# 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.

### 1.1.1 Service structures

### 1.1.2 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

```python
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

This module is responsible for handling all interactions with Azure Cosmos DB, ensuring that data persistence is managed in a centralized and secure manner. The CosmosDBService class abstracts the complexity of creating and maintaining the connection to Cosmos DB, making sure that the required database and container exist before any operation is performed. By depending exclusively on the `CredentialManager` to retrieve its credentials, this class adheres to the Single Responsibility Principle, keeping authentication logic separated from database logic. This approach guarantees scalability, maintainability, and consistent usage of environment-based configurations. The main responsibilities can be divided into the following components:

- **Initialization (`__init__`)** Upon instantiation, the service retrieves Cosmos DB credentials through the CredentialManager and sets up the database connection.
  - It creates a `CosmosClient` with the provided URI and primary key.
  - It ensures the target database exists (or creates it if missing).
  - It ensures the container exists (or creates it if missing) with a default partition key and throughput configuration.

```
def __init__(self, credential_manager: CredentialManager):
    # Initialize the Cosmos DB client and ensure the database and container
exist.

    creds = credential_manager.get_cosmos_db_credentials()
    self._url = creds["uri"]
    self._key = creds["primary_key"]
    self._database_name = creds["database_name"]
    self._container_name = creds["container_name"]
    # Create the Cosmos client
    self._client = CosmosClient(self._url, self._key)
    # Ensure the database exists
    self._database =
self._client.create_database_if_not_exists(id=self._database_name)
    # Ensure the container exists
    self._container = self._database.create_container_if_not_exists(
        id=self._container_name,
        partition_key=PartitionKey(path="/id"),  # Default partition key
        offer_throughput=400
    )
```

## Entities

This module defines the core domain entities of the application. Entities are plain Python classes that represent the business objects of Avanochi, such as **tasks** and **work sessions**. They are designed with **Single Responsibility Principle (SRP)** in mind:

- They only handle their own state and basic transformations.
- They are not aware of database operations or business orchestration.
- They provide serialization methods so that higher layers (repositories, services) can persist or transport them as dictionaries/JSON.

To give further examples, **tasks** and **work sessions** will be explained bellow.

- **Entity Tasks** Represents a single task in the system, having the following fields:
  - `id`: unique identifier (UUID)
  - `title`: short description of the task
  - `completed`: boolean status (default `false`)
  - `created_at`: UTC timestamp of the creation (ISO format)
  - `updated_at`: timestamp of the last update (optional)

  It contains only one method to return the `task` as a dictionary:

```python
def to_dict(self):
    return self.__dict__
```

  The output would be something like this:

```
{
    "id": "3b9f7b8e-6d7a-4e3f-a23c-5a0efb9b72c9",
    "title": "Finish project report",
    "completed": false,
    "created_at": "2025-09-25T10:15:30.123456",
    "updated_at": null
}
```

- **Entity WorkSession** Represents a session of productive work for a given user, having the following fields:
    - `id`: unique identifier (UUID)
    - `user_id`: identifier of the user who owns the session
    - `start_time`: UTC timestamp when the session started (ISO format).
    - `end_time`: UTC timestamp when the session ended (ISO format).
    - `duration`: session length in hours, stored as a float (rounded to 2 decimals).

  It contain two methods:

    - `end_session()`: sets the end_time to current UTC time and calculates duration in hours.
    - `to_dict()`: returns the session as a dictionary for storage or serialization.

```
def end_session(self):
    self.end_time = datetime.utcnow().isoformat()
    start = datetime.fromisoformat(self.start_time)
    end = datetime.fromisoformat(self.end_time)
    self.duration = round((end - start).total_seconds() / 3600, 2)

def to_dict(self):
    return self.__dict__
```

example dictionary structure after ending a session:

```
{
    "id": "f49d0c33-b6cf-4d77-a274-8903b38c8ed2",
    "user_id": "user_123",
    "start_time": "2025-09-25T09:00:00.000000",
    "end_time": "2025-09-25T11:30:00.000000",
    "duration": 2.5
}
```

## Services

This module defines the business logic layer of the application. Services orchestrate operations on domain entities and delegate persistence to repositories.
They are designed with **Single Responsibility Principle (SRP)** in mind:

- They only handle the application logic and orchestration.
- They are not aware of database implementations or infrastructure details.
- They rely on repository interfaces to persist or retrieve domain entities.

To give further examples, **TaskService** and **WorkSessionService** services will be explained below.

- **TaskService**
  Provides the application logic for creating, listing, and completing tasks.

  It exposes the following methods:

  - `create_task(title: str)`: creates a new Task entity and delegates persistence to the repository.
  - `list_tasks()`: retrieves all tasks from the repository.
  - `complete_task(task_id: str)`: marks a task as completed by delegating the update to the repository.

  Example usage:

```
task_service = TaskService(repo)
new_task = task_service.create_task("Finish project report")
tasks = task_service.list_tasks()
completed = task_service.complete_task(new_task["id"])
```

  Example result after creating a task:

```
{
    "id": "3b9f7b8e-6d7a-4e3f-a23c-5a0efb9b72c9",
    "title": "Finish project report",
    "completed": false,
    "created_at": "2025-09-25T10:15:30.123456",
    "updated_at": null
}
```

- **WorkSessionService**
  Provides the application logic for starting and ending productive work sessions.

  It exposes the following methods:

  - `start_session(user_id: str)`: creates a new WorkSession entity for the given user and persists it through the repository.
  - `end_session(session_id: str)`: closes an existing session by calculating the duration and updating it through the repository.

  Example usage:

```
session_service = WorkSessionService(repo)
session = session_service.start_session("user_123")
closed = session_service.end_session(session["id"])
```

  Example result after ending a session:

```
{
    "id": "f49d0c33-b6cf-4d77-a274-8903b38c8ed2",
    "user_id": "user_123",
    "start_time": "2025-09-25T09:00:00.000000",
    "end_time": "2025-09-25T11:30:00.000000",
    "duration": 2.5
}
```

## Repositories

Repositories act as the **public-facing API of the persistence layer**.
They are the only access point available for services and higher layers to communicate with the database, while all lower-level components such as `CosmosDBService` and `CredentialManager` remain private and inaccessible outside this layer.

In practice, repositories define the **data access layer** of the application.
They are responsible for persisting and retrieving domain entities, designed with the **Single Responsibility Principle (SRP)** in mind:

- They only handle communication with the database.
- They are not aware of business logic or entity rules.
- They provide a clean abstraction that services can use without depending on database details.

By exposing **abstract base classes (interfaces)**, repositories enforce clear contracts that any implementation must follow. This ensures that:

- Services and clients remain fully decoupled from the database technology.
- All database operations are centralized and consistent.
- Infrastructure concerns are hidden, following the **Dependency Inversion Principle (DIP)**.
- Future changes in persistence (e.g., switching from CosmosDB to another database) require no modification in the business logic or client code.

To give further examples, **tasks** and **work sessions** repositories will be explained below.

- **Interface ITaskRepository**
  Defines the contract for managing `Task` entities. Any class implementing this interface must provide the following methods:
  - `create_task(task: Task)`: persists a new task.
  - `list_tasks()`: retrieves all tasks.
  - `complete_task(task_id: str)`: marks a task as completed.

- **Repository CosmosTaskRepository (ITaskRepository)**
  Concrete implementation of `ITaskRepository` using Cosmos DB.
    - On initialization, retrieves a container from `CosmosDBService`.
    - `create_task()`: inserts a new task document.
    - `list_tasks()`: returns all task documents in the container.
    - `complete_task()`: finds a task by ID, marks it as completed, and updates it in the database.

  Example usage:

```
repo = CosmosTaskRepository(db_service)
task = Task("Finish report")
repo.create_task(task)
tasks = repo.list_tasks()
repo.complete_task(task.id)
```

  Example document stored in the database:

```
{
    "id": "3b9f7b8e-6d7a-4e3f-a23c-5a0efb9b72c9",
    "title": "Finish project report",
    "completed": true,
    "created_at": "2025-09-25T10:15:30.123456",
    "updated_at": null
}
```

- **Interface IWorkSessionRepository**
  Defines the contract for managing `WorkSession` entities. Any class implementing this interface must provide the following methods:
    - `start_session(session: WorkSession)`: persists a new work session.
    - `end_session(session_id: str)`: closes an existing work session and updates its duration.

- **Repository CosmosWorkSessionRepository (IWorkSessionRepository)**
  Concrete implementation of `IWorkSessionRepository` using Cosmos DB.
    - On initialization, retrieves a container from `CosmosDBService`.
    - `start_session()`: inserts a new work session document.
    - `end_session()`: retrieves a session by ID, calculates its duration, and updates the record in the database.

  Example usage:

```
repo = CosmosWorkSessionRepository(db_service)
session = WorkSession("user_123")
repo.start_session(session)
repo.end_session(session.id)
```

  Example document stored in the database after ending a session:

```
{
    "id": "f49d0c33-b6cf-4d77-a274-8903b38c8ed2",
    "user_id": "user_123",
    "start_time": "2025-09-25T09:00:00.000000",
    "end_time": "2025-09-25T11:30:00.000000",
    "duration": 2.5
}
```