

# Secure your Microservices on AKS — Part 2



Agraj Mangal

Apr 19 · 12 min read ★

In the [first article](#), we created a very simple Spring Boot App, dockerized it and deployed that to an Azure AD managed AKS cluster using Terraform and Azure Devops. In this article, we continue and make our setup more secure by

1. Using Managed Identities to access Azure Key Vault which contains secret connection strings, instead of using base 64 encoded Kubernetes Secrets and use Azure AD Pod Identity to enable Managed Identities for Pods.
2. Enable Azure Policy for our AKS cluster and enforcing some basic governance such as not allowing privileged containers to run, enforcing container CPU and memory resource limits to prevent resource exhaustion attacks in a Kubernetes cluster, use images from trusted registries to reduce the Kubernetes cluster's exposure risk to unknown vulnerabilities, security issues and malicious images.
3. Since we are not using a Private AKS Cluster, we should whitelist the IP addresses that are allowed to access the Kube API Server and make changes to the state of the cluster.
4. Enable Private Link for Key Vault & SQL db and communicate over the Microsoft backbone network instead of using a public endpoint for the Azure PAAS services.
5. Use Network Policies to limit network traffic between pods in the cluster. We will deploy an additional sample microservice to demonstrate this.
6. Deploy a Web Application Firewall with Application Gateway to protect against standard OWASP attacks such as XSS, CSRF, SQL injection etc.

Again its important to note that this is NOT an exhaustive list and one should refer to official documentation like [AKS Security Baseline](#) for comprehensive coverage on the subject. I also don't plan to cover the recently announced [Open Service Mesh Add-on](#)

for AKS ( in public preview as of today ) but very interesting to enable MTLS b/w your microservices and enabling further security features like restricting communication, monitoring & debugging capabilities etc.

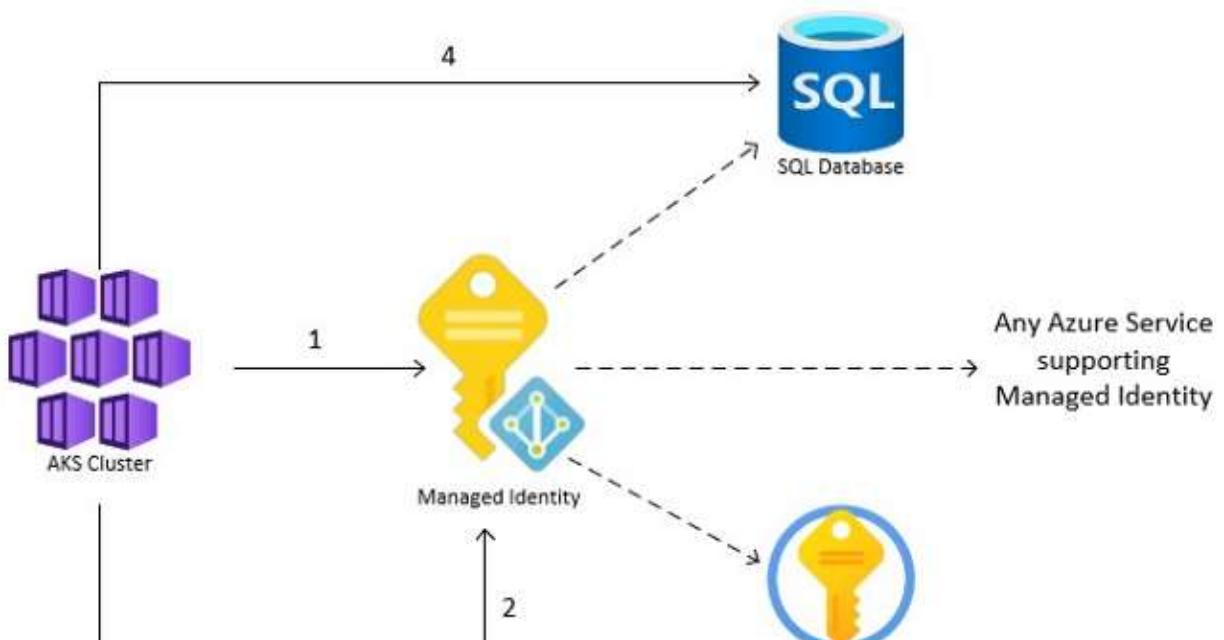
## Key Vault + Azure AD Pod Identity

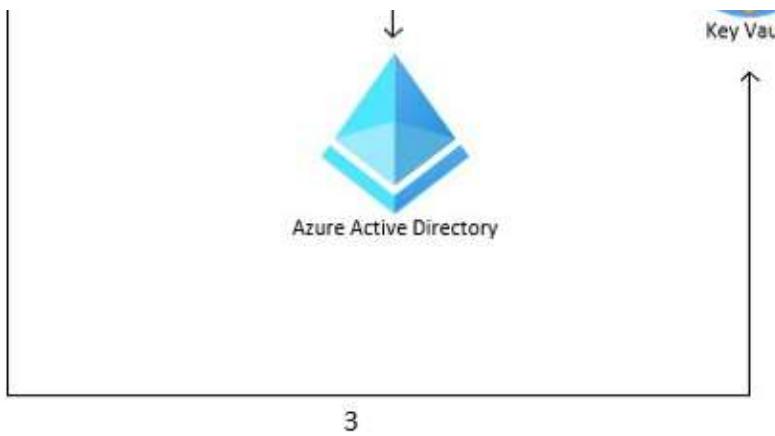
Azure AD Pod Identity is now available as an “add-on” with AKS. Although its in public preview (as of this writing) but it allows you to manage and associate Managed Identities for Azure resources with pods using Kubernetes Custom Resources.

### AD Pod Identity Flow

We will do the following steps:

1. Create a User assigned Managed Identity, a Key Vault for holding secrets & grant it permissions on the Key Vault
2. Modify our Spring Boot App to use Azure Key Vault Integration for secrets.
3. Create Pod Identity & Associate that Managed Identity with the Spring Boot Application Pod
4. Now when the Spring Boot Application will try to talk to KeyVault, the User-assigned Managed Identity will get the access token from Azure AD and use that token to access KeyVault and get our secrets to the application.
5. Once the secrets are passed to the Spring Boot App it will try to connect to SQL using those secrets & run our sample Todos REST API successfully





## How Azure AD Pod Managed Identities Can be Used to Access Azure Resources

### Housekeeping

Since the feature is in preview now, so there are a few housekeeping items you need to perform to enable the feature on your subscription

```
az feature register --name EnablePodIdentityPreview --namespace Microsoft.ContainerService
```

You would also need to update to the preview version of AZ CLI in order to use the CLI to create a pod-identity for your cluster.

```
az extension update --name aks-preview
az aks update -g $AKS_RESOURCE_GROUP -n $AKS_CLUSTER_NAME --enable-pod-identity
```

### Why using AZ CLI and not Terraform ?

It is to be noted that we are consciously not using Terraform for this step (although parts of it like creating the resource group & identity can be done via TF easily) as this feature is still in preview and neither the [AKS module](#) nor the [AKS Kubernetes resource](#) supports the AD Pod Identity at this time. We can still however use the [local provisioner](#) to run the update command on the cluster but until the Terraform resources gets better support its better to stick to AZ CLI for preview features.

Again also very important to note that — we do not recommend using any preview features for your production use cases. The preview features are not supported until

GA and may continue to evolve until they are released. So with that disclaimer in mind, lets get started:

## Step 1: Create a User Managed Identity, Key Vault & Grant permissions to MI on Key Vault

We do this by modifying our Terraform templates and creating key vault and user-assigned Managed Identity using Terraform:

```

1 resource "azurerm_user_assigned_identity" "managed_identity" {
2   resource_group_name = azurerm_resource_group.core.name
3   location            = azurerm_resource_group.core.location
4
5   name    = var.managed_identity_name
6 }
7
8 resource "azurerm_key_vault" "secretstore" {
9   name          = var.keyvault_name
10  location       = azurerm_resource_group.core.location
11  resource_group_name = azurerm_resource_group.core.name
12  enabled_for_disk_encryption = true
13  enabled_for_deployment      = true
14  tenant_id        = var.tenant_id
15  soft_delete_retention_days = 7
16  purge_protection_enabled = false
17  sku_name         = "standard"
18
19  depends_on = [azurerm_user_assigned_identity.managed_identity]
20 }
21
22 output "managed_identity_resource_id" {
23   value = azurerm_user_assigned_identity.managed_identity.id
24 }
25
26 output "managed_identity_client_id" {
27   value = azurerm_user_assigned_identity.managed_identity.client_id
28 }
29
30 output "managed_identity_principal_id" {
31   value = azurerm_user_assigned_identity.managed_identity.principal_id
32 }
```

[terraform\\_keyvault\\_managed\\_identity.tf](#) hosted with ❤ by GitHub

[view raw](#)

To assign the newly created user-assigned Managed Identity to be able to access secrets from Key Vault, we will now create an access policy for the key vault

```
export APP_KEYVAULT_NAME=spbootkeyv2021am #ReplaceWithYourKeyVault
export MANAGED_IDENTITY_ID=$(terraform output -raw
managed_identity_principal_id)
az keyvault set-policy --name $APP_KEYVAULT_NAME --secret-
permissions get list --object-id $MANAGED_IDENTITY_ID
```

## Step 2: Modify the App

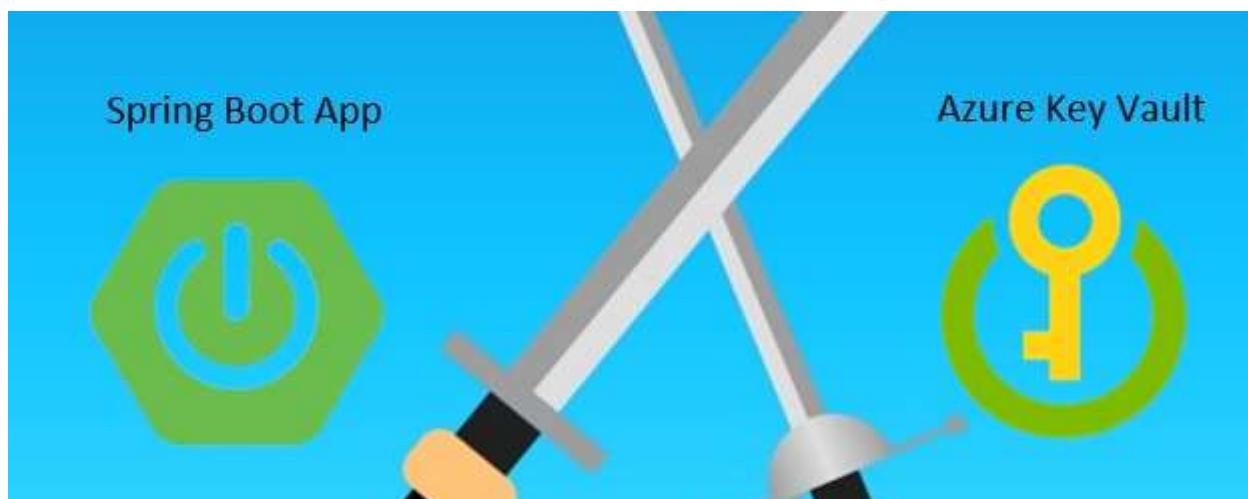
We start by updating our Spring boot application by including the Maven dependency for the [Azure Key Vault Integration](#)

```
<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>azure-spring-boot-starter-keyvault-
secrets</artifactId>
  <version>3.3.0</version>
</dependency>
```

The [Azure Key Vault Integration](#) makes the secrets in Key Vault available as Spring `PropertySource` values to your application, so you can easily access them without having to keep them in configuration files and modifying them at runtime via DevOps pipelines (as we were doing in the first article)

However for this Azure Key Vault integration to work, you would need to put the following properties in your configuration file — `application.properties`

```
azure.keyvault.enabled=true
azure.keyvault.uri=https://<Your_Key_Vault>.vault.azure.net/
azure.keyvault.client-id=<Client_Id_of_MSIS_having_Acces_on_KV>
```





### Spring Boot vs Azure Key Vault on Secret Names & Dot Notations !

Here we run into an interesting problem — It's a well known fact that Azure Key Vault does not support dots . in the names of the secrets and recommends using other characters like dashes - On the other hand, in the Spring world its very common to use dots in your property names -infact we have `spring.datasource.url` , `spring.datasource.username` and `spring.datasource.password` to specify the database connection strings for Azure SQL. So how we do we resolve this when we cannot create a secret with name `spring.datasource.url` in Azure Key Vault and `spring-datasource-url` will not be recognized by Spring ?

Thanks to Stack Overflow, the answer is to build your own `DataSourceConfig` that is annotated as a `Configuration` class so it can be processed by Spring to generate bean definitions & service requests for those beans at runtime. In our case, this is used to construct the `DataSource` that makes use of the secrets in Key Vault and override the built-in `DataSource` that reads the values with dots . in it. Our `DataSource` looks like:

```
1  @Configuration
2  public class DataSourceConfig {
3
4      @Value("${db-user}")
5      String dbUser;
6
7      @Value("${db-password}")
8      String dbPwd;
```

```

9
10    @Value("${db-url}")
11    String dbUrl;
12
13    @Bean
14    public DataSource getDataSource() {
15        DataSourceBuilder dataSourceBuilder = DataSourceBuilder.create();
16        dataSourceBuilder.url(dbUrl);
17        dataSourceBuilder.username(dbUser);
18        dataSourceBuilder.password(dbPwd);
19        return dataSourceBuilder.build();
20    }
21 }
```

[DataSourceSpringKeyVault.java](#) hosted with ❤ by GitHub

[view raw](#)

Now we add the database connection string secrets (`db-url`, `db-user` and `db-password` as required by our custom `DataSource`) to the newly created key vault.

Replace the values with the values for your SQL database (created in the first article)

```

export DATASOURCE_URL="jdbc:sqlserver://#${SQL-SERVER-
NAME}#.database.windows.net:1433;database=#${SQL-DB-
NAME}#;encrypt=true;trustServerCertificate=false;hostNameInCertifica-
te=*.database.windows.net;loginTimeout=30;"
az keyvault secret set --name db-url --value $DATASOURCE_URL --
vault-name $KEYVAULT_NAME
az keyvault secret set --name db-user --value $DATASOURCE_USERNAME --
vault-name $KEYVAULT_NAME
az keyvault secret set --name db-password --value
$DATASOURCE_PASSWORD --vault-name $KEYVAULT_NAME
```

## Cannot Run MSI Locally !

Another catch here is that you cannot run this code locally on your laptop/desktop as with the Managed Identity flow, it would try to communicate with the Instance Metadata Service (IMDS) endpoint ( a well known non-routable endpoint which is not reachable from outside Azure ) to get an access token, which it then uses to access the Key Vault secrets.

Why its not really a problem in this case ? — If we zoom out for a second, we are probably using a Kubernetes cluster even in the Dev environment to deploy our application — so once you update your deployment (by Step 5 below) you would be able to test this. So let's keep our fingers crossed and move on !

As an aside, if its too important for you to test this right here & right now, then a simple workaround would be to spin up a new VM, assign this user-assigned MI to that VM and run your jar on that VM to verify that it works !

### Step 3: Create Pod Identity & Associate MI with it

Before we move ahead with creation of *Pod identity*, we must assign the user-assigned Managed Identity (we created in Step1) the *Reader* permissions on the node resource group of the AKS cluster (in other words, the resource group that contains the VMSS for the AKS cluster)

```
# run these commands in the terraform directory
export MANAGED_IDENTITY_ID=$(terraform output -raw
managed_identity_client_id)
export MANAGED_IDENTITY_RESOURCE_ID=$(terraform output -raw
managed_identity_resource_id)

NODE_GROUP=$(az aks show -g $AKS_RESOURCE_GROUP -n $AKS_CLUSTER_NAME
--query nodeResourceGroup -o tsv)
NODES_RESOURCE_ID=$(az group show -n $NODE_GROUP -o tsv --query
"id")
az role assignment create --role "Reader" --assignee
"$MANAGED_IDENTITY_ID" --scope $NODES_RESOURCE_ID
```

### Create Pod Identity

Let's create a Pod Identity that we can assign this user-assigned Managed Identity and use that Pod Identity with our Spring Boot Deployment so that the node on which the application pods run can communicate with the IMDS service to get access token and then can use the access token to get the secrets from the Key Vault.

```
export POD_IDENTITY_NAME="spboot-pod-identity"
export POD_IDENTITY_NAMESPACE="spboot-app"
az aks pod-identity add --resource-group $AKS_RESOURCE_GROUP --
cluster-name $AKS_CLUSTER_NAME --namespace $POD_IDENTITY_NAMESPACE
--name $POD_IDENTITY_NAME --identity-resource-id
$MANAGED_IDENTITY_RESOURCE_ID
```

The Pod Identity can be either created via Azure CLI (like above) — the `az aks pod-identity` subcommand or by creating Custom Resource Definitions like `AzureIdentity` and `AzureIdentityBinding` and deploying them to the AKS cluster. Infact the above command will create some of those custom resources in your cluster.

The Pod Identity we created above requires certain important parameters

1. `identity-resource-id` The Resource ID for the User-assigned Managed Identity. We got this value using terraform output command.
2. `namespace` The namespace in which this pod identity is created and available for Kubernetes deployments to use

The equivalent YAML for the `AzureIdentity` custom resource is

```

1  apiVersion: aadpodidentity.k8s.io/v1
2  kind: AzureIdentity
3  metadata:
4    name: spboot-pod-identity
5  namespace: spboot-app
6  spec:
7    clientID: $MANAGED_IDENTITY_ID
8    resourceID: $MANAGED_IDENTITY_RESOURCE_ID
9    type: 0
10

```

[AzureIdentity.yaml](#) hosted with ❤ by GitHub

[view raw](#)

The `type: 0` here means user-assigned MSI, `type: 1` for Service Principal with client secret, or `type: 2` for Service Principal with certificate. More info [here](#).

## Step 4: Update the Deployment in AKS

Next we update our deployment yaml to include the `namespace` and assign the Pod Identity we created above using the `aadpodidbinding` label on the Deployment as shown below

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: spring-boot-aks          # Name of the namespace in which Pod Identity is
5    namespace: spboot-app
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10        app: spring-boot-aks
11    template:
12      metadata:

```

```
13 labels:  
14     app: spring-boot-aks  
15     aadpodidbinding: spboot-pod-identity      # Name of the Pod Identity  
16 spec:  
17     containers:  
18         - name: app  
19             image: spbootacr2021am.azurecr.io/spring-boot-aks:v1.0.3  #Replace this by your image  
20         ports:  
21             - containerPort: 8080  
22             imagePullPolicy: Always
```

updated-spring-boot-deployment.yaml hosted with ❤ by GitHub

[view raw](#)

When you try & deploy this to the AKS cluster, it is able to fetch the secrets from Keyvault successfully but is not able to connect to the SQL database (duh!) and fails with the following error:

```
Caused by: com.microsoft.sqlserver.jdbc.SQLServerException: Cannot open server ' [REDACTED]' requested by the login. Client with IP address ' [REDACTED]' is not allowed to access the server. To enable access, use the Windows Azure Management Portal or run sp_set_firewall_rule on the master database to create a firewall rule for this IP address or address range. It may take up to five minutes for this change to take effect.
at com.microsoft.sqlserver.jdbc.SQLServerException.makerUserError(SQLServerException.java:203) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.TDSCommand.setFirewallRule(TDSCommand.java:108) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.TDSParser.parseSetFirewallRule(TDSParser.java:270) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.TDSParser.parseSetFirewallRule(TDSParser.java:270) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.SQLServerConnection.setFirewallRule(SQLServerConnection.java:322) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.SQLServerConnection.access$800(SQLServerConnection.java:85) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.SQLServerConnection$ResponseMonitor.detectSetFirewallRule(SQLServerConnection.java:322) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.TDSCommand.execute(TDSCommand.java:378) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.SQLServerConnection$ResponseMonitor.detectExecute(SQLServerConnection.java:326) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.SQLServerConnection$ResponseMonitor.detectInternal(SQLServerConnection.java:2233) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.SQLServerConnection.connect(SQLServerConnection.java:3276) -[msql-jdbc-8.4.1.jar!/na]
at com.microsoft.sqlserver.jdbc.SQLServerConnection.connect(SQLServerDriver.java:881) -[msql-jdbc-8.4.1.jar!/na]
at com.zaxxer.hikari.HikariUtil.getConnectionFromDataSource(com.zaxxer.hikari.util.HikariUtil.java:380) -[HikariCP-3.4.6.jar!/na]
at com.zaxxer.hikari.PoolBase.newDataSource(PoolBase.java:390) -[HikariCP-3.4.6.jar!/na]
at com.zaxxer.hikari.HikariPool.newDataSource(HikariPool.java:187) -[HikariCP-3.4.6.jar!/na]
at com.zaxxer.hikari.HikariPool.checkFailFast(HikariPool.java:69) -[HikariCP-3.4.6.jar!/na]
at com.zaxxer.hikari.HikariPool.init(HikariPool.java:119) -[HikariCP-3.4.6.jar!/na]
at com.zaxxer.hikari.HikariDataSource.getDataSource(HikariDataSource.java:157) -[HikariCP-3.4.6.jar!/na]
at org.springframework.jdbc.datasource.DataSourceUtils.fetchConnection(DataSourceUtils.java:394) -[Spring-JDBC-5.0.6.jar!/na]
at org.springframework.jdbc.datasource.DataSourceUtils.doGetConnection(DataSourceUtils.java:380) -[Spring-JDBC-5.0.6.jar!/na]
at org.springframework.jdbc.datasource.DataSourceUtils.getConnection(DataSourceUtils.java:792) -[Spring-JDBC-5.0.6.jar!/na]
... 15 common frames omitted
```

## Connection Error !

Don't worry we solve this in Step 5 below.

## Step 5: Virtual Network Service Endpoint & Firewall Rule for SQL Server

For pods running on AKS cluster to be able to access our SQL Server we need to whitelist the AKS subnet for the SQL Server by creating a Service Endpoint for the subnet hosting the AKS cluster. So we make the following changes in our Terraform template

```
1 resource "azurerm_subnet" "akssubnet" {
2     ....
3
4     service_endpoints = [
5         "Microsoft.Sql"
6     ]
7 }
```

```

9 resource "azurerm_sql_virtual_network_rule" "aksvnrule" {
10   name           = "aks-vnet-sql-rule"
11   resource_group_name = azurerm_resource_group.core.name
12   server_name     = azurerm_sql_server.sqlserver.name
13   subnet_id       = azurerm_subnet.akssubnet.id
14 }
15

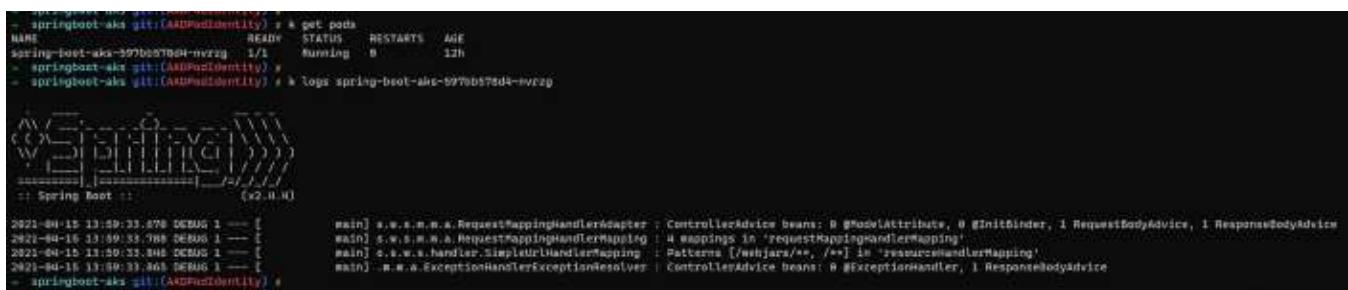
```

[updateserviceendpoitns.tf](#) hosted with ❤ by GitHub

[view raw](#)

### Terraform Code Snippet for VNet Rule for whitelisting AKS Subnet

Apply these changes via `terraform plan` & then `terraform apply` & voila, we are done !  
 Check the pods logs — they are perfectly fine and the Spring App is running without any errors



```

springboot-aks get:(AKSPodIdentity) ✘ * get pods
NAME          READY   STATUS    RESTARTS   AGE
spring-boot-aks-597b8704-mvzg  1/1   Running   0          12h
springboot-aks get:(AKSPodIdentity) ✘ *
springboot-aks get:(AKSPodIdentity) ✘ * logs spring-boot-aks-597b8704-mvzg
```
  (A small ASCII-art tree diagram representing the Spring Boot application structure is shown here)
  Spring Root :: (x2-h-i)
  2021-04-16 13:59:33.870 DEBUG 1 --- [main] o.a.w.s.m.a.RequestMappingHandlerAdapter : ControllerAdvice beans: 0 @ModelAttribute, 0 @InitBinder, 1 RequestBodyAdvice, 1 ResponseBodyAdvice
  2021-04-16 13:59:33.870 DEBUG 1 --- [main] o.a.w.s.m.a.RequestMappingHandlerMapping : 4 Mappings in 'RequestMappingHandlerMapping'.
  2021-04-16 13:59:33.870 DEBUG 1 --- [main] o.a.w.s.h.SimpleUrlHandlerMapping : Patterns [/webjars/**, /**] in 'SimpleUrlHandlerMapping'
  2021-04-16 13:59:33.870 DEBUG 1 --- [main] o.a.w.s.ExceptionHandlerExceptionResolver : ControllerAdvice beans: 0 @ExceptionHandler, 1 ResponseBodyAdvice
  ```

  - springboot-aks get:(AKSPodIdentity) ✘ *

```

Let's deploy a simple Kubernetes Service and access our Todos REST API

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: spboot-svc
5 spec:
6   type: LoadBalancer
7   selector:
8     app: spring-boot-aks
9   ports:
10    - protocol: TCP
11      port: 80
12      targetPort: 8080

```

[springbootservice.yaml](#) hosted with ❤ by GitHub

[view raw](#)

Lets create this `Service` to access the API

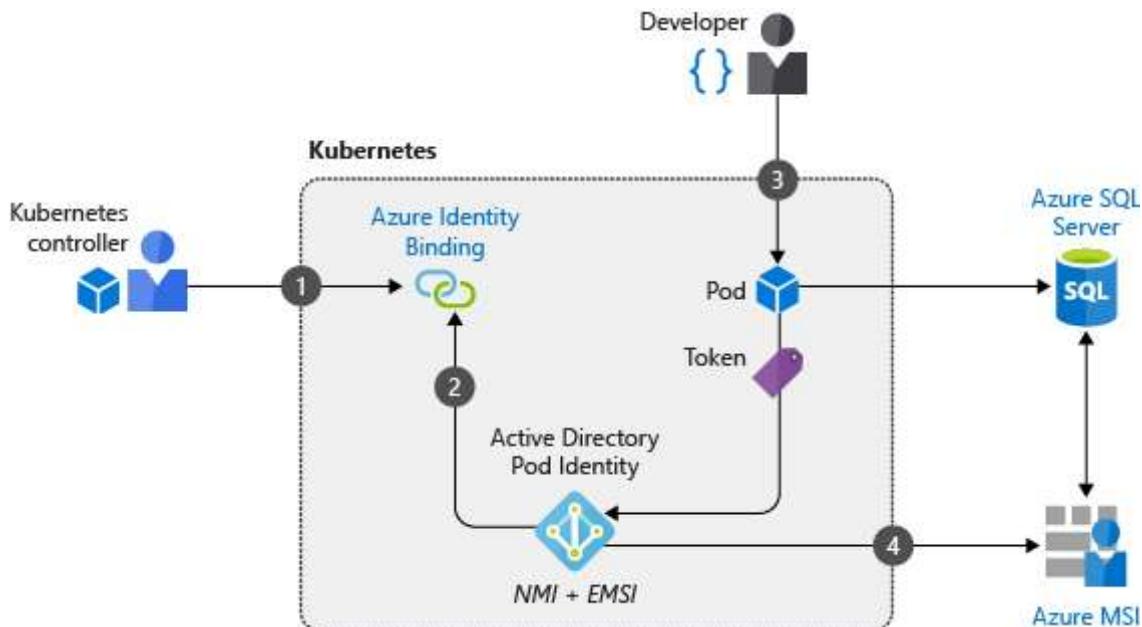
`kubectl apply -f deploy/spring-boot-app-deployment.yaml`

```
→ springboot-aks git:(AADPodIdentity) kubectl get service -n spboot-app
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
spboot-svc  LoadBalancer  10.0.3.30    20.193.42.211  80:30219/TCP  25m
→ springboot-aks git:(AADPodIdentity)
→ springboot-aks git:(AADPodIdentity) curl http://20.193.42.211/todos
[]%
→ springboot-aks git:(AADPodIdentity)
```

Cool so we now have a working Spring Boot Application in AKS using AAD Pod Managed Identities to access Azure Resources like Key Vault. All the code changes done so far can be found in [aadpodidentity branch of our repo](#).

## The Big Picture

Before we close off this chapter it may be a good idea to take a look at the bigger picture of how it all works in AKS.



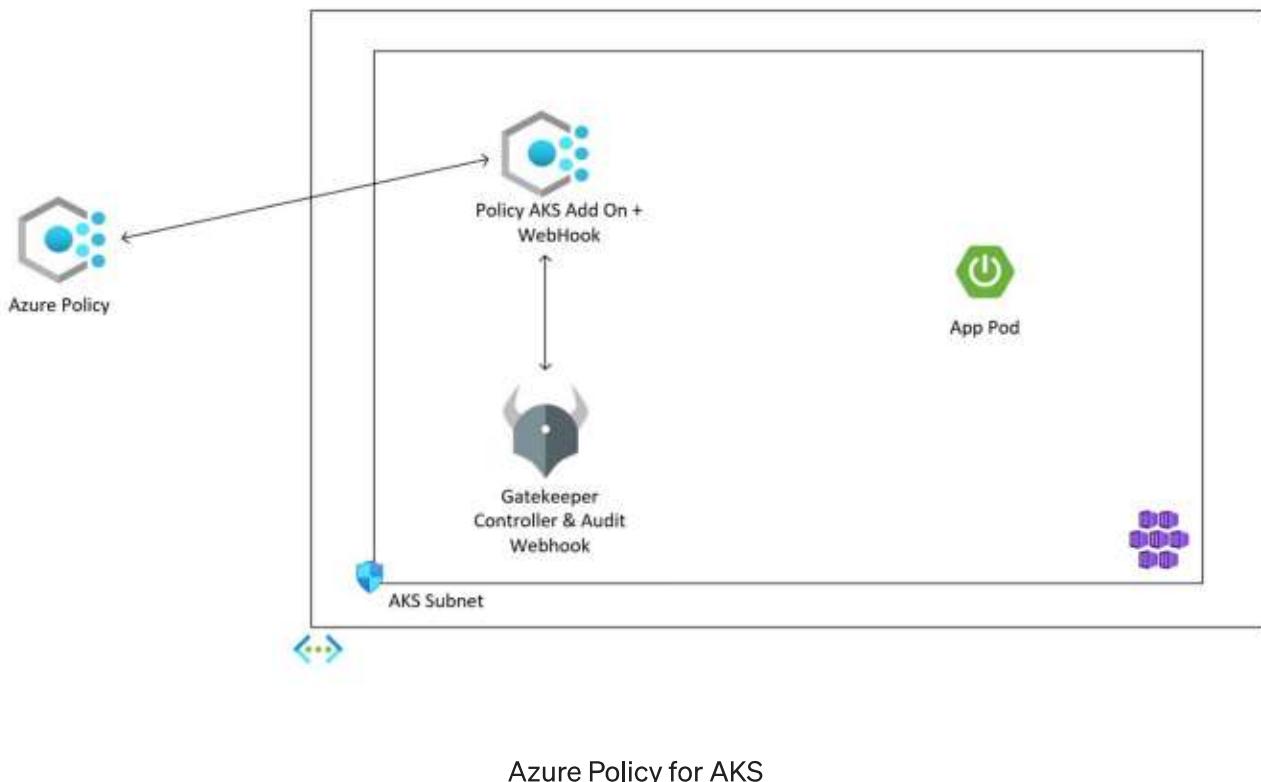
The Big Picture: Azure AD Pod Managed Identity

- Once you enable the Pod Identity on the AKS cluster, the **Node Managed Identity (NMI) server** runs as a DaemonSet on each node on the cluster which intercepts calls to Azure services.
- The NMI server talks to **Azure Resource Provider** to find the Azure Managed Identity associated with the Pod and then use that to get the access token from the AAD.

If your configuration for the AAD Pod Identity is not working, checking logs for the NMI server may help. You can read about the above authentication flow in more details on the official documentation [here](#).

## Azure Policy for AKS

Azure Policy should be your first stop to enforce organizational standards & to assess compliance at scale for any Azure resource & AKS is no different. The AKS add-on for Azure Policy is based on [Gatekeeper v3](#) which uses [Open Policy Agent](#) CRD-based policies to audit & enforce governance. Gatekeeper v3 uses a [validating admission controller](#) & [webhook](#)(with mutating ones currently in development) with CRDs such as `ConstraintTemplate` & `Constraint`. These CRDs uses a high level declarative language `Rego` (pronounced *ray-go*) to specify the actual policy that is evaluated by the Open Policy Agent. The idea is to validate the policy and reject the user request if a policy violation is found before persisting the object. Now Azure Policy has two different modes in which you can work — `Audit` and `Deny`. So even with policy violations you can let the invalid user requests pass through and just log the policy violation for later use.



Azure Policy for AKS

Azure Policy provides a lot of built-in policies tailored for Kubernetes deployments that you can find via the Portal

The screenshot shows the Azure Policy Definitions blade. At the top, there are buttons for '+ Policy definition', '+ Initiative definition', 'Export definitions', and 'Refresh'. Below this is a search bar and a filter section with dropdowns for 'Scope' (set to 'Cluster'), 'Definition type' (set to 'Policy'), 'Type' (set to 'All types'), 'Category' (set to '1 categories'), and a 'Search' bar. A note at the top says 'Now import your definitions and assignments to GitHub and manage them using az cli! Click on 'Export definition' menu action learn more here'. The main area displays a table of policy definitions:

Name	Definition type	Policies	Type	Definition type	Category
Azure Kubernetes Service Private Clusters should be enabled	Built-in	Policy	Kubernetes		
Azure Policy Add-on for Kubernetes service (AKS) should be installed and enabled on your clusters	Built-in	Policy	Kubernetes		
Configure Kubernetes clusters with specified GKE configuration using no secrets	Built-in	Policy	Kubernetes		
Temp disks and cache for agent node pools in Azure Kubernetes Service clusters should be encrypted at rest	Built-in	Policy	Kubernetes		
Deploy - Configure diagnostic settings for Azure Kubernetes Service to Log Analytics workspace	Built-in	Policy	Kubernetes		
Both operating systems and data disks in Azure Kubernetes Service clusters should be encrypted by customer-managed keys	Built-in	Policy	Kubernetes		
[Preview] Azure Arc-enabled Kubernetes clusters should have Azure Defender's extension installed	Built-in	Policy	Kubernetes		
Configure Kubernetes clusters with specified GKE configuration using HTTPS secrets	Built-in	Policy	Kubernetes		
Deploy Azure Policy Add-on to Azure Kubernetes Service clusters	Built-in	Policy	Kubernetes		
Configure Kubernetes clusters with specified GKE configuration using SSH secrets	Built-in	Policy	Kubernetes		
Kubernetes cluster pod hostPath volumes should only use allowed host paths	Built-in	Policy	Kubernetes		
Kubernetes cluster pods should only use allowed volume types	Built-in	Policy	Kubernetes		
Kubernetes clusters should be accessible only over HTTPS	Built-in	Policy	Kubernetes		
Kubernetes clusters should not allow container privilege escalation	Built-in	Policy	Kubernetes		
Kubernetes cluster services should listen only on allowed ports	Built-in	Policy	Kubernetes		
Kubernetes clusters should use internal load balancers	Built-in	Policy	Kubernetes		

## Built-in Azure Policies for AKS

Some of these are defined using standard Azure Policy language and using known constructs. But most of them rely on `ConstraintTemplate` and `Constraint Custom Resources`. As an example, checking the Policy definition for one of the built-in policy for “Kubernetes cluster should not allow privileged containers” gives

### Kubernetes cluster should not allow privileged containers

The screenshot shows the Azure Policy Definition blade for the policy 'Kubernetes cluster should not allow privileged containers'. At the top, there are buttons for 'Assign', 'Edit definition', 'Duplicate definition', 'Delete definition', and 'Export definition'. The main area displays the JSON policy definition:

```

{
    "policyRule": {
        "if": {
            "field": "type",
            "in": [
                "AKS Engine",
                "Microsoft.Kubernetes/connectedClusters",
                "Microsoft.ContainerService/managedClusters"
            ]
        },
        "then": {
            "effect": "[parameters('effect')]",
            "details": {
                "constraintTemplate": "https://store.policy.core.windows.net/kubernetes/container-no-privilege/v1/template.yaml",
                "constraint": "https://store.policy.core.windows.net/kubernetes/container-no-privilege/v1/constraint.yaml",
                "excludedNamespaces": "[parameters('excludedNamespaces')]",
                "namespaces": "[parameters('namespaces')]",
                "labelSelector": "[parameters('labelSelector')]"
            }
        }
    }
}

```

A red box highlights the 'constraintTemplate' and 'constraint' fields.

```

123
124
125
126 "id": "/providers/Microsoft.Authorization/policyDefinitions/95edb821-ddaf-4404-9732-666845e056b4",
127 "type": "Microsoft.Authorization/policyDefinitions",
128 "name": "95edb821-ddaf-4404-9732-666845e056b4"
129

```

The **ConstraintTemplate** custom resource referenced above looks like

```

1  apiVersion: templates.gatekeeper.sh/v1beta1
2  kind: ConstraintTemplate
3  metadata:
4    name: k8sazurecontainernoprivilege
5  spec:
6    crd:
7      spec:
8        names:
9          kind: K8sAzureContainerNoPrivilege
10         listKind: K8sAzureContainerNoPrivilegeList
11         plural: k8sazurecontainernoprivilege
12         singular: k8sazurecontainernoprivilege
13     targets:
14       - target: admission.k8s.gatekeeper.sh
15         rego: |
16           package k8sazurecontainernoprivilege
17
18           violation[{"msg": msg, "details": {}}] {
19             c := input_containers[_]
20             c.securityContext.privileged
21             msg := sprintf("Privileged container is not allowed: %v, securityContext: %v", [c.r
22           }
23
24           input_containers[c] {
25             c := input.review.object.spec.containers[_]
26           }
27
28           input_containers[c] {
29             c := input.review.object.spec.initContainers[_]
30           }

```

constrainttemplate.yaml hosted with ❤ by GitHub

[view raw](#)

ConstraintTemplate — Actual Policy Defined using [Rego](#)

The **ConstraintTemplate** defines the `Rego` logic that defines the policy to be applied, as well as the schema of the CRD and the parameters that can be passed to the **Constraint**, which in this case looks like

```

1 apiVersion: constraints.gatekeeper.sh/v1beta1
2 kind: K8sAzureContainerNoPrivilege
3 metadata:
4   name: container-no-privilege
5 spec:
6   match:
7     kinds:
8       - apiGroups: []
9         kinds: ["Pod"]

```

constraint.yaml hosted with ❤ by GitHub

[view raw](#)

Essentially you can create one or more Constraints based on the same ConstraintTemplate

Once you assign this policy to your subscription or the resource group containing the AKS Cluster, **it takes around 30 minutes for the effect to take place** before you can enforce the constraints or can see the associated CRDs for `Constraint` and `ConstraintTemplate` for this policy in the AKS Cluster. Once the time has passed, you would be able to see new custom resources for this policy

```

→ playingwithpolicies git:(master) ✘ k get crd
NAME                                         CREATED AT
azureidentities.aadpodidentity.k8s.io        2021-04-15T05:42:40Z
azureidentitybindings.aadpodidentity.k8s.io    2021-04-15T05:42:40Z
azurepodidentityexceptions.aadpodidentity.k8s.io 2021-04-15T05:42:40Z
configs.config.gatekeeper.sh                  2021-04-15T04:44:31Z
constraintpodstatuses.status.gatekeeper.sh    2021-04-15T04:44:31Z
constrainttemplatetemplatestatus.status.gatekeeper.sh 2021-04-15T04:44:31Z
constrainttemplates.templates.gatekeeper.sh    2021-04-15T04:44:31Z
healthstates.azmon.container.insights        2021-04-15T04:44:31Z
k8sazurecontainernoprivilege.constraints.gatekeeper.sh 2021-04-19T13:33:55Z
k8sazurepodenforcelabels.constraints.gatekeeper.sh 2021-04-15T05:03:56Z
→ playingwithpolicies git:(master) ✘

```

CRD for the No Privileged Container Policy Constraint

```

→ playingwithpolicies git:(master) ✘
→ playingwithpolicies git:(master) ✘ k get constrainttemplates
NAME          AGE
k8sazurecontainernoprivilege  30m
k8sazurepodenforcelabels    4d9h
→ playingwithpolicies git:(master) ✘
→ playingwithpolicies git:(master) ✘
→ playingwithpolicies git:(master) ✘ k get k8sazurecontainernoprivilege
NAME          AGE
azurepolicy-container-no-privilege-3cee9d65e76ce46d53fb  29m
→ playingwithpolicies git:(master) ✘

```

By default the compliance is checked every 15 minutes by the add-on for Azure Policy & reported back to Azure Policy control plane and is also visible in Azure Portal. Once you can see the right set of CRDs in your cluster, time to test the policy — lets create a privileged pod

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-privileged
5  spec:
6    containers:
7      - name: nginx-privileged
8        image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine
9        securityContext:
10          privileged: true

```

[privileged-pod.yaml](#) hosted with ❤ by GitHub

[view raw](#)

Denied !

```

→ playingwithpolicies git:(master) ✘ kubectl apply -f privileged-pod.yaml
Error from server ([denied by azurepolicy-container-no-privilege-3cee9d65e76ce46d53fb] Privileged container is not allowed: nginx-privileged, securityContext: {"privileged": true}): error when creating "privileged-pod.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [denied by azurepolicy-container-no-privilege-3cee9d65e76ce46d53fb] Privileged container is not allowed: nginx-privileged, securityContext: {"privileged": true}
→ playingwithpolicies git:(master) ✘

```

Also note that our Spring Boot Application is compliant with this policy (since it does not uses privileged containers) Any Application pods that are already deployed when this policy was applied to your cluster are not removed but are reported in the compliance status of Azure Policy — thus you can still remediate any effects of previous deployments against your organizational standards. To complete our testing, let's try and deploy a non-privileged pod running nginx — looks like

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-unprivileged
5  spec:
6    containers:
7      - name: nginx-unprivileged
8        image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine

```

[unprivileged-pod.yaml](#) hosted with ❤ by GitHub

[view raw](#)

Apply this and boom it works !

```
→ playingwithpolicies git:(master) ✘ kubectl apply -f unprivileged-pod.yaml
pod/nginx-unprivileged created
→ playingwithpolicies git:(master) ✘ kubectl get po
NAME           READY   STATUS    RESTARTS   AGE
nginx-unprivileged   1/1     Running   0          8s
→ playingwithpolicies git:(master) ✘
```

Currently the only downside of Azure Policy over using Vanilla Gatekeeper is the limitation of not being able to write your own custom policy with a custom Rego

## Cluster Control Plane IP Whitelisting

Since we are not using a Private AKS cluster — the next best thing you could do to secure your API Server (arguably the most critical component of your Control Plane) and the one that has a public endpoint today — is to whitelist the IP ranges from which the API server can be reached. This is mostly the place from where cluster administrators and/or users try to access the cluster but also take into account the IP address range for your DevOps agents (hosted or otherwise). Unfortunately the AKS module I'm using doesn't offer this yet — to whitelist the [API Server authorized IP ranges](#) but one can always use the local provisioner & Azure CLI to update the AKS cluster

```
az aks update -g $AKS_RESOURCE_GROUP -n $AKS_CLUSTER_NAME --api-server-authorized-ip-ranges $IP_ADDRESS_RANGE
```

The article is already too long so we'll skip the rest ( Private Link, Network Policies, WAF etc) for another time. Till then, enjoy hacking !

Kubernetes    Security    Azure    Managed Identity    Azure Policy

About    Help    Legal

Get the Medium app



