Therapy AI - Developer's Guide for Updates & Modifications for azure cloud

This guide is intended for developers working on the Therapy AI project, the cloud (azure) side of the project. It outlines the process for making updates, adding features, and maintaining the existing codebase.

Table of Contents:

    - Using test_api_flow.py
    - Unit Tests (Future Enhancement)
7. **Building & Deploying Changes to Azure (POC)**
8. **Logging & Debugging**
    - Local Logging
    - Azure Log Inspection
9. **Coding Standards & Best Practices**
10. **Managing Dependencies**

1. Introduction & Project Overview

- **Purpose:** This guide helps developers understand how to contribute to the Therapy AI project, make changes safely, and deploy updates to the Azure POC environment.
- **Tech Stack:**
    - **Backend:** Python, Flask (API), Celery (background tasks)
    - **Database:** Azure Cosmos DB (SQL API)
    - **Messaging/Task Queue:** Azure Cache for Redis (for Celery)
    - **File Storage:** Azure Blob Storage
    - **AI Services:** Azure Cognitive Services (Speech-to-Text, Text Analytics for Sentiment)
    - **Notifications:** Firebase Cloud Messaging (FCM)
    - **Deployment (POC):** Docker, Azure App Service (Web App for Containers) with supervisor
- **Core Functionality:** The application allows therapists to upload audio sessions, processes them for speech-to-text and sentiment analysis, and provides an interface for review and management.

2. Prerequisites & Local Setup

Software Requirements:

- Git
- Python 3.9 (match Dockerfile)
- pip (Python package installer)
- Docker Desktop (for building/running the container locally if needed)
- Azure CLI (for interacting with Azure resources if managing them)
- A code editor (e.g., VS Code, PyCharm).

Cloning the Repository:

```
1. git clone <your-repository-url>
2. cd <repository-directory>
```

Setting up the Python Virtual Environment:

```
1. python -m venv venv
2. source venv/bin/activate  # Linux/macOS
3. # venv\Scripts\act+++ivate    # Windows
4. pip install -r requirements.txt
```

Local Configuration (.env file):

Create a .env file in the project root. This file is for **local development only** and should be in your .gitignore.

Populate it with necessary local or development Azure service credentials. Example:

```
 1.# .env (LOCAL DEVELOPMENT ONLY - DO NOT COMMIT TO GIT)
 2. FLASK_SECRET_KEY=your_local_flask_secret_key_for_dev
 3. JWT_SECRET_KEY=your_local_jwt_secret_key_for_dev
 4. FLASK_DEBUG=True # Enable Flask debug mode locally
 5.
 6. # --- Local Redis (if you run Redis locally via Docker or directly) ---
 7. REDIS_HOST=localhost
 8. REDIS_PORT=6379
 9. REDIS_PASSWORD=
10. REDIS_USE_SSL=False # For local non-SSL Redis
11.
12. # --- Or use a DEV Azure Cache for Redis ---
13. # REDIS_HOST=your-dev-redis.redis.cache.windows.net
14. # REDIS_PORT=6380
15. # REDIS_PASSWORD=your-dev-redis-key
16. # REDIS_USE_SSL=True
17.
```

```
# --- Azure Services (use DEV instances if available, or production for read-only if careful) ---
COSMOS_URI=your_cosmos_uri
COSMOS_KEY=your_cosmos_key
SPEECH_KEY=your_speech_key
SPEECH_REGION=your_speech_region
TEXT_ANALYTICS_KEY=your_text_analytics_key
TEXT_ANALYTICS_ENDPOINT=your_text_analytics_endpoint
AZURE_STORAGE_CONNECTION_STRING="your_storage_connection_string"
AUDIO_UPLOAD_BLOB_CONTAINER_NAME=audio-uploads-dev # Consider a dev container

# --- Firebase ---
FIREBASE_PROJECT_ID=your-firebase-project-id
GOOGLE_APPLICATION_CREDENTIALS=./secrets/firebase_service_account.json # Path to your dev service account key
# Note: For local, the file path is fine. For Azure, use FIREBASE_SERVICE_ACCOUNT_JSON_CONTENT.

# --- Optional Gunicorn/Celery overrides for local testing if not using supervisor directly ---
# (These are mainly for the supervisor config in deployed container)

```

Place your development firebase_service_account.json in ./secrets/.

Running Locally:

- **Flask Development Server:**

```
flask run # Uses app.py if FLASK_APP is set, or set FLASK_APP=app.py
# Or: python app.py (if __name__ == '__main__' block is present)

```

The Flask dev server will run (usually on port 5000 unless FLASK_PORT is set and used by app.run).

Celery Worker (in a separate terminal):

```
celery -A celery_app.celery worker -l info --pool=solo # --pool=solo can be easier for local debugging
```Ensure your `.env` file is configured for Celery to connect to your local or dev Redis instance.
```
```

3. Development Workflow

Version Control (Git): All changes must be tracked using Git.

Branching Strategy:

- main (or master): Represents the production-ready code. Direct commits should be avoided.
- develop: Integration branch for features before they go to main.

Feature Branches: Create a new branch from develop for each new feature or bugfix (e.g., feature/new-reporting-endpoint, fix/login-bug).

```
1. git checkout develop
2. git pull origin develop
3. git checkout -b feature/your-feature-name
```

- **Hotfix Branches:** Branched from main for urgent production fixes, then merged back into main AND develop.

Making Code Changes:

- Implement your feature or fix on your feature branch.
- Follow coding standards (see section 8).
- Write/update tests for your changes.

Committing Changes:

- Use clear, descriptive commit messages.
- Reference issue numbers if applicable (e.g., git commit -m "feat: Add X endpoint for Y (closes #123)").

Pushing and Pull Requests (PRs):

- Push your feature branch to the remote repository:

```
1. git push origin feature/your-feature-name
```

- Create a Pull Request (PR) from your feature branch into develop.
- Ensure PRs are reviewed by at least one other developer.
- Address any feedback from the review.

- **Merging:** Once reviewed and tests pass (CI if implemented), the PR can be merged into develop. Merges into main are typically done from develop during a release process.

4. Azure Services (Quick Overview)

No development instructions are needed for the Azure services in use for the cloud, but it is important to know what they are. In the future developers may change these services to self hosted ones or to other services providers (AWS, Google, etc).

Services:

- Resource Group: A logical container that holds related resources for an Azure solution. It allows you to manage, monitor, and delete all the resources for your solution as a single unit.

- Azure Container Registry (ACR): A managed, private Docker registry service in Azure for storing and managing your container images. Acts like a private repository for your Docker images.

- Azure Cache for Redis: An in-memory data store based on the open-source Redis. It's used to improve the performance and scalability of applications that rely heavily on backend data stores by caching frequently accessed data. We use it mainly for our redis and deep model communications.

- Azure Cosmos DB: A globally distributed, multi-model database service. It provides low-latency access to data with high availability and scalability, making it suitable for modern applications that require rapid access to large amounts of data.

- Azure Blob Storage: A massively scalable and secure object storage service for unstructured data. It's ideal for storing large amounts of data. We use it for audio.

- Azure Cognitive Services - Speech Service: Enables applications to integrate speech processing capabilities. This includes speech-to-text (transcribing spoken audio into text) and text-to-speech (synthesizing natural-sounding speech from text). We use the speech-to-text functionality.

- Azure Cognitive Services - Language Service (Text Analytics): Provides natural language processing (NLP) capabilities to understand and analyze text. This includes features like sentiment analysis, key phrase extraction, named entity recognition, and language detection.

- App Service (Container): A fully managed platform for building, deploying, and scaling web applications, REST APIs, and mobile backends. "Web App for Containers" specifically allows us to deploy and run your containerized web applications directly on the App Service platform.

5. Key Areas for Modification

Project Tree & Core Workflow:

```
├── app.py                      # Flask application entry point (Server initialization, blueprint registration)
├── config.py                   # Centralized configuration (Environment variables, Azure services, JWT settings)
├── celery_app.py               # Celery task queue initialization (Background job processing)
├── requirements.txt            # Python dependencies (Flask, Azure SDKs, ML libraries)
├── Dockerfile                  # Container configuration for deployment
├── supervisord.conf            # Process management configuration
├── generate_pass.py            # Password generation utility
├── seed_database.py            # Database seeding script
├── test_api_flow.py            # API integration tests
├── auth/                       # Authentication & Authorization Module
│   ├── __init__.py             # Module initialization
│   ├── routes.py               # Auth endpoints (login, refresh, password reset)
│   └── utils.py                # Auth utilities (password hashing, token generation)
├── profiles/                   # User Profile Management Module
│   ├── __init__.py             # Module initialization
│   ├── routes.py               # Profile endpoints (search patients, get own profile)
├── sessions/                   # Therapy Session Management Module (Core Business Logic)
│   ├── __init__.py             # Module initialization
│   ├── routes.py               # Session endpoints (upload, details, search, me)
│   ├── tasks.py                # Celery background tasks (STT processing, sentiment analysis)
│   └── utils.py                # Session utilities (sentiment analysis helpers)
├── devices/                    # Device Registration Module
│   ├── __init__.py             # Module initialization
│   └── routes.py               # Device endpoints (FCM token registration)
├── data/                       # Data Management Module
│   ├── __init__.py             # Module initialization
│   └── routes.py               # Data endpoints (general data operations)
├── database/                   # Database Layer (Azure Cosmos DB)
│   ├── __init__.py             # Module initialization
│   └── cosmos.py               # Cosmos DB client, containers, and data access methods
├── notifications/              # Push Notification Module
│   ├── __init__.py             # Module initialization
│   └── fcm.py                  # Firebase Cloud Messaging implementation
└── secrets/                    # Security Credentials
    ├── firebase_service_account.json/  # Firebase service account credentials
    └── therapyai-d137c-faee3122d9ae.json/  # Additional service credentials

# Key Technologies & Services:
# - Flask: Web framework for REST API
# - Azure Cosmos DB: NoSQL database for users, sessions, devices
# - Azure Speech Services: Speech-to-text transcription with diarization
# - Azure Text Analytics: Sentiment analysis for patient speech
# - Azure Blob Storage: Audio file storage and processing
# - Firebase Cloud Messaging: Push notifications to mobile clients
# - Celery + Redis: Asynchronous task processing for STT and sentiment analysis
# - JWT: Authentication and authorization tokens
# - Docker: Containerized deployment
```

```
48.
49. # Core Workflow:
50. # 1. Therapist uploads audio session via /sessions/upload
51. # 2. Audio stored in Azure Blob Storage, session metadata in Cosmos DB
52. # 3. Celery task processes audio with Azure Speech Services (STT + diarization)
53. # 4. Patient speech analyzed for sentiment using Azure Text Analytics
54. # 5. Processed transcript and sentiment scores stored in session details
55. # 6. FCM notification sent to therapist when processing complete
56. # 7. Therapists can review, edit, and finalize session data
57. # 8. Final session details are available for search and retrieval
```

API Endpoints (Flask Routes):

- Located in modules like auth/routes.py, sessions/routes.py, etc.
- New endpoints should be added to the relevant Blueprint.
- Utilize flask_jwt_extended for authentication (@jwt_required()).
- Handle request data (request.get_json(), request.form, request.files).
- Return JSON responses (jsonify()) with appropriate HTTP status codes.
- Add logging for important events and errors (logger.info(), logger.error()).

Background Tasks (Celery):

- Main tasks are in sessions/tasks.py.
- Define new tasks using @celery.task.
- Ensure tasks are idempotent if they might be retried.
- Handle exceptions within tasks gracefully and log errors.
- Update task status in Cosmos DB (e.g., session_processing_data_container) as tasks progress.
- Dispatch tasks from Flask routes using .delay() (e.g., process_audio_stt_task.delay(...)).

Database Interactions (Cosmos DB):

- All database logic is centralized in database/cosmos.py.
- The Scheme and database Overview information are in DB_overview.pdf, please refer to that file for more information about the database.
- Add new functions here for new queries or data manipulation.

- Be mindful of partition keys for performance and cost. Queries that don't specify a partition key will be cross-partition queries (ensure enable_cross_partition_query=True is used where necessary, but aim for single-partition queries if possible).
- Handle cosmos_exceptions.CosmosResourceNotFoundError and other potential database errors.
- If you change data schemas, consider how existing data will be affected (migrations might be needed for production, less critical for POC).
- The app functionality is reliant on the current scheme of the database. When updating and adapting, make sure that the app is being updated accordingly.

Configuration (config.py):

- New configuration variables should be added to config.py.
- Read values from environment variables using os.getenv("VAR_NAME", "default_value").
- **Crucial:** If you add a new configuration variable that needs to be set in Azure, you **must** add it to the Azure App Service "Application settings" (see Deployment Guide).

Frontend API Consumption (Considerations):

- Maintain a clear API contract (paths, methods, request/response schemas).
- Document API changes for the frontend team/developer.
- Consider API versioning if making breaking changes in the future.

Firebase Cloud Messaging (FCM):

- FCM sending logic is in notifications/fcm.py.
- To modify notification content or triggers:
    - Update the send_fcm_notification function or how it's called (e.g., in sessions/tasks.py after STT completion).
    - Ensure GOOGLE_APPLICATION_CREDENTIALS (or FIREBASE_SERVICE_ACCOUNT_JSON_CONTENT) is correctly configured.

6. Testing

Local Manual Testing:

- Use tools like Postman or Insomnia to test API endpoints directly after starting the Flask dev server.
- Test file uploads, authentication, and different request payloads.

Using test_api_flow.py:

- This script provides an end-to-end integration test suite.
- **Before running:**
    - Ensure BASE_URL in the script points to your local Flask server (http://localhost:FLASK_PORT) or your deployed Azure instance.
    - Ensure your local Flask server and Celery worker are running.
    - Ensure your .env file is configured correctly so the application can connect to Redis, Cosmos DB, etc.
    - Ensure the users specified (e.g., THERAPIST_EMAIL) exist in your database (run seed_database.py if needed for a clean test environment).
- Run the script: python test_api_flow.py
- Update this script as new features are added or existing ones change.

Unit Tests (Future Enhancement):

- Implement unit tests for individual functions and classes, especially for business logic in tasks and database interactions.
- Use frameworks like pytest or Python's built-in unittest.
- Mock external dependencies (like Azure services) during unit tests.

7. Building & Deploying Changes to Azure

Refer to the "Cloud Deployment & Maintenance Guide" for the detailed Azure setup. The update process for a developer typically involves:

Merge Changes: Ensure your feature branch is merged into develop (and subsequently into main for a release).

Pull Latest Code:

```
1. git checkout develop # Or main for a release
2. git pull origin develop
```

Ensuring Docker is Running & Azure Sign-In:

```
1. az login
2. az acr login –name YourUniqueACRName
3.
```

Rebuild Docker Image:

```
1. docker build -t YourUniqueACRName.azurecr.io/therapyapp:latest .
2. # Or use a new version tag:
3. # docker build -t YourUniqueACRName.azurecr.io/therapyapp:v1.1.0
4.
```

Push Image to ACR:

```
1. docker push YourUniqueACRName.azurecr.io/therapyapp:latest
2. # Or: docker push YourUniqueACRName.azurecr.io/therapyapp:v1.1.0
```

Update Azure App Service:

1. If using the latest tag, **Restart** the Web App in the Azure portal. It should pull the updated latest image.
2. If using version tags, go to your Web App -> "Deployment Center" or "Container settings" and update the image tag to the new version (e.g., v1.1.0).
3. **Remember:** If you added new environment variables in config.py, you **must** add them to the App Service "Application settings".

8. Logging & Debugging

Local Logging:

- Flask's development server provides request logs and output from print() statements or Python's logging module.
- The Celery worker will also output its logs to the console.
- Configure logging.basicConfig(level=logging.DEBUG) in app.py or celery_app.py during local development for more verbose output if needed.

Azure Log Inspection:

- **App Service Logs (Blob Storage):** The most reliable way. Configure in App Service settings to send container logs to Azure Blob Storage.
- **Kudu (Advanced Tools):** Access LogFiles/docker/ for container logs.
- **Log Stream:** Real-time logs (can be unreliable).
- Check logs from both gunicorn (Flask app) and celery_worker processes, as they are combined by supervisor into the container's stdout/stderr.

9. Coding Standards & Best Practices

PEP 8: Follow PEP 8 Python style guidelines. Use a linter like Flake8 or Pylint.

Clear Naming: Use descriptive names for variables, functions, and classes.

Modularity: Keep functions and classes focused on a single responsibility.

Comments: Add comments to explain complex logic or non-obvious code sections.

Error Handling: Implement robust error handling (try-except blocks) and return meaningful error messages in API responses.

Security:

- Validate all user inputs.
- Be mindful of security when constructing database queries (though direct user input into queries is largely avoided here).
- Keep dependencies updated to patch security vulnerabilities.

No Hardcoded Secrets: All secrets (API keys, passwords, connection strings) must be loaded from environment variables (config.py pattern).

10. Managing Dependencies

All Python dependencies are listed in requirements.txt.

Adding a new dependency:

- Install it in your virtual environment: pip install new_package
- Update requirements.txt: pip freeze > requirements.txt
- Commit the updated requirements.txt.

Updating dependencies:

Periodically update packages to their latest stable versions:

```
1. pip install --upgrade package_name
2. # Or update all:
3. # pip list --outdated --format=freeze | grep -v '^\-e' | cut -d = -f 1  | xargs -n1 pip install -
U
4. pip freeze > requirements.txt
5.
```

Test thoroughly after updating dependencies, as breaking changes can occur.