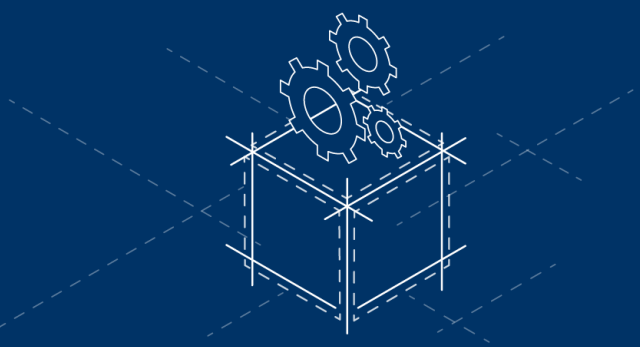
Cloudockit

Container

AWS - INSTALLATION & CONFIGURATION GUIDE



GENERATED BY: Cloudockit

Contents

Introduction

The purpose of this document is to provide the detailed steps to run and configure Cloudockit Docker container image.

There are two types of images that you should run:

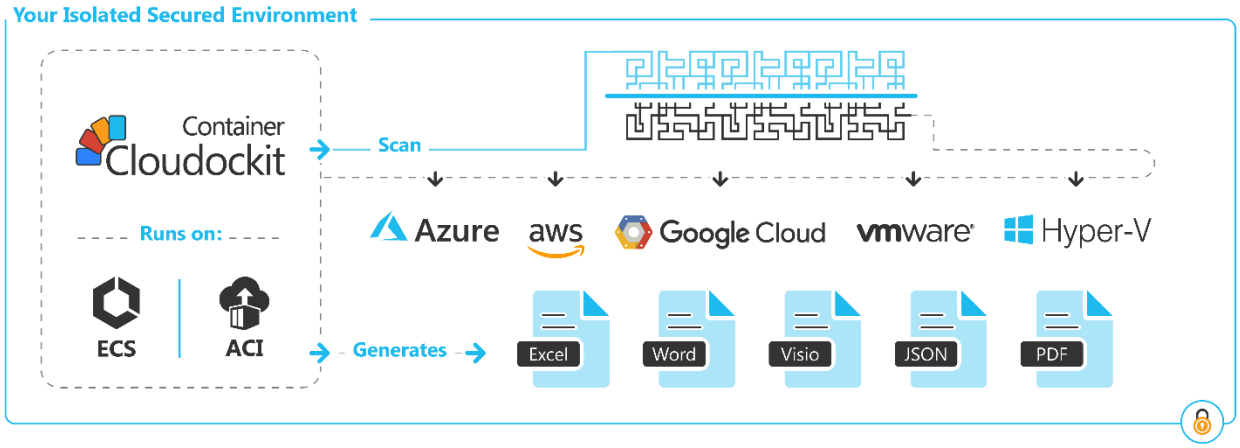
- **cdk-web-linux** that contain the Cloudockit API/Web interface. This is mandatory to run this container.
- **cdk-scheduler-linux** that contain the Cloudockit Scheduling features. This is an optional container you do not need to install if you do not want to use schedules.

The cdk-web image contains the Cloudockit API that you can call from your CI/CD processes or any other process / scenario which fits your business needs.

In addition to the API, we have integrated the complete Cloudockit Web UI in the image so that you can get all the features that you are accustomed to.

Cloudockit Docker container images provide you a way to run Cloudockit into your own isolated Cloud environment and gives you the exact same features as Cloudockit Website and Cloudockit Desktop.

Here is the high-level overview of the solution :



The following hosting environments are currently supported:

- Web App for Containers on Azure - Recommended
- ECS (Elastic Container Services) Fargate on AWS
- ACI (Azure Container Instance) on Azure
- GKE (Google Kubernetes Engine) on GCP

A few important things to note:

- These configurations are for the hosting of the container, not for the environment that you scan which means that you can scan a GCP project using the Cloudockit Container API even if the container runs on Azure.

- Depending on the hosting option that you choose, there could be some limitations. Those limitations are related to the hosting option and not the Cloudockit Container itself. As an example, ACI currently does not yet support private networking (virtual networks) for Windows Based Container.
- The current document does not detail networking configuration like isolation/https setup as this is highly depending on your internal setup.
- Container is currently designed to have one node running which should be more than enough to generate all your documents you need.
- For production environment, we recommend 4vCPU + 8 Gb RAM
- Cloudockit Web UI only supports Azure AD as SSO authentication. If you do not set it up, you will only be able to access the API portion.

The following sections contains the different steps to deploy the Cloudockit Docker container image on the AWS Platform.

Here is an overview of the different steps you must do to deploy Cloudockit Container:

Step A - Deploy Cloudockit Container (cdk-web)

- This Step is subdivided in 8 Different steps
- Those steps have been scripted to ease deployment process so we strongly advice to use the scripted approach

Step B (Optional) - Activate Cloudockit Container UI

- Create an Azure AD Application

Step C (Optional) - Activate the Scheduling feature (cdk-scheduler)

- Set the appropriate settings to activate scheduling

Step D ... - Do some tests

- Test the license validity
- Start some documentation

Requirements

To install Cloudockit Container in your environment, you will need:

- A S3 Bucket
- A Secret Manager to store the secret to pull the image from Cloudockit Repo.
- A Role / Policy to access the Secret.
- A VPC/Subnet to host the ECS Cluster.
- A ECS Cluster (Fargate)
    - Note that this is the recommended approach as this is the one we have scripted but it does not mean that we do not support other types of deployment like AKS or other as Cloudockit Container is a standard plain container.
- (Optional) An Azure Active Directory Application if you want to activate Cloudockit Container Web UI

Important note

We highly recommend that you use the script (based on AWS CLI) provided by Cloudockit Team to provision/test Cloudockit Container.

Step A-ECS – Provision Cloudockit Container (hosted on ECS/Fargate)

STEP 0 – Create a S3 Bucket and upload the license

To run Cloudockit Container, you need to have a S3 Bucket that will be used to store information (license file, settings…). As the license file is linked to this S3 bucket, you need to choose a container Name and send that name to Cloudockit Support Team (support@cloudockit.com) so that they generate a license file.

Once you receive your license file, you can upload the license file into a file named ***cloudockitinternal/license.json***.

Please note that the Json license file is tied to the S3 Bucket name so you need to create/use an S3 Bucket with a name that matches the name provided to Cloudockit Support Team.

Important note

Ensure that the Storage Account exists in your environment before sending it to the Cloudockit Support Team.

Code extract

Refer to STEP 0 in the provided script for **ECS Fargate**.

STEP 1 – Create an AWS Secret in SECRET MANAGER to store information to connect to Cloudockit Container Repository

In this step, you need to create a Secret that will contain the information to connect to Cloudockit Container Registry so that your ECS can pull the image.

Secret should contain *username* and *password* properties, see code for reference.

Code extract

Refer to STEP 1 in the provided script for **ECS Fargate**.

STEP 2 - Creating Policy to access the Secret

Once you have created the secret, you need to create a policy that will allow access to the secret.

The policy needs to provide **Allow Effect** to the action **secretsmanager:GetSecretValue** to the secret previously created.

Code extract

Refer to STEP 2 in the provided script for **ECS Fargate**.

STEP 3 - Creating Role assumed by ECS tasks to access the Secret

Once the policy has been created, we need to create a Role attached to that policy so that ECS assumes the role to be able to pull the remote image.

The role needs to have the **Allow Effect** and needs to have the action **sts:AssumeRole** for the service *ecs-tasks.amazonaws.com*

Code extract

Refer to STEP 3 in the provided script for **ECS Fargate**.

STEP 4 - Creating VPC and Subnet + Open required ports

In this step, we need to configure a VPC, a Subnet, and a Route Table so that the subnet is reachable and so that the subnet has connectivity to the Internet to pull the image. Please note that you can use an existing VPC if it has the appropriate requirements.

Code extract

Refer to STEP 4 in the provided script for **ECS Fargate**.

STEP 5 - Creating Security Group

To Secure the deployment, we create a security group with only port 80 open (please use https/443 for Production usage)

Code extract

Refer to STEP 5 in the provided script for **ECS Fargate**.

STEP 6 - Creating the Task Definition File

Once networking configuration is done, we need to create a task definition file that contains the following information.

Parameters

Name	Value
family	Cloudockit
executionRoleArn	arn:aws:iam::<accntnumber>:role/ecsTaskExecutionRole
networkMode	Awsvpc
requiresCompatibilities	Fargate
ContainerDefinitions	Should refer to the image and the secret to access it

Environment variables

Name	Value
AppInsightKey (optional)	An Azure App Insight Instrumentation Key for advanced login

Name	Value
DockerStorageCloudProvider	Specify if your Storage Account is stored in Azure, AWS or GCP. Possible values are: • Azure • GCP • **AWS** (select this value)
DockerStorageAWSBucketName	Enter the name of the S3 Bucket
DockerStorageAWSAccessKeyId (option 1)	Enter the Access Key Id for Full control of the AWS S3 Bucket
DockerStorageAWSSecretAccessKey (option 1)	Enter the Secret Access Key for Full control of the AWS S3 Bucket
DockerStorageUseAwsRole (option 2)	True (permissions are provided by the ECS that hosts the container)
DockerStorageUseAwsGov (optional)	Specify "True" if you are using **Aws Gov Cloud** or "False" otherwise. If the variable is not present, it's "False" by default

➔ **Note: parameters noted as option 1 and 2 are mutually exclusive**

Code extract

Refer to STEP 6 in the provided script for **ECS Fargate**.

STEP 7 - Creating the Task Definition

Then, from the Task Definition file, we just create a task definition:

Refer to STEP 7 in the provided script for **ECS Fargate**.

STEP 8 - Creating the cluster and register Task Definition.

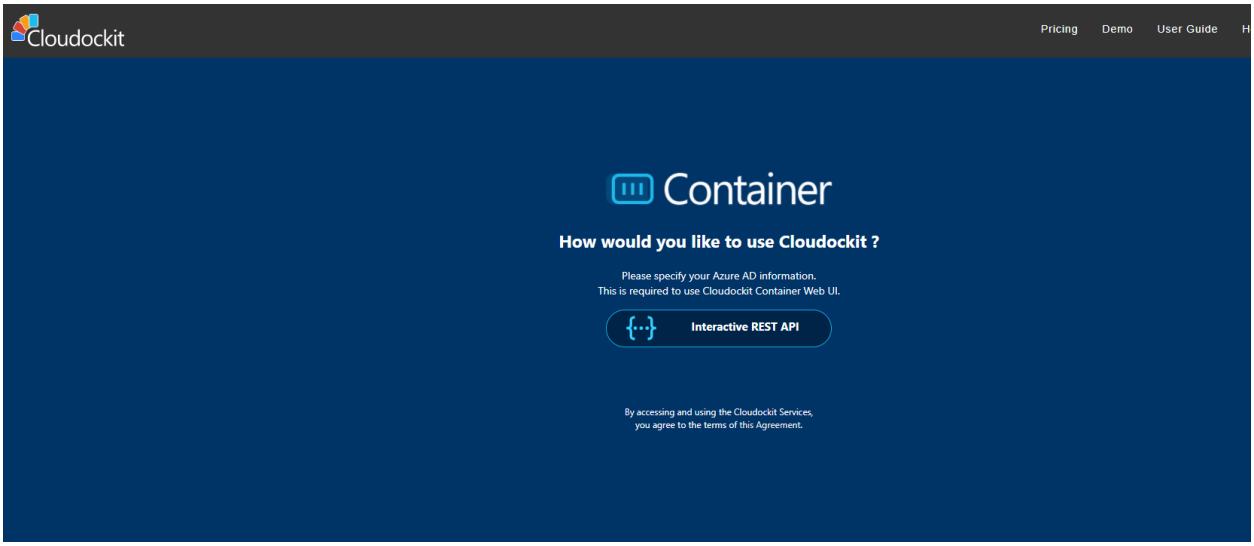
Last step is to create a cluster and start the service:

Refer to STEP 2 in the provided script for **ECS Fargate**.

STEP 9 – Validate Service is Up & Running

From the AWS Console, navigate to the cluster, tasks and then click on the public IP to open it.

You should see Cloudockit Container running:


Cloudockit
Pricing
Demo
User Guide
Container
How would you like to use Cloudockit ?
Please specify your Azure AD information.
This is required to use Cloudockit Container Web UI.
Interactive REST API
By accessing and using the Cloudockit Services,
you agree to the terms of this Agreement.

Step B (Optional) – Configure Cloudockit Web UI

Cloudockit Container supports a Web UI that allows users to authenticate by using Azure AD or Azure User Authentication.

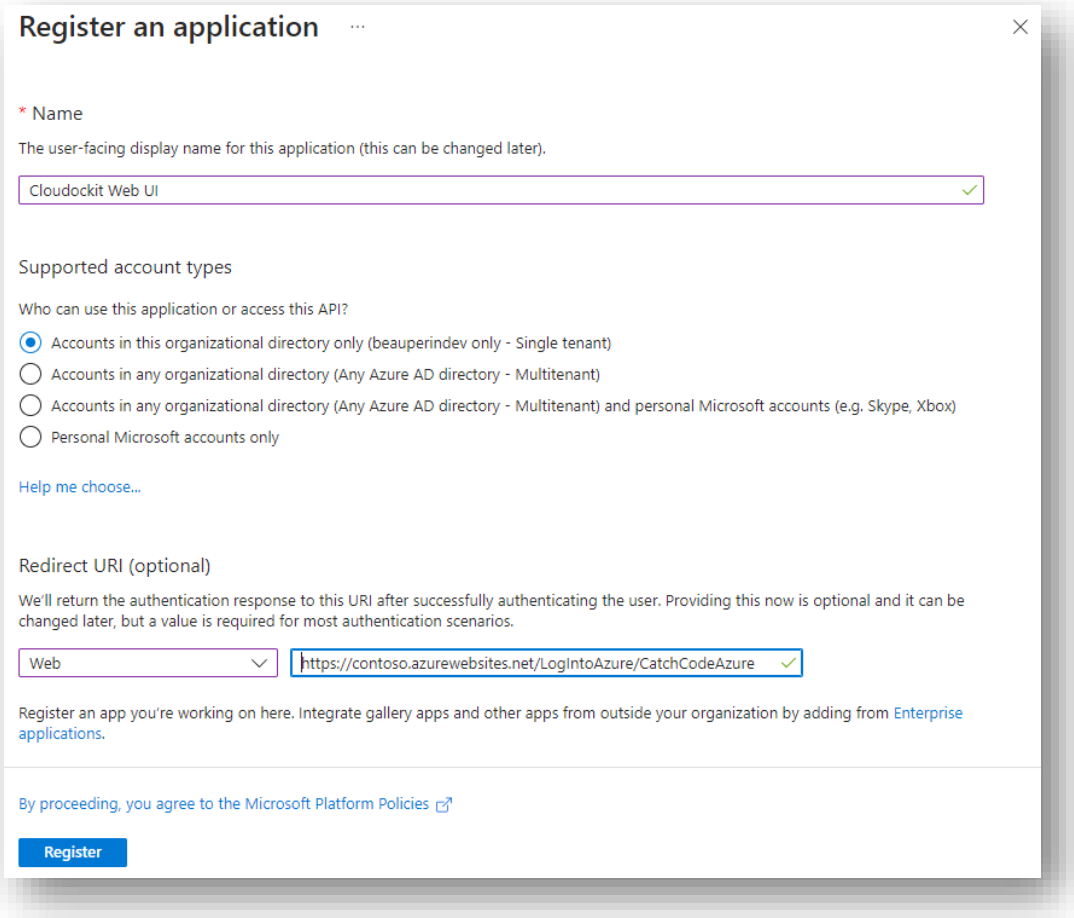
This Web UI supports Azure Active Directory as a first step to authenticate users.

Once connected, you will be able to connect to Azure, AWS and GCP using Service Accounts (Azure AD App, GCP Service Credentials, AWS Access Keys).

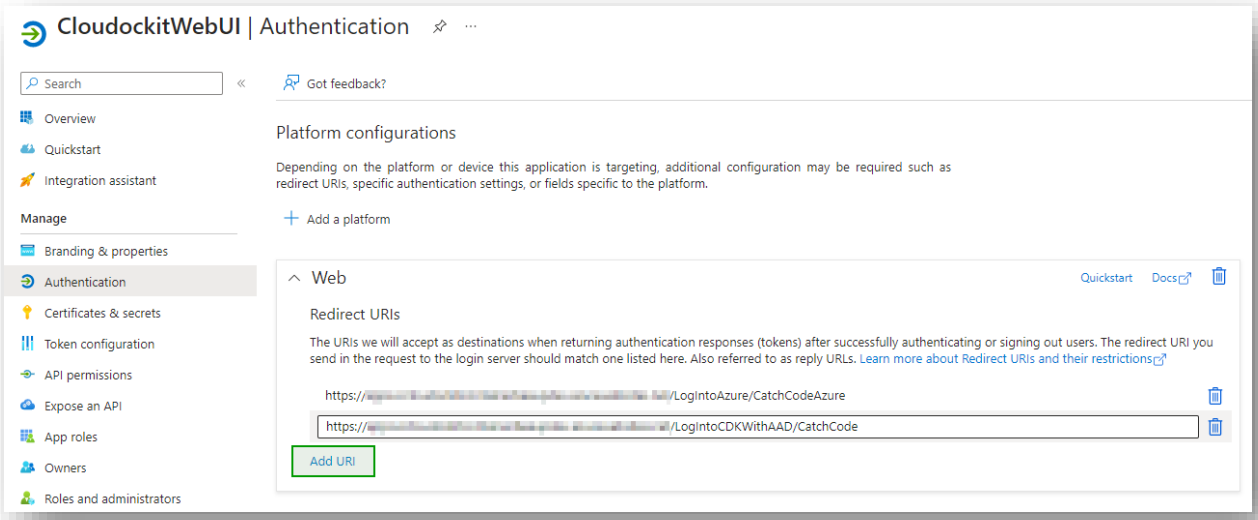
To activate Azure AD Authentication, you need to follow these steps:

- Go to your **Azure Active Directory**
- Click on **App Registration** and then click **New Registration**
- Enter a **Name** (any name you want) and select **Single Tenant**
- Enter the following redirect URIs (reply url):
    - `https://<AppSvcName>.azurewebsites.net/LogIntoAzure/CatchCodeAzure`
    - `https://<AppSvcName>.azurewebsites.net/LogIntoCDKWithAAD/CatchCode`

      where *AppSvcName* is the name of your App Service

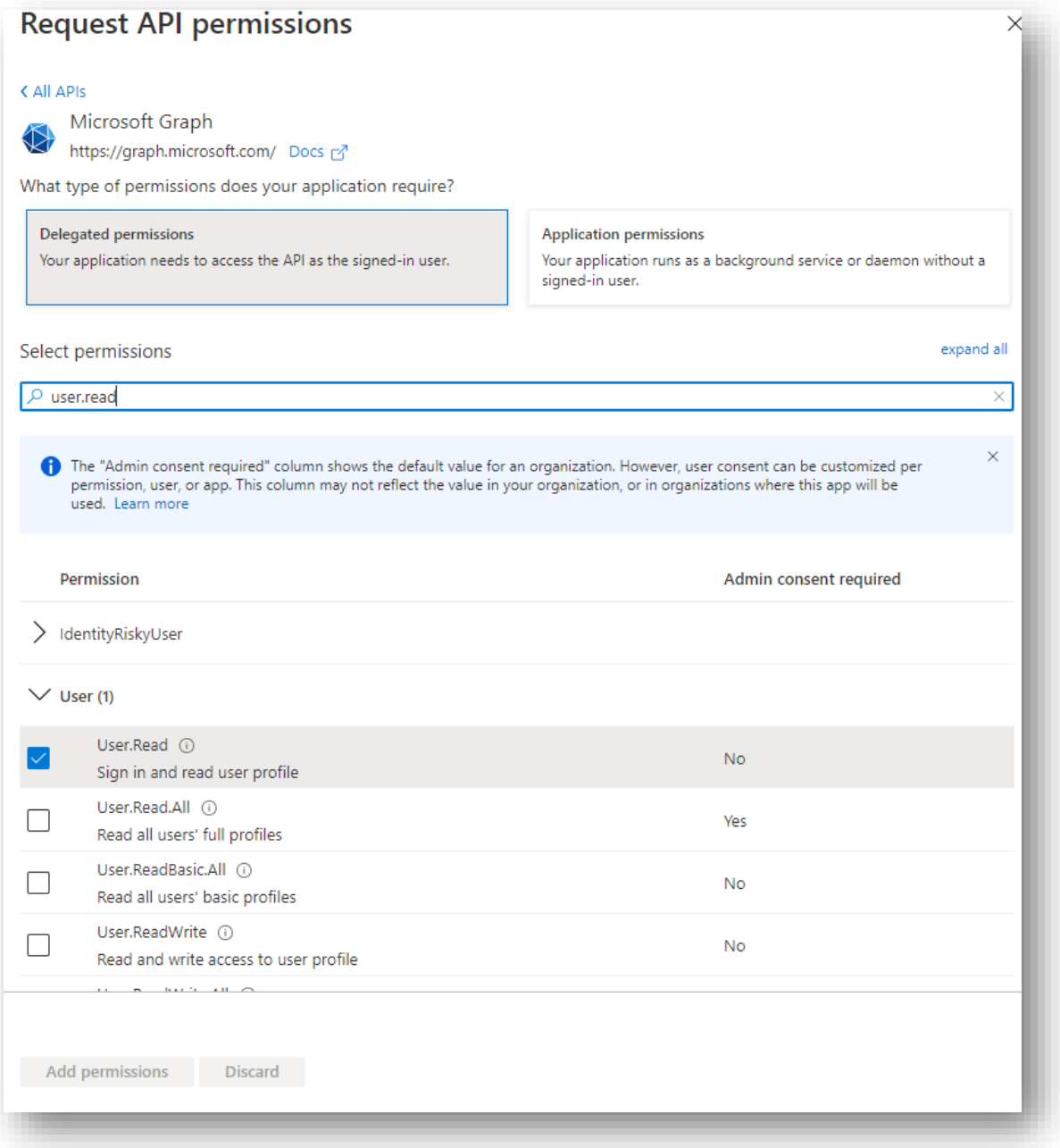


Note: the interface will not let you enter the 2nd URL before clicking on **Register** so you'll have to enter it after registration, in the Authentication page:

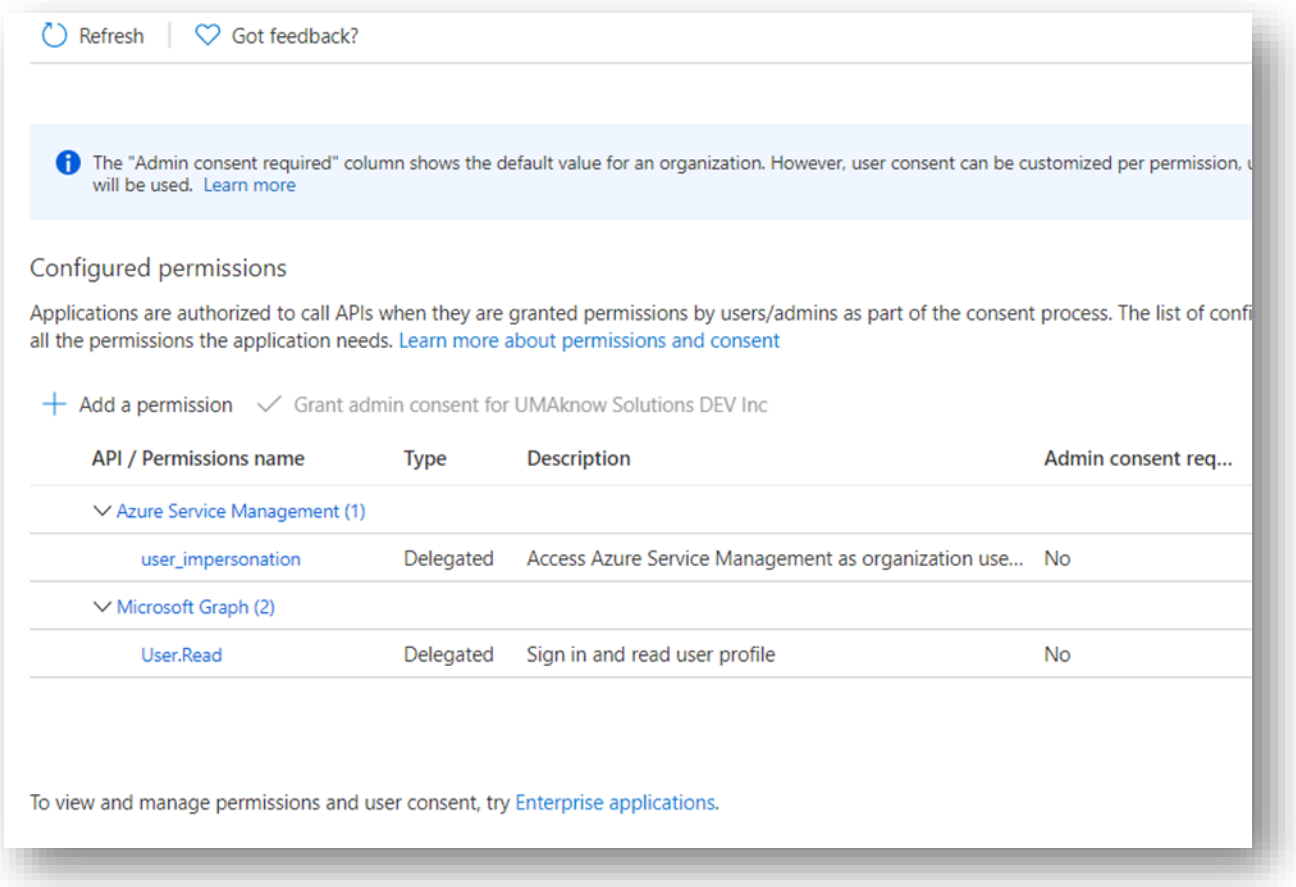


Then, go to **API Permissions**, click on **+Add a permission** and select :

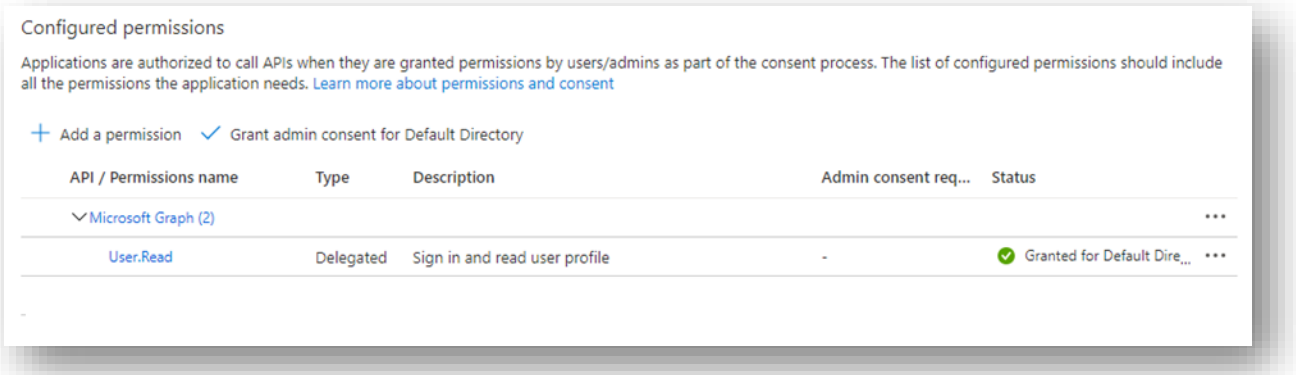
- **Microsoft Graph**, then **Delegated permissions** and then select ***User.Read***:
- **Azure Service Management**, then **Delegated permissions** and then select ***user_impersonation***:



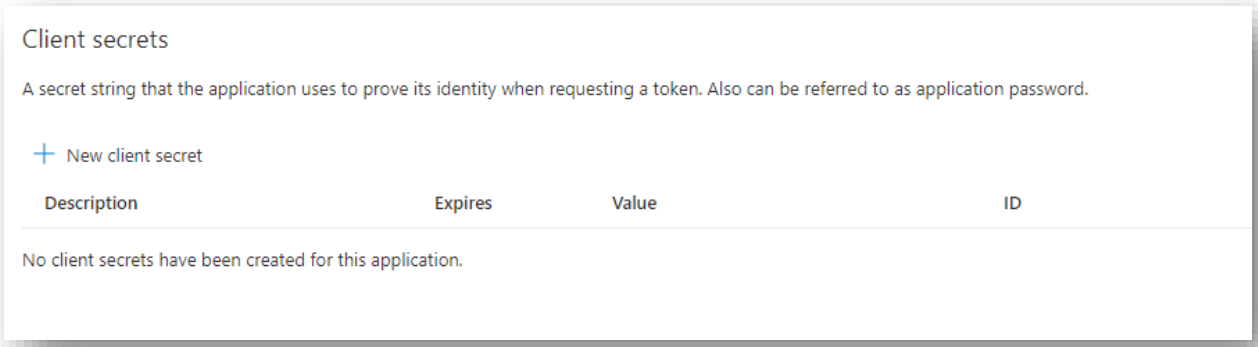
Click **Add permissions**. You should now see the following :



Then, click on **Grant Admin consent** for Default Directory (if you don't have the permissions to click on **Grant admin consent**, please contact your IT admin to do it for you):



Then, take note of the client ID from the Overview tab and then go to **Certificates & Secrets** and generate a new Client Secret, take note of it.



Update the settings file from your storage account (in the cloudockitinternal folder) with the value of the previously created Azure AD Application:

```
{
"AzureADTenant": "mytenant.onmicrosoft.com",
"AzureADAppID": "zzzzz",
"AzureADAppKey": "zzzzz"
}
```

Step C (Optional) – Configure Cloudockit Container to support Scheduling.

Cloudockit Container supports a Scheduling Web UI that allows users to choose when they want to schedule the document generation.

To activate scheduling, you need to spin-up a new container based on the **cloudockitscheduler** image and set the appropriate settings in your settings file.

Start Cloudockit Scheduler Container

You need to follow the same procedure as you did in the previous step to spin up a new Scheduling container. You need to use the **cloudockitscheduler** image. This scheduler is basically reading the schedules files created from the UI and calling the API according to the schedule.

Here are the settings for the container:

- CPU : 1+
- RAM : 1.5GB+
- No inbound networking is required
- Outbound networking needs access to the storage account where the settings are stored and the API URL where Cloudockit is deployed.
- The following 3 environment variables are required:

Name	Value
DockerStorageCloudProvider	Specify if your Storage Account is stored in Azure, AWS or GCP. Possible values are: • Azure • GCP • **AWS**
DockerStorageAWSBucketName	Enter the name of the Bucket
DockerStorageAWSAccessKeyId	Enter the Access Key Id for Full control of the AWS S3 Bucket
DockerStorageAWSSecretAccessKey	Enter the Secret Access Key for Full control of the AWS S3 Bucket
DockerStorageUseAwsGov (optional)	Specify "True" if you are using **Aws Gov Cloud** or "False" otherwise. If the variable is not present, it's "False" by default

Set Settings in the settings file

To activate the scheduling, you need to update the settings file from your storage account (in the *cloudockitinternal* folder) to specify the URL of your Cloudockit Container:

```
{
"DockerUrlForSchedulingStarts" : "https://mycloudockitcontainer"
}
```

This information will be used by Cloudockit Scheduling feature to specify which Web API to call.

Step D (Optional) – Configure Cloudockit Container to support the creation of Compliance Rules, Tailored Diagrams and Settings

Cloudockit Container supports the creation of new Compliance Rules, new Tailored Diagram and new Settings.

This feature requires that you deploy an Azure Cosmos DB to save the Compliance Rules and Tailored Diagrams.

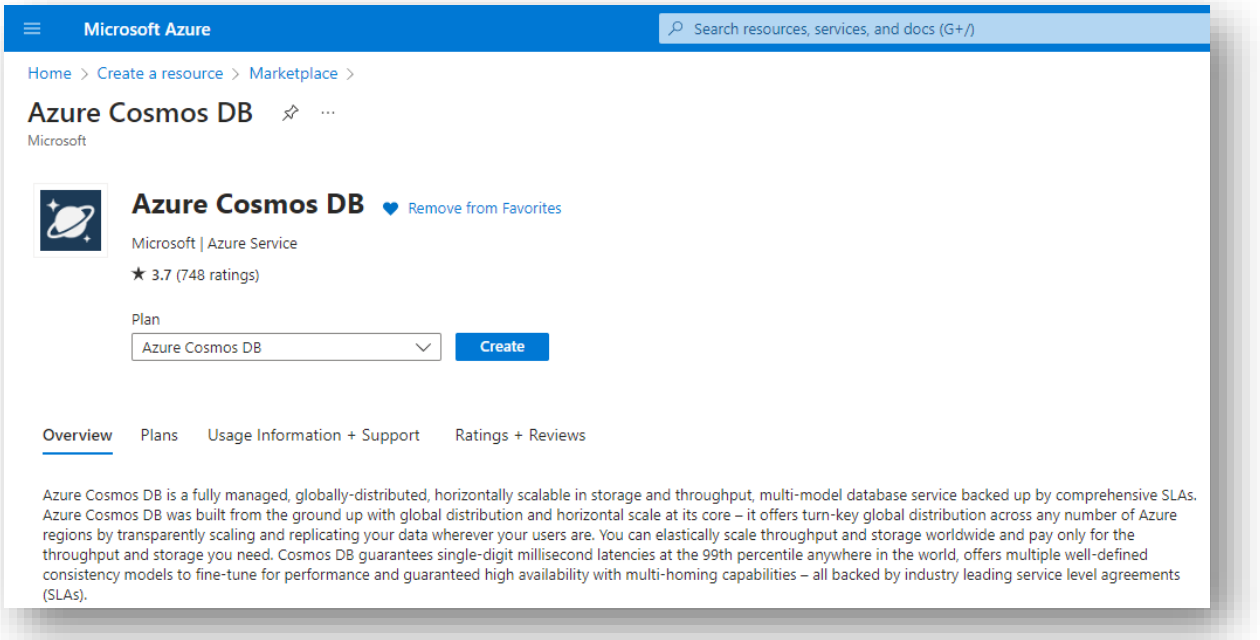
There are two steps required:

- Create (or re-use) an Azure Cosmos DB
- Add environment variables to the Cloudockit Container to specify which Azure Cosmos Database to use

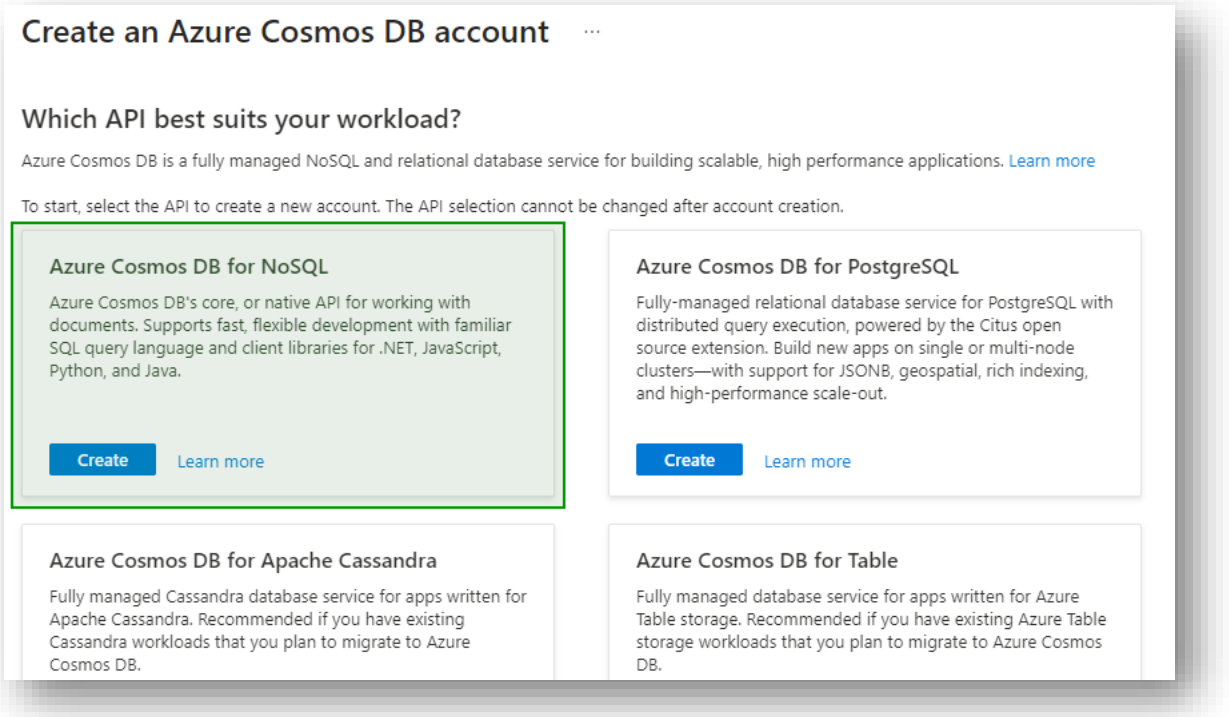
Create (or re-use) an Azure Cosmos DB

From the Azure Portal, create a new Cosmos DB: (you can skip those steps if you already have a Cosmos DB that you want to reuse)

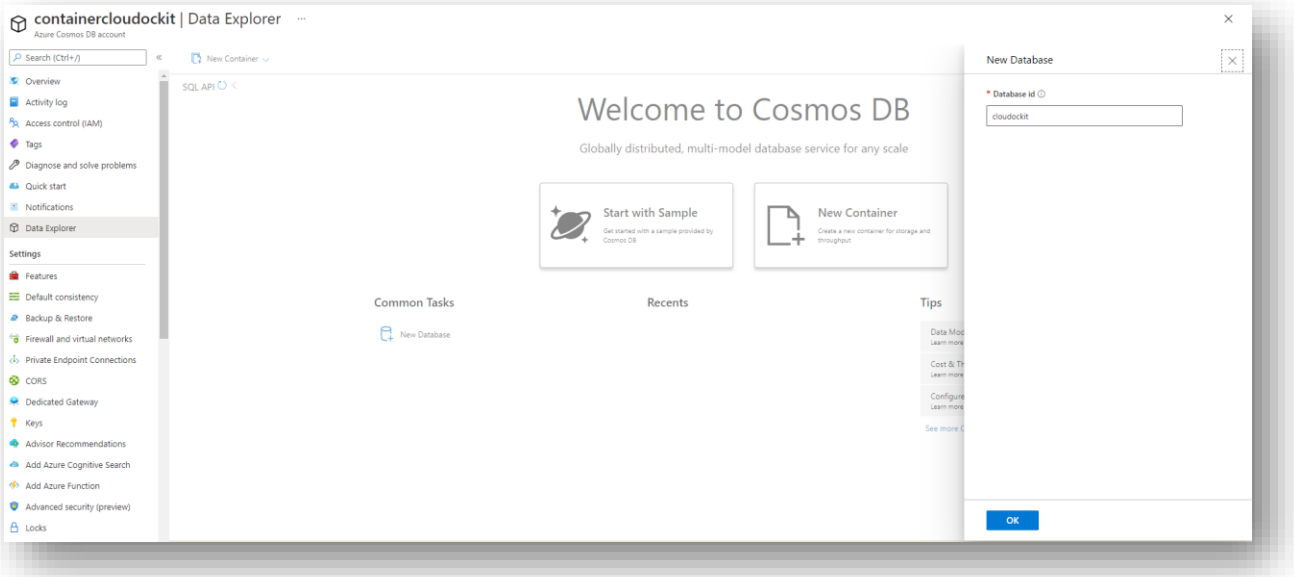
- Create a Cosmos DB



- Choose Azure Cosmos DB for NoSQL for the type



Once the Cosmos DB is created, you need to create a new Database named **cloudockit** :



Configure Cloudockit Container to use the Azure Comos DB

To ensure that the container can connect to the Database, you need to start the container and specify the following 2 required environment variables:

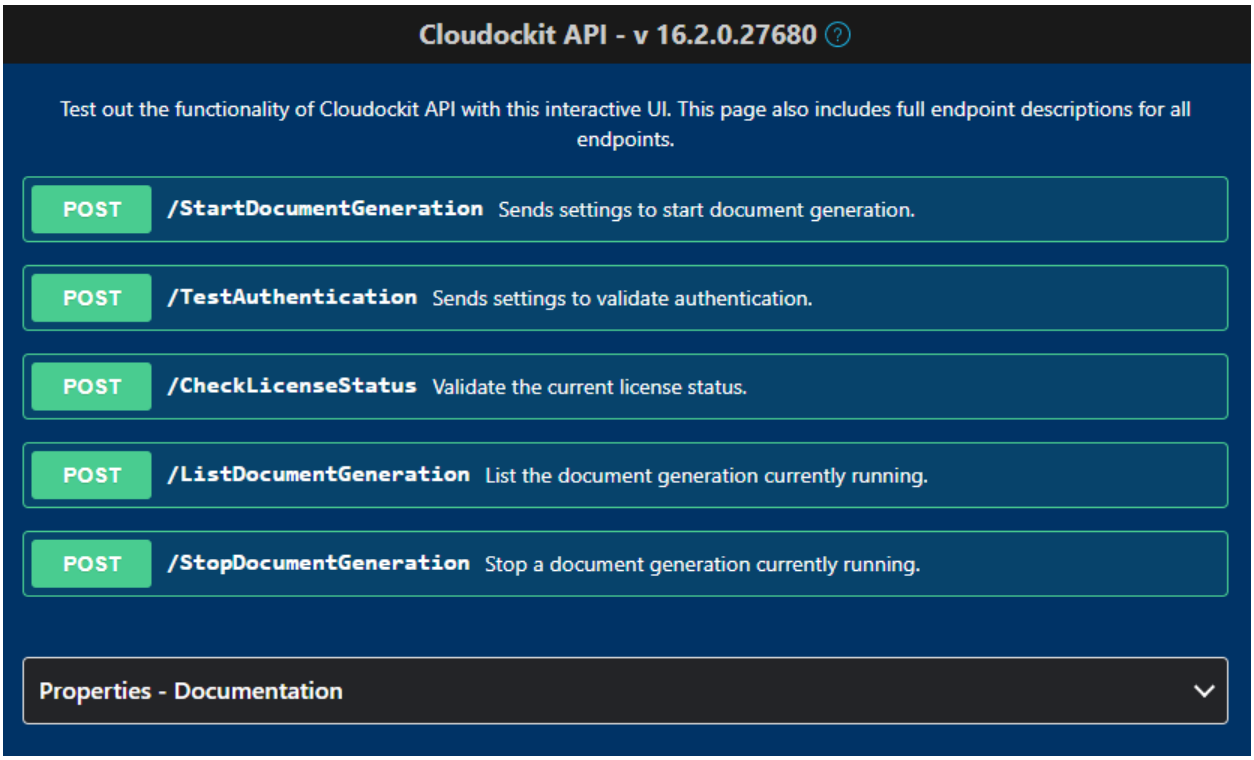
Name	Value
CosmosDb__DatabaseName	Enter the name of the Database that you have created in the previous step (**cloudockit** in the example)
ConnectionStrings__CosmosDb	Azure CosmosDB Connection string

Step E – Understand Cloudockit API Container

Once you have installed the Cloudockit Container, you can navigate to the Container Home Page and you will see the following screen.

It gives you the option to test the different endpoints offered by Cloudockit API.

Please note that you can do everything from command lines/scripts and not use the interface if you prefer.



For simplicity of usage, all the endpoint are POST endpoints. Not all settings are mandatory for each endpoint, and you can refer to that section to see which endpoints require which parameters.

Step F – Test your license

Activate and setup components for your license

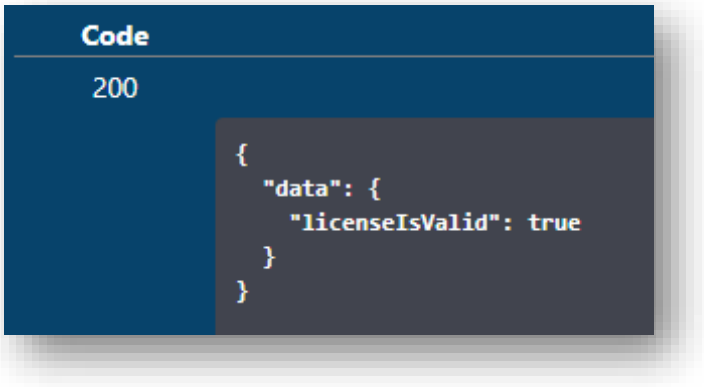
Once you get the API Key from Cloudockit team and you have the appropriate credentials for the license validation, you can check that your API Key is working by using the **/CheckLicenseStatus** endpoint.

First, navigate to the home page of the container and click on **CheckLicenseStatus** and Try it now. Then, replace the following values in the JSON that you are sending to Cloudockit API:

```
{
  "ApiKey": "API Key provided by Cloudockit Team"
}
```

Click on Execute.

You should receive the following response body:



Step G – Validate that you can authenticate to the environment that you want to scan

Once the license validation is successful, you need to test that the authentication to the environment you want to scan is working.

To do that, you need to use the **`/TestAuthentication`** endpoint.

First, you need to ensure that you specify the values from the above Step 2 for license validation.

Then, you need to specify the following additional values:

Name		**Value**
ADKCloudType		Azure/AWS/GCP depending on the platform that you want to scan.
SubscriptionID		Id/Alias of the subscription (Azure) or account (AWS) or project (GCP) that you want to scan.
(for AWS)	AWSAccessKeyId	AWS Access Key
	AWSSecretAccessKey	AWS Secret Access Key
(for Azure)	TenantID	Tenant name of the Azure Subscription to scan
	AppClientIdForAutomation	AAD App ID for the scan
	AppClientKeyForAutomation	AAD App Key for the scan
(for GCP)	GCPServiceAccountCredentials	Content of the JSON Service Credential file
AzureStorageNameForDropOff		**Do not change the name of the parameter for AWS, this is also called AzureStorageNameForDropOff** You should specify <u>one</u> of these values: • the Azure Storage Account Name (it can be the unique Storage Account name that is in the same tenant as the subscription that you scan *or* the complete Azure Storage Account Connection String) • AWS S3 bucket • GCP Bucket where Cloudockit should store the documents generated.

Example of Payload for an AWS environment scan:

```
{
  "ApiKey": "xxxx",
  "AWSAccessKeyId": "XXXX",
  "AWSSecretAccessKey": "8PoBo+4XXXX+/k/MzQ",
  "SubscriptionID": "34XXXX2",
  "AzureStorageNameForDropOff": "XXXdockit",
  "ADKCloudType": "AWS"
}
```

Example of Payload for an Azure environment scan:

```
{
   "ApiKey": "xxxx",
   "TenantID": "X2.onmicrosoft.com",
   "AppClientIdForAutomation": "XXXXX",
   "AppClientKeyForAutomation": "mln/XXXXX=",
   "SubscriptionID": "XXX",
   "AzureStorageNameForDropOff": "XXX",
   "ADKCloudType": "Azure"
}
```

Example of Payload for an GCP environment scan:

```
{
   "ApiKey": "xxxx",
   "GCPServiceAccountCredentials": {"type":
"service_account","project_id": ""cdkXXXX"",""private_key_id"":
""XXXXX"",""private_key"": ""-----BEGIN PRIVATE KEY-----
"nMIIEvQIXXXXXZGy5PArVQS"n2buDJi0URXCKoeWnukG9Cl0fHlP8rFK6+XXXXXX+kJm0Y
xuFOwxdbgpS1n38mQyez7EK"nObnp9wP05ynOxKXJqJx0r1k="n-----END PRIVATE
KEY-----"n"",""client_email"":
""XXXX@cdkproject1.iam.gserviceaccount.com"",""client_id"":
""XXXXX"",""auth_uri"":
""https://accounts.google.com/o/oauth2/auth"",""token_uri"":
""https://oauth2.googleapis.com/token"",""auth_provider_x509_cert_url""
:
""https://www.googleapis.com/oauth2/v1/certs"",""client_x509_cert_url":
""https://www.googleapis.com/robot/v1/metadata/x509/test-
XXXX.iam.gserviceaccount.com"},   "SubscriptionID": "XXXX",
   "AzureStorageNameForDropOff": "XXXX",
   "ADKCloudType": "GCP"
}
```

Step H – Test the document generation

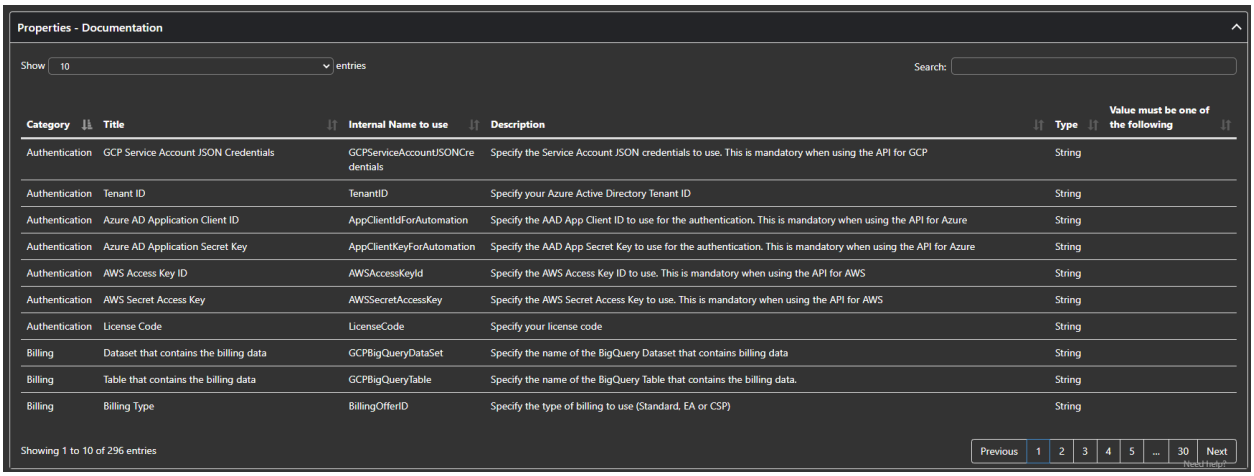
Once all the tests above have been done, you can start the document generation.

To do that, you need to use the **`/StartDocumentGeneration`** endpoint.

First, you need to ensure that you specify the same values as the above steps for CheckLicenseStatus and TestAuthentication endpoints.

Then, you need to specify additional values based on the type of document you want to generate and which option you would like to use.

You get a list of all options from the properties list at the bottom of the screen:



Properties - Documentation

Show 10 entries Search:

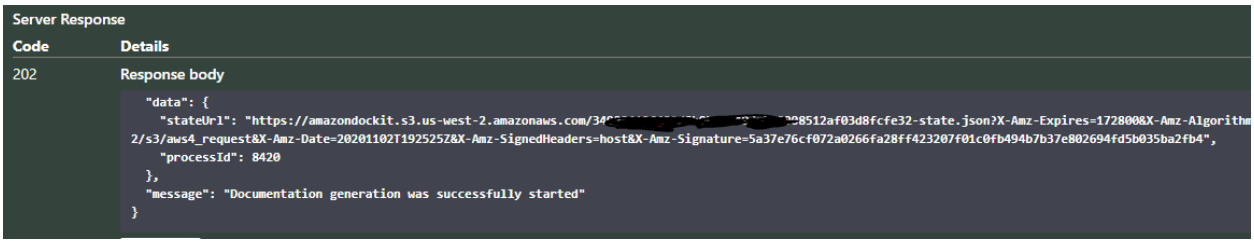
Category	Title	Internal Name to use	Description	Type	Value must be one of the following
Authentication	GCP Service Account JSON Credentials	GCPServiceAccountJSONCredentials	Specify the Service Account JSON credentials to use. This is mandatory when using the API for GCP	String	
Authentication	Tenant ID	TenantID	Specify your Azure Active Directory Tenant ID	String	
Authentication	Azure AD Application Client ID	AppClientIdForAutomation	Specify the AAD App Client ID to use for the authentication. This is mandatory when using the API for Azure	String	
Authentication	Azure AD Application Secret Key	AppClientKeyForAutomation	Specify the AAD App Secret Key to use for the authentication. This is mandatory when using the API for Azure	String	
Authentication	AWS Access Key ID	AWSAccessKeyId	Specify the AWS Access Key ID to use. This is mandatory when using the API for AWS	String	
Authentication	AWS Secret Access Key	AWSSecretAccessKey	Specify the AWS Secret Access Key to use. This is mandatory when using the API for AWS	String	
Authentication	License Code	LicenseCode	Specify your license code	String	
Billing	Dataset that contains the billing data	GCPBigQueryDataSet	Specify the name of the BigQuery Dataset that contains billing data	String	
Billing	Table that contains the billing data	GCPBigQueryTable	Specify the name of the BigQuery Table that contains the billing data.	String	
Billing	Billing Type	BillingOfferID	Specify the type of billing to use (Standard, EA or CSP)	String	

Showing 1 to 10 of 296 entries Previous 1 2 3 4 5 ... 30 Next

As there are many options that you can provide, we strongly advise that you use Cloudockit Website to generate the JSON file with the options.

One of the options that is particularly useful in this scenario are the CallbackURL and CallBackUrlRequired parameters that gives you the ability to be notified once document generation have been done.

When you hit Execute, you get the state URL of the current document generation:



Server Response

Code: 202

Details

Response body


```
{
  "data": {
    "stateUrl": "https://amazondockit.s3.us-west-2.amazonaws.com/34[redacted]8512af03d8fcfe32-state.json?X-Amz-Expires=172800&X-Amz-Algorithm
2/s3/aws4_request&X-Amz-Date=20201102T192525Z&X-Amz-SignedHeaders=host&X-Amz-Signature=5a37e76cf072a0266fa28ff423207f01c0fb494b7b37e802694fd5b035ba2fb4",
    "processId": 8420
  },
  "message": "Documentation generation was successfully started"
}
```


For Payload example, you can simply re-use the previous ones.

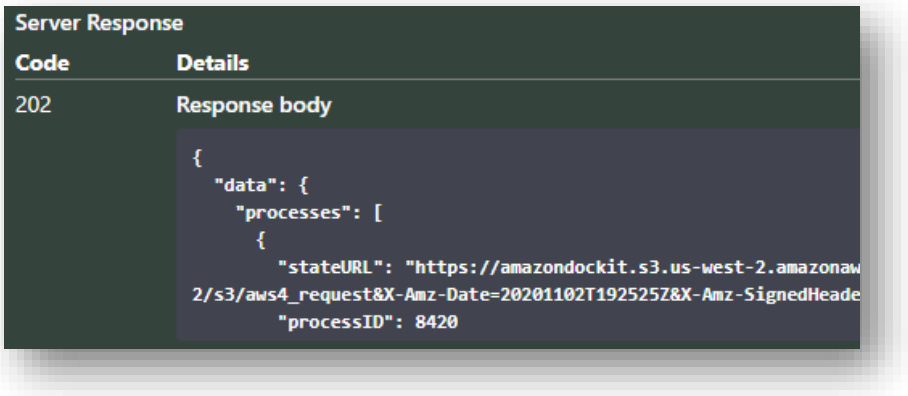
Step I – Manage your document generation

The Cloudockit API offers two endpoints to facilitate the management of document generation.

Please note that for these endpoints, you need to specify an Admin API Key for the ApiKey value.

/ListDocumentGeneration

This will allow you to see which scans are running. It gives you the list of running processes with their Process ID and State:

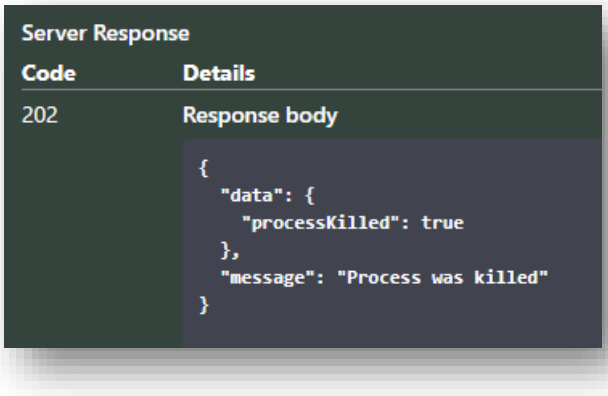


/StopDocumentGeneration

This endpoint is used to kill a running document generation.

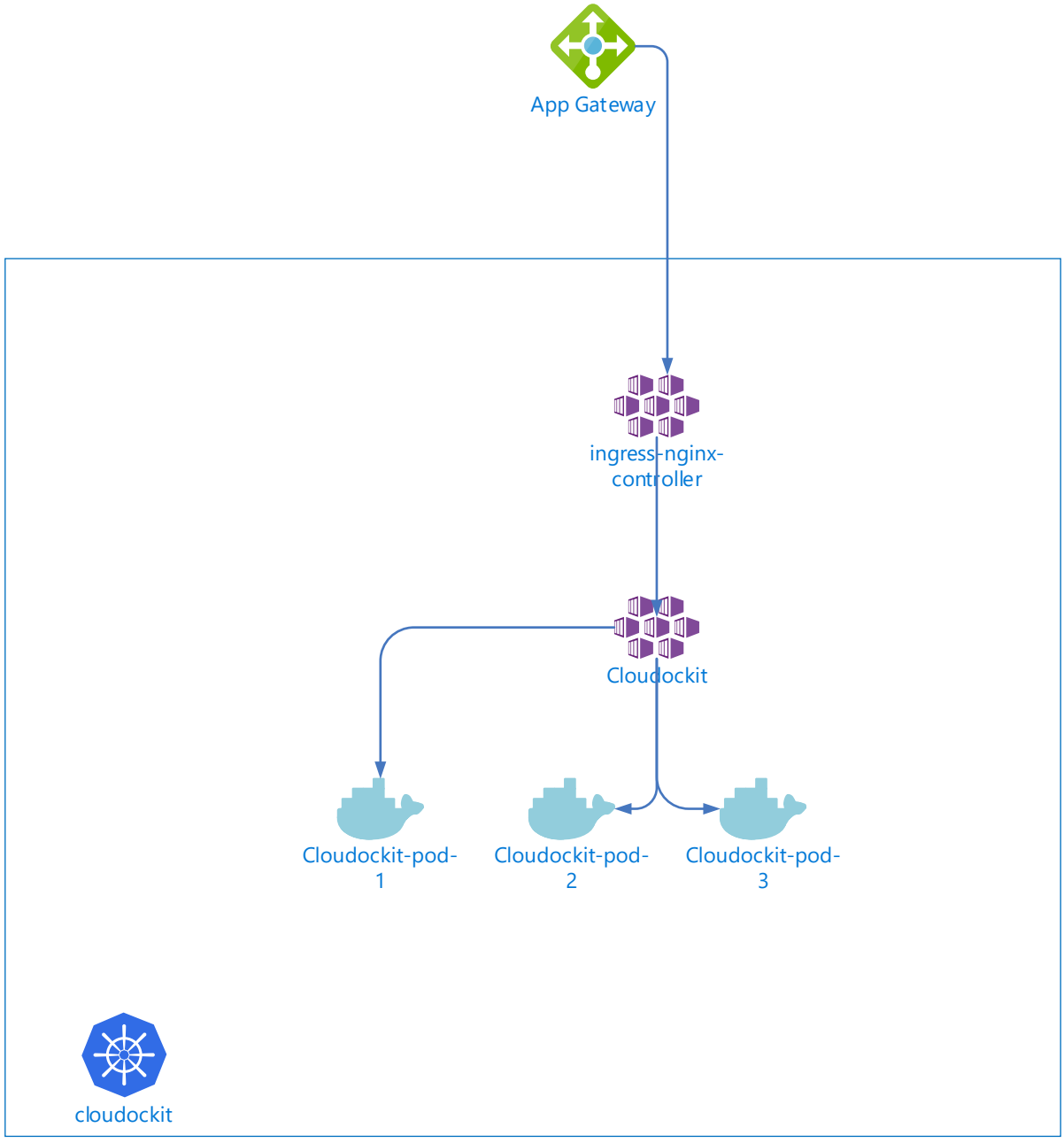
Name	Value
DockerProcessToKill	Value of the process ID to kill

It will reply with a confirmation message that the process has been killed.



Annex – Deploy multiple instances of Cloudockit Container

Cloudockit can be deployed in multiple instances in scenarios like this one:



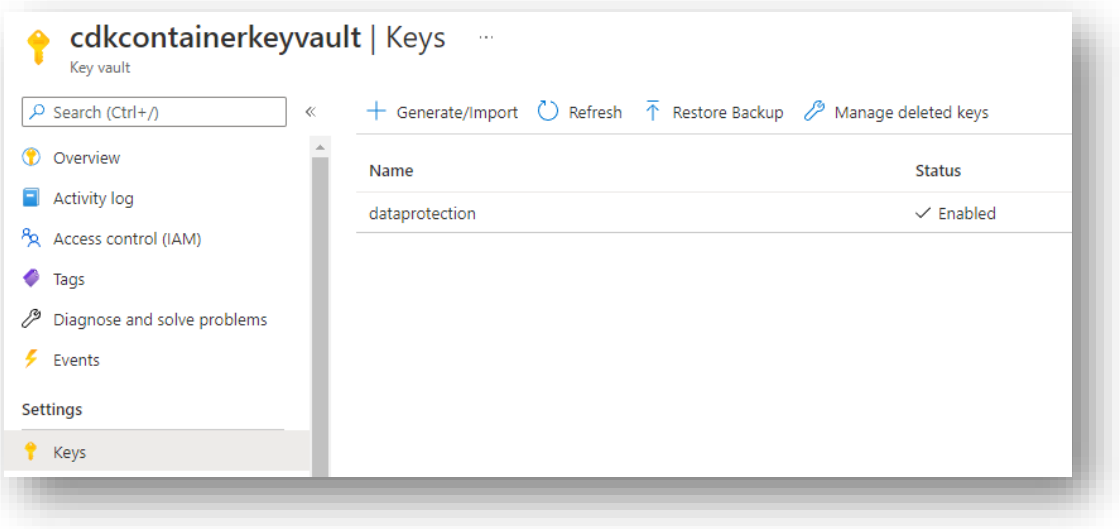
If you plan to use Cloudockit Container in a multi-pods environment, you need to configure some extra components. If you plan to use Cloudockit Containers in multiple instances with sticky session (for example App Services with a Traffic Manager), you do not need those extra components.

Here are the components that you need to configure.

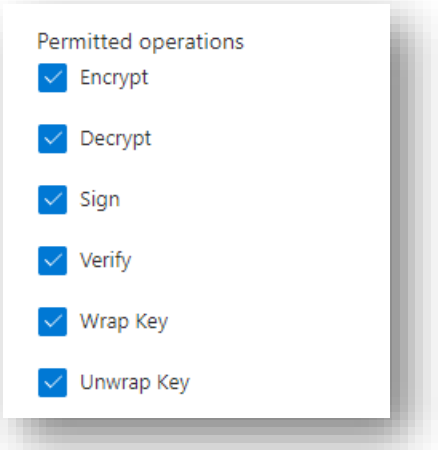
Step 1 – Create / Configure Azure Key Vault

To encrypt the anti-forgery keys used by ASPNETCore, an Azure Key Vault is required. You can create a new Azure Key vault or reuse an existing one.

Once you have the Azure Key Vault, you need to create a Key named **dataprotection**



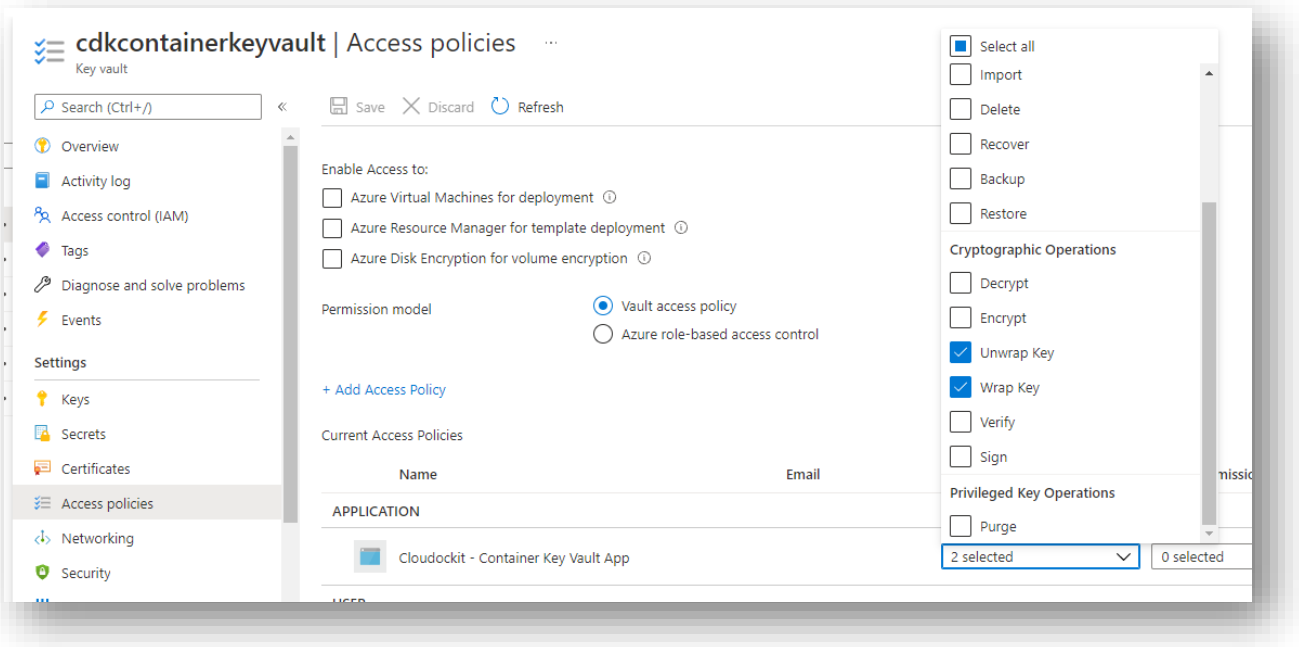
Please ensure that the Key have the following Permitted Operations (by default permissions)



Once you have done that, you need to create an Azure App Registration that will have access to this key. (you can also reuse the Azure AD App that you have created in the steps to configure Cloudockit Web UI if you prefer)

To do that, create a new App Registration (leave default settings) and note the Client ID and Client Secret as you will need that in the next steps.

Go back to the Azure Key Vault and give the Permissions to Unwrap Key / Wrap Key to the App that you just created



Step 2 – Configure Azure Redis Cache

As sessions can sprawl to multi pods, Azure Redis Cache is required to have consistent cache across all nodes.

Create a new Azure Cache for Redis (you can also reuse an existing one if you prefer) and select the Basic C0 (250MB Cache) as only small elements will be cached. Ensure that you select a region that is close to the one where Cloudockit will run for performance optimization.

Once created, take note of the Redis Connection String.

Step 3 – Define the Environment Variables required to run the Cloudockit Container

In addition to the environment variables defined in the step above, you now need to add the following environment variables.

Name	Description	Example
DataProtection__EncryptionKeyUrl	URL of the key vault Key that you have created. You need to specify the **Full Path to the key**, not only the key vault.	https://cdkcontainerkeyvault.vault.azure.net/keys/dataprotection

DataProtection__VaultClientId	Id of the Azure AD App that has privileges to Wrap / Unwrap key	760fb963-57a4-2303-1450-1b2dab513854
DataProtection__VaultSecret	Secret of the Azure AD App	SF7Q~NvuAYKF6.IBFjdewdewd
CacheSettings__UseRedis	Set to true to use redis instead of memory cache	true
ConnectionStrings__Redis	Connection String to the Redis	cdkmultipods.redis.cache.windows.net:6380,password=xx=,ssl=True,abortConnect=False

For reference, here is a sample yaml file to deploy that configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cloudockit
spec:
  replicas: 4
  selector:
    matchLabels:
      app: cloudockit
  template:
    metadata:
      labels:
        app: cloudockit
    spec:
      containers:
        - name: cloudockit
          image: cdkmultipods.azurecr.io/cdk-web-linux:dev
          ports:
            - containerPort: 80
          env:
            - name: DockerStorageCloudProvider
              value: "Azure"
            - name: "DockerStorageAzureCnxString"
              value:
"DefaultEndpointsProtocol=https;AccountName=cloudockitcontainerdebug;AccountKey=xxxx==;EndpointSuffix=core.windows.net"
            - name: AzureADTenant
              value: "umaknowdev.onmicrosoft.com"
            - name: AzureADAppID
              value: "9548a025-xxxx"
```

```
            - name: AzureADAppKey
              value: "W6UXfqC~xxxx"
            - name: Data__ProtectionEncryptionKeyUrl
              value:
"https://cdkcontainerkeyvault.vault.azure.net/keys/dataprotection"
            - name: DataProtection__VaultClientId
              value: "760fb9xxxx"
            - name: DataProtection__VaultSecret
              value: "VSF7xxxx"
            - name: CacheSettings__UseRedis
              value: "true"
            - name: ConnectionStrings__Redis
              value:
"cdkmultipods.redis.cache.windows.net:6380,password=xxxxk9g=,ssl=True,abortConnec
t=False"
            - name: APPINSIGHTS_INSTRUMENTATIONKEY
              value: "c07069xxxx"
            - name: TriggerDeployCount
              value: "5"
---
apiVersion: v1
kind: Service
metadata:
  name: cloudockit
spec:
  type: ClusterIP
  ports:
    - port: 80
  selector:
    app: cloudockit
```

Annex - AWS Container use-cases

Introduction

This annex describes the different use-cases currently available for Cloudockit Container, covering the API and the WebUI host modes, for scenarios such as scanning multiple accounts, using roles, keys, optional drop-off storage. The setups required at the Cloud platform level are listed for each use-case.

AWS ECS Container

To run Cloudockit Container in AWS, the ECS container should already be setup.

A – Scanning multiple AWS accounts using Cross-Account role

Using Cloudockit Container API, it is possible to scan multiple accounts using the Cross-Account Role functionality.
The objective is to give the possibility to a user located in the source account to scan other AWS accounts, by assigning that user a role allowing them to access the target account(s) in read-only mode.
In the following example, ECS Task ‘’EcsTask” located in the source account with account ID “AccountA” wants to scan the resources in the target accounts with account IDs “AccountB”, “AccountC”, etc.

a. Setup in AWS Console for the role to scan multiple accounts

1. Create a role to attach to the ECS Task Definition running Cloudockit.

In the source account where the ECS is running Cloudockit, go to the IAM service, create a new role named (for example) : *EcsTaskRoleForCdk*

- Permissions : Attach the following permission policies
  - *EcsCanAssumeAnyRole* (Customer Managed)
    Create a new permission policy named *EcsCanAssumeAnyRole* with the following JSON:

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "*"
  }
}
```

- Trusted entities : Attach the following Trust Relationship JSON in the role’s trusted entities

```
{
   "Version": "2012-10-17",
   "Statement": [{
       "Effect": "Allow",
       "Principal": {
          "Service": "ecs-tasks.amazonaws.com"
       },
       "Action": "sts:AssumeRole"
     }
  ]
}
```

- In the “EcsTask” Task Definition creation process, attach the created role to the ECS Task Definition (select it in the ECS Task Definition, in dropdown ‘’Task Role”).

2. Create the Multi-Account scan role

In this step we need to define the role which is going to be used in the API to scan all the accounts.

Make sure you have the following setup in every account to scan (source and targets).

- Create a new role named (for example) : *MultiAccountScanRole*
- Permissions : Attach the following permission policies to *MultiAccountScanRole*
    - *ReadOnlyAccess* (AWS managed - job function)
    - *CdkS3BucketAccessPolicy* (Customer Managed)
      Create a new permission policy named *CdkS3BucketAccessPolicy* with the following JSON:

```
{
   "Version": "2012-10-17",
   "Statement": [{
     "Sid": "VisualEditor0",
     "Effect": "Allow",
     "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:PutBucketCORS",
        "s3:PutObject",
        "s3:DeleteObject"
     ],
     "Resource": [
         "arn:aws:s3:::*",
         "arn:aws:s3:::*/*"
     ]
   }
   ]
}
```

- Trusted entities :
  MultiAccountScanRole needs to know the calling entity, which includes the source account id (AccountA) of the calling role, and the role name, which is *EcsTaskRoleForCdk.*
  To do that, attach the following Trust Relationship JSON in the role's Trusted entities.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
        "AWS": [
          "arn:aws:iam::AccountA:role/EcsTaskRoleForCdk"
        ]
    },
    "Action": "sts:AssumeRole",
    "Condition": {}
  }]
}
```

b. Setup for the API Settings to use

In the Cloudockit API, in the ''StartDocumentGeneration" tab, make sure to set the following settings :

- Settings defining the accounts to scan :

```
"SubscriptionId"  : "AccountA"
"AWSRoleToAssume" : "MultiAccountScanRole"
"SelectedSubscriptionsIds": [
  "AccountA",
  "AccountB",
  "AccountC"
]
```

- To drop-off the generated documents in the S3 bucket located in "*AccountA*":

```
"AzureStorageNameForDropOff": "NameOfYourS3BucketInAccountA"
```

- To drop-off the generated documents in the S3 bucket located in another account (in "*AccountB*" for example):

```
"AzureStorageNameForDropOff":
"arn:aws:s3::AccountB:NameOfYourS3BucketInAccountB"
```

- To have the information of all scanned accounts displayed in the same document :

```
"GenerateSingleDocForAllSubscriptions": true
```

Otherwise, by default, the documents will be dropped in the storage account you specify, with a different folder for each account.

B – Using the container with the ECS Task Role instead of IAM user keys

If you don't want to use AccessKeyID/SecretAccessKey when setting up your container, you can use the ECS Task Role assigned to your container.

When creating the Task Definition, instead of setting the Environment Variables

`DockerStorageAWSAccessKeyId` and `DockerStorageAWSSecretAccessKey`

You can set the Environment Variable :

```
DockerStorageUseAwsRole = True
```

Using this option has the following impact :

- The container will access the default container storage (the storage containing the container license file) using the ECS Task role assigned to the container. This will be mainly used for validating the license, reading the user settings file, and getting the saved schedules.

- The scan will still be performed using either the Keys or Role To Assume authentication options :
    - API Settings : AccessKeyID/SecretAccessKey
    - API Settings : AWSRoleToAssume. You can use any role with *ReadOnlyAccess* Policy (including the role assigned to the ECS if you want).
      If you choose the AWSRoleToAssume option, **you should still enter random values** for the AccessKeyID/SecretAccessKey variables, it's a front-end validation glitch that will be fixed in the future.

a. Setup in AWS Console for the ECS Task Role

The ECS Task Role needs to be created with the following permissions :

- Create a new role named (for example) : *EcsTaskRoleContainerIdentity*
- Permissions : Attach the following permission policies to *EcsTaskRoleContainerIdentity*
    - *ReadOnlyAccess* (AWS managed - job function) (necessary if you want to scan your environment using this role)
    - *CdkS3BucketAccessPolicy* (Customer Managed)
      Create a new permission policy named *CdkS3BucketAccessPolicy* with the following JSON:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "VisualEditor0",
    "Effect": "Allow",
    "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:PutBucketCORS",
        "s3:PutObject",
        "s3:DeleteObject"
    ],
    "Resource": [
      "arn:aws:s3:::*",
      "arn:aws:s3:::*/*"
    ]
  }]
}
```

- *AllowEcsDescribeTask* (Customer Managed)
  Create a new permission policy named *AllowEcsDescribeTask* with the following JSON:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [
        "ecs:DescribeTasks",
        "ecs:DescribeTaskDefinition"
    ],
    "Resource": "*"
  }]
}
```

- *EcsCanAssumeAnyRole* (Customer Managed)
  Create a new permission policy named *EcsCanAssumeAnyRole* with the following JSON:

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "*"
  }
}
```

- Trusted entities : attach the following Trust Relationship JSON in the role's trusted entities

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
        "AWS":
     "arn:aws:iam::AccountA:role/EcsTaskRoleContainerIdentity",
        "Service": "ecs-tasks.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }]
}
```

C – Running a scheduled scan in AWS using ECS container.

In this example, we want to run a scheduled scan from within our AWS environment. The container scheduler (the small container triggering the scheduled scans) and the container running the scan are both hosted in AWS.

a. Saving the schedule

The schedule is saved using the Web Interface (Container WebUI), we need to add the Environment Variable referring to the AWS container default S3 bucket which will now contain the saved schedules.

The 2 following steps are necessary if you are using the Container WebUI interface running in Azure to save the schedule.

1. Set the Environment Variable

- Go to a previously setup container in Azure (App service) or setup a new one (ref. ContainerGuide_Azure)
- In the corresponding App Service
  - left panel ➔ section **Settings**
  - **Configuration**
  - Click on **+ New application setting**
    - Name : *DockerStorageNameForAWSScheduler*
    - Value : NameOfYourS3Bucket (write the name of the AWS container default S3 Bucket which contains the license file for the container running AWS)
  - **Save**

2. Save the AWS Schedule

Start your Azure Web App, follow the login steps and save the schedule.

- Sign in with Azure AD
- Select ''AWS'', and enter the Access Key ID & Secret Access Key, log in.
- Once logged-in, set the settings you want for your schedule (make sure the IAM user with the logged-in AccessKeyID/SecretAccessKey have read access to the target account to scan)

- Drop-off tab : Your Storage : enter the name of any S3 bucket where you want your documents to be dropped-off (accessible with the logged-in credentials) and validate.
- Scheduling tab :
    - In the Schedules drop-down, select ''Add new schedule…"
    - Enter the Schedule's Name & CRON (copy/paste CRON expression from http://www.cronmaker.com)
    - Enter the API Key used for the AWS Container (corresponding to the license found in the default S3 bucket)
    - Save Schedule

Your schedule file is now saved in your AWS Container default S3 bucket.

3. Run the schedule in AWS

In this final step, we are going to set the container scheduler in AWS (to trigger the scans) and have the scan executed with the container set up in AWS as well.

3.1. Set you container scheduler

- Create a new task definition/revision for the container scheduler.

In this example we will create a new Task Definition *EcsSchedulerUsingKeys* for the container scheduler, using AccessKeyID/SecretAccessKey environment variables (you can also set it with ECS Task Role instead of the keys, ref. Use-case B)

Launch Type : AWS Fargate | OS : Linux/X86_64

Task size : 1 vCPU & 2GB Memory

Task Roles

- For the ECS Task Role, attach a role (*EcsSchedulerRole*) with the following custom policies :
    - *EcsCanAssumeAnyRole* (Customer Managed) (ref. previous uses-cases)
    - *CdkS3ReadAccessPolicy* (Customer Managed)

```
{
   "Version": "2012-10-17",
   "Statement": [{
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
         "s3:GetBucketLocation",
         "s3:GetObject",
         "s3:ListBucket",
         "s3:PutBucketCORS"
      ],
      "Resource": [
         "arn:aws:s3:::*",
         "arn:aws:s3:::*/*"
      ]}
   ]
}
```

The following Trust relationships :

```
{
   "Version": "2012-10-17",
   "Statement": [{
      "Effect": "Allow",
      "Principal": {
         "Service": "ecs-tasks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
   }]
}
```

- For the Task Execution Role, attach a role with the necessary permissions to access your secret manager to retrieve the secrets required to download the Cloudockit Container image (as described in ContainerGuide_AWS) and to write the container logs (in order to display any errors/warnings and to monitor the state of the schedules running).

  Permissions - Policies to attach :
  - *CloudWatchLogsFullAccess* (AWS Managed)
  - *GetMyRegistrySecretPolicy* (Customer Managed)

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [
      "secretsmanager:GetSecretValue"
    ],
    "Resource": [
        "ArnOfYourSecretFoundInSecretsManager"
    ]
  }
  ]
}
```

The following Trust relationships :

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
      "Service": "ecs-tasks.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }
  ]
}
```

Container details

Set the name, and the Image URI for the scheduler container image (URI ends with *cdk-scheduler-linux:latest*), and enable Private registry authentication.

Environment Variables

Add the following Environment Variables :

- `DockerStorageCloudProvider : AWS`
- `DockerStorageAWSBucketName :` NameOfYourContainerDefaultS3Bucket (same as DockerStorageNameForAWSScheduler set in `1. Set the Environment Variable`)
- `DockerUrlForSchedulingStarts :` URL of the AWS container running the scan (API Host – e.g http://57.137.30.170/) – if this Environment Variable is not assigned in this step, it should be set in the settings.json file found in the S3 bucket next to the license.json file.

To allow the container scheduler to access the storage, you can either use IAM user access keys, **OR** the ECS Task Role to access the storage account containing the schedules :

To use the IAM user access keys :

- `DockerStorageAWSAccessKeyId : Access Key ID of the IAM user`
- `DockerStorageAWSSecretAccessKey : Secret Access Key of the IAM user`

To use the IAM Role assigned to the ECS Task (Task Role) :

- `DockerStorageUseAwsRole : True`

In the case of using the IAM Role, the ECS Task Role previously created also needs to have the following permission policy attached :

  - *AllowEcsDescribeTask* (Customer Managed) (ref. a. Setup in AWS Console for the ECS Task Role)

Modify the Trust relationship to add the ECS Task role's arn in the Principal. It should look like this :

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
      "AWS": "arn:aws:iam::YourAccountId:role/EcsTaskRole",
      "Service": "ecs-tasks.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }]
}
```

Logging

In the Logging section, Enable "Use log collection", use ''Amazon CloudWatch".
The container will automatically create a log group and write the logs.

Once the task definition is created, you can deploy and run it in your cluster.
Monitor the container logs to see the status of the scan (In the running scheduler's container -> tab ''Logs")

Annex – Troubleshooting

Here are resolutions to common cases and how you can help find errors in Cloudockit Container.

- If you activate Cloudockit Container Web UI and noticed that in the upper right corner you have a Welcome message without your name, please check the AAD Credentials in the settings file
- If you are using Private endpoint for your App Service and Storage, please ensure that you activate vNET integration so that the App Service can communicate with the Storage Account
- You can specify an environment variable in your container named `AppInsightKey` that contains an Azure App Insight Instrumentation key so that you can see the logs.
- You can use the -logs.txt file in the storage that you have specified to see what is happening during document generation.
- If you get an error when the document generation starts, please ensure that you have Write privileges to your storage account
- If you see the message that the document generation is starting but do not see any progress, please verify that you have a CORS rule for GET Verb and origin that is your Cloudockit container website (should be done automatically).
- If you get an exception when starting the container that says “APPCMD failed with error code 87”, check that the variables that you are providing do not contain quotes.