# ILL – Working with Service Fabric Services – Part II

## Introduction

Estimated time to complete this lab: 60-90 minutes

After completing this lab, you will be able to:

* Understand the concepts of Service Fabric manifests
* Understand the concepts of Service Fabric stateful services including
  + Versioning data using Bond
  + Custom serialization
  + Advantages of using immutable objects
* Use the Stateless Gateway pattern
* Use the built in reverse proxy

## 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 (14.0.25431.01) and [Service Fabric SDK](http://www.microsoft.com/web/handlers/webpi.ashx?command=getinstallerredirect&appid=MicrosoftAzure-ServiceFabric) v2.4.145.9494. 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 your 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.

## Setup the local cluster

For this lab, we need a multi-node cluster installed locally. After installing the SDK, perform the following steps to create a 5-node cluster.

1. 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**
2. On the Service Fabric icon  in the notification area, right click on the icon and choose **Setup Local Cluster -> 5 Node**. After a few minutes, you’ll have a 5-node cluster running locally on your node.

When copying code from Word into Visual Studio, sometimes the formatting in Word interferes with the required formatting for C# within Visual Studio. After pasting, look over the code block to ensure that there aren’t formatting errors.

## Overview of the lab

This lab builds on Part I where a stateless voting service was built. You should now be familiar with how to build a stateless service, monitor service health and basic service metrics, upgrading a service with no downtime and automatically rolling back if errors occur during deployment. If you are not familiar with these concepts, please review Part I of the lab.

If you haven’t done Part I or you no longer have the code, unzip the completed solution.

## Scenario

When we last visited the Voting application, users can vote on anything, but because each instance stored its own state, depending on which instance a user connected to, the user would see different state. In this lab we’re going to add a singleton stateful service that holds all of the voting data giving a consistent view no matter which instance a user connects to. As part of this, we’ll add some additional health and metrics. We’ll then change the service to be a partitioned stateful service so we can handle voting at scale.

## Reliable Dictionaries and Transactions

To remove an application deployment from your OneBox, perform the following steps.

1. Open PowerShell with **elevated privileges**.
2. Enter **Connect-ServiceFabricCluster**. This will connect to the local cluster.
3. Enter **Remove-ServiceFabricApplication -ApplicationName fabric:/Voting -Force -ForceRemove**
4. Enter **Unregister-ServiceFabricApplicationType -ApplicationTypeName VotingType -ApplicationTypeVersion** x*.x.x* **-Force**. Where x.x.x is the version number displayed in the Service Fabric Explorer.

## Lab Part II – Building a Stateful 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 to support debugging within Visual Studio.
2. Select **File** | **Open** | **Project/Solution…** and select the **Voting.sln** file in the “Voting Lab Code – Start Here” project directory and click **Open**.
3. In the **VotingService** project, locate the **ReportHealthAndLoad** method in **VotingService.cs** and comment out or remove the section at the bottom of the method that creates the failing health report. The section should appear like:

//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);

//}

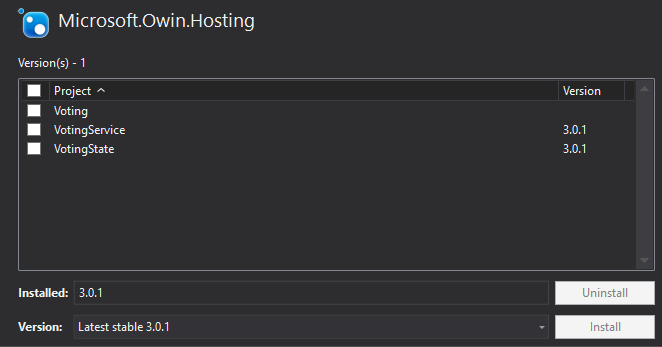
1. In the **VotingService** project, inside the **Controllers** folder, open **VotesController.cs** and add **System.Fabric** to the using block.
2. Locate the **GetFile** method in **VotesController.cs**, Find this line

string path = string.Format(@"..\VotingServicePkg.Code.1.0.1\{0}", file);  
and replace it with this

string path = Path.Combine(FabricRuntime.GetActivationContext().GetCodePackageObject("Code").Path,"index.html");

Now we’re using Service Fabric’s runtime activation context to dynamically find out where the HTML files are located.

1. In the **Voting** project, right client on **Services** and choose **Add** then **New Service Fabric Service**… which displays the New Service Fabric Service dialog. Select **Stateful Service** then enter “**VotingState**” for the Service Name and **click OK**. A new project is added to the solution.
2. Right client on the solution and choose **Manage NuGet Packages for Solution…** This will open the NuGet – Solution tab in Visual Studio. Select the “Microsoft.Owin.Hosting” package. In the window showing the selected projects for the package, select the **VotingState** project and click **Install**. Accept the license in the dialog if it appears. When completed, the window should appear as



*Note: the version number of may be different because you may have selected a newer version of the package. The versions should be the same between the VotingService and VotingState projects*

1. Perform the same steps as in #4 for each of the following packages. Some of them may already have been added because they are dependencies of other packages.
   1. *Microsoft.AspNet.WebApi.Client*
   2. *Microsoft.AspNet.WebApi.Core*
   3. *Microsoft.AspNet.WebApi.Owin*
   4. *Microsoft.AspNet.WebApi.OwinSelfHost*
   5. *Microsoft.Owin*
   6. *Microsoft.Owin.Host.HttpListener*
   7. *Owin*
2. Still in the Manage NuGet Packages for Solution windows you are going to add Bond a package that helps with schematized data to the project. Select **Browse** at the top left and type **Bond.Core.CSharp**. Select the **Bond.Core.CSharp** package, ensure **VotingState** is checked and click **Install**.
3. In the *VotingState* project, create the file **OwinCommunicationsListener.cs** and replace the content of the file with the following:

using System;

using System.Fabric;

using System.Globalization;

using System.Threading;

using System.Threading.Tasks;

using Microsoft.Owin.Hosting;

using Microsoft.ServiceFabric.Services.Communication.Runtime;

using Owin;

using System.Web.Http.Controllers;

using System.Net.Http;

using System.Reflection;

using System.Web.Http.Dispatcher;

using System.Web.Http;

namespace VotingState

{

internal class OwinCommunicationListener : ICommunicationListener, IHttpControllerActivator

{

private readonly ServiceEventSource eventSource;

private readonly StatefulServiceContext serviceContext;

private readonly string endpointName;

private readonly string appRoot;

private IDisposable webApp;

private string publishAddress;

private string listeningAddress;

private readonly IVotingService serviceInstance;

public OwinCommunicationListener(StatefulServiceContext serviceContext,

IVotingService svc, ServiceEventSource eventSource, string endpointName)

: this(serviceContext, svc, eventSource, endpointName, null)

{

}

public OwinCommunicationListener(StatefulServiceContext serviceContext,

IVotingService svc, ServiceEventSource eventSource, string endpointName, string appRoot)

{

if (serviceContext == null) throw new ArgumentNullException(nameof(serviceContext));

if (endpointName == null) throw new ArgumentNullException(nameof(endpointName));

if (eventSource == null) throw new ArgumentNullException(nameof(eventSource));

if (null == svc) throw new ArgumentNullException(nameof(svc));

this.serviceInstance = svc;

this.serviceContext = serviceContext;

this.endpointName = endpointName;

this.eventSource = eventSource;

this.appRoot = appRoot;

}

public bool ListenOnSecondary { get; set; }

public Task<string> OpenAsync(CancellationToken cancellationToken)

{

var serviceEndpoint = this.serviceContext.CodePackageActivationContext.GetEndpoint(this.endpointName);

int port = serviceEndpoint.Port;

if (this.serviceContext is StatefulServiceContext)

{

StatefulServiceContext statefulServiceContext = this.serviceContext as StatefulServiceContext;

this.listeningAddress = string.Format(

CultureInfo.InvariantCulture,

"http://+:{0}/{1}{2}/{3}/{4}",

port,

string.IsNullOrWhiteSpace(this.appRoot)

? string.Empty

: this.appRoot.TrimEnd('/') + '/',

statefulServiceContext.PartitionId,

statefulServiceContext.ReplicaId,

Guid.NewGuid());

}

else

{

throw new InvalidOperationException();

}

this.publishAddress = this.listeningAddress.Replace("+", FabricRuntime.GetNodeContext().IPAddressOrFQDN);

try

{

this.eventSource.ServiceMessage(this.serviceContext, "Starting web server on " + this.listeningAddress);

this.webApp = WebApp.Start(this.listeningAddress, ConfigureApp);

this.eventSource.ServiceMessage(this.serviceContext, "Listening on " + this.publishAddress);

return Task.FromResult(this.publishAddress);

}

catch (Exception ex)

{

this.eventSource.ServiceMessage(this.serviceContext, "Web server failed to open. " + ex.ToString());

this.StopWebServer();

throw;

}

}

public Task CloseAsync(CancellationToken cancellationToken)

{

this.eventSource.ServiceMessage(this.serviceContext, "Closing web server");

this.StopWebServer();

return Task.FromResult(true);

}

public void Abort()

{

this.eventSource.ServiceMessage(this.serviceContext, "Aborting web server");

this.StopWebServer();

}

private void StopWebServer()

{

if (this.webApp != null)

{

try

{

this.webApp.Dispose();

}

catch (ObjectDisposedException)

{

// no-op

}

}

}

// This code configures Web API. This is called from Web.Start.

private void ConfigureApp(IAppBuilder appBuilder)

{

// Configure Web API for self-host.

HttpConfiguration config = new HttpConfiguration();

config.MapHttpAttributeRoutes();

// Replace the default controller activator (to support optional injection of the stateless service into the controllers)

config.Services.Replace(typeof(IHttpControllerActivator), this);

config.Routes.MapHttpRoute(

name: "DefaultApi",

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

defaults: new { id = RouteParameter.Optional }

);

appBuilder.UseWebApi(config);

}

/// <summary>

/// Called to activate an instance of HTTP controller in the WebAPI pipeline

/// </summary>

/// <param name="request">HTTP request that triggered</param>

/// <param name="controllerDescriptor">Description of the controller that was selected</param>

/// <param name="controllerType">The type of the controller that was selected for this request</param>

/// <returns>An instance of the selected HTTP controller</returns>

/// <remarks>This is a cheap way to avoid a framework such as Unity.

/// If already using Unity, that is a better approach.</remarks>

public IHttpController Create(HttpRequestMessage request, HttpControllerDescriptor controllerDescriptor, Type controllerType)

{

// If the controller defines a constructor with a single parameter of the type which implements the service type, create a new instance and inject this.\_serviceInstance

ConstructorInfo ci = controllerType.GetConstructor(BindingFlags.Instance | BindingFlags.Public, null, CallingConventions.HasThis, new[] { typeof(IVotingService) }, new ParameterModifier[0]);

if (null != ci)

{

object[] args = new object[1] { serviceInstance };

return ci.Invoke(args) as IHttpController;

}

// If no matching constructor was found, just call the default parameter-less constructor

return Activator.CreateInstance(controllerDescriptor.ControllerType) as IHttpController;

}

}

}

Lots of code to paste, let’s run though it a bit. This code handles exposing the REST API, it’s essentially the same code as in Part I of the lab with a notable exception. It is initialized with a reference to the service interface. A service reference is passed into the communication listener constructor and kept for initializing the controller. It is passed into the controller by replacing the controller activator in the ConfigureApp method, which causes the pipeline to call the create method. This method looks for the controller constructor that accepts an IVotingService (created in step 17) and creates the controller, injecting the service reference.

1. In the *VotingState* project, open ***VotingState.cs*** and replace the body of the *CreateServiceReplicaListerners* method with

return new ServiceReplicaListener[]

{

new ServiceReplicaListener(serviceContext => new OwinCommunicationListener(

serviceContext, this, ServiceEventSource.Current, "ServiceEndpoint"))

};

1. In the *VotingState* project, open **ServiceManifest.xml** located in the *PackageRoot* folder and replace

<Endpoint Name="ServiceEndpoint" />

with

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

This tells Service Fabric we want to expose a port using HTTP for the service. It will assign an unused port and ACL it correctly. With this change, the project will compile and run, but really won’t do anything new. The next step is to make the stateful service useful.

1. In the *VotingState* project, open **ServiceEventSource.cs** and replace the code block containing ServiceRequestStart and ServiceRequestStop with the following

private const int ServiceRequestStartEventId = 5;

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

public void ServiceRequestStart(string requestTypeName, string activity)

{

WriteEvent(ServiceRequestStartEventId, requestTypeName, activity);

}

private const int ServiceRequestStopEventId = 6;

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

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

{

WriteEvent(ServiceRequestStopEventId, requestTypeName, activity, exception);

}

1. In the *VotingState* project, create a new file named **VoteData.cs** and replace the contents of the file with the following:

using Bond;

using System;

namespace VotingState

{

// Defined as a Bond schema to assist with data versioning and serialization.

// Using a structure and read only properties to make this an immutable entity.

// This helps to ensure Reliable Collections are used properly.

// See https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-

// work-with-reliable-collections

[Bond.Schema]

public struct VotingData

{

// Each field is attributed with an id that is unique. The field definition is never changed.

// This is the name of the vote, which will also be the key of the reliable dictionary.

[Bond.Id(0)]

public string Name { get; private set; }

// This is the current number of votes for this entity.

[Bond.Id(10)]

public int Count { get; private set; }

// This is the maximum number of votes for this entity.

[Bond.Id(20)]

public int MaxCount { get; private set; }

// DateTimeOffset is not a supported data type by Bond

// This is the date and time of the last vote.

[Bond.Id(30), Bond.Type(typeof(long))]

public DateTimeOffset LastVote { get; private set; }

// VotingData constructor.

public VotingData(string name, int count, int maxCount, DateTimeOffset date)

{

this.Name = name;

this.Count = count;

this.MaxCount = maxCount;

this.LastVote = date;

}

// Updates the count of a VotingData structure returning a new instance.

public VotingData UpdateWith(int count)

{

int newMax = Math.Max(this.MaxCount, count);

return new VotingData(this.Name, count, newMax, DateTimeOffset.Now);

}

}

}

1. We’re using Bond for this implementation to demonstrate a way to help version the data being stored in the stateful store. There are other technologies such as [Google’s Protocol Buffers](https://developers.google.com/protocol-buffers/) or [Apache Avro](http://avro.apache.org/) that are similar to Bond. These technologies can convert data stored in older schemas to a newer schema. This is important because it is difficult to keep all data entities within a stateful service at the same version level. Bond help to reduce the burden of versioning.
2. In the *VotingState* project, create a new file named **BondTypeAliasConverter.cs** and replace the contents of the file with the following:

using System;

namespace VotingState

{

/// <summary>

/// This is needed to support DateTimeOffset use in Bond schemas.

/// It converts the DateTimeOffset type to a long value and

/// long values into DateTimeOffsets. See

/// <https://microsoft.github.io/bond/manual/bond_cs.html#converter>

/// </summary>

public static class BondTypeAliasConverter

{

/// <summary>

/// Converts a DateTimeOffset value to a long.

/// </summary>

/// <param name="value">DateTimeOffset value to convert.</param>

/// <returns>Long integer value containing the number of ticks for

/// the DateTimeOffset in UTC.</returns>

public static long Convert(DateTimeOffset value, long unused)

{

return value.UtcTicks;

}

/// <summary>

/// Converts a long value to a DateTimeOffset value.

/// </summary>

/// <param name="value">Long value to convert. This value must

/// be in UTC.</param>

/// <param name="unused"></param>

/// <returns>DateTimeOffset</returns>

public static DateTimeOffset Convert(long value, DateTimeOffset unused)

{

return new DateTimeOffset(value, TimeSpan.Zero);

}

}

}

The *BondTypeAliasConverter* contains static methods to allow transformation between types that are not natively supported by Bond. *DateTimeOffset* is an example of one of these. You can extend the type converter to support types needed in a typical application but are not supported by Bond.

1. In the *VotingState* project, create a new file named **IVotingService.cs** and replace the contents of the file with the following:

using System.Collections.Generic;

using System.Threading;

using System.Threading.Tasks;

namespace VotingState

{

// Voting Service Operations interface.

// For lab purposes, the operations are included

// in the service class. They should be separated

// out to allow for unit testing.

public interface IVotingService

{

// Gets the list of VotingData structures.

Task<IReadOnlyList<VotingData>> GetVotingDataAsync(string id, CancellationToken token);

// Updates the count on a vote or adds a new vote if it doesn't already exist.

Task AddVoteAsync(string key, int count, string id, CancellationToken token);

/// <summary>

/// Tracks the number of requests to the service across all controller instances.

/// </summary>

long RequestCount { get; set; }

}

}

1. In the *VotingState* project, create a new file named **VotingDataSerializer.cs** and replace the contents of the file with the following:

using Microsoft.ServiceFabric.Data;

using System.IO;

using Bond;

using Bond.Protocols;

using Bond.IO.Unsafe;

namespace VotingState

{

// Custom serializer for the VotingData structure.

public sealed class VotingDataSerializer : IStateSerializer<VotingData>

{

private readonly static Serializer<CompactBinaryWriter<OutputBuffer>> Serializer;

private readonly static Deserializer<CompactBinaryReader<InputBuffer>> Deserializer;

static VotingDataSerializer()

{

// Create the serializers and deserializers for FileMetadata.

Serializer = new Serializer<CompactBinaryWriter<OutputBuffer>>(typeof(VotingData));

Deserializer = new Deserializer<CompactBinaryReader<InputBuffer>>(typeof(VotingData));

}

public VotingDataSerializer()

{

}

public VotingData Read(BinaryReader binaryReader)

{

int count = binaryReader.ReadInt32();

byte[] bytes = binaryReader.ReadBytes(count);

var input = new InputBuffer(bytes);

var reader = new CompactBinaryReader<InputBuffer>(input);

return Deserializer.Deserialize<VotingData>(reader);

}

public VotingData Read(VotingData baseValue, BinaryReader binaryReader)

{

return Read(binaryReader);

}

public void Write(VotingData value, BinaryWriter binaryWriter)

{

var output = new OutputBuffer();

var writer = new CompactBinaryWriter<OutputBuffer>(output);

Serializer.Serialize(value, writer);

binaryWriter.Write(output.Data.Count);

binaryWriter.Write(output.Data.Array, output.Data.Offset, output.Data.Count);

}

public void Write(VotingData baseValue, VotingData targetValue, BinaryWriter binaryWriter)

{

Write(targetValue, binaryWriter);

}

}

}

This is a custom serializer for the *VotingData* type. We’re showing how to do custom serialization because the default serializer, the DataContractSerializer, is a bit slow and custom serialization is one of the best practices for increasing stateful service performance.

1. In the *VotingState* project, create a new file names **InitializationCallbackAdapter.cs** and replace the contents of the file with the following:

using Microsoft.ServiceFabric.Data;

using System;

using System.Threading.Tasks;

namespace VotingState

{

// Enables configuration of the state manager.

public sealed class InitializationCallbackAdapter

{

[Obsolete("This method uses a method that is marked as obsolete.", false)]

public Task OnInitialize()

{

// This is marked obsolete, but is supported. This interface is likely

// to change in the future.

var serializer = new VotingDataSerializer();

this.StateManager.TryAddStateSerializer(serializer);

return Task.FromResult(true);

}

public IReliableStateManager StateManager { get; set; }

}

}

This adapter is needed to inject the custom serializer into the *StateManager* instance which can only be done while initializing *VotingState*’s base constructor. We’ll add this to *VotingState* in a few steps. The *OnInitialize* is called to create the custom serializer and add it to the *StateManager* using the *TryAddStateSerializer* method. You’ll notice that this method is marked obsolete, which is why the *OnInitialize* method is also marked obsolete. This is the supported mechanism for custom serialization, but this API will change in the future.

1. In the *VotingState* project, open the **VotingState.cs** file. Add **Microsoft.ServiceFabric.Data** to the using block. Also, add the interface *IVotingService* to the class definition and change the constructor to initialize the custom serializer. It will appear as:

internal sealed class VotingState : StatefulService, IVotingService

{

public VotingState(StatefulServiceContext context, InitializationCallbackAdapter adapter)

: base(context, new ReliableStateManager(context, new ReliableStateManagerConfiguration(onInitializeStateSerializersEvent: adapter.OnInitialize)))

{

adapter.StateManager = this.StateManager;

}

We’re adding the interface to the service as an example of how to isolate the service operation implementation from the actual service implementation. In a real solution, the operations should be separated into a separate class. We’re also providing a new *ReliableStateManager* that is configured to do custom serialization for the *VotingData* type. This initialization is performed by calling the *OnInitialize* method as described previously. The last step is to change **Program.cs** file to accept the context and a new **InitializationCallbackAdapter**. It will appears as:

ServiceRuntime.RegisterServiceAsync("VotingStateType", context => new VotingState(context,

new InitializationCallbackAdapter())).GetAwaiter().GetResult();

1. In **VotingState.cs**, replace the body of the **RunAsync** method with the following:

while (true)

{

cancellationToken.ThrowIfCancellationRequested();

await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);

}

Please note how to correctly handle the *CancellationToken*. If your services are not exiting in a timely manner, check how you are handling the token.

1. At the bottom of the VotingState class, add the following implementation of *IVotingService* and add **using Microsoft.ServiceFabric.Data** to the using block *if needed*. This block of code gets or creates a reliable dictionary, gets the list of *VotingData* structures or adds/updates a VotingData structure instance.

/// <summary>

/// Gets the count dictionary. If it doesn't already exist it is created.

/// </summary>

/// <remarks>This is a good example of how to initialize reliable collections.

/// Rather than initializing and caching a value, this approach will work on

/// both primary and secondary replicas (if secondary reads are enabled.</remarks>

private async Task<IReliableDictionary<string, VotingData>> GetCountDictionaryAsync()

{

return await StateManager.GetOrAddAsync<IReliableDictionary<string, VotingData>>("votingCountDictionary").ConfigureAwait(false);

}

// Track the number of requests to the controller.

private long \_requestCount = 0;

// Returns or increments the request count.

public long RequestCount

{

get { return Volatile.Read(\_requestCount); }

set { Interlocked.Increment(ref \_requestCount); }

}

/// <summary>

/// Gets the list of VotingData items.

/// </summary>

public async Task<IReadOnlyList<VotingData>> GetVotingDataAsync(string id, CancellationToken token)

{

List<VotingData> items = new List<VotingData>();

ServiceEventSource.Current.ServiceRequestStart("VotingState.GetVotingDataAsync", id);

// Get the dictionary.

var dictionary = await GetCountDictionaryAsync();

using (ITransaction tx = StateManager.CreateTransaction())

{

// Create the enumerable and get the enumerator.

var enumItems = (await dictionary.CreateEnumerableAsync(tx)).GetAsyncEnumerator();

while(await enumItems.MoveNextAsync(token))

{

items.Add(enumItems.Current.Value);

if (items.Count > 1000)

break;

}

}

return items;

}

/// <summary>

/// Updates the count on a vote or adds a new vote if it doesn't already exist.

/// </summary>

public async Task AddVoteAsync(string key, int count, string id, CancellationToken token)

{

ServiceEventSource.Current.ServiceRequestStart("VotingState.AddVoteAsync", id);

// Get the dictionary.

var dictionary = await GetCountDictionaryAsync();

using (ITransaction tx = StateManager.CreateTransaction())

{

// Try to get the existing value

ConditionalValue<VotingData> result = await dictionary.TryGetValueAsync(tx, key, LockMode.Update);

if (result.HasValue)

{

// VotingData is immutable to ensure the reference returned from the dictionary

// isn't modified. If it were not immutable, you changed a field’s value, and then the transaction aborts, the value will

// remain modified in memory, corrupting your data.

VotingData newData = result.Value.UpdateWith(result.Value.Count + count);

await dictionary.TryUpdateAsync(tx, key, newData, result.Value);

}

else

{

// Add a new VotingData item to the collection

await dictionary.AddAsync(tx, key, new VotingData(key, count, count, DateTimeOffset.Now));

}

// Commit the transaction.

await tx.CommitAsync();

}

}

This code is the core of the stateful service. *GetCountDictionaryAsync* obtains or creates a new reliable dictionary. As the comments say, this is the preferred way to get the reference. The RequestCount is tracking the number of requests being made by all of the front ends for this partition. The other two methods retrieve, add or update VotingData.

Immutability is important for stateful services, they are NOT the same as .NET collections even though the API seems the same. Never modify a returned object directly as you’ll end up corrupting your data – making *VotingData* immutable avoids this issue.

1. In the *VotingState* project, create a new folder named **Controllers** and create a new file named **StatefulVoteController.cs** and replace the contents of the file with the following:

using System;

using System.Collections.Generic;

using System.Net;

using System.Net.Http;

using System.Threading;

using System.Net.Http.Headers;

using System.Web.Http;

using System.Threading.Tasks;

namespace VotingState.Controllers

{

public sealed class StatefulVoteController : ApiController

{

const string activityHeader = "activity-id";

// Keep an instance of the service.

private IVotingService \_service = null;

// Controller constructor taking a IVotingService instance.

// This is cheap dependency injection done in the listener.

// You can also use your favorite DI framework.

public StatefulVoteController(IVotingService vs)

{

\_service = vs;

}

// GET api/votes

[HttpGet]

[Route("api/votes")]

public async Task<HttpResponseMessage> GetAsync()

{

string activityId = GetHeaderValueOrDefault(Request, activityHeader, () => { return Guid.NewGuid().ToString(); });

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

IReadOnlyList<VotingData> counts = await \_service.GetVotingDataAsync(activityId, CancellationToken.None);

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

foreach (VotingData data in counts)

{

votes.Add(new KeyValuePair<string, int>(data.Name, data.Count));

}

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

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

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

\_service.RequestCount = 1;

return response;

}

[HttpPost]

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

public async Task<HttpResponseMessage> PostAsync(string key)

{

string activityId = GetHeaderValueOrDefault(Request, activityHeader, () => { return Guid.NewGuid().ToString(); });

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

// Update or add the item.

await \_service.AddVoteAsync(key, 1, activityId, CancellationToken.None);

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

\_service.RequestCount = 1;

return Request.CreateResponse(HttpStatusCode.NoContent);

}

/// <summary>

/// Gets a value from a header collection or returns the default value from the function.

/// </summary>

public static string GetHeaderValueOrDefault(HttpRequestMessage request, string headerName, Func<string> getDefault)

{

// If headers are not specified, return the default string.

if ((null == request) || (null == request.Headers))

return getDefault();

// Search for the header name in the list of headers.

IEnumerable<string> values;

if (true == request.Headers.TryGetValues(headerName, out values))

{

// Return the first value from the list.

foreach (string value in values)

return value;

}

// return an empty string as default.

return getDefault();

}

}

}

At the point the stateful service can store and retrieve votes. The REST API is fully functional and can be invoked directly if you want to try it out. POSTing will require a tool such as [Postman](https://www.getpostman.com/) or [Curl](https://curl.haxx.se/) – but we won’t do that as part of this lab.

1. In the stateless **VotingService** project from part one of the lab, we’re going to change the controller to communicate with the stateful service rather than saving the votes locally. To do this open **VotesController.cs**. At the top of the file add the following to the using block **System.Threading.Tasks, Newtonsoft.Json**.
2. Next remove the static \_counts field and then insert

HttpClient \_client = new HttpClient();

1. Replace the **Get** method with the following:

[HttpGet]

[Route("api/votes")]

public async Task<HttpResponseMessage> Get()

{

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

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

Interlocked.Increment(ref \_requestCount);

string url = $"http://localhost:19081/Voting/VotingState/api/votes?PartitionKey=0&PartitionKind=Int64Range";

HttpResponseMessage msg = await \_client.GetAsync(url).ConfigureAwait(false);

string json = await msg.Content.ReadAsStringAsync().ConfigureAwait(false);

List<KeyValuePair<string, int>> votes = JsonConvert.DeserializeObject<List<KeyValuePair<string, int>>>(json);

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

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

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

return response;

}

1. Replace the **Post** method with the following:

[HttpPost]

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

public async Task<HttpResponseMessage> Post(string key)

{

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

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

Interlocked.Increment(ref \_requestCount);

string url = $"http://localhost:19081/Voting/VotingState/api/{key}?PartitionKey=0&PartitionKind=Int64Range";

HttpResponseMessage msg = await \_client.PostAsync(url, null).ConfigureAwait(false);

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

return Request.CreateResponse(msg.StatusCode);

}

1. Replace the **Delete** method with the following:

[HttpDelete]

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

public HttpResponseMessage Delete(string key)

{

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

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

Interlocked.Increment(ref \_requestCount);

// Ignoring delete for this lab.

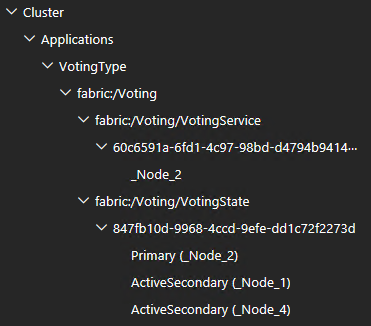
ServiceEventSource.Current.ServiceRequestStop("VotesController.Delete", activityId);

return Request.CreateResponse(HttpStatusCode.NotFound);

}

At this point we can start running the project and interacting with it.

1. Start the project in the debugger by pressing **F5**. The output window will indicate the progress and the Service Fabric Explorer (SFX) will start displaying the deployed application and it’s services. You’ll see that there is a primary and two secondary replicas deployed for the single partition of the stateful service.



1. Open a browser and enter http://localhost:**[Port Number Here]**/api/index.html. Replace [Port Number Here] with the port number of the stateless service. This will display the single page application (SPA) that was created in part one of the lab. You can then add some votes. Everything will work the same as in part one, except delete is not supported.
2. If you want to see what happened while entering votes, look at the Diagnostic viewer (also mentioned in part one) and you’ll see each request and it’s activity flow.
3. Right click on the Service Fabric Local Cluster Manager and select **Manage Cluster**. This will display the Service Fabric Explorer (SFX). Expand the Applications | fabric:/Voting | fabric:/Voting/VotingService and the partition so you can see the nodes. The stateless service should be deployed on a single node at this point. We’re going to change that to 5 nodes.
4. Find the endpoint for the single stateless service and type that URI plus **/api/index.html** into a browser to view the single page app. Go ahead and add a few votes.
5. Open PowerShell
6. Type **Connect-ServiceFabricCluster** to connect to the service fabric cluster.
7. Type **Update-ServiceFabricService -ServiceName fabric:/Voting/VotingService -Stateless -Confirm -Force -InstanceCount 5** then press enter. This will scale out the number of nodes running the stateless service. In SFX you’ll see that there are not 5 nodes assigned to run this service.
8. Now go to a different node and get the type that URI plus **/api/index.html** to get the SPA from a different front end node (it should differ only by the port number). When the application is displayed, press **Refresh** and you’ll see the same votes and counts as you saw in the first browser. You can attach to any number of the stateless services, add on one and refresh on the other and you’ll always get a consistent vote and count.