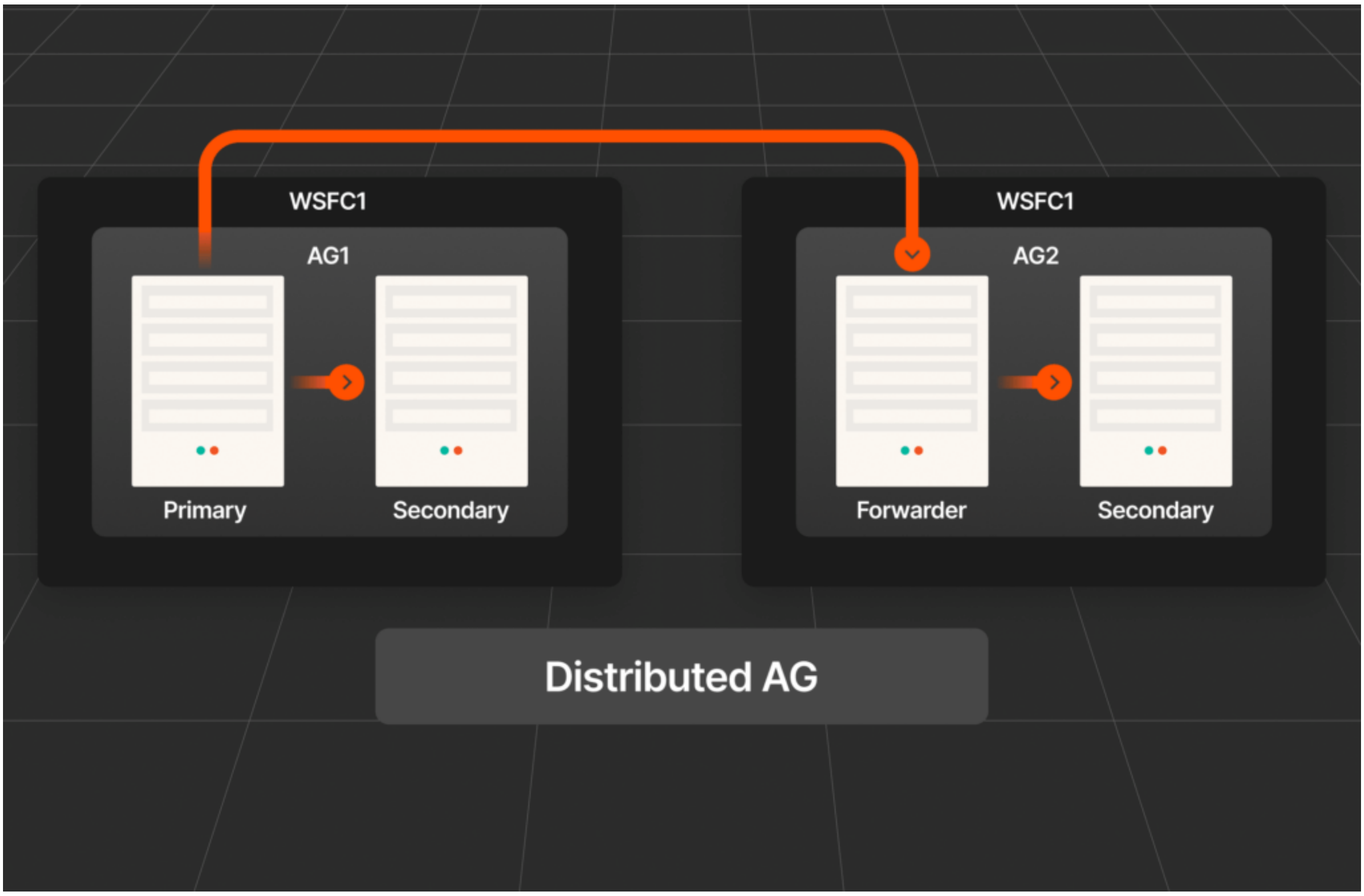
SQL Server Distributed Availability Groups and Kubernetes



This article on SQL Server distributed availability groups originally appeared on Andrew Pruski's blog. It has been republished with the author's credit and consent.

A while back, I wrote about how to use a cross-platform (or clusterless) availability group to seed a database from a Windows SQL instance into a pod in Kubernetes.

I was talking with a colleague last week, and they asked, "What if the existing Windows instance is already in an availability group?"

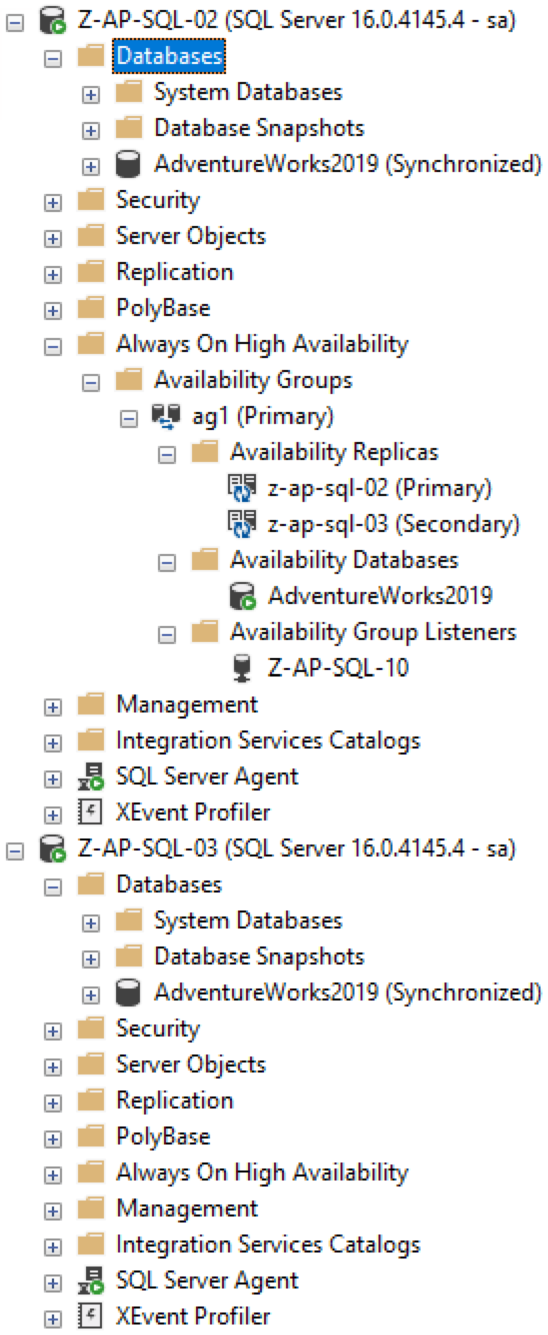
This is a fair question, as it's fairly rare (in my experience) to run a standalone SQL instance in production…most instances are in some form of HA setup, be it a failover cluster instance or an availability group.

Failover cluster instances will work with a clusterless availability group, but it's a different story when it comes to existing availability groups.

A Linux node cannot be added to an existing Windows availability group (trust me, I tried for longer than I'm going to admit), so the only way to do it is to use a distributed availability group.

So let's run through the process!

Here's the existing Windows availability group:

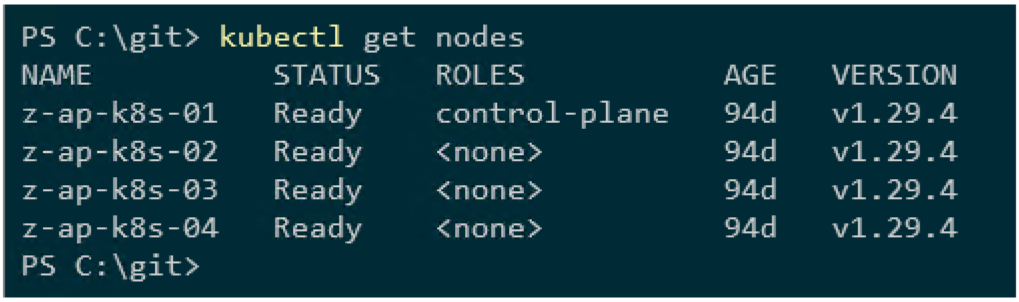


```
⊟ Z-AP-SQL-02 (SQL Server 16.0.4145.4 - sa)
    ⊟ Databases
        ⊞ System Databases
        ⊞ Database Snapshots
        ⊞ AdventureWorks2019 (Synchronized)
    ⊞ Security
    ⊞ Server Objects
    ⊞ Replication
    ⊞ PolyBase
    ⊟ Always On High Availability
        ⊟ Availability Groups
            ⊟ ag1 (Primary)
                ⊟ Availability Replicas
                      z-ap-sql-02 (Primary)
                      z-ap-sql-03 (Secondary)
                ⊟ Availability Databases
                      AdventureWorks2019
                ⊟ Availability Group Listeners
                      Z-AP-SQL-10
    ⊞ Management
    ⊞ Integration Services Catalogs
    ⊞ SQL Server Agent
    ⊞ XEvent Profiler
⊟ Z-AP-SQL-03 (SQL Server 16.0.4145.4 - sa)
    ⊟ Databases
        ⊞ System Databases
        ⊞ Database Snapshots
        ⊞ AdventureWorks2019 (Synchronized)
    ⊞ Security
    ⊞ Server Objects
    ⊞ Replication
    ⊞ PolyBase
    ⊞ Always On High Availability
    ⊞ Management
    ⊞ Integration Services Catalogs
    ⊞ SQL Server Agent
    ⊞ XEvent Profiler
```

Just a standard, two-node AG with one database already synchronized across the nodes. It’s that database we’re going to seed over to the pod running on the Kubernetes cluster using a distributed availability group.

So here’s the Kubernetes cluster:

[crayon-680b8e3b4d3ce524643003/]



```
PS C:\git> kubectl get nodes
NAME           STATUS   ROLES           AGE   VERSION
z-ap-k8s-01    Ready    control-plane   94d   v1.29.4
z-ap-k8s-02    Ready    <none>          94d   v1.29.4
z-ap-k8s-03    Ready    <none>          94d   v1.29.4
z-ap-k8s-04    Ready    <none>          94d   v1.29.4
PS C:\git>
```

Four nodes, one control plane node, and three worker nodes.

OK, so the first thing to do is deploy a *statefulset* running one SQL Server pod (using a file called *sqlserver-statefulset.yaml*):

[crayon-680b8e3b4d3da163023066/]

Here’s the manifest of the statefulset:

[crayon-680b8e3b4d3e0060584920/]

Like my last post, this is pretty stripped down. No resources limits, tolerations, etc. It has two persistent volumes: one for the system databases and one for the user databases from a storage class already configured in the cluster.

One thing to note:

[crayon-680b8e3b4d3e3675201381/]

Here, an entry in the pod’s hosts file is being created for the listener of the Windows availability group.

The next thing to do is deploy two services: one so that we can connect to the SQL instance (on port 1433) and one for the AG (port 5022):

[crayon-680b8e3b4d3e6927324562/]

Here’s the manifest for the services:

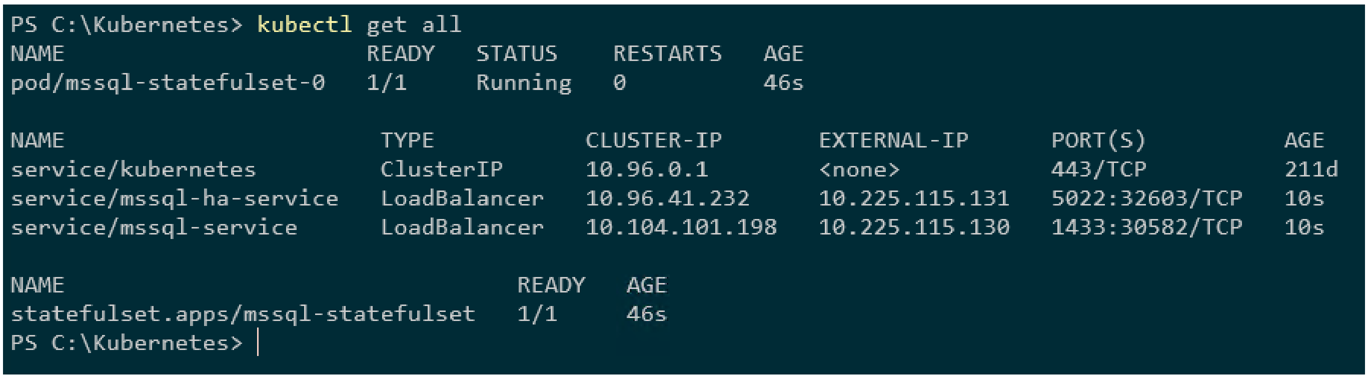
[crayon-680b8e3b4d3f1689615004/]

Note: We could use just one service with multiple ports configured, but I’m keeping them separate here to try and keep things as clear as possible.

Using T-SQL Snapshot Backup: Point-in-time Recovery

Check that everything looks OK:

[crayon-680b8e3b4d3f5668698908/]



```
PS C:\Kubernetes> kubectl get all
NAME                    READY   STATUS    RESTARTS   AGE
pod/mssql-statefulset-0   1/1     Running   0          46s

NAME                      TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)          AGE
service/kubernetes        ClusterIP      10.96.0.1        <none>           443/TCP          211d
service/mssql-ha-service  LoadBalancer   10.96.41.232     10.225.115.131   5022:32603/TCP   10s
service/mssql-service     LoadBalancer   10.104.101.198   10.225.115.130   1433:30582/TCP   10s

NAME                                READY   AGE
statefulset.apps/mssql-statefulset  1/1     46s
PS C:\Kubernetes>
```

Now, we need to create the master key, login, and user in all instances:

[crayon-680b8e3b4d3fb754558993/]

Then, create a certificate in the SQL instance in the pod:

[crayon-680b8e3b4d3fe265104116/]

Back up that certificate:

[crayon-680b8e3b4d401077560700/]

Copy the certificate locally:

[crayon-680b8e3b4d403033112762/]

And then copy the files to the Windows boxes:

[crayon-680b8e3b4d406280470977/]

Once the files are on the Windows boxes, we can create the certificate in each Windows SQL instance:

[crayon-680b8e3b4d409417275144/]

OK, great! Now we need to create a mirroring endpoint in the SQL instance in the pod:

[crayon-680b8e3b4d40b519141313/]

There are already endpoints in the Windows instances, but we need to update them to use the certificate for authentication:

[crayon-680b8e3b4d40e573813945/]

Now, we can create a one-node clusterless availability group in the SQL instance in the pod:

[crayon-680b8e3b4d414481442954/]

No listener here; we're going to use the *mssql-ha-service* as the endpoint for the distributed availability group.

OK, so on the primary node of the Windows availability group, we can create the distributed availability group:

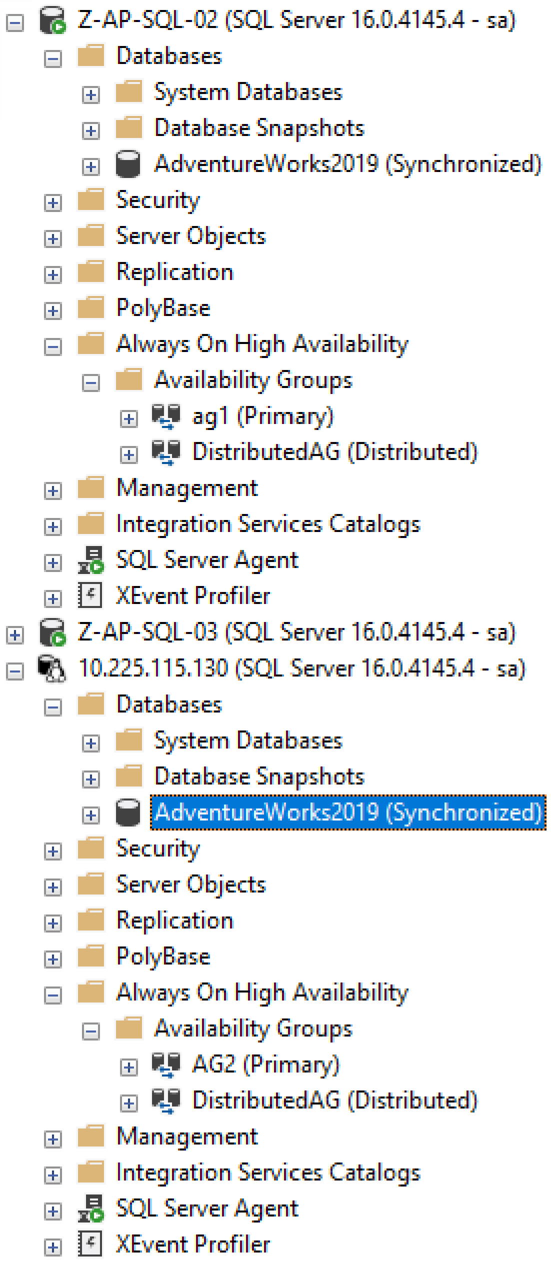
[crayon-680b8e3b4d417724712566/]

We could use a host file entry for the URL in AG2 (I did that in the previous post), but here, we'll just use the IP address of the mssql-ha-service.

OK, nearly there! We now have to join the availability group in the SQL instance in the pod:

[crayon-680b8e3b4d41a435938521/]

And that should be it! If we now connect to the SQL instance in the pod, the database is there!



```
Z-AP-SQL-02 (SQL Server 16.0.4145.4 - sa)
    Databases
        System Databases
        Database Snapshots
        AdventureWorks2019 (Synchronized)
    Security
    Server Objects
    Replication
    PolyBase
    Always On High Availability
        Availability Groups
            ag1 (Primary)
            DistributedAG (Distributed)
    Management
    Integration Services Catalogs
    SQL Server Agent
    XEvent Profiler
Z-AP-SQL-03 (SQL Server 16.0.4145.4 - sa)
10.225.115.130 (SQL Server 16.0.4145.4 - sa)
    Databases
        System Databases
        Database Snapshots
        AdventureWorks2019 (Synchronized)
    Security
    Server Objects
    Replication
    PolyBase
    Always On High Availability
        Availability Groups
            AG2 (Primary)
            DistributedAG (Distributed)
    Management
    Integration Services Catalogs
    SQL Server Agent
    XEvent Profiler
```

There it is! OK, one thing I haven't gone through here is how to get auto-seeding working from Windows into a Linux SQL instance. I went through how that works in my previous post, but the gist is, as long as the database data and log files are located under the Windows SQL instance's default data and log path, they'll auto-seed to the Linux SQL instance's default data and log paths.

So that's how to seed a database from a SQL instance that is in a Windows availability group into a SQL instance running in a pod in a Kubernetes cluster using a distributed availability group.