# Working with Service Fabric Services – Part I

## Introduction

Estimated time to complete this lab: 60-90 minutes

After completing this lab, you will be able to:

* Set up and manage Service Fabric clusters on your development machine
* Understand the concepts of Service Fabric applications, services, stateless services, application lifecycle management, diagnostics and health
* Use Visual Studio and Service Fabric Explorer to efficiently develop Service Fabric applications

## Prerequisites

Before working on this lab, you must have: [Visual Studio 2015](https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx) with Update 3 (14.0.25420.1) and [Service Fabric SDK](http://www.microsoft.com/web/handlers/webpi.ashx?command=getinstallerredirect&appid=MicrosoftAzure-ServiceFabric) v2.3.301.9590. If you are using a different version of Visual Studio or the Service Fabric SDK there may be differences between what is documented in this lab and what you are presented with. See [Prepare you development environment](https://azure.microsoft.com/en-us/documentation/articles/service-fabric-get-started/) for information on how to install a development environment on your machine.

## Overview of the lab

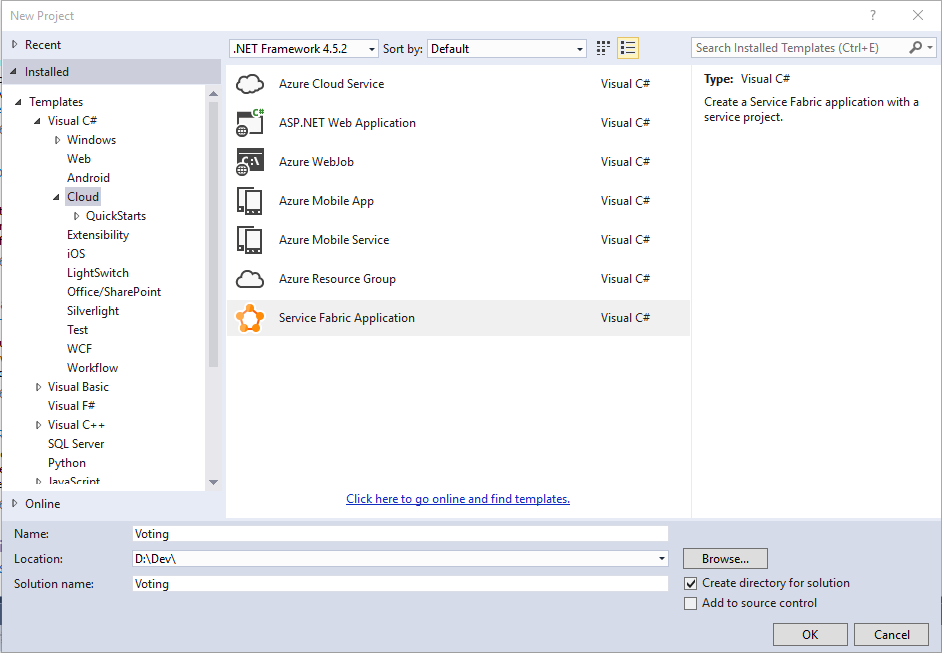
Goal of this lab is to make you familiar with an end to end development flow for Service Fabric applications. You will practice creating a new Service Fabric application on your development machine, working with stateless services, deploying, updating and monitoring an application deployment. Throughout the exercise, you will get accustomed with Visual Studio’s Service Fabric tooling, Service Fabric Explorer and learn how to effectively use both.

## Scenario

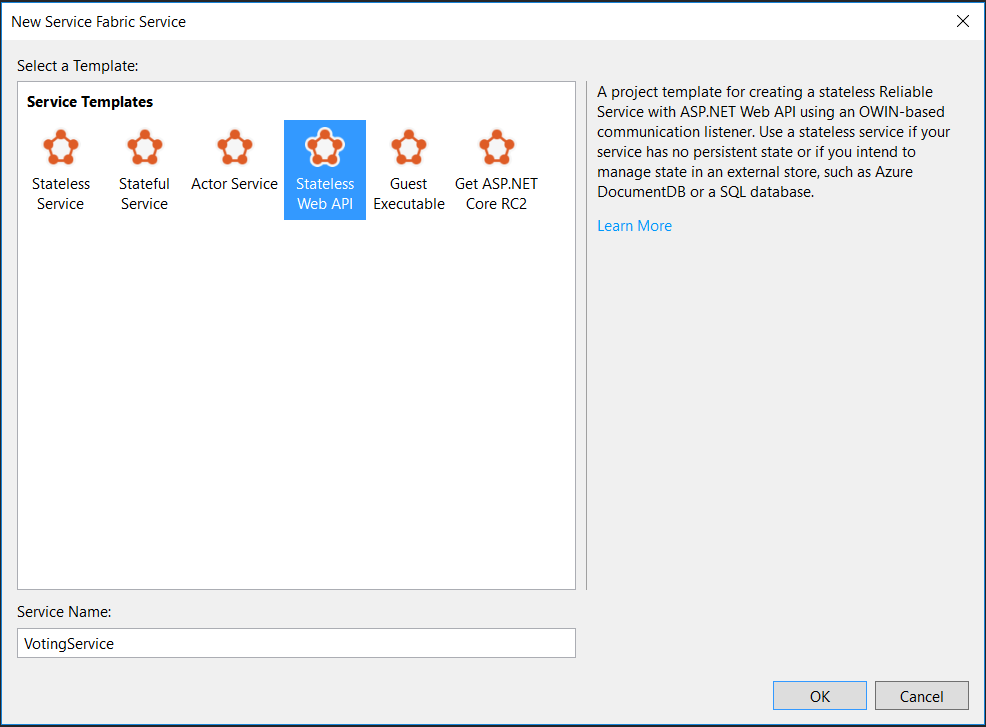
In this scenario, you will build a generic voting service using Service Fabric reliable services. The service listens to an endpoint accessible from a Web browser. You’ll enter vote item strings (such as favorite sodas or cars) using a single page application (SPA). Each time the same vote item is voted on, a counter is incremented; this 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. If you can’t wait to see the user interface, skip ahead to step 18, but make sure you come back here to start the lab.

## Create a stateless service

1. Open Visual Studio with **elevated privileges** by pressing the **Start** () button on the keyboard and typing “*Visual Studio*”, then run Visual Studio by **right clicking** and choosing **Run as Administrator**. Visual Studio must be run using elevated privileges because it must interact with the Service Fabric runtime.
2. Select **File** | **New** | **Project** …
3. Go to **Cloud** and choose **Service Fabric Application**



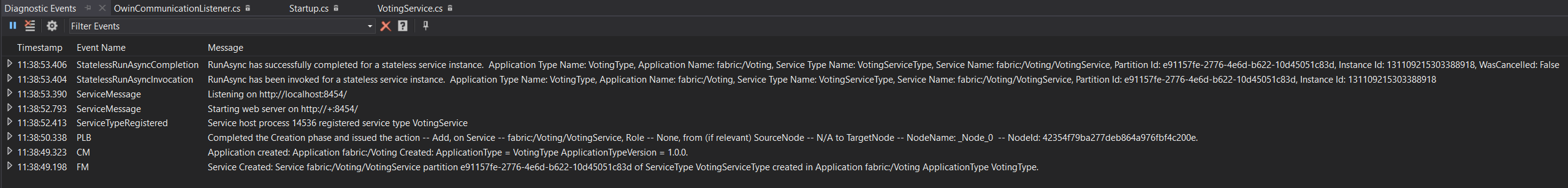
1. Enter “*Voting*” for the Name and Solution name fields and then click **OK**
2. In **Service Templates** choose **Stateless Web API** and enter “VotingService” for the service name. Click **OK**.



1. Visual Studio will create a solution containing two projects, *Voting* and *VotingService*. The *Voting* project is the Service Fabric project containing:
   1. A reference to the *VotingService* project
   2. *ApplicationPackageRoot* folder containing the *ApplicationManifest.xml* file describing your Service Fabric application
   3. *ApplicationParameters* folder containing deployment parameters for local (*Local.1Node.xm and Local.5Node.xmll)* and cloud (*Cloud.xml)* deployments. In this lab we’ll only use the *Local.5Node.xml* parameters
   4. *PublishProfiles* containing deployment profiles for local (*Local.1Node.xml and Local.5Node.xml)* and cloud (*Cloud.xml)* deployments. In this lab we’ll only use the *Local.5Node.xml* profile. The Cloud profile is used to publish to Azure
   5. *Scripts* containing the scripts used for deploying the application to the cluster
   6. *Packages.config* used to indicate the packages associated with this application

The *VotingService* project contains the stateless service implementation and contains:

1. *Controllers* foldercontaining the controllers for this project. An initial controller named *ValuesController.cs* has been generated
2. *PackageRoot* folder containing the service configuration and *ServiceManifest.xml*
3. *OwinCommunicationsListener.cs* contains the ICommunicationListener implementation based on the Owin HTTP hosting framework
4. *Program.cs* which is the host executable of the stateless service
5. *ServiceEventSource.cs* contains the class used for diagnostic events
6. *Startup.cs* containing the application server startup configuration
7. *VotingService.cs* contains the classed implementing the stateless voting service
8. At this point you have a functioning service that can be hosted within Service Fabric. Press **F5** to see the service running. Within Visual Studio, the Diagnostic Events panel will be visible and display messages coming from within the application.

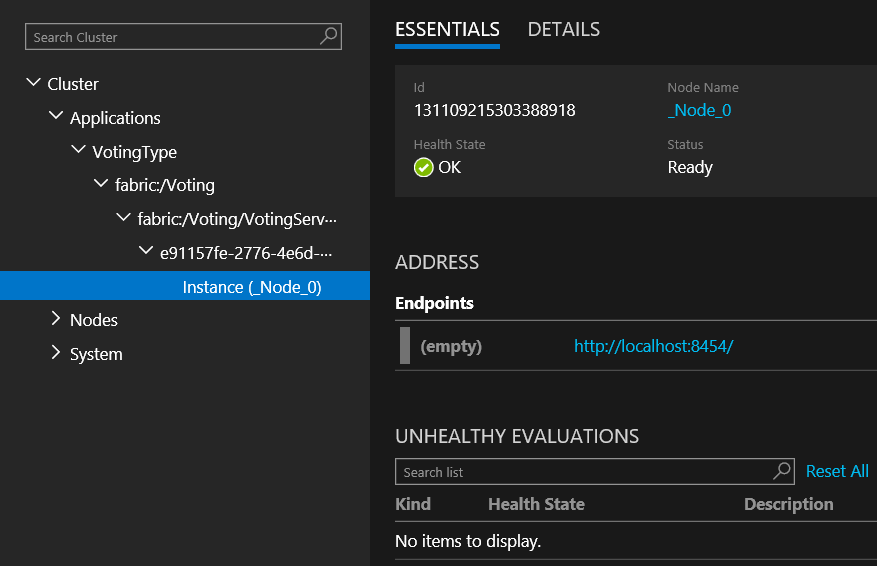


**Note:** In the 5.3 version of the SDK too many Service Fabric events are being generated and they hide the events that are part of this lab. To disable the extra events, click the gear icon in the diagnostic event window and remove the “Microsoft-ServiceFabric:5:0x4000000000000000” line. Then click **Apply**.

1. The deployed application can also be seen in Service Fabric Explorer. On the Service Fabric icon  in the notification area, right click on the icon and choose **Manage Local Cluster**. The Service Fabric Explorer (SFX) will start in a browser.

**Note:** If the icon isn’t present, start the Service Fabric Local Cluster Manager by pressing the **Start** () button on the keyboard and typing “*Service Fabric Local Cluster Manager*”, then run the application by pressing **Enter.** This will start the Service Fabric Local Cluster Manager and the Service Fabric icon  will appear in the notification area. If you haven’t already created a cluster, select Start Local Cluster and close 5 node.

1. On the left side of SFX fully expand the applications tree. You can see that the application *fabric:/Voting* has been deployed and contains a single service named *fabric:/Voting/VotingService*. The service has a single instance that is deployed on a node (\_Node\_0 in this case).
2. Select **Instance (\_Node\_**X**)** where X represent the number displayed. On the right side of SFX, you’ll see more details about the service including the endpoint where it curently resides (<http://localhost:8454/> in this example, your port is likelly to be different). Paste the endpoint there and append “*api/values*” into a browser address bar. This will return a JSON document containing [“value1”, “value2”], which is the standard behavior of this Visual Studio template.



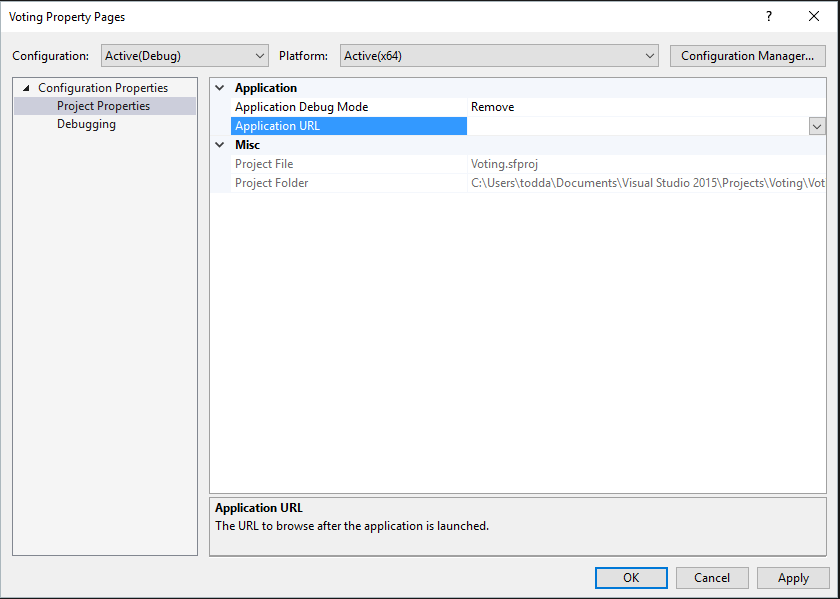
1. Stop the application by exiting the debugger. This will remove the application from Service Fabric.

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

## Add Voting Endpoints

The next step is to add some endpoints that can be used to vote and view the votes. We’ve written a single page application for this purpose.

1. Right click on the *Voting* project and select *Properties*. Remove the Application URL property value and click OK. This will prevent a browser popping up each time we debug. For reference, the *Application Debug Mode* setting is set to automatically remove the Service Fabric application when debugging is stopped.



1. In the *VotingService* project, open the *ServiceManifest.xml* file which is contained in the *PackageRoot* folder. Remove *Port=”XXXX”* from the *Endpoint* element, where XXXX is the port number assigned. In this example the port number is 8454. This allows Service Fabric to assign a random port for your service.

**Change**

<Endpoint Protocol="http" Name="ServiceEndpoint" Type="Input" Port="8454" />

**To**

<Endpoint Protocol="http" Name="ServiceEndpoint" Type="Input" />

We’re allowing Service Fabric to assign ports because later in the lab, we’ll run multiple instances of the service on your development box. Without this change, only the first instance will start successfully. Even in production, it’s better to use dynamically assigned ports to avoid port conflicts with other services that may be running on the node except for nodes exposed to the Azure load balancer.

1. Rename *ValuesController.cs*  to **VotesController.cs**. If prompted to rename the class, select **Yes**. Ensure the class name *ValuesController* has been changed to *VotesController*.
2. Add a new class to the *VotingService* project called “*HtmlMediaFormatter.cs*” and paste the following contents **within the namespace brackets**. Remove extra using directives at the top of the file if necessary.

using System;

using System.IO;

using System.Text;

using System.Net.Http;

using System.Net.Http.Headers;

using System.Net.Http.Formatting;

// This class is needed to be able to return static files from the WebAPI 2 self-host infrastructure.

// It will return the index.html contents to the browser.

public class HtmlMediaFormatter : BufferedMediaTypeFormatter

{

public HtmlMediaFormatter()

{

SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));

SupportedEncodings.Add(new UTF8Encoding(encoderShouldEmitUTF8Identifier: false));

}

public override bool CanReadType(Type type)

{

return false;

}

public override bool CanWriteType(Type type)

{

return (typeof(string) == type) ? true : false;

}

public override void WriteToStream(Type type, object value, Stream writeStream, HttpContent content)

{

Encoding effectiveEncoding = SelectCharacterEncoding(content.Headers);

using (var writer = new StreamWriter(writeStream, effectiveEncoding))

{

writer.Write(value);

}

}

}

1. Open *Startup.*cs and replace the contents of the *ConfigureApp* method with the following code

// Configure Web API for self-host.

HttpConfiguration config = new HttpConfiguration();

config.MapHttpAttributeRoutes(); // NEW

config.Formatters.Add(new HtmlMediaFormatter()); // NEW

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

appBuilder.UseWebApi(config);

1. Add a new HTML file to the *VotingService* project named “*index.html”*. This is the [Angular](https://angular.io/) Single Page Application (SPA) HTML file that displays the user experience and communicates with the service’s REST API. Explaining more about using Angular is beyond the scope of this lab. Paste the following contents:

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml" ng-app="VotingApp" xmlns:ng="http://angularjs.org">

<head>

<meta charset="utf-8" />

<meta content="IE=edge, chrome=1" http-equiv="X-UA-Compatible" />

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1" />

<!-- Stylesheets -->

<link href="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css" rel="stylesheet">

<!-- Application title and icons -->

<title>Voting Service Lab Sample</title>

<!-- IE Fix for HTML5 Tags -->

<!--[if lt IE 9]>

<script src="http://html5shiv.googlecode.com/svn/trunk/html5.js"></script>

<![endif]-->

</head>

<body ng-controller="VotingAppController">

<div class="container-fluid">

<h1>Votes</h1>

<div>

Add new voting item <input id="txtAdd" type="text" class="form-control" placeholder="Enter new voting term" ng-model="item" />

<button id="btnAdd" class="btn btn-primary" ng-click="add(item)">Add</button>

</div>

<br />

<table class="table table-striped table-condensed table-hover">

<thead>

<tr>

<td>Voting Item</td>

<td>Count</td>

<td><button id="btnRefresh" class="btn btn-primary" ng-click="refresh()">Refresh</button></td>

</tr>

</thead>

<tr ng-repeat="vote **in** votes">

<td><button class="btn btn-primary" ng-click="add(vote.Key)">**{{**vote.Key**}}**</button></td>

<td>**{{**vote.Value**}}**</td>

<td><button class="btn btn-default" ng-click="remove(vote.Key)">Remove</button></td>

</tr>

</table>

</div>

<!-- 3rd party libraries -->

<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.0/angular.js"></script>

<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.0/angular-route.js"></script>

<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.0/angular-cookies.js"></script>

<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.0/angular-animate.min.js"></script>

<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.0/angular-sanitize.min.js"></script>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

<script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/0.12.1/ui-bootstrap-tpls.js"></script>

<!-- Load application main script -->

<script>

var app = angular.module('VotingApp', ['ui.bootstrap']);

app.run(function () { });

app.controller('VotingAppController', ['$rootScope', '$scope', '$http', '$timeout', function ($rootScope, $scope, $http, $timeout) {

$scope.refresh = function() {

$http.get('../api/votes')

.success(function (data, status) {

$scope.votes = data;

})

.error(function (data, status) {

$scope.votes = undefined;

});

};

$scope.remove = function (item) {

$http.delete('../api/' + item)

.success(function (data, status) {

$scope.refresh();

})

};

$scope.add = function (item) {

var fd = new FormData();

fd.append('item', item);

$http.post('../api/' + item, fd, {

transformRequest: angular.identity,

headers: { 'Content-Type' : undefined }

})

.success(function(data, status)

{

$scope.refresh();

$scope.item = undefined;

})

};

}]);

</script>

</body>

</html>

1. Right click on **index.html** and select **Properties (Alt+Enter)**. In the properties windows change the property **Copy to Output Directory** to **Copy Always**.
2. Open *VotesController.cs* and paste the following implementation **within the namespace brackets**. Remove extra using directives at the top of the file. Note that the hardcoded path is version specific. When the version is changed later in the lab, the file will no longer be at this location.

using System;

using System.Collections.Generic;

using System.IO;

using System.Net;

using System.Net.Http;

using System.Threading;

using System.Net.Http.Headers;

using System.Web.Http;

public class VotesController : ApiController

{

// Used for health checks.

public static long \_requestCount = 0L;

// Holds the votes and counts. NOTE: THIS IS NOT THREAD SAFE FOR THE PURPOSES OF THE LAB ONLY.

static Dictionary<string, int> \_counts = new Dictionary<string, int>();

// GET api/votes

[HttpGet]

[Route("api/votes")]

public HttpResponseMessage Get()

{

Interlocked.Increment(ref \_requestCount);

List<KeyValuePair<string, int>> votes = new List<KeyValuePair<string, int>>(\_counts.Count);

foreach(KeyValuePair<string, int> kvp in \_counts)

{

votes.Add(kvp);

}

var response = Request.CreateResponse(HttpStatusCode.OK, votes);

response.Headers.CacheControl = new CacheControlHeaderValue() { NoCache = true, MustRevalidate = true };

return response;

}

[HttpPost]

[Route("api/{key}")]

public HttpResponseMessage Post(string key)

{

Interlocked.Increment(ref \_requestCount);

if (false == \_counts.ContainsKey(key))

{

\_counts.Add(key, 1);

}

else

{

\_counts[key] = \_counts[key] + 1;

}

return Request.CreateResponse(HttpStatusCode.NoContent);

}

[HttpDelete]

[Route("api/{key}")]

public HttpResponseMessage Delete(string key)

{

Interlocked.Increment(ref \_requestCount);

if (true == \_counts.ContainsKey(key))

{

if (\_counts.Remove(key))

return Request.CreateResponse(HttpStatusCode.OK);

}

return Request.CreateResponse(HttpStatusCode.NotFound);

}

[HttpGet]

[Route("api/{file}")]

public HttpResponseMessage GetFile(string file)

{

string response = null;

string responseType = "text/html";

Interlocked.Increment(ref \_requestCount);

// Validate file name.

if ("index.html" == file)

{

// This hardcoded path is only for the lab. Later in the lab when the version is changed, this

// hardcoded path must be changed to use the UX. In part 2 of the lab, this will be calculated

// using the connected service path.

string path = string.Format(@"..\VotingServicePkg.Code.1.0.0\{0}", file);

response = File.ReadAllText(path);

}

if (null != response)

return Request.CreateResponse(HttpStatusCode.OK, response, responseType);

else

return Request.CreateErrorResponse(HttpStatusCode.NotFound, "File");

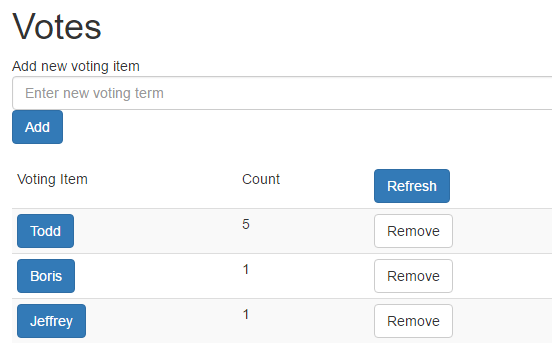
}

}

1. Press **F5** to enter debug mode. After the solution has been deployed locally there are two ways to determine the endpoint to browse to
   1. In the Diagnostic Events window which should be open within Visual Studio, there will be an event named “ServiceMessage” with a message body containing the base URL the service is listening on, e.g. “Listening on <http://localhost:34001>”. If the Diagnostic Events windows is not open, it can be opened in Visual Studio by selecting **View** then **Other Windows** then **Diagnostic Events**.
   2. Open Service Fabric Explorer (SFX), navigate to the instance and view the Endpoints properties as described in step 10.

**Note:** In the 5.3 version of the SDK too many Service Fabric events are being generated and they hide the events that are part of this lab. To disable the extra events, click the gear icon in the diagnostic event window and remove the “Microsoft-ServiceFabric:5:0x4000000000000000” line.

When you have determined the correct base URI, browse to <*base URI>/api/index.html.* This will show the single page app just created, except without the data. Try it out. If you want to see that it is calling the services, you can place breakpoints in the *VotesController* class.



1. When done using the application, exit the debugging session by selecting **Debug** then **Stop Debugging** (Shift+F5). This will uninstall the application from Service Fabric and if viewing in Service Fabric Explorer (SFX) you will see that it is no longer deployed.

## Instrumenting the Code

Any service code must be instrumented to allow monitoring of the service and forensic debugging of the application. It’s not likely that you’ll be attaching a debugger to a running instance in production

1. Open the *ServiceEventSource.cs* file. This file contains the structured events that can viewed in the Diagnostic Events window and can be captured by Azure diagnostics.
2. Expand the Keywords region, you’ll see the static Keywords class. These keywords are the values that you can later filter on using Diagnostic Events window or other ETW based viewers. Add a new keyword definition

public const EventKeywords HealthReport = (EventKeywords)0x4L;

1. Expand the Events region and add three new structured event definitions at the bottom of the region. Each event must have a unique identifier and name. We’re defining structured events rather than a single event that accepts a string is to allow easier searching and filtering of events after they are logged. There is an intentional gap in the number to allow insertion of addition ServiceRequestxxx events if needed later.

private const int HealthReportEventId = 100;

[Event(HealthReportEventId, Level = EventLevel.LogAlways, Message = "Health report. Source '{0}' property {1} is {2}. Partition: {3}, Instance or Replica: {4} Desc: {5}.", Keywords = Keywords.HealthReport)]

public void HealthReport(string healthSourceId, string name, string state, Guid partition, long instanceOrReplica, string description )

{

WriteEvent(HealthReportEventId, healthSourceId, name, state, partition, instanceOrReplica, description);

}

private const int HealthReportIntervalChangedEventId = 101;

[Event(HealthReportIntervalChangedEventId, Level = EventLevel.Informational, Message = "Health report interval changed to {4} seconds for {0} property {1}. Partition: {2} Instance or Replica: {3}.", Keywords = Keywords.HealthReport)]

public void HealthReportIntervalChanged(string healthSourceId, string name, Guid partition, long instanceOrReplica, int duration)

{

WriteEvent(HealthReportIntervalChangedEventId, healthSourceId, name, partition, instanceOrReplica, duration);

}

1. Add an *activityId* argument to the *ServiceReqestStart and* *ServiceRequestStop* event methods. The *activityId* is a unique identifier used to follow the thread of execution through your code. It will also make matching start and stop events for duration possible. The methods will appear as

private const int ServiceRequestStartEventId = 5;

[Event(ServiceRequestStartEventId, Level = EventLevel.Informational, Message = "Service request '{0}' started. Activity id: {1}", Keywords = Keywords.Requests)]

public void ServiceRequestStart(string requestTypeName, string activityId = "")

{

WriteEvent(ServiceRequestStartEventId, requestTypeName, activityId);

}

private const int ServiceRequestStopEventId = 6;

[Event(ServiceRequestStopEventId, Level = EventLevel.Informational, Message = "Service request '{0}' finished Activity id: {1}", Keywords = Keywords.Requests)]

public void ServiceRequestStop(string requestTypeName, string activityId = "", string exception = "")

{

WriteEvent(ServiceRequestStopEventId, requestTypeName, activityId, exception);

}

1. Open *VotesController.cs* file. At the top of each method add the following code replacing the XXX with the method name. If needed, add *using System;* to the using section.

string activityId = Guid.NewGuid().ToString();

ServiceEventSource.Current.ServiceRequestStart("VotesController.XXX", activityId);

and at the end of the method before the return add the following and ensure you use the same string for the *requestTypeName* parameter.

ServiceEventSource.Current.ServiceRequestStop("VotesController.XXX", activityId);

1. Open *VotingService.cs*. Ensure the following are included in the using statements

using System;

using System.Fabric.Health;

add the following to the constructor or create the constructor if it doesn’t already exist

public VotingService(StatelessServiceContext context)

: base(context)

{

// Create the timer here, so we can do a change operation on it later, avoiding creating/disposing of the

// timer.

\_healthTimer = new Timer(ReportHealthAndLoad, null, Timeout.Infinite, Timeout.Infinite);

}

Then paste the following code at the bottom of the class

private TimeSpan \_interval = TimeSpan.FromSeconds(30);

private long \_lastCount = 0L;

private DateTime \_lastReport = DateTime.UtcNow;

private Timer \_healthTimer = null;

private FabricClient \_client = null;

protected override Task OnOpenAsync(CancellationToken cancellationToken)

{

\_client = new FabricClient();

\_healthTimer.Change(\_interval, \_interval);

return base.OnOpenAsync(cancellationToken);

}

public void ReportHealthAndLoad(object notused)

{

// Calculate the values and then remember current values for the next report.

long total = Controllers.VotesController.\_requestCount;

long diff = total - \_lastCount;

long duration = Math.Max((long)DateTime.UtcNow.Subtract(\_lastReport).TotalSeconds, 1L);

long rps = diff / duration;

\_lastCount = total;

\_lastReport = DateTime.UtcNow;

// Create the health information for this instance of the service and send report to Service Fabric.

HealthInformation hi = new HealthInformation("VotingServiceHealth", "Heartbeat", HealthState.Ok)

{

TimeToLive = \_interval.Add(\_interval),

Description = $"{diff} requests since last report. RPS: {rps} Total requests: {total}.",

RemoveWhenExpired = false,

SequenceNumber = HealthInformation.AutoSequenceNumber

};

var sshr = new StatelessServiceInstanceHealthReport(Context.PartitionId, Context.InstanceId, hi);

\_client.HealthManager.ReportHealth(sshr);

// Report the load

Partition.ReportLoad(new[] { new LoadMetric("RPS", (int)rps) });

}

1. Open ServiceManifest.xml in the PackageRoot directory of VotingService. Replace the <StatelessServiceType ServiceTypeName="VotingServiceType"/> with the XML below. This adds a load metric named RPS that has a weight of zero, meaning that is won’t contribute to service balancing.

<StatelessServiceType ServiceTypeName="VotingServiceType">

<LoadMetrics>

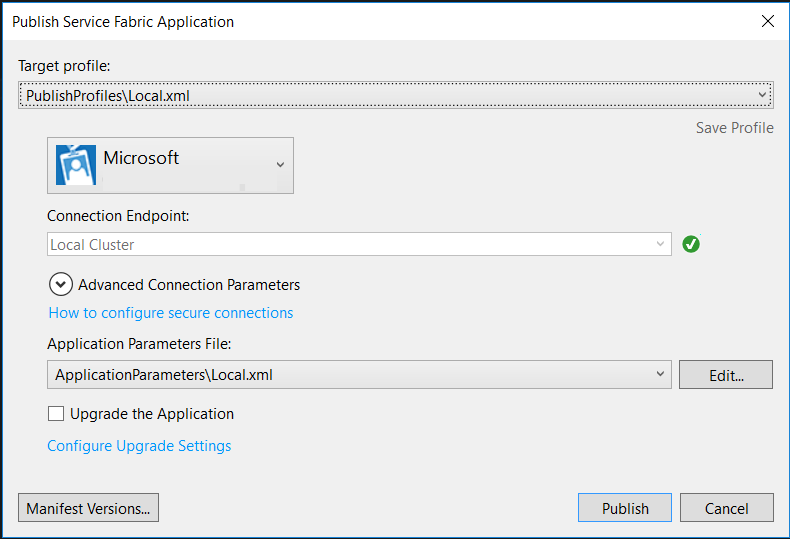
<LoadMetric Weight="Zero" Name="RPS" />

</LoadMetrics>

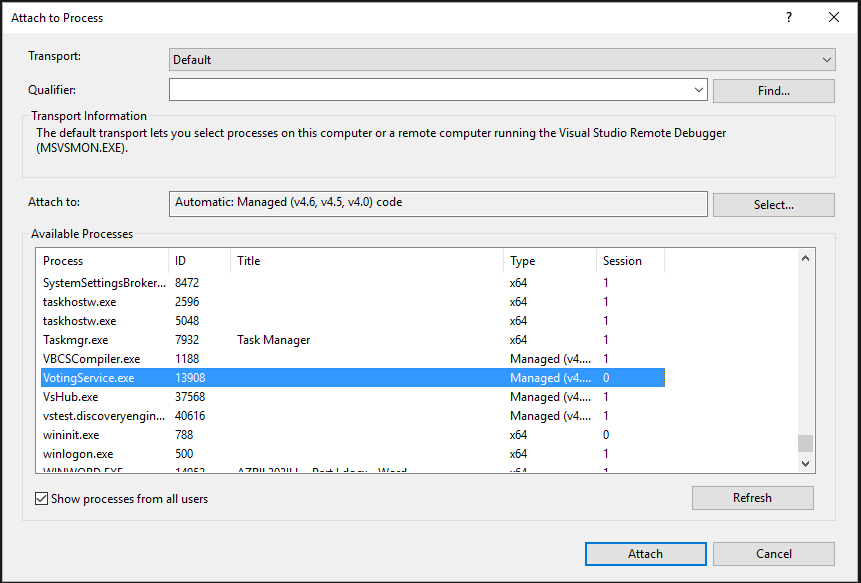
</StatelessServiceType>

</ServiceTypes>

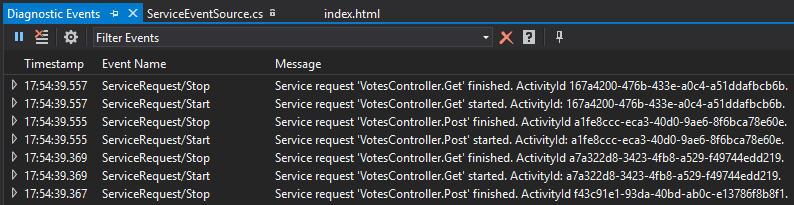
1. Right click on the *Voting* project and select **Publish…**, the Publish Service Fabric Application dialog will appear. Select **PublishProfiles\Local.5Node.xml** for the Target profile, which will select the Local Cluster for the Connection Endpoint and Local.xml for Application Parameters File. Ensure the values are correct and click **Publish** which will start deploying the application to your local cluster.



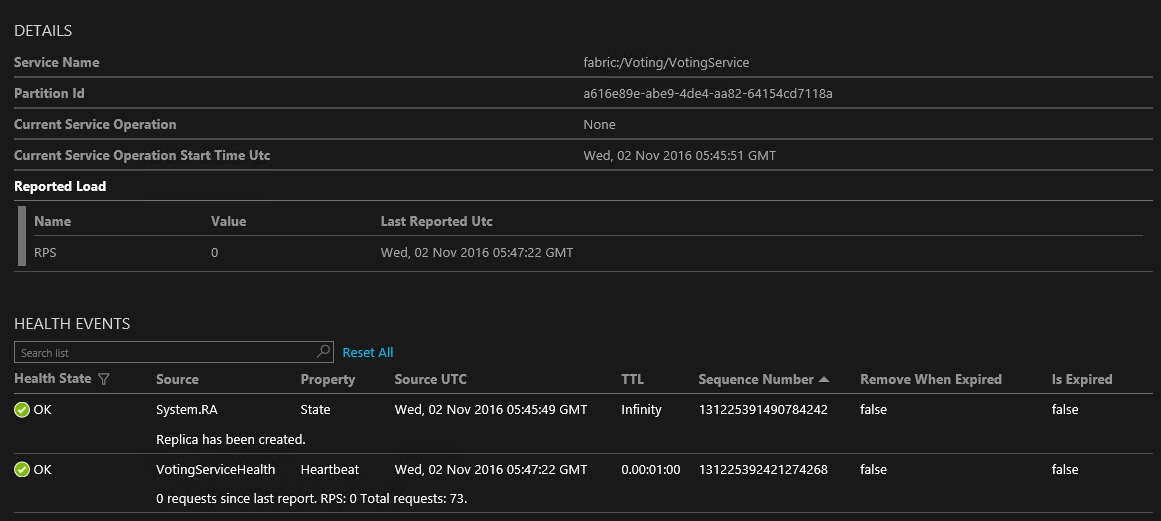
1. Attach a debugger by selecting **Debug** then **Attach to Process...** which will display the Attach to Process dialog box.
2. Ensure that **Show processes from all users** is checked.
3. In the Available Processes list view, choose the Voting*Service.exe* processes and click **Attach.**



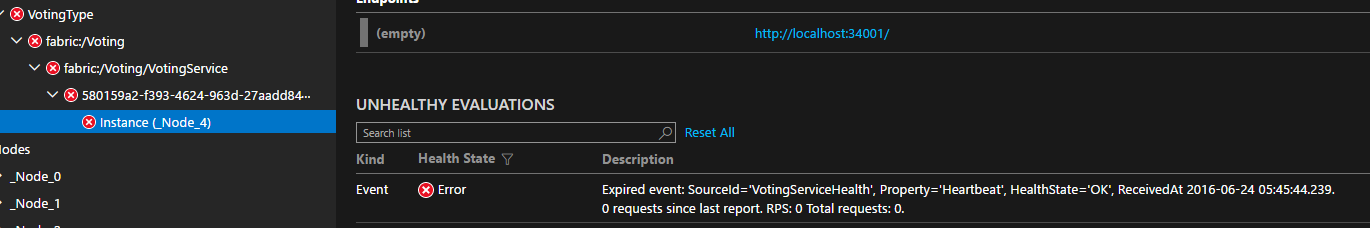
1. Add some voting items and vote for them. As you do this, you’ll see events appearing in the Diagnostic Events window. If events do not appear, close and reopen the window and ensure that the filter has the event source name that is listed at the top of *ServiceEventSource.cs.*



and when looking at SFX, select the instance and then click the **DETAILS** on the instance view and you’ll see the load report and the health report. Revisit the UX making requests and you see the values increase.



1. **Place a breakpoint in the *ReportHealthAndLoad* method in VotingService.cs and allow it to sit at the breakpoint for a few minutes**. In SFX, you’ll see that the instance goes into error because the time to live of the HealthReport expired. This can help detect issues with your service health reporting code.



1. **Remove the breakpoint and allow the service code to run** and after ~30 seconds, the service will once again indicate everything is healthy. This is a simple report of health; much more can be done using the built-in health model.
2. When done using the application, exit the debugging session this time by selecting **Debug** then **Detach All.** This will leave the application running. This is needed because next we’re going to upgrade the running service. If you want to validate that the application is still running, you can verify that SFX still has the application deployed and the site will still be operating.

## Configuration and Upgrade

This section will demonstrate how to use Service Fabric configuration and perform a no downtime upgrade of the application. We’ll deploy some new code and new configuration and watch Service Fabric upgrade the application across each upgrade domain.

1. In Visual Studio, navigate to the VotingService project, PackageRoot folder, then Config folder and open **Settings.xml.** Add the following within the *Settings* element.

<Section Name="Health">

<Parameter Name="HealthCheckIntervalSeconds" Value="40" />

</Section>

1. Open *VotingService.cs*. In the *ReportHealthAndLoad* method add the following line of code to the end of the method. This will send a report to the diagnostic event stream in addition to the health system allowing tracking of historical health values.

// Log the health report.

ServiceEventSource.Current.HealthReport(hi.SourceId, hi.Property, Enum.GetName(typeof(HealthState), hi.HealthState), Context.PartitionId, Context.ReplicaOrInstanceId, hi.Description);

1. Ensure that **System.Fabric.Description** is in the using statements of *VotingService.cs*.
2. At the bottom of the **VotingService** class paste the following at the end of the class

private void CodePackageActivationContext\_ConfigurationPackageModifiedEvent(object sender, PackageModifiedEventArgs<ConfigurationPackage> e)

{

ServiceEventSource.Current.Message("CodePackageActivationContext\_ConfigurationPackageModifiedEvent");

LoadConfiguration();

}

private void LoadConfiguration()

{

ServiceEventSource.Current.Message("LoadConfiguration");

// Get the Health Check Interval configuration value.

ConfigurationPackage pkg = Context.CodePackageActivationContext.GetConfigurationPackageObject("Config");

if (null != pkg)

{

if (true == pkg.Settings?.Sections?.Contains("Health"))

{

ConfigurationSection settings = pkg.Settings.Sections["Health"];

if (true == settings?.Parameters.Contains("HealthCheckIntervalSeconds"))

{

int value = 0;

ConfigurationProperty prop = settings.Parameters["HealthCheckIntervalSeconds"];

if (int.TryParse(prop?.Value, out value))

{

\_interval = TimeSpan.FromSeconds(Math.Max(30, value));

\_healthTimer.Change(\_interval, \_interval);

}

ServiceEventSource.Current.HealthReportIntervalChanged("VotingServiceHealth", "IntervalChanged", Context.PartitionId, Context.ReplicaOrInstanceId, (int) \_interval.TotalSeconds);

}

}

}

}

1. In the **VotingService constructor**, add the following

context.CodePackageActivationContext.ConfigurationPackageModifiedEvent += CodePackageActivationContext\_ConfigurationPackageModifiedEvent;

1. In **OnOpenAsync** **replace the body of the method** with the following

// Force a call to LoadConfiguration because we missed the first event callback.

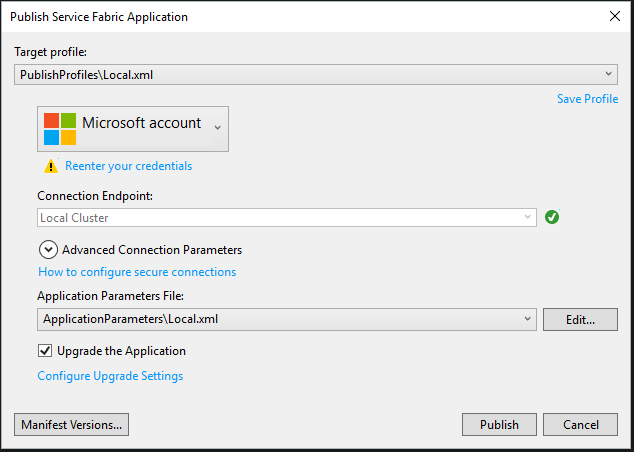
LoadConfiguration();

\_client = new FabricClient();

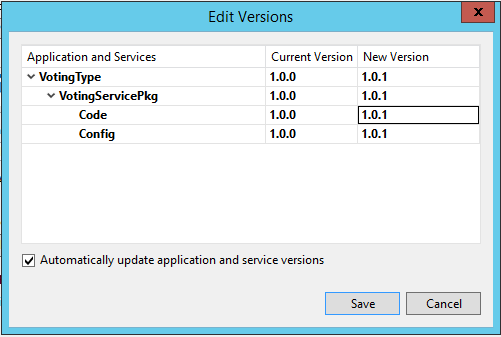
\_healthTimer = new Timer(ReportHealthAndLoad, null, \_interval, \_interval);

return base.OnOpenAsync(cancellationToken);

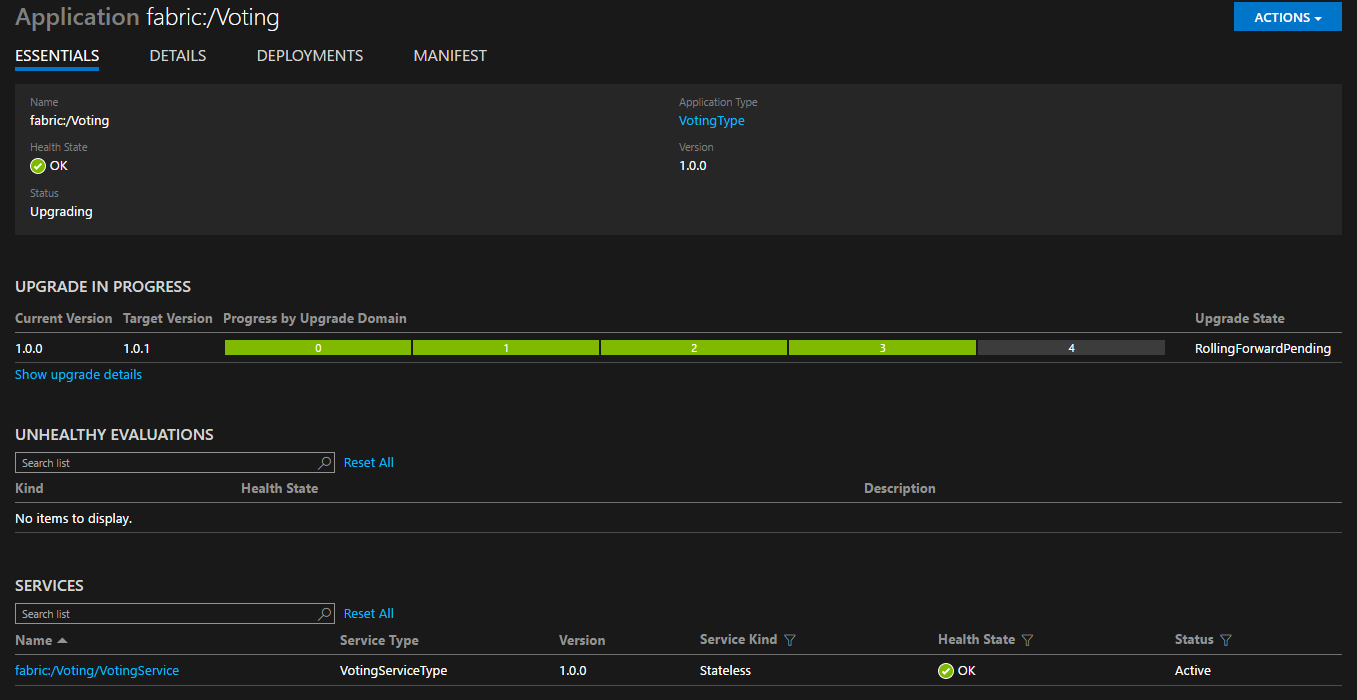
1. In *VotesController.cs* update the version number in the **GetFile** method update the line with the file path. Change the version number from 1.0.0 to 1.0.1. This is the same version that the code will be updated to in step 45.
2. Right click on the Voting project and select **Publish…**, the Publish Service Fabric Application dialog will appear. Select **PublishProfiles\Local.5Node.xml** for the Target profile, which will select the Local Cluster for the Connection Endpoint and **Local.5Node.xml** for Application Parameters File. Ensure the **Upgrade the Application** is selected.



1. Click the **Manifest Versions…** button, the Edit Versions dialog will be displayed. Expand VotingServicePkg. Then change the New Version column value to 1.0.1 for **VotingType**, **VotingServicePkg**, **Code** and **Config**. Click **Save** to close the dialog. This updates the version numbers for the code package, service an application in *ApplicationManifest.xml* and *ServiceManifest.xml*.



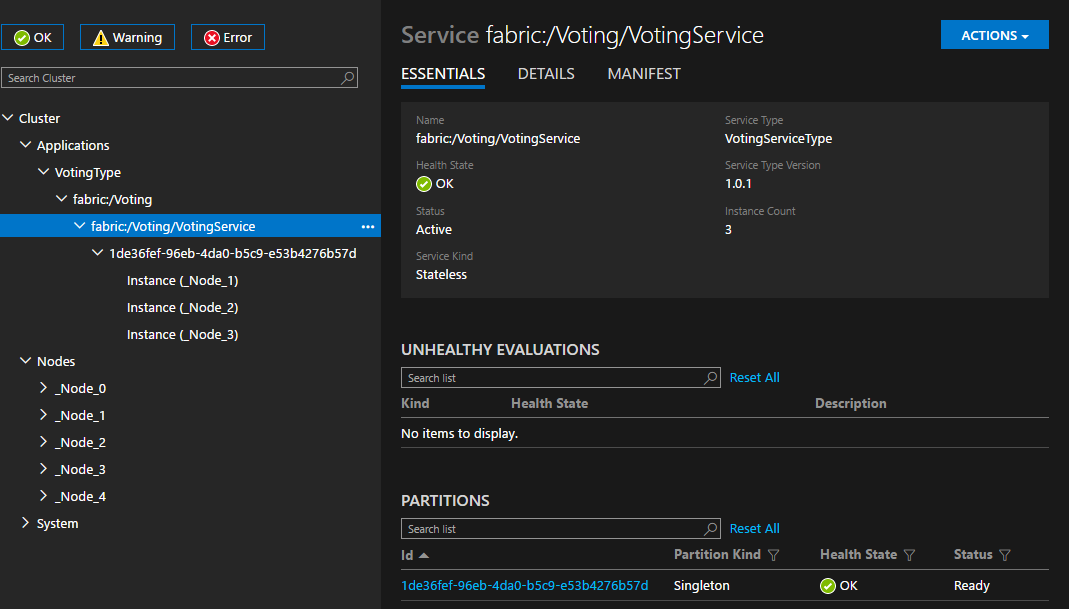
1. Ensure the values are correct and click **Publish** which will start upgrading the application to your local cluster. You can watch progress in SFX by choosing the application fabric:/Voting in the navigation pane.



This page shows the progress as each upgrade domain is being upgraded and shows the original and updated version number.

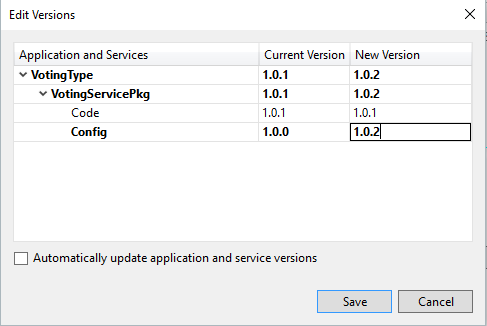
1. Ensure the browser is pointing to the correct endpoint as the instance may have moved nodes during the upgrade. You’ll notice that all the data you entered so far has been lost. This is because during a code upgrade each service instance being upgraded is restarted and because it is a stateless service when it was restarted it lost its cached data – we’ll address that in part II of the lab. Also during the upgrade, the service was unavailable because only one instance was deployed. To address that, we’ll deploy multiple instances of the service. If you view the diagnostic events windows (open using View | Other Windows | Diagnostic Events), you’ll see that the health report interval is being reported every 40 seconds.
2. We want more than a single VM to be hosting this application, so next we’re going to deploy more instances of the service to additional nodes. Visual Studio uses what is called Default Services defined in the ApplicationManifest.xml to deploy services. There is a limitation that you cannot change the configuration of a default application, so instead we’ll use the **Update-ServiceFabricService** Windows PowerShell command to update the number of instances. Open **Windows PowerShell** and type
   1. **Connect-ServiceFabricCluster** and press enter. You should see output indicating you are connected to the cluster.
   2. **Update-ServiceFabricService -ServiceName fabric:/Voting/VotingService -Stateless**

**-InstanceCount 3 -Force** and press enter. You should see a message saying “Update service succeeded” and if you look in SFX, you’ll now see three instances in the navigation page and that the instance count has been changed to three.

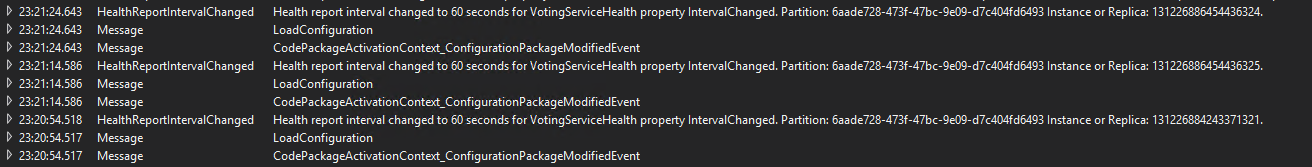


There are now three instances of the stateless service running, each with an independent cache of votes. If you browse to one of them and enter data, then browse to another, the data will of course not be reflected, as mentioned previously, this will be addressed as part of the next lab. The benefit of having multiple instances running is to allow one or more to be down and to still can accept requests from customers.

1. Open **Settings.xml** and change the value of **HealthCheckIntervalSeconds** to 60. We’ll now deploy this configuration change. This will perform a deployment of the configuration only. The *LoadConfiguration* method will be called and the health timer interval will be change to 60 seconds without restarting the code! You can see this by watching for the *HealthReportIntervalChanged* event in the diagnostics viewer.
2. Right click on the Voting project and select **Publish…**, the Publish Service Fabric Application dialog will appear. Ensure **PublishProfiles\Local.5Node.xml** is selected for the Target profile and **Upgrade the Application** is selected.
3. Click the **Manifest Versions…** button, the Edit Versions dialog will be displayed. Change the New Version column value to 1.0.2 for **VotingType**, **VotingServicePkg** and **Config**. Click **Save** to close the dialog.



1. In SFX, select the fabric:/Voting application in the navigation pane. This will allow you to view the upgrade, which should be very fast because it is configuration only and doesn’t require restarting of services.
2. Click **Publish** in the Publish Service Fabric Application dialog when ready. In the diagnostic viewer, you’ll see three sets of the same events, because the deployment is not taking place across the three deployed instances. First you’ll see a Message about the code package event being called, next you’ll see a message that the LoadConfiguration method was called and finally, you’ll see the HealthReportIntervalChanged event indicating the health interval was changed to 60 seconds.



## Upgrade and Rollback

This section will demonstrate how to use Service Fabric configuration and perform a no downtime upgrade of the application but this time checking the application health and rolling back if it doesn’t meet the health policy. So far, all of the upgrades have been ‘UnmonitoredAuto’ upgrades where Service Fabric automates the upgrade but skips the health check. We’ll now move to a ‘Monitored’ upgrade which still automates the upgrade, but includes the health checks.

1. In *VotingService.cs* we’re going to failing health checks to show a rollback. Add the following **to the end** of the **ReportHealthAndLoad** method

// Report failing health report to cause rollback.

var nodeList = \_client.QueryManager.GetNodeListAsync(Context.NodeContext.NodeName).GetAwaiter().GetResult();

var node = nodeList[0];

if ("4" == node.UpgradeDomain || "3" == node.UpgradeDomain || "2" == node.UpgradeDomain)

{

hi = new HealthInformation("VotingServiceHealth", "Error\_Heartbeat", HealthState.Error);

hi.TimeToLive = \_interval.Add(\_interval);

hi.Description = $"Bogus health error to force rollback.";

hi.RemoveWhenExpired = true;

hi.SequenceNumber = HealthInformation.AutoSequenceNumber;

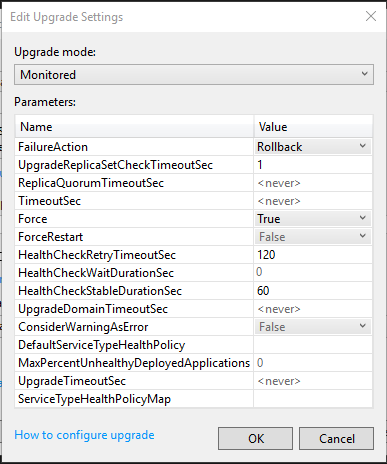
sshr = new StatelessServiceInstanceHealthReport(Context.PartitionId, Context.InstanceId, hi);

\_client.HealthManager.ReportHealth(sshr);

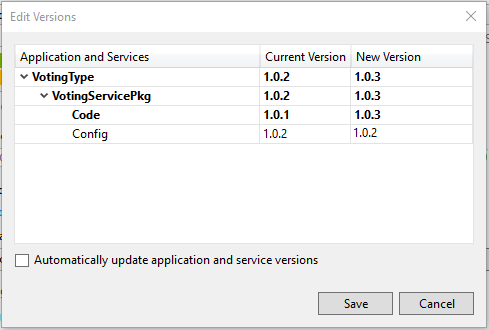
}

1. In Visual Studio, click on **Publish…** in the Voting project which will display the Publish Service Fabric Application dialog. Click on the **Configure Upgrade Settings** option to display the Edit Upgrade Settings dialog. Select **Monitored** for the **Upgrade mode**, which will provide a number of options. Set **HealthCheckRetryTimeoutSec** to 120 and **HealthCheckStableDuractionSec** to 60, then click **OK**.

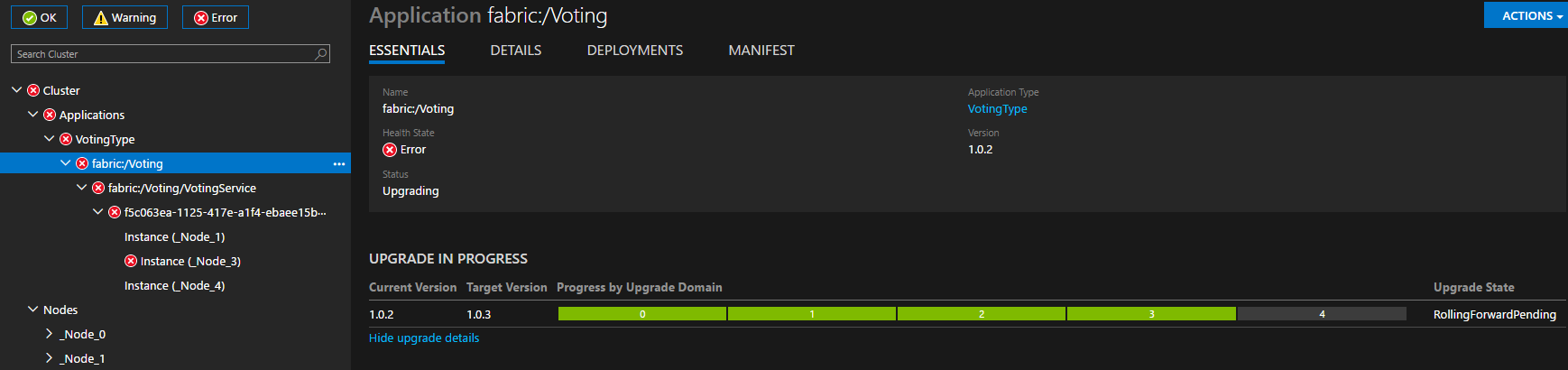
**NOTE**: Settings are not appropriate for general use – they are used here to shorten the lab.



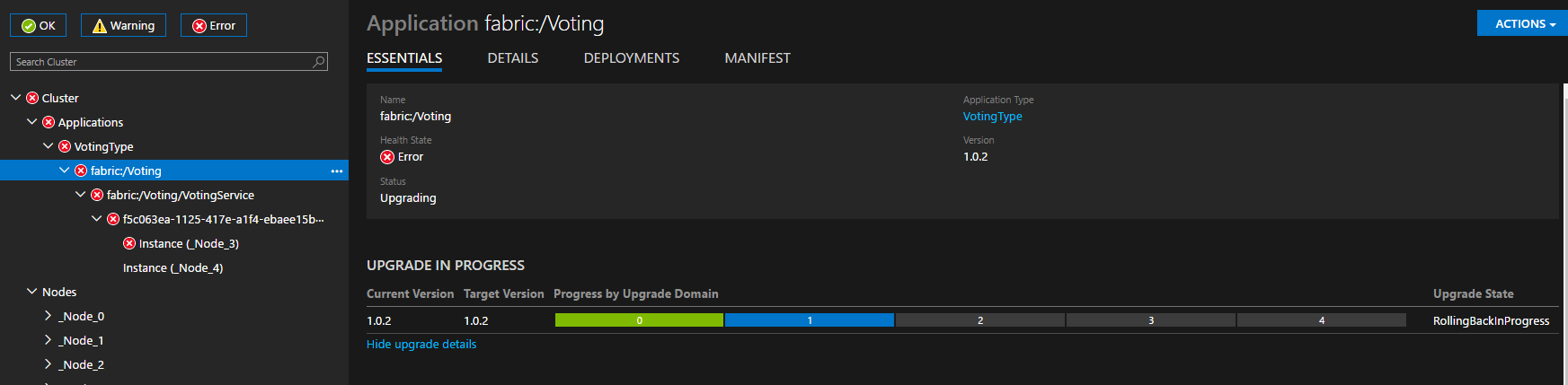
1. Click the **Manifest Versions…** button, the Edit Versions dialog will be displayed. Expand VotingServicePkg. Then change the New Version column value to 1.0.3 for **VotingType**, **VotingServicePkg** and **Code**. Click **Save** to close the dialog. This updates the version numbers for the code package, service an application in *ApplicationManifest.xml* and *ServiceManifest.xml*.



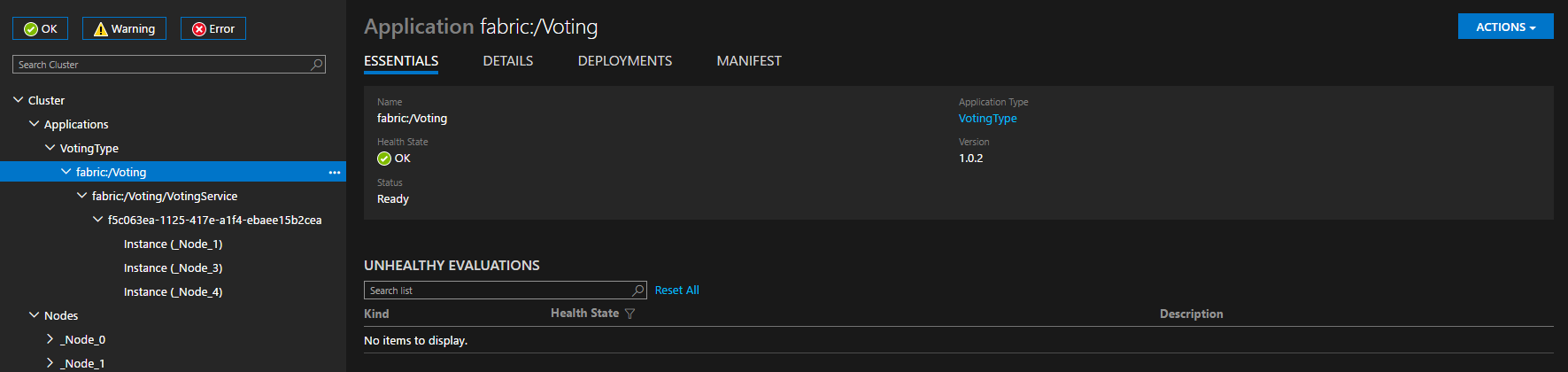
1. Ensure **Upgrade the Application** is selected in the Publish Service Fabric Application dialog, then click **Publish**. Select the Voting application in SFX to watch the progress. How things progress depends on the nodes that your 3 instances were deployed on. Three is important for this demonstration because the code we added, will only work correctly in the first two upgrade domains (UD). In UD2, UD3 and UD4 a health error will be generated. In the example below, instances were deployed on 1, 3. UD0, UD1, UD2 deployed with no error, but UD3 became unhealthy after the health check was performed as shown in the figure below *(if viewing online zoom in to better see the figures)*.



After all of the timeouts have expired, which will be a few minutes event with the reduced duration settings, a rollback to the previous version is automatically performed as shown in the figure below



After rolling back, the previous version 1.0.2 is up and running in all UDs.



You have learned how to create a stateless service, perform logging to ETW, perform health checks, use configuration and perform deployments. In Part II of the lab we’ll fix the issue with each stateful service having its own version of the data by creating a stateful service to contain the voting data. You’ll also learn about the stateless gateway pattern and placement constrains. Part III of the lab we’ll move from a development node to deploying in Azure.

Hope you have enjoyed and learned something new from Part I of the lab!