2024

MachineVision NLPServer documentation

V ALPHA 1.0

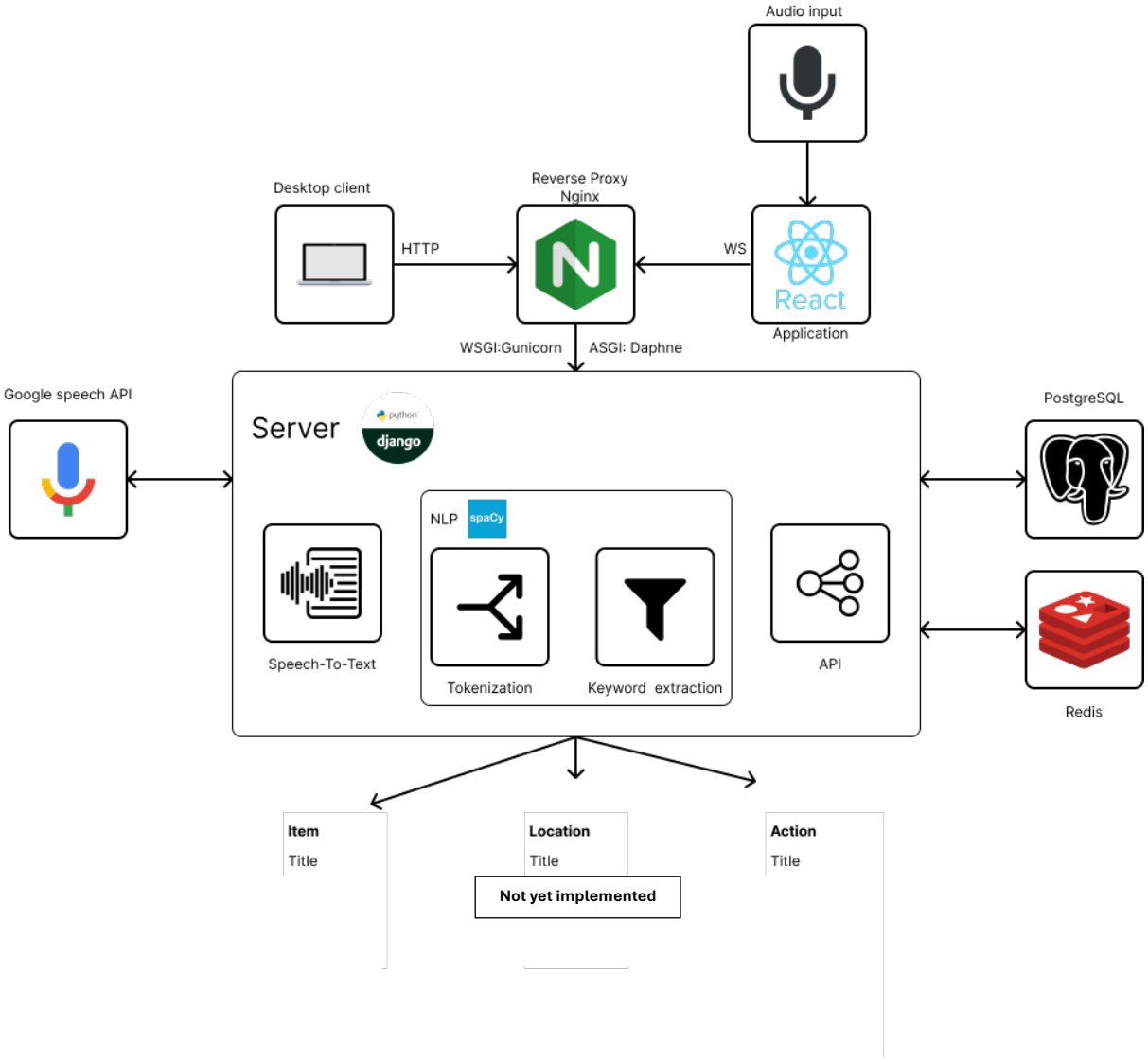
LINARDS LIEPENIEKS

Overview

NLPServer is a Django application for the purpose of receiving user audio and text input and analyzing it using SpaCy library and delivering keywords to client machines for image recognition.

Technical implementation

- React native application – The application can connect user to the server using a ws connection, receive user input both in text and audio form as well as establish a real-time connection with a machine.
- Desktop client – users can connect to the server's administration panel through a web browser.
- Nginx – as web browsers use http and mobile application establishes connection with ws nginx acts as a reverse proxy and routes the connections to the correct web server – gunicorn for http and daphne for ws
- Django – Django functionalities:
    - Validate and authenticate incoming connection requests via api keys – users and machines have separate api keys which distinguish their functionality
    - Receive messages from mobile app users:
        - Transcribe text – receives an audio file and returns transcribed text
        - Analyze text – receives a string and returns the object of the sentence (uses NLP using spaCy library)
        - Connect/disconnect to a machine – if an available to user machine is currently online a user can establish a connection via redis.
    - Forward analyzed information to currently connected machines.
- Redis – cache for managing real time connections between connected users and machines
- PostgreSQL – database for storing user and machine information.
- Google speech API – currently the system utilizes google speech API, where the server delivers the audio file to the google cloud and receives transcribed text



Notes

The architecture of the server is still under discussion – before optimization efforts can be attempted the main point of concern is how and where will the AI capabilities be hosted – mainly the transcription API, and what language model should be used. Currently the server utilizes google cloud STT api and spaCy default language model (en_core_web_sm) – the capabilities of this model need to be tested more thoroughly to discover user experience issues.

Applications

In Django, applications are self-contained modules that encapsulate specific functionality, allowing for modular development and reuse across multiple projects.

API

The API module is responsible for user authentication and connection establishment as well as message management and transmition from application users to client machines.

Models

1. BaseAPIKey Model (Abstract)

- **Purpose**: Serves as an abstract base class for API key models, encapsulating common attributes and behavior.
- **Attributes**:
  - key: A unique UUID field that stores the API key.
  - name: A character field to name or identify the API key, limited to 50 characters.
  - is_active: A boolean field indicating whether the API key is active or not, defaulting to True.
  - created_at: A datetime field that records when the API key was created, set automatically at creation.
- **Methods**:
  - __str__(): Returns a string representation combining the name and owner_name.
  - owner_name (property): Abstract property intended to be implemented by subclasses to return the name of the key's owner.

2. APIKey Model (Inherits from BaseAPIKey)

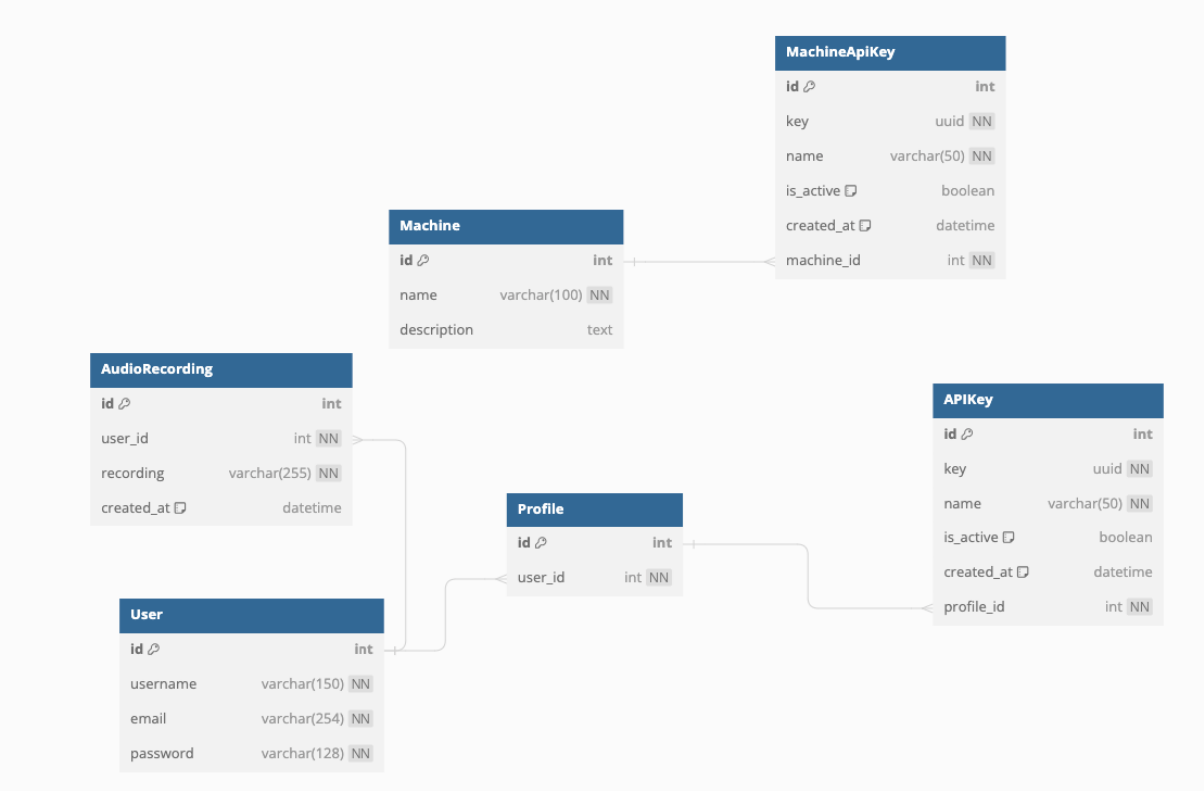
- **Purpose**: Represents an API key specifically associated with a user profile.
- **Attributes**:
  - profile: A foreign key linking the API key to a Profile object (assumed to be defined in the users app). This field establishes a one-to-many relationship, indicating that each API key is associated with a single profile, but each profile can have multiple API keys.
- **Methods**:
  - owner_name (property): Returns the username of the user associated with the profile.

3. MachineApiKey Model (Inherits from BaseAPIKey)

- **Purpose**: Represents an API key specifically associated with a machine.
- **Attributes**:
  - machine: A foreign key linking the API key to a Machine object (defined in the machines app). This field establishes a one-to-many relationship, indicating that each API key is associated with a single machine, but each machine can have multiple API keys.
- **Methods**:
  - owner_name (property): Returns the name of the machine associated with the API key.

4. AudioRecording Model

- **Purpose**: Represents an audio recording made by a user.
- **Attributes**:
  - user: A foreign key linking the recording to a User object, establishing a one-to-many relationship where a user can have multiple recordings.
  - recording: A file field that stores the audio file, which is uploaded to the recordings/ directory.
  - created_at: A datetime field that records when the recording was created, set automatically at creation.
- **Methods**:
  - __str__(): Returns a string representation indicating the user and the timestamp of the recording.



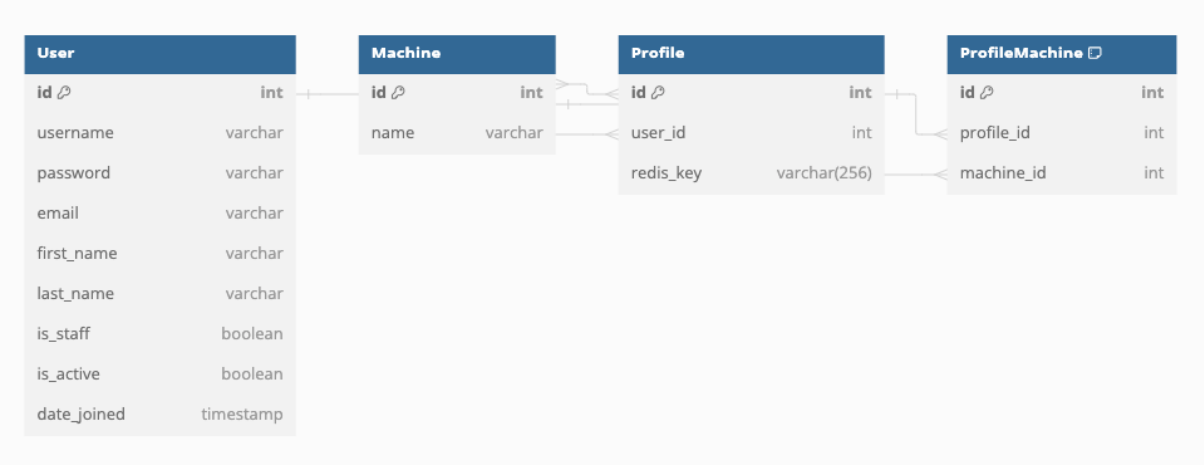
Machines

Machines module is responsible for storing and managing all machines and their API keys, in future this might need more functionality, but currently it is just storing the known machines.

Models

Machine Model
The Machine model represents a machine entity with attributes such as a name, a unique machine identifier, and a Redis key. Below are the details:

- **Attributes**:
  - name: A CharField that stores the name of the machine. It is limited to 255 characters.
  - machine_id: A UUIDField that stores a unique identifier for each machine. It uses Python's uuid.uuid4() to generate a unique UUID, ensuring that each machine has a distinct ID. This field is not editable.
  - redis_key: A CharField that stores a key used in Redis, which is limited to 400 characters. It can be blank and is not editable by default. This key is usually used to store or retrieve machine-specific data in a Redis store.
- **Methods**:
  - __str__(): A string representation method that returns a formatted string containing the machine's name and UUID. This is useful for debugging and logging purposes.



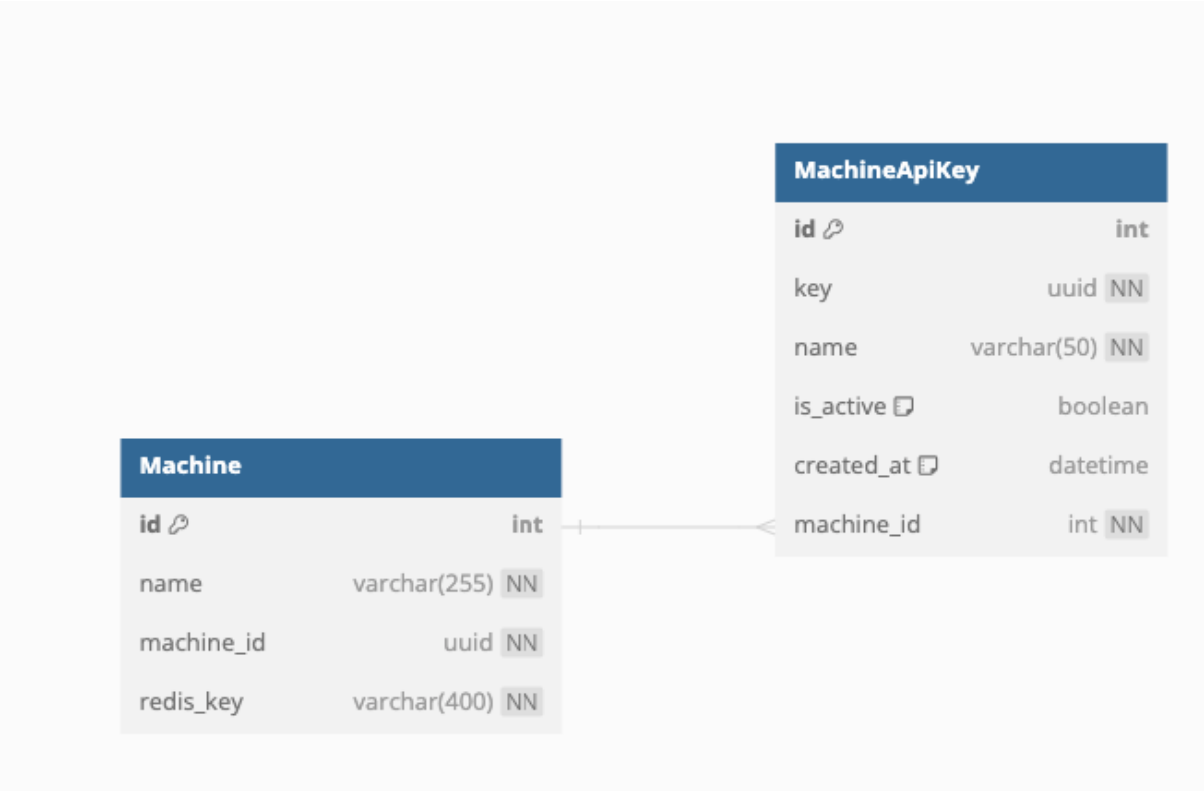
Signals

Django signals allow you to execute specific code at certain points during the execution of your application. In this context, signals are used to automate the creation of an API key for each new machine and to set a Redis key before saving the machine instance.

1. **post_save Signal**:
   - **Purpose**: To automatically create an API key for a machine whenever a new Machine instance is created and saved to the database.
   - **Signal Function**: create_machine_api_key
   - **Trigger**: This function is triggered after a Machine instance is saved to the database.

- **Logic**: If the Machine instance is newly created (created is True), the function generates a new UUID (API key) and creates a new MachineApiKey instance associated with the Machine.

2. **pre_save Signal**:
   - **Purpose**: To set the redis_key field for a Machine instance before it is saved to the database.
   - **Signal Function**: set_redis_key
   - **Trigger**: This function is triggered right before a Machine instance is saved to the database.
   - **Logic**: If the redis_key is not already set, it assigns a value to redis_key based on the machine's name and its UUID, ensuring a unique and meaningful Redis key.



Users

As it is not recommended to modify the django auth user model Users module manages and stores the Profile module and machine connections – each auth user model has a connected profile model which stores additional data.

Models

1. Profile Model

- **Purpose:** The Profile model extends the Django User model with additional functionality:
- **Attributes:**

    - user: A one-to-one relationship with the User model. Each Profile is uniquely associated with one User. If the user is deleted, the associated profile is also deleted (on_delete=models.CASCADE).
    - redis_key: A unique, optional field that stores a Redis key. It can be blank or null.
    - machines: A many-to-many relationship with the Machine model through an intermediary model ProfileMachine. This allows each profile to be associated with multiple machines and vice versa.
- Methods:
    - __str__(): Returns a string representation of the profile, including the username of the associated user.

2. ProfileMachine Model

The ProfileMachine model acts as an intermediary between Profile and Machine, allowing for additional customization and constraints:

- **Attributes**:
    - profile: A foreign key linking to the Profile model. Represents the profile associated with a machine.
    - machine: A foreign key linking to the Machine model. Represents the machine associated with a profile.
- **Meta Class**:
    - UniqueConstraint: Ensures that each combination of profile and machine is unique. This constraint prevents duplicate associations between profiles and machines.
- **Methods**:
    - __str__(): Returns a string representation of the relationship between the profile and the machine.

Signals

1.post_save Signal for User:

- **Purpose**: To automatically create a Profile instance when a new User instance is created.
- **Signal Function**: create_user_profile
- **Trigger**: This function is triggered after a User instance is saved.
- **Logic**: If a new User instance is created (created=True), a new Profile is created for that user. Logs an informational message about the profile creation.

2. post_save Signal for Profile:

- **Purpose**: To create an API key and generate a Redis key for a new Profile.
- **Signal Function**: create_api_and_redis_keys
- **Trigger**: This function is triggered after a Profile instance is saved.
- **Logic**:
    - **API Key Creation**: Generates a new UUID as an API key and creates an APIKey instance associated with the profile.
    - **Redis Key Generation**: If the redis_key is not set, it generates a new Redis key based on the username and saves it to the Profile.

- **Error Handling**: Catches and logs any errors during the key creation process, rolling back the transaction if necessary.

Text Analysis

Text analysis module is responsible for storing various keywords which the NLP algorithm will utilize to send to the client machines.

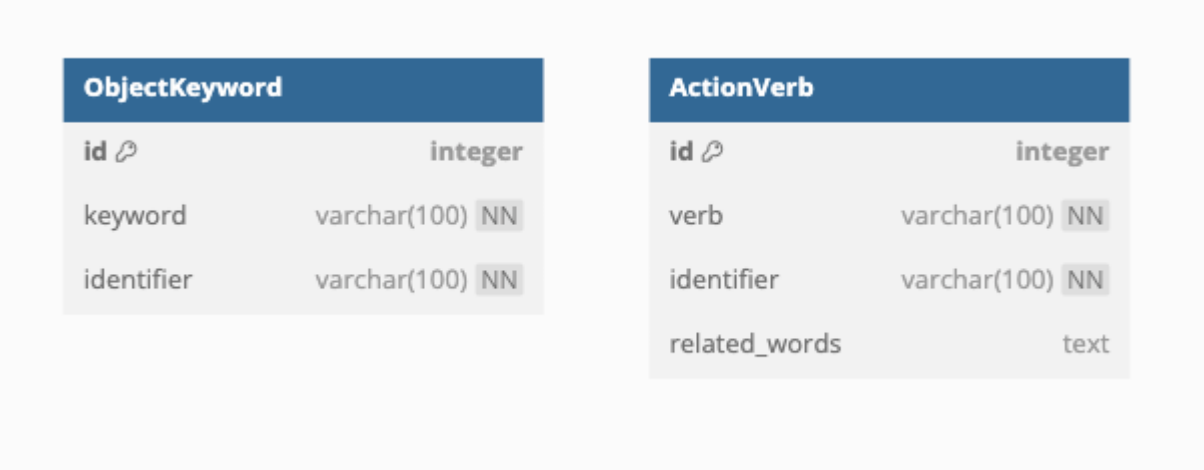
Models

1. ObjectKeyword Model The ObjectKeyword model represents keywords associated with objects:

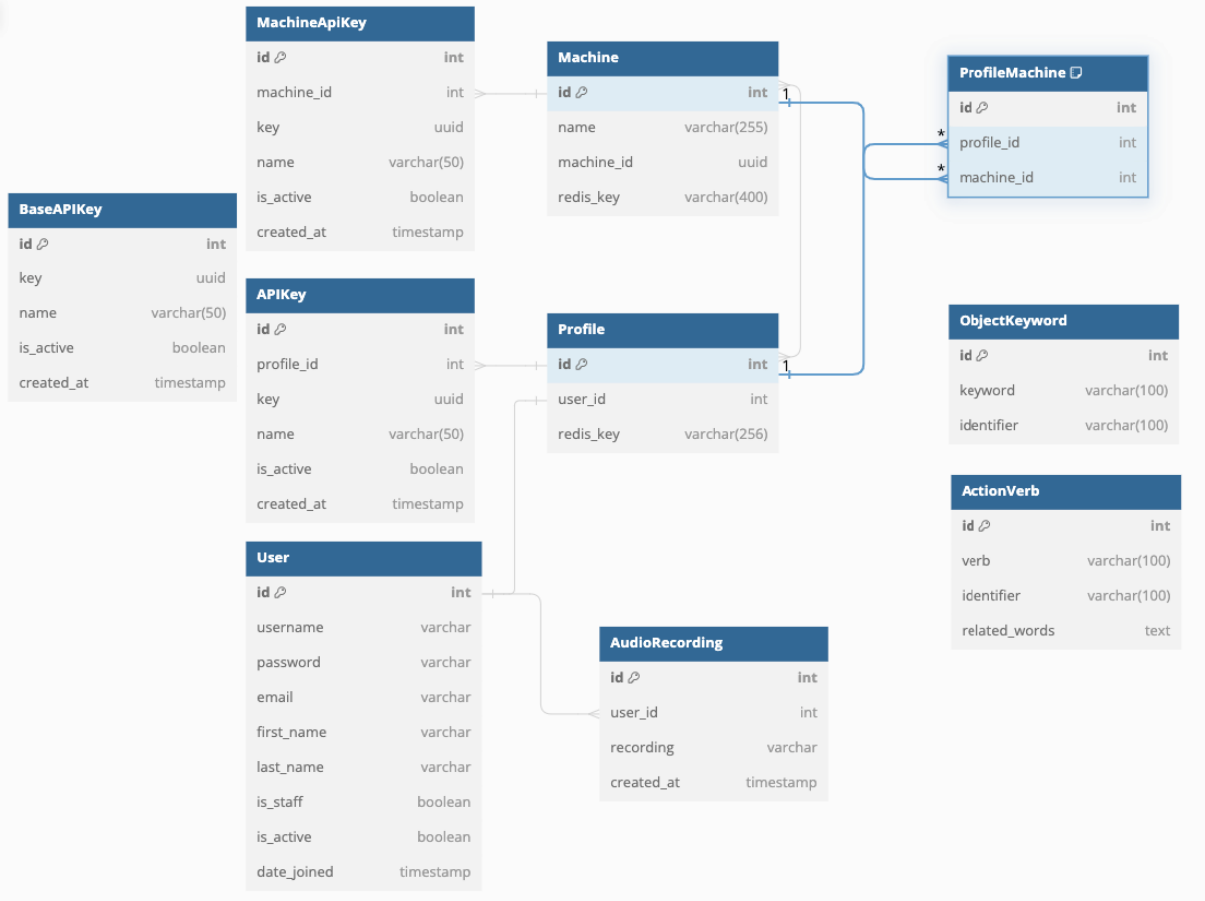
- **Attributes**:
  - keyword: A CharField with a maximum length of 100 characters. It is unique and represents the keyword itself.
  - identifier: A CharField with a maximum length of 100 characters. It is unique and serves as an identifier for the keyword.
- **Methods**:
  - **str**(): Returns a string representation of the ObjectKeyword, combining the keyword and its identifier.

2. ActionVerb Model The ActionVerb model represents verbs associated with actions:

- **Attributes**:
  - verb: A CharField with a maximum length of 100 characters. It is unique and represents the action verb.
  - identifier: A CharField with a maximum length of 100 characters. It is unique and serves as an identifier for the verb.
  - related_words: A TextField that stores related words as comma-separated values. It can be blank.
- **Methods**:
  - set_related_words(words): Takes a list of words and sets the related_words field by joining them with commas.
  - get_related_words(): Returns a list of related words by splitting the related_words field on commas. Returns an empty list if there are no related words.
  - **str**(): Returns a string representation of the ActionVerb, combining the verb and its identifier.



Database Schema



WebSocket connection (ASGI server)

WebSocket connection is implemented with django channels library (version 3.0.5 *newer versions do not work correctly*). django channels uses consumers attached to a module to manage messages and a routing file containing avaliable routes.

Routing

Routes are located in api/routes.py

ENDPOINTS:

1. wsapi/users/

- Handled by: UserConsumer
- Purpose: This route handles WebSocket connections related to user operations.
- uses:
    - Real-time updates for user data
    - User authentication over WebSocket
    - Pushing notifications to specific users

2. wsapi/machines/

- Handled by: MachineConsumer
- Purpose: This route is for WebSocket connections related to machine operations.

- uses:
    - Real-time status updates for machines
    - Sending commands to machines
    - Receiving data streams from machines

Consumers

Overview

This application uses Django Channels to handle WebSocket connections for real-time communication between users and machines. It utilizes Redis for managing real-time connections and group memberships.

ApiConsumer

Base consumer class that handles authentication and connection management.

Key Functions:

- connect(): Establishes the WebSocket connection, validates authentication, and sets up the channel layer.
- disconnect(): Handles the disconnection process, removing the consumer from Redis groups.
- receive(): Processes incoming WebSocket messages, parsing JSON data.

Messages:

- **connect:** Accepts the connection if authentication is valid. Logs errors if authentication fails.
- **disconnect:** Removes the WebSocket connection from Redis sets when disconnected.
- **receive**: Parses incoming messages and determines the type for further handling.

UserConsumer

Handles WebSocket connections for user clients.

Key Functions:

- connect(): Connects the user and adds them to the "user_group" in Redis.
- disconnect(): Removes the user from machine rooms and the user group in Redis.
- receive(): Routes incoming messages to appropriate handlers based on the message type.
- _handle_analyze(): Processes text analysis requests and sends results to connected machines.
- _handle_transcribe(): Handles speech-to-text conversion requests.
- _handle_reload_machines(): Updates the list of available machines and their states.
- _handle_connect_to_machine(): Manages user connection to a specific machine, updating Redis groups.
- _handle_disconnect_from_machine(): Handles user disconnection from a machine, updating Redis groups.
- reload_machines(): Fetches the current state of all machines, using Redis to determine their availability.

- notify_reload_machines(): Notifies other users to reload their machine lists when changes occur.

Messages:

- **analyze**: Analyzes provided text and sends results to connected machines.
- **transcribe**: Receives audio file data, transcribes it to text, and sends the result back to the user.
- **reload_machines**: Requests an update of the list of machines and sends the updated list to the user.
- **connect_to_machine**: Attempts to connect the user to a specified machine and handles connection status.
- **disconnect_from_machine**: Disconnects the user from a specified machine and handles disconnection status.

MachineConsumer

Manages WebSocket connections for machine clients.

Key Functions:

- connect(): Establishes the machine's connection, creates a machine-specific room in Redis, and notifies users.
- disconnect(): Removes the machine from its room and notifies users of the disconnection.
- execute(): Handles execution requests sent from users to the machine.
- _join_machine_room: Adds the machine to its designated Redis room.
- _notify_group: Notifies the specified group of the machine's status change (connected/disconnected).

Messages:

- **connect**: Adds the machine to its room in Redis and notifies the user group of its connection.
- **disconnect**: Removes the machine from its room in Redis, deletes the room, and notifies the user group of its disconnection.
- **execute**: Receives and acknowledges execution requests from users.

Redis Usage

- Redis is used to maintain real-time group memberships for users and machines.
- It helps in managing machine states (Online, Offline, Busy) and user-machine connections.
- Redis groups are used for efficient broadcasting of messages to relevant clients.

Natural language processing

Overview:

The algorithm processes a given sentence to identify and categorize action verbs and object keywords using spaCy and a Django model-based database. The primary purpose is to extract and analyze actions and objects mentioned in the text, checking if they are recognized and/or if they are synonyms of known entries.

Process:

1. **Text Processing:**
    - The sentence is parsed using spaCy's en_core_web_sm model to tokenize the text and identify parts of speech and grammatical dependencies.
2. **Action Verbs Identification:**
    - The algorithm scans each token in the parsed sentence.
    - Verbs are identified by checking if the token's part of speech (pos_) is "VERB" and its dependency (dep_) is either "ROOT" or "xcomp", indicating significant action within the sentence.
    - For each identified verb, the following checks are performed:
        - **Exact Match:** The algorithm queries the ActionVerb model to see if the verb matches any known verbs.
        - **Synonym Check:** If no exact match is found, the algorithm compares the verb with known action verbs using semantic similarity. This involves comparing the similarity score of the verb against pre-defined thresholds to determine if the verb is a synonym.
    - Results are compiled into a list with details including:
        - **Text:** The verb text.
        - **Verb:** The canonical form of the verb (if known).
        - **ID:** Identifier from the database (if known).
        - **Is Known:** Whether the verb is a known action.
        - **Is Synonym:** Whether the verb is a synonym of a known action.
3. **Object Keywords Identification:**
    - The algorithm also scans tokens for direct objects (dobj) and objects of prepositions (pobj).
    - For each identified object, the following checks are performed:
        - **Exact Match:** The algorithm queries the ObjectKeyword model to find if the object matches any known keywords.
        - **Synonym Check:** If no exact match is found, the algorithm compares the object with known keywords using semantic similarity. This involves comparing the similarity score of the object against pre-defined thresholds to determine if it is a synonym.
    - Results are compiled into a list with details including:
        - **Text:** The object text.
        - **Keyword:** The canonical form of the object (if known).
        - **ID:** Identifier from the database (if known).
        - **Is Known:** Whether the object is a known keyword.
        - **Is Synonym:** Whether the object is a synonym of a known keyword.

Output:

The algorithm returns a dictionary with two lists:

- **Actions:** A list of dictionaries, each containing information about an identified action verb.

- **Objects:** A list of dictionaries, each containing information about an identified object keyword.

Note:

The current method for finding synonyms relies on spaCy's similarity model and the thresholds used. This approach may need optimization based on specific use cases and hardware limitations. Future improvements in synonym detection are contingent upon deciding appropriate thresholds and available computational resources.

Closing remarks

In conclusion, while the current system is operational and effectively extracts and categorizes action verbs and object keywords from text, its performance and feasibility are heavily dependent on the specific environment in which it will be deployed—be it remote computing, a dedicated server room, or another setting. Given the computational intensity of the synonym detection process and the variability in hardware capabilities, it is crucial to define a more concrete use case and assess hardware requirements before proceeding. This careful consideration will ensure that the system can be effectively and efficiently implemented, avoiding potential challenges associated with universal deployment.