WorkshopPLUS

Microsoft Azure Service Fabric for Developers

Stateful Services and Partitioning

Student Lab Manual

V1.5, 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

[Build a Stateful Service with Multiple Partitions 4](#_Toc474822134)

[Introduction 4](#_Toc474822135)

[Prerequisites 4](#_Toc474822136)

[Overview 4](#_Toc474822137)

[Task 1 – Creating the stateful service 5](#_Toc474822138)

[Task 2 – Creating the stateless service 12](#_Toc474822139)

[Task 3 – Testing the Application 18](#_Toc474822140)

# Build a Stateful Service with Multiple Partitions

## Introduction

Estimated time to complete this lab

45 minutes

## 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 that the same Resource Group will be used for all Labs.

## Overview

In this lab, we will create a reliable stateful service with multiple partitions. Here, you will build a very simple application where you want to store all last names that start with the same letter in the same partition.

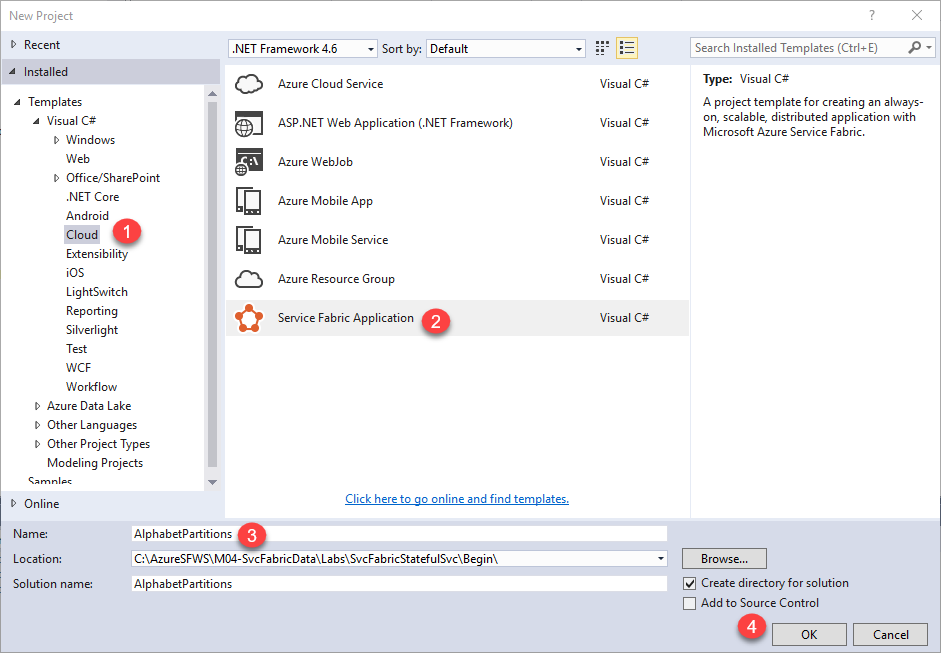
Before you write any code, you need to think about the partitions and partition keys. You need 26 partitions (one for each letter in the alphabet), but what about the low and high keys? As we literally want to have one partition per letter, we can use 0 as the low key and 25 as the high key, as each letter is its own key.

NOTE:

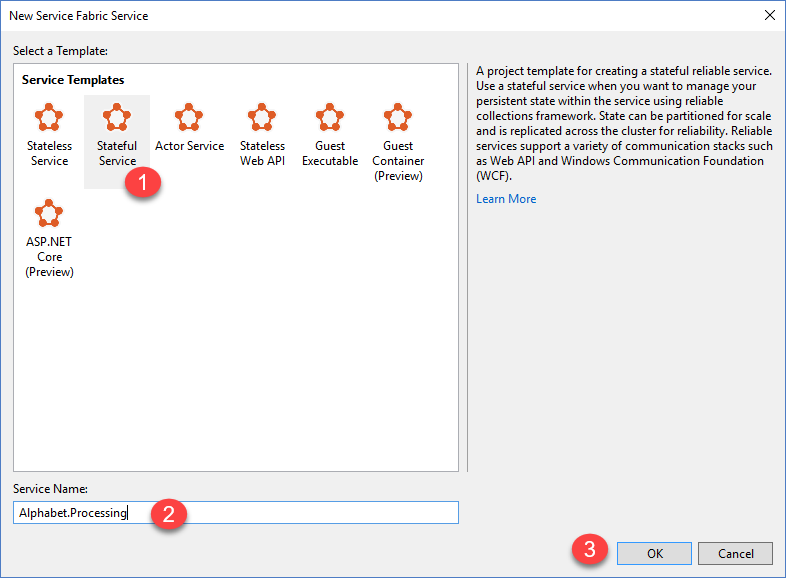
This is a simplified scenario, as in reality the distribution would be uneven. Last names starting with the letters "S" or "M" are more common than the ones starting with "X" or "Y".

## Task 1 – Creating the stateful service

1. Open Visual Studio as Administrator and then select **File | New | Project**.
2. In the New Project dialog box, navigate to **Visual C# | Cloud**, and choose the **Service Fabric** application.
3. Name the project "**AlphabetPartitions**" and place the project in the **.\Labs\SvcFabricStatefulSvc\Begin** directory. Select the **OK** button.



1. In the *Create a Service* dialog box, choose the Stateful Service icon and call it "**Alphabet.Processing**" as shown in the image below. Select the **OK** button.



1. Set the number of partitions. Open the **ApplicationManifest.xml** file in the **AlphabetPartitions** project in the .\ApplicationPackageRoot directory and update the parameter **Processing\_PartitionCount** to **26** as shown below. Also set the replicas and instance counts to 1 when running on your local cluster. This is to allow for running this on more resource constrained machines.

Copy to clipboard

<Parameters>

<Parameter Name="Processing\_MinReplicaSetSize" DefaultValue="1" />

<Parameter Name="Processing\_PartitionCount" DefaultValue="**26**" />

<Parameter Name="Processing\_TargetReplicaSetSize" DefaultValue="1" />

</Parameters>

You also need to update the **LowKey** and **HighKey** properties of the StatefulService element as shown below.

Copy to clipboard

<Service Name="Processing">

<StatefulService ServiceTypeName="ProcessingType"

TargetReplicaSetSize="[Processing\_TargetReplicaSetSize]"

MinReplicaSetSize="[Processing\_MinReplicaSetSize]">

<UniformInt64Partition PartitionCount="[Processing\_PartitionCount]"

**LowKey="0" HighKey="25"** />

</StatefulService>

</Service>

1. For the service to be accessible, open up an endpoint on a port by adding the endpoint element of **ServiceManifest.xml** (located in the Alphabet.Processing projects PackageRoot folder). You can add this as a child of the Endpoints element as shown below:

Copy to clipboard

<Endpoints>

**<Endpoint Name="ProcessingServiceEndpoint" Protocol="http" Type="Internal" />**

…

</Endpoints>

Now the service is configured to listen to an internal endpoint with 26 partitions. We have specified ‘internal’ since we won’t be accessing this port directly from outside the cluster. Also, note that the Type=Internal, which means our intent is to communicate ‘within’ the cluster via http and not from outside the cluster. Service Fabric will assign a port for http communication at runtime.

1. Next, you need to override the CreateServiceReplicaListeners() method of the **Alphabet.Processing** class.

NOTE:

For this sample, we assume that you are using a simple HttpCommunicationListener. For more information on reliable service communication, see [The Reliable Service communication model](https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-services-communication/).

A recommended pattern for the URL that a replica listens on is the following format: {scheme}://{nodeIp}:{port}/{partitionid}/{replicaid}/{guid}. Therefore, you should configure your communication listener to listen on the correct endpoints by using this pattern.

Multiple replicas of this service may be hosted on the same computer, so this address needs to be unique to the replica. This is why partition ID + replica ID are in the URL. An HttpListener can listen on multiple addresses on the same port as long as the URL prefix is unique.

The extra GUID is there for an advanced case where secondary replicas also listen for read-only requests. When that's the case, you want to make sure that a new unique address is used when transitioning from primary to secondary to force clients to re-resolve the address.

'+' is used as the address here so that the replica listens on all available hosts (IP, FQDN, localhost, etc.).

In the **Processing.cs** file of the **Alphabet.Processing** project, Change the

**CreateServiceReplicaListeners** function, and add the **CreateInternalListener** function as

shown below:

protected override IEnumerable<ServiceReplicaListener> CreateServiceReplicaListeners()

{

return new[] {new ServiceReplicaListener(this.CreateInternalListener, "Internal", false)};

}

private ICommunicationListener CreateInternalListener(StatefulServiceContext args)

{

EndpointResourceDescription internalEndpoint =

args.CodePackageActivationContext.GetEndpoint("ProcessingServiceEndpoint");

string uriPrefix = String.Format(

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

internalEndpoint.Protocol,

internalEndpoint.Port,

this.Context.PartitionId,

this.Context.ReplicaId,

Guid.NewGuid());

string nodeIP = FabricRuntime.GetNodeContext().IPAddressOrFQDN;

string uriPublished = uriPrefix.Replace("+", nodeIP);

return new HttpCommunicationListener(uriPrefix, uriPublished,

this.ProcessInternalRequest);

}

Within CreateInternalListener, go to any of the unresolved symbols (with red squiggly lines), hover over them until the lightbulb appears (VS Quick Action), press the down arrow and choose to add the appropriate ‘Using’ statement. **Do not** choose any option that begins with ‘Generate’.

Notice that HttpCommunicationListener and ProcessInternalRequest cannot be resolved to a namespace.

1. The implementation of HttpCommunicationListener has been provided for you. Copy the **HttpCommunicationListener.cs** file from the labs **.\Assets** directory to your **Alphabet.Processing** directory, and add it to your project (right click project and choose **Add | Existing Items**). Copy the namespace from **Processing.cs**, and add it as the namespace in HttpCommunicationListener.cs. This will resolve all but ProcessInternalRequest, which we will implement in the next section.

In the Processing.cs file, examine the code that constructs the URL prefix in ***CreateInternalListener***. It's worth noting that the published URL (uriPublished) is slightly different from the listening URL prefix (uriPrefix). The listening URL is given to *HttpListener*. The published URL is the URL that is published to the Service Fabric Naming Service, which is used for service discovery. You can see this by right-clicking HttpCommunicationListener, choosing ‘Go to definition’, and examining the constructor for the class. Clients will ask for this published address through the discovery service. The address that client request receives needs to have the actual IP or FQDN of the node in order to connect. So you need to replace '+' with the node's IP or FQDN as shown above.

1. The last step is to add the processing logic for **ProcessInternalRequest** to the service in **Processing.cs**, as shown below.

|  |
| --- |
| private async Task ProcessInternalRequest(HttpListenerContext context, CancellationToken cancelRequest)  {  string output = null;  string user = context.Request.QueryString["lastname"].ToString();  try  {  output = await this.AddUserAsync(user);  }  catch (Exception ex)  {  output = ex.Message;  }  using (HttpListenerResponse response = context.Response)  {  if (output != null)  {  byte[] outBytes = Encoding.UTF8.GetBytes(output);  response.OutputStream.Write(outBytes, 0, outBytes.Length);  }  }  } |

|  |
| --- |
| private async Task<string> AddUserAsync(string user)  {  IReliableDictionary<String, String> dictionary = await this.StateManager.GetOrAddAsync<IReliableDictionary<String, String>>("dictionary");  using (ITransaction tx = this.StateManager.CreateTransaction())  {  bool addResult = await dictionary.TryAddAsync(tx, user.ToUpperInvariant(), user);  await tx.CommitAsync();  return String.Format(  "User {0} {1}",  user,  addResult ? "sucessfully added" : "already exists");  }  } |

Again, you can use the Visual Studio Quick Actions to add the appropriate ‘Using’ statements as described above.

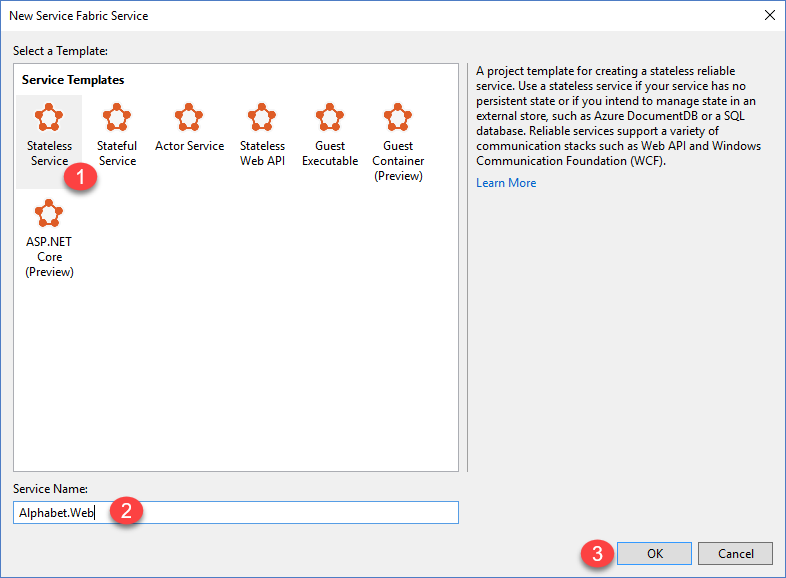
## Task 2 – Creating the stateless service

The previous method you added, ProcessInternalRequest, reads the values of the query string parameter used to call the partition and calls AddUserAsync to add the *lastname* to the reliable dictionary.

Next, Let's add a stateless service as the front-end to the project to see how you can call a particular partition.

This service serves as a simple web interface that accepts the last name as a query string parameter, determines the partition key, and sends it to the **Alphabet.Processing** service for processing.

1. Right click on the main **AlphabetPartitons** *project* and select **Add | New Service Fabric Service**… In the **New Service Fabric Service** dialog box, choose **Stateless** service and name the service "**Alphabet.Web**" as shown below. Select the **OK** button.



.

1. In the **Alphabet.Web** service project, Add a new endpoint in the **ServiceManifest.xml** (in PackageRoot directory) to open up a port as shown below.

Copy to clipboard

<Endpoints>

**<Endpoint Name="WebApiServiceEndpoint" Protocol="http" Port="8091"/>**

</Endpoints>

1. As before, you need to return a collection of ServiceInstanceListeners. Again, you will be using the provided simple HttpCommunicationListener. Follow the same procedure as you did for the stateful service by copying the **HttpCommunicationListener.cs** file to your new project, add it, and change the namespace to **Alphabet.Web**. This would normally be factored out into its own library in production code. Don’t forget to copy and paste the namespace (Alphabet.Web) to the added class.
2. Add the following class level variables to **Web.cs** for the Web class.

|  |
| --- |
| private static readonly Uri alphabetServiceUri = new Uri(@"fabric:/AlphabetPartitions/Processing");  private readonly ServicePartitionResolver servicePartitionResolver = ServicePartitionResolver.GetDefault();  private readonly HttpClient httpClient = new HttpClient(); |

1. You will need to add a reference (right click references in solution explorer > Add Reference) to the **System.Net.Http** class. After this, all using statements can be added through Quick Actions as above.
2. Now, add your own transport by modifying the **CreateServiceInstanceListeners** method as before and then add the supporting **CreateInputListener** method:

|  |
| --- |
| protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()  {  return new[] {new ServiceInstanceListener(this.CreateInputListener, "Input")};  } |

|  |
| --- |
| private ICommunicationListener CreateInputListener(StatelessServiceContext args)  {  EndpointResourceDescription inputEndpoint = args.CodePackageActivationContext.GetEndpoint("WebApiServiceEndpoint");  string uriPrefix = String.Format("{0}://+:{1}/alphabetpartitions/", inputEndpoint.Protocol, inputEndpoint.Port);  string uriPublished = uriPrefix.Replace("+", FabricRuntime.GetNodeContext().IPAddressOrFQDN);  return new HttpCommunicationListener(uriPrefix, uriPublished, this.ProcessInputRequest);  } |

All using statements can be added through Quick Actions as above.

1. In the **Web.cs** file, the CreateInputListener method passes in a reference to the ProcessInputRequest method (to the HttpCommunicationListener). This is done to appropriately handle any cancellation requests that the service may receive.

When a request comes in to the stateless service, the ProcessInputRequest method will be called and this method will process the query string. Paste the method below in to the Web.cs file.

|  |
| --- |
| private async Task ProcessInputRequest(HttpListenerContext context, CancellationToken cancelRequest)  {  String output = null;  try  {  string lastname = context.Request.QueryString["lastname"];  char firstLetterOfLastName = lastname.First();  ServicePartitionKey partitionKey = new ServicePartitionKey(Char.ToUpper(firstLetterOfLastName) - 'A');  ResolvedServicePartition partition = await this.servicePartitionResolver.ResolveAsync(alphabetServiceUri, partitionKey,  cancelRequest);  ResolvedServiceEndpoint ep = partition.GetEndpoint();  JObject addresses = JObject.Parse(ep.Address);  string primaryReplicaAddress =  addresses["Endpoints"]["Internal"].Value<string>();  UriBuilder primaryReplicaUriBuilder = new UriBuilder(primaryReplicaAddress);  primaryReplicaUriBuilder.Query = "lastname=" + lastname;  string result = await  this.httpClient.GetStringAsync(primaryReplicaUriBuilder.Uri);  output = String.Format("Result: {0}. Partition key: '{1}' generated from the first letter '{2}' of input value '{3}'.Processing service partition ID: {4}. Processing service replica address: {5}",  result,  partitionKey.Value,  firstLetterOfLastName,  lastname,  partition.Info.Id,  primaryReplicaAddress);  }  catch (Exception ex)  {  output = ex.Message;  }  using (HttpListenerResponse response = context.Response)  {  if (output != null)  {  byte[] outBytes = Encoding.UTF8.GetBytes(output);  response.OutputStream.Write(outBytes, 0, outBytes.Length);  }  }  } |

In this case, all of the namespaces can be resolved by Quick Actions except for *JObject*. This is contained in the Newtonsoft JSON add on package which you can get from NuGet.

1. Right click on **References** in the **Alphabet.Web** project and choose “**Manage NuGet Packages…**” Once there, you should see **Newtonsoft.Json** as one of the top options. If not, search for Newtonsoft by clicking the Browse link to find it. Click it and choose Install.

Once the package is installed, you will be able to resolve the JObject reference through Quick Actions.

Let's walk through the code you just added in the ProcessInputRequest method step-by-step. The code reads the first letter of the query string parameter lastname into a char. Then, it determines the partition key for this letter by subtracting the hexadecimal value of A from the hexadecimal value of the last names' first letter.

Copy to clipboard

string lastname = context.Request.QueryString["lastname"];

char firstLetterOfLastName = lastname.First();

int partitionKey = Char.ToUpper(firstLetterOfLastName) - 'A';

Remember, for this example, we are using 26 partitions with one partition key per partition. Next, we obtain the service partition partition for this key by using the ResolveAsync method on the servicePartitionResolver object.

Note that in a real scenario, we probably would have chosen a more evenly distributed way to partition the system (such as using a hashing algorithm that hashes a unique key in the dataset (such as email address).

servicePartitionResolver is defined as

Copy to clipboard

private static readonly ServicePartitionResolver servicePartitionResolver = ServicePartitionResolver.GetDefault();

The ResolveAsync method takes the service URI, the partition key, and a cancellation token as parameters. The service URI for the processing service is fabric:/AlphabetPartitions/Processing. Next, we get the endpoint of the partition.

Copy to clipboard

ResolvedServiceEndpoint ep = partition.GetEndpoint()

Finally, we build the endpoint URL plus the querystring and call the processing service.

Copy to clipboard

JObject addresses = JObject.Parse(ep.Address);

string primaryReplicaAddress = addresses["Endpoints"].First()["Value"].Value<string>();

UriBuilder primaryReplicaUriBuilder = new UriBuilder(primaryReplicaAddress);

primaryReplicaUriBuilder.Query = "lastname=" + lastname;

string result = await this.httpClient.GetStringAsync(primaryReplicaUriBuilder.Uri);

Once the processing is done, we write the output back.

1. When you originally set up your local cluster, you had a choice of choosing a 1 node cluster or a 5 node cluster. In the ApplicationPartitions project, ie the actual Service Fabric application project, under the **ApplicationParameters** folder, you will see configuration files named Local.1Node.xml and Local.5Node.xml. Depending on your initial setup configuration of 1 or 5 nodes, the next step will determine which configuration file you modify with the number of partitions.

The last step is to test the service. Visual Studio uses application parameters for the local and cloud deployments. To test the service with 26 partitions locally, you need to update the Local.<xNode>.xml file in the **ApplicationParameters** folder of the **AlphabetPartitions** project as shown below:

Copy to clipboard

<Parameters>

<Parameter Name="Processing\_PartitionCount" Value="**26"** />

<Parameter Name="Web\_InstanceCount" Value="1" />

<Parameter Name="Processing\_MinReplicaSetSize" Value="3" />

<Parameter Name="Processing\_TargetReplicatSetSize" Value="3" />

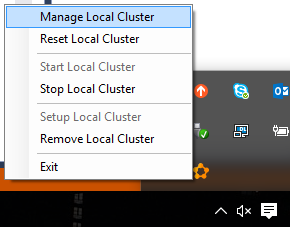
</Parameters>

## Task 3 – Testing the Application

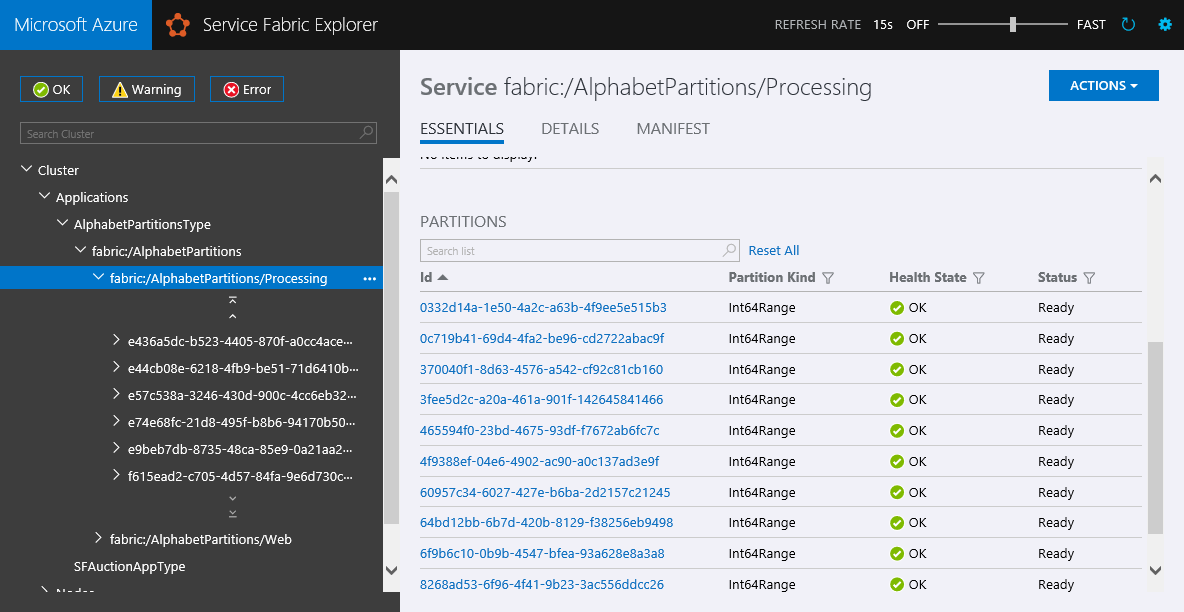
1. Next press **F5**, or the **Run** button to build, deploy, and run the application in debug mode.

Once you finish deployment, you can check the service and all of its partitions in the Service Fabric Explorer.

1. In the system tray, right click on the **Service Fabric Cluster Manager** icon and choose “**Manage Local Cluster**” as shown below.



This will take you to Service Fabric Explorer:



1. In a browser, you can test the partitioning logic by entering http://localhost:8091/AlphabetPartitions/?lastname=<fill in somename here>. You will see that each last name that starts with the same letter is being stored in the same partition.

