WorkshopPLUS

Microsoft Azure Service Fabric for Developers

Introduction to Azure Service Fabric

Student Lab Manual

V3.2, March 15, 2017

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, MS, Windows, Windows NT, MSDN, Active Directory, BizTalk, SQL Server, SharePoint, Outlook, PowerPoint, FrontPage, Visual Basic, Visual C++, Visual J++, Visual InterDev, Visual SourceSafe, Visual C#, Visual J#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Contents

[Introduction to Azure Service Fabric 4](#_Toc474688629)

[Introduction 4](#_Toc474688630)

[Objectives 4](#_Toc474688631)

[Prerequisites 4](#_Toc474688632)

[Exercise 1: Installing the Azure Service Fabric SDK and Creating the Local Cluster 5](#_Toc474688633)

[Task 1 – Installing the Azure Service Fabric SDK 5](#_Toc474688634)

[Task 2 - Creating a new Service Fabric Local Cluster 6](#_Toc474688635)

[Exercise 2: Creating and Deploying the Service Fabric Application 7](#_Toc474688636)

[Overview 7](#_Toc474688637)

[Scenario 7](#_Toc474688638)

[Task 1 – Create a stateful service 7](#_Toc474688639)

[Task 2 – Adding Reliable Dictionaries and Transactions 13](#_Toc474688640)

[Task 3 – Viewing the Replica Endpoints 18](#_Toc474688641)

[Task 4 – Upgrading the named application 21](#_Toc474688642)

[Task 5 – Failover and PowerShell 24](#_Toc474688643)

# Introduction to Azure Service Fabric

## Introduction

Estimated time to complete this lab

60 minutes

#### Overview

Azure Service Fabric is a distributed systems platform used to build scalable, reliable, and easily managed applications for the cloud. Service Fabric addresses the significant challenges in developing and managing cloud applications. By using Service Fabric, developers and administrators can avoid solving complex infrastructure problems and focus instead on implementing mission-critical, demanding workloads knowing that they are scalable, reliable, and manageable. Service Fabric represents the next-generation middleware platform for building and managing these enterprise-class, Tier-1 cloud-scale services.

### Objectives

In this hands-on lab, you will learn how to:

* Install the Azure Service Fabric SDK
* Create a new local Service Fabric Cluster
* How to create a stateful reliable service with a custom listener
* How to use a Reliable dictionary
* How to Upgrade the application in the Service Fabric
* How to failover using PowerShell commands

## Prerequisites

The following is required to complete this hands-on lab:

* Microsoft V[isual Studio 2017 Professional or Enterprise edition](http://www.microsoft.com/visualstudio/)
* [Microsoft Azure SDK for .NET for Visual Studio 2017](http://www.microsoft.com/windowsazure/sdk/)
* Microsoft Azure Service Fabric SDK – 2.4.164
* A Microsoft Azure subscription

Please note, you will use the same Resource Group for all Labs.

# Exercise 1: Installing the Azure Service Fabric SDK and Creating the Local Cluster

In this exercise, you will learn how to create and use a stateful reliable actor. You will create this actor by using the Service Fabric Application template from Visual Studio. You will then proceed to add other pre-existing .Net projects that the actor object will use.

## Task 1 – Installing the Azure Service Fabric SDK

NOTE: If you have already installed the Azure Service Fabric SDK and created a local cluster on your machine, please move to Exercise 2.

1. Open the Microsoft Web Platform Installer.
2. Install the latest version of the Microsoft Azure Service Fabric SDK. If you are prompted to permissions to perform the install, accept the prompts.

NOTE: For Visual Studio 2017, you would install Microsoft Azure Service Fabric SDK – 2.4.164. For Visual Studio 2015, you would install Microsoft Azure Service Fabric SDK and Tools – 2.4.164 (VS2015).

## Task 2 - Creating a new Service Fabric Local Cluster

1. Using Windows Explorer, browse to *C:\Program Files\Microsoft SDKs\Service Fabric\Tools\ServiceFabricLocalClusterManage*r and then double-click on **ServiceFabricLocalClusterManager.exe**. This will start up the local cluster manager.
2. Click on the hidden icons link on the taskbar and then right-click on the **Service Fabric Local Cluster Manager** icon. From here you will select **Setup Local Cluster**. It will take a few minutes to build a local cluster. Choose the 5 node option.

|  |  |
| --- | --- |
|  |  |

NOTE: Is it necessary to setup your local cluster using the method above? No, but the first app you try to deploy to your local cluster will have to create the cluster and start it for you, which can take an extended amount of time.

# Exercise 2: Creating and Deploying the Service Fabric Application

## Overview

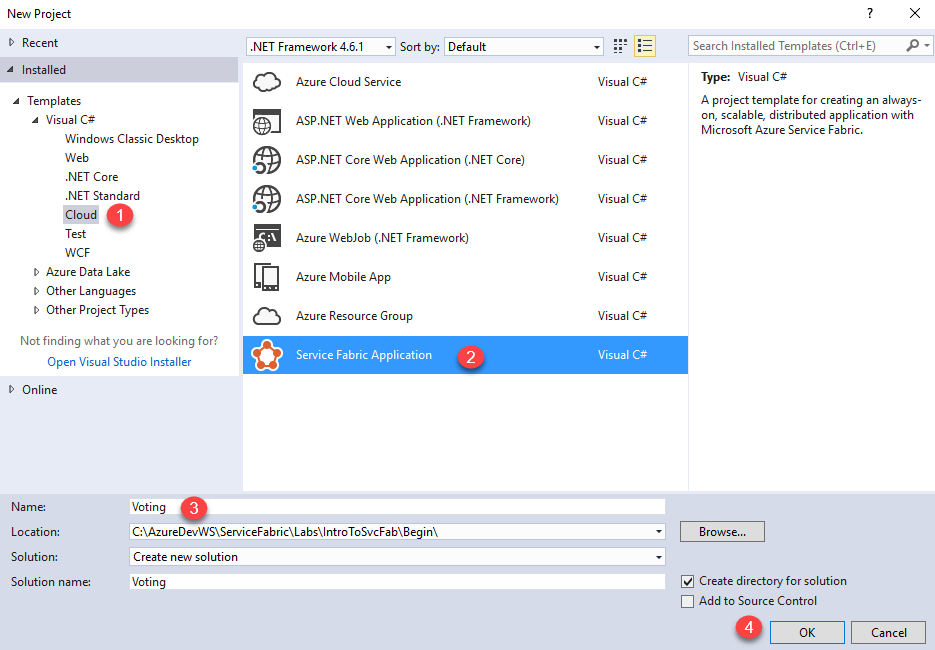
This lab is designed to apply the knowledge you have gained throughout the Service Fabric developer training module. The goal of this lab is to make you familiar with an end-to-end development flow for Service Fabric applications. You will practice working with stateful services, reliable dictionaries and transactions. Throughout the exercise, you will get accustomed with Visual Studio’s Service Fabric tooling, Service Fabric Explorer and learn how to effectively use both.

## Scenario

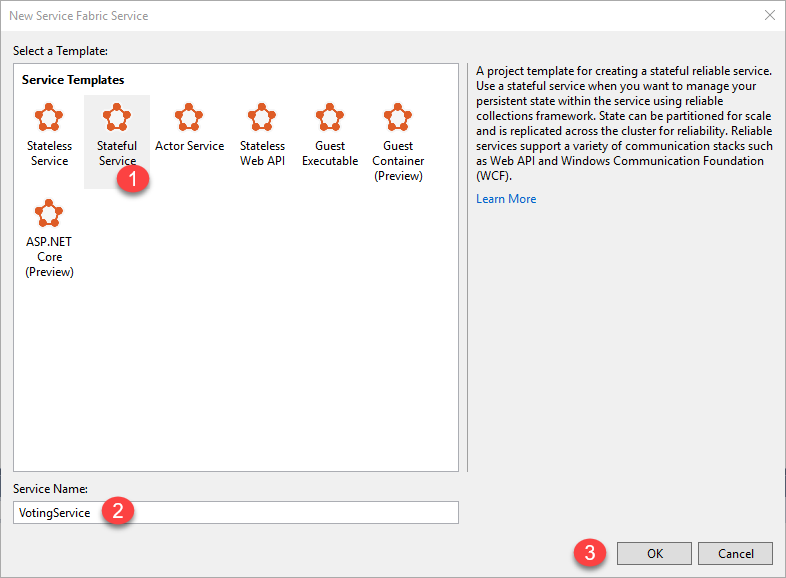
In this scenario, you will build a generic voting service using Service Fabric reliable services. The service listens to an endpoint accessible from a Web browser. You’ll enter vote item strings (such as favorite sodas or cars) by passing in an HTTP query string parameter. Each time the same vote item is passed, a counter is incremented. The incrementing of the counter represents the number of times the item has been voted for. Each HTML response contains all the votes items and the number of times that vote item was voted for.

## Task 1 – Create a stateful service

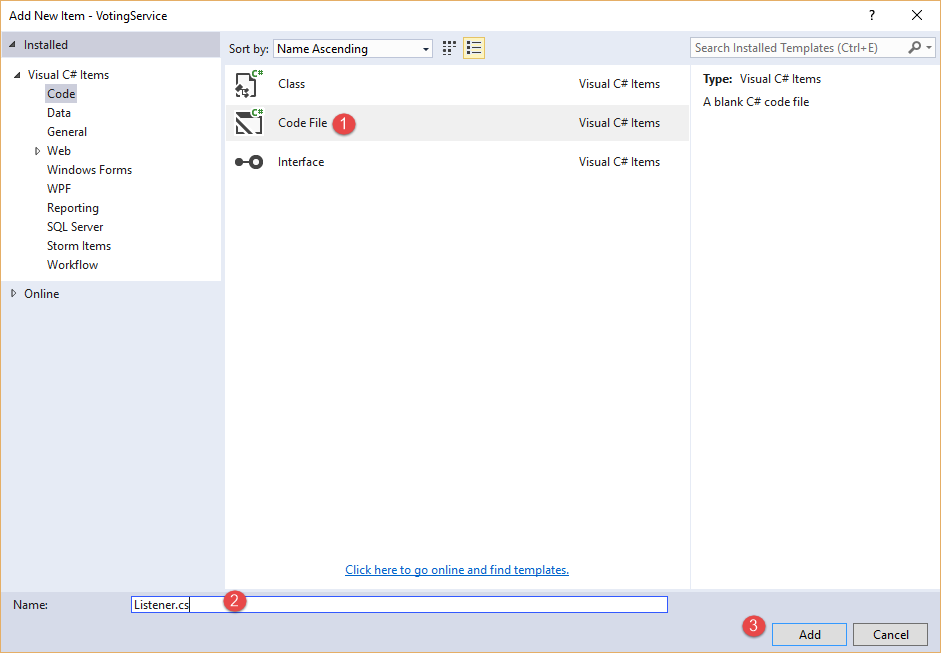
1. Open Visual Studio with Administrator privileges.
2. Select **File | New | Project**.
3. Select **Templates | Visual C# | Cloud | Service Fabric Application.**
4. Enter **Voting** for the Name and the Solution name and set the Location to .**ServiceFabric\Labs\IntroToSvcFabric\Begin**. Then select **Ok**.



1. Select the **Stateful Service** template, enter **VotingService** in **Project Name**, and then click **OK**.



1. For the service, you will need a listener to listen for incoming calls. In the Solution Explorer, right click on the **VotingService** project and then select **Add | New Item** and select to add a new *Code File*. Name the file **Listener.cs**. Select the **Add** button.



1. Copy the code below into the file. This code wraps a .NET HttpListener, allowing Service Fabric to control it. Exactly what this code does is unimportant for now (you can study it later if you wish). When a service replica starts, this listener object is created allowing the replica to process incoming HTTP requests.

using Microsoft.ServiceFabric.Services.Communication.Runtime;

using System;

using System.Fabric;

using System.Fabric.Description;

using System.Net;

using System.Threading;

using System.Threading.Tasks;

using System.Net;

using System.Text;

using Microsoft.ServiceFabric.Data;

public sealed class HttpCommunicationListener : ICommunicationListener

{

private readonly String m\_uriPublished;

private readonly HttpListener m\_httpListener = new HttpListener();

private readonly Func<HttpListenerContext, CancellationToken, Task> m\_processRequest;

private readonly CancellationTokenSource m\_processRequestsCancellation = new CancellationTokenSource();

public HttpCommunicationListener(StatefulServiceContext context, string endpointName, Func<HttpListenerContext, CancellationToken, Task> processRequest)

{

m\_processRequest = processRequest;

// Partition replica's URL is the node's IP, desired port, PartitionId, ReplicaId, Guid

EndpointResourceDescription internalEndpoint = context.CodePackageActivationContext.GetEndpoint(endpointName);

var uriPrefix = $"{internalEndpoint.Protocol}://+:{internalEndpoint.Port}/"

+ $"{context.PartitionId}/{context.ReplicaId}"

+ $"-{Guid.NewGuid()}/"; // Uniqueness

m\_httpListener.Prefixes.Add(uriPrefix);

m\_uriPublished = uriPrefix.Replace("+", FabricRuntime.GetNodeContext().IPAddressOrFQDN);

}

public void Abort()

{

m\_processRequestsCancellation.Cancel(); m\_httpListener.Abort();

}

public Task CloseAsync(CancellationToken cancellationToken)

{

m\_processRequestsCancellation.Cancel();

m\_httpListener.Close(); return Task.FromResult(true);

}

public Task<string> OpenAsync(CancellationToken cancellationToken)

{

m\_httpListener.Start();

var noWarning = ProcessRequestsAsync(m\_processRequestsCancellation.Token);

return Task.FromResult(m\_uriPublished);

}

private async Task ProcessRequestsAsync(CancellationToken processRequests)

{

while (!processRequests.IsCancellationRequested)

{

HttpListenerContext request = await m\_httpListener.GetContextAsync();

// The ContinueWith forces rethrowing the exception if the task fails.

var noWarning = m\_processRequest(request, m\_processRequestsCancellation.Token)

.ContinueWith(async t => await t /\* Rethrow unhandled exception \*/, TaskContinuationOptions.OnlyOnFaulted);

}

}

}

1. Open **VotingService.cs** in the VotingService project and replace **CreateServiceReplicaListeners’s** method body with the following code. When the Service Fabric starts the replica, it calls CreateServiceReplicaListeners. Here you return the set of listeners that you want Service Fabric to open. In our case, you want our replica to have just 1 HTTP listener and when each HTTP client request comes in, it will be processed by the ProcessRequestAsync method:

return new ServiceReplicaListener[] { new ServiceReplicaListener(p => new HttpCommunicationListener(p, "ServiceEndpoint", ProcessRequestAsync)) };

Note: At this point the **ProcessRequestAsync** will render as an error as you have not added this code yet.

1. In the **VotingService** project, open the **ServiceManifest.xml** file (in the *PackageRoot* folder) and **update** the Endpoint element with the “ServiceEndpoint” endpoint by adding a Port attribute with a value of 8080 and a Protocol attribute of http.

<Resources>

<Endpoints>

<Endpoint Name="ServiceEndpoint" Protocol="http" Port="8080"/>

You have now completed the parts related to having your service replicas listen for HTTP client requests. In the next task, you will add the code to process the requests to keep track of the voting items and their counts.

## Task 2 – Adding Reliable Dictionaries and Transactions

1. Just below the CreateServiceReplicaListeners method, insert the following method which processes each HTTP client request.

private async Task ProcessRequestAsync(HttpListenerContext context, CancellationToken ct)

{

String output = null;

var voteDictionary = await this.StateManager.GetOrAddAsync<IReliableDictionary

<string, int>>("voteDictionary");

try

{

// Grab the vote item string from a "Vote=" query string parameter

HttpListenerRequest request = context.Request;

String voteItem = request.QueryString["Vote"];

if (voteItem != null)

{

// The code below prepares the HTML response. It gets all the current

// vote items (and counts) and separates each with a break (<br>)

using (var tx = this.StateManager.CreateTransaction())

{

// Try to read a value from the dictionary whose key is "Counter-1".

var result = await voteDictionary.TryGetValueAsync(tx, voteItem);

// Log whether the value existed or not.

ServiceEventSource.Current.ServiceMessage(this.Context,

"Current Counter Value: {0}",

result.HasValue ? result.Value.ToString()

: "Value does not exist.");

// If the "Counter-1" key doesn't exist, set its value to 0

// else add 1 to its current value.

await voteDictionary.AddOrUpdateAsync(tx, voteItem, 0,

(k, v) => ++v);

// Next block of code will enumerate votes from the reliable

// dictionary for presentation.

// First, obtain enumerator for (voteDictionary) reliable dictionary

var collectionEnumerator = await

voteDictionary.CreateEnumerableAsync(tx);

// Secondly, construct presetation dictionary into which you will

// project vote results

var presentationDictionary = new Dictionary<string, int>();

// Manually iterate reliable dictionary populating data into

// PresentationDictionary

using (IAsyncEnumerator<KeyValuePair<string, int>> enumerator =

collectionEnumerator.GetAsyncEnumerator())

{

while (await enumerator.MoveNextAsync(CancellationToken.None))

{

presentationDictionary.Add(enumerator.Current.Key,

enumerator.Current.Value);

}

}

// Query presentationDictionary projecting desired data shape

var q = from kvp in presentationDictionary

//orderby kvp.Key // Intentionally commented out

select $"Item={kvp.Key}, Votes={kvp.Value + 1}";

output = String.Join("<br>", q);

// Committing transaction serializes changes and writes them to this

// partition's secondary replicas. If an exception is thrown

// before calling CommitAsync, the transaction aborts, all changes

// are discarded, and nothing is sent to this partition's secondary

// replicas.

await tx.CommitAsync();

}

}

}

catch (Exception ex) { output = ex.ToString(); }

// Write response to client:

using (var response = context.Response)

{

if (output != null)

{

Byte[] outBytes = Encoding.UTF8.GetBytes(output);

response.OutputStream.Write(outBytes, 0, outBytes.Length);

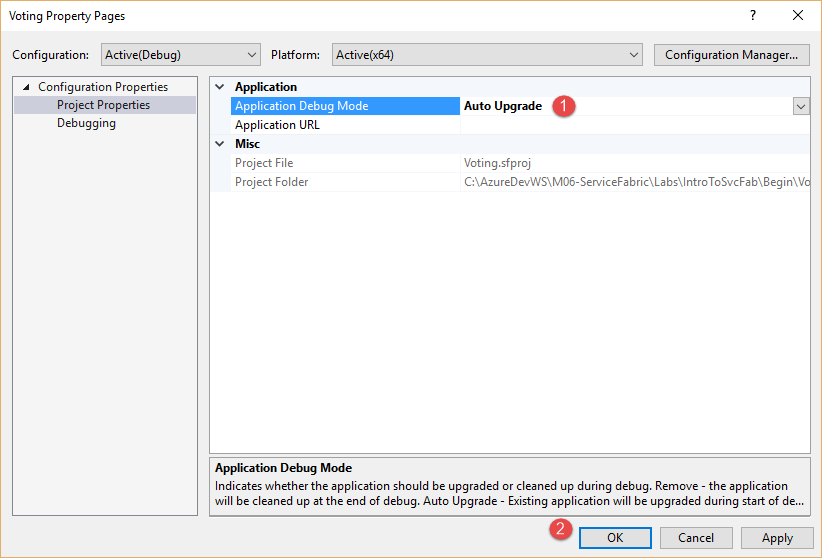
}

}

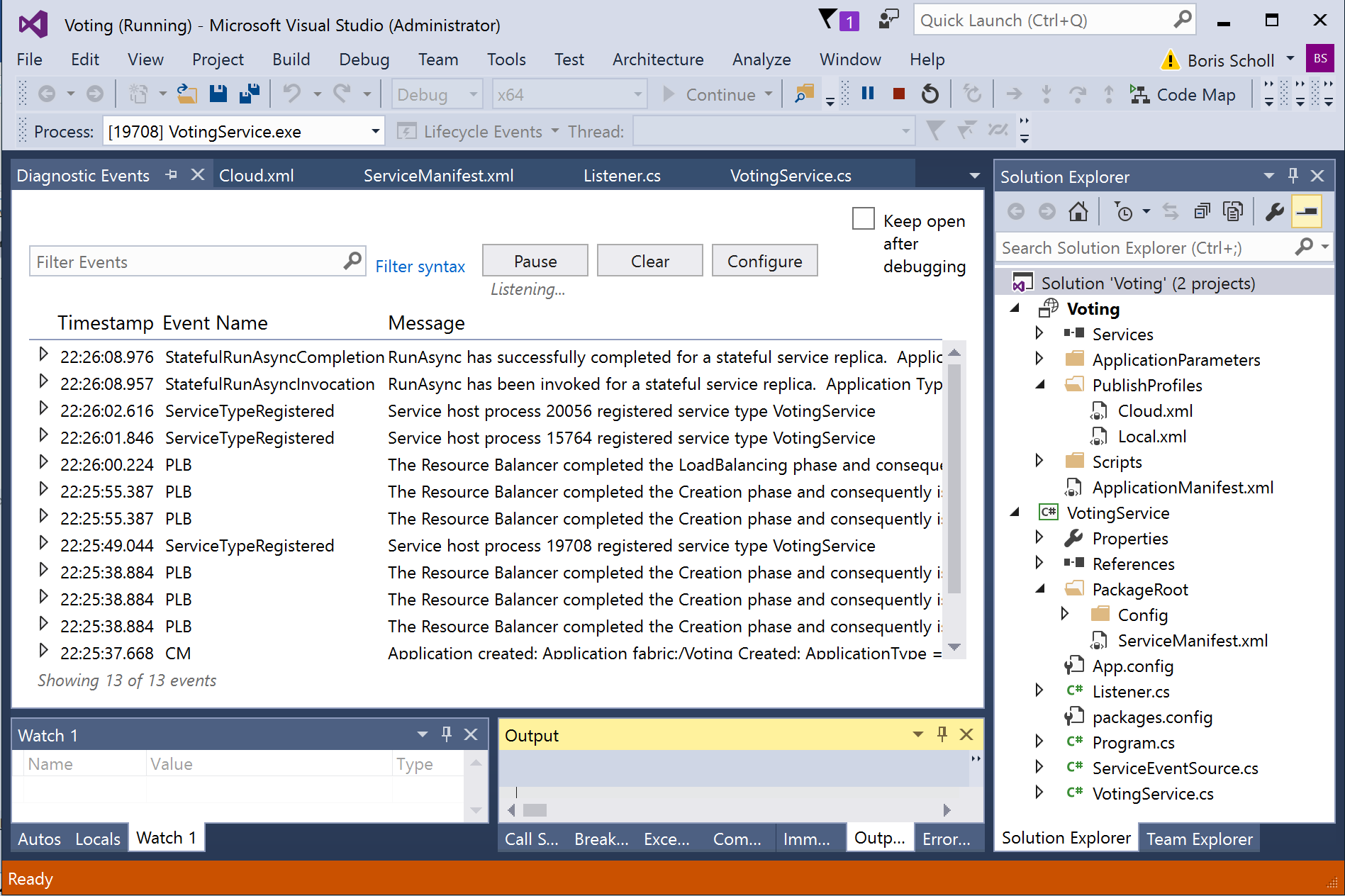
}

Question: Couldn’t you have put the code above in to the RunAsync method which is created by default in our Service? Yes you could, but instead of having code that loops continuously, the ProcessRequestAsync method executes only when the service is called.

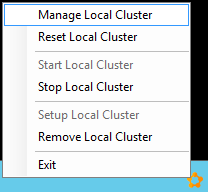
1. Comment out the **RunAsync** method in its entirety as it is not needed in this lab.
2. IMPORTANT STEP!! Right click on the **Voting** project and select **Properties**. Set the **Application Debug Mode** to **Auto Upgrade** and select **Ok**. If you do not do this, then after you close our debug session, the application will be automatically removed from your local cluster (which will defeat the purpose of the next steps of your lab).



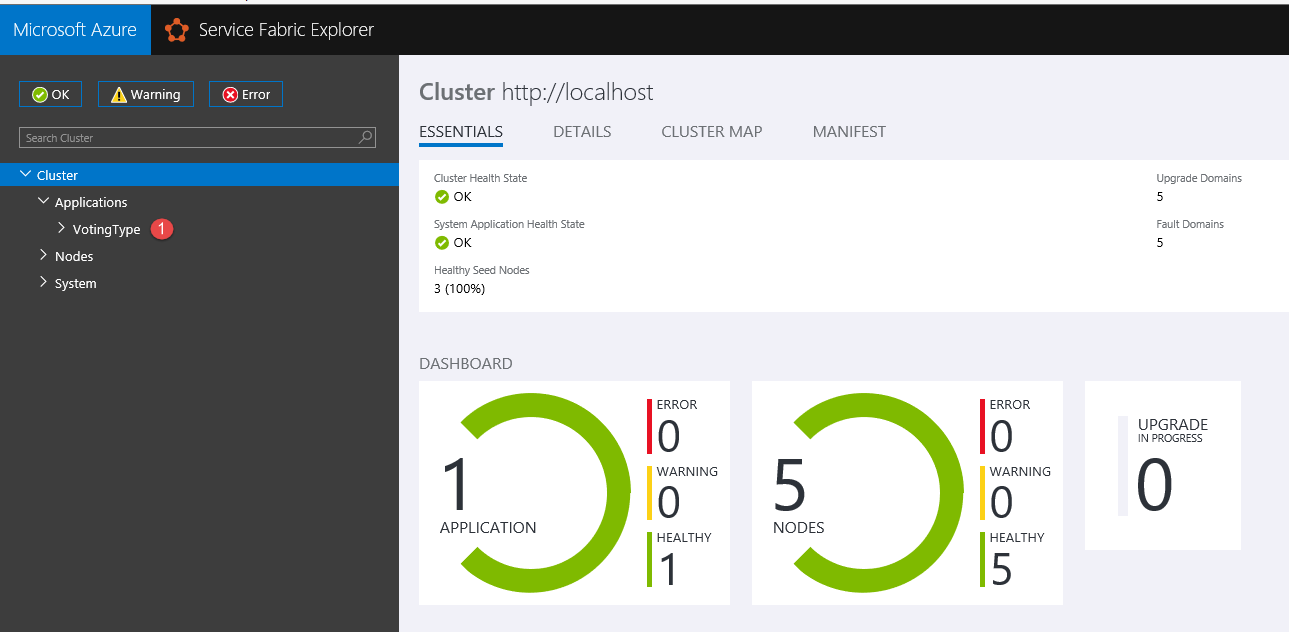
1. Select **F5** to deploy the application to the local development cluster. NOTE: If you previously stopped your local fabric cluster, make sure it is running before you select F5.
2. The application is running when you see events in Visual Studio’s *Diagnostics Events* view. (If you don’t see the window, open it by using **View | Other Windows | Diagnostics Event**)



1. Right-click the **Local Cluster Manager** system tray app and choose **Manage Local Cluster** to launch Service Fabric Explorer.

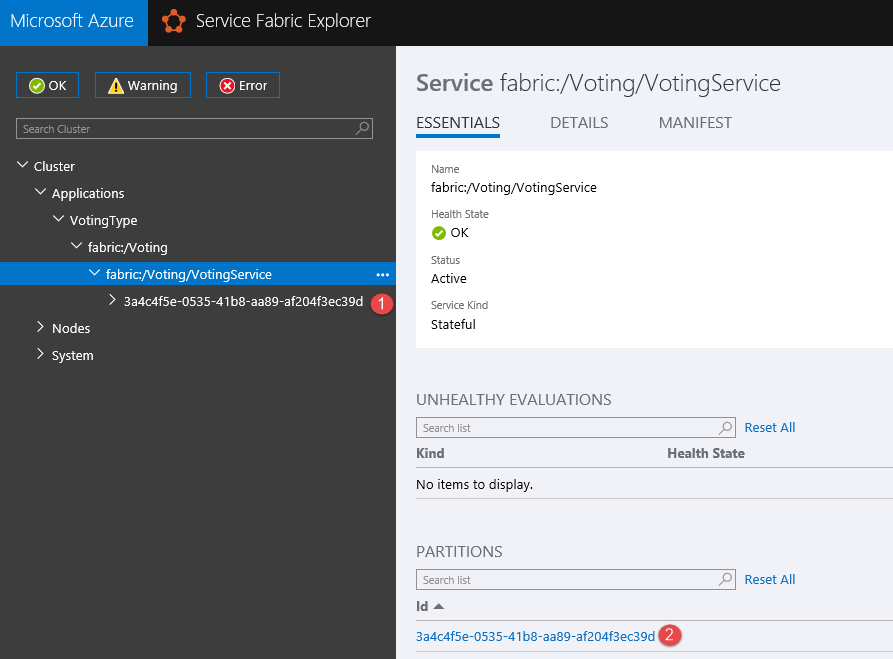


1. The Service Fabric Explorer provides an overview of what’s happening in the cluster. It shows you your deployed application and the number of nodes that are running (virtually) on your own machine.

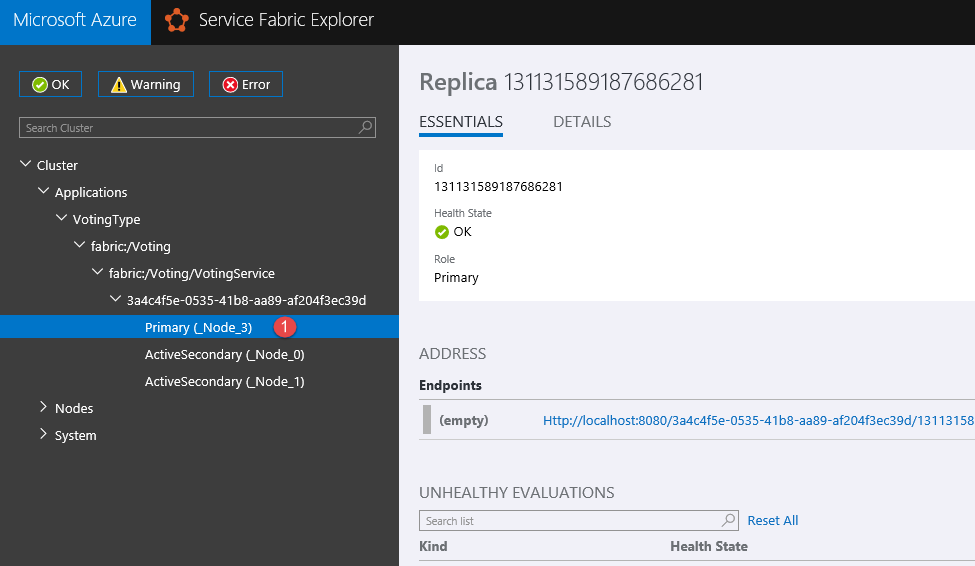


## Task 3 – Viewing the Replica Endpoints

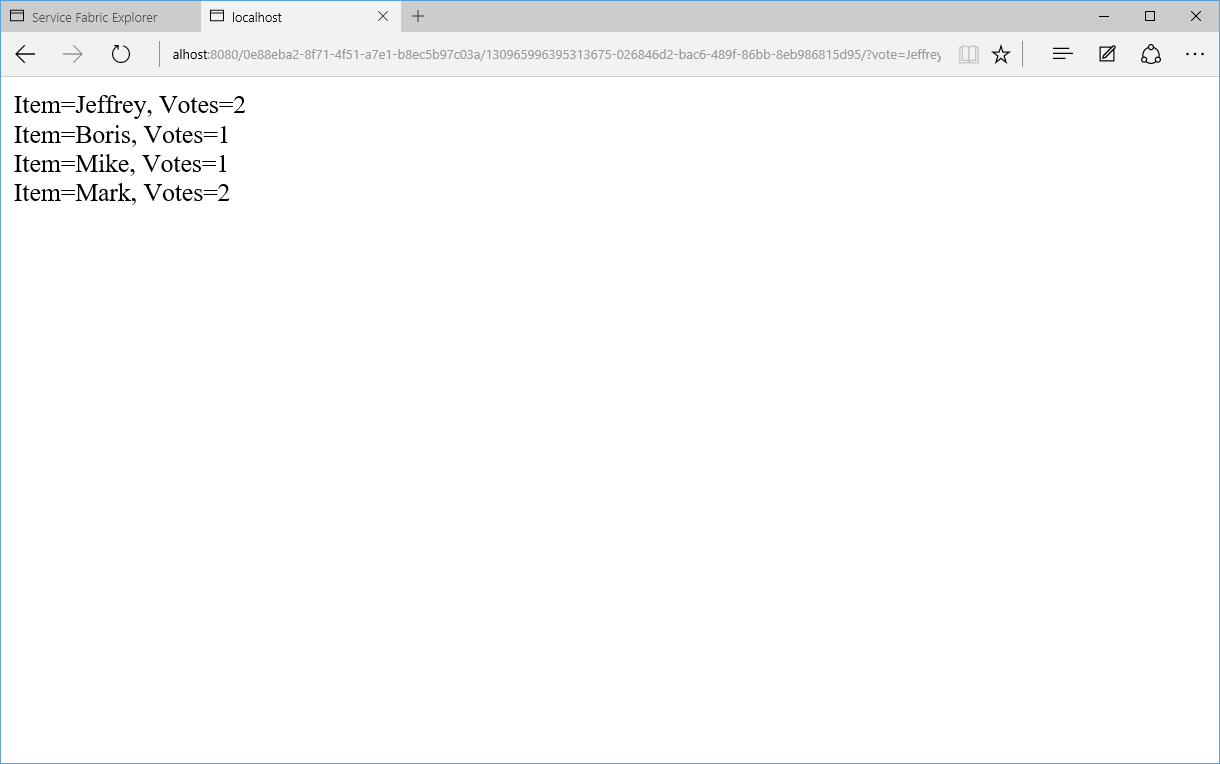
1. Expand the **Applications** section on the left-hand side and click on the **VotingType** entry.
2. Click on the **fabric:/Voting** entry on the right. This uri is the **name** for the running instance of the application. You refer to this as a *named application*.
3. Expand the **fabric:/Voting** entry (on the left) to see the named service running within it (*fabric:/Voting/VotingService*). You see only 1 entry here because you’re running only 1 instance of your service type within the named application.
4. Expand the **fabric:/Voting/VotingService** named service entry (click on the link) to see the partition Id running within it. Partition Ids are GUIDs. In the Figure below the partition Id is [**3a4c4f5e-0535-41b8-aa89-af204f3ec39d**](http://localhost:19080/Explorer/index.html#/apptype/VotingType/app/Voting/service/Voting%252FVotingService/partition/3a4c4f5e-0535-41b8-aa89-af204f3ec39d)**.**



1. Expand the partition Id entry to see the nodes on which each replica resides.
2. Click the entry for the node that holds the partition’s primary replica as shown below.



1. In the Address section in the bottom-right hand corner, note the URL under Endpoints. This is the URL to the endpoint of the service running in the local cluster. It’s important to know that the endpoint consists of the following components:
   1. **Http:** the scheme used to talk to the replica (as defined by our listener)
   2. **localhost**: the IP address of the node
   3. **8080**: the port the listener is listening to (as specified in the ServiceManifest.xml file)
   4. **Guid**: the partition Id (this helps diagnose problems; the URL itself indicates the partition you’re trying to access)
   5. **64-bit integer**: the replica Id (this helps diagnose problems; the URL itself indicates the replica you’re trying to access)
   6. **Guid**: A random guid used to ensure the URL endpoint is unique
2. Click on the URL under the endpoint. Doing so opens a browser with the URL and invokes the underlying service. If you have a breakpoint set in the service, you will see Visual Studio stopped on that breakpoint.
3. In the browser append a query string parameter “?vote=***SomeString***” (replace ***SomeString*** with a value of your choice). Sending this request is how you vote for something.
4. As an example, vote for “Jeffrey” a few times (by refreshing your browser tab) and feel free to change the string you’re voting for as well. Here’s a figure showing a few votes for Jeffrey, Boris, Mike, and Mark:

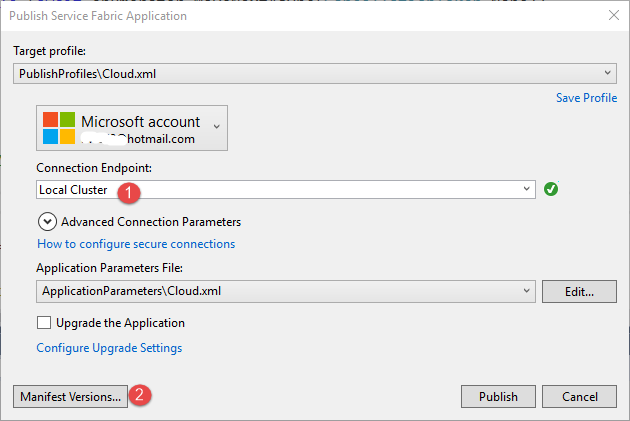


1. Go back to Visual Studio and stop the debugging session. Note that the application will still continue running in the cluster, since you have already deployed it there AND because you set the Voting projects property to Auto Upgrade.

## Task 4 – Upgrading the named application

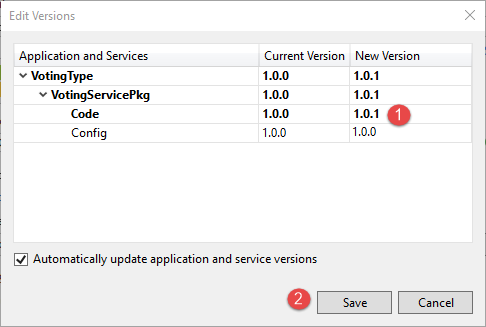
In this task, you will learn how to upgrade the Voting application without any downtime and without any loss of data. The ease of upgrading an application is one of the major features of Azure Service Fabric

1. Inside the **VotingService.cs** file, locate the commented out statement **//orderby kvp.Key** in the **ProcessRequestAsync** method and uncomment it.
2. To upgrade the application, right click on the **Voting** project and select **Publish**. In the Publish dialog box, select Local Cluster and then select the **Manifest Versions** button.

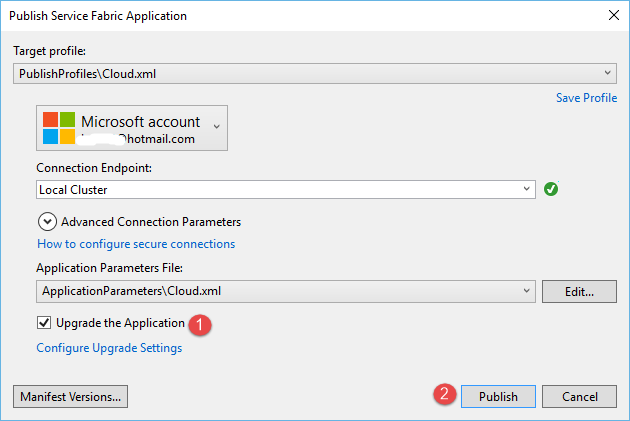


1. The Edit Versions dialog will open, providing you with a UI in order to change either the version number of your application type’s package, service type’s package or code package. You must change at least one of the version number for the application to understand that it is being upgraded.

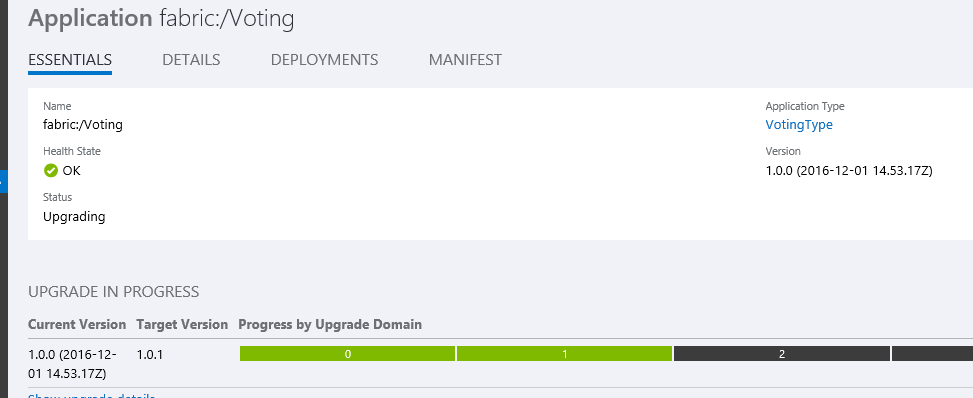
Change the **Code’s** new version to **1.0.1** (this changes VotingType and VotingServicePkg version numbers automatically). Select the **Save** button.



1. Make sure the **Upgrade the Application** checkbox is checked and then select the **Publish** button. If the publish operation fails, right-click on the Voting application, select the Publish menu and the select the Publish button again.



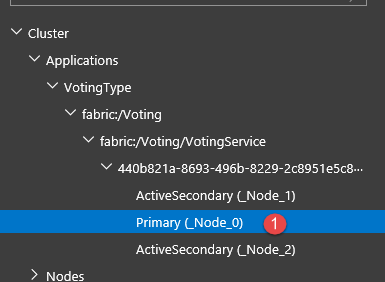
1. Go back to the Service Fabric Explorer, select the **fabric:/Voting** named application. Here, you’ll see the upgrade rolling through the cluster’s Upgrade Domains (UDs). Note: it usually takes a few minutes until all UDs have completed their upgrade.



1. After all upgrade domains have completed their upgrade, you can see the application package’s new version in the **Essentials** tab
2. Click on the entry for the node that holds the primary replica and copy the http endpoint to the clipboard. Note: This endpoint will be different from the earlier one because replica IDs change during an upgrade.
3. Paste the new endpoint into a browser, remove the backslashes and append a query string parameter “?vote=***SomeString***” with a value of your choice. After you execute this, the response should show all the previous vote data since the data was preserved. In addition, you should now see that the vote strings in sorted order since you updated your code.

## Task 5 – Failover and PowerShell

1. Open Service Fabric Explorer and check which node number holds your named service’s primary replica by expanding its partition entry. In this example, it is \_Node\_0.



1. Open PowerShell ISE and connect to the local Service Fabric cluster by entering the command below into the PowerShell command window:

Connect-ServiceFabricCluster

1. In PowerShell ISE, remove the primary replica (the Service Fabric will automatically elect a secondary replica to become the new primary replica).

Remove-ServiceFabricReplica -NodeName ‘<nameofnode>’ -ReplicaOrInstanceId ‘<yourreplicaorinstanceId>’ -PartitionId ‘<yourpartitionId>’

1. Once the primary replica has been changed to a new node, click on the replicas endpoint link in the Service Fabric Explorer. From here, you can grab the new Url (which includes the partition Id and replica Id) from the primary replica node and replace that part of the query string in the browser.

The vote results should still show the same values.