

Hands-On Lab

Asynchronous Workload Handling in Windows Azure

Lab version: 2.0.0

Last updated: 11/29/2010



Contents

[Overview 3](#_Toc278819547)

[Exercise 1: Adding Poison Tracking to Queue Message Processing 5](#_Toc278819548)

[Task 1 – Handling Poison Messages in a Queue 5](#_Toc278819549)

[Task 2 – Allowing Message Re-injection 12](#_Toc278819550)

[Task 3 – Testing Poison Message Handling 21](#_Toc278819551)

[Exercise 2: Dynamically Loading Types to Process Images 30](#_Toc278819552)

[Task 1 – Creating a Plug-in Model for Image Processing 31](#_Toc278819553)

[Task 2 – Managing and Deploying Plug-in Assemblies 35](#_Toc278819554)

[Task 3 – Dynamically Loading Types from Blob Storage 49](#_Toc278819555)

[Task 4 – Updating the Worker Role to Use Plug-ins 62](#_Toc278819556)

[Task 5 – Resolving Bugs with Zero Downtime 69](#_Toc278819557)

[Task 6 – Extending the Worker Role with Plug-Ins for Additional Image Types 78](#_Toc278819558)

[Summary 82](#_Toc278819559)

Overview

* 1. Windows Azure provides a mechanism for simple asynchronous work using queues and worker roles. In this hands-on lab, you will look at some advanced approaches for building asynchronous Windows Azure services that take advantage of these features.
  2. You will start by familiarizing yourself with a simple Windows Azure service, the Guestbook Application from the Introduction to Windows Azure hands-on lab. This service demonstrates how to process image conversion to thumbnails as a background process. In this application, users access a front-end web role to submit new guest book entries and upload images. For each submission, the web role creates a new entry in table storage and then stores the associated image in blob storage. To process the image in the background, the web role posts a work item to a queue. In the back-end, a worker role retrieves messages from the queue, creates the corresponding thumbnails and stores them in blob storage, and then updates the guest book entry with their address. If the role processes an image successfully, it proceeds to delete the message from the queue.
  3. This is a common pattern in cloud applications and enables the separation of compute-bound tasks from the front-end using queues. The advantage of this approach is that front and back ends can be scaled independently.

# Objectives

* 1. In this hands-on lab, you will look at two key improvements to the Guestbook application. You will learn how to:
  + Use the message dequeue count to identify message that causes the worker role to crash, otherwise known as “poisonous” messages.
  + Dynamically load .NET types from blob storage and invoke these in a way that enables the Guestbook worker role to handle a variety of image types without the need to perform a re-deployment of the Worker role.

# Prerequisites

* 1. The following is required to complete this hands-on lab:
  + IIS 7 (with ASP.NET, WCF HTTP Activation)
  + [Microsoft .NET Framework 4.0](http://go.microsoft.com/fwlink/?linkid=186916)
  + [Microsoft Visual Studio 2010](http://msdn.microsoft.com/vstudio/products/)
  + [Windows Azure Tools for Microsoft Visual Studio 1.3](http://go.microsoft.com/fwlink/?LinkID=128752)
  1. **Note:** To complete this lab you should ideally have already completed the **Introduction to Windows Azure** hands-on lab.
  2. This hands-on lab has been designed to use the latest release of the Windows Azure Tools for Visual Studio 2010 (version 1.3).

# Setup

* 1. For convenience, much of the code used in this hands-on lab is available as Visual Studio code snippets. To check the prerequisites of the lab and install the code snippets:
  2. Open a Windows Explorer window and browse to the lab’s **Source\Setup** folder.
  3. Double-click the **Dependencies.dep** file in this folder to launch the Dependency Checker tool and install any missing prerequisites and the Visual Studio code snippets.
  4. If the User Account Control dialog is shown, confirm the action to proceed.
  5. **Note:** This process may require elevation. The *.dep* extension is associated with the Dependency Checker tool during its installation. For additional information about the setup procedure and how to install the Dependency Checker tool, refer to the **Setup.docx** document in the **Assets** folder of the training kit.

# Using the Code Snippets

* 1. Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of that code is provided as Visual Studio Code Snippets, which you can use from within Visual Studio 2010 to avoid having to add it manually.
  2. If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the **Setup.docx** document in the **Assets** folder of the training kit, which contains a section describing how to use them.

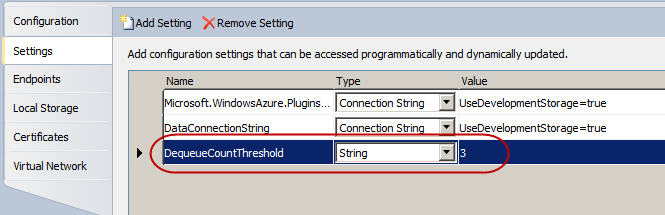
# Exercises

* 1. This hands-on lab includes the following exercises:
  2. Adding Poison Tracking to Queue Message Processing
  3. Dynamically Loading Types to Process Images
  4. Estimated time to complete this lab: **60 minutes**.
  5. **Note:** When you first start Visual Studio, you must select one of the predefined settings collections. Every predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in these procedures that you need to take into account.

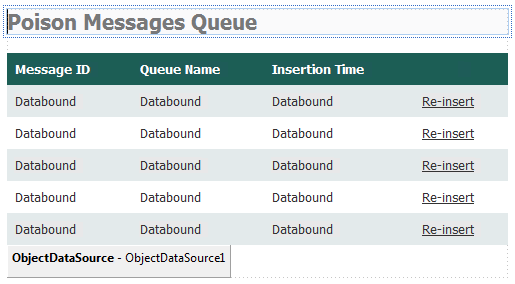
Exercise 1: Adding Poison Tracking to Queue Message Processing

* 1. The Queue storage type provided by Windows Azure works on the basis of messages being de-queued and subsequently deleted permanently. Messages that are not deleted reappear on the queue after a certain period. This approach ensures that each message will be processed to completion at least once. However, under certain circumstances, a message may cause the worker role to crash before it can be processed and removed from the queue, causing it to reappear again and again, crashing the worker role every time. This is broadly referred to as a poison message.
  2. In this exercise, you will learn to use the de-queue count property of Queue messages to determine when a message may have been processed previously. You will set a threshold such that messages that appear to have failed more than a certain number of times are declared poisonous and transferred to another queue.

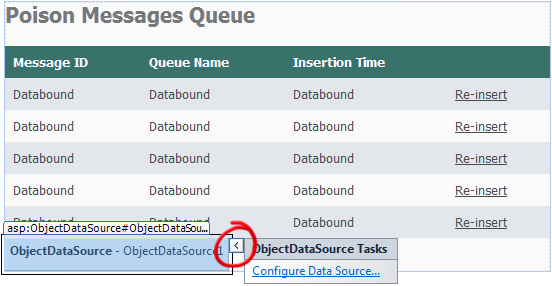
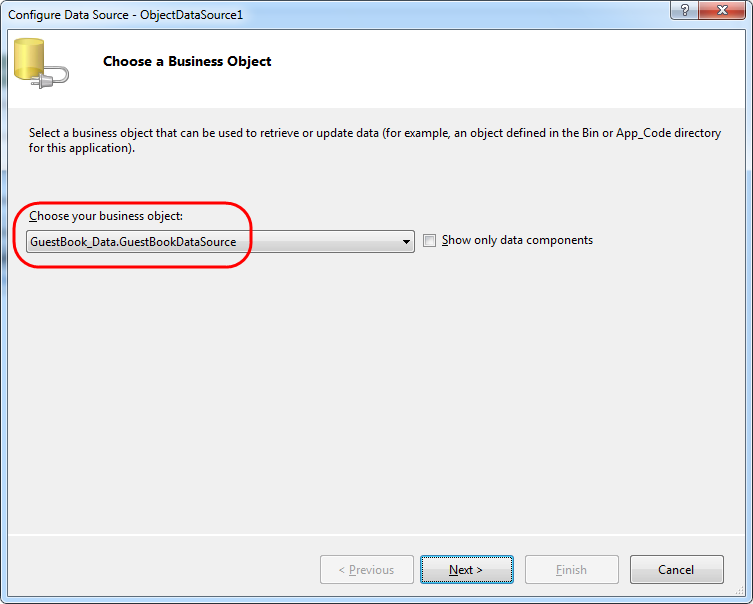
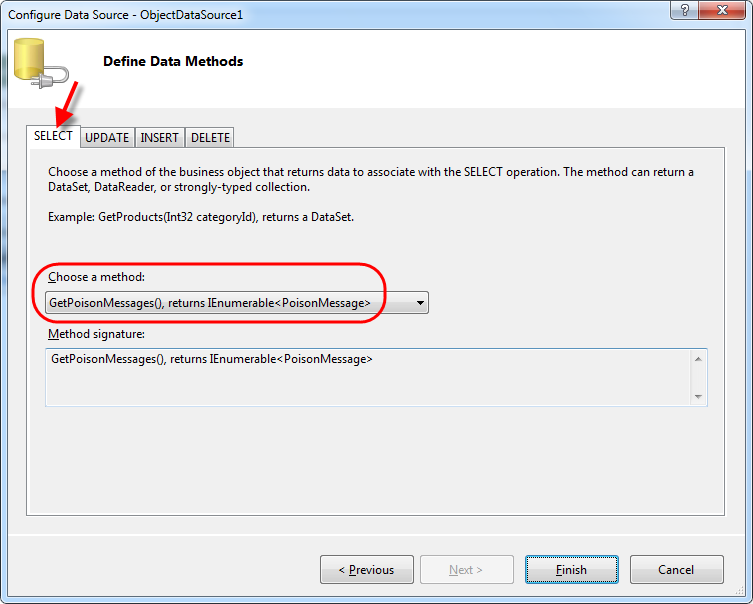
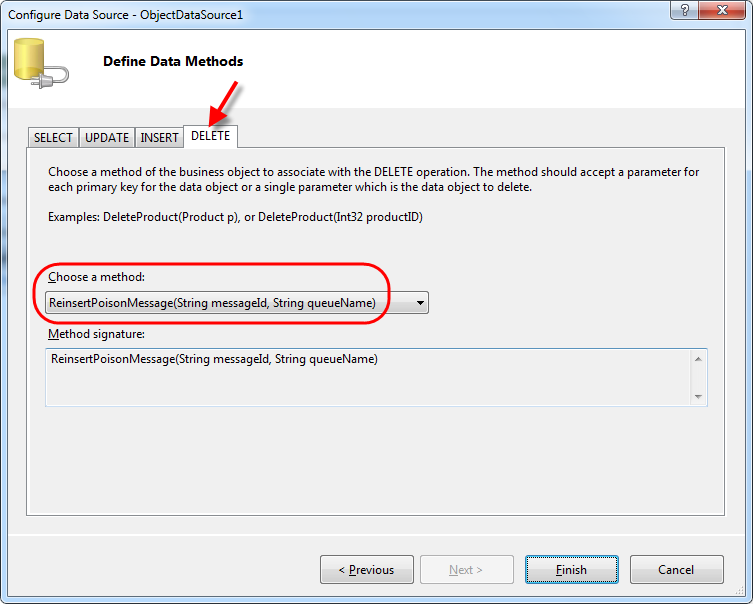
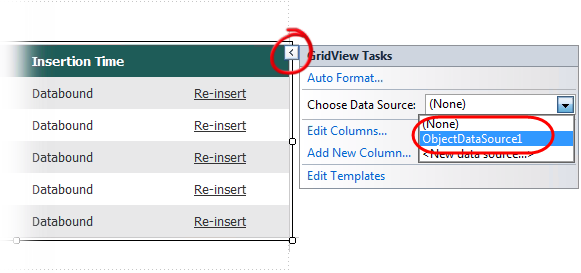
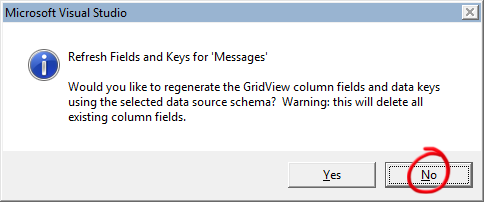
Task 1 – Handling Poison Messages in a Queue

* 1. In this task, you update the worker role code to check the dequeue count as messages are removed from the queue. Checking for poison message status should always be the first operation performed after dequeing a message as any other logic may, in fact, contain the offending code causing the repeated crash.
  2. Open Visual Studioin elevated administrator mode from **Start | All Programs | Microsoft Visual Studio 2010** by right clicking the **Microsoft Visual Studio 2010** shortcut and choosing **Run as administrator**.
  3. If the **User Account Control** dialog appears, click **Continue**.
  4. In the **File** menu, choose **Open** and then **Project/Solution**. In the **Open Project** dialog, browse to **Ex1-PoisonMessageTracking** in the **Source** folder of the lab and choose the folder for the language of your preference (Visual C# or Visual Basic). Select **Begin.sln** in the **Begin** folder and then click **Open**.
  5. Add a new configuration setting to specify the maximum number of times the worker role may retrieve a message from the queue before it discards it and sends it to the poison messages queue. To do this, expand the **Roles** node in the **GuestBook** project and double-click the **GuestBook\_WorkerRole** role node. In the role properties window, switch to the **Settings** page, and then click **Add Setting**. Next, set the **Name** of the new setting to *DequeueCountThreshold*, the **Type** as *String*, and the **Value** as *3*.
     1. 
     2. Figure 1
     3. Configuring the dequeue count threshold
  6. Press **CTRL+S** to save the changes to the role configuration.
  7. Update the worker role code to create a new queue for poison messages. Open the **WorkerRole.cs** file (for Visual C# projects) or **WorkerRole.vb** (for Visual Basic projects), and then insert the following (highlighted) declaration for a field named *poisonQueue* to the **WorkerRole** class.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex1- Poison Queue Declaration-CS*)
     2. C#
     3. public class WorkerRole : RoleEntryPoint
     4. {
     5. private CloudQueue queue;
     6. **private CloudQueue poisonQueue;**
     7. private CloudBlobContainer container;
     8. ...
     9. }
     10. (Code Snippet – *AsynchWorkloadHandling-Ex1- Poison Queue Declaration-VB*)
     11. Visual Basic
     12. Public Class WorkerRole
     13. Inherits RoleEntryPoint
     14. Private queue As CloudQueue
     15. **Private poisonQueue As CloudQueue**
     16. Private container As CloudBlobContainer
     17. ...
     18. End Class
  8. Next, find the **OnStart** method in the **WorkerRole** class and then locate the section of code in this method that initializes queue storage. Insert the following code to initialize the reference to the *poisonQueue* field created in the previous step, as shown (highlighted) below.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex1-OnStart create queue reference-CS*)
     2. C#
     3. public override bool OnStart()
     4. {
     5. ...
     6. // initialize blob storage
     7. CloudBlobClient blobStorage = storageAccount.CreateCloudBlobClient();
     8. container = blobStorage.GetContainerReference("guestbookpics");
     9. // initialize queue storage
     10. CloudQueueClient queueStorage = storageAccount.CreateCloudQueueClient();
     11. queue = queueStorage.GetQueueReference("guestthumbs");
     12. **poisonQueue = queueStorage.GetQueueReference("poisonmessages");**
     13. Trace.TraceInformation("Creating container and queue...");
     14. bool storageInitialized = false;
     15. while (!storageInitialized)
     16. {
     17. ...
     18. }
     19. (Code Snippet – *AsynchWorkloadHandling-Ex1-OnStart create queue reference-VB*)
     20. Visual Basic
     21. Public Overrides Function OnStart() As Boolean
     22. ...
     23. ' initialize blob storage
     24. Dim blobStorage = storageAccount.CreateCloudBlobClient()
     25. container = blobStorage.GetContainerReference("guestbookpics")
     26. ' initialize queue storage
     27. Dim queueStorage = storageAccount.CreateCloudQueueClient()
     28. queue = queueStorage.GetQueueReference("guestthumbs")
     29. **poisonQueue = queueStorage.GetQueueReference("poisonmessages")**
     30. Trace.TraceInformation("Creating container and queue...")
     31. Dim storageInitialized = False
     32. Do While (Not storageInitialized)
     33. ...
     34. End Function
  9. Now, update the storage initialization code in the same method to create the poison messages queue, when the queue does not exist, as shown (highlighted) in the following code snippet.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex1-Poison queue CreateIfNotExist-CS*)
     2. C#
     3. public override bool OnStart()
     4. {
     5. ...
     6. Trace.TraceInformation("Creating container and queue...");
     7. bool storageInitialized = false;
     8. while (!storageInitialized)
     9. {
     10. try
     11. {
     12. // create the blob container and allow public access
     13. container.CreateIfNotExist();
     14. var permissions = container.GetPermissions();
     15. permissions.PublicAccess = BlobContainerPublicAccessType.Container;
     16. container.SetPermissions(permissions);
     17. // create the message queue(s)
     18. queue.CreateIfNotExist();
     19. **poisonQueue.CreateIfNotExist();**
     20. storageInitialized = true;
     21. }
     22. catch (StorageClientException e)
     23. {
     24. ...
     25. }
     26. }
     27. return base.OnStart();
     28. }
     29. (Code Snippet – *AsynchWorkloadHandling-Ex1-Poison queue CreateIfNotExist-VB*)
     30. Visual Basic
     31. Public Overrides Function OnStart() As Boolean
     32. ...
     33. Trace.TraceInformation("Creating container and queue...")
     34. Dim storageInitialized = False
     35. Do While (Not storageInitialized)
     36. Try
     37. ' create the blob container and allow public access
     38. container.CreateIfNotExist()
     39. Dim permissions = container.GetPermissions()
     40. permissions.PublicAccess = BlobContainerPublicAccessType.Container
     41. container.SetPermissions(permissions)
     42. ' create the message queue(s)
     43. queue.CreateIfNotExist()
     44. **poisonQueue.CreateIfNotExist()**
     45. storageInitialized = True
     46. Catch e As StorageClientException
     47. ...
     48. End Try
     49. Loop
     50. Return MyBase.OnStart()
     51. End Function
  10. Finally, adjust the message loop in the worker role to check the dequeue count against the configured threshold and to send any messages that exceed this limit to the poison messages queue. Locate the **Run** method and insert the highlighted block immediately below the lines that retrieve a message from the queue and check whether it is not null.
      1. (Code Snippet – *AsynchWorkloadHandling-Ex1-Check message dequeue count-CS*)
      2. C#
      3. public override void Run()
      4. {
      5. Trace.TraceInformation("Listening for queue messages...");
      6. while (true)
      7. {
      8. try
      9. {
      10. // retrieve a new message from the queue
      11. CloudQueueMessage msg = queue.GetMessage();
      12. if (msg != null)
      13. {
      14. **int dequeueCountThreshold = Convert.ToInt32(RoleEnvironment.GetConfigurationSettingValue("DequeueCountThreshold"));**
      15. **if (msg.DequeueCount >= dequeueCountThreshold)**
      16. **{**
      17. **// add originating queue name and insertion time to message body and transfer message to poison messages queue**
      18. **string content = queue.Name + "," + msg.InsertionTime + "," + msg.AsString;**
      19. **CloudQueueMessage poisonMessage = new CloudQueueMessage(content);**
      20. **poisonQueue.AddMessage(poisonMessage);**
      21. **// remove message from original queue**
      22. **queue.DeleteMessage(msg);**
      23. **Trace.TraceWarning("Unable to process message '{0}'. Moving to poison messages queue.", msg.Id);**
      24. **continue;**
      25. **}**
      26. // parse message retrieved from queue
      27. var messageParts = msg.AsString.Split(new char[] { ',' });
      28. var imageBlobUri = messageParts[0];
      29. ...
      30. }
      31. (Code Snippet – *AsynchWorkloadHandling-Ex1-Check message dequeue count-VB*)
      32. Visual Basic
      33. Public Overrides Sub Run()
      34. Trace.TraceInformation("Listening for queue messages...")
      35. Do
      36. Try
      37. ' retrieve a new message from the queue
      38. Dim msg As CloudQueueMessage = queue.GetMessage()
      39. If msg IsNot Nothing Then
      40. **Dim dequeueCountThreshold = Convert.ToInt32(RoleEnvironment.GetConfigurationSettingValue("DequeueCountThreshold"))**
      41. **If (msg.DequeueCount >= dequeueCountThreshold) Then**
      42. **' add originating queue name and insertion time to message body and transfer message to poison messages queue**
      43. **Dim content = queue.Name + "," + msg.InsertionTime + "," + msg.AsString**
      44. **Dim poisonMessage = New CloudQueueMessage(content)**
      45. **poisonQueue.AddMessage(poisonMessage)**
      46. **' remove message from original queue**
      47. **queue.DeleteMessage(msg)**
      48. **Trace.TraceWarning("Unable to process message '{0}'. Moving to poison messages queue.", msg.Id)**
      49. **Continue Do**
      50. **End If**
      51. ' parse message retrieved from queue
      52. Dim messageParts = msg.AsString.Split(New Char() {","c})
      53. Dim imageBlobUri = messageParts(0)
      54. ...
      55. End Sub
      56. **Note:** The code shown above obtains the dequeue count of the message and compares it with the value retrieved from the *DequeueCountThreshold* configuration setting. If the count exceeds the threshold, it creates a new message after pre-pending the name of the queue and the original insertion time to the original message, posts it to the poison messages queue, and then deletes the original message from the source queue.
      57. Notice that checking the dequeue count is the first action performed immediately after retrieving the message to avoid the possibility that processing the message in any way triggers an unhandled exception that crashes the worker role. If that should happen, the worker role would enter an endless loop where it retrieves a problematic message, crashes, restarts, and then repeats the cycle as the message reappears in the queue after its visibility timeout expires. Moving the message to the poison messages queue without processing it ensures that this never occurs.

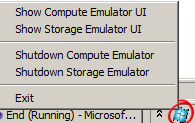
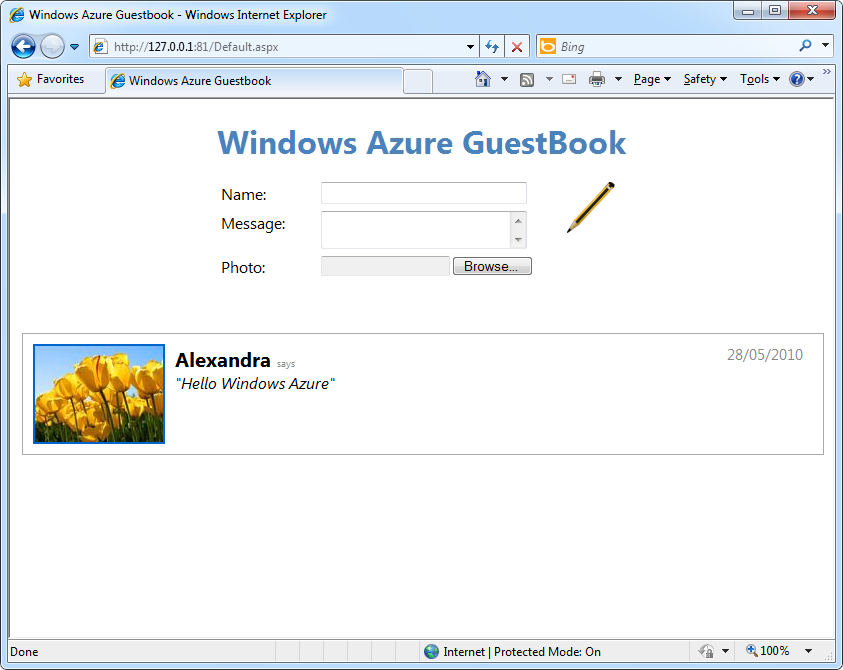
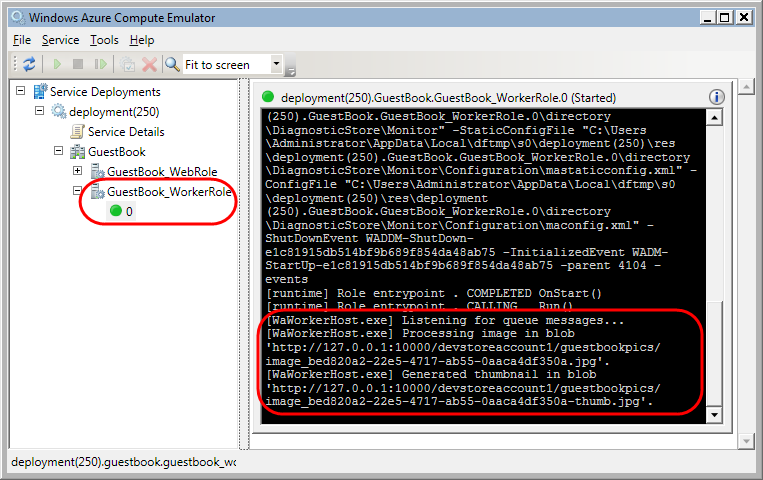
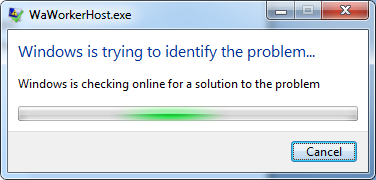
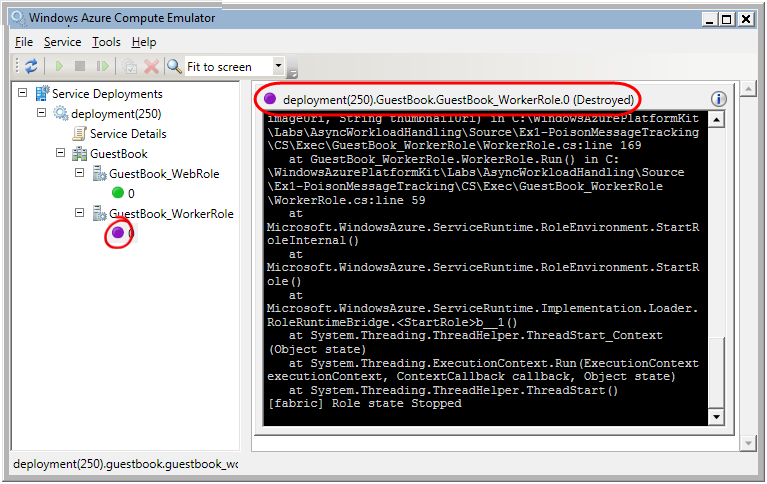
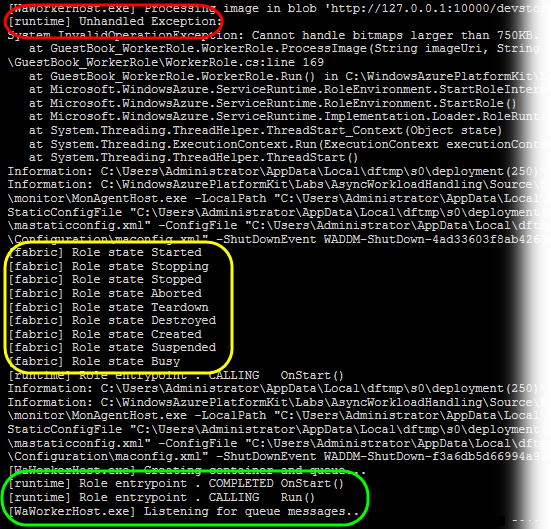
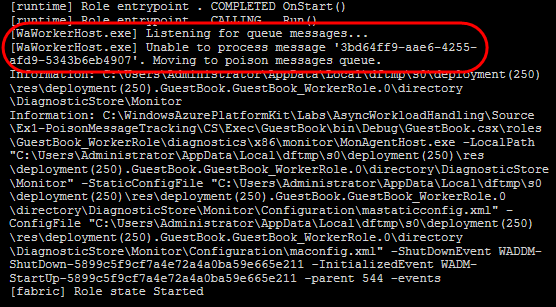
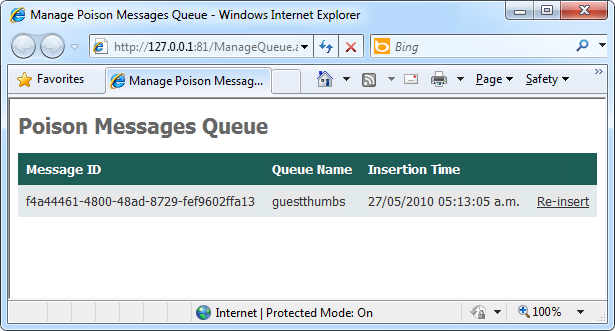
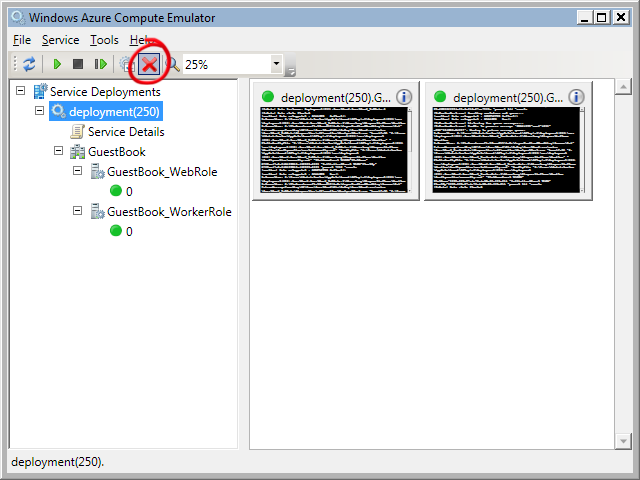
Task 2 – Allowing Message Re-injection

* 1. In this task, you create a simple page that lists the contents of the poison messages queue and allows you to re-inject messages back into their original queue. For simplicity, the starting solution already contains a page with the necessary UI elements. You simply update the code-behind file to implement the required functionality.
  2. Open the provided page template for managing queues. In **Solution Explorer**, right-click **ManageQueue.aspx** in the **GuestBook\_WebRole** project and select **View Designer**.
     1. Notice that the page contains a **GridView** control already configured to show information about messages in the queue, namely *Message ID*, *Queue name*, and *Insertion Time*, as well as a **LinkButton** control to re-insert the corresponding message back into its original queue. The page also contains an **ObjectDataSource** control that you will bind to the **GridView** to populate it.
     2. 
     3. Figure 2
     4. Design view of the queue management page
     5. **Note:** The designer view may be disabled by default depending on your Visual Studio settings. To enable design view in Visual Studio, go to **Options | HTML Designer**, select **Enable HTML Designer**, and then restart Visual Studio.
  3. Now, update the source object that you will bind to the **ObjectDataSource**. To do this, open **GuestBookDataSource.cs** (for Visual C# projects) or **GuestBookDataSource.vb** (for Visual Basic projects) in the **GuestBook\_Data** project.
  4. To define a class to encapsulate messages retrieved from the poison messages queue, add a nested **PoisonMessage** class to the **GuestBookDataSource** class, as shown (highlighted) below.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex1-PoisonMessage class-CS*)
     2. C#
     3. public class GuestBookDataSource
     4. {
     5. ...
     6. **public class PoisonMessage**
     7. **{**
     8. **public PoisonMessage(CloudQueueMessage message)**
     9. **{**
     10. **var messageParts = message.AsString.Split(new char[] { ',' });**
     11. **this.QueueName = messageParts[0];**
     12. **this.InsertionTime = DateTime.Parse(messageParts[1]);**
     13. **this.Body = String.Format("{0},{1},{2}", messageParts[2], messageParts[3], messageParts[4]);**
     14. **this.MessageId = message.Id;**
     15. **}**
     16. **public string MessageId { get; private set; }**
     17. **public string QueueName { get; private set; }**
     18. **public DateTime InsertionTime { get; private set; }**
     19. **public string Body { get; private set; }**
     20. **}**
     21. }
     22. (Code Snippet – *AsynchWorkloadHandling-Ex1-PoisonMessage class-VB*)
     23. Visual Basic
     24. Public Class GuestBookDataSource
     25. ...
     26. **Public Class PoisonMessage**
     27. **Public Sub New(ByVal message As CloudQueueMessage)**
     28. **Dim messageParts = message.AsString.Split(New Char() {","c})**
     29. **Me.QueueName = messageParts(0)**
     30. **Me.InsertionTime = DateTime.Parse(messageParts(1))**
     31. **Me.Body = String.Format("{0},{1},{2}", messageParts(2), messageParts(3), messageParts(4))**
     32. **Me.MessageId = message.Id**
     33. **End Sub**
     34. **Public Property MessageId() As String**
     35. **Public Property QueueName() As String**
     36. **Public Property InsertionTime() As DateTime**
     37. **Public Property Body() As String**
     38. **End Class**
     39. End Class
     40. **Note:** Ensure that you insert this class nested inside the **GuestBookDataSource** class.
  5. Next, add a method to retrieve a list of messages from the poison messages queue, as shown in the following (highlighted) code snippet.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex1-GetPoisonMessages method-CS*)
     2. C#
     3. public class GuestBookDataSource
     4. {
     5. ...
     6. **public IEnumerable<PoisonMessage> GetPoisonMessages()**
     7. **{**
     8. **CloudQueueClient queueStorage = storageAccount.CreateCloudQueueClient();**
     9. **CloudQueue queue = queueStorage.GetQueueReference("poisonmessages");**
     10. **return from message in queue.PeekMessages(32)**
     11. **select new PoisonMessage(message);**
     12. **}**
     13. public class PoisonMessage
     14. {
     15. ...
     16. }
     17. }
     18. (Code Snippet – *AsynchWorkloadHandling-Ex1-GetPoisonMessages method-VB*)
     19. Visual Basic
     20. Public Class GuestBookDataSource
     21. ...
     22. **Public Function GetPoisonMessages() As IEnumerable(Of PoisonMessage)**
     23. **Dim queueStorage As CloudQueueClient = storageAccount.CreateCloudQueueClient()**
     24. **Dim queue As CloudQueue = queueStorage.GetQueueReference("poisonmessages")**
     25. **Return From message In queue.PeekMessages(32) \_**
     26. **Select New PoisonMessage(message)**
     27. **End Function**
     28. Public Class PoisonMessage
     29. ...
     30. End Class
     31. End Class

**Note:** The preceding code uses the **PeekMessages** method of the **CloudQueue** object to return a collection of messages from the poison messages queue. This method retrieves the messages without removing them from the queue. The LINQ query projects the result onto a list of **PoisonMessage** objects to provide easier access to the information contained in each message.

* + 1. Note that the code retrieves 32 messages, which is the maximum allowed for this operation.
  1. To complete the poison message handling methods, add a method to remove a message from the poison messages queue and put it back into its original queue, as shown (highlighted) below.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex1-ReinsertPoisonMessage method-CS*)
     2. C#
     3. public class GuestBookDataSource
     4. {
     5. ...
     6. **public void ReinsertPoisonMessage(string messageId, string queueName)**
     7. **{**
     8. **CloudQueueClient queueStorage = storageAccount.CreateCloudQueueClient();**
     9. **CloudQueue originalQueue = queueStorage.GetQueueReference(queueName);**
     10. **CloudQueue poisonQueue = queueStorage.GetQueueReference("poisonmessages");**
     11. **var message = (from msg in poisonQueue.GetMessages(32, TimeSpan.FromSeconds(1))**
     12. **where msg.Id == messageId**
     13. **select msg).FirstOrDefault();**
     14. **if (message != null)**
     15. **{**
     16. **originalQueue.AddMessage(new CloudQueueMessage(new PoisonMessage(message).Body));**
     17. **poisonQueue.DeleteMessage(messageId, message.PopReceipt);**
     18. **}**
     19. **}**
     20. ...
     21. }
     22. (Code Snippet – *AsynchWorkloadHandling-Ex1-ReinsertPoisonMessage method-VB*)
     23. Visual Basic
     24. Public Class GuestBookDataSource
     25. ...
     26. **Public Sub ReinsertPoisonMessage(ByVal messageId As String, ByVal queueName As String)**
     27. **Dim queueStorage As CloudQueueClient = storageAccount.CreateCloudQueueClient()**
     28. **Dim originalQueue As CloudQueue = queueStorage.GetQueueReference(queueName)**
     29. **Dim poisonQueue As CloudQueue = queueStorage.GetQueueReference("poisonmessages")**
     30. **Dim message = (From msg In poisonQueue.GetMessages(32, TimeSpan.FromSeconds(1)) \_**
     31. **Where msg.Id = messageId \_**
     32. **Select msg).FirstOrDefault()**
     33. **If message IsNot Nothing Then**
     34. **originalQueue.AddMessage(New CloudQueueMessage(New PoisonMessage(message).Body))**
     35. **poisonQueue.DeleteMessage(messageId, message.PopReceipt)**
     36. **End If**
     37. **End Sub**
     38. ...
     39. End Class
  2. Build the project to ensure that the updated methods in the **GuestBookDataSource** class are available to configure the **ObjectDataSource** control.
  3. Return to the **ManageQueue.aspx** page. If necessary, switch to design view, and then click the smart tag in the upper right corner of the **ObjectDataSource** control to open the **ObjectDataSource Tasks** menu and select **Configure Data Source**.
     1. 
     2. Figure 3
     3. Configuring the data source for the grid control
  4. In the **Choose a Business Object** page of the **Configure Data Source** wizard, expand the list labeled **Choose your business object**, select **GuestBook\_Data.GuestBookDataSource** from the drop down list of data types, and then click **Next**.
     1. 
     2. Figure 4
     3. Choosing a business object for the data source in the queue management page
  5. In the **Define Data Methods** page, choose the **SELECT** method, and then pick **GetPoisonMessages** from the drop down list labeled **Choose a method**.
     1. 
     2. Figure 5
     3. Configuring the select method of the data source
  6. Next, select the **DELETE** method and choose **RemovePoisonMessage** from the drop down list that lists the methods in the business object.
     1. 
     2. Figure 6
     3. Choosing the delete method of the data source
  7. Click **Finish** to close the wizard.
  8. Now, click the smart tag in the upper right corner of the **GridView** control to open the **GridView Tasks** menu and then select **ObjectDataSource1** in the drop down list labeled **Choose Data Source**.
     1. 
     2. Figure 7
     3. Configuring a data source for the queue messages grid
  9. After you select a data source, Visual Studio prompts you to refresh the column fields of the **GridView** using the schema of the data source. Answer **No** to this question; otherwise, you will lose the existing column definitions in the page.
     1. 
     2. Figure 8
     3. Avoid regenerating the grid’s column fields

Task 3 – Testing Poison Message Handling

* 1. In order to test that the poison message handling is working correctly, you need to simulate your worker role crashing. To do this, you add code to the image processing worker role such that it throws an exception whenever an image file exceeding a certain size is processed, thus imitating a failure caused by unforeseen conditions.
  2. Open **WorkerRole.cs** (for Visual C# projects) or **WorkerRole.vb** (for Visual Basic projects) in the **GuestBook\_WorkerRole** project, locate the **ProcessImage** method and insert the following (highlighted) code at the start of the method to throw an exception whenever the length of the stream is greater than a given value.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex1-ProcessImage unhandled exception-CS*)
     2. C#
     3. public void ProcessImage(Stream input, Stream output)
     4. {
     5. **// simulate an exception caused by unanticipated conditions**
     6. **if (input.Length > 768000)**
     7. **{**
     8. **throw new InvalidOperationException("Cannot handle image files larger than 750KB.");**
     9. **}**
     10. int width;
     11. int height;
     12. var originalImage = new Bitmap(input);
     13. if (originalImage.Width > originalImage.Height)
     14. {
     15. ...
     16. }
     17. (Code Snippet – *AsynchWorkloadHandling-Ex1-ProcessImage unhandled exception-VB*)
     18. Visual Basic
     19. Private Sub ProcessImage(ByVal input As Stream, ByVal output As Stream)
     20. **' simulate an exception caused by unanticipated conditions**
     21. **If input.Length > 768000 Then**
     22. **Throw New InvalidOperationException("Cannot handle image files larger than 750KB.")**
     23. **End If**
     24. Dim width As Integer
     25. Dim height As Integer
     26. Dim originalImage As New Bitmap(input)
     27. If originalImage.Width > originalImage.Height Then
     28. ...
     29. End Sub
     30. **Note:** Later in the exercise, you will test the application with several image files with sizes that are both below and above the limit that will cause an exception. You may want to adjust this limit to fit the set of files that you intend to use for this test. The chosen value is adequate for the files contained in the *Sample Pictures* folder available in most Windows 7 installations.
  3. Make sure the **GuestBook** project is set as the start-up project by right-clicking the project node in **Solution Explorer** and selecting **Set as StartUp Project**. Then, press **CTRL+F5** to build and run the application.
     1. **Note:** This action launches the application in the Windows Azure Emulator but does not attach the debugger. This is necessary to avoid having the debugger catch the unhandled exception in the worker role.
  4. Open the Compute Emulator UI. To do this, right-click its icon located in the system tray and select **Show Compute Emulator UI**.
     1. 
     2. Figure 9
     3. Showing the compute emulator UI
  5. Switch to the browser window to view the GuestBook application. To add a new entry, type your name and a message, and then choose an image to upload. Pick any JPEG image file whose size is smaller than the limit chosen to trigger the unhandled exception (750KB) and then submit the entry.
     1. As expected, once you insert an entry, the web role creates a new record in the guest book table and uploads the image to blob storage. Initially, the new entry contains a link to the blob that stores the uploaded image so it will appear with the same size as the original image. Eventually, the worker role retrieves the queued message, proceeds to create a thumbnail for the image, and stores it in blob storage. It then updates the entry to point to the image thumbnail instead. A timer on the page refreshes the page every few seconds and shows the thumbnail soon after it is generated.
     2. 
     3. Figure 10
     4. GuestBook application showing the new entry
  6. Switch to the compute emulator UI and examine the log produced by the worker role. Notice that the role processed the image that you submitted and generated a thumbnail.
     1. 
     2. Figure 11
     3. Worker role diagnostic log showing a message processed successfully
  7. Now, add a second entry, except this time, choose an image large enough to cause the exception. Notice that soon after you submit the entry, Windows shows a message box indicating that the worker role process, **WaWorkerHost.exe**, has crashed.
     1. 
     2. Figure 12
     3. Windows unhandled exception message
  8. Click the **Cancel** button in the message box and, as soon as you do this, observe the status of the worker role in the Compute Emulator UI. Notice that after you dismiss the message box, the worker role status changes *briefly* from **Started** to **Destroyed**. If you examine the contents of the diagnostics log, you will see information about the unhandled exception.
     1. 
     2. Figure 13
     3. Compute Emulator UI showing the crashed worker role
  9. Soon after, the compute emulator restarts the worker role instance and the status indicator turns green. Observe the diagnostics log and notice the sequence of states that the worker role goes through following the unhandled exception as the role shuts down, restarts, and then begins to listen for queue messages again.
     1. 
     2. Figure 14
     3. Diagnostics log showing the worker role restart
  10. However, soon after the restart, the message reappears in the queue after its visibility timeout expires. The worker role retrieves the message once again, processes it, and a new unhandled exception occurs. This results in a second message box from Windows indicating that the worker role has crashed again. Click **Cancel** to allow the worker role to restart one more time.
  11. Note that following the second restart, the worker role no longer crashes. Review its event log and notice that the role has moved the offending message to the poison messages queue and is now ready to accept new jobs. You can confirm this by submitting a new entry with a smaller image that does not trigger the exception. Be aware, however, that the guest book entry that caused the crash continues to show the image in its original size because the worker role could not process it.
      1. 
      2. Figure 15
      3. Diagnostics log showing a message moved to the poison messages queue
      4. **Note:** Immediately after retrieving a message from the queue, the worker role checks its dequeue count and, if the count exceeds the configured limit currently set to 3 in the service configuration file, it moves the message to the poison messages queue without processing it.
  12. To examine the contents of the poison messages queue, open a browser window and navigate to <http://127.0.0.1:81/ManageQueue.aspx>. The page lists the messages currently in the queue. If you followed the previous steps, there should be at least one message.
      1. 
      2. Figure 16
      3. Managing the poison messages queue
      4. **Note:** You may need to adjust the port number in the address of the ManageQueue page if the compute emulator launches the Web role using a different port value.
  13. Select the message in the queue and then click **Re-insert** to remove the message from the poison messages queue and insert it back into its original queue. Because the code that processes the message has not changed, the message will once again cause an unhandled exception and you will observe the worker role crash another time. Allow the worker role to restart a couple of times until it moves the message back into the poison messages queue. In the next exercise, you will make changes to the worker role that will enable you to update the running code to fix the problem and allow you to re-submit the message in the poison messages queue successfully without shutting down the worker role.
  14. Close the browser window and stop the running application in the compute emulator. To do this, select the deployment node in the **Service Deployments** tree view and then click the **Remove service deployments** button on the toolbar.
      1. 
      2. Figure 17
      3. Removing a running deployment from the compute emulator
      4. **Note:** If you keep the browser window open after removing the deployment, you may observe errors as the page script attempts to refresh the page while the application is not running. Close the browser window.

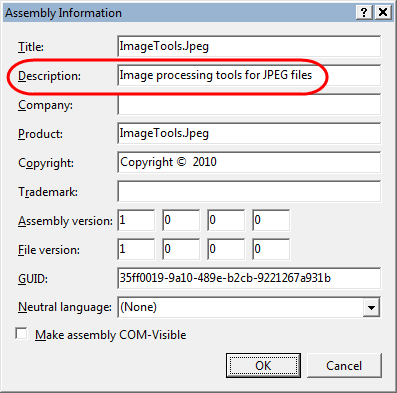
Exercise 2: Dynamically Loading Types to Process Images

A challenge when using worker roles to handle asynchronous processing is that very often, you may not have enough work of a given type to consume a worker role completely—it is desirable to run worker roles at 100% utilization. It is also difficult to reconfigure a worker role to handle different types of work without completely re-deploying the role to place new assemblies on the server.

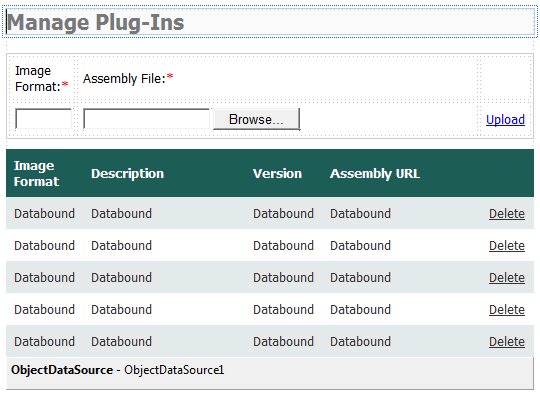
1. In this exercise, you will learn how to create a simple plug-in based approach for Azure worker roles.   
   In order to resize different types of images, you will implement a mechanism whereby the worker role determines the type of image from the queued message. The worker will then load the appropriate type to resize that image from blob storage. The type will be loaded into a new application domain (AppDomain). This mechanism not only allows a plug-in assembly to be replaced without restarting the worker role, but it also allows new plug-ins to be deployed dynamically for additional processing tasks.

Task 1 – Creating a Plug-in Model for Image Processing

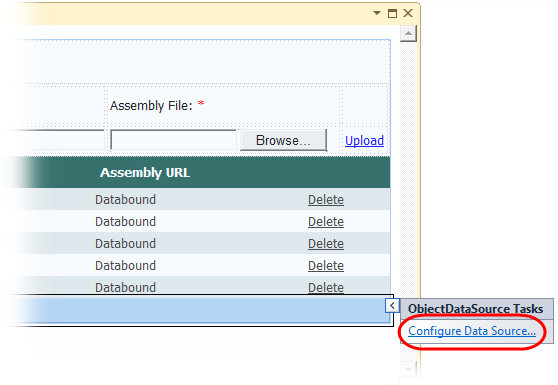
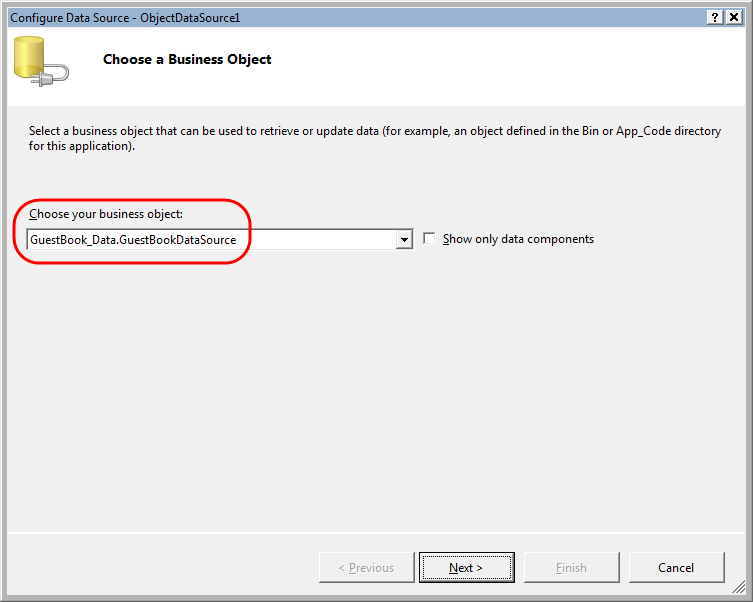
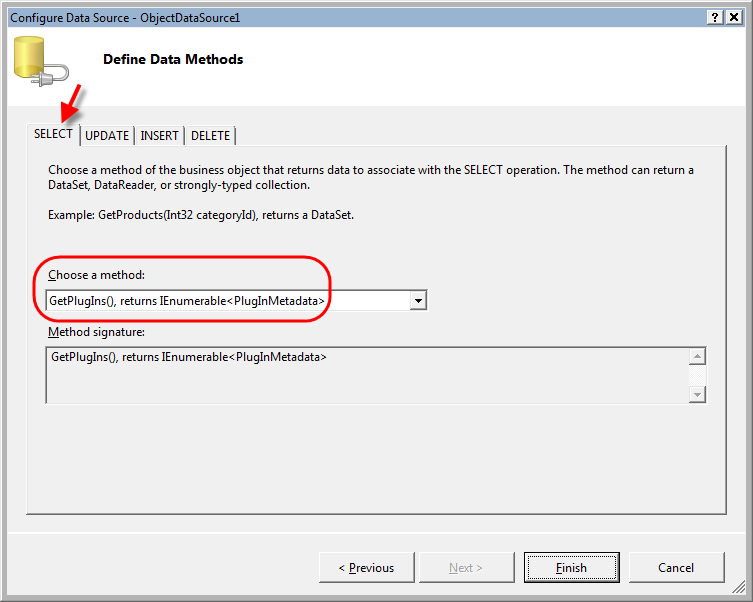
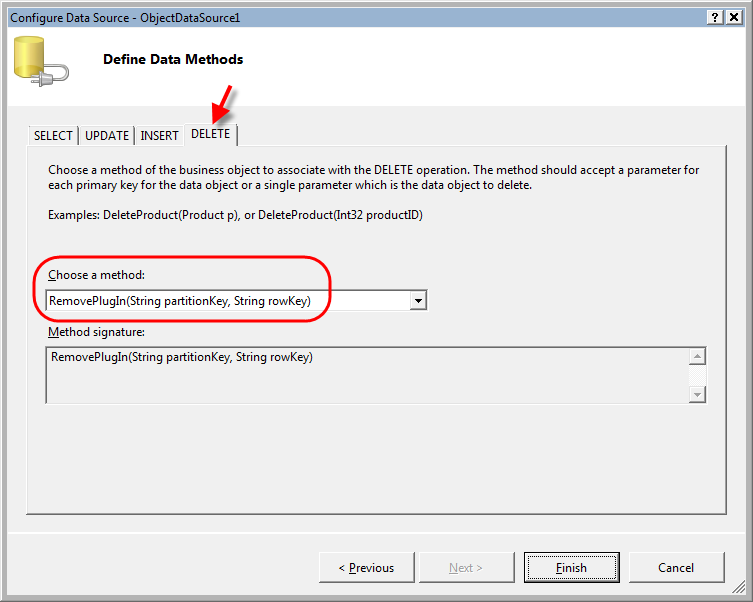
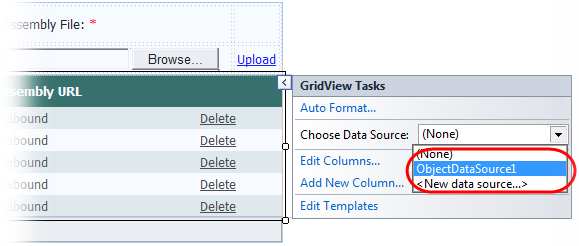
In this task, you create an object model for image processing plug-ins. Plug-ins types implement a well-known .NET interface.

* 1. If not already open, launch Visual Studioin elevated administrator mode from **Start | All Programs | Microsoft Visual Studio 2010** by right-clicking the **Microsoft Visual Studio 2010** shortcut and choosing **Run as administrator**.
  2. If the **User Account Control** dialog appears, click **Continue**.
  3. In the **File** menu, choose **Open** and then **Project/Solution**. In the **Open Project** dialog, browse to **Ex2-DynamicTypeLoading** in the **Source** folder of the lab and choose the folder for the language of your preference (Visual C# or Visual Basic). Select **Begin.sln** in the **Begin** folder and then click **Open**.
  4. Create a project for the contract that an image processing plug-in needs to implement. To create the project, in the **File** menu, point to **Add** and then select **New Project**.
  5. In the **Add New Project** dialog, expand the language of your choice under the **Installed Templates** tree view, select the **Windows** category, and then choose the **Class Library** project template. Set the name to **ImageTools**, leave the proposed location inside the solution folder unchanged, and then click **OK**.
     1. **Note:** The contract for plug-ins is required by every plug-in project, in addition to the worker role project. Therefore, it is best to create a separate assembly to hold the contract and then reference it in each of the projects that requires it.
  6. Delete the **Class1.cs** file (for Visual C# projects) or **Class1.vb** file (for Visual Basic projects) generated by the template. It is not required.
  7. Add a new interface definition file to the plug-in contract project. To do this, right-click **ImageTools** in **Solution Explorer**, point to **Add**, and then select **New Item**. In the **Add New Item** dialog, select the **Interface** template inside the **Code** category, change the name to **IImageProcessor.cs** or **IImageProcessor.vb**, depending on the language of your project, and then click **Add**.
  8. Replace the entire contents of the newly created file with the following code.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-IImageProcessor interface-CS*)
     2. C#
     3. **namespace ImageTools**
     4. **{**
     5. **using System.IO;**
     6. **public interface IImageProcessor**
     7. **{**
     8. **void ProcessImage(Stream input, Stream output);**
     9. **}**
     10. **}**
     11. (Code Snippet – *AsynchWorkloadHandling-Ex2-IImageProcessor interface-VB*)
     12. Visual Basic
     13. **Imports System.IO**
     14. **Public Interface IImageProcessor**
     15. **Sub ProcessImage(ByVal input As Stream, ByVal output As Stream)**
     16. **End Interface**
     17. **Note:** The **IImageProcessor** interface defines a single **ProcessImage** method that receives an input stream containing image data, and an output stream to which it writes the result of the processing.
  9. Now, create a plug-in to process JPEG image files. To create a project for the plug-in, in the **File** menu, point to **Add** and then select **New Project**.
  10. In the **Add New Project** dialog, expand the language of your choice under the **Installed Templates** tree view, select the **Windows** category, and then choose the **Class Library** project template. Set the name to **ImageTools.Jpeg**, leave the proposed location inside the solution folder unchanged, and then click **OK**.
  11. In the new class library project, add a reference to the plug-in contract assembly. To add the reference, right-click the **ImageTools.Jpeg** project in **Solution Explorer**, and then select **Add Reference**. In the **Add Reference** dialog, switch to the **Projects** tab, select the **ImageTools** project, and then click **OK**.
  12. Repeat the previous step to add a reference to the **System.Drawing** assembly, only this time select the assembly in the **.NET** tab of the **Add Reference** dialog.
  13. Now, right-click the **Class1.(cs|.vb)** file in **Solution Explorer** and select **Rename**. Type **ImageProcessor.cs** (for Visual C# projects) or **ImageProcessor.vb** (for Visual Basic projects), and then press **Enter**. When prompted by Visual Studio whether to rename all references to **Class1** in the project, click **Yes**.
  14. Add the following namespace directives at the top of the file, immediately below any existing namespace declarations.
      1. (Code Snippet – *AsynchWorkloadHandling-Ex2-JPEG ImageProcessor namespaces-CS*)
      2. C#
      3. **using System.IO;**
      4. **using System.Drawing;**
      5. **using System.Drawing.Drawing2D;**
      6. **using System.Drawing.Imaging;**
      7. **using ImageTools;**
      8. (Code Snippet – *AsynchWorkloadHandling-Ex2-JPEG ImageProcessor namespaces-VB*)
      9. Visual Basic
      10. **Imports System.IO**
      11. **Imports System.Drawing**
      12. **Imports System.Drawing.Drawing2D**
      13. **Imports System.Drawing.Imaging**
      14. **Imports ImageTools**
  15. Ensure that the visibility of the **ImageProcessor** class is set as public and update its declaration so that it derives from **MarshalByRefObject** and implements the **IImageProcessor** interface, as shown below.
      1. C#
      2. namespace ImageTools.Jpeg
      3. {
      4. **public** class ImageProcessor
      5. : MarshalByRefObject, IImageProcessor
      6. {
      7. }
      8. }
      9. Visual Basic
      10. Public Class ImageProcessor
      11. **Inherits MarshalByRefObject**
      12. **Implements IImageProcessor**
      13. End Class
      14. **Note:** In order to communicate with the plug-in across the boundary of the application domain, it is necessary to use a proxy to exchange messages using remoting mechanisms. To enable this communication, the class that implements the plug-in object must inherit from [MarshalByRefObject](http://msdn.microsoft.com/en-us/library/system.marshalbyrefobject.aspx).
  16. Next, to implement the **IImageProcessor** interface, open the **WorkerRole.cs** file (for Visual C# projects) or **WorkerRole.vb** file (for Visual Basic projects) and locate the **ProcessImage** method. Select the entire method and press **CTRL + X** to remove it and place it into the clipboard. Now, switch back to the **ImageProcessor** class and press **CTRL + V** to paste the contents of the clipboard inside the **ImageProcessor** class.
      1. **Note:** You may need to replace the existing method in the destination class if Visual Studio auto-completed the interface when you declared the class.
  17. For Visual Basic projects, update the method signature to specify that it implements **IImageProcessor.ProcessImage** interface member, as shown below.
      1. Visual Basic
      2. Private Sub ProcessImage(ByVal input As Stream, ByVal output As Stream) **Implements IImageProcessor.ProcessImage**
      3. ...
      4. End Sub
  18. Finally, update the assembly information to provide a description of the assembly for display in the plug-in management UI. To do this, right-click the **ImageTools.Jpeg** project in **Solution Explorer**, choose **Properties** and then select the **Application** page. In this page, click **Assembly Information**, type a descriptive text in the **Description** field and then click **OK**.
      1. 
      2. Figure 18
      3. Setting the description custom attribute of the plug-in assembly

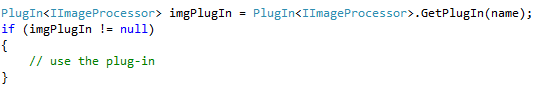
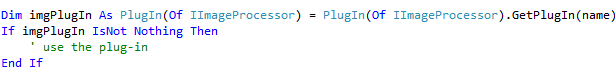
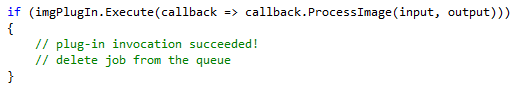
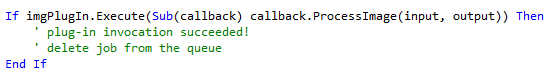
Task 2 – Managing and Deploying Plug-in Assemblies

* 1. In this task, you create a simple form to upload plug-in assemblies into blob storage and register its associated metadata. For simplicity, the begin solution already contains a page with the necessary UI elements. You simply update the code-behind file to implement the required functionality.
  2. Open the provided page template for managing plug-ins. In **Solution Explorer**, right-click **ManagePlugIns.aspx** in the **GuestBook\_WebRole** project and select **View Designer**.
     1. Notice that the page contains a **GridView** control already configured to show information about registered plug-ins, namely *Image Format*, *Description*, *Version*, and *Assembly URL*, as well as a **LinkButton** control to unregister the plug-in and remove it from blob storage. The page also contains an **ObjectDataSource** control that you will bind to the **GridView** to populate it. You will define its data context object shortly.
     2. 
     3. Figure 19
     4. Design view of the plug-in management page
     5. **Note:** The designer view may be disabled by default depending on your Visual Studio settings. To enable design view in Visual Studio, go to **Options | HTML Designer**, select **Enable HTML Designer**, and then restart Visual Studio.
  3. The **GridView** is bound to an **ObjectDataSource** control on the page that retrieves plug-in metadata from blob storage. You will now update the source object that you will bind to the **ObjectDataSource** to provide the plug-in management UI with the methods required to administer plug-ins. To do this, open **GuestBookDataSource.cs** (for Visual C# projects) or **GuestBookDataSource.vb** (for Visual Basic projects) in the **GuestBookData** project.
  4. To define a class to encapsulate plug-in metadata retrieved from blob storage, add a nested **PlugInMetadata** class to the **GuestBookDataSource** class, as shown (highlighted) below.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugInMetadata class-CS*)
     2. C#
     3. public class GuestBookDataSource
     4. {
     5. ...
     6. **public class PlugInMetadata**
     7. **{**
     8. **public string Name { get; set; }**
     9. **public string Description { get; set; }**
     10. **public string AssemblyUrl { get; set; }**
     11. **public string Version { get; set; }**
     12. **}**
     13. }
     14. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugInMetadata class-VB*)
     15. Visual Basic
     16. Public Class GuestBookDataSource
     17. ...
     18. **Public Class PlugInMetadata**
     19. **Public Property Name() As String**
     20. **Public Property Description() As String**
     21. **Public Property AssemblyUrl() As String**
     22. **Public Property Version() As String**
     23. **End Class**
     24. End Class
     25. The **PlugInMetadata** class exposes the properties contained in the metadata of the blob that stores a plug-in assembly. These are:

|  |  |
| --- | --- |
| Field | Description |
| Name | Identifies the format of the image processed by the plug-in (e.g. PNG, JPG, etc). |
| Description | Descriptive text retrieved from the AssemblyDescription attribute of the plug-in assembly and shown in the plug-in management UI. |
| AssemblyUrl | URL of the blob that stores the plug-in assembly. |
| Version | Version of the plug-in retrieved from its assembly metadata attributes. |

* 1. Next, add a method to retrieve a list of plug-ins stored in blob storage, as shown in the following (highlighted) code snippet.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-GetPlugIns method-CS*)
     2. C#
     3. public class GuestBookDataSource
     4. {
     5. ...
     6. **public IEnumerable<PlugInMetadata> GetPlugIns()**
     7. **{**
     8. **CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();**
     9. **CloudBlobDirectory assemblies = blobClient.GetBlobDirectoryReference("assemblies");**
     10. **BlobRequestOptions options = new BlobRequestOptions() { BlobListingDetails = BlobListingDetails.Metadata, UseFlatBlobListing = true };**
     11. **return from CloudBlob blob in assemblies.ListBlobs(options)**
     12. **select new PlugInMetadata**
     13. **{**
     14. **Name = blob.Metadata["Name"],**
     15. **Description = blob.Metadata["Description"],**
     16. **AssemblyUrl = blob.Uri.AbsoluteUri,**
     17. **Version = blob.Metadata["Version"]**
     18. **};**
     19. **}**
     20. ...
     21. }
     22. (Code Snippet – *AsynchWorkloadHandling-Ex2-GetPlugIns method-VB*)
     23. Visual Basic
     24. Public Class GuestBookDataSource
     25. ...
     26. **Public Function GetPlugIns() As IEnumerable(Of PlugInMetadata)**
     27. **Dim blobClient As CloudBlobClient = storageAccount.CreateCloudBlobClient()**
     28. **Dim assemblies As CloudBlobDirectory = blobClient.GetBlobDirectoryReference("assemblies")**
     29. **Dim options As New BlobRequestOptions() With {.BlobListingDetails = BlobListingDetails.Metadata, .UseFlatBlobListing = True}**
     30. **Return From blob As CloudBlob In assemblies.ListBlobs(options) \_**
     31. **Select New PlugInMetadata With {.Name = blob.Metadata("Name"), .Description = blob.Metadata("Description"), .AssemblyUrl = blob.Uri.AbsoluteUri, .Version = blob.Metadata("Version")}**
     32. **End Function**
     33. ...
     34. End Class
     35. **Note:** The preceding method uses the storage client to enumerate the contents of the *assemblies* blob container, retrieving information about the stored assemblies from the metadata of each blob it contains.
  2. Now, add a method to remove a plug-in from blob storage, as shown (highlighted) below.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-RemovePlugIn method-CS*)
     2. C#
     3. public class GuestBookDataSource
     4. {
     5. ...
     6. **public void RemovePlugIn(string assemblyUrl)**
     7. **{**
     8. **CloudBlobClient blobStorage = storageAccount.CreateCloudBlobClient();**
     9. **blobStorage.GetBlobReference(assemblyUrl).Delete();**
     10. **}**
     11. ...
     12. }
     13. (Code Snippet – *AsynchWorkloadHandling-Ex2-RemovePlugIn method-VB*)
     14. Visual Basic
     15. Public Class GuestBookDataSource
     16. ...
     17. **Public Sub RemovePlugIn(ByVal assemblyUrl As String)**
     18. **Dim blobStorage As CloudBlobClient = storageAccount.CreateCloudBlobClient()**
     19. **blobStorage.GetBlobReference(assemblyUrl).Delete()**
     20. **End Sub**
     21. ...
     22. End Class
  3. Finally, to complete the updates to the **GuestBookDataSource** class, update the class to create a container in blob storage to store plug-in assemblies. To do this, add the following (highlighted) code at the end of the type constructor.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-GuestBookDataSource type constructor-CS*)
     2. C#
     3. public class GuestBookDataSource
     4. {
     5. ...
     6. static GuestBookDataSource()
     7. {
     8. storageAccount = CloudStorageAccount.FromConfigurationSetting("DataConnectionString");
     9. CloudTableClient.CreateTablesFromModel(
     10. typeof(GuestBookDataContext),
     11. storageAccount.TableEndpoint.AbsoluteUri,
     12. storageAccount.Credentials);
     13. **CloudBlobClient blobStorage = storageAccount.CreateCloudBlobClient();**
     14. **CloudBlobContainer container = blobStorage.GetContainerReference("assemblies");**
     15. **container.CreateIfNotExist();**
     16. }
     17. ...
     18. }
     19. (Code Snippet – *AsynchWorkloadHandling-Ex2-GuestBookDataSource type constructor-VB*)
     20. Visual Basic
     21. Public Class GuestBookDataSource
     22. ...
     23. Shared Sub New()
     24. storageAccount = CloudStorageAccount.FromConfigurationSetting("DataConnectionString")
     25. CloudTableClient.CreateTablesFromModel(GetType(GuestBookDataContext), storageAccount.TableEndpoint.AbsoluteUri, storageAccount.Credentials)
     26. **Dim blobStorage As CloudBlobClient = storageAccount.CreateCloudBlobClient()**
     27. **Dim container As CloudBlobContainer = blobStorage.GetContainerReference("assemblies")**
     28. **container.CreateIfNotExist()**
     29. End Sub
     30. ...
     31. End Class
  4. Build the **GuestBookDataSource** project to ensure that the updated methods in the class are available to configure the **ObjectDataSource** control. To do this, right-click the project in **Solution Explorer** and select **Build**.
     1. **Note:** You only build the **GuestBookDataSource** project for now. If you build the entire solution at this stage, you will see compilation errors because some of the changes to the worker role are still incomplete.
  5. Return to the **ManagePlugIns.aspx** page. If necessary, switch to design view, and then click the smart tag in the upper right corner of the **ObjectDataSource** control to open the **ObjectDataSource Tasks** menu and then select **Configure Data Source**.
     1. 
     2. Figure 20
     3. Configuring the data source for the plug-ins grid control
  6. In the **Choose a Business Object** page of the **Configure Data Source** wizard, expand the list labeled **Choose your business object**, select **GuestBook\_Data.GuestBookDataSource** from the drop down list of data types, and then click **Next**.
     1. 
     2. Figure 21
     3. Choosing a business object for the data source in the plug-in management page
  7. In the **Define Data Methods** page, choose the **SELECT** method, and then pick **GetPlugIns** from the drop down list labeled **Choose a method**.
     1. 
     2. Figure 22
     3. Configuring the select method to retrieve plug-in data from the data source
  8. Next, select the **DELETE** method and choose **RemovePlugIn** from the drop down list that lists the methods in the business object.
     1. 
     2. Figure 23
     3. Configuring the method to delete a registered plug-in
  9. Click **Finish** to close the wizard.
  10. Now, click the smart tag in the upper right corner of the **GridView** control to open the **GridView Tasks** menu and then select **ObjectDataSource1** in the drop down list labeled **Choose Data Source**.
      1. 
      2. Figure 24
      3. Configuring a data source for the plug-ins grid
  11. After you select a data source, Visual Studio prompts you to refresh the column fields of the **GridView** using the schema of the data source. Answer **No** to this question; otherwise, you will lose the existing column definitions in the page.
  12. To complete the page, create a method to handle the event raised when a new plug-in is uploaded. To create the handler, double-click the **Upload** button in design view. When you do this, Visual Studio opens the code-behind file and positions the cursor on the newly inserted event handler. Insert the following (highlighted) code into the body of the **uploadPlugIn\_Click** event handler.
      1. (Code Snippet – *AsynchWorkloadHandling-Ex2-uploadPlugIn\_Click event handler-CS*)
      2. C#
      3. protected void uploadPlugIn\_Click(object sender, EventArgs e)
      4. {
      5. **if (this.fileUpload.HasFile)**
      6. **{**
      7. **// create container for assemblies, if it does not exist**
      8. **CloudStorageAccount account = CloudStorageAccount.FromConfigurationSetting("DataConnectionString");**
      9. **CloudBlobClient blobStorage = account.CreateCloudBlobClient();**
      10. **CloudBlobContainer container = blobStorage.GetContainerReference("assemblies");**
      11. **container.CreateIfNotExist();**
      12. **// upload assembly to blob storage and save its metadata**
      13. **string imageFormat = this.imageFormat.Text.ToLower();**
      14. **CloudBlob blob = container.GetBlobReference(imageFormat + "/" + this.fileUpload.FileName);**
      15. **blob.UploadByteArray(this.fileUpload.FileBytes);**
      16. **SaveMetadata(blob, imageFormat);**
      17. **this.Response.Redirect(this.Request.Path);**
      18. **}**
      19. }
      20. (Code Snippet – *AsynchWorkloadHandling-Ex2-uploadPlugIn\_Click event handler-VB*)
      21. Visual Basic
      22. Protected Sub uploadPlugIn\_Click(ByVal sender As Object, ByVal e As EventArgs) Handles uploadPlugIn.Click
      23. **If Me.fileUpload.HasFile Then**
      24. **' create container for assemblies, if it does not exist**
      25. **Dim account As CloudStorageAccount = CloudStorageAccount.FromConfigurationSetting("DataConnectionString")**
      26. **Dim blobStorage As CloudBlobClient = account.CreateCloudBlobClient()**
      27. **Dim container As CloudBlobContainer = blobStorage.GetContainerReference("assemblies")**
      28. **container.CreateIfNotExist()**
      29. **' upload assembly to blob storage and save its metadata**
      30. **Dim imageFormat As String = Me.imageFormat.Text.ToLower()**
      31. **Dim blob As CloudBlob = container.GetBlobReference(imageFormat & "/" & Me.fileUpload.FileName)**
      32. **blob.UploadByteArray(Me.fileUpload.FileBytes)**
      33. **SaveMetadata(blob, imageFormat)**
      34. **Me.Response.Redirect(Me.Request.Path)**
      35. **End If**
      36. End Sub
      37. **Note:** The code shown above first ensures that the *assemblies* blob container exists and then uses the storage client library to upload the assembly into a blob inside this container. It then calls the **SaveMetadata** method to retrieve the description and version of the assembly and store them as attributes in the blob’s metadata. The plug-in management UI uses this information to identify each plug-in registered in the system.
  13. Now, add a method to populate the metadata of the blob with the attributes retrieved from the uploaded assembly, as shown in the following (highlighted) code.
      1. (Code Snippet – *AsynchWorkloadHandling-Ex2-SaveMetadata method-CS*)
      2. C#
      3. **private void SaveMetadata(CloudBlob blob, string imageFormat)**
      4. **{**
      5. **// load assembly in secondary appdomain to retrieve its metadata**
      6. **AppDomainSetup setup = new AppDomainSetup() { ApplicationBase = AppDomain.CurrentDomain.BaseDirectory + "\\bin" };**
      7. **AppDomain domain = AppDomain.CreateDomain("ReflectionOnly", null, setup);**
      8. **domain.ReflectionOnlyAssemblyResolve += (sender, args) => Assembly.ReflectionOnlyLoad(args.Name);**
      9. **domain.SetData("fileContent", this.fileUpload.FileContent);**
      10. **domain.DoCallBack(LoadAssembly);**
      11. **// set properties and metadata**
      12. **string description = (string)domain.GetData("description");**
      13. **string version = (string)domain.GetData("version");**
      14. **blob.Metadata.Add("Name", imageFormat);**
      15. **blob.Metadata.Add("Description", String.IsNullOrEmpty(description) ? "Missing assembly description" : description);**
      16. **blob.Metadata.Add("Version", String.IsNullOrEmpty(version) ? "Missing assembly version" : version);**
      17. **blob.SetMetadata();**
      18. **blob.Properties.ContentType = "application/x-msdownload";**
      19. **blob.SetProperties();**
      21. **// unload the assembly from memory**
      22. **AppDomain.Unload(domain);**
      23. **}**
      24. (Code Snippet – *AsynchWorkloadHandling-Ex2-SaveMetadata method-VB*)
      25. Visual Basic
      26. **Private Sub SaveMetadata(ByVal blob As CloudBlob, ByVal imageFormat As String)**
      27. **' load assembly in secondary appdomain to retrieve its metadata**
      28. **Dim setup As New AppDomainSetup() With {.ApplicationBase = AppDomain.CurrentDomain.BaseDirectory & "\bin"}**
      29. **Dim domain As AppDomain = AppDomain.CreateDomain("ReflectionOnly", Nothing, setup)**
      30. **AddHandler domain.ReflectionOnlyAssemblyResolve, Function(sender, args) System.Reflection.Assembly.ReflectionOnlyLoad(args.Name)**
      31. **domain.SetData("fileContent", Me.fileUpload.FileContent)**
      32. **domain.DoCallBack(AddressOf LoadAssembly)**
      33. **' set properties and metadata**
      34. **Dim description As String = CType(domain.GetData("description"), String)**
      35. **Dim version As String = CType(domain.GetData("version"), String)**
      36. **blob.Metadata.Add("Name", imageFormat)**
      37. **blob.Metadata.Add("Description", If(String.IsNullOrEmpty(description), "Missing assembly description", description))**
      38. **blob.Metadata.Add("Version", If(String.IsNullOrEmpty(version), "Missing assembly version", version))**
      39. **blob.SetMetadata()**
      40. **blob.Properties.ContentType = "application/x-msdownload"**
      41. **blob.SetProperties()**
      42. **' unload the assembly from memory**
      43. **AppDomain.Unload(domain)**
      44. **End Sub**
      45. **Note:** The code creates a secondary application domain and then loads the assembly into this domain in the reflection-only context to obtain its **AssemblyDescription** and **Version** attributes and then saves the obtained values as custom metadata properties of the blob where the assembly is stored. Lastly, the code unloads the application domain to remove the plug-in assembly from memory
  14. Finally, add the following method to load the assembly and retrieve its metadata.
      1. (Code Snippet – *AsynchWorkloadHandling-Ex2-LoadAssembly method-CS*)
      2. C#
      3. **private static void LoadAssembly()**
      4. **{**
      5. **Stream stream = (Stream)AppDomain.CurrentDomain.GetData("fileContent");**
      6. **byte[] rawAssembly = new byte[stream.Length];**
      7. **stream.Read(rawAssembly, 0, (int)stream.Length);**
      8. **Assembly pluginAssembly = Assembly.ReflectionOnlyLoad(rawAssembly);**
      9. **var assemblyDescription = CustomAttributeData.GetCustomAttributes(pluginAssembly)**
      10. **.Where(a => a.Constructor.ReflectedType == typeof(AssemblyDescriptionAttribute))**
      11. **.FirstOrDefault();**
      12. **AppDomain.CurrentDomain.SetData("description", assemblyDescription.ConstructorArguments[0].Value);**
      13. **AppDomain.CurrentDomain.SetData("version", pluginAssembly.GetName().Version.ToString());**
      14. **}**
      15. (Code Snippet – *AsynchWorkloadHandling-Ex2-LoadAssembly method-VB*)
      16. Visual Basic
      17. **Private Shared Sub LoadAssembly()**
      18. **Dim stream As Stream = CType(AppDomain.CurrentDomain.GetData("fileContent"), Stream)**
      19. **Dim rawAssembly(stream.Length - 1) As Byte**
      20. **stream.Read(rawAssembly, 0, CInt(stream.Length))**
      21. **Dim pluginAssembly As Assembly = Assembly.ReflectionOnlyLoad(rawAssembly)**
      22. **Dim assemblyDescription = CustomAttributeData.GetCustomAttributes(pluginAssembly).Where(Function(a) a.Constructor.ReflectedType Is GetType(AssemblyDescriptionAttribute)).FirstOrDefault()**
      23. **AppDomain.CurrentDomain.SetData("description", assemblyDescription.ConstructorArguments(0).Value)**
      24. **AppDomain.CurrentDomain.SetData("version", pluginAssembly.GetName().Version.ToString())**
      25. **End Sub**
      26. **Note:** The code loads the assembly from the stream of bytes uploaded by the user. It then uses reflection to retrieve the **AssemblyDescription** and **Version** attributes from its metadata and stores them as domain properties so that they accessible from the primary domain.

Task 3 – Dynamically Loading Types from Blob Storage

* 1. Potentially, each image type can be processed by a different plug-in. To manage plug-ins, you now create a class that fetches plug-in assemblies from blob storage, loads them into their own separate application domain for isolation, and allows their invocation from the application domain of the worker role. The class you build is a generic class named **PlugIn<T>**, where **T** is a well-known interface implemented by each plug-in.
  2. When using this class, you create a new plug-in instance using the static (Shared in Visual Basic) **GetPlugIn** method and pass it the name of the plug-in that you wish to instantiate. The plug-in name is specified when the plug-in assembly is registered and uploaded to blob storage.
  3. 
  4. Figure 25
  5. Creating a new plug-in instance in Visual C#
  6. 
  7. Figure 26
  8. Creating a new plug-in instance in Visual Basic
  10. The **PlugIn<T>** class exposes an **Execute** method. To use the plug-in, you call this method passing in a callback delegate that receives an instance of the plug-in type **T** as a parameter. When you call **Execute**, it sets up a new application domain and creates new instance of the plug-in data type in this domain, then it invokes the callback and provides it with a proxy to the plug-in object.
  11. For example, for a plug-in that uses the **IImageProcessor** interface, you invoke the plug-in in the following manner:
  12. 
  13. Figure 27
  14. Executing the plug-in code in Visual C#
  15. 
  16. Figure 28
  17. Executing the plug-in code in Visual Basic
  19. For the first call to **Execute**, the class creates a new application domain and loads the plug-in assembly into this domain. Subsequent calls re-use the same domain. When the callback uses the proxy to invoke a method of the plug-in interface, it executes in the application domain hosting the plug-in.
  20. The **Execute** method returns *true* if the call succeeds. The method traps any exceptions that occur during the execution of the callback. After an unhandled exception in the plug-in, the hosting application domain state may be corrupt, so it is unloaded. If that occurs, the call fails and the **Execute** method returns *false*. Note that this only happens for exceptions raised in the calling thread. The runtime does not provide a mechanism to catch unhandled exceptions raised in background threads. When developing plug-ins, you need to protect any background operations—this includes work items in the thread pool—with a backstop handler that catches all exceptions occurring in their respective threads. The plug-in may re-raise the exception on the main thread if it cannot handle it adequately.
  21. Isolating plug-ins and allowing them to be serviced independently of the deployed service package is one of the key reasons for this architecture. When faced with a faulty plug-in, you can deploy an updated assembly that fixes the underlying cause of an unhandled exception and then re-attempt the operation.
  22. To update a plug-in loaded in memory, you call its **Unload** method to discard the application domain where it is currently hosted. You can then re-create the plug-in using the updated assembly. Note that this action aborts any threads that are currently executing in that application domain. This does not represent an issue with the code in this hands-on lab because the worker role processes queue messages synchronously and, if it detects that a plug-ins needs to be refreshed, the worker role unloads it between invocations, when no other threads are executing in the remote domain.

Whenever you request a new plug-in instance using **GetPlugIn**, the method first checks blob storage for updates to the assembly. If it finds an updated plug-in assembly, it discards the application domain currently hosting the plug-in and creates a new application domain to load the updated assembly. Note that this incurs a performance penalty as every call to **GetPlugIn** involves a request to blob storage. If you are creating instances of a plug-in at a high rate, you may want to explore a different mechanism to check for updates.

* 1. To create the plug-in class, right-click the **GuestBook\_WorkerRole** project in **Solution Explorer**, point to **Add**, and then select **Class**. In the **Add New Item** dialog, set the name to **PlugIn.cs** (for Visual C# projects) or **PlugIn.vb** (for Visual Basic projects) and click **Add**.
  2. Add the following namespace directives to the new class file, immediately following the existing namespaces at the top of the file.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class namespaces-CS*)
     2. C#
     3. **using System.Diagnostics;**
     4. **using System.Reflection;**
     5. **using Microsoft.WindowsAzure;**
     6. **using Microsoft.WindowsAzure.ServiceRuntime;**
     7. **using Microsoft.WindowsAzure.StorageClient;**
     8. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class namespaces-VB*)
     9. Visual Basic
     10. **Imports System.Diagnostics**
     11. **Imports System.Reflection**
     12. **Imports Microsoft.WindowsAzure**
     13. **Imports Microsoft.WindowsAzure.ServiceRuntime**
     14. **Imports Microsoft.WindowsAzure.StorageClient**
  3. Change the visibility of the **PlugIn** class to internal (Friend in Visual Basic) and make it generic, with a single type parameter that represents the interface type that a plug-in data type implements, as shown below.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class-CS*)
     2. C#
     3. **internal class PlugIn<T>**
     4. **where T : class**
     5. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class-VB*)
     6. Visual Basic
     7. **Friend Class PlugIn(Of T As Class)**
  4. Create the following members inside the **PlugIn<T>** class. These include a field to store the application domain hosting the plug-in assembly and a shared dictionary to cache currently loaded plug-in instances.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class fields-CS*)
     2. C#
     3. internal class PlugIn<T>
     4. where T : class
     5. {
     6. **private const string PLUGIN\_ASSEMBLYNAME = "#PlugInAssembly#";**
     7. **private AppDomain pluginDomain;**
     8. **private static Dictionary<string, PlugIn<T>> plugins = new Dictionary<string, PlugIn<T>>();**
     9. }
     10. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class fields-VB*)
     11. Visual Basic
     12. Friend Class PlugIn(Of T As Class)
     13. **Private Const PLUGIN\_ASSEMBLYNAME = "#PlugInAssembly#"**
     14. **Private pluginDomain As AppDomain**
     15. **Private Shared plugins As New Dictionary(Of String, PlugIn(Of T))()**
     16. End Class
  5. Now, add the following properties to the **PlugIn<T>** class to expose its metadata.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class properties-CS*)
     2. C#
     3. internal class PlugIn<T>
     4. where T : class
     5. {
     6. ...
     7. **public string Name { get; private set; }**
     8. **public string AssemblyUrl { get; private set; }**
     9. **public DateTime LastModified { get; private set; }**
     10. }
     11. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class properties-VB*)
     12. Visual Basic
     13. Friend Class PlugIn(Of T As Class)
     14. ...
     15. **Public Property Name() As String**
     16. **Public Property AssemblyUrl() As String**
     17. **Public Property LastModified() As DateTime**
     18. End Class
  6. Add a private constructor to initialize the information required to instantiate a plug-in from blob storage.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class constructor-CS*)
     2. C#
     3. internal class PlugIn<T>
     4. where T : class
     5. {
     6. ...
     7. **private PlugIn(string name, string assemblyUrl, DateTime lastModified)**
     8. **{**
     9. **this.Name = name;**
     10. **this.AssemblyUrl = assemblyUrl;**
     11. **this.LastModified = lastModified;**
     12. **}**
     13. }
     14. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn class constructor-VB*)
     15. Visual Basic
     16. Friend Class PlugIn(Of T As Class)
     17. ...
     18. **Private Sub New(ByVal name As String, ByVal assemblyUrl As String, ByVal lastModified As DateTime)**
     19. **Me.Name = name**
     20. **Me.AssemblyUrl = assemblyUrl**
     21. **Me.LastModified = lastModified**
     22. **End Sub**
     23. End Class
  7. Now, add a static (Shared in Visual Basic) method to instantiate new plug-ins.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-GetPlugIn method-CS*)
     2. C#
     3. internal class PlugIn<T>
     4. where T : class
     5. {
     6. ...
     7. **public static PlugIn<T> GetPlugIn(string name)**
     8. **{**
     9. **// retrieve metadata for the blob that contains the plug-in assembly**
     10. **var storageAccount = CloudStorageAccount.FromConfigurationSetting("DataConnectionString");**
     11. **CloudBlobClient blobStorage = storageAccount.CreateCloudBlobClient();**
     12. **CloudBlobDirectory assemblies = blobStorage.GetBlobDirectoryReference("assemblies");**
     13. **BlobRequestOptions options = new BlobRequestOptions() { BlobListingDetails = BlobListingDetails.Metadata, UseFlatBlobListing = true };**
     14. **CloudBlob assemblyBlob = (CloudBlob)assemblies.GetSubdirectory(name).ListBlobs(options).FirstOrDefault();**
     15. **if (assemblyBlob == null)**
     16. **{**
     17. **return null;**
     18. **}**
     19. **DateTime lastModified = assemblyBlob.Properties.LastModifiedUtc;**
     20. **string assemblyUrl = assemblyBlob.Uri.AbsoluteUri;**
     21. **PlugIn<T> plugin;**
     22. **if (plugins.TryGetValue(name, out plugin))**
     23. **{**
     24. **// check if plug-in assembly needs to be refreshed from blob storage**
     25. **if ((lastModified > plugin.LastModified) || (assemblyUrl != plugin.AssemblyUrl))**
     26. **{**
     27. **plugin.Unload();**
     28. **}**
     29. **}**
     30. **if (plugin == null)**
     31. **{**
     32. **plugin = new PlugIn<T>(name, assemblyUrl, lastModified);**
     33. **plugins.Add(name, plugin);**
     34. **}**
     35. **return plugin;**
     36. **}**
     37. }
     38. (Code Snippet – *AsynchWorkloadHandling-Ex2-GetPlugIn method-VB*)
     39. Visual Basic
     40. Friend Class PlugIn(Of T As Class)
     41. ...
     42. **Public Shared Function GetPlugIn(ByVal name As String) As PlugIn(Of T)**
     43. **' retrieve metadata for the blob that contains the plug-in assembly**
     44. **Dim storageAccount = CloudStorageAccount.FromConfigurationSetting("DataConnectionString")**
     45. **Dim blobStorage As CloudBlobClient = storageAccount.CreateCloudBlobClient()**
     46. **Dim assemblies As CloudBlobDirectory = blobStorage.GetBlobDirectoryReference("assemblies")**
     47. **Dim options As New BlobRequestOptions() With {.BlobListingDetails = BlobListingDetails.Metadata, .UseFlatBlobListing = True}**
     48. **Dim assemblyBlob As CloudBlob = CType(assemblies.GetSubdirectory(name).ListBlobs(options).FirstOrDefault(), CloudBlob)**
     49. **If assemblyBlob Is Nothing Then**
     50. **Return Nothing**
     51. **End If**
     52. **Dim lastModified As DateTime = assemblyBlob.Properties.LastModifiedUtc**
     53. **Dim assemblyUrl As String = assemblyBlob.Uri.AbsoluteUri**
     54. **Dim plugin As PlugIn(Of T) = Nothing**
     55. **If plugins.TryGetValue(name, plugin) Then**
     56. **' check if plug-in assembly needs to be refreshed from blob storage**
     57. **If (lastModified > plugin.LastModified) OrElse (assemblyUrl <> plugin.AssemblyUrl) Then**
     58. **plugin.Unload()**
     59. **End If**
     60. **End If**
     61. **If plugin Is Nothing Then**
     62. **plugin = New PlugIn(Of T)(name, assemblyUrl, lastModified)**
     63. **plugins.Add(name, plugin)**
     64. **End If**
     65. **Return plugin**
     66. **End Function**
     67. End Class
     68. **Note:** The **GetPlugIn** method searches blob storage for the blob that contains the requested plug-in assembly. To do this, it assumes that the blob is stored in a subdirectory of the *assemblies* container named after the requested plug-in. If it locates a suitable blob, it retrieves its address **Uri** and **LastModifedUtc** properties.
     69. Next, it checks the plug-in cache to determine if there is already a domain that hosts the requested plug-in. If the cache contains a match, it compares the address URI as well as the **LastUpdated** property of the plug-in in the cache with the **LastModifiedUtc** property retrieved from the metadata in blob storage. If it determines that the plug-in was updated since it was last loaded, it unloads the hosting application domain and removes it from the cache.
     70. If the requested plug-in cannot be retrieved from the cache, a new one is created, stored in the cache, and returned; otherwise, the cached instance is used.
  8. Next, add a private static (Shared in Visual Basic) method to load a plug-in assembly from blob storage.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-LoadPlugInAssemblyFromBlobStorage method-CS*)
     2. C#
     3. internal class PlugIn<T>
     4. where T : class
     5. {
     6. ...
     7. **private static void LoadPlugInAssemblyFromBlobStorage(string assemblyName)**
     8. **{**
     9. **// download assembly from blob storage**
     10. **CloudStorageAccount.SetConfigurationSettingPublisher((configName, configSetter) =>**
     11. **configSetter(RoleEnvironment.GetConfigurationSettingValue(configName)));**
     12. **var storageAccount = CloudStorageAccount.FromConfigurationSetting("DataConnectionString");**
     13. **CloudBlobClient blobStorage = storageAccount.CreateCloudBlobClient();**
     14. **CloudBlob assemblyBlob = blobStorage.GetBlobReference(assemblyName);**
     15. **byte[] rawAssembly = assemblyBlob.DownloadByteArray();**
     16. **Assembly pluginAssembly = Assembly.Load(rawAssembly);**
     17. **AppDomain.CurrentDomain.SetData(PLUGIN\_ASSEMBLYNAME, pluginAssembly);**
     18. **// retrieve plug-in data type**
     19. **Type pluginType = pluginAssembly.GetTypes().Where(t => typeof(T).IsAssignableFrom(t)).FirstOrDefault();**
     20. **if (pluginType != null)**
     21. **{**
     22. **AppDomain.CurrentDomain.SetData("typeName", pluginType.FullName);**
     23. **Trace.TraceInformation("Loaded plug-in assembly: " + pluginAssembly.FullName);**
     24. **}**
     25. **else**
     26. **{**
     27. **Trace.TraceInformation("Assembly '{0}' does not contain a valid plug-in for interface '{1}'." + pluginAssembly.FullName, typeof(T).Name);**
     28. **}**
     29. **}**
     30. }
     31. (Code Snippet – *AsynchWorkloadHandling-Ex2-LoadPlugInAssemblyFromBlobStorage method-VB*)
     32. Visual Basic
     33. Friend Class PlugIn(Of T As Class)
     34. ...
     35. **Private Shared Sub LoadPlugInAssemblyFromBlobStorage(ByVal assemblyUrl As String)**
     36. **' download assembly from blob storage**
     37. **CloudStorageAccount.SetConfigurationSettingPublisher(Sub(configName, configSetter) configSetter(RoleEnvironment.GetConfigurationSettingValue(configName)))**
     38. **Dim storageAccount = CloudStorageAccount.FromConfigurationSetting("DataConnectionString")**
     39. **Dim blobStorage As CloudBlobClient = storageAccount.CreateCloudBlobClient()**
     40. **Dim assemblyBlob As CloudBlob = blobStorage.GetBlobReference(assemblyUrl)**
     41. **Dim rawAssembly() As Byte = assemblyBlob.DownloadByteArray()**
     42. **Dim pluginAssembly As System.Reflection.Assembly = System.Reflection.Assembly.Load(rawAssembly)**
     43. **AppDomain.CurrentDomain.SetData(PLUGIN\_ASSEMBLYNAME, pluginAssembly)**
     44. **' retrieve plug-in data type**
     45. **Dim pluginType As Type = pluginAssembly.GetTypes().Where(Function(t) GetType(T).IsAssignableFrom(t)).FirstOrDefault()**
     46. **If pluginType IsNot Nothing Then**
     47. **AppDomain.CurrentDomain.SetData("typeName", pluginType.FullName)**
     48. **Trace.TraceInformation("Loaded plug-in assembly: " & pluginAssembly.FullName)**
     49. **Else**
     50. **Trace.TraceInformation("Assembly '{0}' does not contain a valid plug-in for interface '{1}'." & pluginAssembly.FullName, GetType(T).Name)**
     51. **End If**
     52. **End Sub**

End Class

**Note:** The **LoadPlugInAssemblyFromBlobStorage** method uses the Storage Client to load assemblies from blob storage. Even though the worker role previously called the **SetConfigurationSettingPublisher** method during its initialization, this is only effective for the current application domain, so this method needs to call it again in the target domain to prepare the Storage Client configuration system.

Next, the method downloads the contents of the blob that contains the plug-in assembly into a byte array, loads the corresponding assembly into the current application domain, and saves a reference to the assembly in the application domain’s properties to provide the application domain’s **AssemblyResolve** event handler with a reference to the plug-in assembly.

Finally, the method uses reflection to search the assembly for a data type that implements the interface of the plug-in assembly (T), and then saves the type name as an application domain property too.

* 1. Insert the following method to set up the plug-in hosting domain, invoke the plug-in and trap any exceptions that result from its execution.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2- PlugIn Execute method-CS*)
     2. C#
     3. internal class PlugIn<T>
     4. where T : class
     5. {
     6. ...
     7. **public bool Execute(Action<T> callback)**
     8. **{**
     9. **try**
     10. **{**
     11. **if (this.pluginDomain == null)**
     12. **{**
     13. **// setup a new application domain and load the plug-in**
     14. **AppDomainSetup setupInfo = new AppDomainSetup()**
     15. **{**
     16. **ApplicationBase = AppDomain.CurrentDomain.BaseDirectory,**
     17. **AppDomainInitializerArguments = new string[] { this.AssemblyUrl },**
     18. **AppDomainInitializer = (args) => LoadPlugInAssemblyFromBlobStorage(args[0])**
     19. **};**
     20. **this.pluginDomain = AppDomain.CreateDomain(this.Name, null, setupInfo);**
     21. **this.pluginDomain.AssemblyResolve += (sender, args) => (Assembly)AppDomain.CurrentDomain.GetData(args.Name);**
     22. **}**
     23. **// create an instance of the plug-in type in the AppDomain**
     24. **string typeName = (string)this.pluginDomain.GetData("typeName");**
     25. **T plugin = this.pluginDomain.CreateInstanceAndUnwrap(PLUGIN\_ASSEMBLYNAME, typeName) as T;**
     26. **if (plugin == null)**
     27. **{**
     28. **throw new InvalidOperationException(String.Format("Plug-in type '{0}' does not implement the required '{1}' interface.", typeName, typeof(T).Name));**
     29. **}**
     30. **callback(plugin);**
     31. **return true;**
     32. **}**
     33. **// handle all exceptions by unloading the AppDomain**
     34. **catch (Exception ex)**
     35. **{**
     36. **Trace.TraceError("Unrecoverable error in plug-in '{0}'.\n{1}", this.Name, ex.Message);**
     37. **Unload();**
     38. **}**
     39. **return false;**
     40. **}**
     41. ...
     42. }
     43. (Code Snippet – *AsynchWorkloadHandling-Ex2- PlugIn Execute method-VB*)
     44. Visual Basic
     45. Friend Class PlugIn(Of T As Class)
     46. ...
     47. **Public Function Execute(ByVal callback As Action(Of T)) As Boolean**
     48. **Try**
     49. **If Me.pluginDomain Is Nothing Then**
     50. **' setup a new application domain and load the plug-in**
     51. **Dim setupInfo As New AppDomainSetup() With { \_**
     52. **.ApplicationBase = AppDomain.CurrentDomain.BaseDirectory, \_**
     53. **.AppDomainInitializerArguments = New String() {Me.AssemblyUrl}, \_**
     54. **.AppDomainInitializer = Sub(args) LoadPlugInAssemblyFromBlobStorage(args(0))}**
     55. **Me.pluginDomain = AppDomain.CreateDomain(Me.Name, Nothing, setupInfo)**
     56. **AddHandler Me.pluginDomain.AssemblyResolve, Function(sender, args) CType(AppDomain.CurrentDomain.GetData(args.Name), Assembly)**
     57. **End If**
     58. **' create an instance of the plug-in type in the AppDomain**
     59. **Dim typeName As String = CType(Me.pluginDomain.GetData("typeName"), String)**
     60. **Dim plugin As T = TryCast(Me.pluginDomain.CreateInstanceAndUnwrap(PLUGIN\_ASSEMBLYNAME, typeName), T)**
     61. **If plugin Is Nothing Then**
     62. **Throw New InvalidOperationException(String.Format("Plug-in type '{0}' does not implement the required '{1}' interface.", typeName, GetType(T).Name))**
     63. **End If**
     64. **callback(plugin)**
     65. **Return True**
     66. **' handle all exceptions by unloading the AppDomain**
     67. **Catch ex As Exception**
     68. **Trace.TraceError("Unrecoverable error in plug-in '{0}'." & Constants.vbLf & "{1}", Me.Name, ex.Message)**
     69. **Unload()**
     70. **End Try**
     71. **Return False**
     72. **End Function**
     73. ...
     74. End Class

**Note:** Several things occur before a plug-in is loaded and executed. First, the **Execute** method verifies whether it needs to set up a new application domain. This happens during the first call to this method or whenever the application domain is unloaded, which could happen as the result of an unhandled exception in the plug-in or the **Unload** method being called. During the initialization of the application domain, the application base of the new domain is set to be the same as the base directory of the current (default) application domain. This ensures that the loader is able to locate assemblies that the secondary domain shares with the default domain. In addition, the domain initialization code attaches an event handler for the **AssemblyResolve** event. The loader raises this event whenever an assembly cannot be located.

The method then retrieves the data type name of the plug-in object from the domain’s properties, where it was stored by the **LoadPlugInAssemblyFromBlobStorage** method when the assembly was initially loaded and searched for the plug-in data type. Next, the code creates an instance of the plug-in data type in the remote domain. Note that the call to **CreateInstanceAndUnwrap** uses the *PLUGIN\_ASSEMBLYNAME* constant instead of the actual name to specify the assembly that contains the plug-in object. When the plug-in assembly was initially loaded, a reference to the assembly was stored in the application domain’s property dictionary under the same name, *PLUGIN\_ASSEMBLYNAME*. When the loader attempts to load an assembly by this name, it ultimately raises the **AssemblyResolve** event. In the handler for this event, any requests for an assembly with this name look up the corresponding value in the application domain’s properties and return the assembly that was loaded from blob storage.

* + 1. Finally, the **Execute** method calls the callback delegate that and passes it the plug-in object instance that it created. This method wraps the execution of the plug-in within a try/catch block that handles all exceptions by unloading the application domain of the plug-in. Note, however, that certain critical exceptions might indicate that the state of the entire process is corrupt and you should consider restarting the worker role.
  1. Finally, to complete the definition of the **PlugIn** class, add a method to remove the plug-in from memory by unloading the application domain that hosts it.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn Unload method-CS*)
     2. C#
     3. internal class PlugIn<T>
     4. where T : class
     5. {
     6. ...
     7. **// NOTE: this will abort any threads currently executing in the appdomain**
     8. **public void Unload()**
     9. **{**
     10. **if (this.pluginDomain != null)**
     11. **{**
     12. **Trace.TraceInformation("Unloading AppDomain for plugin '{0}'.", this.Name);**
     13. **AppDomain.Unload(this.pluginDomain);**
     14. **this.pluginDomain = null;**
     15. **}**
     16. **}**
     17. ...
     18. }
     19. (Code Snippet – *AsynchWorkloadHandling-Ex2-PlugIn Unload method-VB*)
     20. Visual Basic
     21. Friend Class PlugIn(Of T As Class)
     22. ...
     23. **' NOTE: this will abort any threads currently executing in the appdomain**
     24. **Public Sub Unload()**
     25. **If Me.pluginDomain IsNot Nothing Then**
     26. **Trace.TraceInformation("Unloading AppDomain for plugin '{0}'.", Me.Name)**
     27. **AppDomain.Unload(Me.pluginDomain)**
     28. **Me.pluginDomain = Nothing**
     29. **End If**
     30. **End Sub**
     31. ...
     32. End Class

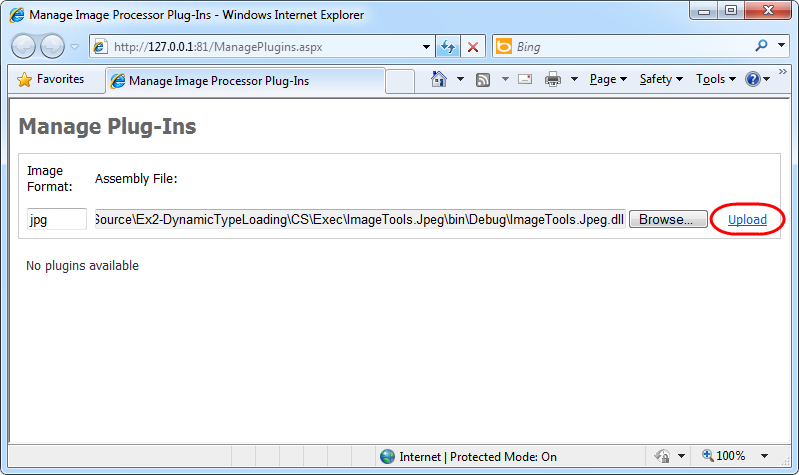
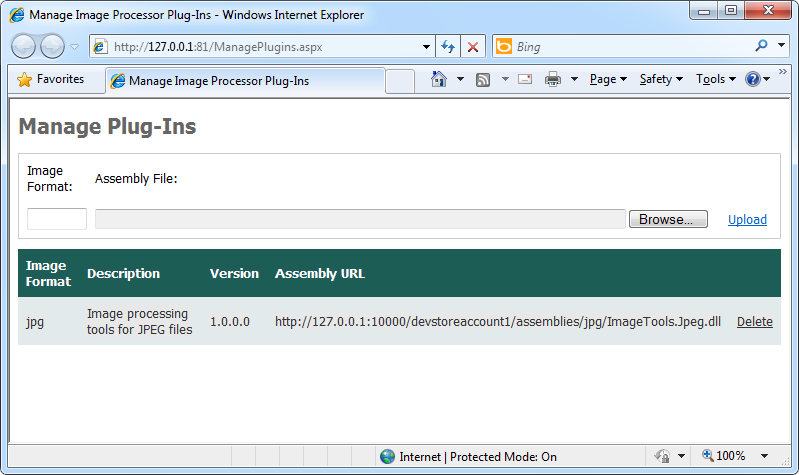
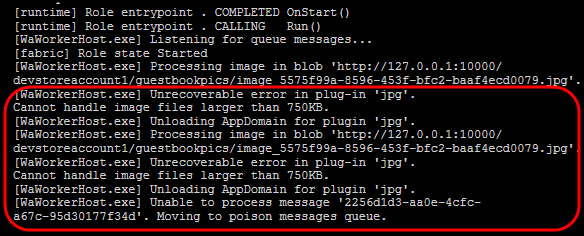
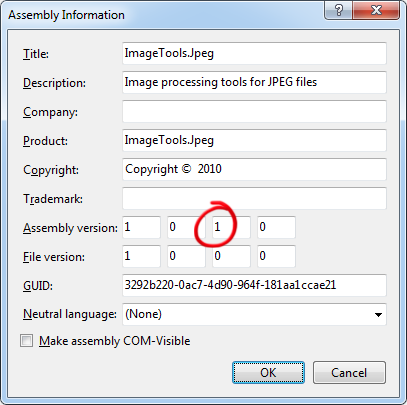
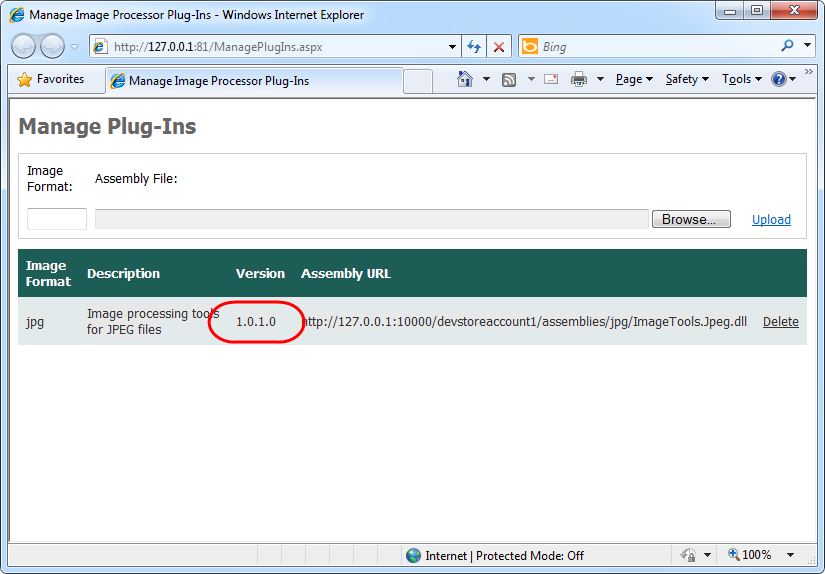
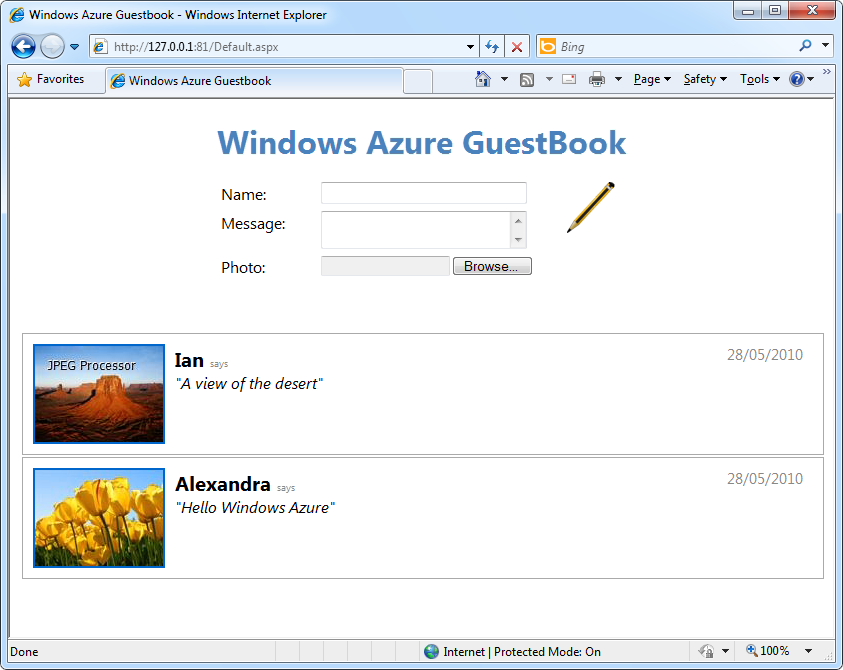
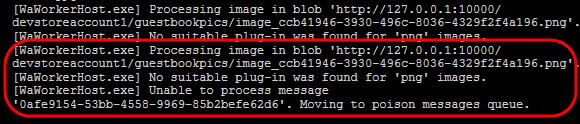
Task 4 – Updating the Worker Role to Use Plug-ins

* 1. In this task, you update the code in the worker role to allow it to determine the type of image that it needs to process from the queue and, based on this format, load a plug-in class that can handle it.
  2. Add a reference in the worker role project to the assembly with the contract that image processing plug-ins need to implement. Note that you are not referring to any concrete implementation of a plug-in here, only to its contract. To add the reference, right-click the **GuestBook\_WorkerRole** project in **Solution Explorer** and select **Add Reference**. In the **Add Reference** dialog, switch to the **Projects** tab, select the **ImageTools** project, and then click **OK**.
  3. Open **WorkerRole.cs** (for Visual C# projects) or **WorkerRole.vb** (for Visual Basic projects) in the **GuestBook\_WorkerRole** project.
  4. Append the following namespace directive to the existing list of namespaces at the top of the file.
     1. C#
     2. **using ImageTools;**
     3. Visual Basic
     4. **Imports ImageTools**
  5. Now, locate the **Run** method and insert the following (highlighted) block of code to determine the format of the input image from the “extension” of the blob URI. Place the code immediately following the line that generates the URI of the output thumbnail image, as shown (highlighted) below.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run image format-CS*)
     2. C#
     3. public override void Run()
     4. {
     5. ...
     6. // parse message retrieved from queue
     7. var messageParts = msg.AsString.Split(new char[] { ',' });
     8. var imageBlobUri = messageParts[0];
     9. var partitionKey = messageParts[1];
     10. var rowkey = messageParts[2];
     11. Trace.TraceInformation("Processing image in blob '{0}'.", imageBlobUri);
     12. string thumbnailBlobUri = System.Text.RegularExpressions.Regex.Replace(imageBlobUri, "([^\\.]+)(\\.[^\\.]+)?$", "$1-thumb$2");
     13. **// get image format from its extension**
     14. **string imageFormat = Path.GetExtension(imageBlobUri);**
     15. **if (imageFormat.StartsWith("."))**
     16. **{**
     17. **imageFormat = imageFormat.Substring(1, imageFormat.Length - 1).ToLower();**
     18. **}**
     19. CloudBlob inputBlob = container.GetBlobReference(imageBlobUri);
     20. CloudBlob outputBlob = container.GetBlobReference(thumbnailBlobUri);
     21. using (Stream input = inputBlob.OpenRead())
     22. using (Stream output = outputBlob.OpenWrite())
     23. {
     24. ...
     25. }
     26. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run image format-VB*)
     27. Visual Basic
     28. Public Overrides Sub Run()
     29. ...
     30. ' parse message retrieved from queue
     31. Dim messageParts = msg.AsString.Split(New Char() {","c})
     32. Dim imageBlobUri = messageParts(0)
     33. Dim partitionKey = messageParts(1)
     34. Dim rowKey = messageParts(2)
     35. Trace.TraceInformation("Processing image in blob '{0}'.", imageBlobUri)
     36. Dim thumbnailBlobUri As String = System.Text.RegularExpressions.Regex.Replace(imageBlobUri, "([^\\.]+)(\\.[^\\.]+)?$", "$1-thumb$2")
     37. **' get image format from its extension**
     38. **Dim imageFormat As String = Path.GetExtension(imageBlobUri)**
     39. **If imageFormat.StartsWith(".") Then**
     40. **imageFormat = imageFormat.Substring(1, imageFormat.Length - 1).ToLower()**
     41. **End If**
     42. ' download original image from blob storage
     43. Dim inputBlob As CloudBlockBlob = container.GetBlockBlobReference(imageBlobUri)
     44. Dim outputBlob As CloudBlockBlob = container.GetBlockBlobReference(thumbnailBlobUri)
     45. Using input As Stream = inputBlob.OpenRead()
     46. Using output As Stream = outputBlob.OpenWrite()
     47. ...
     48. End Sub
     49. **Note:** The inserted code extracts the extension from the name of the input blob and uses it to determine the format of the image. Note that an alternative approach for determining the type of image might involve examining the **ContentType** property of the blob. However, when uploading images, different browsers could interpret the image format differently and produce slightly different content types, for example, *image/jpeg* and *image/pjpeg*. To avoid any ambiguity and simplify the code in this hands-on lab, the code uses the file extension for this purpose.
  6. Next, add the following (highlighted) code to instantiate a new plug-in. Place the code immediately after the lines inserted in the previous step.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run GetPlugIn-CS*)
     2. C#
     3. public override void Run()
     4. {
     5. ...
     6. // get image format from its extension
     7. string imageFormat = Path.GetExtension(imageBlobUri);
     8. if (imageFormat.StartsWith("."))
     9. {
     10. imageFormat = imageFormat.Substring(1, imageFormat.Length - 1).ToLower();
     11. }
     12. **PlugIn<IImageProcessor> imageProcessorPlugIn = PlugIn<IImageProcessor>.GetPlugIn(imageFormat);**
     13. **if (imageProcessorPlugIn != null)**
     14. **{**
     15. CloudBlob inputBlob = container.GetBlobReference(imageBlobUri);
     16. CloudBlob outputBlob = container.GetBlobReference(thumbnailBlobUri);
     17. ...
     18. }
     19. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run GetPlugIn-VB*)
     20. Visual Basic
     21. Public Overrides Sub Run()
     22. ...
     23. ' get image format from its extension
     24. Dim imageFormat As String = Path.GetExtension(imageBlobUri)
     25. If imageFormat.StartsWith(".") Then
     26. imageFormat = imageFormat.Substring(1, imageFormat.Length - 1).ToLower()
     27. End If
     28. **Dim imageProcessorPlugIn As PlugIn(Of IImageProcessor) = PlugIn(Of IImageProcessor).GetPlugIn(imageFormat)**
     29. **If imageProcessorPlugIn IsNot Nothing Then**
     30. ' download original image from blob storage
     31. Dim inputBlob As CloudBlockBlob = container.GetBlockBlobReference(imageBlobUri)
     32. Dim outputBlob As CloudBlockBlob = container.GetBlockBlobReference(thumbnailBlobUri)
     33. ...
     34. End Sub
  7. Complete the change by closing the conditional block inserted in the previous step, as shown in the following (highlighted) code fragment.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run end conditional block-CS*)
     2. C#
     3. public override void Run()
     4. {
     5. ...
     6. PlugIn<IImageProcessor> imageProcessorPlugIn = PlugIn<IImageProcessor>.GetPlugIn(imageFormat);
     7. if (imageProcessorPlugIn != null)
     8. {
     9. CloudBlob inputBlob = container.GetBlobReference(imageBlobUri);
     10. CloudBlob outputBlob = container.GetBlobReference(thumbnailBlobUri);
     11. using (BlobStream input = inputBlob.OpenRead())
     12. using (BlobStream output = outputBlob.OpenWrite())
     13. {
     14. ...
     15. }
     16. **}**
     17. **else**
     18. **{**
     19. **Trace.TraceError(String.Format("No suitable plug-in was found for '{0}' images.", imageFormat));**
     20. **}**
     21. ...
     22. }
     23. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run end conditional block-VB*)
     24. Visual Basic
     25. Public Overrides Sub Run()
     26. ...
     27. Dim imageProcessorPlugIn As PlugIn(Of IImageProcessor) = PlugIn(Of IImageProcessor).GetPlugIn(imageFormat)
     28. If imageProcessorPlugIn IsNot Nothing Then
     29. ' download original image from blob storage
     30. Dim inputBlob As CloudBlockBlob = container.GetBlockBlobReference(imageBlobUri)
     31. Dim outputBlob As CloudBlockBlob = container.GetBlockBlobReference(thumbnailBlobUri)
     32. Using input As BlobStream = inputBlob.OpenRead()
     33. Using output As BlobStream = outputBlob.OpenWrite()
     34. ...
     35. End Using
     36. End Using
     37. **Else**
     38. **Trace.TraceError(String.Format("No suitable plug-in was found for '{0}' images.", imageFormat))**
     39. **End If**
     40. ...
     41. End Sub
  8. Now, replace the previous call to **ProcessImage** with a call to the **Execute** method of the plug-in instead, as shown in the following (highlighted) code snippet.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run Execute plug-in-CS*)
     2. C#
     3. public override void Run()
     4. {
     5. ...
     6. PlugIn<IImageProcessor> imageProcessorPlugIn = PlugIn<IImageProcessor>.GetPlugIn(imageFormat);
     7. if (imageProcessorPlugIn != null)
     8. {
     9. CloudBlob inputBlob = container.GetBlobReference(imageBlobUri);
     10. CloudBlob outputBlob = container.GetBlobReference(thumbnailBlobUri);
     11. using (BlobStream input = inputBlob.OpenRead())
     12. using (BlobStream output = outputBlob.OpenWrite())
     13. {
     14. ~~ProcessImage(input, output);~~
     15. **if (imageProcessorPlugIn.Execute(callback => callback.ProcessImage(input, output)))**
     16. **{**
     17. // commit the blob and set its properties
     18. output.Commit();
     19. outputBlob.Properties.ContentType = "image/jpeg";
     20. outputBlob.SetProperties();
     21. ...
     22. ...
     23. }
     24. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run Execute plug-in-VB*)
     25. Visual Basic
     26. Public Overrides Sub Run()
     27. ...
     28. Dim imageProcessorPlugIn As PlugIn(Of IImageProcessor) = PlugIn(Of IImageProcessor).GetPlugIn(imageFormat)
     29. If imageProcessorPlugIn IsNot Nothing Then
     30. ' download original image from blob storage
     31. Dim inputBlob As CloudBlockBlob = container.GetBlockBlobReference(imageBlobUri)
     32. Dim outputBlob As CloudBlockBlob = container.GetBlockBlobReference(thumbnailBlobUri)
     33. Using input As BlobStream = inputBlob.OpenRead()
     34. Using output As BlobStream = outputBlob.OpenWrite()
     35. ~~ProcessImage(input, output)~~
     36. **If imageProcessorPlugIn.Execute(Sub(callback) callback.ProcessImage(input, output)) Then**
     37. ' commit the blob and set its properties
     38. output.Commit()
     39. outputBlob.Properties.ContentType = "image/jpeg"
     40. outputBlob.SetProperties()
     41. ...

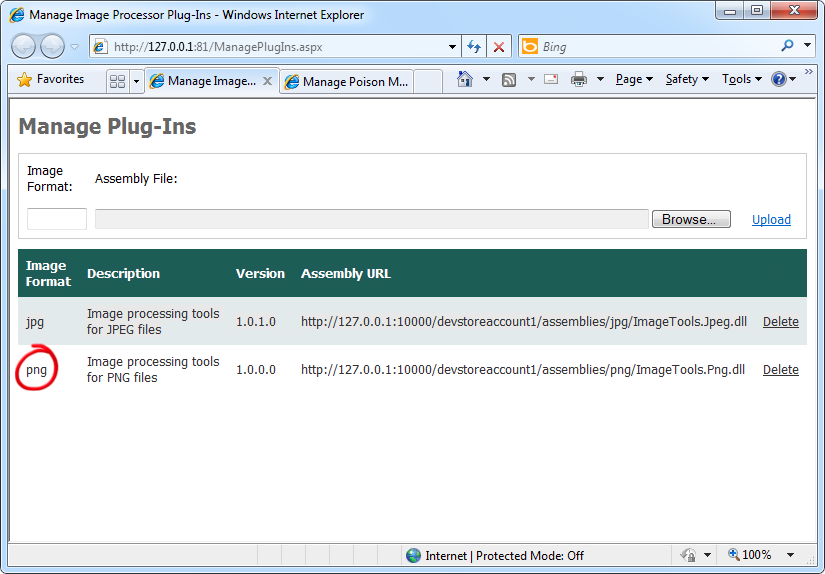
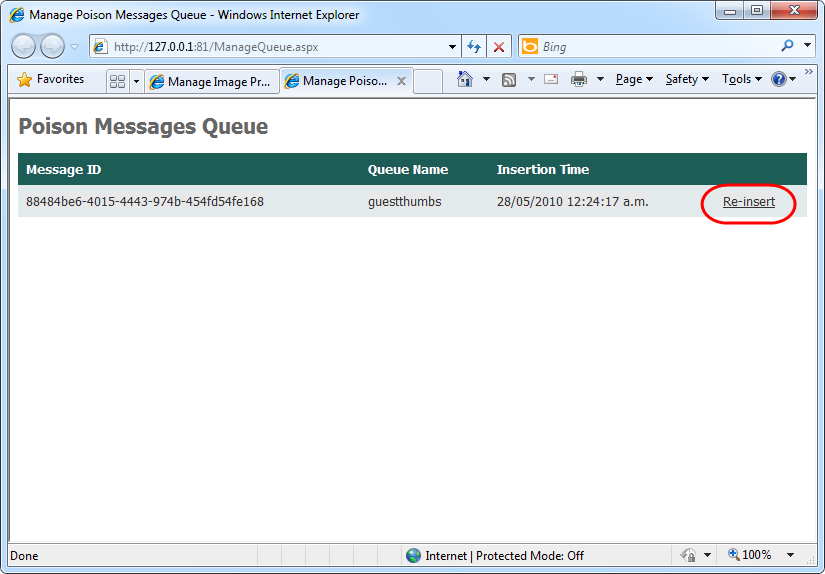
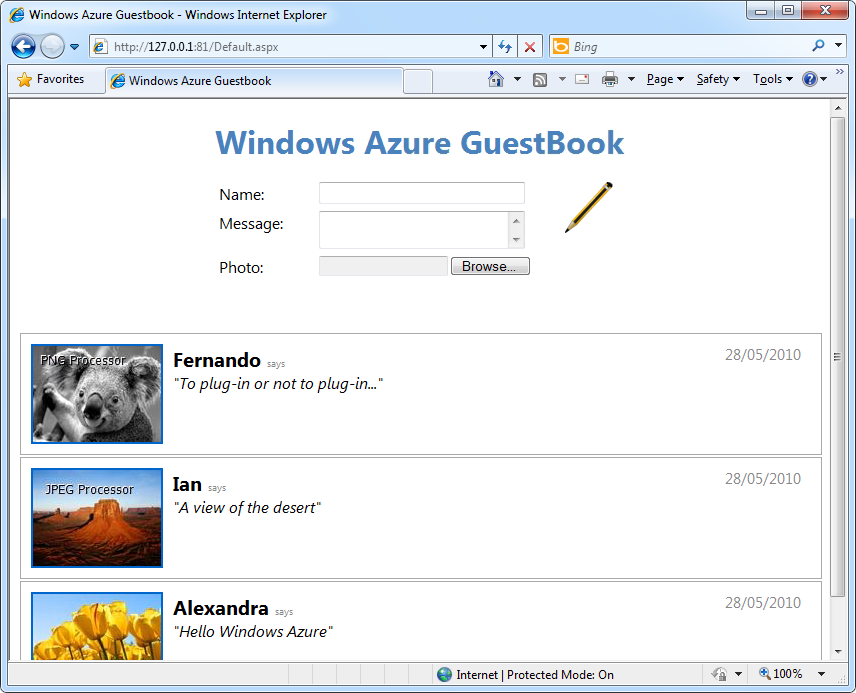
End Sub

* 1. Complete the conditional block added in the previous step by inserting the following (highlighted) code.
     1. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run Execute end conditional-CS*)
     2. C#
     3. public override void Run()
     4. {
     5. ...
     6. if (imageProcessorPlugIn.Execute(callback => callback.ProcessImage(input, output)))
     7. {
     8. // commit the blob and set its properties
     9. output.Commit();
     10. outputBlob.Properties.ContentType = "image/jpeg";
     11. outputBlob.SetProperties();
     12. // update the entry in table storage to point to the thumbnail
     13. GuestBookDataSource ds = new GuestBookDataSource();
     14. ds.UpdateImageThumbnail(partitionKey, rowkey, thumbnailBlobUri);
     15. // remove message from queue
     16. queue.DeleteMessage(msg);
     17. Trace.TraceInformation("Generated thumbnail in blob '{0}'.", thumbnailBlobUri);
     18. **}**
     19. **else**
     20. **{**
     21. **output.Abort();**
     22. **}**
     23. ...
     24. }
     25. (Code Snippet – *AsynchWorkloadHandling-Ex2-Run Execute end conditional-VB*)
     26. Visual Basic
     27. Public Overrides Sub Run()
     28. ...
     29. If imageProcessorPlugIn.Execute(Sub(callback) callback.ProcessImage(input, output)) Then
     30. ' commit the blob and set its properties
     31. output.Commit()
     32. outputBlob.Properties.ContentType = "image/jpeg"
     33. outputBlob.SetProperties()
     34. ' update the entry in table storage to point to the thumbnail
     35. Dim ds = New GuestBookDataSource()
     36. ds.UpdateImageThumbnail(partitionKey, rowKey, thumbnailBlobUri)
     37. ' remove message from queue
     38. queue.DeleteMessage(msg)
     39. Trace.TraceInformation("Generated thumbnail in blob '{0}'.", thumbnailBlobUri)
     40. **Else**
     41. **output.Abort()**
     42. **End If**
     43. ...
     44. End Sub
     45. **Note:** During this task, you inserted several blocks of code but you will see that the changes only affected a minor portion of the code. The bulk of the message processing code remains the same.
     46. The inserted code extracts the extension from the name of the input blob, uses it to determine the format of the image, and then retrieves a plug-in instance that can process the required format. The code that creates the image streams from blob storage remains the same. The only other significant change is that instead of calling **ProcessImage** directly, the code now uses the **Execute** method of the plug-in class passing in a callback delegate, expressed as a lambda expression, to invoke the corresponding method of the plug-in in the remote domain. If **Execute** returns *true*, it means that the call succeeded and the worker role can update the guest book entry and delete the corresponding message from the queue.

Task 5 – Resolving Bugs with Zero Downtime

* 1. In this task, you test the dynamic worker role. You will then resolve the ‘bug’ that you introduced in the previous exercise and upload an updated plug-in assembly without restarting the worker role. After that, you will re-inject the poisonous messages back into the original queue and watch as they are successfully processed.
  2. Press **CTRL+F5** to build and run the application.
  3. Navigate to the <http://127.0.0.1:81/ManagePlugins.aspx> page in the application.
     1. **Note:** You may need to adjust the port number in the address of the ManagePlugins page if the compute emulator launches the Web role using a different port value.
  4. In the plug-in management page, type *jpg* in the **Image Format** field. Next, press **Browse** and navigate to the folder where the **ImageTools.Jpeg** project built its output—it should be **bin/debug** inside the plug-in project’s folder provided you have not changed the default settings. Select the **ImageTools.Jpeg.dll** file and then click **Upload** to register the plug-in.
     1. 
     2. Figure 29
     3. Uploading a new plug-in assembly
  5. Once the plug-in uploads and the page refreshes, notice that the UI shows the metadata retrieved from the assembly, including its description and version.
     1. 
     2. Figure 30
     3. Plug-in management UI showing registered plug-ins
  6. Next, navigate to the main page of the application (<http://127.0.0.1:81/Default.aspx>) and create a new guest book entry. For the first entry, pick any suitable JPEG image file whose size is smaller than the limit chosen to trigger the unhandled exception (750KB). This allows you to verify that the plug-in framework works as expected. If you experience any problems, you may need to review the previous steps in the exercise to verify that you have followed them correctly.
  7. Now, create a new entry using an image large enough to provoke the exception in the image processor plug-in. Instead, you may also re-inject one of the messages that the worker role could not process when you tested the application earlier and are currently in the poison messages queue.
  8. Switch to the compute emulator UI and examine the diagnostics log. Notice that even though the worker role is still unable to process the message, it does not crash. Instead, the log shows the failed attempts to process the image. Note that each failure results in the unloading of the application domain that hosts the plug-in. As before, when the dequeue count limit is exceeded, the message is re-routed to the poison messages queue.
     1. 
     2. Figure 31
     3. Diagnostics log showing failed attempts to process the image
  9. Return to Visual Studio and open **ImageProcessor.cs** (for Visual C# projects) or **ImageProcessor.vb** (for Visual Basic projects) in the **ImageTools.Jpeg** project. Locate the **ProcessImage** method and identify the lines of code that you inserted in the previous exercise to force the exception when processing large image files. Comment these lines out or remove them.
     1. C#
     2. public void ProcessImage(Stream input, Stream output)
     3. {
     4. **~~//// simulate an unexpected exception caused by unanticipated conditions~~**
     5. **~~//if (input.Length > 768000)~~**
     6. **~~//{~~**
     7. **~~// throw new InvalidOperationException("Cannot handle image files larger than 750KB.");~~**
     8. **~~//}~~**
     9. int width;
     10. int height;
     11. var originalImage = new Bitmap(input);
     12. if (originalImage.Width > originalImage.Height)
     13. {
     14. ...
     15. }
     16. Visual Basic
     17. Private Sub ProcessImage(ByVal input As Stream, ByVal output As Stream)
     18. **~~'' simulate an unexpected exception caused by unanticipated conditions~~**
     19. **~~'If input.Length > 768000 Then~~**
     20. **~~' Throw New InvalidOperationException("Cannot handle image files larger than 750KB.")~~**
     21. **~~'End If~~**
     22. Dim width As Integer
     23. Dim height As Integer
     24. Dim originalImage As New Bitmap(input)
     25. If originalImage.Width > originalImage.Height Then
     26. ...
     27. End Sub
  10. Now, insert the following (highlighted) code to overlay a caption on the thumbnail image. This step is optional but allows you to identify which plug-in processed any given image.
      1. (Code Snippet – *AsynchWorkloadHandling-Ex2-ProcessImage caption-CS*)
      2. C#
      3. public void ProcessImage(Stream input, Stream output)
      4. {
      5. ...
      6. using (Graphics graphics = Graphics.FromImage(thumbnailImage))
      7. {
      8. graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
      9. graphics.SmoothingMode = SmoothingMode.AntiAlias;
      10. graphics.PixelOffsetMode = PixelOffsetMode.HighQuality;
      11. graphics.DrawImage(originalImage, 0, 0, width, height);
      12. **string caption = "JPEG Processor";**
      13. **using (Font font = new Font("Tahoma", 10F, FontStyle.Regular))**
      14. **{**
      15. **graphics.SmoothingMode = SmoothingMode.AntiAlias;**
      16. **SizeF size = graphics.MeasureString(caption, font, thumbnailImage.Width);**
      17. **graphics.DrawString(caption, font, Brushes.White, 5, 5);**
      18. **graphics.DrawString(caption, font, Brushes.Black, 6, 6);**
      19. **}**
      20. }
      21. thumbnailImage.Save(output, ImageFormat.Jpeg);
      22. }
      23. (Code Snippet – *AsynchWorkloadHandling-Ex2-ProcessImage caption-VB*)
      24. Visual Basic
      25. Private Sub ProcessImage(ByVal input As Stream, ByVal output As Stream)
      26. ...
      27. Using graphic = Graphics.FromImage(thumbnailImage)
      28. graphic.InterpolationMode = InterpolationMode.HighQualityBicubic
      29. graphic.SmoothingMode = SmoothingMode.AntiAlias
      30. graphic.PixelOffsetMode = PixelOffsetMode.HighQuality
      31. graphic.DrawImage(originalImage, 0, 0, width, height)
      32. **Dim caption = "JPEG Processor"**
      33. **Using font As New Font("Tahoma", 10.0F, FontStyle.Regular)**
      34. **graphic.SmoothingMode = SmoothingMode.AntiAlias**
      35. **Dim size As SizeF = graphic.MeasureString(caption, font, thumbnailImage.Width)**
      36. **graphic.DrawString(caption, font, Brushes.White, 5, 5)**
      37. **graphic.DrawString(caption, font, Brushes.Black, 6, 6)**
      38. **End Using**
      39. End Using
      40. thumbnailImage.Save(output, ImageFormat.Jpeg)
      41. End Sub
  11. Next, update the version number of the plug-in assembly. To do this, right-click the **ImageTools.Jpeg** project in **Solution Explorer** and select **Properties**. In the properties window, select the **Application** tab, and then click **Assembly Information**. Increment the minor version number, for example, change it to 1.0.1.0, and click **OK**.
      1. 
      2. Figure 32
      3. Changing the version of the plug-in assembly
  12. Finally, right-click the **ImageTools.Jpeg** project in **Solution Explorer** and select **Build** to generate an updated plug-in assembly.
      1. **Important:** Be careful not to build the entire solution, otherwise, Visual Studio forces the running deployment in the compute emulator to unload.
  13. In your browser window, navigate once again to the plug-in management page in the application (<http://127.0.0.1:81/ManagePlugins.aspx>) and upload the updated plug-in assembly following the same procedure that you used when you registered it the first time. Provided you use the same image format, the uploaded assembly replaces the one that you previously registered. Verify that the assembly version displayed matches the one that you configured earlier, when you updated the plug-in project.
      1. 
      2. Figure 33
      3. Plug-in management UI showing the updated assembly version
  14. Now, browse to the poison messages queue at <http://127.0.0.1:81/ManageQueue.aspx> and re-insert one of the messages that failed earlier.
  15. This time, with the updated plug-in, the worker role should process the image successfully. If you return to the main page of the application, you should observe the thumbnail generated by the new image processor and the overlay caption that identifies it.
      1. 
      2. Figure 34
      3. Application showing the image processed by the updated plug-in
  16. Finally, create a new guest book entry specifying a PNG image file instead. Switch back to the compute emulator UI and examine the diagnostics log for the worker role. Notice that the worker role is unable to find a suitable plug-in to process this type of image file and eventually, moves the message to the poison messages queue.
      1. **Note:** You may use a JPEG image file for this test, as long as you change its extension to PNG.
      2. 
      3. Figure 35
      4. Diagnostics log showing that a required plug-in is missing
      5. In the next task, you will see how to use the plug-in framework to dynamically update the running worker role and allow it to process new image types.
      6. **Note:** Do not stop the deployment currently executing in the compute emulator. You will require it for the next task.

Task 6 – Extending the Worker Role with Plug-Ins for Additional Image Types

* 1. In this task, you create a plug-in to process PNG files and update the running worker role to load it. To test the new plug-in, you re-insert the message that failed to be processed during the previous task from the poison messages queue back into its source queue and verify that the message can now be processed successfully.
  2. To create a plug-in project to process PNG image files, in the **File** menu, point to **Add** and then select **New Project**.
  3. In the **Add New Project** dialog, expand the language of your choice under the **Installed Templates** tree view, select the **Windows** category, and then choose the **Class Library** project template. Set the name to **ImageTools.Png**, leave the proposed location inside the solution folder unchanged, and then click **OK**.
  4. In the new class library project, add a reference to the plug-in contract assembly. To add the reference, right-click the **ImageTools.Png** project in **Solution Explorer**, and then select **Add Reference**. In the **Add Reference** dialog, switch to the **Projects** tab, select the **ImageTools** project, and then click **OK**.
  5. Repeat the previous step to add a reference to the **System.Drawing** assembly, only this time select the assembly in the **.NET** tab of the **Add Reference** dialog.
  6. Now, right-click the **Class1.cs** file in **Solution Explorer** and select **Delete**. You will not require it.
  7. Since the purpose of the exercise is not learning image processing techniques, for simplicity, you will use an existing implementation for the plug-in provided with the lab materials. To add the file, right-click the **ImageTools.Png** project in **Solution Explorer**, point to **Add**, and then select **Existing Item**. In the **Add Existing Item** dialog, browse to **Assets** in the **Source** folder of the lab, choose **ImageProcessor.cs** or **ImageProcessor.vb**, depending on the language of your project, and then click **Add**.
  8. If you open the file and examine the implementation of the **ImageProcessor** class that it contains, you will see that the class is almost identical to the JPEG processor that you built previously. To differentiate both processors and create a visible difference in the processed output, the PNG processor converts its output to produce gray-scale images. This difference is entirely arbitrary.
  9. Next, right-click the **ImageTools.Png** project in **Solution Explorer** and select **Properties**. Select the **Application** page, and then click **Assembly Information**. In the **Assembly Information** dialog, enter a suitable description for the plug-in, and then click **OK**.
  10. Build the project for the PNG image files plug-in. To do this, right-click the **ImageTools.Png** project in **Solution Explorer** and select **Build** to generate an assembly.
      1. **Important:** Be careful not to build the entire solution, otherwise, Visual Studio forces the running deployment in the compute emulator to unload.
  11. Now, deploy the new plug-in assembly. If you followed the instructions, the deployment created in the previous task should still be running in the compute emulator. In your browser, open the plug-in management page (<http://127.0.0.1:81/ManagePlugIns.aspx>), and then register the new plug-in using PNG as the image format.
      1. 
      2. Figure 36
      3. Plug-in management page showing the new plug-in for PNG image files
  12. To re-insert the message that the worker role was unable to process during the previous task, open the poison messages queue management page in your browser (<http://127.0.0.1:81/ManageQueue.aspx>), locate the message, and then click **Re-insert**.
      1. 
      2. Figure 37
      3. Re-processing a failed message after registering the new plug-in
  13. Go back to the main page of the GuestBook application and wait a few seconds for the page to refresh and show the grays-scale thumbnail for the PNG image file.
      1. 
      2. Figure 38
      3. GuestBook page showing a gray-scale image produced by the PNG processor

Summary

* 1. By completing this hands-on lab, you learnt techniques that allow you to build resilient Windows Azure services that continue to operate under unexpected conditions. You explored an asynchronous processing model that uses queues and saw how to take advantage of the dequeue count to temporarily discard messages that cannot be processed successfully by diverting them into a poison messages queue.
  2. The lab showed you how to design an Azure role to make use of plug-in components hosted in separate application domains, allowing you to service a faulty component with no downtime. You saw how this enables you fix a defect and then process a failed job again by re-inserting messages in the poison messages queue.
  3. Finally, you saw how a plug-in model enables you to incorporate new functionality in a worker role dynamically without the need to re-deploy the service.