CMAP

Azure CI/CD Cloud Deployment (DEV)

**v1.0 as of 07.03.2025**

**Document Revision History**

Each time this document is modified, increment the version number appropriately and add a new row to the table below. In the Comments column, document the changes that were made in sufficient detail.

| **Date** | **Author** | **Version** | **Comments** |
| --- | --- | --- | --- |
| 07/03/2025 | Syneren Technologies Corp | 1.0 | Created |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[1. Introduction 2](#_Toc202373226)

[2. Cloud Services Overview 2](#_Toc202373227)

[3. Frontend Deployment 3](#_Toc202373228)

[Using Azure Static Web App 3](#_Toc202373229)

[4. Backend Deployment 4](#_Toc202373230)

[4a. Node.js Backend Using Azure App Service 4](#_Toc202373231)

[4b. Python Backend Using Azure App Service and Azure Container Registry (ACR) 6](#_Toc202373232)

[Docker Image and Container Setup 6](#_Toc202373233)

[App Service Creation 7](#_Toc202373234)

[5. Database Setup 8](#_Toc202373235)

[Using Azure Cosmos DB with MongoDB API 8](#_Toc202373236)

[Creating the Database 8](#_Toc202373237)

[Importing the Database 9](#_Toc202373238)

[6. Document Storage 11](#_Toc202373239)

[Using Azure Blob Storage 11](#_Toc202373240)

[7. Logging 12](#_Toc202373241)

[Using Azure Application Insights 12](#_Toc202373242)

[8. Key Management 13](#_Toc202373243)

[Using Azure Key Vault 13](#_Toc202373244)

[9. Authentication 14](#_Toc202373245)

[Using Azure App Registrations 14](#_Toc202373246)

[10. Code Environment Changes 16](#_Toc202373247)

[10a. Frontend Environment 16](#_Toc202373248)

[10a. Backend Environment 16](#_Toc202373249)

[Final Notes and Best Practices 17](#_Toc202373250)

# Introduction

This document covers deployment of CMAP frontend and backend on Azure with CI/CD pipelines, Cosmos DB database, Blob Storage for documents, Application Insights for logging, and Key Vault for secrets management.

# Cloud Services Overview

| **Component** | **Technology/Service** | **CI/CD Setup / Notes** |
| --- | --- | --- |
| Frontend | Azure Static Web Apps | Automatic CI/CD pipeline via Azure Repos |
| Backend (Node.js) | Azure App Service (Linux) | CI/CD via Azure Repos + YAML pipeline |
| Backend (Python) | Azure App Service (Linus) + Azure Container Registry (ACR) | CI/CD with Docker build & deploy pipeline |
| Database | Azure Cosmos DB (MongoDB API) | Connection string managed as env variable |
| Document Storage | Azure Blob Storage | For storing application documents |
| Logging | Azure Application Insights | Capture custom logs, traces, and telemetry |
| Key Management | Azure Key Vault | Secure storage of keys, secrets, and connection strings |
| Authentication | App Registrations | Secured Entra ID login |
| Code Environment Changes | None | Changing the URI and other variables in the Code environment |

# Frontend Deployment

## Using Azure Static Web App

Azure Static Web Apps provide a simple way to deploy static frontend apps with built-in CI/CD capabilities tightly integrated with Azure Repos.

**Steps:**

1. **Create an Azure Repo**
   * Go to Azure DevOps → Repos
   * Create a new repository (or use existing)
   * Push your frontend code (Vite project) to this repository
2. **Create an Azure Static Web App**
   * In Azure Portal, click **Create a resource** → Search **Static Web Apps**
   * Click **Create**
3. **Configure Static Web App**
   * **Subscription & Resource Group**: Choose your subscription and resource group
   * **Name**: Provide a name for the Static Web App
   * **Region**: Choose a region close to your users
   * **Environment**: Select **Custom**
   * **Source**: Choose **Azure Repos**
   * **Repository**: Select your frontend repository
   * **Branch**: Select the branch you want to deploy (e.g., master)
   * **Build Presets**: Select **Custom**
   * **App Location**: / or your frontend folder path (e.g., /frontend)
   * **Output Location**: Set to ‘**dist**’ (default output folder for Vite)
4. **Plan and Pricing**
   * Choose an appropriate plan (Free, Standard, etc.) based on your requirements
5. **Review & Create**
   * Click **Review + Create**
   * Once validated, click **Create**
6. **Custom Domain**

**Secure with Custom Domain and HTTPS**

* + In Static Web App settings → Custom Domains
  + Map your domain and enable HTTPS with Azure-managed certificate.

**Result:**

* Azure automatically creates a CI/CD pipeline for the frontend.
* Every commit pushed to the selected branch triggers a build and deployment.

# Backend Deployment

## 4a. Node.js Backend Using Azure App Service

Deploy Node.js backend app with automated CI/CD from Azure Repos.

**Steps:**

1. Push Node.js Backend to Azure Repos
   * Create or use an existing Azure Repo for the backend
   * Push your Node.js code to this repository
2. **Create Azure App Service**
   * In Azure Portal → **Create a resource** → Search **App Service**
   * Select **Create**
   * Configure:
     + Subscription and Resource Group
     + Name for the App Service
     + Publish: **Code**
     + Runtime stack: **Node.js LTS version** (choose latest stable LTS)
     + Operating System: **Linux**
     + Region: Select region close to users
   * Restrict Networking
     + Navigate to **Networking** → **Access Restrictions**
     + Add an **Allow** rule only for your Static Web App’s **outbound IPs**
     + Add a **Deny** rule for 0.0.0.0/0 to block all other public access if not needed
3. **Link Repository**
   * If Azure Repos option is available during creation, select the backend repo and branch to enable direct deployment.
4. **If Azure Repos option NOT available:**
   * Create a YAML pipeline manually in Azure DevOps:
     + In Azure DevOps Pipelines → Create new pipeline
     + Select your repository
     + Configure YAML pipeline to build and deploy:
       - Use Linux build agent
       - Build Node.js app (npm install, npm run build if needed)
       - Use azure/webapps-deploy task to deploy to your App Service
5. **Configure App Settings**
   * Go to your App Service → **Configuration** → **Application Settings**
   * Add necessary environment variables such as:
     + MONGO\_URI (your MongoDB connection string)
     + Any other environment-specific variables
6. **Enable Logging**
   * Go to **App Service Logs**
   * Enable **Application Logging (Filesystem)**
   * Set retention as needed for troubleshooting

**Result:**

* Automatic build and deploy on every push to your backend repo branch.
* Backend app runs on Azure App Service with configured environment variables.

## 4b. Python Backend Using Azure App Service and Azure Container Registry (ACR)

For Python backend, containerize your app and deploy on Azure App Service with Docker containers, using Azure Container Registry (ACR) to store images.

**Steps:**

### Docker Image and Container Setup

#### Option A: Docker Image Local Environment

1. **Containerize Python App**
   * Create a Dockerfile for your Python backend app
   * Build your Docker image locally and test
2. **Push Docker Image to Azure Container Registry (ACR)**
   * Create an Azure Container Registry in Azure Portal
   * Tag your image with ACR login server URL
   * Push image to ACR

#### Option B: Docker Image Remotely

1. **Login to Azure CLI**
   * az login
2. **Create Resource Group (if not created already)**
   * az group create --name cmap-python-backend-rg –location eastus
   * Free tier is available in eastus, centralus, and a few other regions.
3. **Create Azure Container Registry (Free)**

* az acr create \

--resource-group cmap-python-backend-rg \

--name cmapacrbackend \

--sku Basic \

--admin-enabled true

* --sku Basic is the smallest, and free up to a few GB/month.

1. **Build the Image Remotely (no local docker)**
   * + - * az acr build --registry cmapacrbackend --image python-ocr:latest
         * Run from the folder where your Dockerfile is.

### App Service Creation

1. **Create Azure App Service for Containers**
   * Create new App Service
   * In **Publish**, choose **Docker Container**
   * Set Linux OS
   * Configure container settings to pull from your ACR image
   * **Image Source**: Azure Container Registry
   * **Registry**: select your ACR (e.g., cmapacr)
   * **Image**: select your image (e.g., cmap-backend)
   * **Tag**: select your tag (e.g., prod)
     + - * Restrict Networking

Navigate to **Networking** → **Access Restrictions**

Add an **Allow** rule only for your Static Web App’s **outbound IPs**

Add a **Deny** rule for 0.0.0.0/0 to block all other public access if not needed

1. **Configure App Settings**
   * Add environment variables (e.g., MONGO\_URI) in the App Service configuration
2. **CI/CD Pipeline**
   * Setup Azure DevOps pipeline to:
     + Build Docker image on code push
     + Push to ACR
     + Trigger deployment on App Service
3. **Enable Logging**
   * Enable container logs and App Service logs as needed

# Database Setup

## Using Azure Cosmos DB with MongoDB API

Azure Cosmos DB offers a fully managed, scalable NoSQL database service. For MongoDB-compatible apps, Cosmos DB provides a MongoDB API.

### Creating the Database

**Steps:**

1. **Create Cosmos DB Account**
   * Go to Azure Portal → **Create a resource** → Search **Azure Cosmos DB**
   * Click **Create**
   * Select **API** as **MongoDB**
   * Choose the subscription and resource group
   * Provide an account name
   * Choose region closest to your backend App Service for low latency
   * Select capacity mode and other settings as needed
   * ***Security:*** Enable **Data Encryption** (enabled by default with Microsoft-managed keys, optionally bring your own key [CMK])
2. **Secure Networking (Highly Recommended)**
   * Before proceeding further:
   * Go to your Cosmos DB resource → **Networking**
   * Set **Public network access** to:
     + Disabled (to block external access)
   * Enable **Private Endpoint** (optional, for internal Azure access only)
   * Add **Firewall rules** (if public access is enabled temporarily), specifying:
     + Your App Service outbound IPs only

This ensures that **only trusted services** (e.g., your backend App Service) can access the database.

1. **Get Connection String**
   * Once the Cosmos DB account is created, navigate to **Keys**
   * Copy the **Primary Connection String**
2. **Use Connection String**
   * Set the copied connection string as the MONGO\_URI environment variable in your backend App Service configuration (Node.js)
   * Example:

mongodb://username:password@clustername:port/dbname?ssl=true&replicaSet=globaldb&retrywrites=false&maxIdleTimeMS=120000&appName=@your-account@

### Importing the Database

#### Option A: Via MongoDB Compass (For Initial Setup)

**Steps:**

1. Install MongoDB Compass
   * Download MongoDB Compass for your operating system
2. Unzip DB file
   * Unzip the db.zip, and extract the ‘cmap’ folder
3. Create the connection with the Cosmos MongoDB URI
   * Click on New Connection
   * Enter the MongoDB URI (something like this:

mongodb+srv://<username>:<password\_with\_%for\_special\_char>@<cluster-name>.global.mongocluster.cosmos.azure.com/<dbname>?tls=true&authMechanism=SCRAM-SHA-256&retrywrites=false&maxIdleTimeMS=120000

* + Click on Save and Connect

1. Create new DB
   * Click on New Database
   * Enter the Database name as ‘cmap’
   * Enter the collection name as ‘user’
2. Import User
   * Once the user collection is created, click on import and choose the user.json from cmap extracted db
3. Import Systems
   * Once the systems collection is created, click on import and choose the systems.json from cmap extracted db
4. Continue or Exit
   * You can continue doing same for all collections if you need.
   * Download MongoDB Tools to use this command to import the whole new db if it works:

mongorestore --db cmap your\_extracted\_db\_location\cmap

#### Option B: Via CMAP Web Application

**Steps:**

Once the user and systems collection are added to the db you can continue adding others via CMAP Web Application UI.

1. Login to CMAP via Global Admin
2. Click on Import/Export DB button
   * Click on Import tab
   * Choose the db.zip file
   * Select Append/Override as per your choice
   * Keep the db name as cmap
   * Hit Import

# Document Storage

## Using Azure Blob Storage

Azure Blob Storage is used to store unstructured data like documents, images, or any files your application needs to serve or process.

**Steps to set up:**

1. **Create an Azure Storage Account**
   * In Azure Portal → **Create a resource** → Search **Storage account**
   * Choose subscription, resource group, name, and region (preferably same region as your app)
   * Choose **Standard** performance and **BlobStorage** or **General Purpose v2** account kind
2. **Create Blob Container**
   * In the Storage Account → **Containers** → **+ Container**
   * Name the container (e.g., documents)
   * Set Public access level as needed (usually private)
3. **Get Access Credentials**
   * Use either:
     + **Connection String** (found under Storage Account → Access keys)
     + Or better: Use **Azure Key Vault** (covered below) to securely store these keys
4. **Integrate with Application**
   * Use Azure SDKs (e.g., @azure/storage-blob for Node.js, azure-storage-blob for Python) to upload/download documents programmatically
   * Store the connection string or SAS tokens in environment variables or Azure Key Vault

# Logging

## Using Azure Application Insights

Azure Application Insights provides deep application performance monitoring, telemetry, and the ability to log custom events and traces.

**Setup and Integration:**

1. **Create Application Insights Resource**
   * In Azure Portal → **Create a resource** → Search **Application Insights**
   * Choose subscription, resource group, region, and application type (e.g., Node.js or Python)
2. **Configure Backend Application**
   * **Node.js:**
     + Install Application Insights SDK:

npm install applicationinsights

* + - Initialize in your code (e.g., app.js):

Examle: Track custom event

client.trackEvent({ name: "custom event", properties: { customProperty: "value" } });

* + **Python:**
    - Install Application Insights SDK:

1. **Set Environment Variable**
   * Add environment variable APPINSIGHTS\_INSTRUMENTATIONKEY to your App Service (or container environment) with the Instrumentation Key from the Application Insights resource.
2. **Monitor and Query Logs**
   * Use the Azure Portal Application Insights blade to:
     + View performance metrics
     + Analyze request rates and failures
     + Query custom events and traces using Kusto Query Language (KQL)
     + Set up alerts and dashboards for proactive monitoring

# Key Management

## Using Azure Key Vault

Azure Key Vault securely stores secrets, keys, and certificates used by your applications.

**Steps:**

1. **Create Azure Key Vault**
   * Azure Portal → **Create a resource** → Search **Key Vault**
   * Enter name, subscription, resource group, region
2. **Add Secrets / Keys**
   * In your Key Vault → **Secrets** → **+ Generate/Import**
   * Add secrets such as:
     + Cosmos DB connection string
     + Storage account keys or SAS tokens
     + Any other sensitive credentials
3. **Access Key Vault from App Services**
   * Grant your App Service managed identity **Get** permissions on the Key Vault secrets via Access Policies
   * In your application, use Azure SDK or environment variables referencing Key Vault references (e.g., via @Microsoft.KeyVault syntax in App Settings)
4. **Use Key Vault references in App Service**
   * In App Service Configuration, set values like

@Microsoft.KeyVault(SecretUri=https://<your-vault-name>.vault.azure.net/secrets/<secret-name>/<secret-version>)

* + This allows your app to retrieve secrets securely at runtime without hardcoding them

# Authentication

## Using Azure App Registrations

Azure App Registrations allow a third-party application to interact with Microsoft APIs like msal, graph. It can also be used to login via Microsoft Entra ID.

**Steps:**

1. **Go to App Registrations**
   * Azure Portal → **App Registrations** → Click New Registration
   * Enter name
     + CMAP\_AUTH
   * Supported account types
     + Choose Accounts in this organizational directory only if you want only that org to have access to the web application else choose Accounts in any organizational directory
   * Redirect URI (optional at this step)
     + Select Single-page application and enter the Custom Domain URL of frontend
2. **Go to Authentication**
   * Add a platform
   * Select Single-page application
   * Click add uri, and enter the frontend URI
   * Hit save
3. **Go to Certificates & Secrets**
   * Click on New client secret
   * Enter Description as ‘CMAP secret’
   * Choose expiration duration
   * Hit ‘Add’
4. **Go to Expose an API**
   * Click on Add a scope
   * Put the scope name as access\_as\_user
   * Select ‘Admins and Users’ in Who can Consent?
   * Admin consent display name: ‘Access CMAP API’
   * Admin consent description: ‘Allows the app to call CMAP API on your behalf.’
   * State: Enabled
   * Hit Save
5. **Go to API Permissions**
   * Click ‘Add a permission’
   * Select Microsoft Graph
   * Choose Delegated Permissions
   * From the dropdown/search bar, select **email**, **openid**, and **profile**
   * Hit Add permissions.
   * Again click ‘Add a permission’
   * Select My APIs and choose ‘CMAP\_AUTH’
   * Choose Delegated Permissions
   * From the dropdown/search bar, select **access\_as\_user**
   * Hit Add permissions
6. **Go to Overview and Note Down the Application (Client) ID and Directory (Tenant) ID.**

# Code Environment Changes

## 10a. Frontend Environment

**Steps:**

**Open ‘.env’ file in frontend code root**

Replace the following values

**Authentication Changes:**

* + VITE\_AZURE\_CLIENT\_ID: (put **Application (Client) ID** that you noted in 9. Authentication Step)
  + VITE\_AUDIENCE: (put **Application (Client) ID** that you noted in 9. Authentication Step)
  + VITE\_TENANT\_ID: (put **Directory (Tenant) ID** that you noted in 9. Authentication Step)
  + VITE\_REDIRECT\_URI: (put the **frontend URL**)

**URI Changes:**

* + VITE\_API\_URL: (put the **node backend API URL**)
  + PYTHON\_API\_URL: (put the **python backend API URL**)

## 10b. Backend Environment

**Steps:**

**Open ‘.env’ file in node.js backend code root**

Replace the following values

**Authentication Changes:**

* + AZURE\_CLIENT\_ID: (put **Application (Client) ID** that you noted in 9. Authentication Step)
  + AUDIENCE: (put **Application (Client) ID** that you noted in 9. Authentication Step)
  + TENANT\_ID: (put **Directory (Tenant) ID** that you noted in 9. Authentication Step)

**URI Changes:**

* + FRONTEND\_URL: (put the **frontend URL**)

**PORT Changes:**

* + PORT: (put the **PORT** in use for backend)

**DB Changes:**

MONGO\_URI: (put the **Cosmos MongoDB URI**, somethinglike this mongodb+srv://<username>:<password\_with\_%for\_special\_char>@<cluster-name>.global.mongocluster.cosmos.azure.com/<dbname>?tls=true&authMechanism=SCRAM-SHA-256&retrywrites=false&maxIdleTimeMS=120000

**Third-party Credentials Changes:**

* + OUTLOOK\_EMAIL: (put the email\_id of outlook from which you want to send emails)
  + OUTLOOK\_EMAIL\_PWD: (enter the password of that outlook account)
  + SN\_USERNAME: (put the username of ServiceNow from which you want to add support tickets)
  + SN\_PASSWORD: (enter the password of that ServiceNow account)

**Keys/Secret:**

If you don’t want to handle Keys/Secrets in Azure Key Vault, then you will have to handle them locally in the environment.

* + FILE\_ENCRYPTION\_KEY: (put the encryption key aes-256-gcm)
  + FILE\_SECRET: (put the secret aes-256-cbc)

# Final Notes and Best Practices

* **CI/CD Automation**: Azure Static Web Apps and App Service (with Azure Repos or YAML pipelines) enable fully automated deployments triggered by code changes.
* **Regional Proximity**: Ensure that all services (App Services, Cosmos DB, Blob Storage, Application Insights, and Key Vault) are created in the same region for **low latency**, **reduced costs**, and **better performance**.
* **Secure Secrets Management**: Store sensitive data like MONGO\_URI, APPINSIGHTS\_INSTRUMENTATIONKEY, and storage keys in **Azure Key Vault** instead of hardcoding them or using plain environment variables.
* **Managed Identities**: Use **Azure Managed Identity** to grant access to Key Vault and other services without requiring credentials in your code.
* **Custom Logging**: Use **Azure Application Insights** to track:
* Backend performance
* Request failures and exceptions
* Custom events and traces (e.g., user actions, debug info)
* **Document Storage**: Use **Azure Blob Storage** for storing documents, files, or media. Access securely using **SAS tokens** or credentials stored in Key Vault.
* **Monitoring and Alerts**: Configure alerts in Application Insights for:
* Failed requests
* Performance bottlenecks
* Custom error tracking  
  Use Kusto Query Language (KQL) for advanced diagnostics and dashboards.
* **Containerization (Python)**: Containerizing the Python backend via Docker and deploying through **Azure Container Registry (ACR)** + App Service ensures consistent environments and reproducible builds.
* **Logging Enablement**: Always enable **App Service diagnostics logging** alongside Application Insights for holistic observability.
* **Scalability**: Azure services like Static Web Apps, Cosmos DB, and Blob Storage scale seamlessly, supporting future growth in users and data.