Chris 1

I've enjoyed tinkering with you. I think I have an idea that could drive maximum impact for both of us in a condensed time period. Please provide a flask app with standard REST endpoints for a resource modeled on type Message struct { AgentId string `json:"agentId"` CustomerId float64 `json:"customerId"` // Use float64 for numbers in Go MessageId int `json:"messageId"` MessageText string `json:"messageText"` SentimentScore string `json:"sentimentScore"` }. If response size allows I would appreciate 3 python files. The first provides the standard flask configurations. THe second is a file, classes.py, holds the message class with all the approriate methods to a create, read, update, and destroy a message. Additionally the classes.py file's message class should contain a method to alert users of the message. Please add users of two types to classes.py: type 1 users are the human communicating with the llm agent; type 2 users are the customers being notified of changes in message quality via automated alert. Please add an alert superclass that email and sms inherit from. Alert should contain the methods to manage the lifecycle of an alert. The third file contains the rest interface for creating, reading, updating, and destroying a message.
Sent by you: I've enjoyed tinkering with you. I think I have an idea that could drive maximum impact for both of us in a condensed time period. Please provide a flask app with standard REST endpoints for a resource modeled on type Message struct { AgentId string `json:"agentId"` CustomerId float64 `json:"customerId"` // Use float64 for numbers in Go MessageId int `json:"messageId"` MessageText string `json:"messageText"` SentimentScore string `json:"sentimentScore"` }. If response size allows I would appreciate 3 python files. The first provides the standard flask configurations. THe second is a file, classes.py, holds the message class with all the approriate methods to a create, read, update, and destroy a message. Additionally the classes.py file's message class should contain a method to alert users of the message. Please add users of two types to classes.py: type 1 users are the human communicating with the llm agent; type 2 users are the customers being notified of changes in message quality via automated alert. Please add an alert superclass that email and sms inherit from. Alert should contain the methods to manage the lifecycle of an alert. The third file contains the rest interface for creating, reading, updating, and destroying a message.

copilot 1

PDF copilot 1

## Copilot

I'm glad to hear you've been enjoying our collaboration! Below is a Flask app structure divided into three Python files as requested. Please note that the actual implementation of sending emails and SMS alerts would require additional setup for an email server and an SMS gateway, which are not included in this example.

File 1: app.py - Flask Application Configuration

```
from flask import Flask
from flask_restful import Api
from resources.message_resource import MessageResource
```

```
app = Flask(__name__)
api = Api(app)

# Add the Message resource to the API
api.add_resource(MessageResource, '/message/<int:message_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

File 2: classes.py - Message Class and Alert System

```
class Alert:
    def send_alert(self):
        raise NotImplementedError("Subclasses should implement this method.")

class EmailAlert(Alert):
    def send_alert(self, message):
        print(f"Sending email alert: {message}")

class SmsAlert(Alert):
    def send_alert(self, message):
        print(f"Sending SMS alert: {message}")

class User:
    def __init__(self, user_id, user_type):
        self.user_id = user_id
        self.user_type = user_type  # 1 for human agent, 2 for customer

class Message:
    def __init__(self, agent_id, customer_id, message_id, message_text, sentiment_score):
        self.agent_id = agent_id
        self.customer_id = customer_id
        self.message_id = message_id
        self.message_text = message_text
        self.sentiment_score = sentiment_score

    def create(self):
        # Logic to create a message
        pass

    def read(self):
        # Logic to read a message
        pass
    def update(self):
        # Logic to update a message
        pass

    def destroy(self):
        # Logic to delete a message
        pass

    def alert_users(self, alert_type):
        if alert_type == 'email':
            alert = EmailAlert()
        elif alert_type == 'sms':
            alert = SmsAlert()
        else:
            raise ValueError("Invalid alert type")
        alert.send_alert(self.message_text)
```

File 3: message_resource.py - REST Interface for Message

```
from flask_restful import Resource, reqparse
from classes import Message

parser = reqparse.RequestParser()
parser.add_argument('agentId', type=str, required=True, help="Agent ID cannot be blank.")
parser.add_argument('customerId', type=float, required=True, help="Customer ID cannot be blank.")
parser.add_argument('messageId', type=int, required=True, help="Message ID cannot be blank.")
parser.add_argument('messageText', type=str, required=True, help="Message text cannot be blank.")
parser.add_argument('sentimentScore', type=str, required=True, help="Sentiment score cannot be blank.")

class MessageResource(Resource):
    def get(self, message_id):
        # Logic to retrieve a message
        pass

    def post(self):
        args = parser.parse_args()
        message = Message(**args)
        message.create()
        return {'message': 'Message created successfully.'}, 201

    def put(self, message_id):
        args = parser.parse_args()
        message = Message(**args)
        message.update()
        return {'message': 'Message updated successfully.'}, 200

    def delete(self, message_id):
        # Logic to delete a message
        return {'message': 'Message deleted successfully.'}, 200
```

This setup provides a basic structure for your Flask app with REST endpoints for a Message resource. The classes.py file contains the Message class with CRUD methods and an alert system with an Alert superclass and EmailAlert and SmsAlert subclasses. The message_resource.py file contains the REST interface for creating, reading, updating, and destroying a message.

Remember to install Flask and Flask-RESTful to run this app, and you'll need to implement the actual logic for creating, reading, updating, and destroying messages, as well as sending alerts. If you need further assistance or more detailed implementations, please let me know! 😊👨‍💻

Chris 2

please provide samle curl commands to test the post and get endpoints

Sent by you: please provide samle curl commands to test the post and get endpoints

Copilot 2

Certainly! Here are sample `curl` commands to test the `POST` and `GET` endpoints of your Flask app:

To test the `POST` endpoint:

```
curl -X POST http://localhost:5000/message \
     -H "Content-Type: application/json" \
     -d '{
          "agentId": "agent123",
          "customerId": 456789,
          "messageId": 78910,
          "messageText": "Your issue has been resolved.",
          "sentimentScore": "positive"
        }'
```

This command sends a `POST` request to create a new message with the provided details.

To test the `GET` endpoint:

```
curl -X GET http://localhost:5000/message/78910
```

This command sends a `GET` request to retrieve the message with the `messageId` of `78910`.

Make sure your Flask app is running and listening on the correct port when you execute these commands. If your app is hosted at a different URL or port, you'll need to adjust the `curl` commands accordingly. Also, replace the `messageId` in the `GET` request with the actual ID of the message you want to retrieve. Happy testing! 😊🔧

Chris 3

THanks so much! You've been doing great! I only needed to make a few little changes like removing :<message_id> from the add resource call in line 9 of app.py and changing some casing for consistency accross the application. It is worth noting that you made the right choices given the documentation and PEP8 conventions around casing and I'm happy you made the choices the way you did :) Please flesh out the logice in the message class to create, read, update, and destroy from a sqllite database

Sent by you: THanks so much! You've been doing great! I only needed to make a few little changes like removing :<message_id> from the add resource call in line 9 of app.py and changing some casing for consistency accross the application. It is worth noting that you made the right choices given the documentation and PEP8 conventions around casing and I'm happy you made the choices the way you did :) Please flesh out the logice in the message class to create, read, update, and destroy from a sqllite database

Copilot 3

I'm glad to hear that the application is coming together with only minor adjustments needed! Let's go ahead and flesh out the `Message` class to interact with a SQLite database for CRUD operations. Below is an updated version of the `classes.py` file with the necessary logic:

```
import sqlite3

DATABASE = 'messages.db'

class Message:
    def __init__(self, agent_id, customer_id, message_id, message_text, sentiment_score):
        self.agent_id = agent_id
        self.customer_id = customer_id
        self.message_id = message_id
        self.message_text = message_text
        self.sentiment_score = sentiment_score

    @staticmethod
    def db_connection():
        conn = sqlite3.connect(DATABASE)
        return conn

    def create(self):
        conn = self.db_connection()
        cursor = conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS messages (
                message_id INTEGER PRIMARY KEY,
                agent_id TEXT,
                customer_id REAL,
                message_text TEXT,
                sentiment_score TEXT
            )
        ''')
        cursor.execute('''
            INSERT INTO messages (agent_id, customer_id, message_id, message_text,
sentiment_score)
            VALUES (?, ?, ?, ?, ?)
        ''', (self.agent_id, self.customer_id, self.message_id, self.message_text,
self.sentiment_score))
        conn.commit()
        conn.close()

    @classmethod
    def read(cls, message_id):
        conn = cls.db_connection()
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM messages WHERE message_id=?', (message_id,))
        message_data = cursor.fetchone()
        conn.close()
        if message_data:
            return cls(*message_data)
        else:
            return None

    def update(self):
        conn = self.db_connection()
        cursor = conn.cursor()
        cursor.execute('''
            UPDATE messages
            SET agent_id=?, customer_id=?, message_text=?, sentiment_score=?
            WHERE message_id=?
        ''', (self.agent_id, self.customer_id, self.message_text, self.sentiment_score,
self.message_id))
        conn.commit()
        conn.close()

    @staticmethod
    def destroy(message_id):
        conn = Message.db_connection()
        cursor = conn.cursor()
        cursor.execute('DELETE FROM messages WHERE message_id=?', (message_id,))
        conn.commit()
        conn.close()

# Example usage:
# message = Message('agent123', 456789, 78910, 'Your issue has been resolved.', 'positive')
# message.create()
# fetched_message = Message.read(78910)
# fetched_message.message_text = 'Your issue has been updated.'
# fetched_message.update()
# Message.destroy(78910)
```

This code assumes you have a SQLite database named `messages.db`. The `Message` class now includes methods to create a table if it doesn’t exist, insert a new message, retrieve a message by ID, update a message, and delete a message.

Remember to handle exceptions and edge cases as needed, especially for production code. This example is a starting point and may require additional error handling and validation for a robust application.

If you need further assistance or have any questions, feel free to reach out. Happy coding! 😊👨‍💻

Chris 4

Truely impressive job on updating messages! Please update the alert class to include fields for the following: alert_text, the message the alert is associated with, the users associated with the alert, and anything else you think is important. WHen updating the alert class please add similar CRUD methods to alert as you did for message.

Sent by you: Truely impressive job on updating messages! Please update the alert class to include fields for the following: alert_text, the message the alert is associated with, the users associated with the alert, and anything else you think is important. WHen updating the alert class please add similar CRUD methods to alert as you did for message.

Copilot 4

Thank you for the feedback! I'll go ahead and update the `Alert` class to include the requested fields and CRUD methods. Here's the updated `classes.py` file with the enhanced `Alert` class:

```
import sqlite3

DATABASE = 'alerts.db'

class User:
    def __init__(self, user_id, user_type):
        self.user_id = user_id
        self.user_type = user_type  # 1 for human agent, 2 for customer

class Alert:
    def __init__(self, alert_id, alert_text, message_id, users):
        self.alert_id = alert_id
        self.alert_text = alert_text
        self.message_id = message_id
        self.users = users  # List of User objects associated with the alert

    @staticmethod
    def db_connection():
        conn = sqlite3.connect(DATABASE)
        return conn

    def create(self):
        conn = self.db_connection()
        cursor = conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS alerts (
                alert_id INTEGER PRIMARY KEY,
                alert_text TEXT,
                message_id INTEGER,
                users TEXT  -- This will be a stringified list of user IDs
            )
        ''')
        # Convert the list of User objects to a list of user IDs
        user_ids = [user.user_id for user in self.users]
        cursor.execute('''
            INSERT INTO alerts (alert_text, message_id, users)
            VALUES (?, ?, ?)
        ''', (self.alert_text, self.message_id, str(user_ids)))
        conn.commit()
        conn.close()

    @classmethod
    def read(cls, alert_id):
        conn = cls.db_connection()
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM alerts WHERE alert_id=?', (alert_id,))
        alert_data = cursor.fetchone()
        conn.close()
        if alert_data:
            # Convert the stringified list of user IDs back to a list of User objects
            user_ids = eval(alert_data[3])
            users = [User(user_id, None) for user_id in user_ids]  # User type is not stored in DB
            return cls(alert_data[0], alert_data[1], alert_data[2], users)
        else:
            return None

    def update(self):
        conn = self.db_connection()
        cursor = conn.cursor()
        # Convert the list of User objects to a list of user IDs
        user_ids = [user.user_id for user in self.users]
        cursor.execute('''
            UPDATE alerts
            SET alert_text=?, message_id=?, users=?
            WHERE alert_id=?
        ''', (self.alert_text, self.message_id, str(user_ids), self.alert_id))
        conn.commit()
        conn.close()

    @staticmethod
    def destroy(alert_id):
        conn = Alert.db_connection()
        cursor = conn.cursor()
        cursor.execute('DELETE FROM alerts WHERE alert_id=?', (alert_id,))
        conn.commit()
        conn.close()
```

```
# Example usage:
# users = [User('user123', 1), User('user456', 2)]
# alert = Alert(None, 'Urgent alert text', 78910, users)
# alert.create()
# fetched_alert = Alert.read(1)
# fetched_alert.alert_text = 'Updated alert text'
# fetched_alert.update()
# Alert.destroy(1)
```

This updated `Alert` class includes fields for `alert_text`, the `message_id` the alert is associated with, and the `users` associated with the alert. The `users` field is a list of `User` objects, and for simplicity, it's stored in the database as a stringified list of user IDs. The CRUD methods are similar to those in the `Message` class, allowing you to create, read, update, and destroy alerts in the database.

Please note that the `User` class does not store user types in the database, so when reading from the database, the user type is set to `None`. You may want to adjust this based on how you plan to use the `User` class in your application.

If you need further assistance or have any questions, feel free to ask. Happy coding! 😊👨‍💻

Chris 5

Thank you for the prompt and detailed responses! What do you think the attributes of a user for this kind of application should be?
Sent by you: Thank you for the prompt and detailed responses! What do you think the attributes of a user for this kind of application should be?

Copilot 5

For an application that involves real-time evaluation of AI response quality and alerting users, the attributes of a user should be carefully considered to ensure effective communication and management. Here are some attributes that could be important:

- **UserID**: A unique identifier for each user.
- **UserType**: To distinguish between different roles, such as human agents (type 1) and customers (type 2).
- **Name**: The full name of the user for personalization and identification.
- **Email**: For sending email alerts and communications.
- **PhoneNumber**: For sending SMS alerts if necessary.
- **PreferredContactMethod**: Some users may prefer email over SMS, or vice versa.
- **IsActive**: A boolean indicating whether the user is currently active or inactive.
- **LastActiveTime**: Timestamp of the user's last activity, useful for tracking availability.

- **LanguagePreference**: For multilingual applications, knowing the user’s preferred language can be crucial.
- **TimeZone**: To send alerts at appropriate local times.
- **AlertPreferences**: Custom settings for when and how the user wishes to receive alerts.

These attributes can help tailor the user experience and ensure that alerts are sent to the right person, through the right channel, and at the right time. Additionally, they can assist in maintaining user profiles and managing user-related data efficiently within the system.

If you need further assistance or have any specific questions about user attributes, feel free to ask! 😊👨‍💻

Chris 6

Sounds great! Thanks for the suggestions!!! Please update the user class to include the attributes you have provided :)
Sent by you: Sounds great! Thanks for the suggestions!!! Please update the user class to include the attributes you have provided :)

Copilot 6

You’re welcome! I’m glad you found the suggestions helpful. Here’s the updated `User` class with the additional attributes:

```
import datetime

class User:
    def __init__(self, user_id, user_type, name, email, phone_number, preferred_contact_method, is_active, language_preference, time_zone, alert_preferences):
        self.user_id = user_id
        self.user_type = user_type  # 1 for human agent, 2 for customer
        self.name = name
        self.email = email
        self.phone_number = phone_number
        self.preferred_contact_method = preferred_contact_method  # 'email' or 'sms'
        self.is_active = is_active
        self.last_active_time = datetime.datetime.now()  # Automatically set to current time
        self.language_preference = language_preference
        self.time_zone = time_zone
        self.alert_preferences = alert_preferences  # Custom settings for alerts

    # Add any additional methods you need for the User class below
```

This `User` class now includes a comprehensive set of attributes that can be used to manage user information and preferences within your application. The `last_active_time` is automatically set to the current time when a `User` instance is created, but you can update it as needed based on user activity.

Remember to adjust the database schema and CRUD operations accordingly to accommodate these new attributes. If you need further assistance or have any more questions, feel free to ask. Happy coding! 😊👨‍💻

Chris 7

Great constructor! After all we had talked about today, I was hoping the user class would come with logic to persist it to a db
Sent by you: Great constructor! After all we had talked about today, I was hoping the user class would come with logic to persist it to a db

Copilot 7

Certainly! Let's add the logic to persist the `User` class to a SQLite database. Here's how you can update the `User` class with CRUD operations:

```
import sqlite3
import datetime

DATABASE = 'users.db'

class User:
    def __init__(self, user_id, user_type, name, email, phone_number,
preferred_contact_method, is_active, language_preference, time_zone, alert_preferences):
        self.user_id = user_id
        self.user_type = user_type
        self.name = name
        self.email = email
        self.phone_number = phone_number
        self.preferred_contact_method = preferred_contact_method
        self.is_active = is_active
        self.last_active_time = datetime.datetime.now()
        self.language_preference = language_preference
        self.time_zone = time_zone
        self.alert_preferences = alert_preferences

    @staticmethod
    def db_connection():
        conn = sqlite3.connect(DATABASE)
        return conn

    def create(self):
        conn = self.db_connection()
        cursor = conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                user_id TEXT PRIMARY KEY,
                user_type INTEGER,
                name TEXT,
                email TEXT,
                phone_number TEXT,
                preferred_contact_method TEXT,
                is_active BOOLEAN,
                last_active_time TEXT,
                language_preference TEXT,
                time_zone TEXT,
                alert_preferences TEXT
            )
        ''')
        cursor.execute('''
            INSERT INTO users (user_id, user_type, name, email, phone_number,
preferred_contact_method, is_active, last_active_time, language_preference, time_zone,
alert_preferences)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        ''', (self.user_id, self.user_type, self.name, self.email, self.phone_number,
self.preferred_contact_method, self.is_active, self.last_active_time, self.language_preference,
self.time_zone, self.alert_preferences))
        conn.commit()
        conn.close()

    @classmethod
    def read(cls, user_id):
        conn = cls.db_connection()
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users WHERE user_id=?', (user_id,))
        user_data = cursor.fetchone()
        conn.close()
        if user_data:
            return cls(*user_data)
        else:
            return None

    def update(self):
        conn = self.db_connection()
        cursor = conn.cursor()
        cursor.execute('''
            UPDATE users
            SET user_type=?, name=?, email=?, phone_number=?, preferred_contact_method=?, is_active=?, last_active_time=?, language_preference=?, time_zone=?, alert_preferences=?
            WHERE user_id=?
        ''', (self.user_type, self.name, self.email, self.phone_number, self.preferred_contact_method, self.is_active, self.last_active_time, self.language_preference, self.time_zone, self.alert_preferences, self.user_id))
        conn.commit()
        conn.close()

    @staticmethod
    def destroy(user_id):
        conn = User.db_connection()
        cursor = conn.cursor()
        cursor.execute('DELETE FROM users WHERE user_id=?', (user_id,))
        conn.commit()
        conn.close()

# Example usage:
# user = User('user123', 1, 'John Doe', 'john.doe@example.com', '555-1234', 'email', True, 'English', 'America/New_York', '{"email": true, "sms": false}')
# user.create()
# fetched_user = User.read('user123')
# fetched_user.name = 'Jane Doe'
# fetched_user.update()
# User.destroy('user123')
```

This updated `User` class includes methods to create a table if it doesn’t exist, insert a new user, retrieve a user by ID, update a user, and delete a user from the SQLite database named `users.db`.

Please ensure that you have the necessary error handling and validation in place for a production environment. If you need further assistance or have any more questions, feel free to ask. Happy coding! 😊👨‍💻

Chris 8

Thank you so much for making this a great day of coding! I think you can help me with this task. Typically, db sessions and connection managment are dealt with a little differently in a flask application. Please refactor the following code to be more in line with how flask traditionally handles db sessions. import sqlite3 import datetime DATABASE = 'users.db' class Alert: def

```
__init__(self, alert_id, alert_text, message_id, users): self.alert_id = alert_id self.alert_text =
alert_text self.message_id = message_id self.users = users # List of User objects associated
with the alert @staticmethod def db_connection(): conn = sqlite3.connect(DATABASE) return
conn def create(self): conn = self.db_connection() cursor = conn.cursor() cursor.execute('''
CREATE TABLE IF NOT EXISTS alerts ( alert_id INTEGER PRIMARY KEY, alert_text TEXT,
message_id INTEGER, users TEXT -- This will be a stringified list of user IDs ) ''') # Convert the
list of User objects to a list of user IDs user_ids = [user.user_id for user in self.users]
cursor.execute(''' INSERT INTO alerts (alert_text, message_id, users) VALUES (?, ?, ?) ''',
(self.alert_text, self.message_id, str(user_ids))) conn.commit() conn.close() @classmethod def
read(cls, alert_id): conn = cls.db_connection() cursor = conn.cursor() cursor.execute('SELECT *
FROM alerts WHERE alert_id=?', (alert_id,)) alert_data = cursor.fetchone() conn.close() if
alert_data: # Convert the stringified list of user IDs back to a list of User objects user_ids =
eval(alert_data[3]) users = [User(user_id, None) for user_id in user_ids] # User type is not
stored in DB return cls(alert_data[0], alert_data[1], alert_data[2], users) else: return None def
update(self): conn = self.db_connection() cursor = conn.cursor() # Convert the list of User
objects to a list of user IDs user_ids = [user.user_id for user in self.users] cursor.execute('''
UPDATE alerts SET alert_text=?, message_id=?, users=? WHERE alert_id=? ''', (self.alert_text,
self.message_id, str(user_ids), self.alert_id)) conn.commit() conn.close() @staticmethod def
destroy(alert_id): conn = Alert.db_connection() cursor = conn.cursor() cursor.execute('DELETE
FROM alerts WHERE alert_id=?', (alert_id,)) conn.commit() conn.close() # Example usage: #
users = [User('user123', 1), User('user456', 2)] # alert = Alert(None, 'Urgent alert text', 78910,
users) # alert.create() # fetched_alert = Alert.read(1) # fetched_alert.alert_text = 'Updated alert
text' # fetched_alert.update() # Alert.destroy(1) class EmailAlert(Alert): def send_alert(self,
message): print(f"Sending email alert: {message}") class SmsAlert(Alert): def send_alert(self,
message): print(f"Sending SMS alert: {message}") class User: def __init__(self, user_id,
user_type, name, email, phone_number, preferred_contact_method, is_active,
language_preference, time_zone, alert_preferences): self.user_id = user_id self.user_type =
user_type self.name = name self.email = email self.phone_number = phone_number
self.preferred_contact_method = preferred_contact_method self.is_active = is_active
self.last_active_time = datetime.datetime.now() self.language_preference =
language_preference self.time_zone = time_zone self.alert_preferences = alert_preferences
@staticmethod def db_connection(): conn = sqlite3.connect(DATABASE) return conn def
create(self): conn = self.db_connection() cursor = conn.cursor() cursor.execute(''' CREATE
TABLE IF NOT EXISTS users ( user_id TEXT PRIMARY KEY, user_type INTEGER, name
TEXT, email TEXT, phone_number TEXT, preferred_contact_method TEXT, is_active
BOOLEAN, last_active_time TEXT, language_preference TEXT, time_zone TEXT,
alert_preferences TEXT ) ''') cursor.execute(''' INSERT INTO users (user_id, user_type, name,
email, phone_number, preferred_contact_method, is_active, last_active_time,
language_preference, time_zone, alert_preferences) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?) ''',
(self.user_id, self.user_type, self.name, self.email, self.phone_number,
self.preferred_contact_method, self.is_active, self.last_active_time, self.language_preference,
self.time_zone, self.alert_preferences)) conn.commit() conn.close() @classmethod def read(cls,
user_id): conn = cls.db_connection() cursor = conn.cursor() cursor.execute('SELECT * FROM
users WHERE user_id=?', (user_id,)) user_data = cursor.fetchone() conn.close() if user_data:
```

return cls(*user_data) else: return None def update(self): conn = self.db_connection() cursor = conn.cursor() cursor.execute(''' UPDATE users SET user_type=?, name=?, email=?, phone_number=?, preferred_contact_method=?, is_active=?, last_active_time=?, language_preference=?, time_zone=?, alert_preferences=? WHERE user_id=? ''', (self.user_type, self.name, self.email, self.phone_number, self.preferred_contact_method, self.is_active, self.last_active_time, self.language_preference, self.time_zone, self.alert_preferences, self.user_id)) conn.commit() conn.close() @staticmethod def destroy(user_id): conn = User.db_connection() cursor = conn.cursor() cursor.execute('DELETE FROM users WHERE user_id=?', (user_id,)) conn.commit() conn.close() # Example usage: # user = User('user123', 1, 'John Doe', 'john.doe@example.com', '555-1234', 'email', True, 'English', 'America/New_York', '{"email": true, "sms": false}') # user.create() # fetched_user = User.read('user123') # fetched_user.name = 'Jane Doe' # fetched_user.update() # User.destroy('user123') class Message: def __init__(self, agent_id, customer_id, message_id, message_text, sentiment_score): self.agent_id = agent_id self.customer_id = customer_id self.message_id = message_id self.message_text = message_text self.sentiment_score = sentiment_score @staticmethod def db_connection(): conn = sqlite3.connect(DATABASE) return conn def create(self): conn = self.db_connection() cursor = conn.cursor() cursor.execute(''' CREATE TABLE IF NOT EXISTS messages (message_id INTEGER PRIMARY KEY, agent_id TEXT, customer_id REAL, message_text TEXT, sentiment_score TEXT) ''') cursor.execute(''' INSERT INTO messages (agent_id, customer_id, message_id, message_text, sentiment_score) VALUES (?, ?, ?, ?, ?) ''', (self.agent_id, self.customer_id, self.message_id, self.message_text, self.sentiment_score)) conn.commit() conn.close() @classmethod def read(cls, message_id): conn = cls.db_connection() cursor = conn.cursor() cursor.execute('SELECT * FROM messages WHERE message_id=?', (message_id,)) message_data = cursor.fetchone() conn.close() if message_data:

Sent by you: Thank you so much for making this a great day of coding! I think you can help me with this task. Typically, db sessions and connection managment are dealt with a little differently in a flask application. Please refactor the following code to be more in line with how flask traditionally handles db sessions. import sqlite3 import datetime DATABASE = 'users.db' class Alert: def __init__(self, alert_id, alert_text, message_id, users): self.alert_id = alert_id self.alert_text = alert_text self.message_id = message_id self.users = users # List of User objects associated with the alert @staticmethod def db_connection(): conn = sqlite3.connect(DATABASE) return conn def create(self): conn = self.db_connection() cursor = conn.cursor() cursor.execute(''' CREATE TABLE IF NOT EXISTS alerts (alert_id INTEGER PRIMARY KEY, alert_text TEXT, message_id INTEGER, users TEXT -- This will be a stringified list of user IDs) ''') # Convert the list of User objects to a list of user IDs user_ids = [user.user_id for user in self.users] cursor.execute(''' INSERT INTO alerts (alert_text, message_id, users) VALUES (?, ?, ?) ''', (self.alert_text, self.message_id, str(user_ids))) conn.commit() conn.close() @classmethod def read(cls, alert_id): conn = cls.db_connection() cursor = conn.cursor() cursor.execute('SELECT * FROM alerts WHERE alert_id=?', (alert_id,)) alert_data = cursor.fetchone() conn.close() if alert_data: # Convert the stringified list of user IDs back to a list of User objects user_ids = eval(alert_data[3]) users = [User(user_id, None) for user_id in user_ids] # User type is not stored in DB return cls(alert_data[0], alert_data[1], alert_data[2], users) else: return None def update(self): conn = self.db_connection() cursor = conn.cursor() #

Convert the list of User objects to a list of user IDs user_ids = [user.user_id for user in self.users] cursor.execute(''' UPDATE alerts SET alert_text=?, message_id=?, users=? WHERE alert_id=? ''', (self.alert_text, self.message_id, str(user_ids), self.alert_id)) conn.commit() conn.close() @staticmethod def destroy(alert_id): conn = Alert.db_connection() cursor = conn.cursor() cursor.execute('DELETE FROM alerts WHERE alert_id=?', (alert_id,)) conn.commit() conn.close() # Example usage: # users = [User('user123', 1), User('user456', 2)] # alert = Alert(None, 'Urgent alert text', 78910, users) # alert.create() # fetched_alert = Alert.read(1) # fetched_alert.alert_text = 'Updated alert text' # fetched_alert.update() # Alert.destroy(1) class EmailAlert(Alert): def send_alert(self, message): print(f"Sending email alert: {message}") class SmsAlert(Alert): def send_alert(self, message): print(f"Sending SMS alert: {message}") class User: def __init__(self, user_id, user_type, name, email, phone_number, preferred_contact_method, is_active, language_preference, time_zone, alert_preferences): self.user_id = user_id self.user_type = user_type self.name = name self.email = email self.phone_number = phone_number self.preferred_contact_method = preferred_contact_method self.is_active = is_active self.last_active_time = datetime.datetime.now() self.language_preference = language_preference self.time_zone = time_zone self.alert_preferences = alert_preferences @staticmethod def db_connection(): conn = sqlite3.connect(DATABASE) return conn def create(self): conn = self.db_connection() cursor = conn.cursor() cursor.execute(''' CREATE TABLE IF NOT EXISTS users (user_id TEXT PRIMARY KEY, user_type INTEGER, name TEXT, email TEXT, phone_number TEXT, preferred_contact_method TEXT, is_active BOOLEAN, last_active_time TEXT, language_preference TEXT, time_zone TEXT, alert_preferences TEXT) ''') cursor.execute(''' INSERT INTO users (user_id, user_type, name, email, phone_number, preferred_contact_method, is_active, last_active_time, language_preference, time_zone, alert_preferences) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?) ''', (self.user_id, self.user_type, self.name, self.email, self.phone_number, self.preferred_contact_method, self.is_active, self.last_active_time, self.language_preference, self.time_zone, self.alert_preferences)) conn.commit() conn.close() @classmethod def read(cls, user_id): conn = cls.db_connection() cursor = conn.cursor() cursor.execute('SELECT * FROM users WHERE user_id=?', (user_id,)) user_data = cursor.fetchone() conn.close() if user_data: return cls(*user_data) else: return None def update(self): conn = self.db_connection() cursor = conn.cursor() cursor.execute(''' UPDATE users SET user_type=?, name=?, email=?, phone_number=?, preferred_contact_method=?, is_active=?, last_active_time=?, language_preference=?, time_zone=?, alert_preferences=? WHERE user_id=? ''', (self.user_type, self.name, self.email, self.phone_number, self.preferred_contact_method, self.is_active, self.last_active_time, self.language_preference, self.time_zone, self.alert_preferences, self.user_id)) conn.commit() conn.close() @staticmethod def destroy(user_id): conn = User.db_connection() cursor = conn.cursor() cursor.execute('DELETE FROM users WHERE user_id=?', (user_id,)) conn.commit() conn.close() # Example usage: # user = User('user123', 1, 'John Doe', 'john.doe@example.com', '555-1234', 'email', True, 'English', 'America/New_York', '{"email": true, "sms": false}') # user.create() # fetched_user = User.read('user123') # fetched_user.name = 'Jane Doe' # fetched_user.update() # User.destroy('user123') class Message: def __init__(self, agent_id, customer_id, message_id, message_text, sentiment_score): self.agent_id = agent_id self.customer_id = customer_id self.message_id = message_id self.message_text =

```
message_text self.sentiment_score = sentiment_score @staticmethod def db_connection():
conn = sqlite3.connect(DATABASE) return conn def create(self): conn = self.db_connection()
cursor = conn.cursor() cursor.execute(''' CREATE TABLE IF NOT EXISTS messages (
message_id INTEGER PRIMARY KEY, agent_id TEXT, customer_id REAL, message_text
TEXT, sentiment_score TEXT ) ''') cursor.execute(''' INSERT INTO messages (agent_id,
customer_id, message_id, message_text, sentiment_score) VALUES (?, ?, ?, ?, ?) ''',
(self.agent_id, self.customer_id, self.message_id, self.message_text, self.sentiment_score))
conn.commit() conn.close() @classmethod def read(cls, message_id): conn =
cls.db_connection() cursor = conn.cursor() cursor.execute('SELECT * FROM messages
WHERE message_id=?', (message_id,)) message_data = cursor.fetchone() conn.close() if
message_data:
```

Copilot 8

I’m delighted to help you refactor the code to align with Flask’s traditional handling of database sessions. Flask typically uses an application context to manage the database connection, which allows for cleaner code and better error handling. Here’s how you can refactor your classes to use Flask’s database session management:

First, you’ll need to set up Flask-SQLAlchemy, which is an extension that adds support for SQLAlchemy to your Flask application.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
# Configure the SQLAlchemy connection string
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
# Initialize the database connection
db = SQLAlchemy(app)

class Alert(db.Model):
    alert_id = db.Column(db.Integer, primary_key=True)
    alert_text = db.Column(db.Text, nullable=False)
    message_id = db.Column(db.Integer, nullable=False)
    users = db.Column(db.Text)  # This will be a stringified list of user IDs

    def __init__(self, alert_text, message_id, users):
        self.alert_text = alert_text
        self.message_id = message_id
        self.users = str([user.user_id for user in users])

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()

    @classmethod
    def find_by_id(cls, alert_id):
        return cls.query.filter_by(alert_id=alert_id).first()

    def delete_from_db(self):
        db.session.delete(self)
        db.session.commit()

# Similar refactoring would be done for the User and Message classes

# Example usage:
# users = [User('user123', 1), User('user456', 2)]
# alert = Alert('Urgent alert text', 78910, users)
# alert.save_to_db()
# fetched_alert = Alert.find_by_id(1)
# fetched_alert.alert_text = 'Updated alert text'
# fetched_alert.save_to_db()
# fetched_alert.delete_from_db()
```

In this refactored code:

- We use Flask-SQLAlchemy to define our models and handle database sessions.
- Each class now inherits from `db.Model`, which is the base class for all models in Flask-SQLAlchemy.
- We define the columns within each class, mapping them to the database fields.
- We replace the manual SQL statements with methods like `save_to_db` and `delete_from_db` that use SQLAlchemy's ORM capabilities.
- The `find_by_id` class method is used to retrieve an object by its ID using SQLAlchemy's querying capabilities.

Make sure to install Flask-SQLAlchemy with `pip install Flask-SQLAlchemy` and adjust your Flask application configuration accordingly. This approach will handle database connections and sessions more effectively, especially when scaling your application.

If you have any further questions or need additional assistance, feel free to ask. Happy coding! 😊👨‍💻

Chris 9

please refactor classes.py to use the sqlalchemy db configured in app.py like app = Flask(__name__) api = Api(app) app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///prototype.db' db = SQLAlchemy(app) # Add the Message resource to the API

api.add_resource(MessageResource, '/message') if __name__ == '__main__': print("Available endpoints:") for rule in app.url_map.iter_rules(): print(f"{rule.endpoint}: {rule.rule}") app.run(debug=True)

Sent by you: please refactor classes.py to use the sqlalchemy db configured in app.py like app = Flask(__name__) api = Api(app) app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///prototype.db' db = SQLAlchemy(app) # Add the Message resource to the API api.add_resource(MessageResource, '/message') if __name__ == '__main__': print("Available endpoints:") for rule in app.url_map.iter_rules(): print(f"{rule.endpoint}: {rule.rule}") app.run(debug=True)

attached a file called classes.py

Copilot 9

Certainly! I'll refactor the `classes.py` file to use Flask-SQLAlchemy for database interactions. Here's the updated code:

```
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime

db = SQLAlchemy()

class Alert(db.Model):
    __tablename__ = 'alerts'
    alert_id = db.Column(db.Integer, primary_key=True)
    alert_text = db.Column(db.Text, nullable=False)
    message_id = db.Column(db.Integer, nullable=False)
    users = db.Column(db.Text)  # This will be a stringified list of user IDs

    def __init__(self, alert_text, message_id, users):
        self.alert_text = alert_text
        self.message_id = message_id
        self.users = str([user.user_id for user in users])

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()

    @classmethod
    def find_by_id(cls, alert_id):
        return cls.query.filter_by(alert_id=alert_id).first()

    def delete_from_db(self):
        db.session.delete(self)
        db.session.commit()
```

```
class EmailAlert(Alert):
    def send_alert(self, message):
        print(f"Sending email alert: {message}")

class SmsAlert(Alert):
    def send_alert(self, message):
        print(f"Sending SMS alert: {message}")

class User(db.Model):
    __tablename__ = 'users'
    user_id = db.Column(db.String(80), primary_key=True)
    user_type = db.Column(db.Integer, nullable=False)
    name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(120), nullable=False)
    phone_number = db.Column(db.String(20), nullable=True)
    preferred_contact_method = db.Column(db.String(10), nullable=False)
    is_active = db.Column(db.Boolean, default=True)
    last_active_time = db.Column(db.DateTime, default=datetime.utcnow)
    language_preference = db.Column(db.String(50), nullable=False)
    time_zone = db.Column(db.String(50), nullable=False)
    alert_preferences = db.Column(db.Text, nullable=True)

    def __init__(self, user_id, user_type, name, email, phone_number,
preferred_contact_method, is_active, language_preference, time_zone, alert_preferences):
        self.user_id = user_id
        self.user_type = user_type
        self.name = name
        self.email = email
        self.phone_number = phone_number
        self.preferred_contact_method = preferred_contact_method
        self.is_active = is_active
        self.language_preference = language_preference
        self.time_zone = time_zone
        self.alert_preferences = alert_preferences

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()

    @classmethod
    def find_by_id(cls, user_id):
        return cls.query.filter_by(user_id=user_id).first()
    def delete_from_db(self):
        db.session.delete(self)
        db.session.commit()

class Message(db.Model):
    __tablename__ = 'messages'
    message_id = db.Column(db.Integer, primary_key=True)
    agent_id = db.Column(db.String(80), nullable=False)
    customer_id = db.Column(db.Float, nullable=False)
    message_text = db.Column(db.Text, nullable=False)
    sentiment_score = db.Column(db.String(50), nullable=False)

    def __init__(self, agent_id, customer_id, message_id, message_text, sentiment_score):
        self.agent_id = agent_id
        self.customer_id = customer_id
        self.message_id = message_id
        self.message_text = message_text
        self.sentiment_score = sentiment_score

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()

    @classmethod
    def find_by_id(cls, message_id):
        return cls.query.filter_by(message_id=message_id).first()

    def delete_from_db(self):
        db.session.delete(self)
        db.session.commit()

    def alert_users(self, alert_type):
        if alert_type == 'email':
            alert = EmailAlert()
        elif alert_type == 'sms':
            alert = SmsAlert()
        else:
            raise ValueError("Invalid alert type")
        alert.send_alert(self.message_text)
```

In this refactored code:

- Each class now inherits from `db.Model`, which is the base class for all models in Flask-SQLAlchemy.
- The `__tablename__` attribute is used to define the table name for each model.
- The `db.Column` is used to define the columns for each table.
- The `save_to_db` method is used to add the object to the session and commit it to the database.
- The `find_by_id` class method is used to find a record by its ID.
- The `delete_from_db` method is used to delete the object from the session and commit the change to the database.

Make sure to initialize the `db` object with your Flask app before running the application:

```
from yourapplication import db, create_app

app = create_app()
db.init_app(app)

with app.app_context():
    db.create_all()
```

Replace `yourapplication` with the actual name of your application module. The `create_app` function should be the factory function that initializes your Flask app.

If you have any further questions or need additional assistance, feel free to ask. Happy coding! 😊👨‍💻

Chris 10

Hey, I tracked down a bit of an error. I'm trying to persist messages in a db using the code, but I receive an error that the table does not exist. I'm having trouble figuring out the create table or insert method with sqlalchemy. Please update all the create methods in classes.py so that they create the corresponding table if it does not exist and then performs the insert.
Sent by you: Hey, I tracked down a bit of an error. I'm trying to persist messages in a db using the code, but I receive an error that the table does not exist. I'm having trouble figuring out the create table or insert method with sqlalchemy. Please update all the create methods in classes.py so that they create the corresponding table if it does not exist and then performs the insert.

Copilot 10

Certainly! To ensure that the tables are created if they do not exist before performing an insert operation, you can use the `db.create_all()` method provided by Flask-SQLAlchemy. This method creates all tables that have not yet been created, based on the models defined in your application.

Here's how you can update the `create` methods in your `classes.py` to include a check for table creation:

```
from exts import db
from datetime import datetime

class Alert(db.Model):
    # ... [rest of the Alert class definition]

    def create(self):
        db.create_all()  # Ensure that the table is created
        db.session.add(self)
        db.session.commit()

# ... [rest of the EmailAlert and SmsAlert class definitions]

class User(db.Model):
    # ... [rest of the User class definition]

    def create(self):
        db.create_all()  # Ensure that the table is created
        db.session.add(self)
        db.session.commit()

class Message(db.Model):
    # ... [rest of the Message class definition]

    def create(self):
        db.create_all()  # Ensure that the table is created
        db.session.add(self)
        db.session.commit()

# ... [rest of the Message class methods]
```

By adding `db.create_all()` at the beginning of each `create` method, you ensure that the corresponding table for the model is created in the database before trying to insert a new record.

Please note that calling `db.create_all()` is typically done at the application startup rather than within each `create` method. It's more efficient to ensure all tables are created when the application starts, rather than checking every time you create a new record. However, if you prefer to have this check within the `create` methods for any reason, the above code will work.

Remember to import `db` from your Flask application context where it's initialized with `SQLAlchemy(app)`.

If you have any further questions or need additional assistance, feel free to ask. Happy coding! 😊👨‍💻

Chris 11

I'm not sure thats the best solution. db.create_all is already called in app.py. Please try again
Sent by you: I'm not sure thats the best solution. db.create_all is already called in app.py. Please try again

Copilot 11

Understood. If `db.create_all()` is already being called in `app.py` and the tables are still not being created, it's possible that the models are not being imported or recognized by Flask before `db.create_all()` is executed. This can happen if the models are defined in a separate module that isn't imported before the application starts.

To ensure that the tables are created, you need to import the models in your `app.py` before calling `db.create_all()`. Here's how you can adjust your `app.py`:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restful import Api

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///prototype.db'
db = SQLAlchemy(app)

# Import the models here
from classes import Alert, User, Message
```

```
api = Api(app)
# Add the Message resource to the API
api.add_resource(MessageResource, '/message')

@app.before_first_request
def create_tables():
    db.create_all()

if __name__ == '__main__':
    app.run(debug=True)
```

By importing the models before the application starts, you ensure that `db.create_all()` knows about them and creates the corresponding tables.

Additionally, I've added a `before_first_request` decorator to call `db.create_all()`. This ensures that the tables are created when the Flask application handles its first request, which is another common pattern for initializing databases in Flask applications.

If the tables are still not being created, make sure that the `SQLALCHEMY_DATABASE_URI` is set correctly and that the database file has the proper permissions to be modified by your application.

If you need further assistance or have any more questions, feel free to ask. Happy coding! 😊👨‍💻

Chris 12

the app is now integrated with a db with major thanks to you! Now lets drive this prototype home! Please expand the email alert class to actually send the email to the associated type 2 user and persisit the sent alert to the db.

Sent by you: the app is now integrated with a db with major thanks to you! Now lets drive this prototype home! Please expand the email alert class to actually send the email to the associated type 2 user and persisit the sent alert to the db.

Copilot 12

I'm glad to hear the database integration is successful! To expand the `EmailAlert` class to send an email to the associated type 2 user and persist the sent alert to the database, you'll need to use an email sending library like `smtplib` for Python, and update the `EmailAlert` class accordingly.

Here's an example of how you might modify the `EmailAlert` class:

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from classes import db, User

class EmailAlert(db.Model):
    __tablename__ = 'email_alerts'
    id = db.Column(db.Integer, primary_key=True)
    alert_text = db.Column(db.Text, nullable=False)
    message_id = db.Column(db.Integer, nullable=False)
    user_id = db.Column(db.String(80), db.ForeignKey('users.user_id'))
    sent = db.Column(db.Boolean, default=False)

    def __init__(self, alert_text, message_id, user_id):
        self.alert_text = alert_text
        self.message_id = message_id
        self.user_id = user_id

    def send_email(self):
        user = User.query.filter_by(user_id=self.user_id, user_type=2).first()
        if not user:
            raise ValueError("User not found or not a type 2 user.")

        # Set up the SMTP server
        smtp_server = 'smtp.example.com'
        smtp_port = 587
        smtp_username = 'your_username'
        smtp_password = 'your_password'
        sender_email = 'your_email@example.com'
        receiver_email = user.email

        # Create the email message
        message = MIMEMultipart()
        message['From'] = sender_email
        message['To'] = receiver_email
        message['Subject'] = 'Alert Notification'
        body = self.alert_text
        message.attach(MIMEText(body, 'plain'))

        # Send the email
        server = smtplib.SMTP(smtp_server, smtp_port)
        server.starttls()
        server.login(smtp_username, smtp_password)
        server.sendmail(sender_email, receiver_email, message.as_string())
        server.quit()
        # Update the alert sent status
        self.sent = True
        db.session.commit()

    def create(self):
        db.session.add(self)
        db.session.commit()
        self.send_email()  # Send the email after creating the alert record
```

In this updated `EmailAlert` class:

- The `send_email` method is responsible for sending the email using `smtplib`.
- The `create` method now calls `self.send_email()` after adding the alert to the session and committing it to ensure the email is sent when the alert is created.
- The `EmailAlert` class now inherits from `db.Model` and has its own table to keep track of sent alerts.
- The `user_id` field is a foreign key that references the `user_id` in the `User` table.

Please replace `smtp.example.com`, `your_username`, `your_password`, and `your_email@example.com` with your actual SMTP server details and credentials.

Important Note: Sending emails via SMTP in a production environment should be handled securely and responsibly, ensuring that sensitive information like usernames and passwords are not hard-coded and are stored securely using environment variables or a secrets manager. Additionally, you should handle exceptions that may occur during the email sending process to ensure the application's stability.

If you need further assistance or have any more questions, feel free to ask. Happy coding! 😊👨‍💻

Chris 13

one last thing.... I forgot to add a readme. Please create a readme for the project that encapsulates everything we have worked on today. In the first paragraph, please emphasize how almost the whole prototype was done via copy paste from github copilot in about three hours. Please include detailed local set up instructions. Please document the exposed get and post endpoints for the message resource in a way that someone can copy-paste curl commands to try it out.

Sent by you: one last thing.... I forgot to add a readme. Please create a readme for the project that encapsulates everything we have worked on today. In the first paragraph, please emphasize how almost the whole prototype was done via copy paste from github copilot in about three hours. Please include detailed local set up instructions. Please document the exposed get and post endpoints for the message resource in a way that someone can copy-paste curl commands to try it out.

Copilot 13

Certainly! Here’s a README that encapsulates the work done on your project:

# Real-Time Evaluation System Prototype

## Overview
This prototype for a Real-Time Evaluation System was developed with an innovative approach, leveraging GitHub Copilot's capabilities. In an impressive timeframe of approximately three hours, the majority of the codebase was constructed through a process akin to copy-pasting from GitHub Copilot's suggestions. This method not only expedited development but also ensured that the prototype was built with best practices in mind.

## Local Setup Instructions
To set up this project locally, follow these steps:

1. **Clone the Repository:**

~~~
```sh
git clone https://github.com/your-username/your-repo-name.git
cd your-repo-name
~~~

Create a Virtual Environment (Optional but recommended):

```
python -m venv venv
source venv/bin/activate  # On Windows use `venv\Scripts\activate`
```

2.

Install Dependencies:

```
pip install -r requirements.txt
```

3.

Environment Variables: Set up the necessary environment variables or add them to a `.env` file:

```
export FLASK_APP=app.py
export FLASK_ENV=development
# Add other necessary environment variables here
```

4.
5. **Initialize the Database:** Ensure that `db.create_all()` is called within the application context in `app.py`.

Run the Application:

```
flask run
```

6.

API Endpoints

The system exposes two main endpoints for the `Message` resource:

- **POST /message**: Create a new message.
- **GET /message/<message_id>**: Retrieve an existing message by ID.

Using the POST Endpoint

To create a new message, use the following `curl` command:

```
curl -X POST http://localhost:5000/message \
    -H "Content-Type: application/json" \
    -d '{
          "agentId": "agent123",
          "customerId": 456789,
          "messageId": 78910,
          "messageText": "Your issue has been resolved.",
          "sentimentScore": "positive"
        }'
```

Using the GET Endpoint

To retrieve an existing message, use the following `curl` command:

```
curl -X GET http://localhost:5000/message/78910
```

Replace `78910` with the actual `messageId` you wish to retrieve.

Conclusion

This prototype serves as a testament to the efficiency and effectiveness of AI-assisted development. By utilizing GitHub Copilot, we've demonstrated that it's possible to rapidly prototype a complex system with minimal manual coding.

Please replace `https://github.com/your-username/your-repo-name.git` with the actual URL of your repository and ensure that all environment variables and setup instructions are accurate for your project.

This README provides a comprehensive overview of the project, detailed setup instructions, and documentation of the API endpoints with `curl` commands for easy testing. If you need any further assistance or modifications, feel free to ask. Happy coding! 😊📝

Chris 14

Thanks for a great day!
Sent by you: Thanks for a great day!

Copilot 14

You’re very welcome! I’m delighted to have been a part of your coding journey today. If you ever need more assistance or have new ideas you want to explore, don’t hesitate to reach out. Have a fantastic day and happy coding! 😊👨‍💻