AVANOCHI: OFFICIAL TECHNICAL DOCUMENTATION

Mike Marañón Quero & Marta Fraile Jara
AVANOCHI Avanade

Table of Contents

Introduction

This is the official documentation of the AVANOCHI project. Every code and architectural decision involved will be explained here in detail.

AVANOCHI is a productivity application with a playful and gamified approach, designed to enhance **time management**, **work performance**, and **well-being** during the workday. The project introduces a virtual companion — a “productive tamagotchi” — that not only motivates employees but also assists in organizing tasks, tracking performance, and promoting healthy work habits.

At its core, AVANOCHI bridges **task management**, **behavioral gamification**, and **AI-powered insights** to create an engaging system that goes beyond traditional productivity tools. Unlike conventional task trackers, AVANOCHI establishes an emotional connection with the user: the character reacts to the employee’s actions, encourages breaks, and evolves alongside the worker’s performance. This interaction transforms daily work into a dynamic and rewarding experience, increasing motivation while reducing stress and fatigue.

The platform is designed with **modularity** and **scalability** in mind, leveraging modern cloud resources and AI-driven features. It integrates with Microsoft Azure services for data storage, analytics, AI recommendations, and voice/chat interaction. This ensures that the system not only adapts to individual needs but also continuously improves through **machine learning and data-driven feedback**.

Purpose of the project

The main goal of AVANOCHI is to support employees in achieving higher productivity while maintaining a healthy work-life balance. To accomplish this, the system:

- Encourages **organization** through daily and weekly task management.
- Provides **real-time performance tracking** and visual statistics.
- Promotes **healthy habits** such as hydration, breaks, and proper lunch times.
- Uses **gamification elements** (achievements, rewards, character evolution) to increase engagement.
- Offers **personalized recommendations** powered by AI, adapting to each employee’s work rhythm.
- Predicts risks of **work overload and burnout**, sending alerts before they happen.
- Extends productivity beyond the app itself with **multichannel integration** (Teams, Slack, voice assistant).

In summary, AVANOCHI is not just a productivity app, but a **digital companion** that combines technology, gamification, and well-being to redefine the way employees interact with their workday.

1. General Architecture: Azure serverless service

Avanochi is built on a **serverless architecture** using Microsoft Azure Functions. This approach was chosen to ensure scalability, modularity, and cost efficiency. Instead of relying on a traditional web server that requires continuous maintenance and resource allocation, serverless functions allow us to run code only when specific events are triggered. This event-driven model aligns perfectly with Avanochi's needs, where different modules—such as task tracking, AI recommendations, and health reminders—can operate independently without interfering with each other.

By adopting a serverless structure, we gain several advantages: reduced operational overhead, automatic scaling according to workload, and seamless integration with other Azure services like Cosmos DB, AI Foundry, and Speech Services. This design ensures that Avanochi remains lightweight yet powerful, focusing development efforts on **business logic and user experience** rather than infrastructure management.

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 layout naturally aligns with Azure Functions' event-driven model: each directory encapsulates functions tied to specific triggers and responsibilities.

Such organization simplifies both development and testing by isolating functionalities into independent domains. Each domain can evolve on its own lifecycle, making it easier to extend or replace components without affecting the rest of the system. The result is a clean, predictable architecture that supports continuous integration and long-term maintainability, while keeping the cloud logic transparent and easy to navigate.

1.1.1 Structure Overview

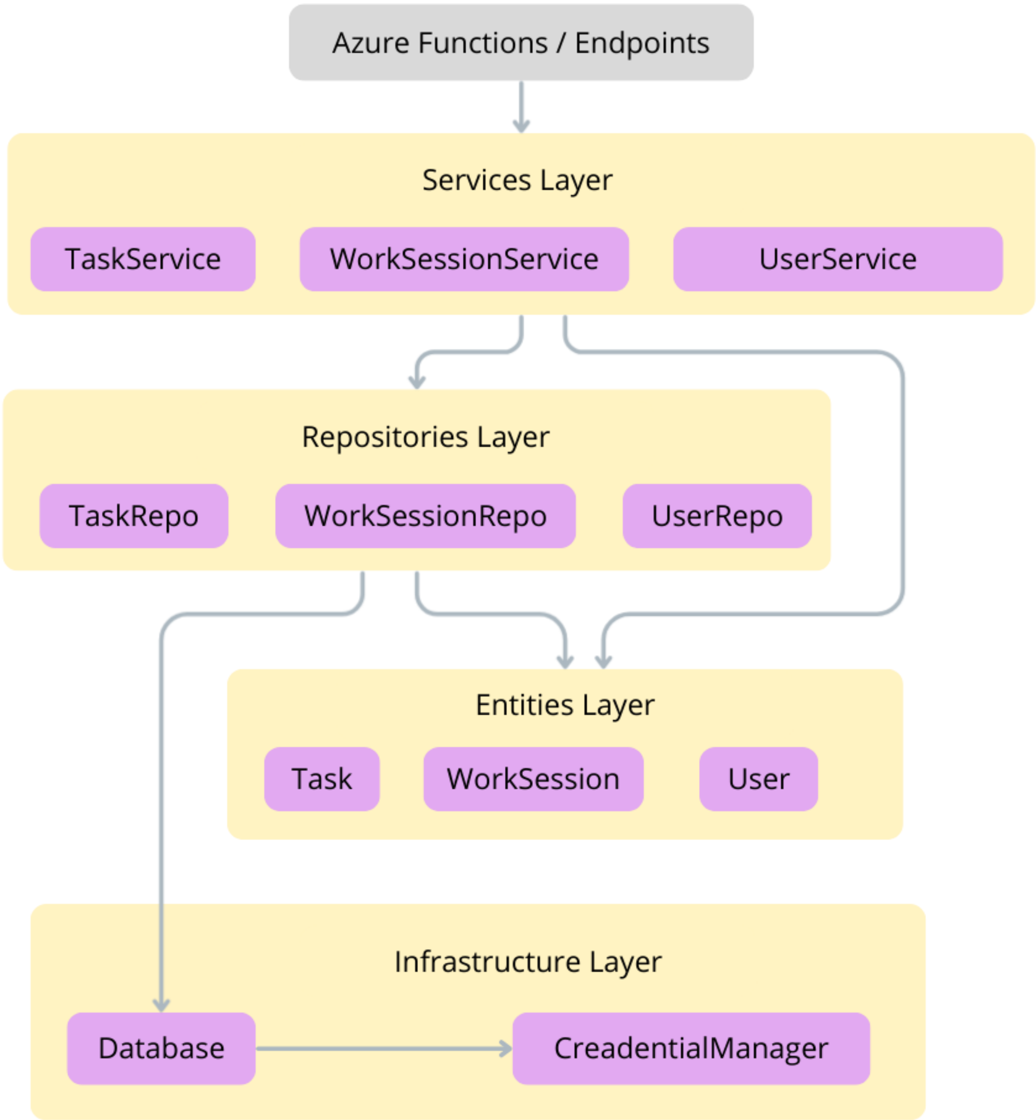
- `.shared/`: This module ensures consistency, reusability, and maintainability, reducing code duplication across the system.
  - Centralizes cross-cutting concerns and reusable components for all other modules.
  - init.py: Marks the package as importable and optionally exposes shared interfaces.
  - `credential_manager.py`: Handles secure authentication and credential rotation.
  - `database.py`: Defines the database connection layer and abstracts CRUD operations.
  - `entities/`: Declares data models and schemas used across functions.
  - `repos/`: Implements the repository pattern, linking entities with database logic.
  - `services/`: 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

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
    )
```

- **Database methods**: The `CosmosDBService` provides a set of generic CRUD methods and query execution helpers that abstract direct interaction with Cosmos DB:
    - `create_item()`: Inserts a new document into the container. Automatically generates a unique `id` if not provided and validates the presence of a `user_id`.

```python
def create_item(self, item: dict) -> dict:
    try:
        # Ensure unique ID
        if "id" not in item:
            item["id"] = str(uuid.uuid4())

        if "user_id" not in item:
            raise DatabaseError("Missing required field: 'user_id'")

        created = self._container.create_item(body=item)
        logging.info(f"Item created with id={created['id']}")
        return created
    except exceptions.CosmosHttpResponseError as e:
        logging.error(f"Failed to create item: {e.message}")
        raise DatabaseError(f"Failed to create item: {e.message}") from e
```

    - `read_item()`: Retrieves a single document by its `id` and partition key. Raises a `DatabaseError` if the item is not found.

```python
def read_item(self, item_id: str, partition_key: str) -> dict:
    try:
        return self._container.read_item(item=item_id,
partition_key=partition_key)
    except exceptions.CosmosResourceNotFoundError:
        raise DatabaseError(f"Item with id '{item_id}' not found.")
    except exceptions.CosmosHttpResponseError as e:
        raise DatabaseError(f"Failed to read item '{item_id}': {e.message}") from
e
```

    - `upsert_item()`: Inserts or updates a document. Ensures that `id` and `user_id` are present before performing the operation.

```python
def upsert_item(self, item: dict) -> dict:
    try:
        if "id" not in item:
            item["id"] = str(uuid.uuid4())

        if "user_id" not in item:
            raise DatabaseError("Missing required field: 'user_id'")

        upserted = self._container.upsert_item(body=item)
        logging.info(f"Item upserted with id={upserted['id']}")
        return upserted
    except exceptions.CosmosHttpResponseError as e:
        logging.error(f"Failed to upsert item: {e.message}")
        raise DatabaseError(f"Failed to upsert item: {e.message}") from e
```

    - `delete_item()`: Deletes a document by its `id` and partition key. Logs the deletion attempt and raises a `DatabaseError` if the item does not exist.

```
def delete_item(self, item_id: str, partition_key: str) -> None:
    try:
        logging.info(f"Attempting to delete item with id={item_id}")
        self._container.delete_item(item=item_id, partition_key=partition_key)
    except exceptions.CosmosResourceNotFoundError:
        raise DatabaseError(f"Item with id '{item_id}' not found, cannot
delete.")
    except exceptions.CosmosHttpResponseError as e:
        raise DatabaseError(f"Failed to delete item '{item_id}': {e.message}")
from e
```

- `send_query()`: Executes a custom SQL-like query against the container, supporting optional parameters and cross-partition queries. Debug logs include the query and provided parameters.

```
def send_query(self, query: str, parameters: list = None) -> list[dict]:
    if parameters is None:
        parameters = []
    try:
        logging.debug(f"Executing query: {query} | Parameters: {parameters or
'None'}")
        results = self._container.query_items(
            query=query,
            parameters=parameters,
            enable_cross_partition_query=True
        )
        return [item for item in results]
    except exceptions.CosmosHttpResponseError as e:
        raise DatabaseError(f"Query failed: {e.message}") from e
```

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**, **work sessions**, and **users**.

They are designed with the **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 (`to_dict`) so that higher layers (repositories, services) can persist or transport them as dictionaries/JSON.

The folder structure will be explained bellow with each entity:

```
shared/
└── entities/
    ├── task.py
    ├── work_session.py
    └── user.py
```

- **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:

  ```
  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
}
```

- **Entity User** Represents a user of the system.

  Fields:

  - `id`: unique identifier (UUID)
  - `name`: display name of the user
  - `created_at`: UTC timestamp of user creation (ISO format)
  - `updated_at`: timestamp of the last update (optional)

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

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

  output would be something like this:

  ```json
  {
      "id": "21e4e82b-03d2-4e15-8d73-ff3b8737f8b0",
      "name": "Alice",
      "created_at": "2025-09-26T08:15:45.123456",
      "updated_at": null
  }
  ```

Repositories

Repositories provide the formal interface to the persistence layer: they are the only components that encapsulate direct data access logic and present a consistent API for the rest of the application. Note that services are the layer that should be called by endpoints—services orchestrate business logic and call repositories; endpoints must not access CosmosDBService or CredentialManager directly, as those are implementation details of the persistence 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.

The folder structure will be explained bellow with each repo:

```
shared/
└── repos/
    ├── base_repo.py
    ├── task_repo.py
    ├── user_repo.py
    └── work_session_repo.py
```

- **BaseRepository**
  `BaseRepository` is an **abstract class** that defines generic CRUD operations and query execution.
  All concrete repositories inherit from this base class and implement their own `entity_type()` to identify the type of document they manage in the database.

  The methods provided are the following:

  - `create(entity: dict) -> dict`
    Persists a new entity in the database. The repository automatically injects its `type` before delegating to Cosmos. Returns the stored entity as a dictionary.
  - `get(entity_id: str) -> dict`
    Retrieves a single entity by its unique identifier. If the entity does not exist, a `DatabaseError` will be raised.
  - `update(entity: dict) -> dict`
    Updates an existing entity in the database, or creates it if it does not exist (Cosmos upsert operation). Returns the updated entity as a dictionary.
  - `delete(entity_id: str) -> None`
    Deletes an entity by its unique identifier. If the entity does not exist, the operation raises a `DatabaseError`.
  - `query(query: str, params: list = None) -> list[dict]`
    Executes a SQL-like query against the container. Parameters can be passed as a list of dictionaries (`{"name": ..., "value": ...}`). Returns a list of matching entities.

  By extending `BaseRepository`, all repositories benefit from these generic operations without duplicating code.

- **TaskRepository**
  Manages `Task` entities.

  Methods:

  - `create_task(task: Task)`: persists a new task.
  - `list_tasks()`: retrieves all tasks.
  - `complete_task(task_id: str)`: marks a task as completed.

  Example usage:

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

- **WorkSessionRepository**
  Manages `WorkSession` entities.

  Methods:

  - `start_session(session: WorkSession)`: persists a new work session.
  - `end_session(session_id: str)`: closes an existing session and updates its duration.
  - `get_active_session(user_id: str)`: returns the current active session for a user.
  - `list_sessions(user_id: str)`: retrieves all sessions for a given user.

  Example usage:

  ```
  repo = WorkSessionRepository(db_service)
  session = WorkSession("user_123")
  repo.start_session(session)
  repo.end_session(session.id)
  active = repo.get_active_session("user_123")
  sessions = repo.list_sessions("user_123")
  ```

- **UserRepository**
  Manages `User` entities.

  Methods:

  - `create_user(user: User)`: persists a new user.
  - `get_user(user_id: str)`: retrieves a user by ID.
  - `update_user(user: User)`: updates an existing user.
  - `delete_user(user_id: str)`: deletes a user by ID.
  - `list_users()`: lists all users in the database.

  Example usage:

  ```
  repo = UserRepository(db_service)
  user = User("Alice")
  repo.create_user(user)
  fetched = repo.get_user(user.id)
  repo.update_user(user)
  all_users = repo.list_users()
  repo.delete_user(user.id)
  ```

Services

This directory 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 repositories to persist or retrieve domain entities.

In practice, **services act as the API of the application**: they are the only entry point that higher layers (such as Azure Functions or REST endpoints) should use.
Repositories are never called directly from outside — all interactions must go through services.

The folder structure will be explained bellow with each service:

```
shared/
└── entities/
    ├── base_service.py
    ├── task_service.py
    ├── user_service.py
    └── work_session_service.py
```

- **BaseService**
  An abstract base class that defines a contract for all services in the application.

  It exposes the following method:

  - `get_entity_type() -> str`: returns the type of entity handled by the service (e.g., `"Task"`, `"WorkSession"`, `"User"`).

  All services inherit from this class to ensure consistency across the application.

- **TaskService**
  Provides the application logic for creating, listing, and completing tasks.
  This service validates input (such as empty titles) and creates `Task` entities before delegating persistence to the `TaskRepository`.

  It exposes the following methods:

  - `create_task(title: str)`: creates a new Task entity and persists it through the repository.
  - `list_tasks()`: retrieves all tasks from the repository.
  - `complete_task(task_id: str)`: marks a task as completed and updates it through 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, ending, and listing productive work sessions.
  This service validates input (such as missing `user_id`) and creates `WorkSession` entities before delegating persistence to the `WorkSessionRepository`.

  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 its duration and updating it through the repository.
  - `get_active_session(user_id: str)`: retrieves the currently active session for the given user.
  - `list_sessions(user_id: str)`: retrieves all sessions associated with the given user.

  Example usage:

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

  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
  }
  ```

- **UserService**
  Provides the application logic for creating, retrieving, updating, and deleting users.
  This service validates input (such as empty names) and creates `User` entities before delegating persistence to the `UserRepository`.

  It exposes the following methods:

  - `create_user(name: str)`: creates a new User entity and persists it through the repository.
  - `get_user(user_id: str)`: retrieves a user by ID from the repository.
  - `update_user(user: User)`: updates an existing user in the repository.
  - `delete_user(user_id: str)`: deletes a user from the repository.
  - `list_users()`: retrieves all users from the repository.

  Example usage:

  ```
  user_service = UserService(repo)
  new_user = user_service.create_user("Alice")
  fetched = user_service.get_user(new_user["id"])
  updated = user_service.update_user(new_user)
  all_users = user_service.list_users()
  user_service.delete_user(new_user["id"])
  ```

  Example result after creating a user:

  ```
  {
      "id": "a8c91a7e-4a3b-45c1-9f27-97c847cf3d11",
      "name": "Alice",
      "created_at": "2025-09-25T14:45:00.000000",
      "updated_at": null
  }
  ```