

Azure Study Group

AZ-300 - Microsoft Azure Architect Technologies

Jeff Wagner

Partner Technology Strategist



Create and Deploy Apps (5-10%)

Agenda

1

Series
Agenda

2

Speaker
Introduction

3

Feedback
Loop

4

Objective
Review

5

Open Mic

Series Agenda

1	Deploy and Configure Infrastructure (25-30%)
2	Implement Workloads and Security (20-25%)
3	Create and Deploy Apps (5-10%)
4	Implement Authentication and Secure Data (5-10%)
5	Develop for the Cloud (20-25%)

<https://aka.ms/azurecsg>

Series Agenda

1	Deploy and Configure Infrastructure (25-30%)
2	Implement Workloads and Security (20-25%)
3	Create and Deploy Apps (5-10%)
4	Implement Authentication and Secure Data (5-10%)
5	Develop for the Cloud (20-25%)

<https://aka.ms/azurecsg>

Speaker Introduction - Jeff Wagner

- Partner Technology Strategist based in Atlanta
- 21+ years with Microsoft, more in the industry
- Been working with Microsoft Azure when we weren't sure if it was called Windows *Azure* or Windows *Azure*
- Constant learner - *Ancora Imparo*



Feedback Loop

Objectives

Create web apps by using PaaS

May include but not limited to: Create an Azure App Service Web App; create documentation for the API; create an App Service Web App for containers; create an App Service background task by using WebJobs; enable diagnostics logging

Design and develop apps that run in containers

May include but not limited to: Configure diagnostic settings on resources; create a container image by using a Docker file; create an Azure Container Service (ACS/AKS); publish an image to the Azure Container Registry; implement an application that runs on an Azure Container Instance; manage container settings by using code

Create web apps by using PaaS

- Create an Azure App Service Web App
- create documentation for the API
- create an App Service Web App for containers
- create an App Service background task by using WebJobs
- enable diagnostics logging



Understanding Azure Service Fabric



Azure Service Fabric overview

A distributed systems platform that:

Simplifies building, deploying, and managing distributed and scalable applications consisting of microservices and containers running on managed multi-node clusters.

Provides runtime for stateless and stateful microservices running in containers:

Stateless microservices (such as protocol gateways and web proxies) do not maintain a mutable state outside a request and its response from the service.

Stateful microservices (such as user accounts, databases, devices, shopping carts, and queues) maintain a mutable, authoritative state beyond the request and its response.

Powers many existing Microsoft cloud services (e.g. Azure SQL Database, Azure Cosmos DB, Cortana, Power BI, Intune, Event Hubs, IoT Hub, Dynamics 365)

Service Fabric app scenarios

Applications composed of stateful and stateless microservices:

Commonly encountered scenario includes:

stateless web apps (ASP.NET, Node.js, etc.)

stateless and stateful business middle-tier services

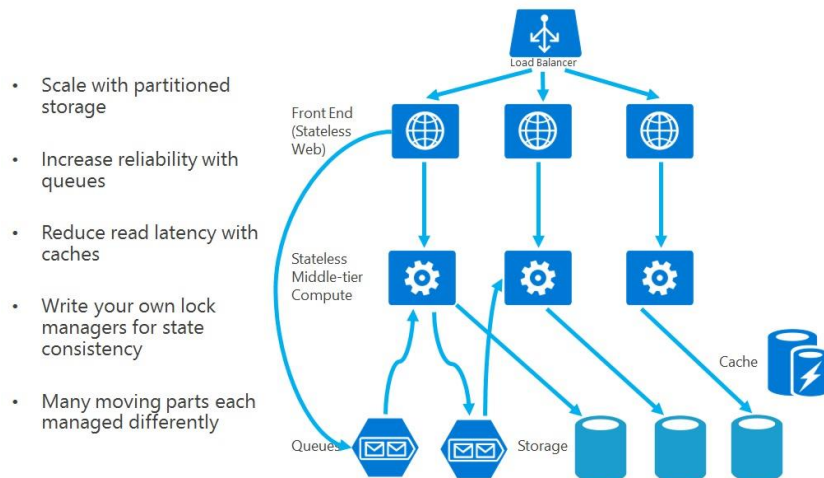
Each services is independent with regard to scale, reliability, and resource usage

The use of stateful services reduces complexity:

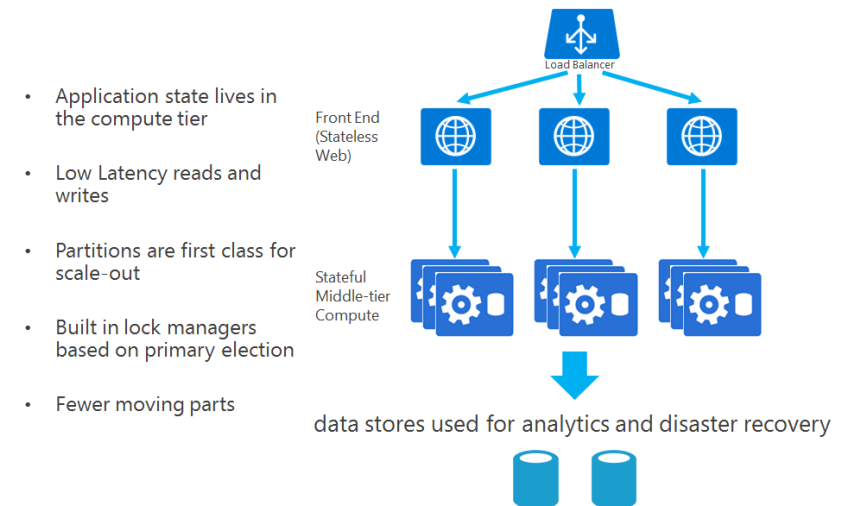
by leveraging Reliable Services and Reliable Actors programming models

By eliminating the need queues and caches

A sample application built using stateless services



A sample application built using stateful services



Reliable Services concepts

Service type:

This is your service implementation. It is defined by the class you write that extends `StatelessService` and any other code or dependencies used therein, along with a name and a version number.

Named service instance:

To run your service, you create named instances of your service type, much like you create object instances of a class type. A service instance has a name in the form of a URI using the "fabric://" scheme, such as "fabric:/MyApp/MyService".

Service host:

The named service instances you create need to run inside a host process. The service host is just a process where instances of your service can run.

Service registration:

Registration brings everything together. The service type must be registered with the Service Fabric runtime in a service host to allow Service Fabric to create instances of it to run.

Creating a reliable service

Creating a stateless service in Visual Studio

Prerequisites:

A computer running supported version of Windows

Visual Studio 2017 with Azure Service Fabric Tools and Azure Service Fabric SDK

Implementation steps:

From Visual Studio:

Create a new Service Fabric Application project:

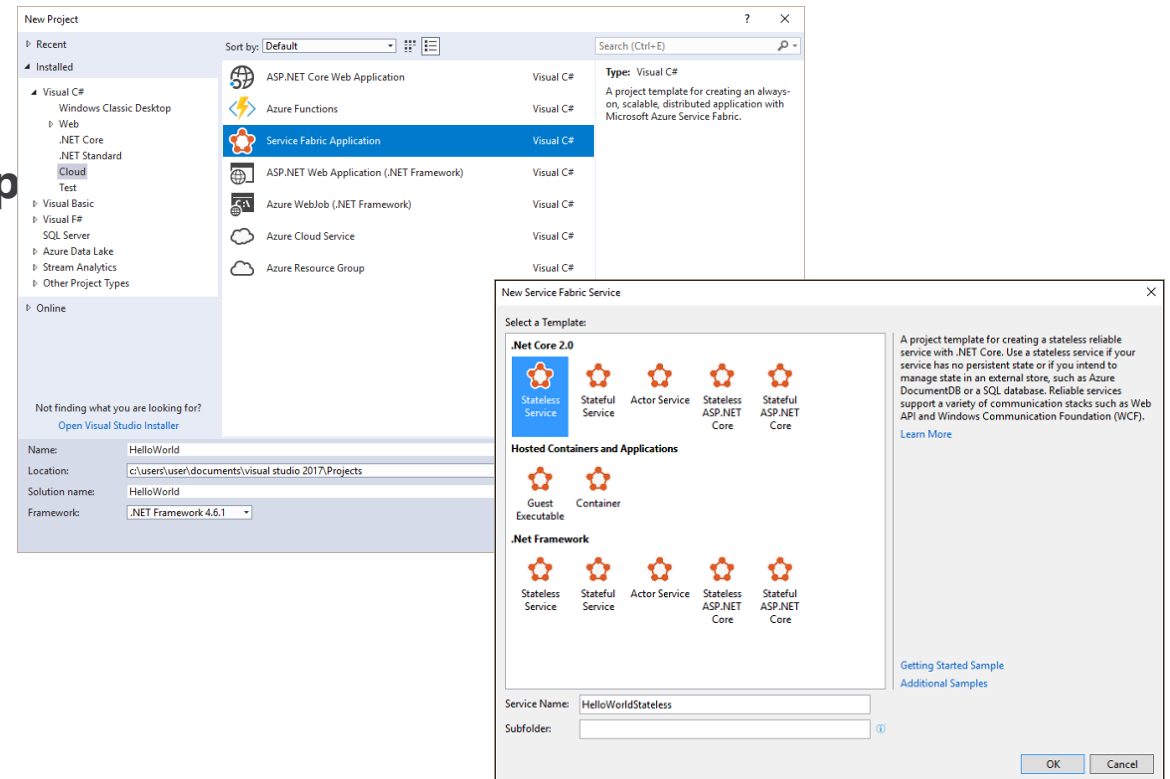
Named HelloWorld

Intended to host services, application manifest, and script

Create a stateless service project using .NET Core 2.0

Named HelloWorldStateless

Intended to host implementation of a stateless service



Implement the service

Implementation steps:

Open the HelloWorldStateless.cs file in the service project.

Note that the service API provides two entry points:

An open-ended entry point method, called `RunAsync`, where you can begin executing any workloads.

A communication entry point where you can plug in your communication stack of choice, such as ASP.NET Core. This is where you can start receiving requests from users and other services.

Review the `RunAsync()` entry point method:

The platform calls this method when an instance of a service is placed and ready to execute.

The project template includes a sample implementation of `RunAsync()` that increments a rolling count.

Problem:

In this stateless service example, the count is stored in a local variable. But because this is a stateless service, the value that's stored exists only for the current lifecycle of its service instance. When the service moves or restarts, the value is lost.

Creating a stateful service in Visual Studio

Objective:

Use a stateful service to convert a counter value from stateless to highly available and persistent, even when the service moves or restarts.

Implementation steps:

From Visual Studio:

Add a stateful service project using .NET Core 2.0

Named HelloWorldStateful

Intended to host implementation of a stateful service

Note that the service API provides the same entry points as the stateless service:

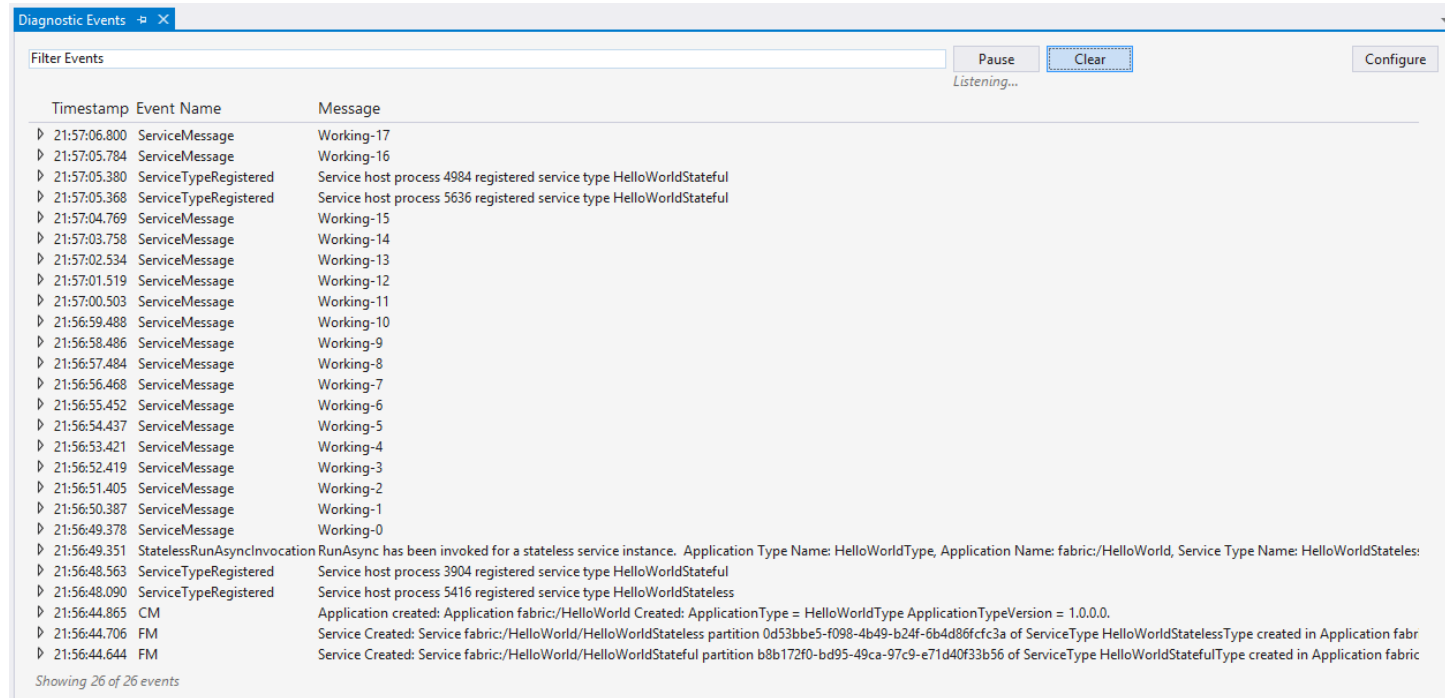
The main difference is the availability of a state provider that can store state reliably. Service Fabric comes with a state provider implementation called Reliable Collections, which lets you create replicated data structures through the Reliable State Manager. A stateful Reliable Service uses this state provider by default.

Review the RunAsync() entry point method:

RunAsync() operates similarly in stateful and stateless services. However, in a stateful service, the platform performs additional work on your behalf before it executes RunAsync(). This work can include ensuring that the Reliable State Manager and Reliable Collections are ready to use.

Run the application

Build and deploy the application from Visual Studio
To monitor its execution, examine Diagnostic Events:
Event Tracing for Windows (ETW) events
From both the stateless service and the stateful service in the application



Timestamp	Event Name	Message
21:57:06.800	ServiceMessage	Working-17
21:57:05.784	ServiceMessage	Working-16
21:57:05.380	ServiceTypeRegistered	Service host process 4984 registered service type HelloWorldStateful
21:57:05.368	ServiceTypeRegistered	Service host process 5636 registered service type HelloWorldStateful
21:57:04.769	ServiceMessage	Working-15
21:57:03.758	ServiceMessage	Working-14
21:57:02.534	ServiceMessage	Working-13
21:57:01.519	ServiceMessage	Working-12
21:57:00.503	ServiceMessage	Working-11
21:56:59.488	ServiceMessage	Working-10
21:56:58.486	ServiceMessage	Working-9
21:56:57.484	ServiceMessage	Working-8
21:56:56.468	ServiceMessage	Working-7
21:56:55.452	ServiceMessage	Working-6
21:56:54.437	ServiceMessage	Working-5
21:56:53.421	ServiceMessage	Working-4
21:56:52.419	ServiceMessage	Working-3
21:56:51.405	ServiceMessage	Working-2
21:56:50.387	ServiceMessage	Working-1
21:56:49.378	ServiceMessage	Working-0
21:56:49.351	StatelessRunAsyncInvocation	RunAsync has been invoked for a stateless service instance. Application Type Name: HelloWorldType, Application Name: fabric:/HelloWorld, Service Type Name: HelloWorldStateless
21:56:48.563	ServiceTypeRegistered	Service host process 3904 registered service type HelloWorldStateful
21:56:48.090	ServiceTypeRegistered	Service host process 5416 registered service type HelloWorldStateless
21:56:44.865	CM	Application created: Application fabric:/HelloWorld Created: ApplicationType = HelloWorldType ApplicationTypeVersion = 1.0.0.0.
21:56:44.706	FM	Service Created: Service fabric:/HelloWorld/HelloWorldStateless partition 0d53bbe5-f098-4b49-b24f-6b4d86fcfc3a of ServiceType HelloWorldStatelessType created in Application fabric
21:56:44.644	FM	Service Created: Service fabric:/HelloWorld/HelloWorldStateful partition b8b172f0-bd95-49ca-97c9-e71d40f33b56 of ServiceType HelloWorldStatefulType created in Application fabric

Showing 26 of 26 events

Creating a Reliable Actors App



Introduction to Service Fabric Reliable Actors

Application framework based on the Virtual Actor pattern:

Provides a single-threaded programming model built on guarantees of Service Fabric.

Leverages the concept of an Actor:

An actor is an isolated, independent unit of compute and state with single-threaded execution.

A large number of actors can execute simultaneously and independently of each other

Actors can communicate with each other and they can create more actors.

Each Service Fabric Reliable Actor service is a partitioned, stateful Reliable Service.

Service Fabric actors are virtual:

Their lifetime is not tied to their in-memory representation, so they do not need to be explicitly created or destroyed.

An actor is automatically activated (and its objects constructed) the first time a message is sent to its actor ID.

If an actor is not used for a period of time, the Reliable Actors runtime garbage-collects the in-memory object.

Applies to a number of scenarios:

The problem space involves thousands (or more) of small, independent, and isolated units of state and logic.

The work involves single-threaded objects that do not interact heavily with external components.

Actor instances will not block callers with unpredictable delays by issuing I/O operations.

Creating the project in Visual Studio

Implementation steps:

1. In Visual Studio, create a new Service Fabric Application project.
2. Choose .NET Core 2.0 Actor Service template and provide a name for the service.
3. Examine the project structure consisting of three projects:

The application project (MyApplication): packages all of the services together for deployment and contains:

ApplicationManifest.xml

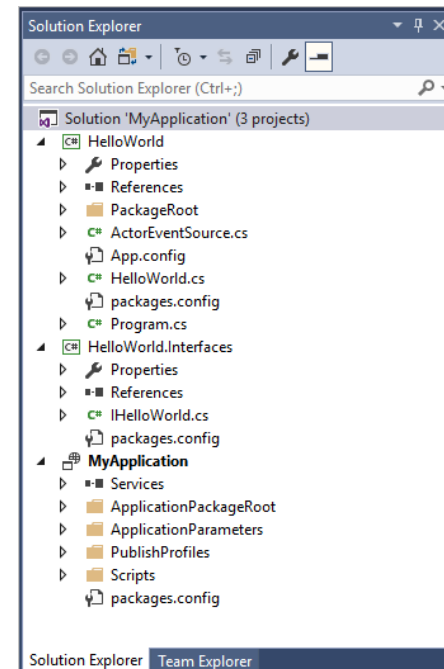
PowerShell scripts for managing the application.

The interface project (HelloWorld.Interfaces):
contains the interface definition for the actor.

The actor service project (HelloWorld):
defines the service that will host the actor and
contains:

the implementation of the actor, HellowWorld.cs,
which, in turn, implements the interfaces defined
in the MyActor.Interfaces project.

Program.cs, which registers actor classes with the
Service Fabric runtime using
`ActorRuntime.RegisterActorAsync<T>()`.



Customizing the actor

Implementation steps:

1. The project template defines methods in the IHelloWorld interface and implements them in the HelloWorld actor implementation. Replace those methods so the actor service returns a simple “Hello World” string:

In the HelloWorld.Interfaces project, in the IHelloWorld.cs file, replace the interface definition

In the HelloWorld project, in HelloWorld.cs, replace the class definition

```
[StatePersistence(StatePersistence.Persisted)]
internal class HelloWorld : Actor, IHelloWorld
{
    public HelloWorld(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task<string> GetHelloWorldAsync()
    {
        return Task.FromResult("Hello from my reliable actor!");
    }
}
```

```
public interface IHelloWorld : IActor
{
    Task<string> GetHelloWorldAsync();
}
```

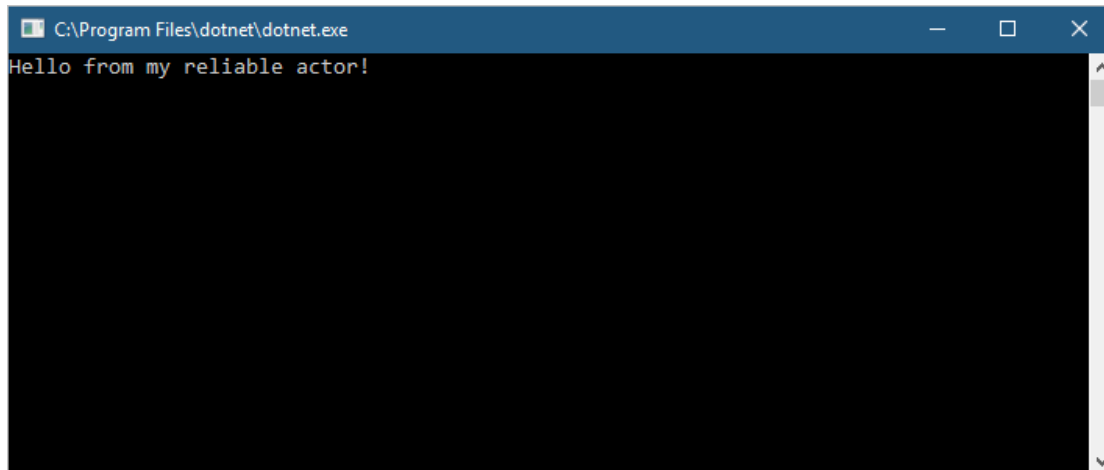
Adding a client

Implementation steps:

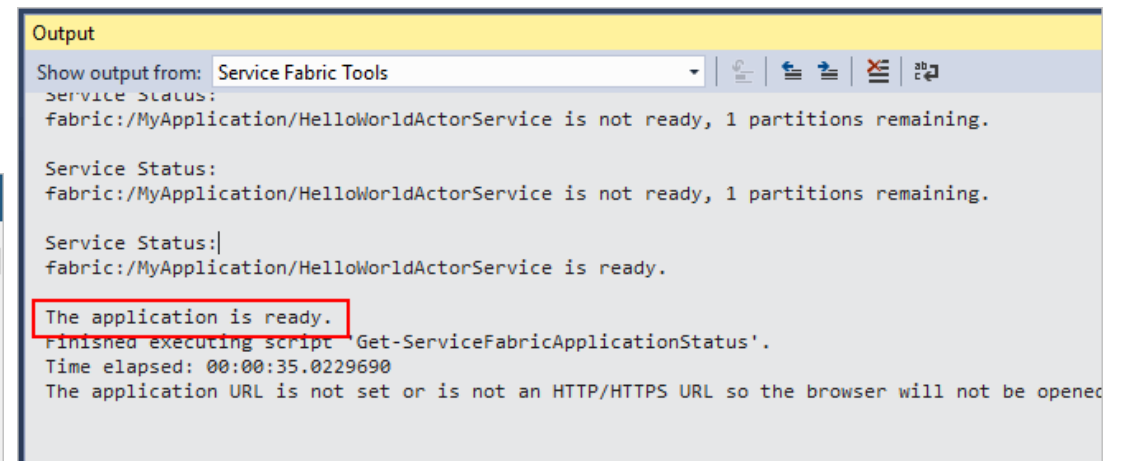
1. In Visual Studio, add a new project.
2. Choose the .NET Core Console App (.NET Core) project type and name the project ActorClient.
3. In Solution Explorer, set the Platform target of the ActorClient project to x64:
[The console application must be 64-bit to maintain compatibility with the interface project.](#)
4. From Package Manager Console of NuGet Package Manager, add the reliable actors package and its dependencies.
5. In the ActorClient project dependencies, add reference to the HelloWorld.Interfaces.
6. In the ActorClient project, replace the entire contents of Program.cs.

Running and debugging

- Implementation steps:
 - 1. Build, deploy, and run the application in the Service Fabric development cluster.
 - 2. Monitor the deployment progress in the Output window.
 - 3. In Solution Explorer, right-click on the ActorClient project, then click Debug > Start new instance. The command line application should display the output from the actor service.



```
C:\Program Files\dotnet\dotnet.exe
Hello from my reliable actor!
```



```
Output
Show output from: Service Fabric Tools
Service Status:
fabric:/MyApplication/HelloWorldActorService is not ready, 1 partitions remaining.

Service Status:
fabric:/MyApplication/HelloWorldActorService is not ready, 1 partitions remaining.

Service Status:|
fabric:/MyApplication/HelloWorldActorService is ready.

The application is ready.
Finished executing script 'Get-ServiceFabricApplicationStatus'.
Time elapsed: 00:00:35.0229690
The application URL is not set or is not an HTTP/HTTPS URL so the browser will not be opened
```


Working with Reliable Collections



Reliable Collections overview

Constitute an evolution of the System.Collections classes:

Implement a set of collections that automatically make your state highly available.

Rely on Reliable Collection APIs to manage the replicated and local state. The state is kept locally in the service instance while also being made highly available:

All writes incur the minimum number of network IOs, resulting in low latency and high-throughput writes.

All reads are local, resulting in low latency and high-throughput reads.

Are inherently:

Replicated

Persisted

Asynchronous

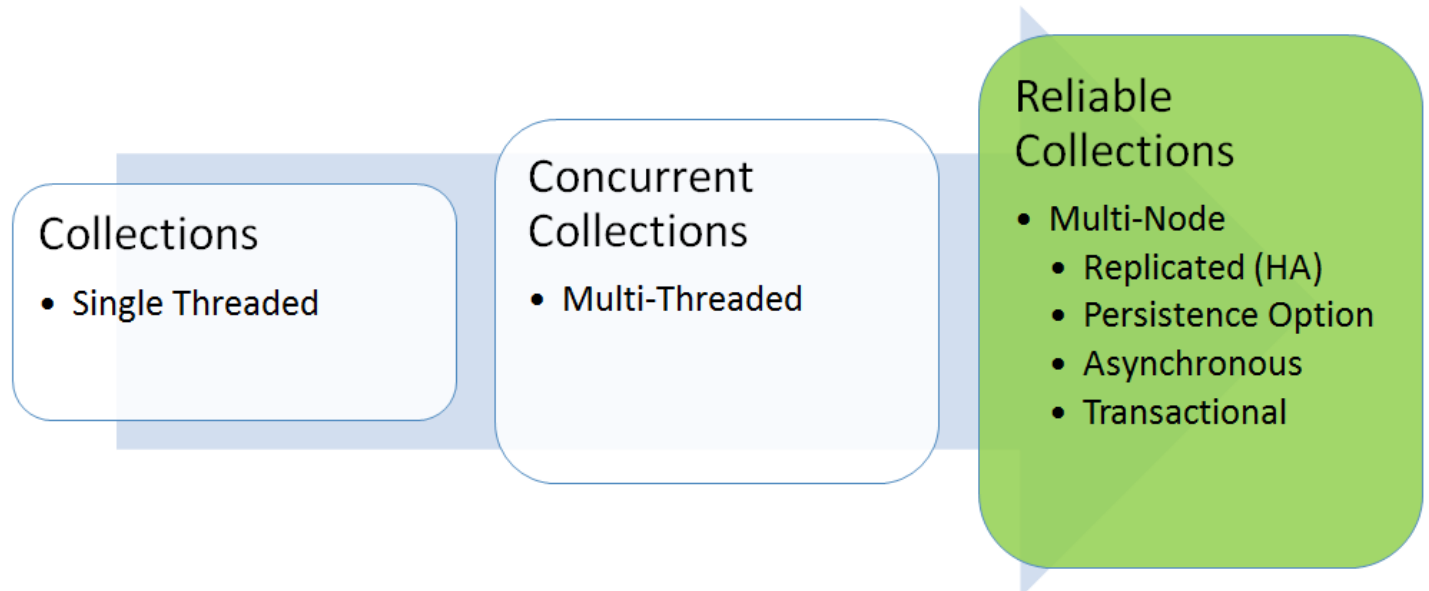
Transactional

Offer three collection types:

Reliable Dictionary

Reliable Queue

Reliable Concurrent Queue



Working with Reliable Collections

Reliable Collections offers a stateful .NET-based programming model:

Based on the reliable dictionary and reliable queue classes:

Providing automatic state management to ensure:

Partitioning (for scalability)

Replication (for availability)

Transaction support (for ACID semantics)

A typical usage of reliable dictionary objects:

All operations (except ClearAsync) require ITTransaction object.

The object is passed to a reliable dictionary's AddAsync method.

If AddAsync modifies the key, the key's write lock is taken.

Any other threads attempting to modify the key are blocked, resulting in a TimeoutException.

The code retries the operation (after exponential back-off).

After acquiring a lock, AddAsync adds the key object references to a dictionary associated with the ITTransaction object.

A call to CommitAsync commits all transaction's operations.

```
///retry:

try {
    // Create a new Transaction object for this partition
    using (ITransaction tx = base.StateManager.CreateTransaction()) {
        // AddAsync takes key's write lock; if >4 secs, TimeoutException
        // Key & value put in temp dictionary (read your own writes),
        // serialized, redo/undo record is logged & sent to
        // secondary replicas
        await m_dic.AddAsync(tx, key, value, cancellationToken);

        // CommitAsync sends Commit record to log & secondary replicas
        // After quorum responds, all locks released
        await tx.CommitAsync();
    }
    // If CommitAsync not called, Dispose sends Abort
    // record to log & all locks released
}
catch (TimeoutException) {
    await Task.Delay(100, cancellationToken); goto retry;
}
```

Design and develop apps that run in containers

- Configure diagnostic settings on resources
- create a container image by using a Docker file
- create an Azure Container Service (ACS/AKS)
- publish an image to the Azure Container Registry
- implement an application that runs on an Azure Container Instance
- manage container settings by using code



Creating an Azure Kubernetes Service Cluster



About Azure Kubernetes Service

Managed Kubernetes cluster hosted in Azure with a range of benefits:

Flexible deployment options

Identity and security management

Integrated logging and monitoring

Cluster node scaling

Cluster node upgrades

HTTP application routing

GPU enabled nodes

Development tooling integration

Virtual network integration

Private container registry

Deploy an AKS cluster using Azure CLI

Implementation steps:

1. Create a resource group:

```
az group create --name myAKSCluster --location eastus
```

2. Create an AKS cluster:

```
az aks create --resource-group myAKSCluster --name myAKSCluster --node-count 1 --enable-addons monitoring --generate-ssh-keys
```

3. Connect to the AKS cluster:

```
az aks get-credentials --resource-group myAKSCluster \
    --name myAKSCluster
```

4. Create a Kubernetes manifest file (azure-vote.yaml)

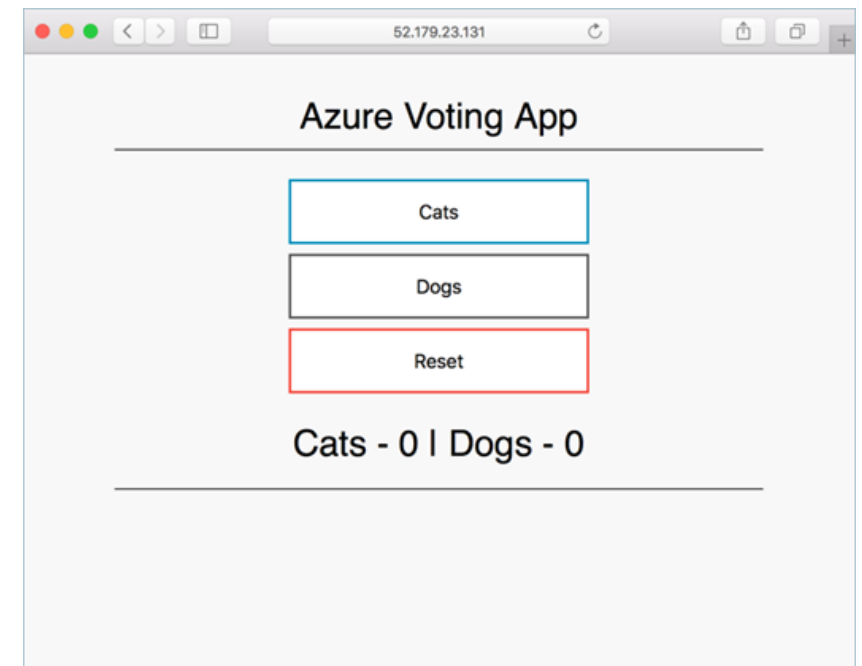
5. Run the containerized application:

```
kubectl apply -f azure-vote.yaml
```

6. Monitor the progress of the deployment:

```
kubectl get service azure-vote-front -w
```

7. Monitor health and logs (from the Azure portal)



Deploy an AKS cluster using Azure Portal

Implementation steps:

1. Create a resource > Kubernetes Service:

Basics: Project details, Cluster details, Scale

Authentication: A new or existing service principal and RBAC settings

Networking: Http application routing and Network configuration (Basic or Advanced)

Monitoring: A new or existing Log Analytics workspace.

2. Connect to the cluster:

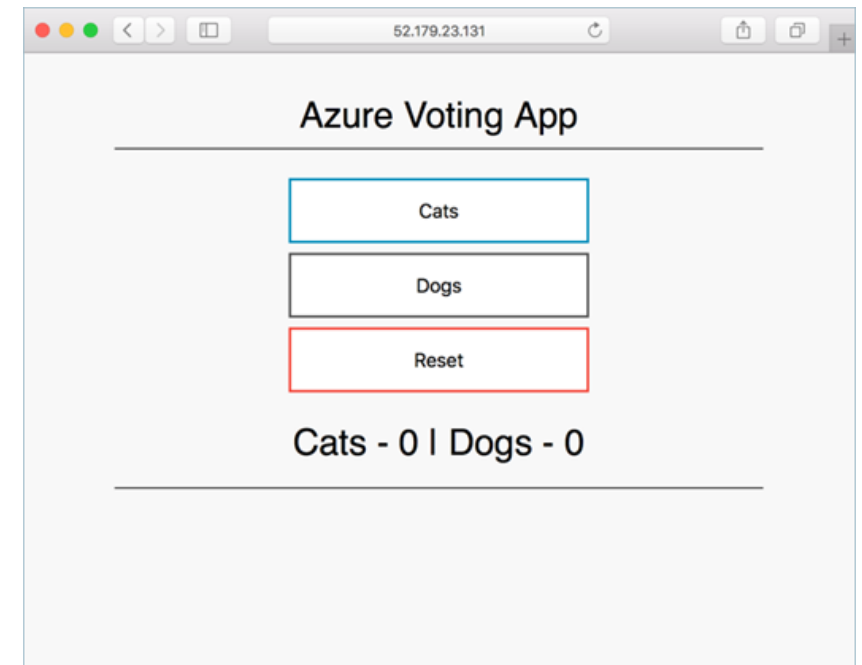
```
az aks get-credentials --resource-group myAKSCluster \  
                        --name myAKSCluster
```

3. Create a Kubernetes manifest file (azure-vote.yaml)

4. Run the containerized application:

```
kubectl apply -f azure-vote.yaml
```

5. Monitor health and logs (from the Azure portal)



Azure Container Registry



Azure Container Registry overview

- Key concepts:
 - **Registry**
 - **Repository**
 - **Image**
 - **Container**
- ACR is a managed container registry based on Docker Registry 2.0:
 - **Integrates with Azure Container Registry Build (for building container images)**
 - **Is available in three service tiers: Basic, Standard, and Premium**
 - **Offers:**
 - **Webhook integration**
 - **Azure AD authentication**
 - **Delete functionality**
 - **Geo-replication (with the Premium tier)**

Deploy an image to ACR using Azure CLI

- Implementation steps:
 - **1. Create a resource group:**
 - `az group create --name myResourceGroup --location eastus`
 - **2. Create a container registry:**
 - `az acr create --resource-group myResourceGroup --name myContainerRegistry007 --sku Basic`
 - **3. Log in to ACR:**
 - `az acr login --name <acrName>`
 - **4. Push a new image to ACR (or pull and tag an existing image first):**
 - `docker pull microsoft/aci-helloworld`
 - `docker tag microsoft/aci-helloworld <acrLoginServer>/aci-helloworld:v1`
 - `docker push <acrLoginServer>/aci-helloworld:v1`
 - **5. Deploy a container by using an image in ACR:**
 - `az container create --resource-group myResourceGroup --name acr-quickstart --image <acrLoginServer>/aci-helloworld:v1 --cpu 1 --memory 1 --registry-username <acrName> --registry-password <acrPassword> --dns-name-label aci-demo --ports 80`

Azure Container Instances



Azure Container Instances Overview

- A managed container hosting option offering a range of benefits:
 - **Fast startup times**
 - **Public IP connectivity and DNS name**
 - **Hypervisor-level security**
 - **Custom sizes**
 - **Persistent storage**
 - **Linux and Windows containers**
 - **Co-scheduled groups**

Implement an application using Virtual Kubelet

- Virtual Kubelet provider is an experimental open source project:
 - **Allows scheduling AKS-managed containers on Azure Container Instances**
 - **Combines functional benefits of AKS with low pricing of ACI**
 - **Supports both Windows and Linux containers**
- To install the Virtual Kubelet provider:
 - **Ensure that the prerequisites are satisfied:**
 - An existing AKS cluster
 - Azure CLI version 2.0.33 and Helm installed
 - A service account and role binding for use with Tiller (for RBAC-enabled AKS clusters)
 - **Run the az aks install-connector command:**
 - `az aks install-connector --resource-group myAKSCluster --name myAKSCluster --connector-name virtual-kubelet --os-type Both`
 - **Once the installation completes, you can deploy Windows and Linux containers to ACI:**
 - In the .yaml file, replace `kubernetes.io/hostname` with the name of the Linux/Windows Virtual Kubelet node

Questions?



Homework Assignment

<https://aka.ms/AZ300>



Open
Mic ...