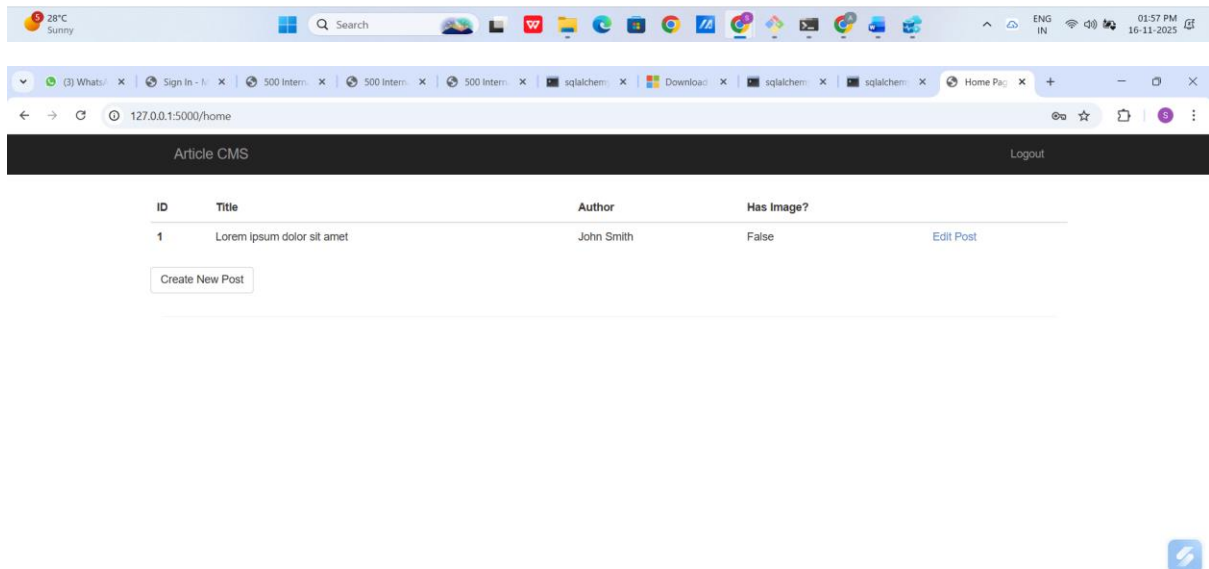
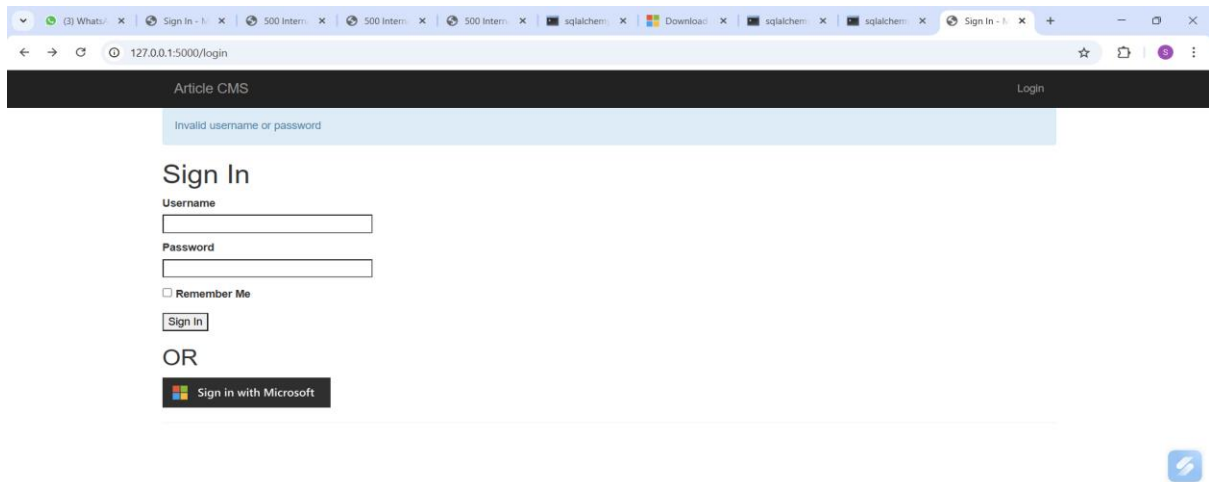


Project: Deploy An Article CMS to Azure

1. A screenshot of an article created in the Article CMS on Azure. The screenshot must also include the URL. The article should have the following fields set:

- Title: "Hello World!"
- Author: "Jane Doe"
- Body: "My name is Jane Doe and this is my first article!"
- An image of your choice. It must be either a .png or .jpg.



Article CMS

Logout

Edit Post

Title


Hello World!

Author

Jane Doe

Body

My name is Jane Doe and this is my first article!



Image

Choose File No file chosen

Save

27°C Sunny

Search

ENG IN 04:27 PM 16-11-2025

Article CMS

Logout

ID	Title	Author	Has image?	
1	Lorem ipsum dolor sit amet	John Smith	False	Edit Post
2	Hello World!	Jane Doe	True	Edit Post
3	Cat	Rita	True	Edit Post

Create New Post

27°C Sunny

Search

ENG IN 04:27 PM 16-11-2025

images - Microsoft Azure

portal.azure.com/#view/Microsoft_Azure_Storage/ContainerMenuBlade/~/overview/storageAccountId/%2F439253e6-9257-449b-87ba-079b323b67d0...

Microsoft Azure

Search resources, services, and docs (G+J)

Copilot

odl_user_290018@udaci...
UDACITY - DS (UDACITYHOLON...

Home > images11 | Containers >

images
Container

Search

+ Add Directory Upload Change access level Refresh Delete Copy Paste Rename Acquire lease Break lease Edit columns

Overview

Diagnose and solve problems

Access Control (IAM)

Settings

Authentication method: Access key (Switch to Microsoft Entra user account)

Add filter

Search blobs by prefix (case-sensitive)

Only show active blobs

Showing all 2 items

Name	Last modified	Access tier	Blob type	Size	Lease state
6Y9U1K4UM03NK2CL5YRFB1D22HSSXAQ6.jpg	11/16/2025, 4:27:22 PM	Cool (Inferred)	Block blob	4.52 KiB	Available
RSRJFBY2UVP1M0617WM8J9MSNFC746K.jpg	11/16/2025, 4:26:15 PM	Cool (Inferred)	Block blob	5.05 KiB	Available

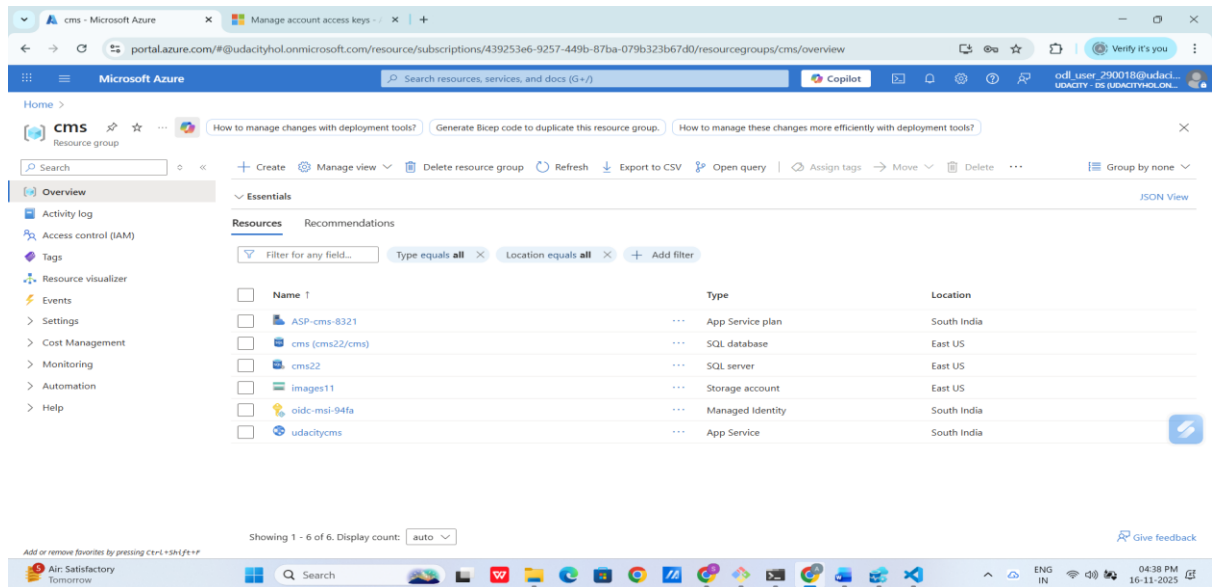
Add or remove favorites by pressing Ctrl+Shift+F

Air Satisfactory
Tomorrow

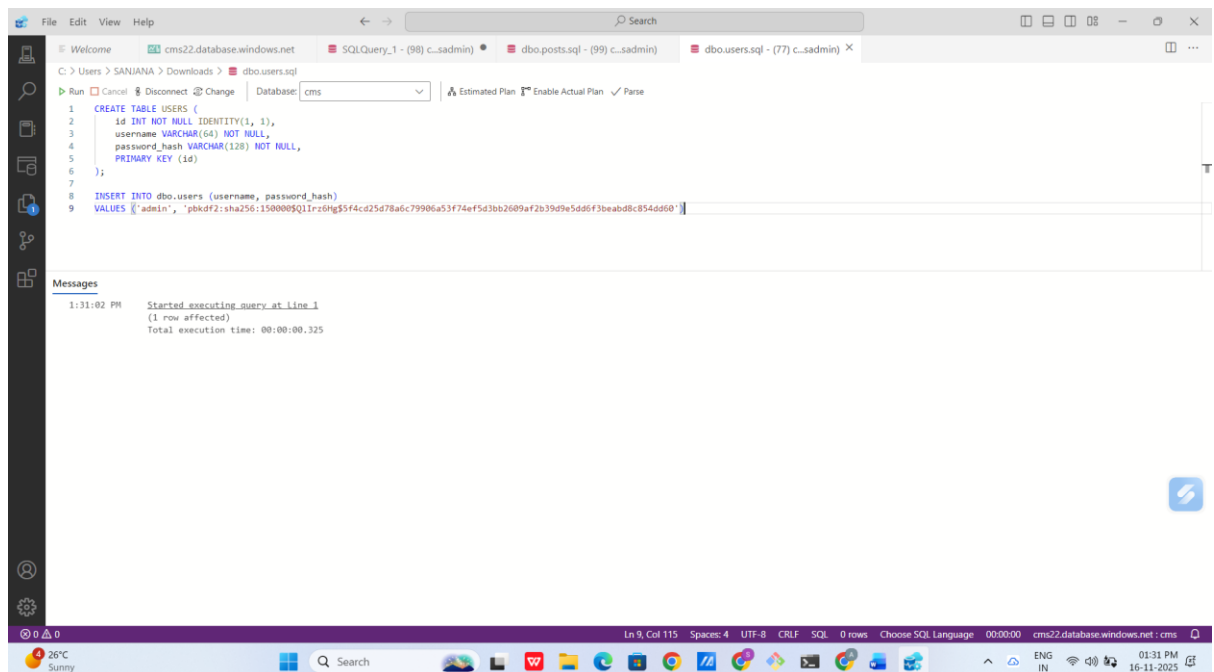
Search

ENG IN 05:36 PM 16-11-2025

2. A screenshot of the resource group from the Azure Portal including all of the resources that were created to complete this project. (see sample screenshot above).



3. A screenshot showing the created tables and one query of data from the initial scripts in the SQL database (see example in the project repository).



SQLQuery_1 - (98) c_sadmin

```
1 CREATE TABLE dbo.posts (
2     id INT NOT NULL IDENTITY(1, 1),
3     title VARCHAR(150) NOT NULL,
4     author VARCHAR(75) NOT NULL,
5     body VARCHAR(800) NOT NULL,
6     image_path VARCHAR(100) NULL,
7     timestamp DATETIME NOT NULL DEFAULT(GETDATE()),
8     user_id INT NOT NULL,
9     PRIMARY KEY (id),
10    FOREIGN KEY (user_id) REFERENCES users(id)
11 )
12
13 INSERT INTO dbo.posts (title, author, body, user_id)
14 VALUES (
```

Messages

1:31:32 PM Started executing query at line 1
(1 row affected)
Total execution time: 00:00:00.445

SQLQuery_1 - (98) c_sadmin

```
1 SELECT * FROM dbo.users;
2 SELECT * FROM dbo.posts;
3
```

Results

id	username	password_hash
1	admin	pbkdf2:sha256:150000\$Q1lrz0lg5f4cd25d78a6c79906a53f74ef5...

id	title	author	body	image_path	timestamp	user_id
1	Lorem ipsum dolor sit amet	John Smith	Proin sit amet mi ornare, ultrices augue quis, facilisis ...	NULL	2025-11-16 08:01:33.293	1

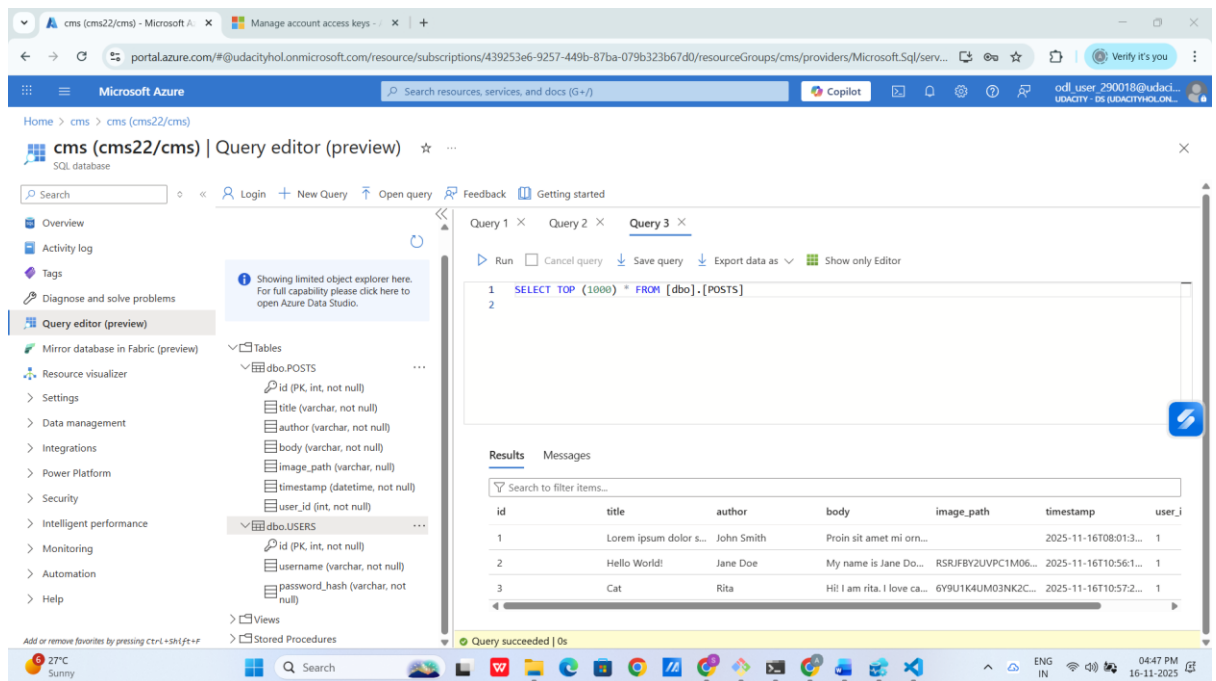
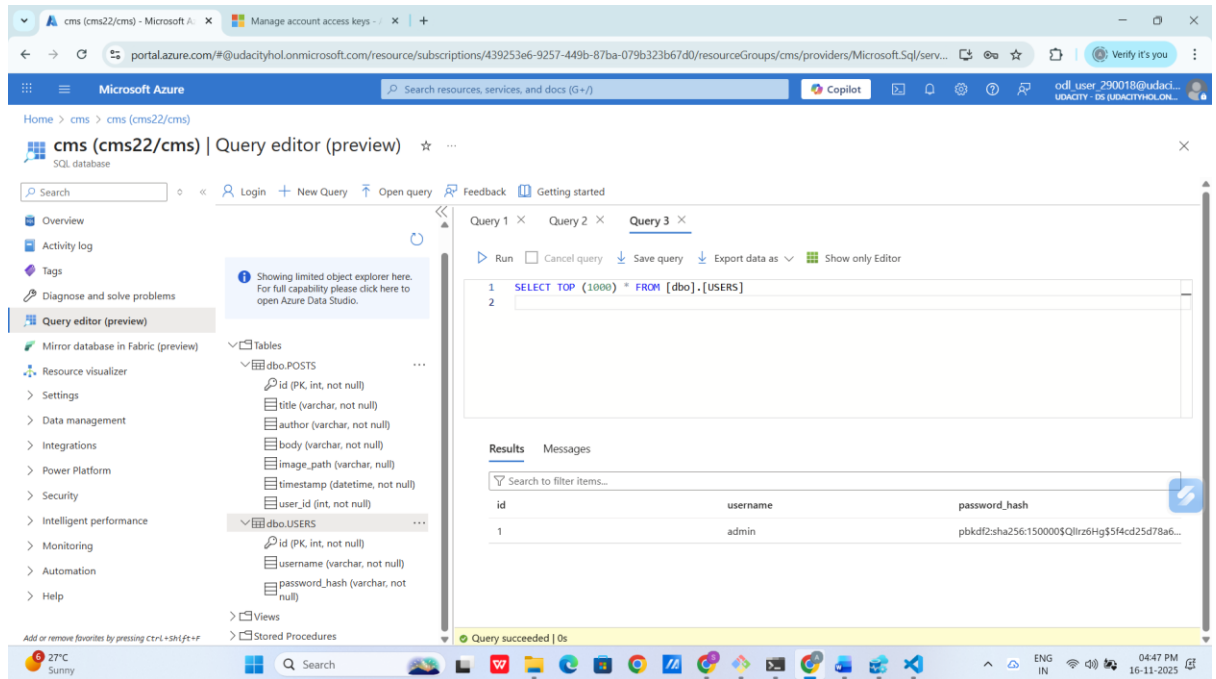
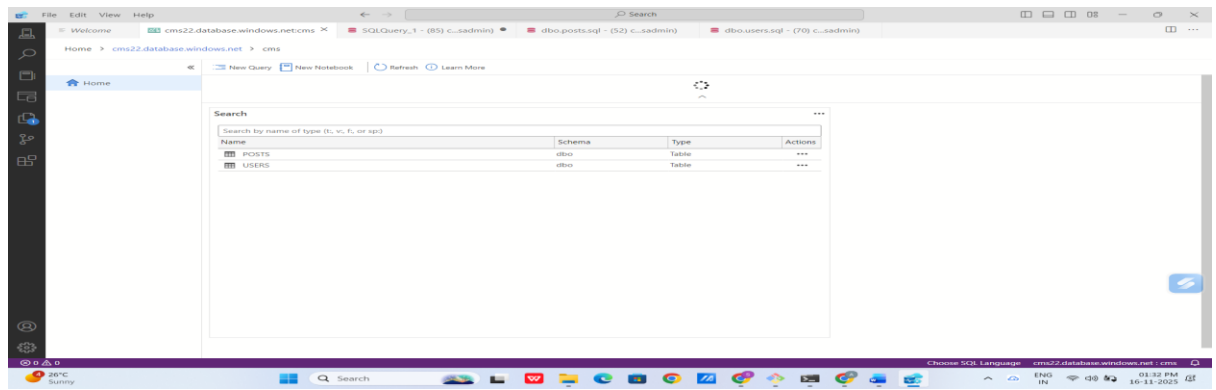
Home > cms22.database.windows.net

Version: 12.0.2000.8

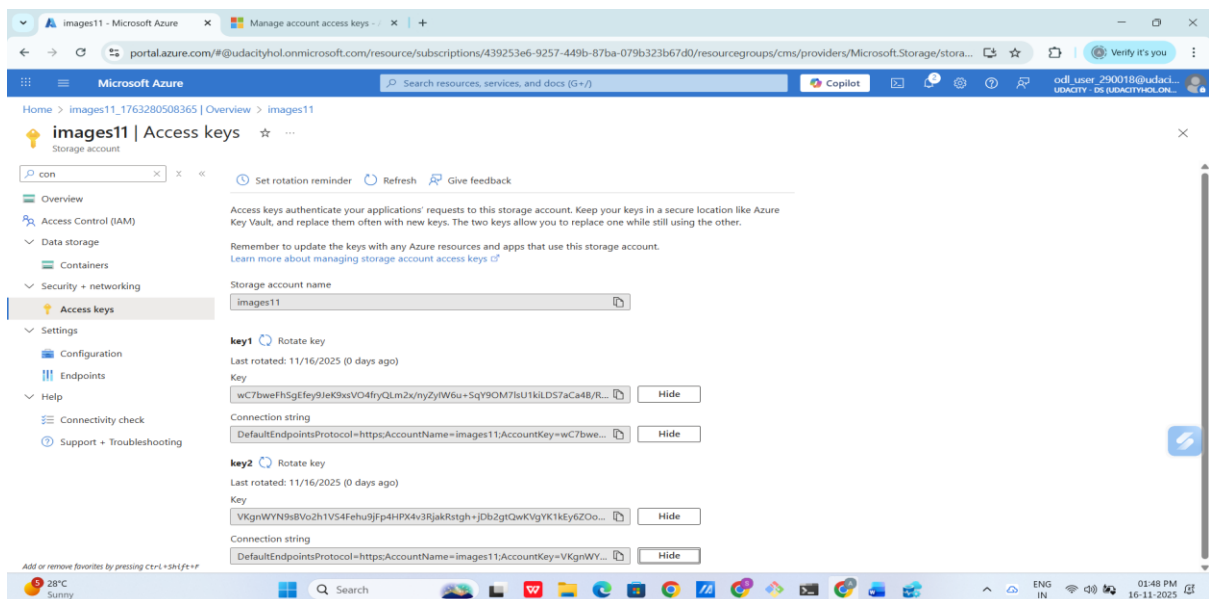
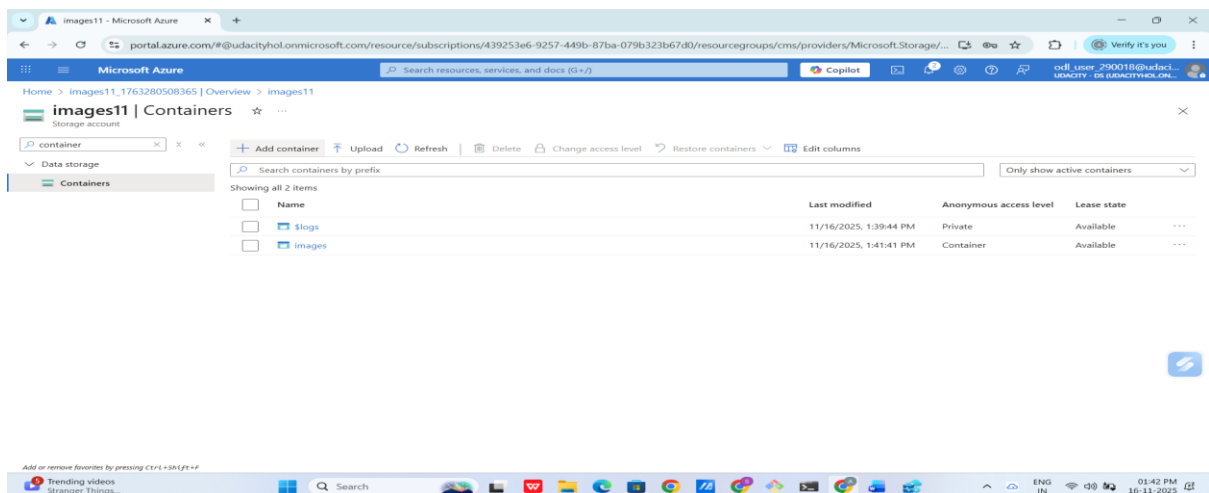
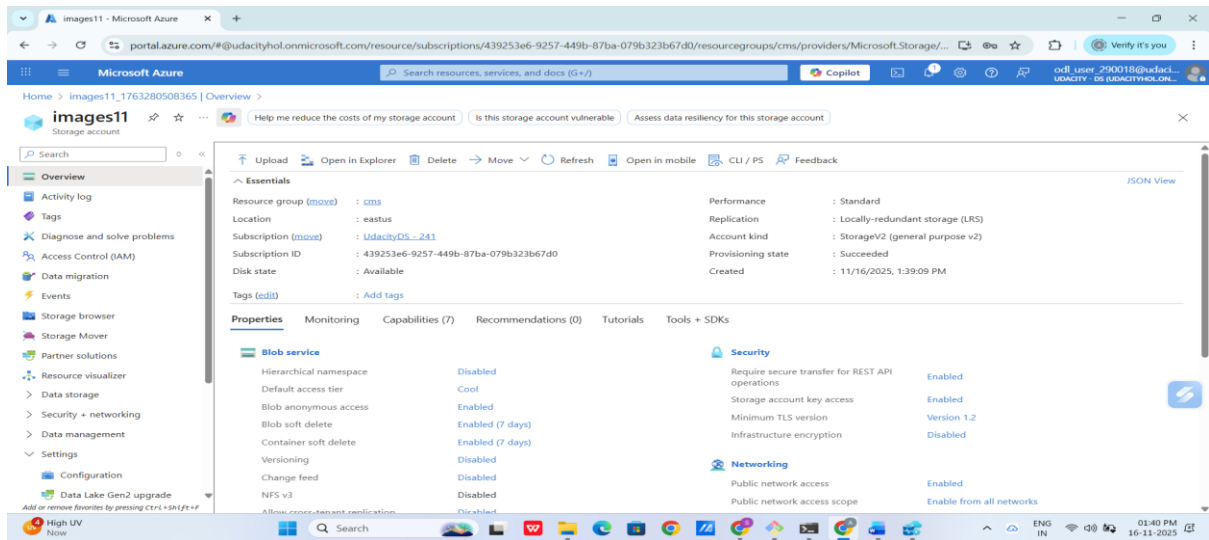
Type: Azure SQL DB

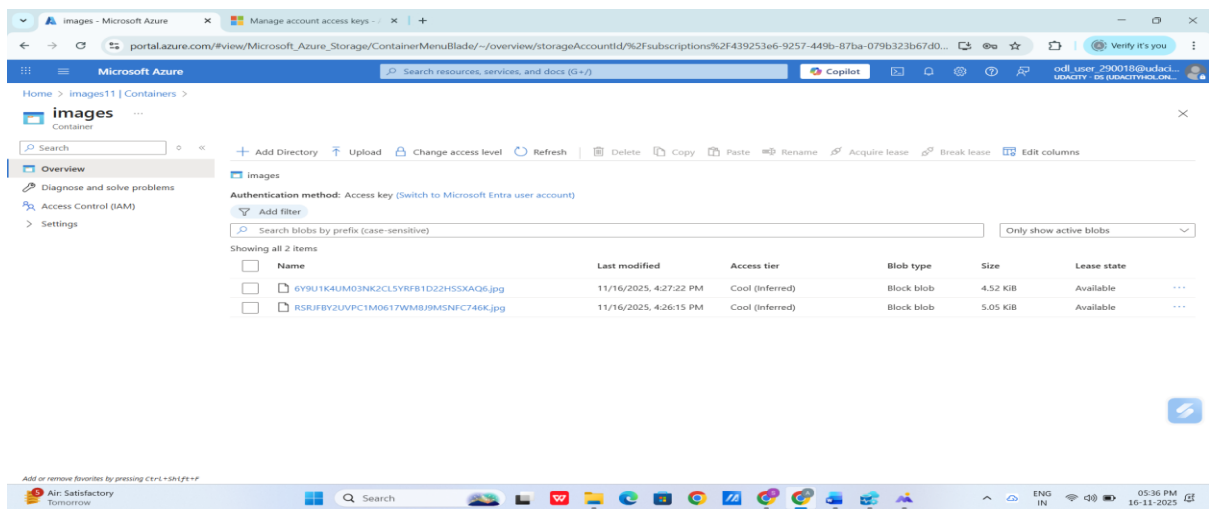
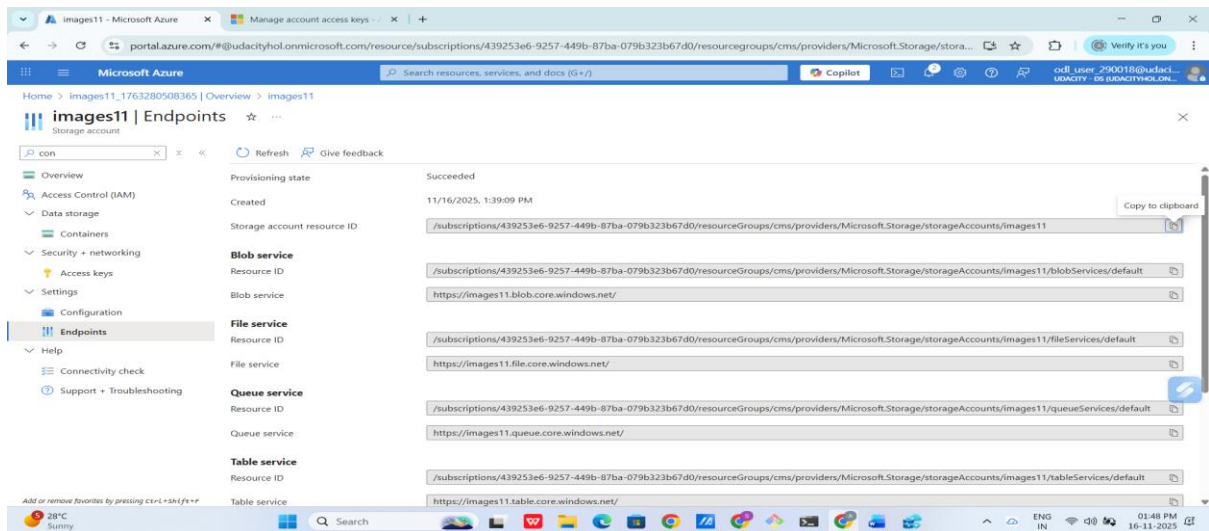
Search

Name	Status	Size (MB)	Actions
cms	ONLINE		...
master	ONLINE		...

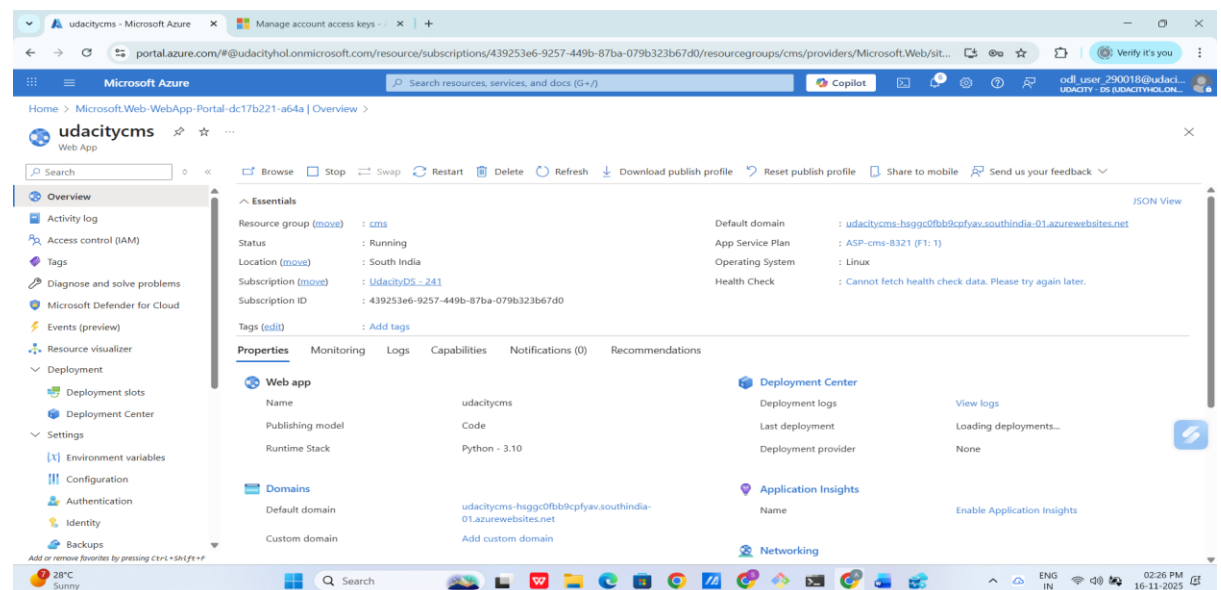


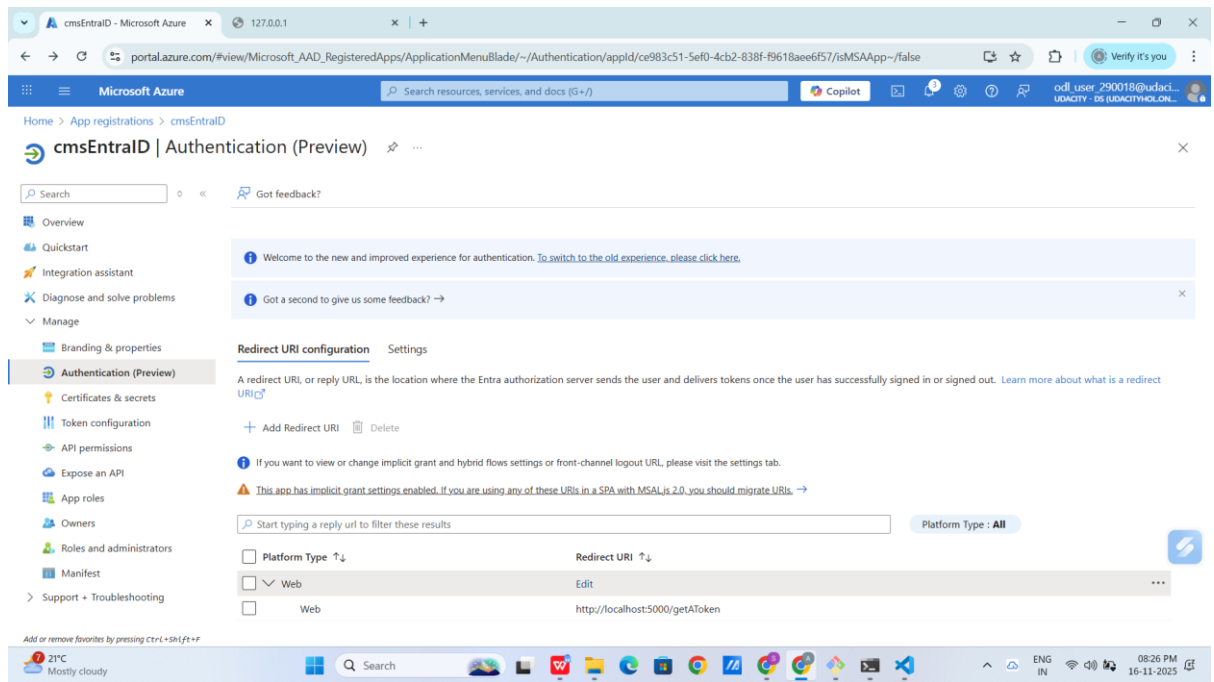
4. A screenshot showing an example of blob endpoints where images are sent for storage (see example in the project repository).



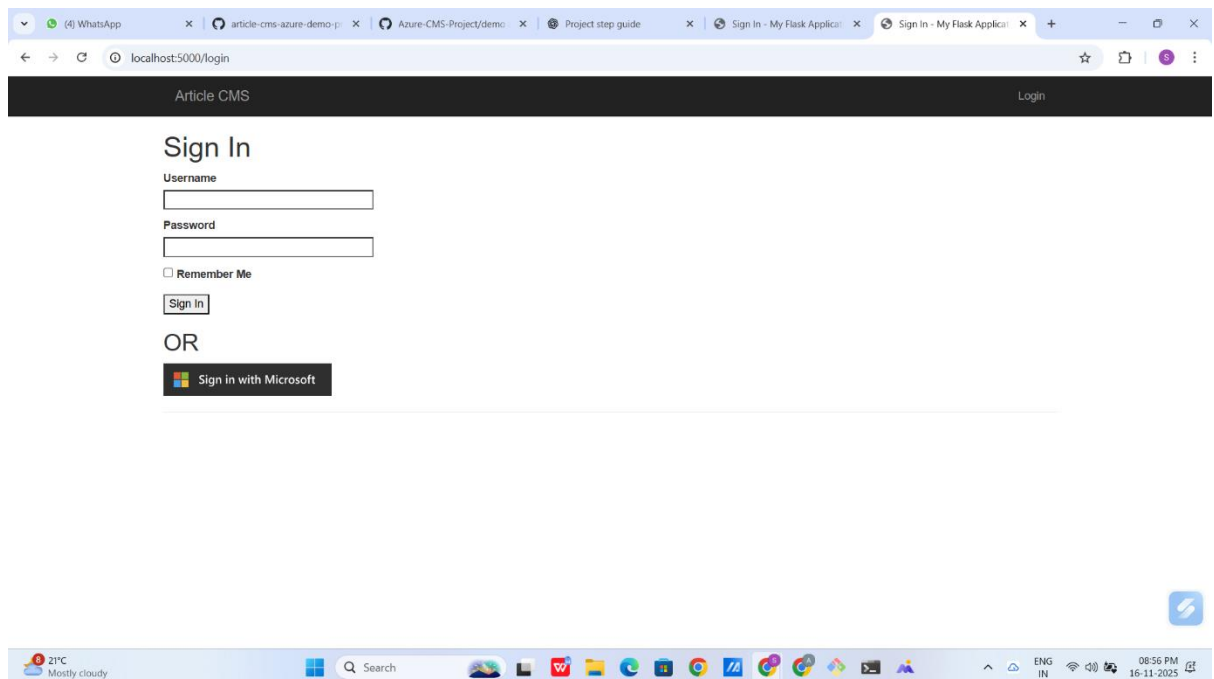


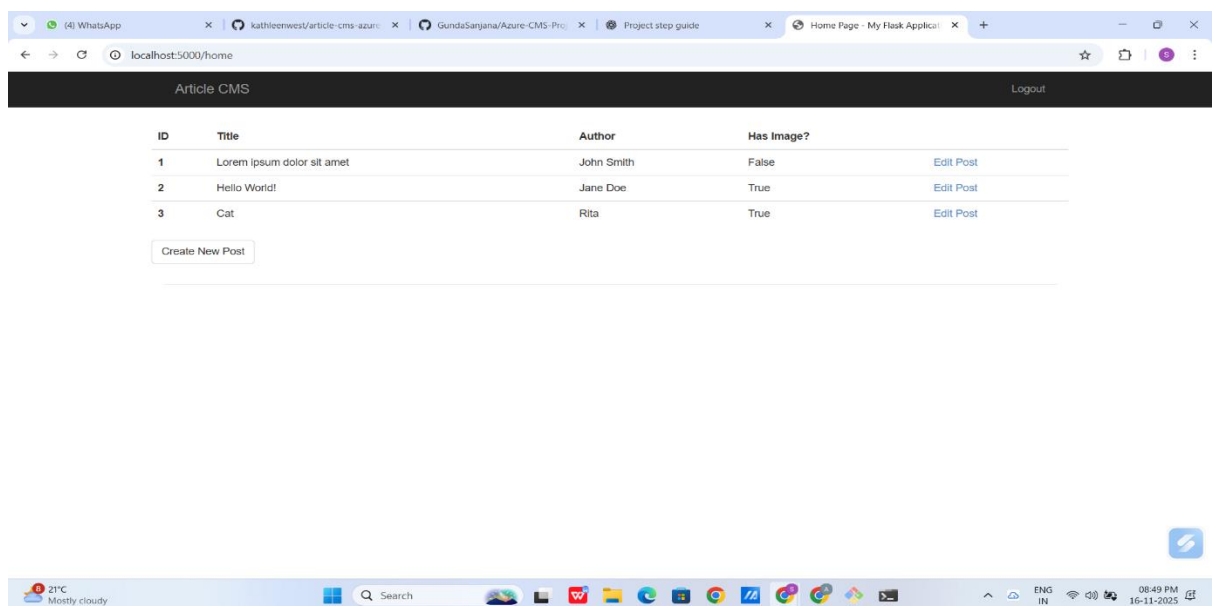
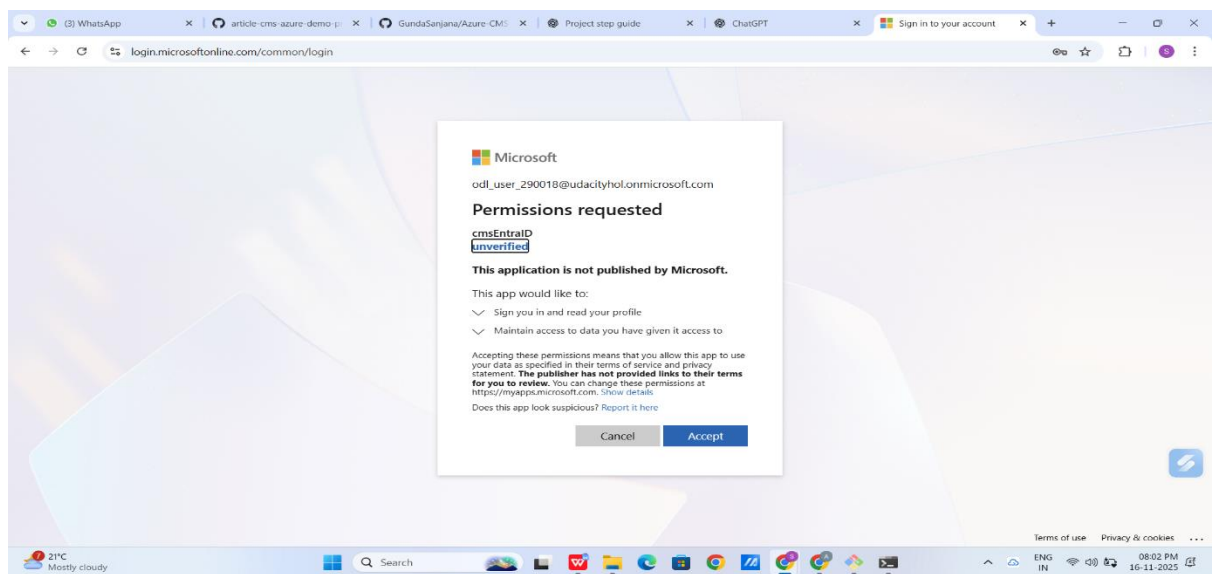
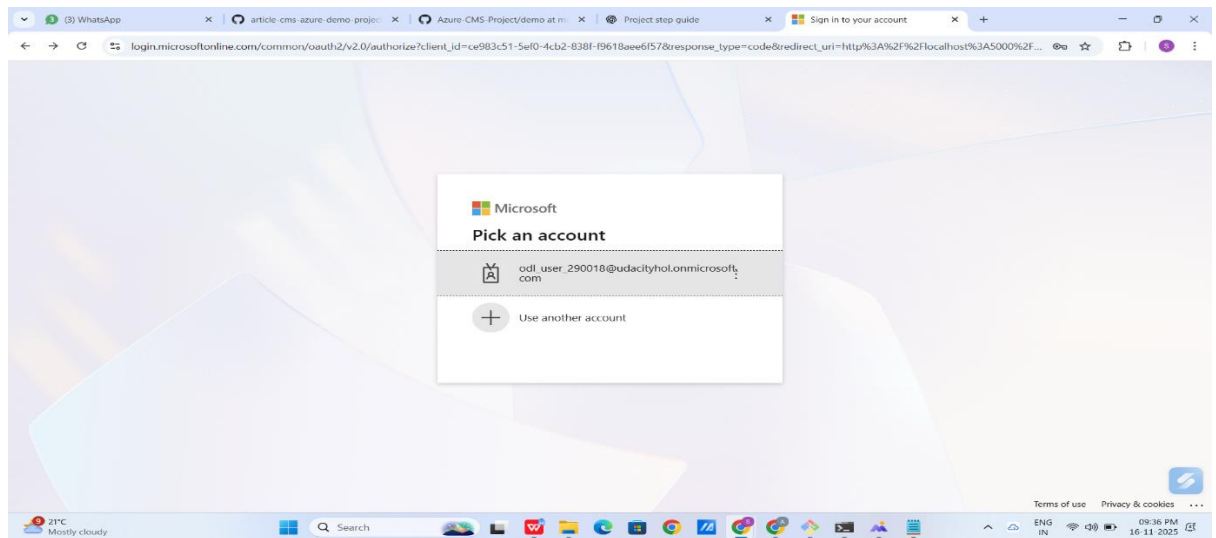
5. A screenshot of the redirect URIs related to Microsoft authentication (see example in project repository).

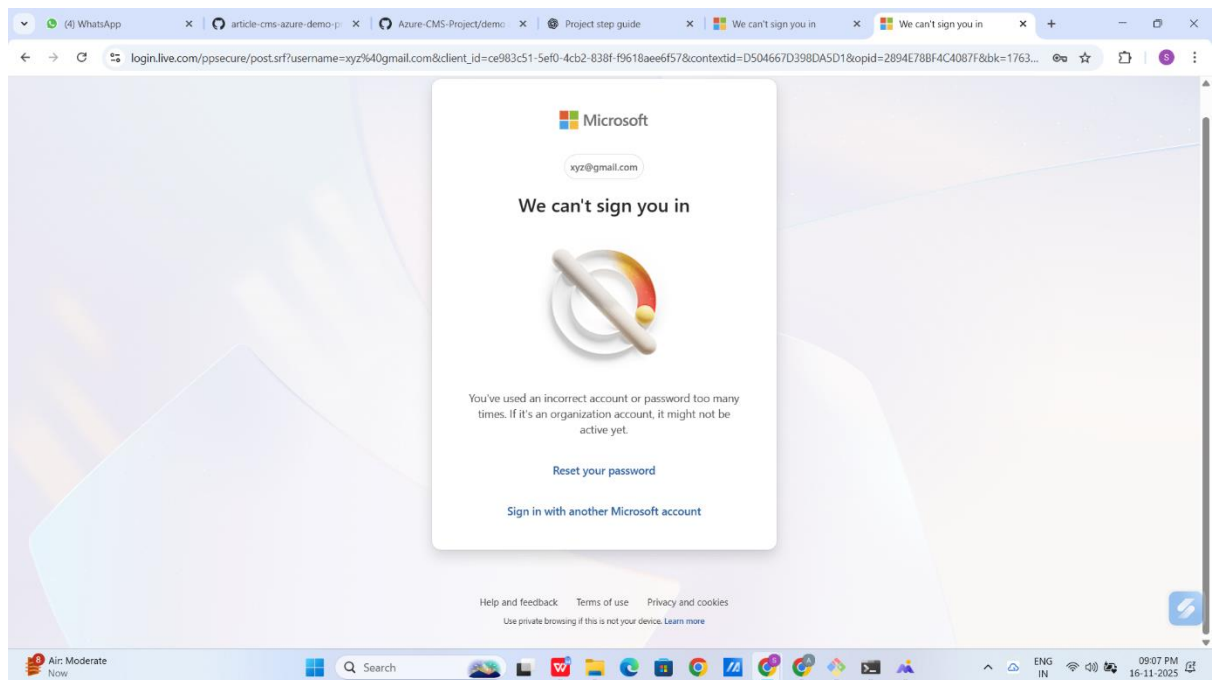
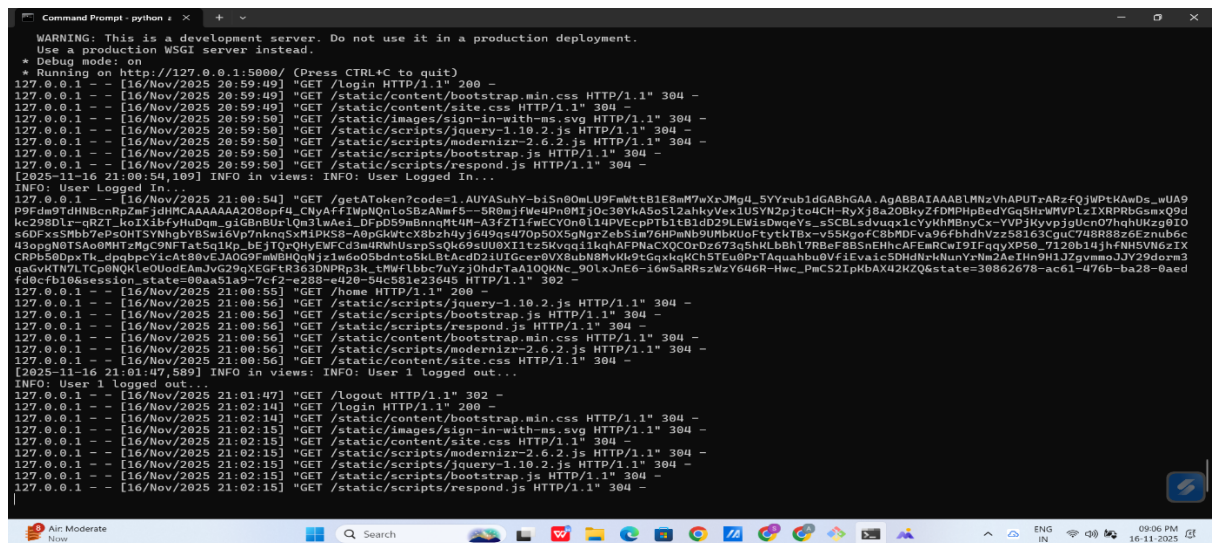
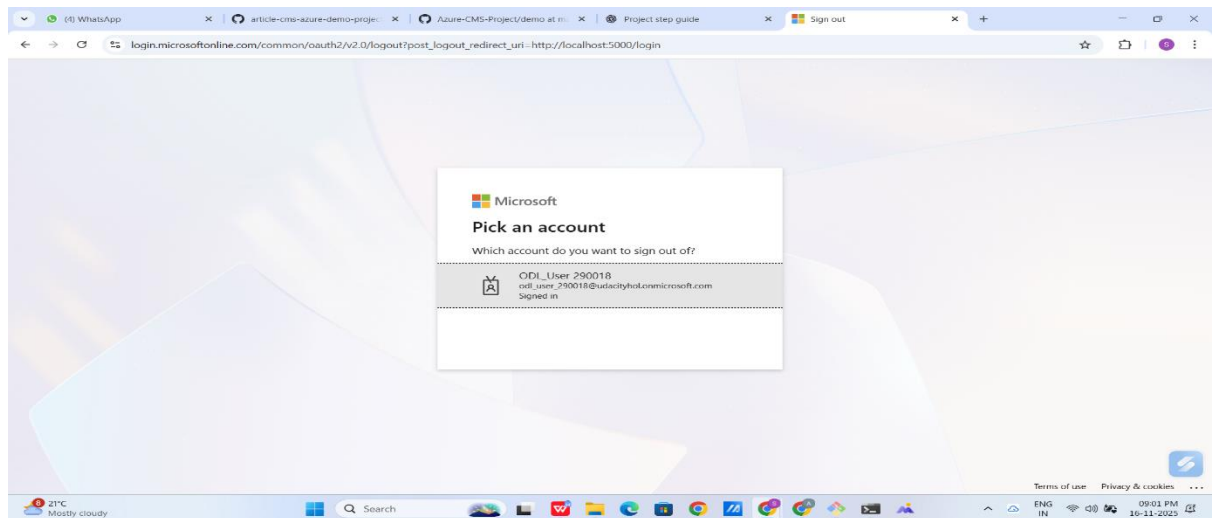


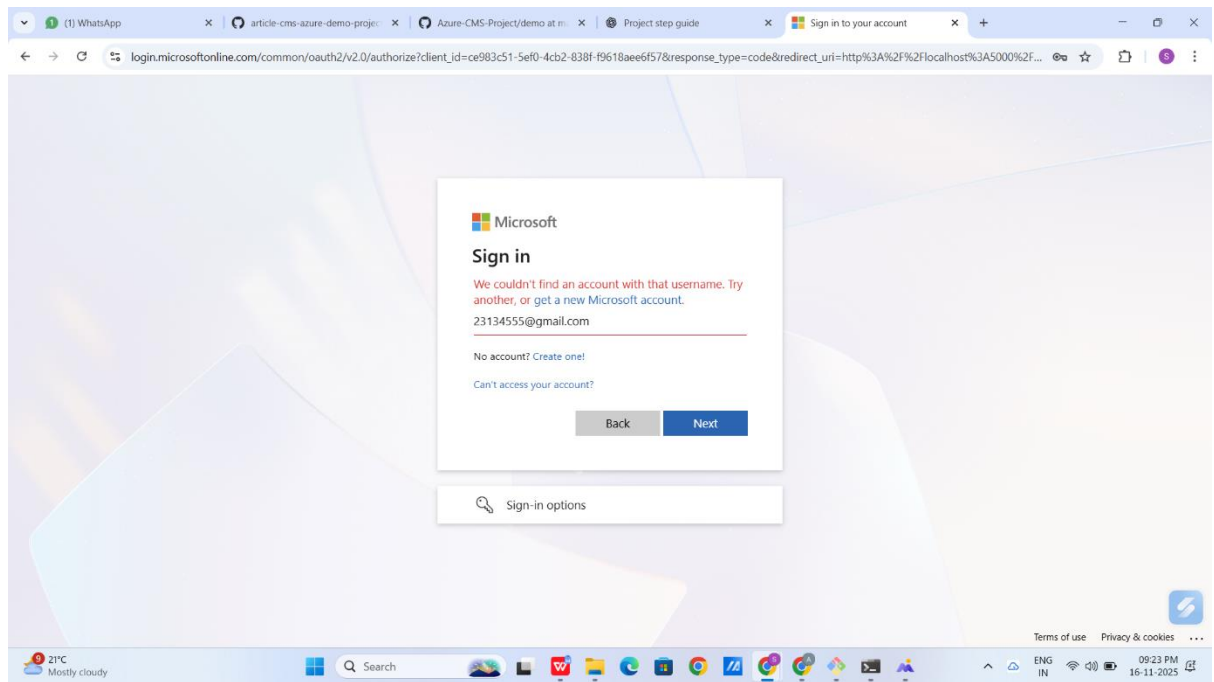


6. A screenshot showing one potential form of logging with an "Invalid login attempt" and "admin logged in successfully", taken from the app's Log stream or other logs you create and store (see example in project repository). You can customize your log messages as you see fit for these situations.









7. Your application code—most importantly `__init__.py` and `views.py` since they will contain your updates for Auth and Logging.

7.1 `__init__.py`:

```

"""
The flask application package.
"""

import logging
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager
from flask_session import Session
from flask.logging import create_logger

app = Flask(__name__)
app.config.from_object(Config)
# TODO: Add any logging levels and handlers with app.logger
# DONE (Note: Instructor's original app.logger code not pylint compliant)
LOG = create_logger(app)
LOG.setLevel(logging.INFO)
streamHandler = logging.StreamHandler()
streamHandler.setLevel(logging.INFO)
LOG.addHandler(streamHandler)

Session(app)
db = SQLAlchemy(app)
login = LoginManager(app)
login.login_view = 'login'

```

```
import FlaskWebProject.views
```

7.2 views.py:

```
"""
```

```
Routes and views for the flask application.
```

```
"""
```

```
from datetime import datetime
from flask import render_template, flash, redirect, request, session, url_for
from werkzeug.urls import url_parse
from config import Config
from FlaskWebProject import app, db
from FlaskWebProject.forms import LoginForm, PostForm
from flask_login import current_user, login_user, logout_user, login_required
from FlaskWebProject.models import User, Post
from FlaskWebProject import LOG
import msal
import uuid
```

```
imageSourceUrl = 'https://' + app.config['BLOB_ACCOUNT'] + '.blob.core.windows.net/' +
app.config['BLOB_CONTAINER'] + '/'
```

```
@app.route('/')
@app.route('/home')
@login_required
def home():
    user = User.query.filter_by(username=current_user.username).first_or_404()
    posts = Post.query.all()
    return render_template(
        'index.html',
        title='Home Page',
        posts=posts
    )
```

```
@app.route('/new_post', methods=['GET', 'POST'])
@login_required
def new_post():
    form = PostForm(request.form)
    if form.validate_on_submit():
        post = Post()
        post.save_changes(form, request.files['image_path'], current_user.id, new=True)
        # LOG Informational
        LOG.info('INFO: New post added by user: ' + str(current_user.id))
        return redirect(url_for('home'))
    return render_template(
        'post.html',
        title='Create Post',
        imageSource=imageSourceUrl,
```

```

        form=form
    )

@app.route('/post/<int:id>', methods=['GET', 'POST'])
@login_required
def post(id):
    post = Post.query.get(int(id))
    form = PostForm(formdata=request.form, obj=post)
    if form.validate_on_submit():
        post.save_changes(form, request.files['image_path'], current_user.id)
        # LOG Informational
        LOG.info('INFO: Post ' + str(id) + ' edited by user: ' + str(current_user.id))
        return redirect(url_for('home'))
    return render_template(
        'post.html',
        title='Edit Post',
        imageSource=imageSourceUrl,
        form=form
    )

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        # LOG Informational
        LOG.info('INFO: User ' + str(current_user.id) + ' is authenticated...')
        return redirect(url_for('home'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            # LOG Unsuccessful login
            LOG.warning('WARNING: Login Unsuccessful....Invalid username or password for
user:%s' + str(user))
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        LOG.info('INFO: Local login successful for user: %s', user.username)
        next_page = request.args.get('next')
        if not next_page or url_parse(next_page).netloc != "":
            next_page = url_for('home')
        return redirect(next_page)
    session["state"] = str(uuid.uuid4())
    auth_url = _build_auth_url(scopes=Config.SCOPE, state=session["state"])
    return render_template('login.html', title='Sign In', form=form, auth_url=auth_url)

@app.route(Config.REDIRECT_PATH) # Its absolute URL must match your app's
redirect_uri set in AAD
def authorized():
    if request.args.get('state') != session.get("state"):
        return redirect(url_for("home")) # No-OP. Goes back to Index page
    if "error" in request.args: # Authentication/Authorization failure

```

```

    #LOG Error
    LOG.error('ERROR: Authentication/Authorization failure...')
    return render_template("auth_error.html", result=request.args)
if request.args.get('code'):
    cache = _load_cache()
    # TODO: Acquire a token from a built msal app, along with the appropriate redirect URI
    # DONE
    result = _build_msal_app(cache=cache).acquire_token_by_authorization_code(
request.args['code'],
scopes=Config.SCOPE,
redirect_uri=url_for('authorized', _external=True, _scheme='http'))
    if "error" in result:
        #LOG Error
        LOG.error('ERROR: Did not acquire a token for OAUTH...')
        return render_template("auth_error.html", result=result)
    session["user"] = result.get("id_token_claims")
    # Note: In a real app, we'd use the 'name' property from session["user"] below
    # Here, we'll use the admin username for anyone who is authenticated by MS
    user = User.query.filter_by(username="admin").first()
    login_user(user)
    _save_cache(cache)
    # LOG
    LOG.info('INFO: User Logged In...')
    return redirect(url_for('home'))

@app.route('/logout')
def logout():
    #LOG
    LOG.info('INFO: User ' + str(current_user.id) + ' logged out...')
    logout_user()
    if session.get("user"): # Used MS Login
        # Wipe out user and its token cache from session
        session.clear()
        # Also logout from your tenant's web session
        return redirect(
            Config.AUTHORITY + "/oauth2/v2.0/logout" +
            "?post_logout_redirect_uri=" + url_for("login", _external=True))

    return redirect(url_for('login'))

def _load_cache():
    # TODO: Load the cache from `msal`, if it exists
    # DONE
    cache = msal.SerializableTokenCache()
    if session.get('token_cache'):
        cache.deserialize(session['token_cache'])
    return cache

def _save_cache(cache):
    # TODO: Save the cache, if it has changed
    # DONE

```

```

if cache.has_state_changed:
    session['token_cache'] = cache.serialize()

def _build_msal_app(cache=None, authority=None):
    # TODO: Return a ConfidentialClientApplication
    # DONE
    return msal.ConfidentialClientApplication(
        Config.CLIENT_ID, authority=authority or Config.AUTHORITY,
        client_credential=Config.CLIENT_SECRET, token_cache=cache)

def _build_auth_url(authority=None, scopes=None, state=None):
    # TODO: Return the full Auth Request URL with appropriate Redirect URI
    # DONE
    return _build_msal_app(authority=authority).get_authorization_request_url(
        scopes or [],
        state=state or str(uuid.uuid4()),
        redirect_uri=url_for('authorized', _external=True, _scheme='http'))

```

8. Your WRITEUP.md file analyzing and explaining your choice between a VM or an App Service.

Write-up Template

Analyze, choose, and justify the appropriate resource option for deploying the app.

For **both**** a VM or App Service solution for the CMS app.**

- **Analyze costs, scalability, availability, and workflow**
- **Choose the appropriate solution (VM or App Service) for deploying the app**
- **Justify your choice**

a) Virtual Machine (VM)

Cost:

A VM is billed like a full server: you pay for the selected size (CPU, RAM, storage) as long as it's allocated, regardless of how busy the app is. For this CMS, a small VM like Standard B1ls is enough, but the cost is still higher than a free or low-tier App Service plan, especially if I forget to stop or delete the VM. I would also be responsible for managing OS patches and any extra software, which indirectly adds "operational cost" in terms of my time.

Scalability:

A single VM scales vertically: to handle more load, I'd have to resize it to a bigger SKU or manually add more VMs behind a load balancer. This is more complex to manage and usually requires planning and downtime. Auto-scaling is possible, but I have to configure it explicitly using scale sets or other Azure services.

Availability:

VM availability depends on how I configure it. A single VM has a single point of failure. To achieve higher availability, I'd need multiple VMs in an availability set or zone, plus a load balancer. I am responsible for OS updates, security patches, and restarts, which can cause downtime if not handled carefully.

Workflow:

Using a VM gives me full control over the OS and runtime stack. I can install any version of Python, configure Nginx/Gunicorn manually, and host other services on the same machine. However, deployment is more manual: SSH into the VM, clone the GitHub repo, create a virtual environment, install dependencies, configure Nginx, and manage services. This provides flexibility but also increases complexity and maintenance effort.

b) App Service (Web App)

Cost:

App Service offers a Free F1 tier, which is ideal for this project and development scenarios. I don't pay for idle VMs directly; instead, I pay per App Service plan, which can host multiple web apps. For small workloads like this article CMS, the free or basic tiers are cheaper than running a dedicated VM 24/7. Scaling up to higher tiers is still generally cost-effective compared to managing multiple VMs myself.

Scalability:

App Service has built-in horizontal scaling. I can increase the number of instances with a slider or configure auto-scale rules based on CPU, memory, or schedule. It abstracts away the individual servers, so I don't have to manage the underlying OS. This makes it easier to handle traffic spikes and future growth of the CMS without re-architecting the deployment.

Availability:

App Service is a managed platform (PaaS), so Microsoft handles OS patching, infrastructure redundancy, and many aspects of uptime. The service is integrated with Azure's availability features, and I can choose higher tiers with built-in SLA guarantees. I don't need to manually set up multiple machines or a load balancer just to get a highly available endpoint.

Workflow:

The developer workflow with App Service is very streamlined. I can connect the App Service to GitHub via Deployment Center so that pushes to a branch automatically deploy to Azure. Configuration like connection strings and secrets can be stored in Application Settings rather than hard-coded in the code. I don't manage the OS, web server, or runtime directly; I only focus on the app code and settings. This reduces operational overhead and speeds up development.

VM or App Service? (and why):

For this project, I chose to deploy the Flask CMS using Azure App Service (Web App).

The main reasons are:

Cost – The App Service free tier is enough for this small CMS, so I don't pay for a dedicated VM. It's more economical for a learning project and small workloads.

Simplicity & Workflow – App Service gives me an easy deployment pipeline via GitHub and a simple configuration experience through Application Settings. I don't have to manage the OS, patching, or web server directly, which lets me focus on the Flask app itself.

Built-in scalability & availability – If the CMS ever needs to handle more users or traffic, I can scale up or out using the App Service plan without redesigning the infrastructure or manually configuring load balancers and extra VMs.

Because of these factors, App Service is a better fit for this application than a VM for my current needs.

Assess app changes that would change your decision.

If the CMS application changed in ways that demanded more low-level infrastructure control, my choice might switch toward a VM (or even a container-based solution). For example, if I needed custom OS-level software, unusual networking requirements, or background services that App Service doesn't support easily, a VM would give me full control to install and configure everything exactly as needed.

Similarly, if the application expanded into a multi-component system (e.g., additional microservices, custom message queues, or specialized database engines) that needed tight control over the environment and network topology, managing everything on VMs (or using containers orchestrated by services like AKS) might become more suitable. In that case, I would accept the extra operational complexity of VMs in exchange for the flexibility they provide.

Right now, though, the CMS is a straightforward web app that primarily needs a Python runtime, connection to Azure SQL and Blob Storage, and an OAuth2 login flow. For that scenario, App Service remains the most practical and maintainable option.