

Hands-On Lab

Debugging Applications in Windows Azure

Lab version: 1.0.0

Last updated: 4/25/2012



Contents

[Overview 3](#_Toc323113657)

[Exercise 1: Learn What Features and Techniques are Available in Visual Studio and Windows Azure 5](#_Toc323113658)

[Task 1 – Exploring the Fabrikam Insurance Application 5](#_Toc323113659)

[Task 2 – Running the Application as a Windows Azure Project 7](#_Toc323113660)

[Exercise 2: Adding IntelliTrace to the Application 14](#_Toc323113661)

[Task 1 – Adding Tracing Support to the Application 14](#_Toc323113662)

[Task 2 – Creating a Log Viewer Tool 26](#_Toc323113663)

[Verification 34](#_Toc323113664)

[Summary 39](#_Toc323113665)

Overview

* 1. Using Visual Studio, you can debug applications in your local machine by stepping through code, setting breakpoints, and examining the value of program variables. For Windows Azure applications, the compute emulator allows you to run the code locally and debug it using these same features and techniques, making this process relatively straightforward.
  2. Ideally, you should take advantage of the compute emulator and use Visual Studio to identify and fix most bugs in your code, as this provides the most productive environment for debugging. Nevertheless, some bugs might remain undetected and will only manifest themselves once you deploy the application to the cloud. These are often the result of missing dependencies or caused by differences in the execution environment. For addition information on environment issues, see [Differences Between the Compute Emulator and Windows Azure](http://msdn.microsoft.com/en-us/library/ee923628.aspx).
  3. Once you deploy an application to the cloud, you are no longer able to attach a debugger and instead, need to rely on debugging information written to logs in order to diagnose and troubleshoot application failures. Windows Azure provides comprehensive diagnostic facilities that allow capturing information from different sources, including Windows Azure application logs, IIS logs, failed request traces, Windows event logs, custom error logs, and crash dumps. The availability of this diagnostic information relies on the Windows Azure Diagnostics Monitor to collect data from individual role instances and transfer this information to Windows Azure storage for aggregation. Once the information is in storage, you can retrieve it and analyze it.

# Objectives

* 1. In this hands-on lab, you will:
  + Learn what features and techniques are available in Visual Studio and Windows Azure to debug applications once deployed to Windows Azure.
  + Use a simple TraceListener to log directly to table storage and a viewer to retrieve these logs.

# 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 1.6](http://www.microsoft.com/windowsazure/sdk/)

# Setup

* 1. In order to execute the exercises in this hands-on lab you need to set up your environment.
  2. Open a Windows Explorer window and browse to the lab’s **Source** folder.
  3. Double-click the **Setup.cmd** file in this folder to launch the setup process that will configure your environment and install the Visual Studio code snippets for this lab.
  4. If the User Account Control dialog is shown, confirm the action to proceed.
  5. **Note:** Make sure you have checked all the dependencies for this lab before running the setup.

# 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.

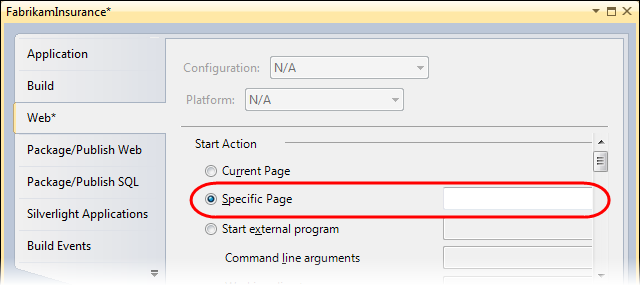
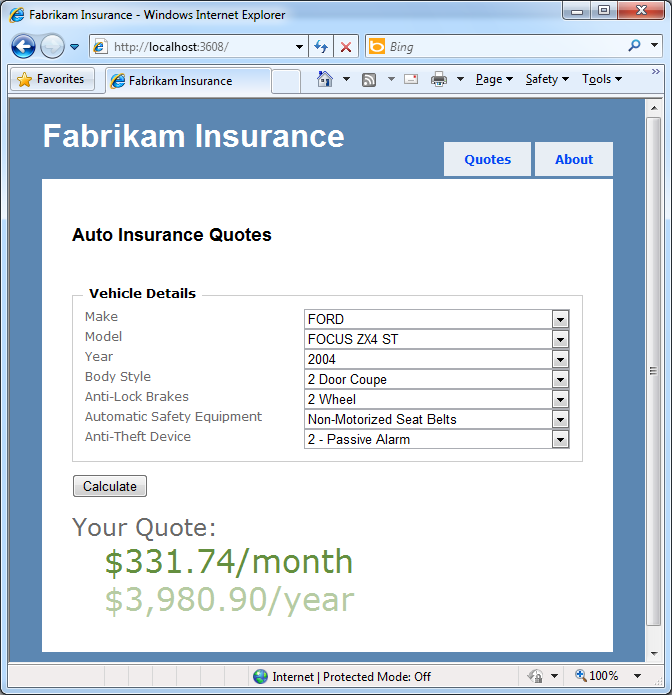
# Exercises

* 1. This hands-on lab includes the following exercise:
  2. Learn What Features and Techniques are Available in Visual Studio and Windows Azure
  3. Adding IntelliTrace to Debug the Application
  4. Estimated time to complete this lab: **40 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.

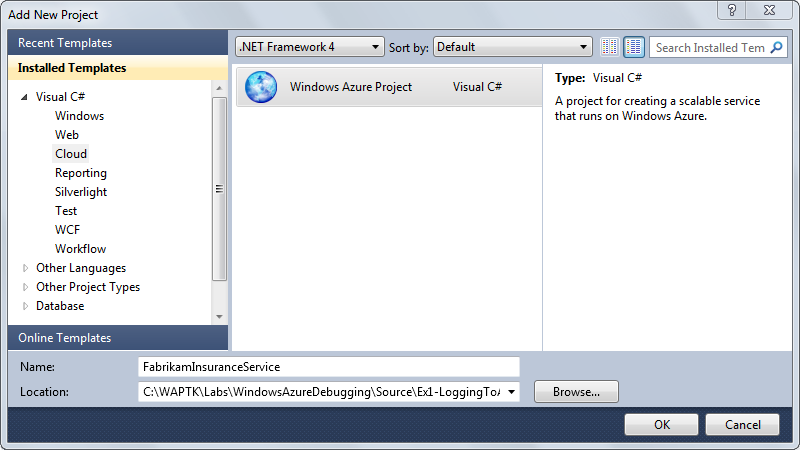
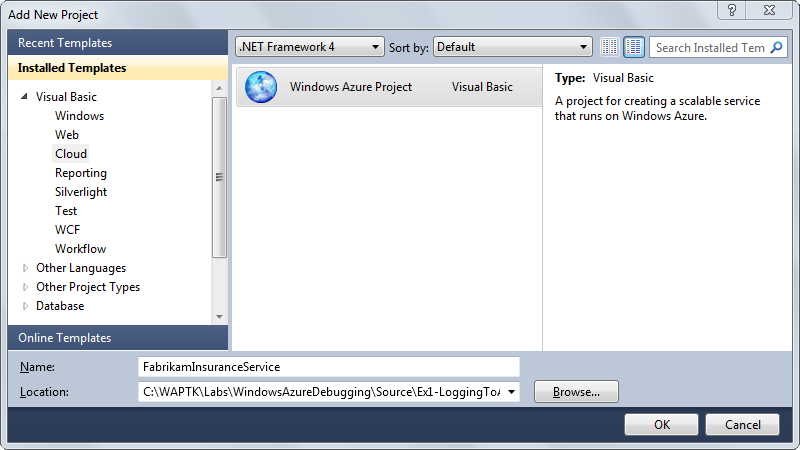
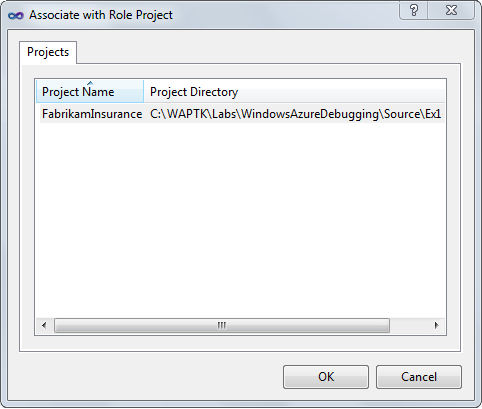
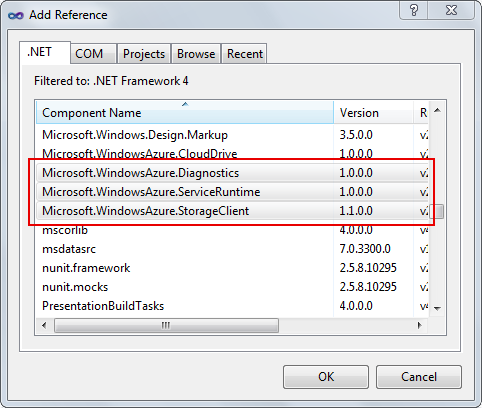
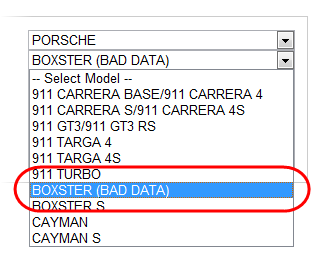
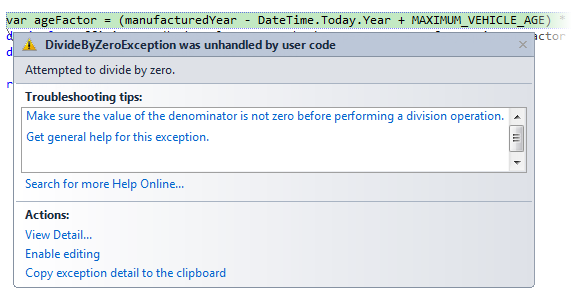
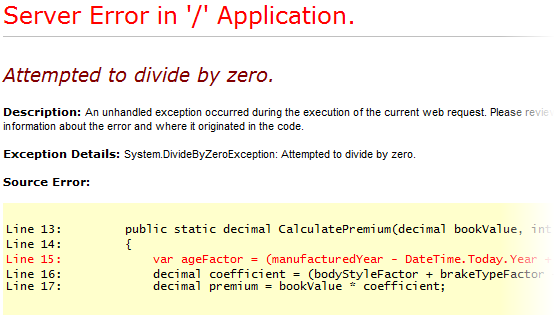
Exercise 1: Learn What Features and Techniques are Available in Visual Studio and Windows Azure

* 1. Because Windows Azure Diagnostics is oriented towards operational monitoring and has to cater for gathering information from multiple role instances, it requires that diagnostic data first be transferred from local storage in each role to Windows Azure storage, where it is aggregated. This requires programming scheduled transfers with the diagnostic monitor to copy logging data to Windows Azure storage at regular intervals, or else requesting a transfer of the logs on-demand. Moreover, information obtained in this manner provides a snapshot of the diagnostics data available at the time of the transfer. To retrieve updated data, a new transfer is necessary. When debugging a single role, and especially during the development phase, these actions add unnecessary friction to the process. To simplify the retrieval of diagnostics data from a deployed role, it is simpler to read information directly from Windows Azure storage, without requiring additional steps.

Task 1 – Exploring the Fabrikam Insurance Application

* 1. In this task, you build and run the Fabrikam Insurance application in the Web Development Server to become familiar with its operation.
  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 **LoggingToAzureStorage** 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. Set the start action of the project. To do this, in **Solution Explorer**, right-click the **FabrikamInsurance** project and then select **Properties**. In the properties window, switch to the **Web** tab and then, under **Start Action**, select the **Specific Page** option. Leave the page value blank.
     1. 
     2. Figure
     3. Configuring the start action of the project
  6. Press **F5** to build and run the solution. The application should launch in the Web Development Server and open its **Auto Insurance Quotes** page in your browser.
  7. To explore its operation, complete the form by choosing any combination of values from the **Vehicle Details** drop down lists and then click **Calculate** to obtain a quote for the insurance premium. Notice that after you submit the form, the page refreshes and shows the calculated amount.
     1. 
     2. Figure
     3. Exploring the Fabrikam Insurance application
  8. Press **SHIFT + F5** to stop debugging and shut down the application.

Task 2 – Running the Application as a Windows Azure Project

* 1. In this task, you create a new Windows Azure Project to prepare the application for deployment to Windows Azure.
  2. Add a new Windows Azure Project to the solution. To do this, in the **File** menu, point to **Add** and then select **New Project**. In the **Add** **New Project** dialog, expand the language of your preference (Visual C# or Visual Basic) in the **Installed Templates** list and then select **Cloud**. Choose the **Windows Azure Project** template, set the **Name** of the project to **FabrikamInsuranceService** and accept the proposed location in the folder of the solution. Click **OK** to create the project.
     1. ****
     2. Figure
     3. Creating a new Windows Azure Project (C#)
     4. 
     5. Figure
     6. Creating a new Windows Azure Project (Visual Basic)
  3. In the **New Windows Azure Project** dialog, click **OK** without adding any new roles to the solution.
  4. Now, in **Solution Explorer**, right-click the **Roles** node in the new **FabrikamInsuranceService** project, point to **Add**, and then select **Web Role Project in solution**. Then, in the **Associate with Role Project** dialog, select the **FabrikamInsurance** project, and click **OK**.
     1. 
     2. Figure
     3. Associating the MVC application with the Windows Azure Project
  5. Add references to the Windows Azure support assemblies. To do this, in **Solution Explorer**, right-click the **FabrikamInsurance** project, and then select **Add Reference**. In the **Add Reference** dialog, switch to the **.NET** tab, select the **Microsoft.WindowsAzure.Diagnostics**, **Microsoft.WindowsAzure.ServiceRuntime**, and **Microsoft.WindowsAzure.StorageClient** components, and then click **OK**.
     1. 
     2. Figure
     3. Adding references to the Windows Azure support assemblies to the project
  6. Now, add a role entry point to the MVC application. To do this, in **Solution Explorer**, right-click the **FabrikamInsurance** project, point to **Add**, and then select **Existing Item**. In the **Add Existing Item** dialog, browse to **Assets** in the **Source** folder of the lab. Inside this folder, choose the folder for the language of your project (Visual C# or Visual Basic), select **WebRole.cs** or **WebRole.vb**, and then click **Add**.
     1. **Note:** The **WebRole** class is a **RoleEntryPoint** derived class that contains methods that Windows Azure calls when it starts, runs, or stops the role. The provided code is the same that Visual Studio generates when you create a new Windows Azure Project.
  7. You are now ready to test the Windows Azure Project application. To launch the application in the compute emulator, press **F5**. Wait until the deployment completes and the browser opens to show its main page.
  8. Again, complete the entry form by choosing a combination of values from the drop down lists and then click **Calculate**. Ensure that you receive a valid response with the calculated premium as a result.
  9. Once you have verified that everything works in the compute emulator just as it did when hosted by the Web Development Server, you will now cause an exception by making the application process bad data that it does not handle correctly. To do this, change the values used for the calculation by setting the **Make** to “*PORSCHE”* and the **Model** to “*BOXSTER (BAD DATA)”*.
     1. 
     2. Figure
     3. Choosing make and model for the insurance premium calculation
  10. Click **Calculate** to re-submit the form with new values. Notice that an unhandled exception occurs and execution halts in the Visual Studio debugger at the line that caused the error.
      1. 
      2. Figure
      3. Unhandled exception in the application caused by bad data
      4. **Note:** Within the Visual Studio debugger, you are able to step through code, set breakpoints, and examine the value of program variables. Debugging applications hosted in the compute emulator provides the same experience that you typically have when debugging other programs to which you can attach the Visual Studio debugger. Using the debugger under these conditions is covered extensively and will not be explored here. For more information, see [Debugging in Visual Studio](http://msdn.microsoft.com/en-us/library/sc65sadd.aspx).
  11. Press **F5** to continue execution and let ASP.NET handle the exception. Notice that the unhandled exception handler provides details about the exception, including the line in the source code that raised the exception.
      1. 
      2. Figure
      3. ASP.NET default unhandled exception handler
      4. **Note:** Unhandled exceptions are typically handled by ASP.NET, which can report the error in its response including details about an error and the location in the source code where the exception was raised. However, for applications that are available publicly, exposing such information is not recommended to prevent unnecessary disclosure of internal details about the application that may compromise its security. Instead, errors and other diagnostics output should be written to a log that can only be retrieved after proper authorization.
      5. You can configure how information is displayed by ASP.NET when an unhandled error occurs during the execution of a Web request.For more information, see [customErrors Element (ASP.NET Settings Schema)](http://msdn.microsoft.com/en-us/library/h0hfz6fc.aspx).
      6. In this case, the unhandled exception error page includes full details for the error because the default mode for the **customErrors** element is *remoteOnly* and you are accessing the page locally. When you deploy the application to the cloud and access it remotely, the page shows a generic error message instead.
  12. Press **SHIFT + F5** to stop debugging and shut down the application.

1. Exercise 2: Adding IntelliTrace to the Application
   1. In this exercise, you debug a simple application by configuring a special trace listener that can write its output directly into a table in Windows Azure storage emulator. To produce diagnostic data, you instrument the application to write its trace information using standard methods in the System.Diagnostics namespace. Finally, you create a simple log viewer application that can retrieve and display the contents of the diagnostics table.
   2. The application that you will use for this exercise simulates an online auto insurance policy calculator. It has a single form where users can enter details about their vehicle and then submit the form to obtain an estimate on their insurance premium. Behind the scenes, the controller action that processes the form uses a separate assembly to calculate premiums based on the input from the user. The assembly contains a bug that causes it to raise an exception for input values that fall outside the expected range.

Task 1 – Adding Tracing Support to the Application

* 1. In the previous exercise, you briefly saw how to debug your application with Visual Studio when it executes locally in the compute emulator. To debug the application once you deploy it to the cloud, you need to write debugging information to the logs in order to diagnose an application failure.
  2. In this task, you add a TraceListener to the project capable of logging diagnostics data directly into table storage, where you can easily retrieve it with a simple query. The source code for this project is already provided for you in the **Assets** folder of the lab. More information on the Trace Listener can be found here: <http://msdn.microsoft.com/en-us/library/system.diagnostics.tracelistener.aspx>
  3. In **Solution Explorer**, right-click the **Begin** solution, point to **Add** and then select **Existing Project**. In the **Add Existing Project** dialog, browse to **Assets** in the **Source** folder of the lab, select the folder for the language of your choice (Visual C# or Visual Basic), then navigate to **AzureDiagnostics** inside this folder, select the **AzureDiagnostics** project file and click **Open**.
  4. Add a reference to the **AzureDiagnostics** library in the web role project. To do this, in **Solution Explorer**, right-click the **FabrikamInsurance** project, and select **Add Reference**. In the **Add Reference** dialog, switch to the **Projects** tab, select **AzureDiagnostics** in the list of projects, and then click **OK**.
  5. Open **Global.asax.cs** (for Visual C# projects) or **Global.asax.vb** (for Visual Basic projects) in the **FabrikamInsurance** project and insert the following namespace directives.
     1. C#
     2. **using Microsoft.WindowsAzure;**
     3. **using Microsoft.WindowsAzure.ServiceRuntime;**
     4. VB
     5. **Imports Microsoft.WindowsAzure**
     6. **Imports Microsoft.WindowsAzure.ServiceRuntime**
  6. Add the following (highlighted) method inside the **MvcApplication** class.

(Code Snippet – WindowsAzureDebugging-Ex1-ConfigureTraceListener-CS)

* + 1. C#
    2. public class MvcApplication : System.Web.HttpApplication

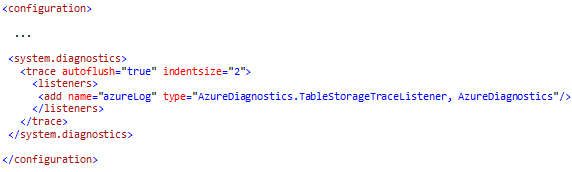
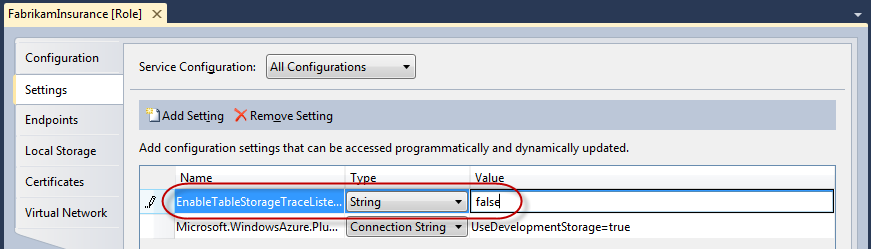
{

* + 1. ...
    2. **private static void ConfigureTraceListener()**
    3. **{**
    4. **bool enableTraceListener = false;**
    5. **string enableTraceListenerSetting = RoleEnvironment.GetConfigurationSettingValue("EnableTableStorageTraceListener");**
    6. **if (bool.TryParse(enableTraceListenerSetting, out enableTraceListener))**
    7. **{**
    8. **if (enableTraceListener)**
    9. **{**
    10. **AzureDiagnostics.TableStorageTraceListener listener =**
    11. **new AzureDiagnostics.TableStorageTraceListener("Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString")**
    12. **{**
    13. **Name = "TableStorageTraceListener"**
    14. **};**
    15. **System.Diagnostics.Trace.Listeners.Add(listener);**
    16. **System.Diagnostics.Trace.AutoFlush = true;**
    17. **}**
    18. **else**
    19. **{**
    20. **System.Diagnostics.Trace.Listeners.Remove("TableStorageTraceListener");**
    21. **}**
    22. **}**
    23. **}**
    24. }
    25. (Code Snippet – *WindowsAzureDebugging-Ex1-ConfigureTraceListener-VB*)
    26. Visual Basic
    27. Public Class MvcApplication
    28. Inherits System.Web.HttpApplication
    29. ...
    30. **Private Shared Sub ConfigureTraceListener()**
    31. **Dim enableTraceListener As Boolean = False**
    32. **Dim enableTraceListenerSetting As String = RoleEnvironment.GetConfigurationSettingValue("EnableTableStorageTraceListener")**
    33. **If Boolean.TryParse(enableTraceListenerSetting, enableTraceListener) Then**
    34. **If enableTraceListener Then**
    35. **Dim listener As New AzureDiagnostics.TableStorageTraceListener("Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString") With {.Name = "TableStorageTraceListener"}**
    36. **System.Diagnostics.Trace.Listeners.Add(listener)**
    37. **System.Diagnostics.Trace.AutoFlush = True**
    38. **Else**
    39. **System.Diagnostics.Trace.Listeners.Remove("TableStorageTraceListener")**
    40. **End If**
    41. **End If**
    42. **End Sub**
    43. End Class
    44. **Note:** The **ConfigureTraceListener** method retrieves the *EnableTableStorageTraceListener* configuration setting and, if its value is *true*, it creates a new instance of the **TableStorageTraceListener** class, defined in the project that you added to the solution earlier, and then adds it to the collection of available trace listeners. Note that the method also enables the **AutoFlush** property of the **Trace** object to ensure that trace messages are written immediately to table storage, allowing you to retrieve them as they occur.
  1. Now, insert the following (highlighted) code in the **Application\_Start** method to set up the Windows Azure storage configuration settings publisher and to enable the **TableStorageTraceListener**.

(Code Snippet – WindowsAzureDebugging-Ex1- Application\_Start-CS)

* + 1. C#
    2. public class MvcApplication : System.Web.HttpApplication
    3. {
    4. ...
    5. protected void Application\_Start()
    6. {
    7. **CloudStorageAccount.SetConfigurationSettingPublisher((configName, configSetter) =>**
    8. **{**
    9. **configSetter(RoleEnvironment.GetConfigurationSettingValue(configName));**
    10. **});**
    11. **ConfigureTraceListener();**
    12. AreaRegistration.RegisterAllAreas();
    14. RegisterRoutes(RouteTable.Routes);
    15. }
    16. ...
    17. }
    18. (Code Snippet – *WindowsAzureDebugging-Ex1-Application\_Start-VB*)
    19. Visual Basic
    20. Public Class MvcApplication
    21. Inherits System.Web.HttpApplication
    22. ...
    23. Sub Application\_Start()
    25. **CloudStorageAccount.SetConfigurationSettingPublisher(Sub(configName, configSetter) configSetter(RoleEnvironment.GetConfigurationSettingValue(configName)))**
    26. **ConfigureTraceListener()**

AreaRegistration.RegisterAllAreas()

* + 1. RegisterRoutes(RouteTable.Routes)
    2. End Sub
    3. ...
    4. End Class
    5. **Note:** TraceListeners can be added by configuring them in the **system.diagnostics** section of the configuration file. However, in this case, the role creates the listener programmatically allowing you to enable the listener only when you need it and while the service is running.
    6. 
    7. Figure
    8. Enabling the TableStorageTraceListener in the configuration file
  1. Next, define a configuration setting to control the diagnostics logging with the **TableStorageTraceListener**. To create the setting, expand the Roles node in the **FabrikamInsuranceService** project and then double-click the **FabrikamInsurance** role. In the role properties window, switch to the **Settings** page, click **Add Setting**, and then set the name of the new setting to *EnableTableStorageTraceListener*, the type as *String*, and the value as *false*.
     1. 
     2. Figure
     3. Creating a configuration setting to enable the trace listener
  2. Locate the **RoleEnvironmentChanging** event handler inside the **WebRole** class and replace its body with the following (highlighted) code.

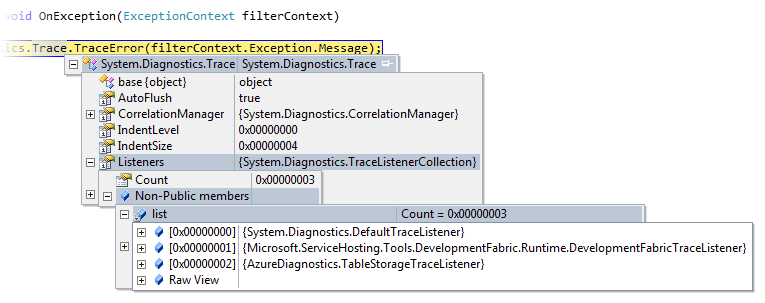
(Code Snippet – WindowsAzureDebugging-Ex1-WebRole RoleEnvironmentChanging event handler-CS)

* + 1. C#
    2. public class WebRole : RoleEntryPoint
    3. {
    4. ...
    5. private void RoleEnvironmentChanging(object sender, RoleEnvironmentChangingEventArgs e)
    6. {
    7. **// for any configuration setting change except EnableTableStorageTraceListener**
    8. **if (e.Changes.OfType<RoleEnvironmentConfigurationSettingChange>().Any(change => change.ConfigurationSettingName != "EnableTableStorageTraceListener"))**
    9. **{**
    10. **// Set e.Cancel to true to restart this role instance**
    11. **e.Cancel = true;**
    12. **}**
    13. }
    14. ...
    15. }
    16. (Code Snippet – *WindowsAzureDebugging-Ex1-WebRole RoleEnvironmentChanging event handler-VB*)
    17. Visual Basic
    18. Public Class WebRole
    19. Inherits RoleEntryPoint
    20. ...
    21. Private Sub RoleEnvironmentChanging(ByVal sender As Object, ByVal e As RoleEnvironmentChangingEventArgs)
    22. **' for any configuration setting change except EnableTableStorageTraceListener**
    23. **If e.Changes.OfType(Of RoleEnvironmentConfigurationSettingChange)().Any(Function(change) change.ConfigurationSettingName <> "EnableTableStorageTraceListener") Then**
    24. **' Set e.Cancel to true to restart this role instance**
    25. **e.Cancel = True**
    26. **End If**
    27. End Sub
    28. ...
    29. End Class
    30. **Note:** The **RoleEnvironmentChanging** event occurs before a change to the service configuration is applied to the running instances of the role. The updated handler scans the collection of changes and restarts the role instance for any configuration setting change, unless the change only involves the value of the *EnableTableStorageTraceListener* setting. If this particular setting changes, the role instance is allowed to apply the change without restarting it.
  1. Now, add the following (highlighted) code to define a handler for the **RoleEnvironmentChanged** event into the **Global.asax.cs** (for Visual C# projects) or **Global.asax.vb** (for Visual Basic projects).

(Code Snippet – WindowsAzureDebugging-Ex1-Global RoleEnvironmentChanged event handler-CS)

* + 1. C#
    2. public class MvcApplication : System.Web.HttpApplication
    3. {
    4. ...
    5. **private void RoleEnvironmentChanged(object sender, RoleEnvironmentChangedEventArgs e)**
    6. **{**
    7. **// configure trace listener for any changes to EnableTableStorageTraceListener**
    8. **if (e.Changes.OfType<RoleEnvironmentConfigurationSettingChange>().Any(change => change.ConfigurationSettingName == "EnableTableStorageTraceListener"))**
    9. **{**
    10. **ConfigureTraceListener();**
    11. **}**
    12. **}**
    13. ...
    14. }
    15. (Code Snippet – *WindowsAzureDebugging-Ex1-Global RoleEnvironmentChanged event handler-VB*)
    16. Visual Basic
    17. Public Class MvcApplication
    18. Inherits System.Web.HttpApplication
    19. ...
    20. **Private Sub RoleEnvironmentChanged(ByVal sender As Object, ByVal e As RoleEnvironmentChangedEventArgs)**
    21. **' configure trace listener for any changes to EnableTableStorageTraceListener**
    22. **If e.Changes.OfType(Of RoleEnvironmentConfigurationSettingChange)().Any(Function(change) change.ConfigurationSettingName = "EnableTableStorageTraceListener") Then**
    23. **ConfigureTraceListener()**
    24. **End If**
    25. **End Sub**
    26. ...
    27. End Class
    28. **Note:** The **RoleEnvironmentChanged** event handler occurs after a change to the service configuration has been applied to the running instances of the role. If this change involves the *EnableTableStorageTraceListener* configuration setting, the handler calls the **ConfigureTraceListener** method to enable or disable the trace listener.
  1. Finally, insert the following (highlighted) line into the **Application\_Start** method, immediately after to the call to the **ConfigureTraceListener** method, to subscribe to the **Changed** event of the **RoleEnvironment**.
     1. C#
     2. public class MvcApplication : System.Web.HttpApplication
     3. {
     4. ...
     5. protected void Application\_Start()
     6. {
     7. CloudStorageAccount.SetConfigurationSettingPublisher((configName, configSetter) =>
     8. {
     9. configSetter(RoleEnvironment.GetConfigurationSettingValue(configName));
     10. });
     11. ConfigureTraceListener();
     13. **RoleEnvironment.Changed += RoleEnvironmentChanged;**
     15. AreaRegistration.RegisterAllAreas();
     16. RegisterRoutes(RouteTable.Routes);
     17. }
     18. ...
     19. }
     20. Visual Basic
     21. Public Class MvcApplication
     22. Inherits System.Web.HttpApplication
     23. ...
     24. Sub Application\_Start()
     26. CloudStorageAccount.SetConfigurationSettingPublisher(Sub(configName, configSetter) configSetter(RoleEnvironment.GetConfigurationSettingValue(configName)))
     27. ConfigureTraceListener()
     28. **AddHandler RoleEnvironment.Changed, AddressOf RoleEnvironmentChanged**
     29. AreaRegistration.RegisterAllAreas()
     30. RegisterRoutes(RouteTable.Routes)
     31. End Sub
     32. ...
     33. End Class
  2. To instrument the application and write diagnostics information to the error log, add a global error handler to the application. To do this, insert the following method into the **MVCApplication** class.

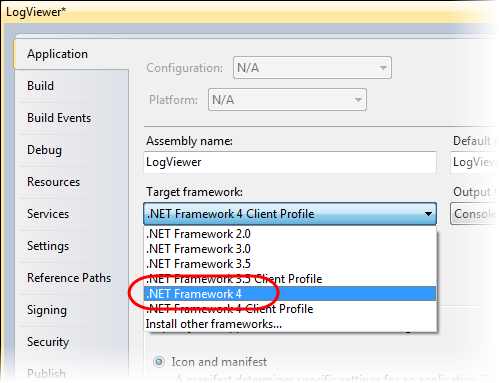
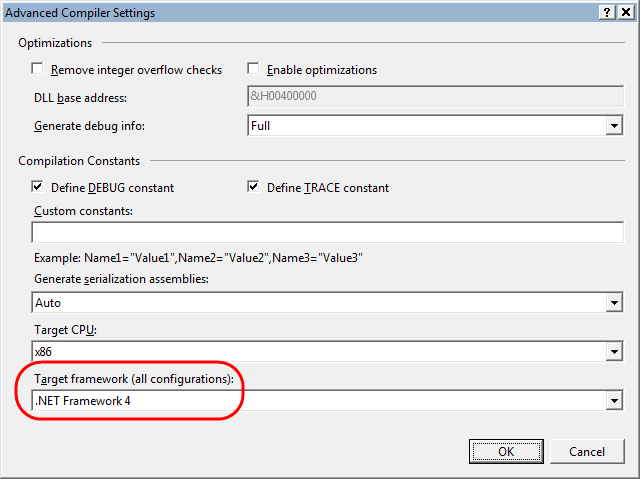
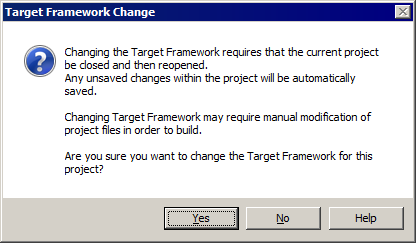
(Code Snippet – WindowsAzureDebugging-Ex1-Application\_Error-CS)

* + 1. C#
    2. public class MvcApplication : System.Web.HttpApplication
    3. {
    4. ...
    5. **protected void Application\_Error()**
    6. **{**
    7. **var lastError = Server.GetLastError();**
    8. **System.Diagnostics.Trace.TraceError(lastError.Message);**
    9. **}**
    10. }
    11. (Code Snippet – *WindowsAzureDebugging-Ex1-Application\_Error-VB*)
    12. Visual Basic
    13. Public Class MvcApplication
    14. Inherits System.Web.HttpApplication
    15. ...
    16. **Protected Sub Application\_Error()**
    17. **Dim lastError = Server.GetLastError()**
    18. **System.Diagnostics.Trace.TraceError(lastError.Message)**
    19. **End Sub**
    20. End Class
    21. **Note:** The **Application\_Error** event is raised to catch any unhandled ASP.NET errors while processing a request. The event handler shown above retrieves a reference to the unhandled exception object using **Server.GetLastError** and then uses the **TraceError** method of the **System.Diagnostics.Trace** class to log the error message.
    22. Note that the **Trace** object outputs the message to each listener in its **Listeners** collection, including the **TableStorageTraceListener**, provided you enable it in the configuration settings. Typically, the collection also contains instances of the **DefaultTraceListener** class and, when executing the solution in the compute emulator, the **DevelopmentFabricTraceListener**. The latter writes its output to a log that you can view from the Compute Emulator UI.
    23. To write to the Windows Azure diagnostics log, a **DiagnosticMonitorTraceListener** can also be added to the **Web.config** or **App.config** file of the role. When using this type of trace listener, the logs are gathered locally in each role. To retrieve them, you first need to instruct the diagnostic monitor to copy the information to storage services. The role project templates included with the Windows Azure Tools for Microsoft Visual Studio already include the settings required to use the **DiagnosticMonitorTraceListener** in the configuration files it generates.
    24. 
    25. Figure
    26. Trace object Listeners collection showing configured trace listeners
  1. Open the **QuoteController.cs** (for Visual C# projects) or **QuoteController.vb** (for Visual Basic projects) file in the **Controllers** folder of the **FabrikamInsurance** project and add the following method.

(Code Snippet – WindowsAzureDebugging-Ex1-Controller OnException method-CS)

* + 1. C#
    2. [HandleError]
    3. public class QuoteController : Controller
    4. {
    5. ...
    6. **protected override void OnException(ExceptionContext filterContext)**
    7. **{**
    8. **System.Diagnostics.Trace.TraceError(filterContext.Exception.Message);**
    9. **}**
    10. }
    11. (Code Snippet – *WindowsAzureDebugging-Ex1-Controller OnException method-VB*)
    12. Visual Basic
    13. <HandleError()>
    14. Public Class QuoteController
    15. Inherits Controller
    16. ...
    17. **Protected Overrides Sub OnException(ByVal filterContext As ExceptionContext)**
    18. **System.Diagnostics.Trace.TraceError(filterContext.Exception.Message)**
    19. **End Sub**
    20. End Class
    21. **Note:** The **OnException** method is called when an unhandled exception occurs during the processing of an action in a controller. For MVC applications, unhandled errors are typically caught at the controller level, provided they occur during the execution of a controller action and that the action (or controller) has been decorated with a **HandleErrorAttribute**. To log exceptions in controller actions, you need to override the **OnException** method of the controller because the **Application\_Error** is bypassed when the error-handling filter catches the exceptions.
    22. By default, when an action method with the **HandleErrorAttribute** attribute throws any exception, MVC displays the **Error** view that is located in the **~/Views/Shared** folder.
  1. In addition to error logging, tracing can also be useful for recording other significant events during the execution of the application. For example, for registering whenever a given controller action is invoked. To show this feature, insert the following (highlighted) tracing statement at the start of the **Calculator** method to log a message whenever this action is called.
     1. C#
     2. public class QuoteController : Controller
     3. {
     4. ...
     5. public ActionResult Calculator()
     6. {
     7. **System.Diagnostics.Trace.TraceInformation("Calculator called...");**
     8. QuoteViewModel model = new QuoteViewModel();
     9. PopulateViewModel(model, null);
     10. return View(model);
     11. }
     12. ...
     13. }
     14. Visual Basic
     15. Public Class QuoteController
     16. Inherits Controller
     17. ...
     18. Public Function Calculator() As ActionResult
     19. **System.Diagnostics.Trace.TraceInformation("Calculator called...")**
     20. Dim model As New QuoteViewModel()
     21. PopulateViewModel(model, Nothing)
     22. Return View(model)
     23. End Function
     24. ...
     25. End Class
  2. Similarly, add a tracing statement to the **About** action, as shown (highlighted) below.
     1. C#
     2. public class QuoteController : Controller
     3. {
     4. ...
     5. public ActionResult About()
     6. {
     7. **System.Diagnostics.Trace.TraceInformation("About called...");**
     8. return View();
     9. }
     10. ...
     11. }
     12. Visual Basic
     13. Public Class QuoteController
     14. Inherits Controller
     15. ...
     16. Public Function About() As ActionResult
     17. **System.Diagnostics.Trace.TraceInformation("About called...")**
     18. Return View()
     19. End Function
     20. ...
     21. End Class

Task 2 – Creating a Log Viewer Tool

* 1. At this point, the application is ready for tracing and can send all its diagnostics output to a table in storage services. To view the trace logs, you now create a simple log viewer application that will periodically query the table and retrieve all entries added since it was last queried.
  2. Add a new console application project to the solution. To create the project, in the **File** menu, point to **Add**, and then select **New Project**. In the **Add New Project** dialog, expand node for the language of your choice (Visual C# or Visual Basic) in the **Installed Templates** tree view, select the **Windows** category, and then the **Console Application** template. Set the name of the project to **LogViewer**, accept the proposed location inside the solution folder, and then click **OK**.
  3. Right-click the new **LogViewer** project in **Solution Explorer** and select **Properties**.
     1. **For Visual C# projects**:
     2. In the properties window, switch to the **Application** page, and then change the **Target framework** to *.NET Framework 4*.
     3. 
     4. Figure
     5. Configuring the target framework for the project (Visual C#)
     6. **For Visual Basic projects:**
     7. In the properties window, switch to the **Compile** page and then click **Advanced Compile Options**. In the **Advanced Compiler Settings** dialog, select *.NET Framework 4* in the **Target framework** drop down list, and then click **OK**.
     8. 
     9. Figure
     10. Configuring the target framework for the project (Visual Basic)
     11. **Note:** The client profile is not suitable in this case because the application will use the StorageClient API to retrieve log data from table storage. This API relies on functionality available only in the full .NET Framework 4 distribution.
  4. If the **Target Framework Change** dialog appears, click **Yes**.
     1. 
     2. Figure
     3. Target Framework Change
  5. Add references to the assemblies required by this project. To do this, in **Solution Explorer**, right-click the **LogViewer** project and select **Add Reference**. In the **Add Reference** dialog, switch to the **.NET** tab and, while holding down the **CTRL** key to select multiple items, select **System.Configuration**, **Microsoft.WindowsAzure.StorageClient**, and **System.Data.Services.Client**, and then click **OK**.
  6. Next, add a reference to the diagnostics project in the solution. Repeat the previous step to open the **Add Reference** dialog, only this time select the **Projects** tab, select the **AzureDiagnostics** project and click **OK**.
  7. Add a class to display a simple progress indicator in the console window to the project. To do this, in **Solution Explorer**, right-click **LogViewer**, point to **Add**, and select **Existing Item**. In the **Add Existing Item** dialog, browse to **Assets** in the **Source** folder of the lab, select the folder for the language of the project (Visual C# or Visual Basic), select the **ProgressIndicator.[cs|.vb]** file, and then click **Add**.
  8. In **Solution Explorer**, double-click **Program.cs** or **Module1.vb** to open this file and insert the following namespace declarations at the top of the file.

(Code Snippet – WindowsAzureDebugging-Ex1-LogViewer namespaces-CS)

* + 1. C#
    2. **using System.Configuration;**
    3. **using System.Data.Services.Client;**
    4. **using System.Threading;**
    5. **using Microsoft.WindowsAzure;**
    6. **using Microsoft.WindowsAzure.StorageClient;**
    7. **using AzureDiagnostics;**
    8. (Code Snippet – *WindowsAzureDebugging-Ex1-LogViewer namespaces-VB*)
    9. Visual Basic
    10. **Imports System.Configuration**
    11. **Imports System.Data.Services.Client**
    12. **Imports System.Threading**
    13. **Imports Microsoft.WindowsAzure**
    14. **Imports Microsoft.WindowsAzure.StorageClient**
    15. **Imports AzureDiagnostics**
  1. For Visual Basic projects only, reformulate the **Sub Main** making it **Public** and adding a string array parameter named **args**.
     1. Visual Basic
     2. Module Module1
     3. **Public Sub Main(ByVal args() As String)**
     4. End Sub
     5. End Module
  2. Define the following (highlighted) members in the **Program** class (for Visual C# projects) or the **Module1** module (for Visual Basic projects).
     1. (Code Snippet – *WindowsAzureDebugging-Ex1-LogViewer static members-CS*)
     2. C#
     3. class Program
     4. {
     5. **private static string lastPartitionKey = String.Empty;**
     6. **private static string lastRowKey = String.Empty;**
     7. static void Main(string[] args)
     8. {
     9. }
     10. }
     11. (Code Snippet – *WindowsAzureDebugging-Ex1-LogViewer static members-VB*)
     12. Visual Basic
     13. Module Module1
     14. **Private lastPartitionKey As String = String.Empty**
     15. **Private lastRowKey As String = String.Empty**
     16. Public Sub Main(ByVal args() As String)
     17. End Sub
     18. End Module
  3. Next, insert the **QueryLogTable** method into the class or module.

(Code Snippet – WindowsAzureDebugging-Ex1-QueryLogTable method-CS)

* + 1. C#
    2. class Program
    3. {
    4. ...
    5. **private static void QueryLogTable(CloudTableClient tableStorage)**
    6. **{**
    7. **TableServiceContext context = tableStorage.GetDataServiceContext();**
    8. **DataServiceQuery query = context.CreateQuery<LogEntry>(TableStorageTraceListener.DIAGNOSTICS\_TABLE)**
    9. **.Where(entry => entry.PartitionKey.CompareTo(lastPartitionKey) > 0**
    10. **|| (entry.PartitionKey == lastPartitionKey && entry.RowKey.CompareTo(lastRowKey) > 0))**
    11. **as DataServiceQuery;**
    12. **foreach (AzureDiagnostics.LogEntry entry in query.Execute())**
    13. **{**
    14. **Console.WriteLine("{0} - {1}", entry.Timestamp, entry.Message);**
    15. **lastPartitionKey = entry.PartitionKey;**
    16. **lastRowKey = entry.RowKey;**
    17. **}**
    18. **}**

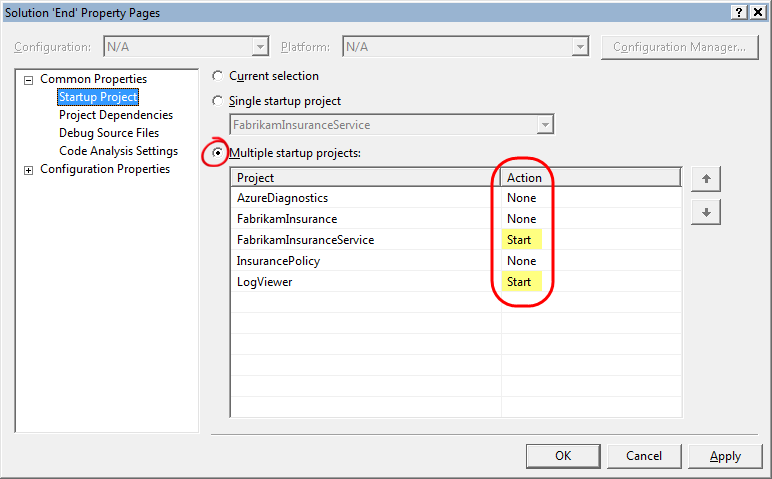
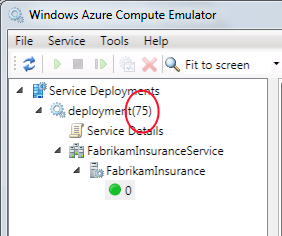
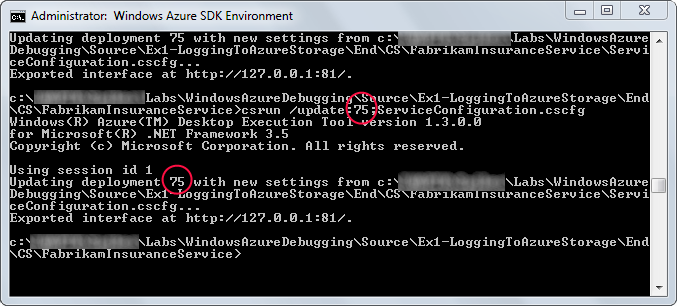
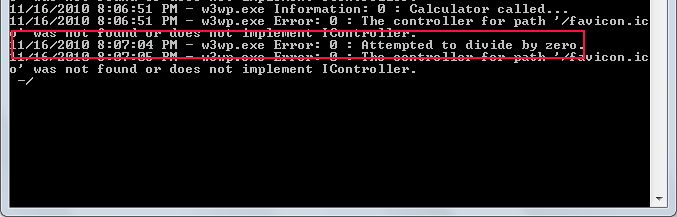
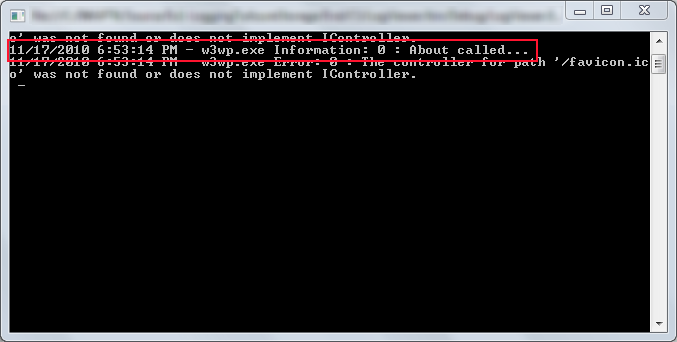
...

* + 1. }
    2. (Code Snippet – *WindowsAzureDebugging-Ex1-QueryLogTable method-VB*)
    3. Visual Basic
    4. Module Module1
    5. ...
    6. **Private Sub QueryLogTable(ByVal tableStorage As CloudTableClient)**
    7. **Dim context As TableServiceContext = tableStorage.GetDataServiceContext()**
    8. **Dim query As DataServiceQuery = TryCast(context.CreateQuery(Of LogEntry)(TableStorageTraceListener.DIAGNOSTICS\_TABLE).Where(Function(entry) entry.PartitionKey.CompareTo(lastPartitionKey) > 0 OrElse (entry.PartitionKey = lastPartitionKey AndAlso entry.RowKey.CompareTo(lastRowKey) > 0)), DataServiceQuery)**
    9. **For Each entry As AzureDiagnostics.LogEntry In query.Execute()**
    10. **Console.WriteLine("{0} - {1}", entry.Timestamp, entry.Message)**
    11. **lastPartitionKey = entry.PartitionKey**
    12. **lastRowKey = entry.RowKey**
    13. **Next**
    14. **End Sub**
    15. ...
    16. End Module
    17. **Note:** The rows in the diagnostic log table are stored with a primary key composed by the partition and row key properties, where both are based on the event tick count of the corresponding log entry and are thus ordered chronologically. The **QueryLogTable** method queries the table to retrieve all rows whose primary key value is greater than the last value obtained during the previous invocation of this method. This ensures that each time it is called, the method only retrieves new entries added to the log.
  1. Finally, to complete the changes, insert the following (highlighted) code into the body of method **Main**.

(Code Snippet – WindowsAzureDebugging-Ex1-LogViewer Main method-CS)

* + 1. C#
    2. class Program
    3. {
    4. ...
    5. static void Main(string[] args)
    6. {
    7. **string connectionString = (args.Length == 0) ? "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString" : args[0];**
    8. **CloudStorageAccount account = CloudStorageAccount.Parse(ConfigurationManager.AppSettings[connectionString]);**
    9. **CloudTableClient tableStorage = account.CreateCloudTableClient();**
    10. **tableStorage.CreateTableIfNotExist(TableStorageTraceListener.DIAGNOSTICS\_TABLE);**
    11. **Utils.ProgressIndicator progress = new Utils.ProgressIndicator();**
    12. **Timer timer = new Timer((state) =>**
    13. **{**
    14. **progress.Disable();**
    15. **QueryLogTable(tableStorage);**
    16. **progress.Enable();**
    17. **}, null, 0, 10000);**
    18. **Console.ReadKey(true);**
    19. }
    20. }
    21. (Code Snippet – *WindowsAzureDebugging-Ex1-LogViewer Main method-VB*)
    22. Visual Basic
    23. Module Module1
    24. ...
    25. Public Sub Main(ByVal args() As String)
    26. **Dim connectionString As String = If((args.Length = 0), "Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString", args(0))**
    27. **Dim account As CloudStorageAccount = CloudStorageAccount.Parse(ConfigurationManager.AppSettings(connectionString))**
    28. **Dim tableStorage As CloudTableClient = account.CreateCloudTableClient()**
    29. **tableStorage.CreateTableIfNotExist(TableStorageTraceListener.DIAGNOSTICS\_TABLE)**
    30. **Dim progress As New ProgressIndicator()**
    31. **Dim timer As New Timer(Sub(state)**
    32. **progress.Disable()**
    33. **QueryLogTable(tableStorage)**
    34. **progress.Enable()**
    35. **End Sub, Nothing, 0, 10000)**
    36. **Console.ReadKey(True)**
    37. End Sub
    38. End Module
    39. **Note:** The inserted code initializes the Windows Azure storage account information, creates the diagnostics table if necessary, and then starts a timer that periodically calls the **QueryLogMethod** defined in the previous step to display new entries in the diagnostics log.
  1. To complete the viewer application, open the **App.config** file in the **LogViewer** project and insert the following (highlighted) **appSettings** section to define the *DiagnosticsConnectionString* setting required to initialize the storage account information.
     1. XML
     2. <configuration>
     3. ...
     4. **<appSettings>**
     5. **<add key="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString" value="UseDevelopmentStorage=true"/>**
     6. **</appSettings>**
     7. <startup>
     8. <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0" />
     9. </startup>
     10. </configuration>

Verification

* 1. You are now ready to execute the solution in the compute emulator. To enable the Table Storage trace listener dynamically without stopping the running service, you initially deploy the service with the *EnableTraceStorageTraceListener* setting disabled and then, you change the setting in the configuration file to enable the listener and then upload it to re-configure the running service. Using the log viewer application, you examine the trace messages produced by the application.
  2. Open the **Web.config** file of the **FabrikamInsurance** project and insert the following (highlighted) **customErrors** section as a direct child of the **system.web** element.
     1. XML
     2. <configuration>
     3. ...
     4. <system.web>
     5. ...
     6. **<customErrors mode="On" />**
     7. </system.web>
     8. ...
     9. </configuration>
     10. **Note:** When you set the **customErrors** mode to *On*, ASP.NET displays generic error messages for both local and remote clients. With **customErrors** set to its default setting of *RemoteOnly*, once the application is deployed to Windows Azure and you access it remotely, you will also see the generic errors, so this step is not strictly necessary. However, it allows you to reproduce locally the behavior that you would observe once you deploy the application to the cloud.
  3. To test the solution, you need to configure the Windows Azure Project and the log viewer application so that they both start simultaneously. To define the start up projects, right-click the solution node in **Solution Explorer** and select **Set StartUp Projects**. In the **Solution ‘Begin’ Property Pages** window, make sure to select **Startup Project** under **Common Properties**, and then select the option labeled **Multiple startup projects**. Next, set the **Action** for both the **LogViewer** and **FabrikamInsuranceService** projects to *Start*, leaving the remaining projects as *None*. Click **OK** to save the changes to the start-up configuration.
     1. 
     2. Figure
     3. Configuring the start-up projects for the solution
  4. Press **CTRL + F5** to launch the application without attaching a debugger. Again, this reproduces the conditions that you would have once you deploy the application to the cloud. Wait until the deployment completes and the browser opens to show its main page.
  5. In the browser window, complete the form making sure that you choose “*PORSCHE”* for the **Make** of the vehicle and “*BOXSTER (BAD DATA)*” for the **Model**. Notice that this time, because you enabled the **customErrors** setting in the **Web.config** file, the application shows a generic error page instead of the exception details that you saw earlier. This is what you would also see had the application been deployed to Windows Azure.
     1. 
     2. Figure
     3. Application error with customErrors enabled
  6. Examine the output from the log viewer application. Notice that, despite the error, the console window is still empty because the table storage trace listener is currently disabled.
  7. Switch back to Visual Studio and, in **Solution Explorer**, expand the **Roles** node of the **FabrikamInsuranceService** project, and then double-click the **FabrikamInsurance** role to open its properties window. Select the **Settings** page, and then change the value of the *EnableTableStorageTraceListener* setting to *true*.
  8. Press **CTRL + S** to save the changes to the configuration.
  9. Open the compute emulator console by right-clicking its icon located in the system tray and selecting **Show Compute Emulator UI**. Record the ID for the current deployment. This is the numeric value shown enclosed in parenthesis, next to the deployment label.
     1. 
     2. Figure
     3. Compute Emulator UI showing the current deployment ID
  10. Now, open a Windows Azure SDK command prompt from **Start | All Programs | Windows Azure SDK v1.X | Windows Azure SDK Command Prompt**. To launch the command prompt as an administrator, right-click its shortcut in the **Start** menu and choose **Run as administrator**.
  11. Change the current directory to the location of the **FabrikamInsuranceService** cloud project inside the current solution’s folder. This folder contains the service configuration files, select ServiceConfiguration.Local.cscfg.
  12. At the command prompt, execute the following command to update the configuration of the running deployment. Replace the [DEPLOYMENTID] placeholder with the value that you recorded earlier.
      1. Windows Azure Command Prompt
      2. **csrun /update:[DEPLOYMENTID];ServiceConfiguration.Local.cscfg**
      3. 
      4. Figure
      5. Updating the configuration of the running service
      6. **Note:** For applications deployed to the cloud, you would normally update the configuration of your running application through the Windows Azure Developer Portal or by using the Windows Azure Management API to upload a new configuration file.
  13. Once you have updated the configuration and enabled the trace listener, return to the browser window, browse to the **Quotes** page, and re-enter the same parameters that caused the error previously (make *“PORSCHE”*, model *“BOXSTER (BAD DATA)”*). Then, click **Calculate** to submit the form again. The response should still show the error page.
  14. Switch to the log viewer window and wait a few seconds until it refreshes. Notice that the console now shows an entry with the error message for the unhandled exception, showing that the trace output generated by the running application is written directly to table storage.
      1. 
      2. Figure
      3. Viewer showing the error logged to table storage
  15. To view the output from other informational trace messages, return to the browser window and click **About** followed by **Quotes** to execute both actions in the controller. Recall that you inserted trace messages at the start of each method. Notice that the viewer console now displays a message for each of these actions.
      1. 
      2. Figure
      3. Viewer showing informational trace messages for the controller actions
  16. In the log viewer window, press any key to exit the program.
  17. Finally, delete the running deployment in the compute emulator. To do this, right-click the deployment in the **Service Deployments** tree view and select **Remove**.

Summary

* 1. By completing this hands-on lab, you learnt how to apply simple debugging techniques to troubleshoot your Windows Azure application once you deploy it to the cloud. You saw how to use standard .NET diagnostics to write diagnostics output directly into table storage with a custom trace listener.