Cloud Deployment & Maintenance Guide: Therapy AI – Cloud (Azure)

This guide outlines the steps to deploy your Python-based Therapy AI application (Flask + Celery) to Azure App Service using a single Docker container. It also covers Firebase Cloud Messaging (FCM) setup and other cloud infrastructure notes.

Disclaimer: Azure constantly changes commands, UI elements and names. If the following installation guide fails in relation to the Azure parts, please troubleshoot with Microsoft\Azure docs.

Table of Contents:

- Updating the Application
- Scaling
- Common Issues

10. **Future Considerations (Post-POC)**

1. Prerequisites

- **Azure Account:** An active Azure subscription.
- **Azure CLI:** Installed and configured. (Install Azure CLI)
- **Docker Desktop:** Installed and running. (Install Docker Desktop)
- **Python & Pip:** Python 3.9 (as per Dockerfile) installed locally for testing.
- **Git:** For version control.
- **Project Code:** Your application code as provided.
- **Code Editor:** VS Code, PyCharm, or your preferred editor.

2. Project Structure Overview

Your project consists of (more detailed description available in the developer guide):

- **app.py**: Main Flask application entry point, initializes Flask, JWT, Celery, and registers blueprints. Handles Firebase credential setup.
- **celery_app.py**: Celery application definition and configuration.
- **config.py**: Centralized configuration, reads from environment variables.
- **requirements.txt**: Python dependencies.
- **Dockerfile**: Defines the container image, including supervisor for running Flask (via Gunicorn) and Celery.
- **supervisord.conf**: Configuration for supervisor to manage Gunicorn and Celery processes.
- **auth/**: Authentication routes and utilities.
- **profiles/**: User profile related routes.
- **sessions/**: Session management routes (/upload, /details, etc.) and Celery tasks (tasks.py).
- **devices/**: Device registration routes for FCM.
- **data/**: Routes for handling pending transcript data and submission (/pending, /transcript).
- **database/cosmos.py**: Cosmos DB interaction logic.
- **notifications/fcm.py**: FCM sending logic.
- **seed_database.py**: Script to populate Cosmos DB with initial data (not real data).
- **test_api_flow.py**: API integration test script (initial system check on fake seed data)

3. Firebase Cloud Messaging (FCM) Setup

1. **Firebase Project:**
    - Go to the Firebase Console.
    - Create a new project or use an existing one.
2. **Enable Cloud Messaging:**
    - In your Firebase project, ensure Cloud Messaging API (Legacy) and Firebase Cloud Messaging API (V1) are enabled. Usually, they are by default. (You can check this in Google Cloud Console under "APIs & Services" for your Firebase project's linked Google Cloud project).
3. **Generate a Service Account Key:**
    - In the Firebase Console, go to "Project settings" (gear icon) -> "Service accounts".
    - Select "Python" and click "Generate new private key". If you are missing a service account, you will be prompted to create one, and you need to do so.
    - A JSON file will be downloaded (e.g., your-project-id-firebase-adminsdk-xxxxx-xxxxxxxxxx.json). **This is the file your GOOGLE_APPLICATION_CREDENTIALS will point to conceptually.** place it in the root cloud dir and rename it to "firebase_service_account.json" if you wish to use in a local development setting.
    - **Secure this file!** Do not commit it to Git directly.
4. **Android/iOS Client Setup:** Your mobile application will need its own FCM setup (google-services.json for Android, GoogleService-Info.plist for iOS) to receive messages and obtain a device registration token. This guide focuses on the server-side. Refer to the android application side to complete this step.

4. Local Environment Setup (Brief)

1. **Open Cloud Dir.**
2. **Create a Python virtual environment:**
   - python -m venv venv
   - source venv/bin/activate # Linux/macOS
     venv\Scripts\activate # Windows
3. **Install dependencies:**
   pip install -r requirements.txt
4. **Create a .env file** in your project root for **local development** (this file is NOT deployed to Azure; settings are managed via App Service Application Settings):
   # .env (Example for LOCAL DEVELOPMENT - DO NOT COMMIT SENSITIVE DATA)

   - FLASK_SECRET_KEY=your_flask_secret_key
   - JWT_SECRET_KEY=your_jwt_secret_key
   - COSMOS_URI=your_cosmos_db_uri
   - COSMOS_KEY=your_cosmos_db_primary_key
   - SPEECH_KEY=your_azure_speech_service_key
   - SPEECH_REGION=your_azure_speech_service_region
   - TEXT_ANALYTICS_KEY=your_azure_text_analytics_key
   - TEXT_ANALYTICS_ENDPOINT=your_azure_text_analytics_endpoint
   - AZURE_STORAGE_CONNECTION_STRING="your_azure_storage_connection_string"
   - AUDIO_UPLOAD_BLOB_CONTAINER_NAME=audio-uploads

   # For local Redis (if not using Azure Cache for Redis locally)

   - REDIS_HOST=localhost
   - REDIS_PORT=6379
   - REDIS_PASSWORD=
   - REDIS_USE_SSL=False # Important for local non-SSL Redis

   # For FCM

- FIREBASE_PROJECT_ID=your-firebase-project-id
- GOOGLE_APPLICATION_CREDENTIALS=firebase_service_account.json
- GUNICORN_WORKERS=2
- CELERY_CONCURRENCY=2

5. Place your `firebase_service_account.json in the cloud root directory for local testing if using the `GOOGLE_APPLICATION_CREDENTIALS` path method.
THIS WILL NOT BE USED BY THE CLOUD!!! PLEASE FOLLOW SECTION 6 OF THIS GUIDE!!!

5. Azure Resource Provisioning

The Azure cloud MUST include the following services, please make sure to:

1. set up a subscription to your account.
2. create a resource group for all the services.
3. create an Azure Container Registries - ACR (This guide assumes Admin User is allowed as access key).
4. create Azure Cache for Redis.
5. create Azure Cosmos DB (NoSQL).
6. create Azure Blob Storage.
7. create Azure Cognitive Services - Speech Service.
8. create Azure Cognitive Services - Language Service (Text Analytics).
9. create Azure App Service.

Key points:

- It is recommended to use lowercase letters when choosing the name for ACR.
- When creating the ACR, push the image (code) in order to be able to create the app service, see section 7 of this guide.
- While different ways are possible, this guide assumes we will push images where Admin User is allowed as an access key. To do so, after you create ACR, go to ACR -> settings\access keys -> enable Admin User.
- The App Service is a linux container in this guide, we also assume you will use admin credentials as authentication.
- This project uses access keys to access the services, ensuring that access keys are **not disabled** (e.g. redis, by default, sometimes disables them) is a must.
  This setting is usually located in the authentication\access keys tab of the service, make sure that access keys are **Enabled.** If certain services don't work in the app, this might be the cause.

All of the above steps can be done using the Azure Portal. All the information for each service and the fields when building the service can be found in the service page you are building in the Azure Portal.

Please build and adjust the services in accordance to desired functionalities. Add VLANs, private DNS zones and other services and settings for desired security levels, restrictions and more advanced functionalities
We, the writers of this guide, were not able to expand upon other settings because of restrictions on our azure accounts.

6. Application Configuration in Azure

Your config.py reads settings from environment variables. In Azure App Service, these are set as "Application settings".

1. Navigate to your **Web App (YourUniqueWebAppName)** in the Azure portal.
2. Go to **"Settings"** -> **"Environment variables"**.
3. Click **"+ Add"** under app settings for each of the following (get values from your provisioned Azure resources):

Name	Example Value / Source	Notes
FLASK_SECRET_KEY	Source: Generate a strong random string. At your own discretion	For Flask sessions, CSRF.
JWT_SECRET_KEY	Source: Generate another strong random string. At your own discretion	For JWT signing.
FLASK_DEBUG	False	Always False in production/staging.
PORT	8000	Internal port Gunicorn binds to; App Service maps 80/443 to this.
PYTHONUNBUFFERED	1	Ensures logs appear promptly.
COSMOS_URI	Source: URI From Cosmos DB account in the Overview section.	should look like: https://YourUniqueCosmosDBAccountName.documents.azure.com:443/

COSMOS_KEY	Source: PRIMARY KEY from Cosmos DB account "Keys" blade in the azure portal.	
SPEECH_KEY	Source: KEY 1 from Speech Service "Keys and Endpoint" blade in the azure portal.	
SPEECH_REGION	Source: Location/Region from Speech Service "Keys and Endpoint" blade.	
TEXT_ANALYTICS_KEY	Source: KEY 1 from Language Service "Keys and Endpoint" blade.	
TEXT_ANALYTICS_ENDPOINT	Source: Endpoint from Language Service "Keys and Endpoint" blade.	
AZURE_STORAGE_CONNECTION_STRING	Source: Connection string from azure storage account Access Keys blade.	
AUDIO_UPLOAD_BLOB_CONTAINER_NAME	audio-uploads	Name for the container at your own discretion.
REDIS_HOST	Source: Host name from Azure Cache for Redis "Properties" blade.	should look like: YourUniqueRedisCacheName.redis.cache.windows.net
REDIS_PORT	6380 (for SSL) or 6379 (if non-SSL enabled)	Recommended to use SSL. 6379 might cause errors and security risks.

REDIS_PASSWORD	Source: Primary from Azure Cache for Redis "Authentication->Access keys" blade.	
REDIS_USE_SSL	True or False	if REDIS_PORT=6380 set True, otherwise False.
FIREBASE_PROJECT_ID	Source: Project id from Firebase project settings.	
FIREBASE_SERVICE_ACCOUNT_JSON_CONTENT	<Paste the entire content of your service account JSON file here>	**Crucial for FCM.**
GUNICORN_WORKERS	2 (or adjust based on App Service Plan SKU)	For supervisord.conf.
CELERY_CONCURRENCY	2 (or adjust based on App Service Plan SKU)	For supervisord.conf.
GUNICORN_THREADS	4 (optional override)	For supervisord.conf.
GUNICORN_TIMEOUT	300 (optional override)	For supervisord.conf.

4. **Save** the Application Settings. This will usually trigger a restart of your App Service.

Handling firebase_service_account.json:
app.py includes logic to read FIREBASE_SERVICE_ACCOUNT_JSON_CONTENT which contains the contents of the GOOGLE_APPLICATION_CREDENTIALS.json file. paste the contents of GOOGLE_APPLICATION_CREDENTIALS.json onto the key's value.

7. Building and Deploying the Docker Image

- In this section, the following are placeholders:
  YourUniqueACRName - ACR service name
  therapyapp - Image name
  latest - Tag name
- Run these commands from the cloud dir (./cloud/ relative to the project root).

1. **Ensure your Dockerfile and supervisord.conf are in the cloud dir.**
2. **Login to your ACR (if not already):**

az login

az acr login --name YourUniqueACRName

3. **Build the Docker image:**

docker build -t YourUniqueACRName.azurecr.io/therapyapp:latest .

4. **Push the image to ACR:**

docker push YourUniqueACRName.azurecr.io/therapyapp:latest

5. **Deploy to App Service:**
   - When you push a new image with the same tag (latest in this case) that the Web App is configured to use, you often need to **Restart** the Web App from the Azure portal for it to pull the new image version.
   - Alternatively, you can set up a webhook in ACR to trigger the App Service on new image pushes (Continuous Deployment). For now, a manual restart is fine for POC.

Common Issues:

- Use lowercase letters when replacing the placeholder YourUniqueACRName, as it may cause problems when trying to push the image to the cloud.

8. Database Seeding

Your seed_database.py script populates Cosmos DB, **ADJUST IT TO NEEDED DATA**!
Currently the script holds mock data.
If adapting means changing the keys in the entries, make sure to update the android application \ cloud code accordingly.

How to run: Set up all necessary Azure environment variables for the DB in a local .env (create .env if not present and place it in the cloud dir). Specifically make sure that the following keys are present in the .env file:

```
AUDIO_UPLOAD_BLOB_CONTAINER_NAME
AZURE_STORAGE_ACCOUNT_NAME
COSMOS_KEY
COSMOS_URI
```

Use section 6 for reference for the above variable. Then run:

```
python seed_database.py
```

Make sure all the needed python dependencies are present for the script. one can download the project libraries using:

```
pip install -r requirements.txt
```

Important: The current script will create users with the default password:

```
6!'DA;ozi5j2
```

test_api_flow.py uses this and assumes that the 4 hardcoded users in the seed script appear in the DB. If any alterations are made to the seed file, the test_api_flow.py might not work.

9. Maintenance and Troubleshooting

- **Viewing Logs:**
  We recommend using **Log Stream** for quick testing and **App Service Logs** for deeper understanding of the runs.
    - **Log Stream:** In Azure portal -> Your Web App -> "Monitoring" -> "Log stream". Can be unreliable at times.
    - **App Service Logs (to Blob/Filesystem):** In Azure portal -> Your Web App -> "Monitoring" -> "App Service logs". Configure "Application Logging (Blob)" to store container stdout/stderr persistently in Azure Blob Storage. This is the most reliable way but requires more work on set up.
    - **Kudu (Advanced Tools):**
        - URL: https://YourUniqueWebAppName.scm.azurewebsites.net
        - Navigate to "Debug console" -> "CMD" or "PowerShell".
        - Logs are typically in LogFiles/docker/.
    - **Diagnostic Settings (to Log Analytics):** For advanced querying and monitoring, send AppServiceConsoleLogs to a Log Analytics workspace.
- **Updating the Application:**
    0. Make code changes.
    1. Rebuild the Docker image with a new tag (e.g., :v1.0.1) or update latest.
    2. Push the image to ACR.
    3. Update the image tag in your Web App's "Deployment Center" or "Container settings", or simply restart the Web App if using the latest tag (though specific tags are better for rollbacks).
- **Scaling:**
    0. **Vertical Scaling:** Increase the tier of your App Service Plan (e.g., from B1 to B2, or to a Standard tier S1). This gives more CPU/memory to the single container running both Flask and Celery.
    1. **Horizontal Scaling (Not ideal for this single-container setup):** Scaling out App Service instances would mean multiple containers, each running Flask AND Celery. This isn't efficient for Celery (multiple workers might fight for

same tasks without proper Celery setup for distributed workers, and it's not true independent scaling).

- **Common Issues:**

0. **Environment Variable Misconfiguration:** Double-check names and values in App Service Application Settings.
1. **Secrets Not Loaded:** Ensure FIREBASE_SERVICE_ACCOUNT_JSON_CONTENT is correctly set and the app.py logic writes the file.
2. **Port Mismatches:** supervisord.conf uses PORT env var (set by Azure). Gunicorn binds to this.
3. **Celery Can't Connect to Redis:** Check Redis host, port, password, and SSL settings in Azure App Settings and config.py. Test Redis connectivity (your /test-socket-to-redis route is good for checking Gunicorn's view, but Celery's view matters too).
4. **Container Startup Failures:** Check Kudu Docker logs immediately.
5. **Service Names Don’t Match Guide**: See if services changed their naming scheme since the writing of this guide.
6. **Resource Exhaustion:** If your App Service Plan is too small (e.g., Free or small Basic tier), Gunicorn and Celery together might consume all memory/CPU, leading to crashes. Monitor metrics.
7. **Disabled Access key usage:** Some services disable by default the usage of access keys with the service (e.g. redis), this can lead to problems in the app, please ensure that the services allow the usage of access keys if such option exists
8. **Restarts**: azure services onw the free tear have a lot of shortcomings, delays, and the need to restart. If the app doesn't work with the cloud of timeouts or bad requests all of the sudden, please restart the cloud service, reopen the app and try again.

11. Future Considerations

For a production-ready system, you should evolve beyond the single-container setup:

1. **Separate Flask and Celery:**
    - **Flask/Gunicorn:** Run in Azure App Service or Azure Container Apps.
    - **Celery Workers:** Run as separate Azure Container Apps (recommended for long-running background tasks) or Azure Container Instances. This allows independent scaling.
2. **CI/CD Pipeline:** Use Azure DevOps or GitHub Actions to automate building, testing, and deploying your application and infrastructure (using Infrastructure as Code like Bicep or ARM templates).
3. **Azure Key Vault:** Store all secrets (API keys, connection strings, passwords) securely.
4. **Managed Identities:** Use Managed Identities for Azure resources to authenticate to other Azure services (Cosmos DB, Blob, Key Vault, ACR) instead of storing keys/connection strings in App Settings.
5. **Application Insights:** Integrate for comprehensive monitoring, performance tracking, and distributed tracing.
6. **VNet Integration:** Secure your services within a Virtual Network.
7. **Load Balancing/Traffic Management:** Azure Front Door or Application Gateway if needed.
8. **Celery Monitoring:** Tools like Flower (can be run as another container) for Celery task monitoring.