views\Unsubscribe* folder, add the *Index.cshtml* file from the downloaded project.

In the *Index.cshtml* file, the `Layout = null` line specifies that the `_Layout.cshtml` file should not be used to display this page. The Unsubscribe page displays a simple UI without the headers and footers that are used for the administrator pages.

```

@model MvcWebRole.Models.UnsubscribeVM

@{
    ViewBag.Title = "Unsubscribe";
    Layout = null;
}

```

In the body of the page, the `Confirmed` property determines what will be displayed on the page: Confirm and Cancel buttons if the property is null, unsubscribe-confirmed message if the property is true, unsubscribe-canceled message if the property is false.

```

@using (Html.BeginForm()) {

```

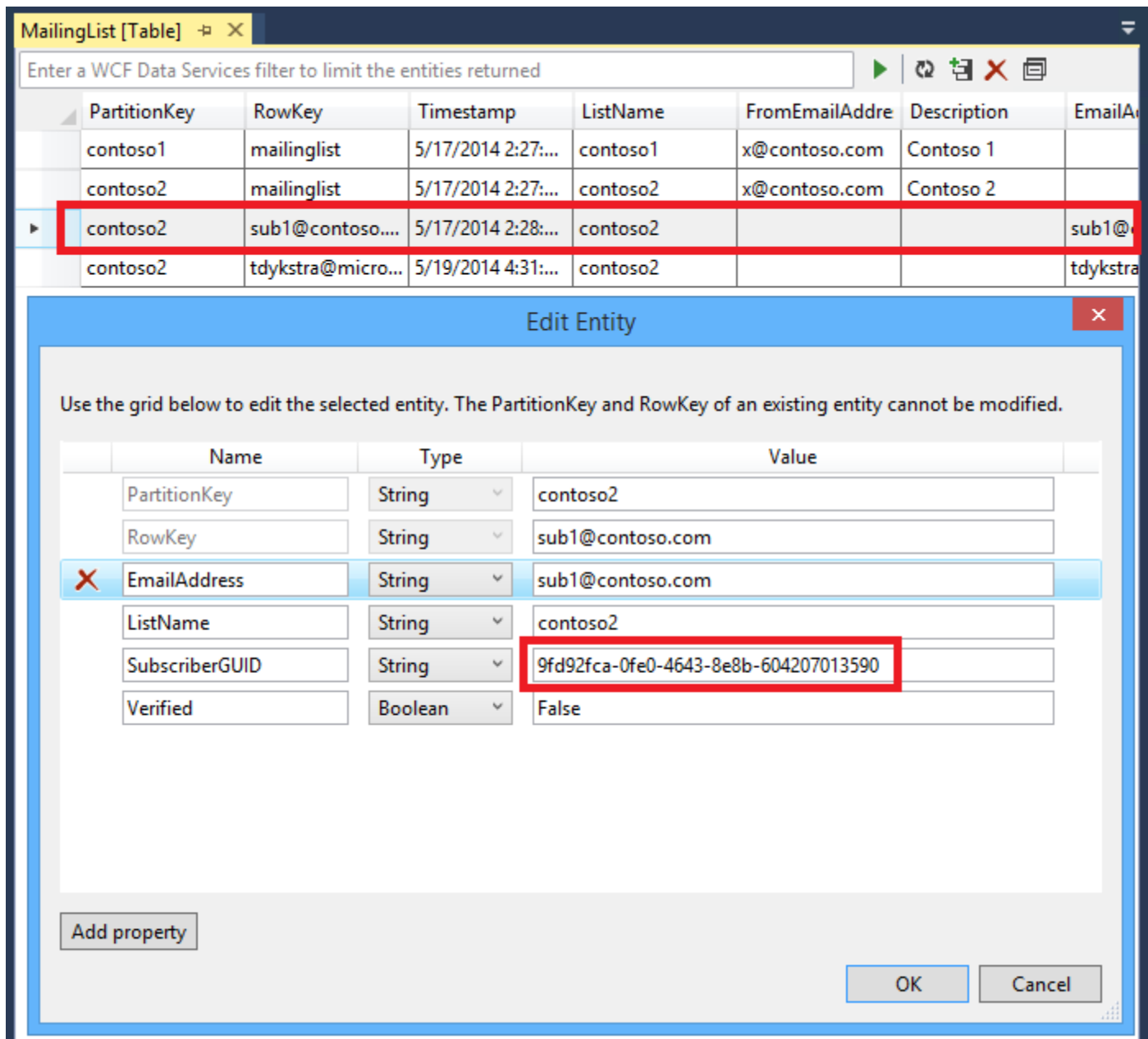
```
@Html.AntiForgeryToken()
@Html.ValidationSummary(true)
<fieldset>
    <legend>Unsubscribe from Mailing List</legend>
    @Html.HiddenFor(model => model.SubscriberGUID)
    @Html.HiddenFor(model => model.EmailAddress)
    @Html.HiddenFor(model => model.ListName)
    @if (Model.Confirmed == null) {
        <p>
            Do you want to unsubscribe @Html.DisplayFor(model =>
model.EmailAddress) from: @Html.DisplayFor(model =>
model.ListDescription)?
        </p>
        <br />
        <p>
            <input type="submit" value="Confirm" name="action"/>
            &nbsp; &nbsp; &nbsp;
            <input type="submit" value="Cancel" name="action"/>
        </p>
    }
    @if (Model.Confirmed == false) {
        <p>
            @Html.DisplayFor(model => model.EmailAddress) will
NOT be unsubscribed from: @Html.DisplayFor(model =>
model.ListDescription).
        </p>
    }
    @if (Model.Confirmed == true) {
        <p>
            @Html.DisplayFor(model => model.EmailAddress) has
been unsubscribed from: @Html.DisplayFor(model => model.ListDescription).
        </p>
    }
</fieldset>
}
```

Test the application

1. Run the project by pressing CTRL-F5, and then click Subscribers.
2. Click Create and create a new subscriber for any mailing list that you created when you were testing earlier.

Leave the browser window open on the Subscribers Index page.

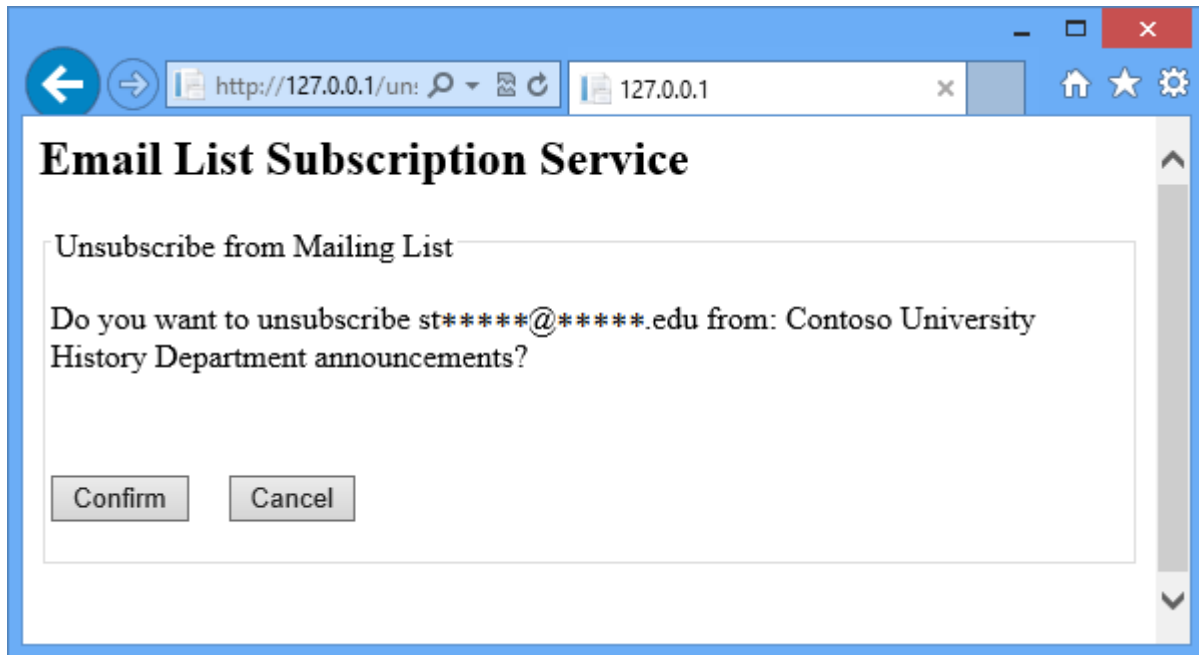
3. Open Server Explorer, and then select the Azure / Storage / (Development) node.
4. Expand Tables, right-click the MailingList table, and then click View Table.
5. Double-click the subscriber row that you added.
6. In the Edit Entity dialog box, select and copy the `SubscriberGUID` value.



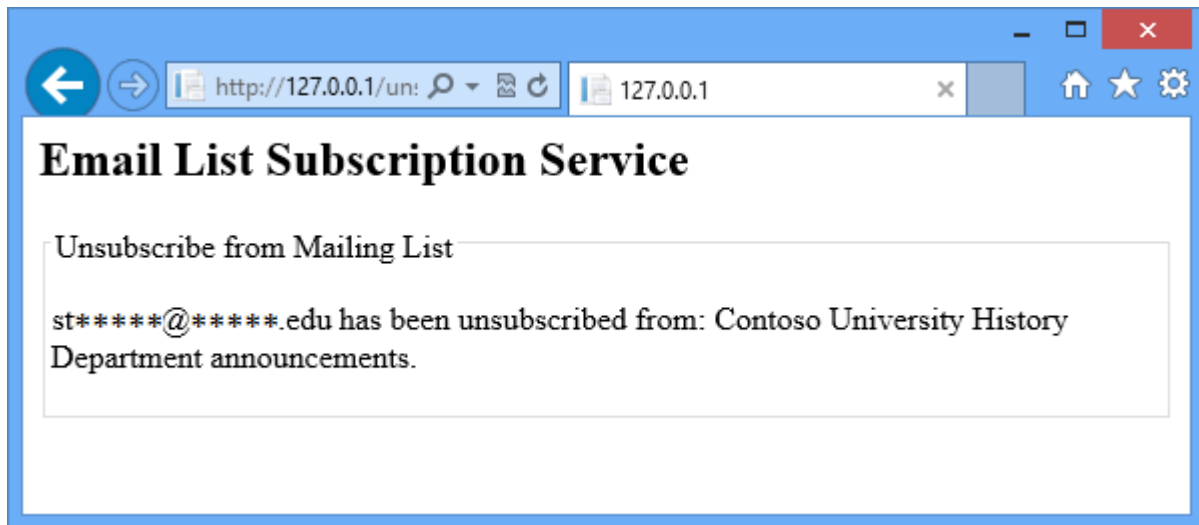
- Switch back to your browser window. In the address bar of the browser, change "Subscriber" in the URL to "unsubscribe?ID=[guidvalue]&listName=[listname]" where [guidvalue] is the GUID that you copied from Azure Storage Explorer, and [listname] is the name of the mailing list. For example:

```
http://127.0.0.1/unsubscribe?ID=b7860242-7c2f-48fb-9d27-d18908ddc9aa&listName=contoso1
```

The version of the Unsubscribe page that asks for confirmation is displayed:



8. Click Confirm and you see confirmation that the email address has been unsubscribed.



9. Go back to the Subscribers Index page to verify that the subscriber row is no longer there.

Next steps

As explained in the first tutorial in the series, this tutorial doesn't show the components of the subscribe process because the ASP.NET Web API service method doesn't implement shared secret security. However, the IP restriction that you set up in the second tutorial also protects the service method and you can add the subscribe functionality by copying the following files from the downloaded project.

For the ASP.NET Web API service method:

Controllers\SubscribeAPI.cs

For the web page that subscribers get when they click on the Confirm link in the email that is generated by the service method:

Models\SubscribeVM.cs

Controllers\SubscribeController.cs

Views\Subscribe\Index.cshtml

In the next tutorial you'll configure and program worker role A, the worker role that schedules emails.

Building worker role A (email scheduler) for the Azure Email Service application - 4 of 5.

This is the fourth tutorial in a series of five that show how to build and deploy the Azure Email Service sample application. For information about the application and the tutorial series, see the first tutorial in the series.

In this tutorial you'll learn:

How to query and update Azure Storage tables.

How to add work items to a queue for processing by another worker role.

How to handle planned shut-downs by overriding the `OnStop` method.

How to handle unplanned shut-downs by making sure that no emails are missed and no duplicate emails are sent.

How to test a worker role that uses Azure Storage tables, by using Server Explorer.

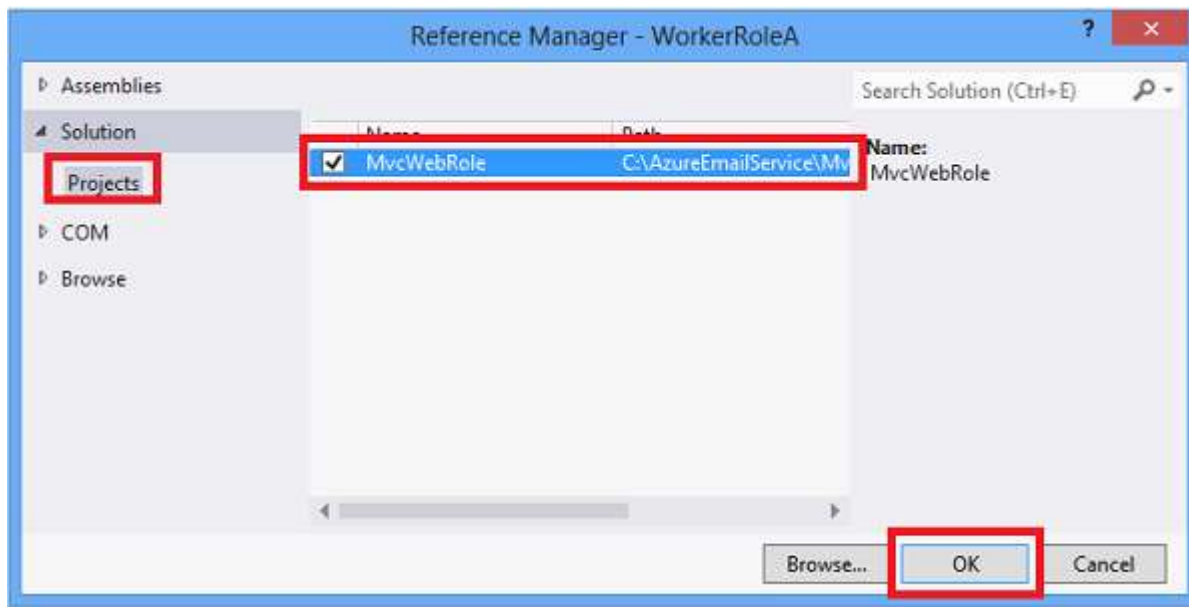
You already created the worker role A project when you created the cloud service project. So all you have to do now is program the worker role.

Add project referenceAdd a reference to the web project

You need a reference to the web project because that is where the entity classes are defined. You'll use the same entity classes in worker role B to read and write data in the Azure tables that the application uses.

Note: In a production application you wouldn't set a reference to a web project from a worker role project, because this results in referencing a number of dependent assemblies that you don't want or need in the worker role. Normally you would keep shared model classes in a class library project, and both web and worker role projects would reference the class library project. To keep the solution structure simple, model classes are stored in the web project for this tutorial.

1. Right-click the WorkerRoleA project, and choose Add, and then choose Reference.
2. In Reference Manager, add a reference to the MvcWebRole project, and then click OK.



Add the SendEmail model

Worker role A creates the `SendEmail` rows in the `Message` table, and worker Role B reads those rows in order to get the information it needs for sending emails. The following image shows a subset of properties for two `Message` rows and three `SendEmail` rows in the `Message` table.

EmailAddress	EmailSent	FromEmailAddress	ListName	SubjectLine	Status
joe@sm.com	False	ab@cd.com	conNBA	NBA draft!	
rick@bc.com	False	ab@cd.com	conNBA	NBA draft!	
sam@bc.edu	False	ab@cd.com	conNBA	NBA draft!	
			conNBA	NBA draft!	Processing
			conNBA	fantacy NBA c	Pending

These rows in the `Message` table serve several purposes:

They provide all of the information that worker role B needs in order to send a single email.

They track whether an email has been sent, in order to prevent duplicates from being sent in case a worker role restarts after a failure.

They make it possible for worker role A to determine when all emails for a message have been sent, so that it can be marked as `Complete`.

For reading and writing the `SendEmail` rows, a model class is required. Since it must be accessible to both worker role A and worker role B, and since all of the other model classes are defined in the web project, it makes sense to define this one in the web project also.

1. In the **Models** folder in the web project, add the **SendEmail.cs** file from the downloaded project.

The code here is similar to the other model classes, except that no `DataAnnotations` attributes are included because there is no UI associated with this model -- it is not used in an MVC controller.

```
public class SendEmail : TableEntity
{
    public long MessageRef { get; set; }
    public string EmailAddress { get; set; }
    public DateTime? ScheduledDate { get; set; }
    public string FromEmailAddress { get; set; }
    public string SubjectLine { get; set; }
    public bool? EmailSent { get; set; }
    public string SubscriberGUID { get; set; }
    public string ListName { get; set; }
}
```

Add worker role code

Add code that runs when the worker role starts

In the `WorkerRoleA` project, the project template created `WorkerRole.cs` with the following code:

```
public class WorkerRole : RoleEntryPoint
{
    public override void Run()
    {
        // This is a sample worker implementation. Replace with your
        logic.

        Trace.WriteLine("WorkerRole1 entry point called",
            "Information");

        while (true)
        {
            Thread.Sleep(10000);
            Trace.WriteLine("Working", "Information");
        }
    }
}
```

```

public override bool OnStart()
{
    // Set the maximum number of concurrent connections
    ServicePointManager.DefaultConnectionLimit = 12;

    // For information on handling configuration changes
    // see the MSDN topic at
http://go.microsoft.com/fwlink/?LinkId=166357.

    return base.OnStart();
}

```

This is the default template code for the worker role. There is an `OnStart` method in which you can put initialization code that runs only when an instance of the worker role starts, and a `Run` method that is called after the `OnStart` method completes. You'll replace this code with your own initialization and run code.

1. Delete *WorkerRole.cs* in the WorkerRoleA project, and then add the *WorkerRoleA.cs* file from the downloaded project.

The OnStart method

The `OnStart` method initializes the context objects that you need in order to work with Azure Storage entities. It also makes sure that all of the tables, queues, and blob containers that you'll be using in the `Run` method exist. The code that performs these tasks is similar to what you saw earlier in the MVC controller constructors. You'll configure the connection string that this method uses later.

```

public override bool OnStart()
{
    ServicePointManager.DefaultConnectionLimit =
Environment.ProcessorCount * 12;

    ConfigureDiagnostics();
    Trace.TraceInformation("Initializing storage account in WorkerA");
    var storageAccount =
CloudStorageAccount.Parse(RoleEnvironment.GetConfigurationSettingValue("StorageConnectionString"));

```

```

        CloudQueueClient queueClient =
storageAccount.CreateCloudQueueClient();
        sendEmailQueue = queueClient.GetQueueReference("azuremailqueue");
        var tableClient = storageAccount.CreateCloudTableClient();
        mailingListTable = tableClient.GetTableReference("mailinglist");
        messageTable = tableClient.GetTableReference("message");
        messagearchiveTable =
tableClient.GetTableReference("messagearchive");

        // Create if not exists for queue, blob container, SentEmail
table.

        sendEmailQueue.CreateIfNotExists();
        messageTable.CreateIfNotExists();
        mailingListTable.CreateIfNotExists();
        messagearchiveTable.CreateIfNotExists();

        return base.OnStart();
    }

```

You may have seen earlier documentation on working with Azure Storage that shows the initialization code in a loop that checks for transport errors. This is no longer necessary because the API now has a built-in retry mechanism that absorbs transient network failures for up to 3 additional attempts.

The `ConfigureDiagnostics` method that the `OnStart` method calls sets up tracing so that you will be able to see the output from `Trace.Information` and `Trace.Error` methods. This method is explained in [the second tutorial](#).

The [ServicePointManager.DefaultConnectionLimit](#) setting specifies the maximum number of concurrent TCP connections that can be opened in .NET. The documentation for the [ServicePointManager](#) class explains that each unique host that the worker role instance connects to is a separate connection. For example, this worker role would have three concurrent connections, one for tables, one for blobs, and one for queues. For some connections, such as for SQL Server, client software does connection pooling, which can reduce the number of concurrent connections managed by `ServicePointManager`. The best number for `DefaultConnectionLimit` depends in part on the backend service you are connecting to. For some services it might be fine to have 500 open connections, another service might be overwhelmed with only 5

connections. The limit of 12 connections for each processor is a general guideline that will work well in many scenarios.

The OnStop method

The `OnStop` method sets the global variable `onStopCalled` to `true`, then it waits for the `Run` method to set the global variable `returnedFromRunMethod` to `true`, which signals it is ready to do a clean shutdown.

```
public override void OnStop()
{
    onStopCalled = true;
    while (returnedFromRunMethod == false)
    {
        System.Threading.Thread.Sleep(1000);
    }
}
```

The `OnStop` method is called when the worker role is shutting down for one of the following reasons:

Azure needs to reboot the virtual machine (the web role or worker role instance) or the physical machine that hosts the virtual machine.

You stopped your cloud service by using the Stop button on the Azure Management Portal.

You deployed an update to your cloud service project.

The `Run` method monitors the variable `onStopCalled` and stops pulling any new work items to process when that variable changes to `true`. This coordination between the `OnStop` and `Run` methods enables a graceful shutdown of the worker process.

Azure periodically installs operating system updates in order to ensure that the platform is secure, reliable, and performs well. These updates typically require the machines that host your cloud service to shut down and reboot. For more information, see [Role Instance Restarts Due to OS Upgrades](#).

The Run method

The `Run` method performs two functions:

Scans the `message` table looking for messages scheduled to be sent today or earlier, for which queue work items haven't been created yet.

Scans the message table looking for messages that have a status indicating that all of the queue work items were created but not all of the emails have been sent yet. If it finds one, it scans SendEmail rows for that message to see if all emails were sent, and if they were, it updates the status to Completed and archives the message row.

The method also checks the global variable onStopCalled. When the variable is true, the method stops pulling new work items to process, and it returns when already-started tasks are completed.

```
public override void Run()
{
    Trace.TraceInformation("WorkerRoleA entering Run()");
    while (true)
    {
        try
        {
            var tomorrow = DateTime.Today.AddDays(1.0).ToString("yyyy-MM-dd");

            // If OnStop has been called, return to do a graceful shutdown.
            if (onStopCalled == true)
            {
                Trace.TraceInformation("onStopCalled WorkerRoleB");
                returnedFromRunMethod = true;
                return;
            }

            // Retrieve all messages that are scheduled for tomorrow or earlier
            // and are in Pending or Queuing status.
            string typeAndDateFilter = TableQuery.CombineFilters(
                TableQuery.GenerateFilterCondition("RowKey",
                QueryComparisons.GreaterThan, "message"),
                TableOperators.And,
                TableQuery.GenerateFilterCondition("PartitionKey",
                QueryComparisons.LessThan, tomorrow));

            var query = (new
            TableQuery<Message>().Where(typeAndDateFilter));
```

```

        var messagesToProcess =
messageTable.ExecuteQuery(query).ToList();
        TableOperation replaceOperation;
        // Process each message (queue emails to be sent).
        foreach (Message messageToProcess in messagesToProcess)
        {
            string restartFlag = "0";
            // If the message is already in Queuing status,
            // set flag to indicate this is a restart.
            if (messageToProcess.Status == "Queuing")
            {
                restartFlag = "1";
            }

            // If the message is in Pending status, change
            // it to Queuing.
            if (messageToProcess.Status == "Pending")
            {
                messageToProcess.Status = "Queuing";
                replaceOperation =
TableOperation.Replace(messageToProcess);
                messageTable.Execute(replaceOperation);
            }

            // If the message is in Queuing status,
            // process it and change it to Processing status;
            // otherwise it's already in processing status, and
            // in that case check if processing is complete.
            if (messageToProcess.Status == "Queuing")
            {
                ProcessMessage(messageToProcess, restartFlag);

                messageToProcess.Status = "Processing";
                replaceOperation =
TableOperation.Replace(messageToProcess);

```

```

        messageTable.Execute(replaceOperation);
    }
    else
    {
        CheckAndArchiveIfComplete(messageToProcess);
    }
}

// Sleep for one minute to minimize query costs.
System.Threading.Thread.Sleep(1000 * 60);
}
catch (Exception ex)
{
    string err = ex.Message;
    if (ex.InnerException != null)
    {
        err += " Inner Exception: " +
ex.InnerException.Message;
    }
    Trace.TraceError(err);
    // Don't fill up Trace storage if we have a bug in queue
process loop.
    System.Threading.Thread.Sleep(1000 * 60);
}
}
}

```

Notice that all of the work is done in an infinite loop in a `while` block, and all of the code in the `while` block is wrapped in a `try-catch` block to prevent an unhandled exception. If an unhandled exception occurs, Azure will raise the [UnhandledException](#) event, the worker process is terminated, and the role is taken offline. The worker role will be restarted by Azure, but this takes several minutes. The `try` block calls `TraceError` to record the error and then sleeps for 60 seconds so that if the error is persistent the error message won't be repeated too many times. In a production application you might send an email to an administrator in the `try` block.

The Run method processes a query for message rows in the message table that have scheduled date before tomorrow:

```
        // Retrieve all messages that are scheduled for tomorrow
or earlier

        // and are in Pending or Queuing status.
        string typeAndDateFilter = TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition("RowKey",
QueryComparisons.GreaterThan, "message"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.LessThan, tomorrow));

        var query = (new
TableQuery<Message>()).Where(typeAndDateFilter));

        var messagesToProcess =
messageTable.ExecuteQuery(query).ToList();
```

Note: One of the benefits of moving message rows to the messagearchive table after they are processed is that this query only needs to specify PartitionKey and RowKey as search criteria. If we did not archive processed rows, the query would also have to specify a non-key field (Status) and would have to search through more rows. The table size would increase, and the query would take longer and could start getting continuation tokens.

If a message is in Pending status, processing has not yet begun; if it is in Queuing status, processing did begin earlier but was interrupted before all queue messages were created. In that case an additional check has to be done in worker role B when it is sending each email to make sure the email hasn't already been sent. That is the purpose of the restartFlag variable.

```
        string restartFlag = "0";
        if (messageToProcess.Status == "Queuing")
        {
            restartFlag = "1";
        }
```

Next, the code sets message rows that are in Pending status to Queuing. Then, for those rows plus any that were already in Queuing status, it calls the ProcessMessage method to create the queue work items to send emails for the message.

```

        if (messageToProcess.Status == "Pending")
        {
            messageToProcess.Status = "Queuing";
            replaceOperation =
TableOperation.Replace(messageToProcess);
            messageTable.Execute(replaceOperation);
        }

        if (messageToProcess.Status == "Queuing")
        {
            ProcessMessage(messageToProcess, restartFlag);

            messageToProcess.Status = "Processing";
            replaceOperation =
TableOperation.Replace(messageToProcess);
            messageTable.Execute(replaceOperation);
        }
        else
        {
            CheckAndArchiveIfComplete(messageToProcess);
        }
    }

```

After processing a message in Queuing status the code sets the Message row status to Processing. Rows in the message table that are not in Pending or Queuing status are already in Processing status, and for those rows the code calls a method that checks if all of the emails for the message were sent. If all emails have been sent, the message row is archived.

After processing all records retrieved by the query, the code sleeps for one minute.

```

// Sleep for one minute to minimize query costs.
System.Threading.Thread.Sleep(1000*60);

```

There is a minimal charge for every Azure Storage query, even if it doesn't return any data, so continuously re-scanning would unnecessarily add to your Azure expenses. As this tutorial is being written, the cost is \$0.10 per million transactions (a query counts as a transaction), so the sleep time could be made much less than a minute and the cost of scanning the tables for messages to be sent would still be minimal. For more information about pricing, see the first tutorial.

Note on threading and optimal CPU utilization: There are two tasks in the `Run` method (queuing emails and checking for completed messages), and they run sequentially in a single thread. A small virtual machine (VM) has 1.75 GB RAM and only one CPU, so it's probably OK to run these tasks sequentially with a single thread. Suppose your application needed more memory than the small VM provided to run efficiently. A medium VM provides 3.5 GB RAM and 2 CPU's, but this application would only use one CPU, because it's single threaded. To take advantage of all the CPUs, you would need to create a worker thread for each CPU. Even so, a single CPU is not fully utilized by one thread. When a thread makes network or I/O calls, the thread must wait for the I/O or network call to complete, and while it waits, it's not doing useful work. If the `Run` method was implemented using two threads, when one thread was waiting for a network or I/O operation to complete, the other thread could be doing useful work.

The `ProcessMessage` method

The `ProcessMessage` method gets all of the email addresses for the destination email list, and creates a queue work item for each email address. As it creates queue work items, it also creates `SendEmail` rows in the `Message` table. These rows provide worker role B with the information it needs to send emails and includes an `EmailSent` property that tracks whether each email has been sent.

```
private void ProcessMessage(Message messageToProcess, string
restartFlag)
{
    // Get Mailing List info to get the "From" email address.
    var retrieveOperation =
TableOperation.Retrieve<MailingList>(messageToProcess.ListName,
"mailinglist");
    var retrievedResult = mailingListTable.Execute(retrieveOperation);
    var mailingList = retrievedResult.Result as MailingList;
    if (mailingList == null)
    {
        Trace.TraceError("Mailing list not found: " +
messageToProcess.ListName + " for message: " +
messageToProcess.MessageRef);
        return;
    }
    // Get email addresses for this Mailing List.
    string filter = TableQuery.CombineFilters(
```

```

        TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, messageToProcess.ListName),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("RowKey",
QueryComparisons.NotEqual, "mailinglist"));
    var query = new TableQuery<Subscriber>().Where(filter);
    var subscribers = mailingListTable.ExecuteQuery(query).ToList();

    foreach (Subscriber subscriber in subscribers)
    {
        // Verify that the subscriber email address has been verified.
        if (subscriber.Verified == false)
        {
            Trace.TraceInformation("Subscriber " +
subscriber.EmailAddress + " not Verified, so not queuing ");
            continue;
        }

        // Create a SendEmail entity for this email.
        var sendEmailRow = new SendEmail
        {
            PartitionKey = messageToProcess.PartitionKey,
            RowKey = messageToProcess.MessageRef.ToString() +
subscriber.EmailAddress,
            EmailAddress = subscriber.EmailAddress,
            EmailSent = false,
            MessageRef = messageToProcess.MessageRef,
            ScheduledDate = messageToProcess.ScheduledDate,
            FromEmailAddress = mailingList.FromEmailAddress,
            SubjectLine = messageToProcess.SubjectLine,
            SubscriberGUID = subscriber.SubscriberGUID,
            ListName = mailingList.ListName
        };

        // When we try to add the entity to the SendEmail table,

```

```

        // an exception might happen if this worker role went
        // down after processing some of the email addresses and then
restarted.

        // In that case the row might already be present, so we do an
Upsert operation.
        try
        {
            var upsertOperation =
TableOperation.InsertOrReplace(sendEmailRow);
            messageTable.Execute(upsertOperation);
        }
        catch (Exception ex)
        {
            string err = "Error creating SendEmail row: " +
ex.Message;

            if (ex.InnerException != null)
            {
                err += " Inner Exception: " + ex.InnerException;
            }
            Trace.TraceError(err);
        }

        // Create the queue message.
        string queueMessageString =
            sendEmailRow.PartitionKey + "," +
            sendEmailRow.RowKey + "," +
            restartFlag;
        var queueMessage = new CloudQueueMessage(queueMessageString);
        sendEmailQueue.AddMessage(queueMessage);
    }

    Trace.TraceInformation("ProcessMessage end PK: "
        + messageToProcess.PartitionKey);
}

```

The code first gets the mailing list row from the mailinglist table for the destination mailing list. This row has the "from" email address which needs to be provided to worker role B for sending emails.

```
// Get Mailing List info to get the "From" email address.
var retrieveOperation =
TableOperation.Retrieve<MailingList>(messageToProcess.ListName,
"mailinglist");
var retrievedResult = mailingListTable.Execute(retrieveOperation);
var mailingList = retrievedResult.Result as MailingList;
if (mailingList == null)
{
    Trace.TraceError("Mailing list not found: " +
messageToProcess.ListName + " for message: " +
messageToProcess.MessageRef);
    return;
}
```

Then it queries the mailinglist table for all of the subscriber rows for the destination mailing list.

```
// Get email addresses for this Mailing List.
string filter = TableQuery.CombineFilters(
    TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, messageToProcess.ListName),
    TableOperators.And,
    TableQuery.GenerateFilterCondition("RowKey",
QueryComparisons.NotEqual, "mailinglist"));
var query = new TableQuery<Subscriber>().Where(filter);
var subscribers = mailingListTable.ExecuteQuery(query).ToList();
```

In the loop that processes the query results, the code begins by checking if subscriber email address is verified, and if not no email is queued.

```
// Verify that the subscriber email address has been verified.
if (subscriber.Verified == false)
{
    Trace.TraceInformation("Subscriber " +
subscriber.EmailAddress + " not Verified, so not queuing ");
    continue;
}
```

```
}
```

Next, the code creates a `SendEmail` row in the message table. This row contains the information that worker role B will use to send an email. The row is created with the `EmailSent` property set to `false`.

```
// Create a SendEmail entity for this email.
var sendEmailRow = new SendEmail
{
    PartitionKey = messageToProcess.PartitionKey,
    RowKey = messageToProcess.MessageRef.ToString() +
subscriber.EmailAddress,
    EmailAddress = subscriber.EmailAddress,
    EmailSent = false,
    MessageRef = messageToProcess.MessageRef,
    ScheduledDate = messageToProcess.ScheduledDate,
    FromEmailAddress = mailingList.FromEmailAddress,
    SubjectLine = messageToProcess.SubjectLine,
    SubscriberGUID = subscriber.SubscriberGUID,
    ListName = mailingList.ListName
};
try
{
    var upsertOperation =
TableOperation.InsertOrReplace(sendEmailRow);
    messageTable.Execute(upsertOperation);
}
catch (Exception ex)
{
    string err = "Error creating SendEmail row: " +
ex.Message;

    if (ex.InnerException != null)
    {
        err += " Inner Exception: " + ex.InnerException;
    }

    Trace.TraceError(err);
}
```

The code uses an "upsert" operation because the row might already exist if worker role A is restarting after a failure.

The last task to be done for each email address is to create the queue work item that will trigger worker role B to send an email. The queue work item contains the partition key and row key value of the `SendEmail` row that was just created, plus the restart flag that was set earlier. The `SendEmail` row contains all of the information that worker role B needs in order to send an email.

```
// Create the queue message.
string queueMessageString =
    sendEmailRow.PartitionKey + "," +
    sendEmailRow.RowKey + "," +
    restartFlag;
var queueMessage = new CloudQueueMessage(queueMessageString);
sendEmailQueue.AddMessage(queueMessage);
```

The CheckAndUpdateStatusIfComplete method

The `CheckAndUpdateStatusIfComplete` method checks messages that are in `Processing` status to see if all emails have been sent. If it finds no unsent emails, it updates the row status to `Completed` and archives the row.

```
private void CheckAndArchiveIfComplete(Message messageToCheck)
{
    // Get the list of emails to be sent for this message: all
    SendEmail rows
    // for this message.
    string pkrkFilter = TableQuery.CombineFilters(
        TableQuery.GenerateFilterCondition("PartitionKey",
        QueryComparisons.Equal, messageToCheck.PartitionKey),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("RowKey",
        QueryComparisons.LessThan, "message"));
    var query = new TableQuery<SendEmail>().Where(pkrkFilter);
    var emailToBeSent =
    messageTable.ExecuteQuery(query).FirstOrDefault();

    if (emailToBeSent != null)
```



```

    {
        return;
    }

    // All emails have been sent; copy the message row to the archive
    table.

    // Insert the message row in the messagearchive table
    var messageToDelete = new Message { PartitionKey =
messageToCheck.PartitionKey, RowKey = messageToCheck.RowKey, ETag = "*" };
    messageToCheck.Status = "Complete";
    var insertOrReplaceOperation =
TableOperation.InsertOrReplace(messageToCheck);
    messagearchiveTable.Execute(insertOrReplaceOperation);

    // Delete the message row from the message table.
    var deleteOperation = TableOperation.Delete(messageToDelete);
    messageTable.Execute(deleteOperation);
}

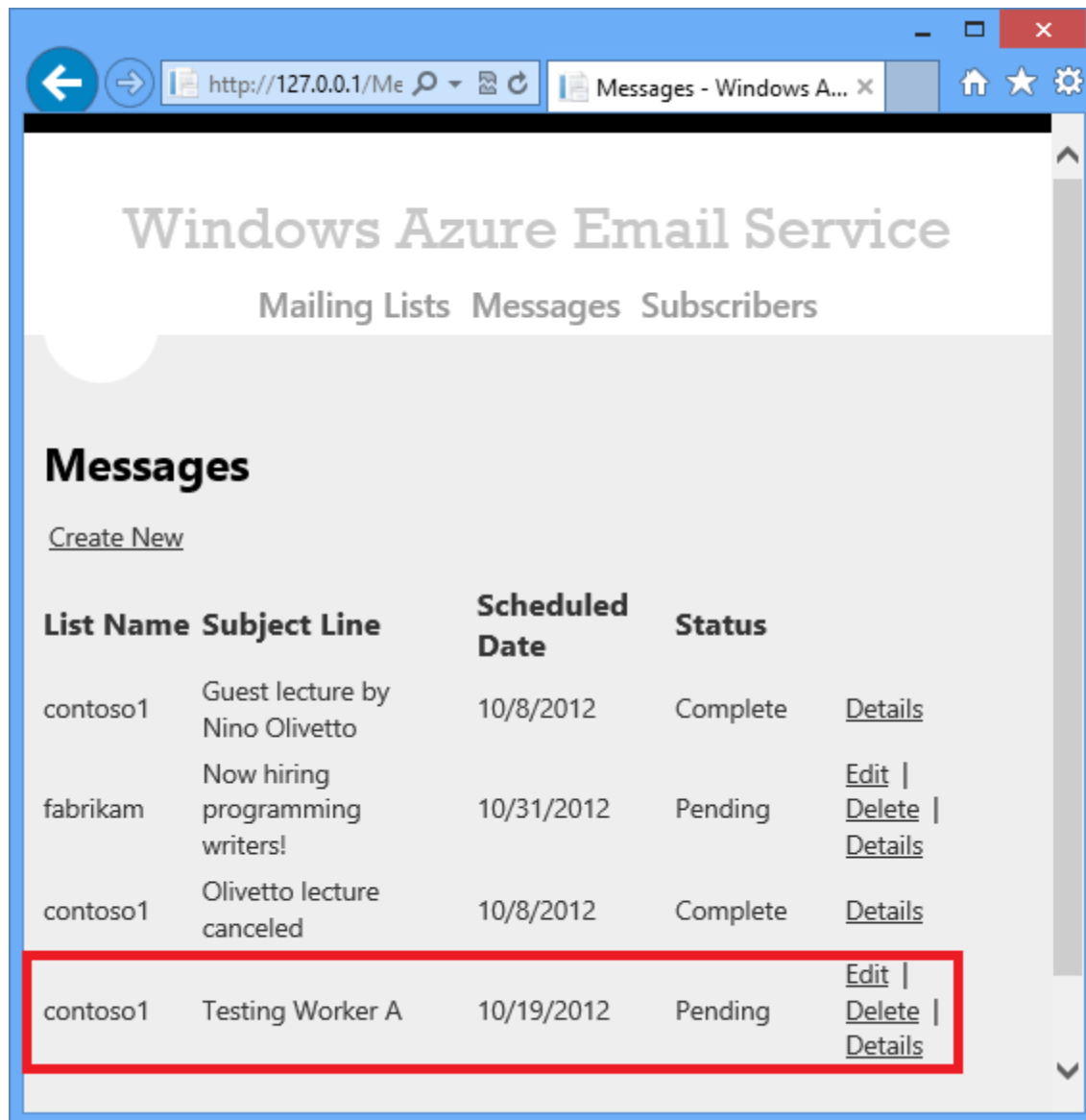
```

Test worker role A

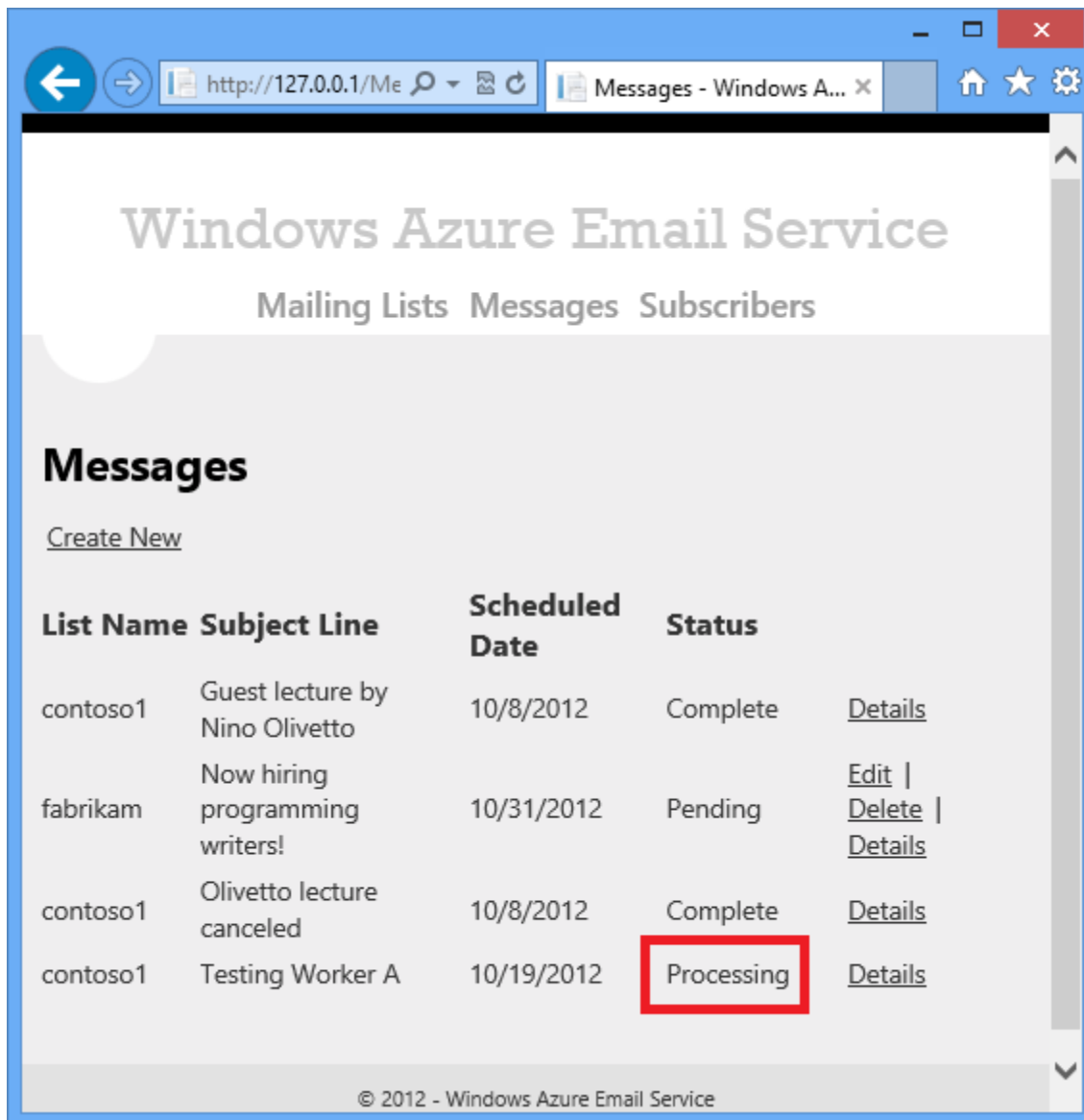
1. Run the application by pressing F5.
2. Use the administrator web pages to create a mailing list and create subscribers to the mailing list. Set the `Verified` property to `true` for at least one of the subscribers, and set the email address to an address that you can receive mail at.

No emails will be sent until you implement worker role B, but you'll use the same test data for testing worker role B.

3. Create a message to be sent to the mailing list you created, and set the scheduled date to today or a date in the past.

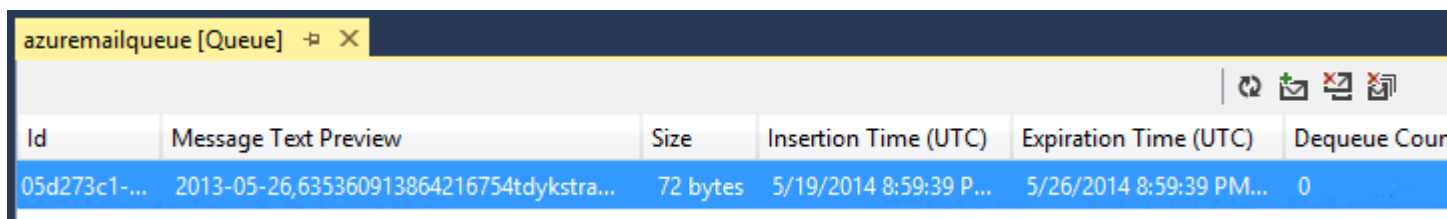


4. In a little over a minute (because of the one minute sleep time in the Run method), refresh the Messages web page and you see the status change to Processing. (You might see it change to Queuing first, but chances are it will go from Queuing to Processing so quickly that you won't see Queuing.)



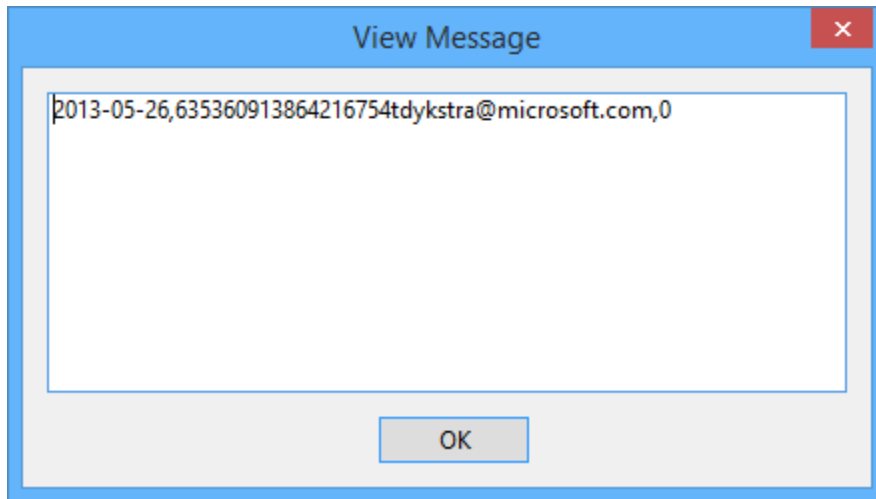
- Open Server Explorer and open the node for Development Storage.
- Expand Queues, right-click azuremailqueue, and then click View Queue.

You see one queue message for each verified subscriber in your destination email list.



- Double-click a queue message.

You see the contents of the queue message: the partition key (the date), the row key (the MessageRef value and the email address), and the restart flag, delimited by a comma.



8. Close the View Message dialog box.
9. Expand the Tables node, right-click the Message table, and then click View Table.

You see the message you scheduled, with "message" in the row key, followed by a row for each verified subscriber, with the email address in the row key.

10. Double-click a row that has "message" in the row key, to see the contents of the row that the web role created.

Edit Entity

Use the grid below to edit the selected entity. The PartitionKey and RowKey of an existing entity cannot be modified.

Name	Type	Value
PartitionKey	String	2013-05-26
RowKey	String	message635360913864216754
ListName	String	contoso2
MessageRef	Int64	635360913864216754
ScheduledDate	DateTime	5/26/2013 12:00:00 AM
Status	String	Processing
SubjectLine	String	azure email service test

Add property

OK

Cancel

- Double-click a row that has an email address in the row key, to see the contents of the `SendEmail` row that worker role A created.

Edit Entity

Use the grid below to edit the selected entity. The PartitionKey and RowKey of an existing entity cannot be modified.

Name	Type	Value
PartitionKey	String	2013-05-26
RowKey	String	635360913864216754td@microsoft.com
EmailAddress	String	tdykstra@microsoft.com
EmailSent	Boolean	False
FromEmailAddress	String	x@contoso.com
ListName	String	contoso2
MessageRef	Int64	635360913864216754
ScheduledDate	DateTime	5/26/2013 12:00:00 AM
SubjectLine	String	azure email service test

Add property

OK

Cancel

Next steps

You have now built worker role A and verified that it creates the queue messages and table rows that worker role B needs in order to send emails. In the [next tutorial](#), you'll build and test worker role B.

Building worker role B (email sender) for the Azure Email Service application - 5 of 5.

This is the fifth tutorial in a series of five that show how to build and deploy the Azure Email Service sample application. For information about the application and the tutorial series, see the first tutorial in the series.

In this tutorial you'll learn:

How to add a worker role to a cloud service project.

How to poll a queue and process work items from the queue.

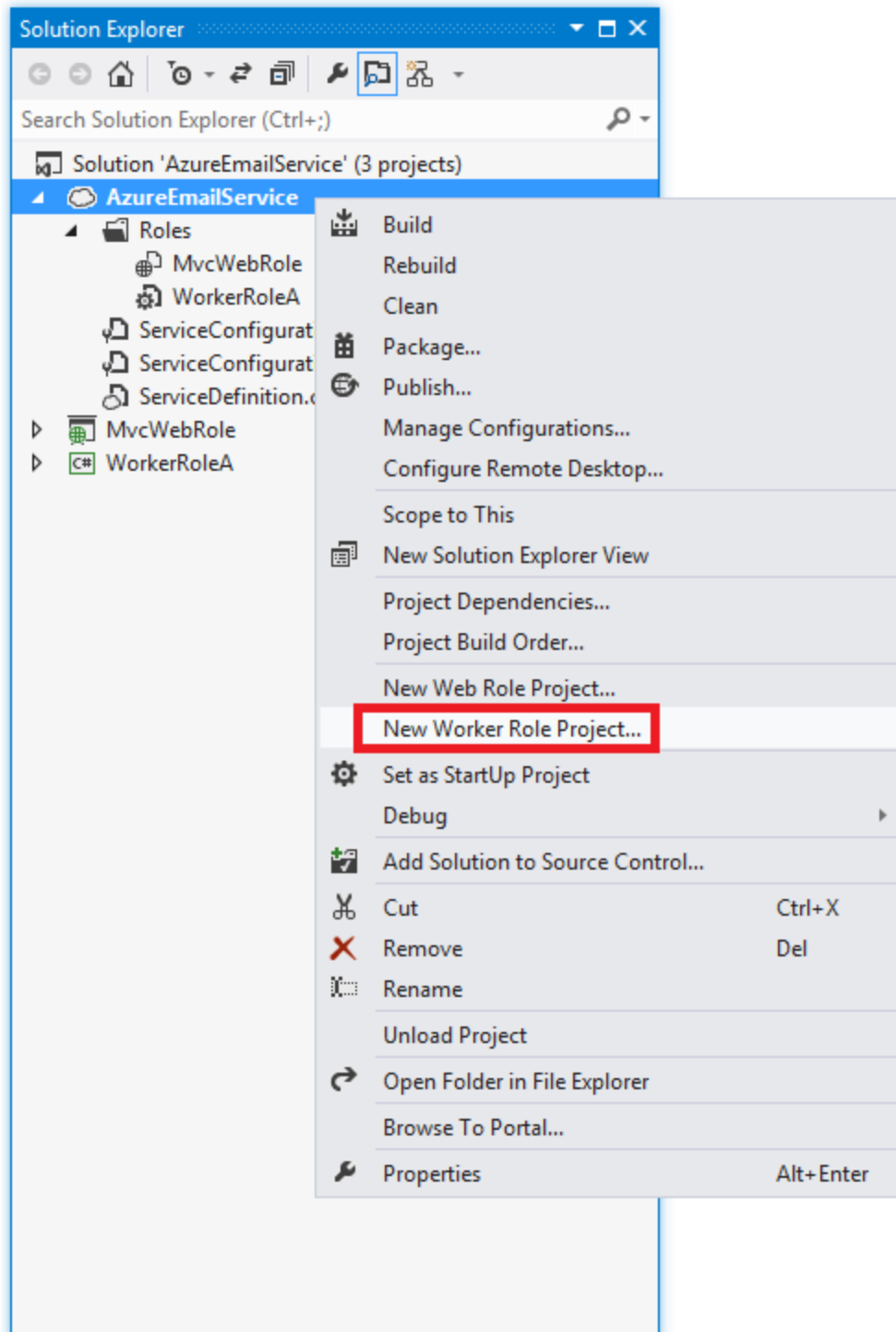
How to send emails by using SendGrid.

How to handle planned shut-downs by overriding the `OnStop` method.

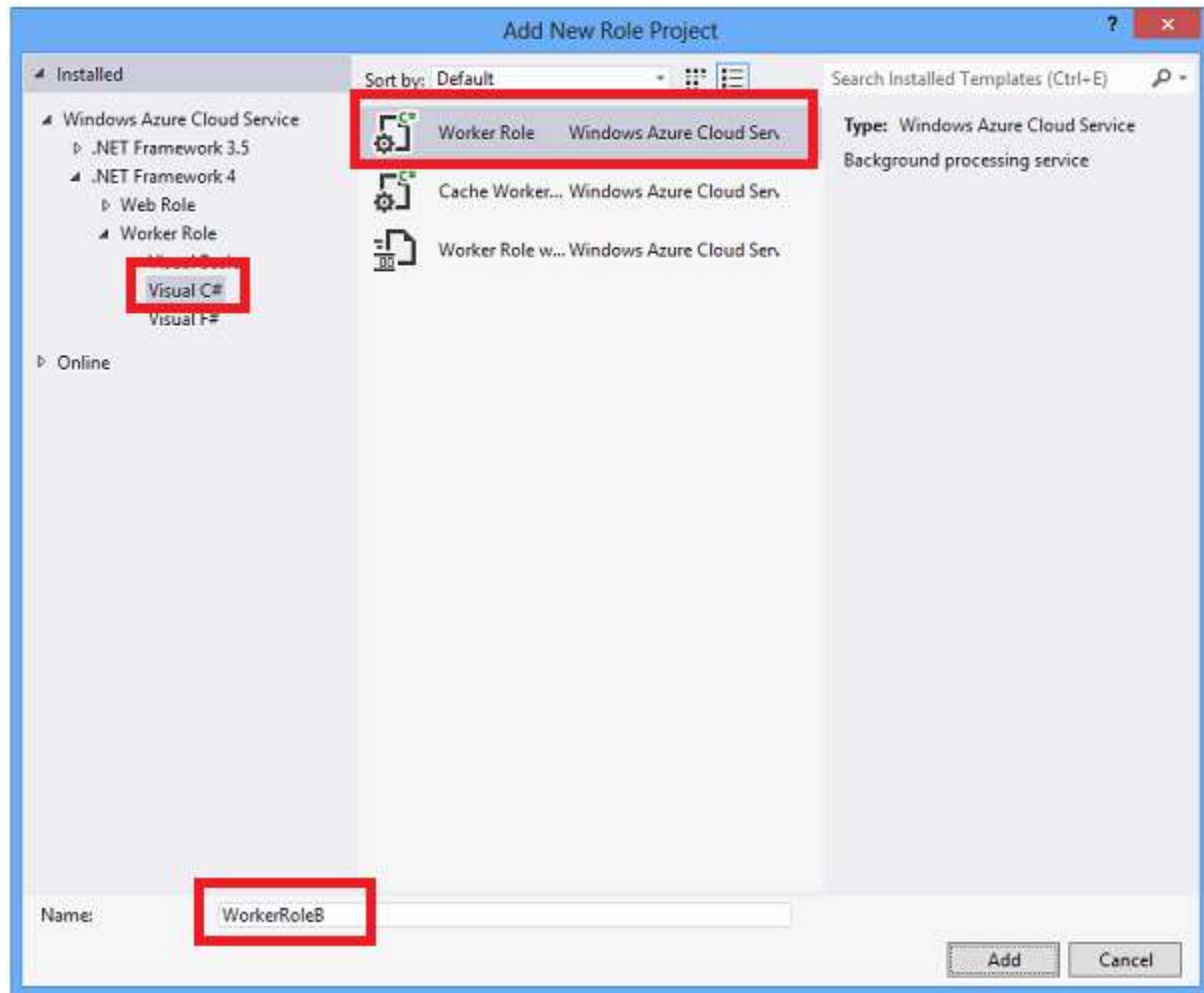
How to handle unplanned shut-downs by making sure that no duplicate emails are sent.

Add worker role BAdd worker role B project to the solution

1. In Solution Explorer, right-click the cloud service project, and choose New Worker Role Project.



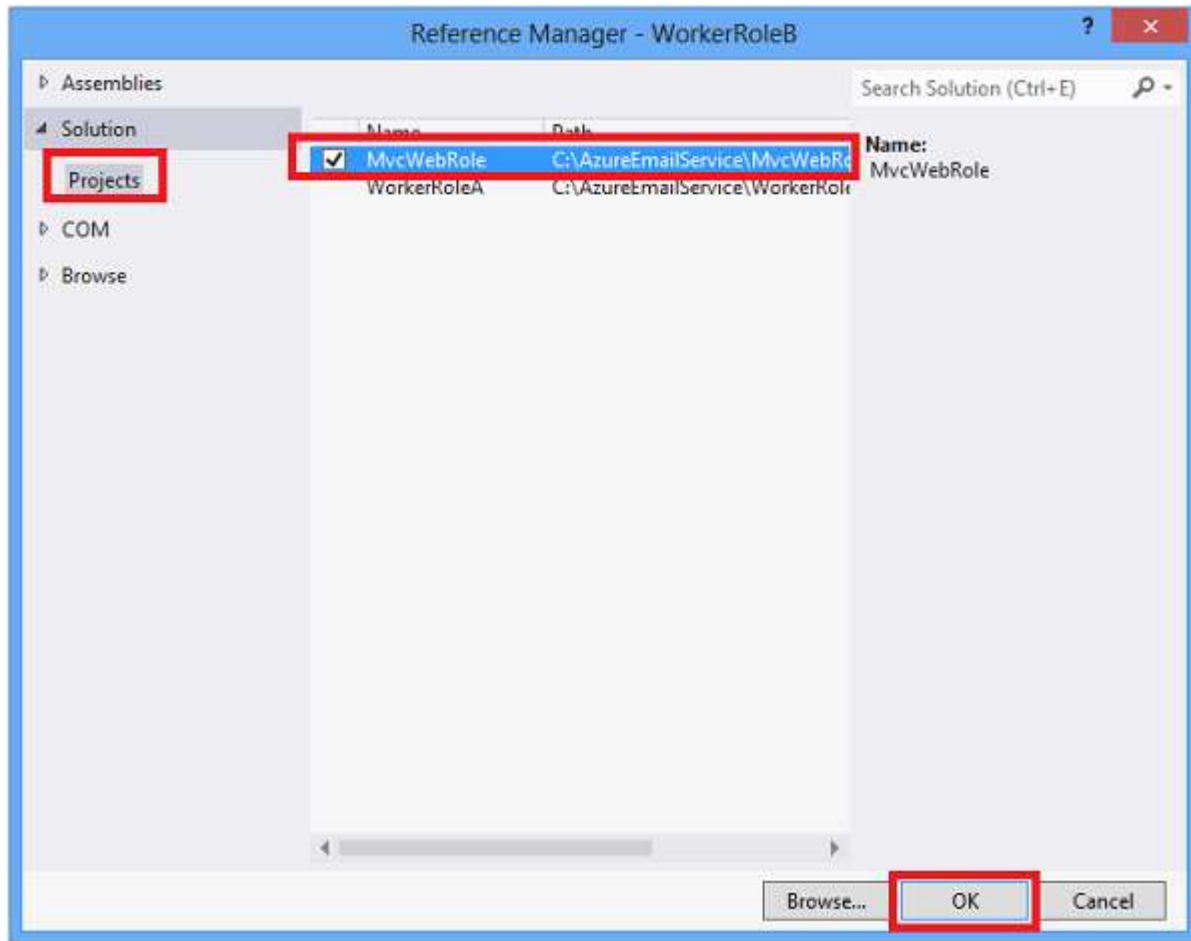
2. In the Add New Role Project dialog box, select C#, select Worker Role, name the project WorkerRoleB, and click Add.



Add a reference to the web project

You need a reference to the web project because that is where the entity classes are defined. You'll use the entity classes in worker role B to read and write data in the Azure tables that the application uses.

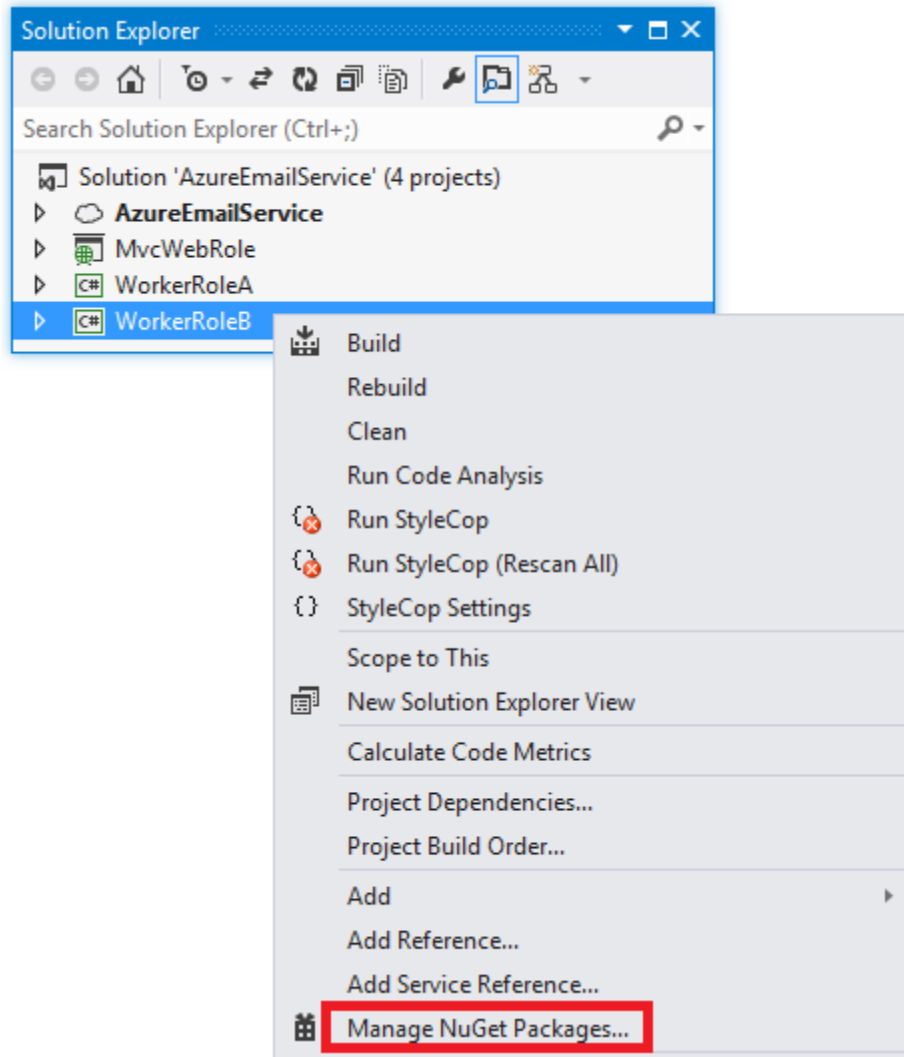
1. Right-click the WorkerRoleB project, and choose Add - Reference.
2. In Reference Manager, add a reference to the MvcWebRole project.



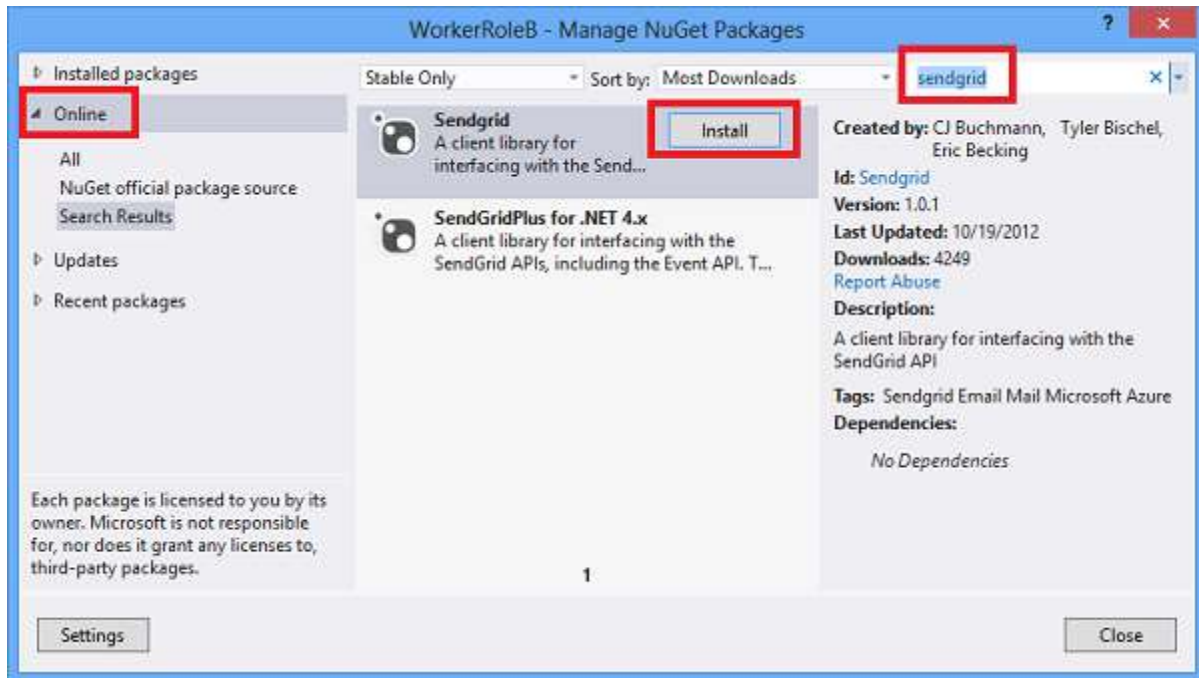
Add the SendGrid NuGet package to the project

To send email by using SendGrid, you need to install the SendGrid NuGet package.

1. In Solution Explorer, right-click the WorkerRoleB project and choose Manage NuGet Packages.



2. In the Manage NuGet Packages dialog box, select the Online tab, enter "sendgrid" in the search box, and press Enter.
3. Click Install on the SendGrid package.



4. Close the dialog box.

Add project settings

Like worker role A, worker role B needs storage account credentials to work with tables, queues, and blobs. In addition, in order to send email, the worker role needs to have credentials to embed in calls to the SendGrid service. And in order to construct an unsubscribe link to include in emails that it sends, the worker role needs to know the URL of the application. These values are stored in project settings.

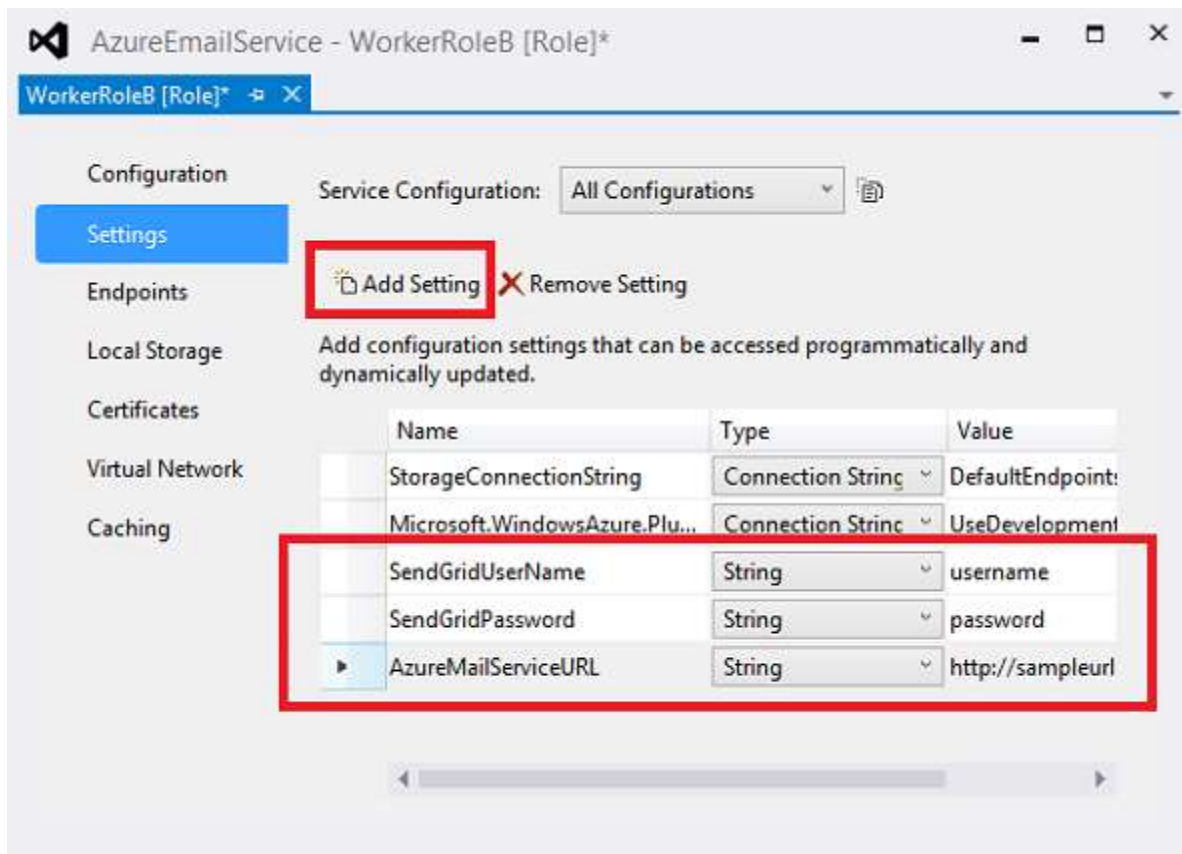
For storage account credentials, the procedure is the same as what you saw in [the third tutorial](#).

1. In Solution Explorer, under Roles in the cloud project, right-click WorkerRoleB and choose Properties.
2. Select the Settings tab.
3. Make sure that All Configurations is selected in the Service Configuration drop-down list.
4. Select the Settings tab and then click Add Setting.
5. Enter "StorageConnectionString" in the Name column.
6. Select Connection String in the Type drop-down list.
7. Click the ellipsis (...) button at the right end of the line to open the Storage Account Connection String dialog box.

8. In the Create Storage Connection String dialog, click the Azure storage emulator radio button, and then click OK.

Next, you create and configure the three new settings that are only used by worker role B.

1. In the Settings tab of the Properties window, Click Add Setting, and then add three new settings of type String:
 - o Name: SendGridUserName, Value: the SendGrid user name that you established in [the second tutorial](#).
 - o Name: SendGridPassword, Value: the SendGrid password.
 - o Name: AzureMailServiceURL, Value: the base URL that the application will have when you deploy it, for example: `http://sampleurl.cloudapp.net`.



Add code that runs when the worker role starts

1. In the WorkerRoleB project, delete WorkerRole.cs.
2. In the WorkerRoleB project, add the WorkerRoleB.cs file from the downloaded project.
3. Build the solution.

If you get a build error, you might have ended up with multiple versions of the Azure Storage NuGet package. In that case, to resolve the problem, go into the NuGet Package Manager for the solution, select Installed packages, and scroll down to see if there are two instances of the Azure Storage package. Select the entry for Azure Storage 4.0.0, and delete it from the projects it's installed in. Then select the entry for Azure Storage 3.0.x and install it in the projects it's missing from. Close and restart Visual Studio, and then build the solution again.

4. Make sure that the cloud service project is still the startup project for the solution.

The Onstart method

As you already saw in worker role A, the `OnStart` method initializes the context classes that you need in order to work with Azure storage entities. It also makes sure that all of the tables, queues, and blob containers you need in the `Run` method exist.

```
public override bool OnStart()
{
    ServicePointManager.DefaultConnectionLimit =
Environment.ProcessorCount * 12;

    // Read storage account configuration settings
    ConfigureDiagnostics();
    Trace.TraceInformation("Initializing storage account in worker
role B");
    var storageAccount =
CloudStorageAccount.Parse(RoleEnvironment.GetConfigurationSettingValue("St
orageConnectionString"));

    // Initialize queue storage
    Trace.TraceInformation("Creating queue client.");
    CloudQueueClient queueClient =
storageAccount.CreateCloudQueueClient();
    this.sendEmailQueue =
queueClient.GetQueueReference("azuremailqueue");
    this.subscribeQueue =
queueClient.GetQueueReference("azuremailsubscribequeue");

    // Initialize blob storage
```

```

        CloudBlobClient blobClient =
storageAccount.CreateCloudBlobClient();
        this.blobContainer =
blobClient.GetContainerReference("azuremailblobcontainer");

        // Initialize table storage
        var tableClient = storageAccount.CreateCloudTableClient();
        tableServiceContext = tableClient.GetDataServiceContext();

        Trace.TraceInformation("WorkerB: Creating blob container, queue,
tables, if they don't exist.");
        this.blobContainer.CreateIfNotExists();
        this.sendEmailQueue.CreateIfNotExists();
        this.subscribeQueue.CreateIfNotExists();
        var messageTable = tableClient.GetTableReference("Message");
        messageTable.CreateIfNotExists();
        var mailingListTable =
tableClient.GetTableReference("MailingList");
        mailingListTable.CreateIfNotExists();

        return base.OnStart();
    }

```

The difference compared to worker role A is the addition of the blob container and the subscribe queue among the resources to create if they don't already exist. You'll use the blob container to get the files that contain the HTML and plain text for the email body. The subscribe queue is used for sending subscription confirmation emails.

The Run method

The Run method processes work items from two queues: the queue used for messages sent to email lists (work items created by worker role A), and the queue used for subscription confirmation emails (work items created by the subscribe API method in the MvcWebRole project).

```

public override void Run()
{
    CloudQueueMessage msg = null;

```

```

Trace.TraceInformation("WorkerRoleB start of Run()");
while (true)
{
    try
    {
        bool messageFound = false;

        // If OnStop has been called, return to do a graceful
shutdown.
        if (onStopCalled == true)
        {
            Trace.TraceInformation("onStopCalled WorkerRoleB");
            returnedFromRunMethod = true;
            return;
        }
        // Retrieve and process a new message from the send-email-
to-list queue.
        msg = sendEmailQueue.GetMessage();
        if (msg != null)
        {
            ProcessQueueMessage(msg);
            messageFound = true;
        }

        // Retrieve and process a new message from the subscribe
queue.
        msg = subscribeQueue.GetMessage();
        if (msg != null)
        {
            ProcessSubscribeQueueMessage(msg);
            messageFound = true;
        }

        if (messageFound == false)

```



```

        {
            System.Threading.Thread.Sleep(1000 * 60);
        }
    }
    catch (Exception ex)
    {
        string err = ex.Message;
        if (ex.InnerException != null)
        {
            err += " Inner Exception: " +
ex.InnerException.Message;
        }
        if (msg != null)
        {
            err += " Last queue message retrieved: " +
msg.AsString;
        }
        Trace.TraceError(err);
        // Don't fill up Trace storage if we have a bug in either
process loop.
        System.Threading.Thread.Sleep(1000 * 60);
    }
}
}

```

This code runs in an infinite loop until the worker role is shut down. If a work item is found in the main queue, the code processes it and then checks the subscribe queue.

```

        // Retrieve and process a new message from the send-email-
to-list queue.
        msg = this.sendEmailQueue.GetMessage();
        if (msg != null)
        {
            ProcessQueueMessage(msg);
            messageFound = true;
        }
    }
}

```

```

        // Retrieve and process a new message from the subscribe
queue.

        msg = this.subscribeQueue.GetMessage();
        if (msg != null)
        {
            ProcessSubscribeQueueMessage(msg);
            messageFound = true;
        }

```

If nothing is waiting in either queue, the code sleeps 60 seconds before continuing with the loop.

```

        if (messageFound == false)
        {
            System.Threading.Thread.Sleep(1000 * 60);
        }

```

The purpose of the sleep time is to minimize Azure Storage transaction costs, as explained in [the previous tutorial](#).

When a queue item is pulled from the queue by the [GetMessage](#) method, that queue item becomes invisible for 30 seconds to all other worker and web roles accessing the queue. This is what ensures that only one worker role instance will pick up any given queue message for processing. You can explicitly set this *exclusive lease* time (the time the queue item is invisible) by passing a [visibility timeout](#) parameter to the `GetMessage` method. If the worker role could take more than 30 seconds to process a queue message, you should increase the exclusive lease time to prevent other role instances from processing the same message.

On the other hand, you don't want to set the exclusive lease time to an excessively large value. For example, if the exclusive lease time is set to 48 hours and your worker role unexpectedly shuts down after dequeuing a message, another worker role would not be able to process the message for 48 hours. The exclusive lease maximum is 7 days.

The [GetMessages](#) method (notice the "s" at the end of the name) can be used to pull up to 32 messages from the queue in one call. Each queue access incurs a small transaction cost, and the transaction cost is the same whether 32 messages are returned or zero messages are returned. The following code fetches up to 32 messages in one call and then processes them.

```

foreach (CloudQueueMessage msg in sendEmailQueue.GetMessages(32))
{

```

```
        ProcessQueueMessage(msg);  
        messageFound = true;  
    }
```

When using `GetMessages` to remove multiple messages, be sure the visibility timeout gives your application enough time to process all the messages. Once the visibility timeout expires, other role instances can access the message, and once they do, the first instance will not be able to delete the message when it finishes processing the work item.

The `ProcessQueueMessage` method

The `Run` method calls `ProcessQueueMessage` when it finds a work item in the main queue:

```
private void ProcessQueueMessage(CloudQueueMessage msg)  
{  
    // Log and delete if this is a "poison" queue message (repeatedly  
processed  
    // and always causes an error that prevents processing from  
completing).  
    // Production applications should move the "poison" message to a  
"dead message"  
    // queue for analysis rather than deleting the message.  
    if (msg.DequeueCount > 5)  
    {  
        Trace.TraceError("Deleting poison message:    message {0} Role  
Instance {1}.",  
            msg.ToString(), GetRoleInstance());  
        sendEmailQueue.DeleteMessage(msg);  
        return;  
    }  
    // Parse message retrieved from queue.  
    // Example:  2012-01-01,0123456789email@domain.com,0  
    var messageParts = msg.AsString.Split(new char[] { ',' });  
    var partitionKey = messageParts[0];  
    var rowKey = messageParts[1];  
    var restartFlag = messageParts[2];
```

```

        Trace.TraceInformation("ProcessQueueMessage start:  partitionKey
{0} rowKey {1} Role Instance {2}.",
        partitionKey, rowKey, GetRoleInstance());
        // If this is a restart, verify that the email hasn't already been
sent.
        if (restartFlag == "1")
        {
            var retrieveOperationForRestart =
TableOperation.Retrieve<SendEmail>(partitionKey, rowKey);
            var retrievedResultForRestart =
messagearchiveTable.Execute(retrieveOperationForRestart);
            var messagearchiveRow = retrievedResultForRestart.Result as
SendEmail;
            if (messagearchiveRow != null)
            {
                // SendEmail row is in archive, so email is already sent.
                // If there's a SendEmail Row in message table, delete it,
                // and delete the queue message.
                Trace.TraceInformation("Email already sent: partitionKey="
+ partitionKey + " rowKey= " + rowKey);
                var deleteOperation = TableOperation.Delete(new SendEmail
{ PartitionKey = partitionKey, RowKey = rowKey, ETag = "*" });
                try
                {
                    messageTable.Execute(deleteOperation);
                }
                catch
                {
                }
                sendEmailQueue.DeleteMessage(msg);
                return;
            }
        }

        // Get the row in the Message table that has data we
need to send the email.

```

```

        var retrieveOperation =
TableOperation.Retrieve<SendEmail>(partitionKey, rowKey);
        var retrievedResult = messageTable.Execute(retrieveOperation);
        var emailRowInMessageTable = retrievedResult.Result as SendEmail;
        if (emailRowInMessageTable == null)
        {
            Trace.TraceError("SendEmail row not found:  partitionKey {0}
rowKey {1} Role Instance {2}.",
                partitionKey, rowKey, GetRoleInstance());
            return;
        }
        // Derive blob names from the MessageRef.
        var htmlMessageBodyRef = emailRowInMessageTable.MessageRef +
".htm";
        var textMessageBodyRef = emailRowInMessageTable.MessageRef +
".txt";
        // If the email hasn't already been sent, send email and archive
the table row.
        if (emailRowInMessageTable.EmailSent != true)
        {
            SendEmailToList(emailRowInMessageTable, htmlMessageBodyRef,
textMessageBodyRef);

            var emailRowToDelete = new SendEmail { PartitionKey =
partitionKey, RowKey = rowKey, ETag = "*" };
            emailRowInMessageTable.EmailSent = true;

            var upsertOperation =
TableOperation.InsertOrReplace(emailRowInMessageTable);
            messagearchiveTable.Execute(upsertOperation);
            var deleteOperation = TableOperation.Delete(emailRowToDelete);
            messageTable.Execute(deleteOperation);
        }

        // Delete the queue message.

```

```

        sendEmailQueue.DeleteMessage(msg);

        Trace.TraceInformation("ProcessQueueMessage complete:
partitionKey {0} rowKey {1} Role Instance {2}.",
            partitionKey, rowKey, GetRoleInstance());
    }

```

Poison messages are those that cause the application to throw an exception when they are processed. If a message has been pulled from the queue more than five times, we assume that it cannot be processed and remove it from the queue so that we don't keep trying to process it. Production applications should consider moving the poison message to a "dead message" queue for analysis rather than deleting the message.

The code parses the queue message into the partition key and row key needed to retrieve the SendEmail row, and a restart flag.

```

var messageParts = msg.AsString.Split(new char[] { ',' });
var partitionKey = messageParts[0];
var rowKey = messageParts[1];
var restartFlag = messageParts[2];

```

If processing for this message has been restarted after an unexpected shut down, the code checks the messagearchive table to determine if this email has already been sent. If it has already been sent, the code deletes the SendEmail row if it exists and deletes the queue message.

```

    if (restartFlag == "1")
    {
        var retrieveOperationForRestart =
TableOperation.Retrieve<SendEmail>(partitionKey, rowKey);
        var retrievedResultForRestart =
messagearchiveTable.Execute(retrieveOperationForRestart);
        var messagearchiveRow = retrievedResultForRestart.Result as
SendEmail;
        if (messagearchiveRow != null)
        {
            Trace.TraceInformation("Email already sent: partitionKey="
+ partitionKey + " rowKey= " + rowKey);
            var deleteOperation = TableOperation.Delete(new SendEmail
{ PartitionKey = partitionKey, RowKey = rowKey, ETag = "*" });

```

```

        try
        {
            messageTable.Execute(deleteOperation);
        }
        catch
        {
        }

        sendEmailQueue.DeleteMessage(msg);
        return;
    }
}

```

Next, we get the SendEmail row from the message table. This row has all of the information needed to send the email, except for the blobs that contain the HTML and plain text body of the email.

```

        var retrieveOperation =
TableOperation.Retrieve<SendEmail>(partitionKey, rowKey);
        var retrievedResult = messageTable.Execute(retrieveOperation);
        var emailRowInMessageTable = retrievedResult.Result as SendEmail;
        if (emailRowInMessageTable == null)
        {
            Trace.TraceError("SendEmail row not found:  partitionKey {0}
rowKey {1} Role Instance {2}.",
                partitionKey, rowKey, GetRoleInstance());
            return;
        }

```

Then the code sends the email and archives the SendEmail row.

```

        if (emailRowInMessageTable.EmailSent != true)
        {
            SendEmailToList(emailRowInMessageTable, htmlMessageBodyRef,
textMessageBodyRef);

            var emailRowToDelete = new SendEmail { PartitionKey =
partitionKey, RowKey = rowKey, ETag = "*" };
            emailRowInMessageTable.EmailSent = true;

```

```
        var upsertOperation =  
TableOperation.InsertOrReplace(emailRowInMessageTable);  
        messagearchiveTable.Execute(upsertOperation);  
        var deleteOperation = TableOperation.Delete(emailRowToDelete);  
        messageTable.Execute(deleteOperation);  
    }
```

Moving the row to the messagearchive table can't be done in a transaction because it affects multiple tables.

Finally, if everything else is successful, the queue message is deleted.

```
sendEmailQueue.DeleteMessage(msg);
```

The SendEmailToList method

The actual work of sending the email by using SendGrid is done by the `SendEmailToList` method. If you want to use a different service than SendGrid, all you have to do is change the code in this method.

Note: If you have invalid credentials in the project settings, the call to SendGrid will fail but the application will not get any indication of the failure. If you use SendGrid in a production application, consider setting up separate credentials for the web API in order to avoid causing silent failures when an administrator changes his or her SendGrid user account password. For more information, see [SendGrid MultiAuth - Multiple Account Credentials](#). You can set up credentials at <https://sendgrid.com/credentials>.

```
private void SendEmailToList(string emailAddress, string  
fromEmailAddress, string subjectLine,  
    string htmlMessageBodyRef, string textMessageBodyRef)  
{  
    var email = SendGrid.GetInstance();  
    email.From = new MailAddress(fromEmailAddress);  
    email.AddTo(emailAddress);  
    email.Html = GetBlobText(htmlMessageBodyRef);  
    email.Text = GetBlobText(textMessageBodyRef);  
    email.Subject = subjectLine;  
    var credentials = new  
NetworkCredential(RoleEnvironment.GetConfigurationSettingValue("SendGridUs  
erName"),
```



```

RoleEnvironment.GetConfigurationSettingValue("SendGridPassword"));
    var transportREST = Web.GetInstance(credentials);
    transportREST.Deliver(email);
}

private string GetBlobText(string blogRef)
{
    var blob = blobContainer.GetBlockBlobReference(blogRef);
    blob.FetchAttributes();
    var blobSize = blob.Properties.Length;
    using (var memoryStream = new MemoryStream((int)blobSize))
    {
        blob.DownloadToStream(memoryStream);
        return
System.Text.Encoding.UTF8.GetString(memoryStream.ToArray());
    }
}

```

In the `GetBlobText` method, the code gets the blob size and then uses that value to initialize the `MemoryStream` object for performance reasons. If you don't provide the size, what the `MemoryStream` does is allocate 256 bytes, then when the download exceeds that, it allocates 512 more bytes, and so on, doubling the amount allocated each time. For a large blob this process would be inefficient compared to allocating the correct amount at the start of the download.

The `ProcessSubscribeQueueMessage` method

The `Run` method calls `ProcessSubscribeQueueMessage` when it finds a work item in the subscribe queue:

```

private void ProcessSubscribeQueueMessage(CloudQueueMessage msg)
{
    // Log and delete if this is a "poison" queue message (repeatedly
processed
    // and always causes an error that prevents processing from
completing).

```

```

        // Production applications should move the "poison" message to a
"dead message"
        // queue for analysis rather than deleting the message.
        if (msg.DequeueCount > 5)
        {
            Trace.TraceError("Deleting poison subscribe message:
message {0}.",
                msg.AsString, GetRoleInstance());
            subscribeQueue.DeleteMessage(msg);
            return;
        }
        // Parse message retrieved from queue. Message consists of
// subscriber GUID and list name.
        // Example: 57ab4c4b-d564-40e3-9a3f-81835b3e102e,contosol
        var messageParts = msg.AsString.Split(new char[] { ',' });
        var subscriberGUID = messageParts[0];
        var listName = messageParts[1];
        Trace.TraceInformation("ProcessSubscribeQueueMessage start:
subscriber GUID {0} listName {1} Role Instance {2}.",
            subscriberGUID, listName, GetRoleInstance());
        // Get subscriber info.
        string filter = TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, listName),
            TableOperators.And,
            TableQuery.GenerateFilterCondition("SubscriberGUID",
QueryComparisons.Equal, subscriberGUID));
        var query = new TableQuery<Subscriber>().Where(filter);
        var subscriber =
mailingListTable.ExecuteQuery(query).ToList().Single();
        // Get mailing list info.
        var retrieveOperation =
TableOperation.Retrieve<MailingList>(subscriber.ListName, "mailinglist");
        var retrievedResult = mailingListTable.Execute(retrieveOperation);
        var mailingList = retrievedResult.Result as MailingList;

```

```

        SendSubscribeEmail(subscriberGUID, subscriber, mailingList);

        subscribeQueue.DeleteMessage(msg);

        Trace.TraceInformation("ProcessSubscribeQueueMessage complete:
subscriber GUID {0} Role Instance {1}.",
            subscriberGUID, GetRoleInstance());
    }

```

This method performs the following tasks:

If the message is a "poison" message, logs and deletes it.

Gets the subscriber GUID from the queue message.

Uses the GUID to get subscriber information from the MailingList table.

Sends a confirmation email to the new subscriber.

Deletes the queue message.

As with emails sent to lists, the actual sending of the email is in a separate method, making it easy for you to change to a different email service if you want to do that.

```

private static void SendSubscribeEmail(string subscriberGUID,
Subscriber subscriber, MailingList mailingList)
{
    var email = SendGrid.GetInstance();
    email.From = new MailAddress(mailingList.FromEmailAddress);
    email.AddTo(subscriber.EmailAddress);
    string subscribeURL =
RoleEnvironment.GetConfigurationSettingValue("AzureMailServiceURL") +
        "/subscribe?id=" + subscriberGUID + "&listName=" +
subscriber.ListName;
    email.Html = String.Format("<p>Click the link below to subscribe
to {0}. " +
        "If you don't confirm your subscription, you won't be
subscribed to the list.</p>" +

```

```

        "<a href=\"{1}\">Confirm Subscription</a>",
        mailingList.Description, subscribeURL);

        email.Text = String.Format("Copy and paste the following URL into
        your browser in order to subscribe to {0}. " +
        "If you don't confirm your subscription, you won't be
        subscribed to the list.\n" +
        "{1}", mailingList.Description, subscribeURL);
        email.Subject = "Subscribe to " + mailingList.Description;
        var credentials = new
        NetworkCredential(RoleEnvironment.GetConfigurationSettingValue("SendGridUs
        erName"),
        RoleEnvironment.GetConfigurationSettingValue("SendGridPassword"));
        var transportREST = Web.GetInstance(credentials);
        transportREST.Deliver(email);
    }

```

Test Worker Role B

1. Run the application by pressing F5.
2. Go to the Messages page to see the message you created to test worker role A. After a minute or so, refresh the web page and you will see that the row has disappeared from the list because it has been archived.
3. Check the email inbox where you expect to get the email. Note that there might be delays in the sending of emails by SendGrid or delivery to your email client, so you might have to wait a while to see the email. You might need to check your junk mail folder also.

Next steps

You have now built the Azure Email Service application from scratch, and what you have is the same as the completed project that you downloaded. To deploy to the cloud, test in the cloud, and promote to production, you can use the same procedures that you saw in [the second tutorial](#).

For a sample application that shows how to use LINQ in Azure Storage table service queries, see [PhluffyFotos](#).

To learn more about Azure storage, see the following resource:

[Essential Knowledge for Windows Azure Storage](#) (Bruno Terkaly's blog)

To learn more about the Azure Table service, see the following resources:

[Essential Knowledge for Azure Table Storage](#) (Bruno Terkaly's blog)

[How to get the most out of Windows Azure Tables](#) (Azure Storage team blog)

[How to use the Table Storage Service in .NET](#)

[Windows Azure Storage Client Library 2.0 Tables Deep Dive](#) (Azure Storage team blog)

[Real World: Designing a Scalable Partitioning Strategy for Azure Table Storage](#)

To learn more about the Azure Queue service and Azure Service Bus queues, see the following resources:

[Queue-Centric Work Pattern \(Building Real-World Cloud Apps with Windows Azure\)](#)

[Azure Queues and Azure Service Bus Queues - Compared and Contrasted](#)

[How to use the Queue Storage Service in .NET](#)

To learn more about the Azure Blob service, see the following resources:

[Unstructured Blob Storage \(Building Real-World Cloud Apps with Windows Azure\)](#)

[How to use the Azure Blob Storage Service in .NET](#)

To learn more about autoscaling Azure Cloud Service roles, see the following resources:

[How to Use the Autoscaling Application Block](#)

[Autoscaling and Azure](#)

[Building Elastic, Autoscalable Solutions with Azure](#) (MSDN channel 9 video)

Acknowledgments

These tutorials and the sample application were written by [Rick Anderson](#) and Tom Dykstra. We would like to thank the following people for their assistance:

Barry Dorrans (Twitter [@blowdart](#))

Cory Fowler (Twitter [@SyntaxC4](#))

Joe Giardino

Don Glover

Jai Haridas

[Scott Hunter](#) (Twitter: [@coolcsh](#))

[Brian Swan](#)

[Daniel Wang](#)

The members of the Developer Advisory Council who provided feedback:

- Damir Arh
- Jean-Luc Boucho
- Carlos dos Santos
- Mike Douglas
- Robert Fite
- Gianni Rosa Gallina
- Fabien Lavocat
- Karl Ots
- Carlos-Alejandro Perez
- Sunao Tomita
- Perez Jones Tsisah
- Michiel van Otegem