MyProject Documentation

Contents

Module app

Sub-modules

- app.core
- app.main
- app.v1

Module app.core

Sub-modules

- app.core.models
- app.core.settings

Module app.core.models

Sub-modules

- app.core.models.models

Module `app.core.models.models`

Functions

Function `generate_id`

```
def generate_id()
```

Generate a unique identifier (UUID).

This function generates a universally unique identifier (UUID) and returns it as a string. UUIDs are unique across space and time, making them suitable for various purposes, including as unique identifiers for database records, objects, or entities.

Returns —-= str : A string representation of the generated UUID.

Example Usage: unique_id = generate_id()

Note —-= - UUIDs generated by this function are typically 36 characters long and have a format like “xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx”. - The probability of generating the same UUID twice is extremely low, making UUIDs suitable for unique identification purposes.

Classes

Class `Feedback`

```
class Feedback(
    **data: Any
)
```

Model representing feedback.

This class extends the base FeedbackBase model and includes an attribute for the feedback_id, which represents a unique identifier for the feedback.

Attributes —-= - feedback_id (str): A unique identifier for the feedback. Example Usage: # Creating an instance of Feedback with sample data feedback_data = Feedback(feedback_id=str(uuid.uuid4()), created_at=datetime.utcnow(), session_id=“98766545431”, feedback=“positive/negative/neutral”, feedback_stage=“chat”, data=“Optional”)

Note —-= - This class is used to represent feedback data with a specified feedback_id. - It is typically used when retrieving or working with existing feedback data.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.FeedbackBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `feedback_id` Type: `str`

Variable `model_config`

Variable `model_fields`

Class `FeedbackBase`

```
class FeedbackBase(
    **data: Any
)
```

Base model for feedback data.

This class defines a base data model for feedback, including attributes such as the feedback creation timestamp, session ID, feedback content, feedback stage, and additional data.

Attributes —-= - created_at (Optional[datetime]): The timestamp when the feedback was created. Default is None. - session_id (Optional[str]): The session ID associated with the feedback. Default is None. - feedback (Optional[str]): The feedback content (e.g., positive/negative/neutral). Default is None. - feedback_stage (Optional[str]): The stage at which the feedback was given (e.g., “chat”). Default is None. - data (Optional[str]): Additional data related to the feedback. Default is None. Example Usage: # Creating an instance of FeedbackBase with sample data feedback_data = FeedbackBase(created_at=datetime.utcnow(), session_id=“98766545431”, feedback=“positive/negative/neutral”, feedback_stage=“chat”, data=“Optional”)

Note —-= - This class serves as a base model for feedback data, and its attributes can be optional to accommodate various use cases.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Descendants

- app.core.models.models.Feedback
- app.core.models.models.FeedbackCreate

Class variables

Variable `created_at` Type: `Optional[datetime.datetime]`

Variable `data` Type: `Optional[str]`

Variable `feedback` Type: `Optional[str]`

Variable `feedback_stage` Type: `Optional[str]`

Variable `model_config`

Variable `model_fields`

Variable `session_id` Type: `Optional[str]`

Class `FeedbackCreate`

```
class FeedbackCreate(
    **data: Any
)
```

Model for creating feedback.

This class extends the base FeedbackBase model and adds an attribute for the feedback_id, which represents a unique identifier for the feedback being created.

Attributes —-= - feedback_id (Optional[str]): A unique identifier for the feedback. It is generated using the generate_id() function by default. Example Usage: # Creating an instance of FeedbackCreate with sample data feedback_data = FeedbackCreate(feedback_id=str(uuid.uuid4()), created_at=datetime.utcnow(), session_id="98766545431", feedback="positive/negative/neutral", feedback_stage="chat", data="Optional")

Note —-= - This class is used to represent feedback data when creating new feedback, and it includes additional attributes beyond the base FeedbackBase.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.FeedbackBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `feedback_id` Type: `Optional[str]`

Variable `model_config`

Variable `model_fields`

Class `Message`

```
class Message(
    **data: Any
)
```

Model representing a chat message.

This class extends the base MessageBase model and includes an attribute for the message_id, which represents a unique identifier for the message.

Attributes —-= - message_id (str): A unique identifier for the message. Example Usage: # Creating an instance of Message with sample data message_data = Message(message_id=str(uuid.uuid4()), created_at=datetime.utcnow(), exchange="chat from the user or bot", message_type="AI/USER", session_id="98766545431")

Note —-= - This class is used to represent a chat message with a specified message_id. - It is typically used when retrieving or working with existing chat messages.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.MessageBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `message_id` Type: `str`

Variable `model_config`

Variable `model_fields`

Class `MessageBase`

```
class MessageBase(
    **data: Any
)
```

Base model for chat messages.

This class defines a base data model for chat messages. It includes attributes such as the message creation timestamp, exchange source, message type, and session ID.

Attributes —-= - created_at (Optional[datetime]): The timestamp when the message was created. Default is the current UTC time. - exchange (Optional[str]): The source of the message (e.g., “chat from the user or bot”). Default is None. - message_type (Optional[str]): The type of message (e.g., “AI/USER”). Default is None. - session_id (Optional[str]): The session ID associated with the message. Default is None. Example Usage: # Creating an instance of MessageBase with sample data message_data = MessageBase(created_at=datetime.utcnow(), exchange=“chat from the user or bot”, message_type=“AI/USER”, session_id=“98766545431”)

Note —-= - This class serves as a base model for chat messages, and its attributes can be optional to accommodate various use cases.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Descendants

- app.core.models.models.Message
- app.core.models.models.MessageCreate

Class variables

Variable `created_at` Type: `Optional[datetime.datetime]`

Variable `exchange` Type: `Optional[str]`

Variable `message_type` Type: `Optional[str]`

Variable `model_config`

Variable `model_fields`

Variable `session_id` Type: `Optional[str]`

Class `MessageCreate`

```
class MessageCreate(
    **data: Any
)
```

Model for creating a chat message.

This class extends the base MessageBase model and adds an attribute for the message_id, which represents a unique identifier for the message being created.

Attributes —-= - message_id (Optional[str]): A unique identifier for the message. It is generated using the generate_id() function by default. Example Usage: # Creating an instance of MessageCreate with sample data message_data = MessageCreate(message_id=str(uuid.uuid4()), created_at=datetime.utcnow(), exchange="chat from the user or bot", message_type="AI/USER", session_id="98766545431")

Note —-= - This class is used to represent a chat message when creating a new message, and it includes additional attributes beyond the base MessageBase.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.MessageBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `message_id` Type: `Optional[str]`

Variable `model_config`

Variable `model_fields`

Class `Session`

```
class Session(
    **data: Any
)
```

Model representing a session.

This class extends the base SessionBase model and includes an attribute for the session_id, which represents a unique identifier for the session.

Attributes —–= - session_id (str): A unique identifier for the session. Example Usage: # Creating an instance of Session with sample data session_data = Session(session_id=str(uuid.uuid4()), created_at=datetime.utcnow(), ended_at=datetime.utcnow() + timedelta(minutes=30), user_email="user@example.com", user_name="John Doe", user_role="Owner/Applicant/Realtor", data_user_consent=True, metadata=SessionMetadata(ip="111.111.11.111", device="Mobile/Desktop", browser="Chrome", os="Windows/Linux/MacOS"))

Note —–= - This class is used to represent a session, including its attributes and associated data. - Example data is provided in the json_schema_extra for reference, demonstrating the expected structure of session data.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.SessionBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `model_config`

Variable `model_fields`

Variable `session_id` Type: `str`

Class `SessionBase`

```
class SessionBase(
    **data: Any
)
```

Base model for session data.

This class defines a base data model for capturing session-related information. It includes attributes such as session creation and end times, user email, user name, user role, user consent, and metadata about the session.

Attributes —–= - created_at (Optional[datetime]): The timestamp when the session was created. Default is the current UTC time. - ended_at (Optional[datetime]): The timestamp when the session ended. Default is None. - user_email (Optional[EmailStr]): The email address of the user associated with the session. Default is None. - user_name (Optional[str]): The name of the user associated with the session. Default is None. - user_role (Optional[str]): The role of the user associated with the session. Default is None. - data_user_consent (Optional[bool]): A flag indicating whether the user has given consent for data collection. Default is False. - metadata (Optional[SessionMetadata]): Additional metadata about the session, such as IP address, device, browser, and OS. Default is None. Example Usage: # Creating an instance of SessionBase with sample data session_data = SessionBase(user_email="user@example.com", user_name="John Doe", user_role="Admin", data_user_consent=True, metadata=SessionMetadata(ip="111.111.11.111", device="Mobile/Desktop", browser="Chrome", os="Windows/Linux/MacOS"))

Note —–= - This class serves as a base model for capturing session data, and its attributes can be optional to accommodate various use cases. - Example data is provided in the model_config for reference, demonstrating the expected structure of session data.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Descendants

- app.core.models.models.Session
- app.core.models.models.SessionCreate

Class variables

Variable `created_at` Type: `Optional[datetime.datetime]`

Variable `data_user_consent` Type: `Optional[bool]`

Variable `ended_at` Type: `Optional[datetime.datetime]`

Variable `metadata` Type: `Optional[app.core.models.models.SessionMetadata]`

Variable `model_config`

Variable `model_fields`

Variable `user_email` Type: `Optional[pydantic.networks.EmailStr]`

Variable `user_name` Type: `Optional[str]`

Variable `user_role` Type: `Optional[str]`

Class `SessionCreate`

```
class SessionCreate(
    **data: Any
)
```

Model for creating a session.

This class extends the base SessionBase model and adds an attribute for the session_id, which represents a unique identifier for the session being created.

Attributes —–= - session_id (str): A unique identifier for the session. It is generated using the generate_id() function by default. Example Usage: # Creating an instance of SessionCreate with sample data session_data = SessionCreate(session_id=str(uuid.uuid4()), created_at=str(datetime.utcnow()), ended_at=datetime.utcnow() + timedelta(minutes=30), user_email="user@example.com", user_name="John Doe", user_role="Owner/Applicant/Realtor", data_user_consent=True, metadata=SessionMetadata(ip="111.111.11.111", device="Mobile/Desktop", browser="Chrome", os="Windows/Linux/MacOS"))

Note —–= - This class is used to represent session data when creating a new session, and it includes additional attributes beyond the base SessionBase. - Example data is provided in the json_schema_extra for reference, demonstrating the expected structure of session data when creating a session.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.SessionBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `model_config`

Variable `model_fields`

Variable `session_id` Type: `str`

Class `SessionMetadata`

```
class SessionMetadata(
    **data: Any
)
```

Session metadata model.

This class defines a data model for capturing session metadata, including information about the user's IP address, device type, browser, and operating system.

Attributes —-= - ip (Optional[str]): The user's IP address. Default is None. - device (Optional[str]): The type of device used (e.g., Mobile or Desktop). Default is None. - browser (Optional[str]): The user's browser (e.g., Chrome). Default is None. - os (Optional[str]): The user's operating system (e.g., Windows, Linux, or MacOS). Default is None. Example Usage: # Creating an instance of SessionMetadata with sample data metadata = SessionMetadata(ip="111.111.11.111", device="Mobile/Desktop", browser="Chrome", os="Windows/Linux/MacOS")

Note —-= - This class is used to represent session metadata, and the attributes can be optional, allowing for flexibility in capturing session-related information. - Example data is provided in the model_config for reference, demonstrating the expected structure of session metadata.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `browser` Type: `Optional[str]`

Variable `device` Type: `Optional[str]`

Variable `ip` Type: `Optional[str]`

Variable `model_config`

Variable `model_fields`

Variable `os` Type: `Optional[str]`

Class `Ticket`

```
class Ticket(
    **data: Any
)
```

Model representing a ticket.

This class extends the base TicketBase model and includes an attribute for the ticket_id, which represents a unique identifier for the ticket.

Attributes —-= - ticket_id (str): A unique identifier for the ticket. Example Usage: # Creating an instance of Ticket with sample data ticket_data = Ticket(ticket_id=str(uuid.uuid4()), created_at=datetime.utcnow(), session_id="98766545431", data="Optional")

Note —-= - This class is used to represent ticket data with a specified ticket_id. - It is typically used when retrieving or working with existing ticket data.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.TicketBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `model_config`

Variable `model_fields`

Variable `ticket_id` Type: `str`

Class `TicketBase`

```
class TicketBase(
    **data: Any
)
```

Base model for tickets.

This class defines a base data model for tickets, including attributes such as the ticket creation timestamp, session ID, and additional data.

Attributes —-= - created_at (Optional[datetime]): The timestamp when the ticket was created. Default is the current UTC time. - session_id (Optional[str]): The session ID associated with the ticket. Default is None. - data (Optional[str]): Additional data related to the ticket. Default is None. Example Usage: # Creating an instance of TicketBase with sample data ticket_data = TicketBase(created_at=datetime.utcnow(), session_id="98766545431", data="Optional")

Note —-= - This class serves as a base model for ticket data, and its attributes can be optional to accommodate various use cases.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Descendants

- app.core.models.models.Ticket
- app.core.models.models.TicketCreate

Class variables

Variable `created_at` Type: `Optional[datetime.datetime]`

Variable `data` Type: `Optional[str]`

Variable `model_config`

Variable `model_fields`

Variable `session_id` Type: `Optional[str]`

Class `TicketCreate`

```
class TicketCreate(
    **data: Any
)
```

Model for creating a ticket.

This class extends the base TicketBase model and adds an attribute for the ticket_id, which represents a unique identifier for the ticket being created.

Attributes —-= - ticket_id (Optional[str]): A unique identifier for the ticket. It is generated using the generate_id() function by default. Example Usage: # Creating an instance of TicketCreate with sample data ticket_data = TicketCreate(ticket_id=str(uuid.uuid4()), created_at=datetime.utcnow(), session_id="98766545431", data="Optional")

Note —-= - This class is used to represent ticket data when creating new tickets, and it includes additional attributes beyond the base TicketBase.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- app.core.models.models.TicketBase
- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `model_config`

Variable `model_fields`

Variable `ticket_id` Type: `Optional[str]`

Module `app.core.settings`

Functions

Function `get_client_uri`

```
def get_client_uri(
    USER: str,
    PASSWORD: str,
    HOST: str,
    PORT: str,
    DB: str
) -> str
```

Generate a MongoDB client URI.

This function generates a MongoDB client URI based on the provided parameters, which include the MongoDB user, password, host, port, and database name. The generated URI can be used to establish a connection to the MongoDB server.

Parameters —–= - USER (str): The MongoDB user for authentication. - PASSWORD (str): The password associated with the user. - HOST (str): The hostname or IP address of the MongoDB server. - PORT (str): The port number on which the MongoDB server is running. - DB (str): The name of the MongoDB database to connect to.

Returns —–= str : A MongoDB client URI that can be used to connect to the MongoDB server

with the specified authentication credentials and database. Example: mongodb://{USER}:{PASSWORD}@
Example Usage: client_uri = get_client_uri(“myuser”, “mypassword”, “localhost”, “27017”, “mydb”)

Note —–= - The generated URI includes authentication information and specifies the authentication mechanism and source database.

Classes

Class `APISettings`

```
class APISettings(
    **values: Any
)
```

API settings configuration class.

This class represents the configuration settings for the API. It inherits from BaseSettings provided by the pydantic library and defines various attributes to configure the API behavior.

Attributes —–= - API_KEY (str): The API key used for authentication. Default is “secret”. - ALG (str): The algorithm used for token generation or validation. Default is “HS512”. - model_config (SettingsConfigDict): A configuration dictionary for loading environment variables and settings from a .env file. This can be used to customize the behavior of the API based on environment-specific settings. Example Usage: # Creating an instance of APISettings api_settings = APISettings()

Note —–= - This class is designed to be used for configuring an API and customizing its behavior based on the provided attributes. - The default values of attributes can be overridden by setting environment variables or using a .env file.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic_settings.main.BaseSettings
- pydantic.main.BaseModel

Class variables

Variable `ALG` Type: `str`

Variable `API_KEY` Type: `str`

Variable `model_config` Type: `ClassVar[pydantic_settings.main.SettingsConfigDict]`

Variable `model_fields`

Class `ChatDBSettings`

```
class ChatDBSettings(
    **values: Any
)
```

Chat database settings configuration class.

This class represents the configuration settings for a chat database. It inherits from BaseSettings provided by the pydantic library and defines various attributes to configure the database connection and specify database-related settings.

Attributes —–= - MONGO_CHATLOG_USERNAME (str): The username used for MongoDB authentication. Default is “nonroot”. - MONGO_CHATLOG_PASSWORD (str): The password associated with the MongoDB user. Default is “secret”. - MONGO_CHAT_DB_HOST (str): The hostname or IP address of the MongoDB server. Default is “172.17.0.1”. - MONGO_CHAT_DB_PORT (str): The port number on which the MongoDB server is running. Default is “27017”. - MONGO_CHATLOG_DB_NAME (str): The name of the MongoDB database to connect to. Default is “chatlog”. - MONGO_CHATLOG_COLLECTION_MESSAGES (str): The name of the collection in the MongoDB database where chat messages are stored. - MONGO_CHATLOG_COLLECTION_SESSIONS (str): The name of the collection in the MongoDB database where chat sessions are stored. - MONGO_CHATLOG_COLLECTION_FEEDBACK (str): The name of the collection in the MongoDB database where feedback data is stored. - MONGO_CHATLOG_COLLECTION_TICKETS (str): The name of the collection in the MongoDB database where support tickets are stored. - model_config (SettingsConfigDict): A configuration dictionary for loading environment variables and settings from a .env file. This can be used to customize the behavior of the database connection based on environment-specific settings. Example Usage: # Creating an instance of ChatDBSettings db_settings = ChatDBSettings()

Note —–= - This class is designed to be used for configuring database-related settings for a chat application. - The default values of attributes can be overridden by setting environment variables or using a .env file to customize the database connection settings.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic_settings.main.BaseSettings
- pydantic.main.BaseModel

Class variables

Variable `MONGO_CHATLOG_COLLECTION_FEEDBACK` Type: `str`

Variable `MONGO_CHATLOG_COLLECTION_MESSAGES` Type: `str`

Variable `MONGO_CHATLOG_COLLECTION_SESSIONS` Type: `str`

Variable `MONGO_CHATLOG_COLLECTION_TICKETS` Type: `str`

Variable `MONGO_CHATLOG_DB_NAME` Type: `str`

Variable `MONGO_CHATLOG_PASSWORD` Type: `str`

Variable `MONGO_CHATLOG_USERNAME` Type: `str`

Variable `MONGO_CHAT_DB_HOST` Type: `str`

Variable `MONGO_CHAT_DB_PORT` Type: `str`

Variable `model_config` Type: `ClassVar[pydantic_settings.main.SettingsConfigDict]`

Variable `model_fields`

Class `MetricsDBSettings`

```
class MetricsDBSettings(
    **values: Any
)
```

Metrics database settings configuration class.

This class represents the configuration settings for a metrics database. It inherits from BaseSettings provided by the pydantic library and defines various attributes to configure the metrics database and specify types of metrics data.

Attributes —-= - MONGO_MESSAGES_TYPE (str): The type identifier for messages metrics in the metrics database. - MONGO_SESSIONS_TYPE (str): The type identifier for sessions metrics in the metrics database. - MONGO_TICKETS_TYPE (str): The type identifier for support tickets metrics in the metrics database. - MONGO_TICKETSP_TYPE (str): The type identifier for premium support tickets metrics in the metrics database. - MONGO_AVG_MESSAGES_TYPE (str): The type identifier for average messages metrics in the metrics database. - MONGO_KEYWORDS_TYPE (str): The type identifier for keywords metrics in the metrics database. - MONGO_TOPICS_TYPE (str): The type identifier for topics metrics in the metrics database. - model_config (SettingsConfigDict): A configuration dictionary for loading environment variables and settings from a .env file. This can be

used to customize the behavior of the metrics database based on environment-specific settings. Example Usage: # Creating an instance of MetricsDBSettings metrics_settings = MetricsDBSettings()

Note —-= - This class is designed to be used for configuring settings related to a metrics database, including the types of metrics data stored. - The default values of attributes can be overridden by setting environment variables or using a .env file to customize the metrics database configuration.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic_settings.main.BaseSettings
- pydantic.main.BaseModel

Class variables

Variable `MONGO_AVG_MESSAGES_TYPE` Type: `str`

Variable `MONGO_KEYWORDS_TYPE` Type: `str`

Variable `MONGO_MESSAGES_TYPE` Type: `str`

Variable `MONGO_SESSIONS_TYPE` Type: `str`

Variable `MONGO_TICKETSP_TYPE` Type: `str`

Variable `MONGO_TICKETS_TYPE` Type: `str`

Variable `MONGO_TOPICS_TYPE` Type: `str`

Variable `model_config` Type: `ClassVar[pydantic_settings.main.SettingsConfigDict]`

Variable `model_fields`

Class `Settings`

```
class Settings(
    **values: Any
)
```

Application settings configuration class.

This class represents the configuration settings for an application. It inherits from BaseSettings provided by the pydantic library and defines attributes to configure various aspects of the application, including database settings, API settings, metrics settings, and a configuration dictionary for loading environment variables from a .env file.

Attributes —-= - db (ChatDBSettings): An instance of ChatDBSettings that holds database-related configuration settings. - api (APISettings): An instance of APISettings that holds

API-related configuration settings. - metrics (MetricsDBSettings): An instance of MetricsDBSettings that holds metrics database-related configuration settings. - model_config (SettingsConfigDict): A configuration dictionary for loading environment variables and settings from a .env file. This can be used to customize various aspects of the application's behavior based on environment- specific settings. Example Usage: # Creating an instance of Settings app_settings = Settings()

Note —-= - This class is designed to be used for configuring various aspects of an application, including its database, API, and metrics settings. - It provides a structured way to access and manage application configuration. - The default values of attributes can be overridden by setting environment variables or using a .env file to customize the application's behavior.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

__init__ uses __pydantic_self__ instead of the more common self for the first arg to allow self as a field name.

Ancestors (in MRO)

- pydantic_settings.main.BaseSettings
- pydantic.main.BaseModel

Class variables

Variable `api` Type: `app.core.settings.APISettings`

Variable `db` Type: `app.core.settings.ChatDBSettings`

Variable `metrics` Type: `app.core.settings.MetricsDBSettings`

Variable `model_config` Type: `ClassVar[pydantic_settings.main.SettingsConfigDict]`

Variable `model_fields`

Module `app.main`

Module `app.v1`

Sub-modules

- app.v1.app
- app.v1.routers
- app.v1.utils

Module `app.v1.app`

Functions

Function `read_items`

```
async def read_items()
```

Module app.v1.routers

Sub-modules

- app.v1.routers.feedback
- app.v1.routers.messages
- app.v1.routers.metrics
- app.v1.routers.sessions
- app.v1.routers.tickets

Module app.v1.routers.feedback

Functions

Function create_feedback

```
def create_feedback(
    feedback: core.models.models.FeedbackCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Feedback
```

Create a new feedback record in the database.

This endpoint allows you to create a new feedback record by providing the necessary data in the request body. The feedback data will be inserted into the database and a response containing the newly created feedback will be returned.

Parameters —-= - feedback (FeedbackCreate): The feedback data to be created. - client (MongoClient, optional): An optional MongoDB client obtained from the dependency injection. If not provided, a new client will be created.

Returns —-= FeedbackModel : A response model containing the newly created feedback data.

Raises —-= HTTPException (status_code=500): If an error occurs while inserting the feedback data into the database, an internal server error is raised, and the error details are provided in the response.

HTTPException (status_code=404): If the feedback data could not be found in the database, a not found error is raised, and the error details are provided in the response.

Note —-= - The database operation is performed within a transactional session to ensure atomicity and consistency. - If a new MongoDB client is created, it will be automatically closed after processing the request. - The MongoDB collection used for storing feedback data is specified in the application settings (settings.db.MONGO_CHATLOG_COLLECTION_FEEDBACK).

Function delete_feedback

```
def delete_feedback(
    feedback_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Feedback
```

Delete an existing feedback record from the database.

This endpoint allows you to delete an existing feedback record from the database by providing its unique identifier (feedback_id) as a path parameter. The specified feedback record will be retrieved from the database and then deleted. A response containing the deleted feedback data will be returned.

Parameters —–= - feedback_id (str): The unique identifier of the feedback record to delete. - client (MongoClient, optional): An optional MongoDB client obtained from the dependency injection. If not provided, a new client will be created.

Returns —–= FeedbackModel : A response model containing the deleted feedback data.

Raises —–= HTTPException (status_code=400): If the feedback_id parameter is invalid, the request data is invalid, or the request is malformed, a bad request error is raised, and the error details are provided in the response.

HTTPException (status_code=404): If the specified feedback record is not found in the database, a not found error is raised, and the error details are provided in the response.

HTTPException (status_code=500): If an unexpected server error occurs while querying or deleting the database record, an internal server error is raised, and the error details are provided in the response.

Note —–= - The database query and delete operations are performed within a transactional session to ensure atomicity and consistency. - If a new MongoDB client is created, it will be automatically closed after processing the request. - The MongoDB collection used for storing feedback data is specified in the application settings (settings.db.MONGO_CHATLOG_COLLECTION_FEEDBACK).

Function `list_feedbacks`

```
def list_feedbacks(
    skip: int = 0,
    limit: int = 10,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> List[core.models.models.Feedback]
```

List feedback records from the database.

This endpoint allows you to retrieve a list of feedback records from the database with optional pagination parameters (skip and limit). The retrieved feedback records are returned as a list of FeedbackModel objects in the response.

Parameters —–= - skip (int, optional): The number of records to skip before starting to return feedback records. Default is 0. - limit (int, optional): The maximum number of feedback records to return in the response. Default is 10. - client (MongoClient, optional): An optional MongoDB client obtained from the dependency injection. If not provided, a new client will be created.

Returns —–= List[FeedbackModel] : A list of FeedbackModel objects containing the retrieved

feedback records.

Raises —–= HTTPException (status_code=400): If the skip or limit parameters are invalid or the request is malformed, a bad request error is raised, and the error details are provided in the response.

HTTPException (status_code=404): If no feedback records are found in the database based on the specified skip and limit parameters, a not found error is raised, and the error details are provided in the response.

HTTPException (status_code=500): If an unexpected server error occurs while querying the database, an internal server error is raised, and the error details are provided in the response.

Note —–= - The database query is performed within a transactional session to ensure atomicity and consistency. - If a new MongoDB client is created, it will be automatically closed after processing the request. - The MongoDB collection used for storing feedback data is specified in the application settings (settings.db.MONGO_CHATLOG_COLLECTION_FEEDBACK).

Function `read_feedback`

```
def read_feedback(
    feedback_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Feedback
```

Retrieve a feedback record by its unique identifier.

This endpoint allows you to retrieve a feedback record from the database by providing its unique identifier (feedback_id) as a path parameter. The specified feedback record will be retrieved from the database using a MongoDB client, and a response containing the retrieved feedback data will be returned.

Parameters —–= - feedback_id (str): The unique identifier of the feedback record to retrieve. - client (MongoClient, optional): An optional MongoDB client obtained from the dependency injection. If not provided, a new client will be created.

Returns —–= FeedbackModel : A response model containing the retrieved feedback data.

Raises —–= HTTPException (status_code=400): If the feedback_id parameter is invalid or the request is malformed, a bad request error is raised, and the error details are provided in the response.

HTTPException (status_code=404): If the specified feedback record is not found in the database, a not found error is raised, and the error details are provided in the response.

HTTPException (status_code=500): If an unexpected server error occurs while querying the database, an internal server error is raised, and the error details are provided in the response.

Note —–= - The database query is performed within a transactional session to ensure atomicity and consistency. - If a new MongoDB client is created, it will be automatically closed after processing the request. - The MongoDB collection used for storing feedback data is specified in the application settings (settings.db.MONGO_CHATLOG_COLLECTION_FEEDBACK).

Function `update_feedback`

```
def update_feedback(
    feedback_id: str,
    message: core.models.models.FeedbackCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Feedback
```

Update an existing feedback record in the database.

This endpoint allows you to update an existing feedback record in the database by providing its unique identifier (feedback_id) as a path parameter and the updated feedback data in the request body (message). The specified feedback record will be retrieved from the database, and the provided updates will be applied. The updated feedback data will then be saved to the database, and a response containing the updated feedback will be returned.

Parameters —–= - feedback_id (str): The unique identifier of the feedback record to update. - message (FeedbackCreate): The updated feedback data to be applied. - client (MongoClient, optional): An optional MongoDB client obtained from the dependency injection. If not provided, a new client will be created.

Returns —–= FeedbackModel : A response model containing the updated feedback data.

Raises —–= HTTPException (status_code=400): If the feedback_id parameter is invalid, the request data is invalid, or the request is malformed, a bad request error is raised, and the error details are provided in the response.

HTTPException (status_code=404): If the specified feedback record is not found in the database, a not found error is raised, and the error details are provided in the response.

HTTPException (status_code=500): If an unexpected server error occurs while querying or updating the database, an internal server error is raised, and the error details are provided in the response.

Note —-= - The database query and update operations are performed within a transactional session to ensure atomicity and consistency. - If a new MongoDB client is created, it will be automatically closed after processing the request. - The MongoDB collection used for storing feedback data is specified in the application settings (settings.db.MONGO_CHATLOG_COLLECTION_FEEDBACK).

Module `app.v1.routers.messages`

Functions

Function `create_message`

```
def create_message(
    message: core.models.models.MessageCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Message
```

Create a new chat message.

This endpoint allows the creation of a new chat message. It inserts the message data into the database collection specified in the settings.

Args —-= - message (MessageCreate): The message data to be created. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - MessageModel: The created chat message.

Raises —-= - HTTPException: If there is an issue with inserting the data into the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: POST /messages/ { “message_id”: “abcdef12345”, “created_at”: “2023-09-22T12:34:56.789Z”, “exchange”: “chat from the user or bot”, “message_type”: “AI/USER”, “session_id”: “98766545431” }

Function `delete_message`

```
def delete_message(
    message_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Message
```

Delete a chat message by its ID.

This endpoint allows deleting a chat message by its unique identifier (message_id) from the database collection specified in the settings.

Args —-= - message_id (str): The unique identifier of the message to delete. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - MessageModel: The deleted chat message.

Raises —-= - HTTPException 404: If the message with the given message_id is not found in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with deleting the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: DELETE /messages/abcdef12345

Function `list_messages`

```
def list_messages(
    skip: int = 0,
    limit: int = 10,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> List[core.models.models.Message]
```

List chat messages.

This endpoint allows listing chat messages with optional pagination (skip and limit). It retrieves messages from the database collection specified in the settings.

Args —–= - skip (int, optional): The number of messages to skip (pagination). Defaults to 0. - limit (int, optional): The maximum number of messages to return (pagination). Defaults to 10. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - List[MessageModel]: A list of chat messages.

Raises —–= - HTTPException 404: If no messages are found in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with retrieving the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: GET /messages/?skip=0&limit=10

Function `read_message`

```
def read_message(
    message_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Message
```

Retrieve a chat message by its ID.

This endpoint allows retrieving a chat message by its unique identifier (message_id) from the database collection specified in the settings.

Args —–= - message_id (str): The unique identifier of the message to retrieve. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - MessageModel: The retrieved chat message.

Raises —–= - HTTPException 404: If the message with the given message_id is not found in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with retrieving the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: GET /messages/abcdef12345

Function `update_message`

```
def update_message(
    message_id: str,
    message: core.models.models.MessageCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Message
```

Update a chat message by its ID.

This endpoint allows updating a chat message by its unique identifier (message_id) in the database collection specified in the settings.

Args —–= - message_id (str): The unique identifier of the message to update. - message (MessageCreate): The updated message data. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - MessageModel: The updated chat message.

Raises —-= - HTTPException 404: If the message with the given message_id is not found in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with updating the data in the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: PUT /messages/abcdef12345 { “message_id”: “abcdef12345”, “created_at”: “2023-09-22T12:34:56.789Z”, “exchange”: “chat from the user or bot”, “message_type”: “AI/USER”, “session_id”: “98766545431” }

Module `app.v1.routers.metrics`

Functions

Function `read_metric`

```
async def read_metric(
    metric_type: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> Any
```

Read a specific metric data.

This endpoint allows reading a specific metric’s data based on the provided metric_type. The metric_type should match one of the keys in the FUNC_MAPPING dictionary, which maps metric types to corresponding async functions for data retrieval.

Args —-= - metric_type (str): The type of metric to retrieve (e.g., “messages”, “sessions”). - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - dict: The metric data, typically hourly values.

Raises —-= - HTTPException 404: If the provided metric_type does not match any known metrics in FUNC_MAPPING, it raises an HTTPException with a status code of 404 (Not Found). Example Usage: GET /metrics/messages

Notes —-= - This endpoint retrieves metric data based on the specified metric_type. - The FUNC_MAPPING dictionary maps metric types to corresponding async functions responsible for data retrieval. - If the metric_type is not found in FUNC_MAPPING, it raises a 404 status code.

Module `app.v1.routers.sessions`

Functions

Function `create_session`

```
def create_session(
    session_data: core.models.models.SessionCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Session
```

Create a new session.

This endpoint allows the creation of a new session. It inserts the session data into the database collection specified in the settings.

Args —-= - session_data (SessionCreate): The session data to be created. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - SessionModel: The created session.

Raises —-= - HTTPException: If there is an issue with inserting the data into the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: POST /sessions/ { “session_id”: “abcdef12345”, “created_at”: “2023-09-22T12:34:56.789Z”, “ended_at”: null, “user_email”: “user@email.com”, “user_name”: “John Doe”, “user_role”: “Owner/Applicant/Realtor”, “data_user_consent”: true, “meta-data”: { “ip”: “111.111.11.111”, “device”: “Mobile/Desktop”, “browser”: “Chrome”, “os”: “Windows/Linux/MacOS” } }

Function `delete_session`

```
def delete_session(
    session_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Session
```

Delete a session.

This endpoint allows deleting a specific session based on the provided session_id.

Args —-= - session_id (str): The ID of the session to delete. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - SessionModel: The deleted session.

Raises —-= - HTTPException 404: If the specified session_id does not exist in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with deleting the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: DELETE /sessions/abcdef12345

Function `get_messages_for_session`

```
def get_messages_for_session(
    session_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> List[core.models.models.Message]
```

Get messages for a session.

This endpoint allows retrieving messages associated with a specific session based on the provided session_id.

Args —-= - session_id (str): The ID of the session to retrieve messages for. - client (Mongo-Client, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - List[MessageModel]: A list of messages associated with the session.

Raises —-= - HTTPException 404: If there are no messages found for the session, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with retrieving the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: GET /sessions/abcdef12345/messages/

Function `list_sessions`

```
def list_sessions(
    skip: int = 0,
    limit: int = 10,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> List[core.models.models.Session]
```

List all sessions.

This endpoint allows listing all sessions with optional pagination.

Args —–= - skip (int, optional): The number of sessions to skip. Defaults to 0. - limit (int, optional): The maximum number of sessions to retrieve. Defaults to 10. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - List[SessionModel]: A list of sessions.

Raises —–= - HTTPException 404: If there are no sessions found, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with retrieving the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: GET /sessions/?skip=0&limit=10

Function `read_session`

```
def read_session(
    session_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Session
```

Read a session.

This endpoint allows reading a specific session's data based on the provided session_id.

Args —–= - session_id (str): The ID of the session to retrieve. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - SessionModel: The retrieved session.

Raises —–= - HTTPException 404: If the specified session_id does not exist in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with retrieving the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: GET /sessions/abcdef12345

Function `update_session`

```
def update_session(
    session_id: str,
    session_data: core.models.models.SessionCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Session
```

Update a session.

This endpoint allows updating a specific session's data based on the provided session_id.

Args —–= - session_id (str): The ID of the session to update. - session_data (SessionCreate): The updated session data. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - SessionModel: The updated session.

Raises —–= - HTTPException 404: If the specified session_id does not exist in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with updating the data in the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: PUT /sessions/abcdef12345 { "session_id": "abcdef12345", "created_at": "2023-09-22T12:34:56.789Z", "ended_at": "2023-09-22T13:00:00.000Z", "user_email": "updated_email@email.com", "user_name": "Updated Name", "user_role": "Updated Role", "data_user_consent": true, "metadata": { "ip": "111.111.11.111", "device": "Mobile/Desktop", "browser": "Chrome", "os": "Windows/Linux/MacOS" } }

Module `app.v1.routers.tickets`

Functions

Function `create_ticket`

```
def create_ticket(
    ticket: core.models.models.TicketCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Ticket
```

Create a new ticket.

This endpoint allows creating a new ticket with the provided ticket data.

Args —–= - ticket (TicketCreate): The ticket data to create a new ticket. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - TicketModel: The created ticket.

Raises —–= - HTTPException 500: If there is an issue with inserting the data into the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: POST /tickets/ Body: { “ticket_id”: “abcdef12345”, “created_at”: “2023-09-22T12:00:00”, “session_id”: “98766545431”, “data”: “Optional” }

Function `delete_ticket`

```
def delete_ticket(
    ticket_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Ticket
```

Delete a ticket.

This endpoint allows deleting a specific ticket based on the provided ticket_id.

Args —–= - ticket_id (str): The ID of the ticket to delete. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - TicketModel: The deleted ticket.

Raises —–= - HTTPException 404: If the specified ticket_id does not exist in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with deleting the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: DELETE /tickets/abcdef12345

Function `list_tickets`

```
def list_tickets(
    skip: int = 0,
    limit: int = 10,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> List[core.models.models.Ticket]
```

List all tickets.

This endpoint allows listing all the tickets in the database with optional pagination.

Args —–= - skip (int, optional): The number of tickets to skip before returning results. Defaults to 0. - limit (int, optional): The maximum number of tickets to return. Defaults to 10. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —–= - List[TicketModel]: A list of tickets.

Raises —-= - HTTPException 404: If there are no tickets found in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with retrieving the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: GET /tickets/

Function `read_ticket`

```
def read_ticket(
    ticket_id: str,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Ticket
```

Get a ticket by ID.

This endpoint allows retrieving a specific ticket based on the provided ticket_id.

Args —-= - ticket_id (str): The ID of the ticket to retrieve. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - TicketModel: The retrieved ticket.

Raises —-= - HTTPException 404: If the specified ticket_id does not exist in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with retrieving the data from the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: GET /tickets/abcdef12345

Function `update_ticket`

```
def update_ticket(
    ticket_id: str,
    message: core.models.models.TicketCreate,
    client: pymongo.mongo_client.MongoClient = Depends(get_client)
) -> core.models.models.Ticket
```

Update a ticket.

This endpoint allows updating a specific ticket based on the provided ticket_id.

Args —-= - ticket_id (str): The ID of the ticket to update. - ticket_data (TicketCreate): The updated ticket data. - client (MongoClient, optional): The MongoDB client obtained from the dependency get_client. Defaults to None.

Returns —-= - TicketModel: The updated ticket.

Raises —-= - HTTPException 404: If the specified ticket_id does not exist in the database, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with updating the data in the database, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: PUT /tickets/abcdef12345 Body: { “created_at”: “2023-09-22T13:00:00”, “session_id”: “98766545431”, “data”: “Updated data” }

Module `app.v1.utils`

Sub-modules

- app.v1.utils.metric_utils
- app.v1.utils.utils

Module `app.v1.utils.metric_utils`

Functions

Function `daily_avg_messages_estimation`

```
async def daily_avg_messages_estimation(
    client: pymongo.mongo_client.MongoClient
)
```

Estimate daily average messages per session.

This asynchronous function calculates and returns the daily average messages per session from messages and sessions in the database.

Args —–= - client (MongoClient): The MongoDB client obtained from the dependency get_client.

Returns —–= - str: JSON representation of daily average messages per session.

Raises —–= - HTTPException 404: If there are no messages or sessions found for estimation, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with estimating the metrics, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: await daily_avg_messages_estimation(client)

Function `daily_messages_estimation`

```
async def daily_messages_estimation(
    client: pymongo.mongo_client.MongoClient
)
```

Estimate daily message counts from USER messages.

This asynchronous function calculates and returns the daily message counts from messages of type “USER” in the database.

Args —–= - client (MongoClient): The MongoDB client obtained from the dependency get_client.

Returns —–= - str: JSON representation of daily message counts.

Raises —–= - HTTPException 404: If there are no messages found for estimation, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with estimating the metrics, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: await daily_messages_estimation(client)

Function `daily_sessions_estimation`

```
async def daily_sessions_estimation(
    client: pymongo.mongo_client.MongoClient
)
```

Estimate daily session counts.

This asynchronous function calculates and returns the daily session counts from sessions in the database.

Args —–= - client (MongoClient): The MongoDB client obtained from the dependency get_client.

Returns —–= - str: JSON representation of daily session counts.

Raises —–= - HTTPException 404: If there are no sessions found for estimation, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is

an issue with estimating the metrics, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: await daily_sessions_estimation(client)

Function `daily_tickets_estimation`

```
async def daily_tickets_estimation(
    client: pymongo.mongo_client.MongoClient
)
```

Estimate daily ticket counts.

This asynchronous function calculates and returns the daily ticket counts from tickets in the database.

Args —-= - client (MongoClient): The MongoDB client obtained from the dependency get_client.

Returns —-= - str: JSON representation of daily ticket counts.

Raises —-= - HTTPException 404: If there are no tickets found for estimation, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with estimating the metrics, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: await daily_tickets_estimation(client)

Function `daily_ticketsp_estimation`

```
async def daily_ticketsp_estimation(
    client
)
```

Estimate daily tickets per session.

This asynchronous function calculates and returns the daily tickets per session from tickets and sessions in the database.

Args —-= - client (MongoClient): The MongoDB client obtained from the dependency get_client.

Returns —-= - str: JSON representation of daily tickets per session.

Raises —-= - HTTPException 404: If there are no tickets or sessions found for estimation, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with estimating the metrics, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: await daily_ticketsp_estimation(client)

Function `keywords_estimation`

```
async def keywords_estimation(
    client: pymongo.mongo_client.MongoClient
)
```

Estimate keywords from user messages.

This asynchronous function calculates and returns a JSON representation of keywords found in user messages in the database.

Args —-= - client (MongoClient): The MongoDB client obtained from the dependency get_client.

Returns —-= - str: JSON representation of keywords.

Raises —-= - HTTPException 404: If there are no user messages found for keyword estimation, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with estimating the metrics, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: await keywords_estimation(client)

Function `topics_estimation`

```
async def topics_estimation(
    client: pymongo.mongo_client.MongoClient
)
```

Estimate topics from AI messages.

This asynchronous function calculates and returns a JSON representation of topics found in AI messages in the database.

Args —-= - client (MongoClient): The MongoDB client obtained from the dependency get_client.

Returns —-= - str: JSON representation of topics.

Raises —-= - HTTPException 404: If there are no AI messages found for topic estimation, it raises an HTTPException with a status code of 404 (Not Found). - HTTPException 500: If there is an issue with estimating the metrics, it raises an HTTPException with a status code of 500 (Internal Server Error). Example Usage: await topics_estimation(client)

Module `app.v1.utils.utils`

Functions

Function `get_client`

```
def get_client()
```

Function `get_client_uri`

```
def get_client_uri(
    USER: Optional[str] = 'nonroot',
    PASSWORD: Optional[str] = 'secret',
    HOST: Optional[str] = '172.191.159.98',
    PORT: Optional[str] = '27017',
    DB: Optional[str] = 'chatlog'
) -> str
```

Function `has_access`

```
async def has_access(
    credentials: fastapi.security.http.HTTPAuthorizationCredentials = Depends(HTTPBearer)
)
```

Function that is used to validate the token in the case that it requires it

---