Official documentation for AVANOCHI

this is the official documentation of the AVANOCHI project. every code involved will be explained here

Table of Contents

Introduction

1. General Architecture: Azure serverless service

1.1 AZ_functions directory

The AZ_functions module serves as the backbone of Avanochi's cloud logic. It organizes all serverless functions into thematic domains, each responsible for a specific aspect of the application. This modular design ensures scalability, maintainability, and adherence to the SOLID principles.

Structure Overview

- `.shared/`: This module ensures consistency, reusability, and maintainability, reducing code duplication across the system.
  - Centralizes cross-cutting concerns and reusable components for all other modules.
  - init.py: Marks the package as importable and optionally exposes shared interfaces.
  - `credential_manager.py`: Handles secure authentication and credential rotation.
  - `database.py`: Defines the database connection layer and abstracts CRUD operations.
  - `models.py`: Declares data models and schemas used across functions.
  - `repositories.py`: Implements the repository pattern, linking models with database logic.
  - `services.py`: Provides high-level services that orchestrate business logic and integrations.
- `assistant/`
  - Includes AI-driven features such as *narratives*, *predictions*, and *recommendations*.
  - These functions process historical data, generate natural language summaries, and provide personalized insights to the user.
  - Acts as the decision-making hub of Avanochi's assistant role.
- `interaction/`
  - Handles user interaction layers such as bot communication, notifications, and voice integration.
  - Designed for multi-channel experiences, enabling seamless integration with chat platforms (Teams, Slack) and voice systems (Azure Speech Services).
- `health/`
  - Manages well-being functionalities: hydration, meals, and rests.
  - Functions here send reminders, track user inputs, and contribute to the healthy-life dimension of the project.
  - These elements directly influence the gamified Avanochi "persona".
- `work/`
  - Dedicated to productivity tracking: achievements, stats, tasks, and work sessions.
  - Core logic for task management, progress measurement, and session recording resides here.
  - This module feeds most of the gamification system, unlocking achievements and performance insights.
- Configuration files

    - `host.json` and `local.settings.json` define the runtime enviroment and local development setup.
    - Ensure consistency between cloud deployment and local testing
- `templates/`
    - Stores pre-defined templates such as `cosmosdb.json`
    - Provides ready-to-use schemas for database bindings and resource definitions

Shared Architecture

The `.shared` module is a foundational layer that centralizes all cross-cutting concerns within the AZ_functions ecosystem. It ensures that domain-specific functions (assistant, health, interaction, work) can focus purely on their logic, while shared components provide common services, models, and infrastructure handling.

Here we have an structural Overview of the logic it follows

Credential Manager

This module is responsible for managing authentication and secure access to the project’s external resources, such as **CosmosDB**, **Azure services**, or third-party APIs. It centralizes all credential-related logic, including retrieval from secure environments and, when required, automatic rotation of secrets. By handling this complexity in one place, the rest of the system is shielded from directly managing tokens or sensitive keys. This not only reduces the exposure of confidential information but also ensures compliance with security best practices. In essence, it acts as a security gateway, guaranteeing that all external connections are performed safely and consistently.

The main working logic can be divided into two main functions:

- **General method `get()`**
  This method will get an input of the expected key and a default value, so that any key can be looked up for in the credential library. if no key is found, it will return the default value

  ```
  def get(self, key: str, default: str = None):
     # Retrieve any environment variable by key.
     return os.getenv(key, default)
  ```

- **Specific methods**
  For each service that requires authentication methods through API keys or special endpoints, there will be specific methods such as the `get_cosmos_credentials()` method, that will return a list of keys only needed in the `CosmosDBService` class. any other service method shall be coded in `CredentialManager` class to keep a clean use of the credential extraction and not abuse the `get()` method

  ```
  def get_cosmos_credentials(self):
      # Retrieve Cosmos DB credentials from environment variables.
      return {
          "account_name": os.getenv("COSMOS_DB_ACCOUNT", ""),
          "database_name": os.getenv("COSMOS_DB_DATABASE", ""),
          "container_name": os.getenv("COSMOS_DB_CONTAINER", ""),
          "uri": os.getenv("COSMOS_DB_URI", ""),
          "primary_key": os.getenv("COSMOS_DB_PRIMARY_KEY", "")
      }
  ```

Database: CosmosDB