Design Documentation: WhatsApp Integration Project

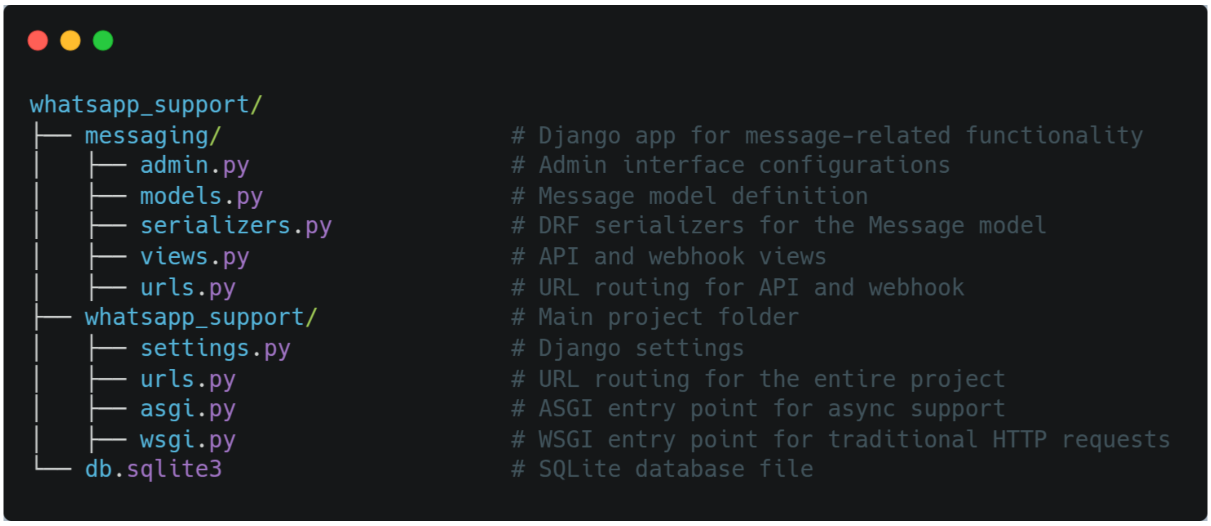
Project Overview

The project implements a basic WhatsApp Integration for a customer support system using Django. It allows users to send and receive WhatsApp messages asynchronously through a webhook. It includes a REST API to handle outgoing messages and an admin interface for monitoring and managing the system.

Core Features

1. **Message Model -** The Message model represents a WhatsApp message, containing details such as the sender, receiver, content, timestamp, and status (either "sent" or "failed").
2. **Webhook Endpoint -** A webhook endpoint (/api/webhook/) accepts incoming WhatsApp messages. It saves the received data in the database, allowing the application to process incoming messages.
3. **Send Message API -** A REST API endpoint (/api/send-message/) allows sending messages to a specified receiver. It saves the message in the database and sets its status to "sent."
4. **Admin Interface -** A custom Django admin interface for managing and viewing messages. Admin users can also send messages directly from the admin panel using a button.
5. **Logging and Error Handling -** The application features basic logging and error handling mechanisms to log successful operations and catch failures.
6. **Async Message Processing -** The message sending and webhook handling operations are processed asynchronously using Python's asyncio and Django's asynchronous support.

Project Structure



```
whatsapp_support/
├── messaging/                     # Django app for message-related functionality
│   ├── admin.py                   # Admin interface configurations
│   ├── models.py                  # Message model definition
│   ├── serializers.py             # DRF serializers for the Message model
│   ├── views.py                   # API and webhook views
│   ├── urls.py                    # URL routing for API and webhook
├── whatsapp_support/              # Main project folder
│   ├── settings.py                # Django settings
│   ├── urls.py                    # URL routing for the entire project
│   ├── asgi.py                    # ASGI entry point for async support
│   ├── wsgi.py                    # WSGI entry point for traditional HTTP requests
└── db.sqlite3                     # SQLite database file
```

Design Decisions

1. **Message Model -** The model stores basic message details such as sender, receiver, content, timestamp, and status. The status field can be "sent" or "failed," which allows us to track the delivery status of the message. The timestamp is automatically set when a message is created.
2. **Webhook Endpoint -** The webhook receives incoming messages via a POST request and processes the payload. It creates a new message in the database with the status set to "received." The process is asynchronous to avoid blocking other requests while handling incoming data.
3. **Send Message API -** The API receives a POST request with sender, receiver, and content as data. It saves the message in the database and sets the status to "sent." This API is wrapped in asynchronous code for better performance when dealing with high loads.
4. **Admin Interface -** A custom admin interface was created to allow administrators to view, filter, and search messages. Additionally, a "Send Message" button is provided in the list view, which enables admins to quickly send messages directly from the admin panel.
5. **Async Processing -** Async functionality is implemented using Python's asyncio and Django's async_to_sync method. The goal is to handle incoming messages and sending operations asynchronously for better scalability.

Future Improvements

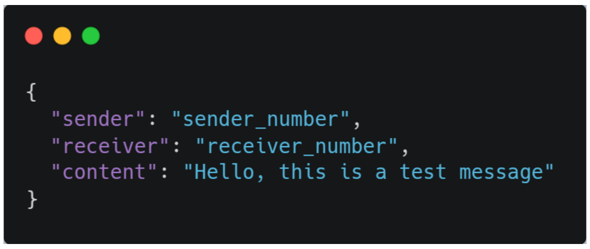
1. **Enhanced Error Handling -** Currently, error handling is basic. The system can be enhanced with custom error messages and retry mechanisms for failed message sends.
2. **Message Queuing -** Instead of sending messages directly in the API view, implementing a message queue (e.g., Celery or Django Q) could improve performance and reliability for message processing, especially in high-traffic environments.
3. **User Authentication -** Implementing user authentication for the API would help secure the message-sending functionality, ensuring that only authorized users can send messages.
4. **Delivery Status Updates -** Implementing a mechanism to track delivery status updates for sent messages (e.g., delivered, read, etc.) could enhance the messaging system's functionality.
5. **Testing -** While basic functionality is covered, comprehensive unit and integration tests can be added to ensure the system’s reliability and handle edge cases.

Assumptions

1. **Message Content -** For the sake of simplicity, the content of the message is treated as a basic string, with no special formatting or multimedia content.
2. **SQLite Database -** The project uses SQLite as the database, which is sufficient for a small-scale application. However, for a production system, a more robust database system (e.g., PostgreSQL) would be recommended.
3. **No External Integrations -** The integration with WhatsApp itself is simulated. In a real-world scenario, an actual WhatsApp API (e.g., Twilio or WhatsApp Business API) would be used for sending and receiving messages.

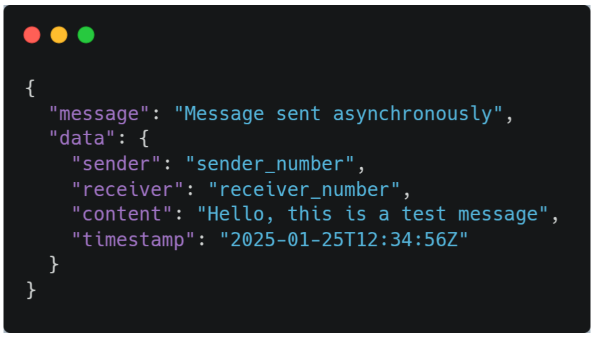
API Endpoints

1. **Send Message**
    a. **URL:** /api/send-message/
    b. **Method:** POST
    c. **Payload:**



```
{
  "sender": "sender_number",
  "receiver": "receiver_number",
  "content": "Hello, this is a test message"
}
```

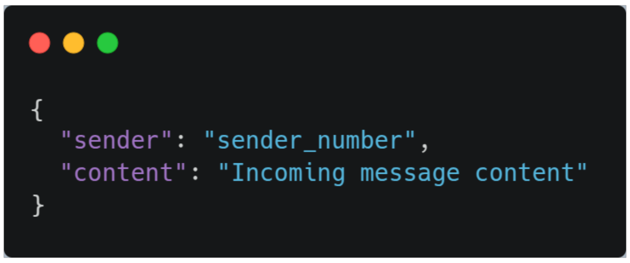
    d. **Response:**



```
{
  "message": "Message sent asynchronously",
  "data": {
    "sender": "sender_number",
    "receiver": "receiver_number",
    "content": "Hello, this is a test message",
    "timestamp": "2025-01-25T12:34:56Z"
  }
}
```

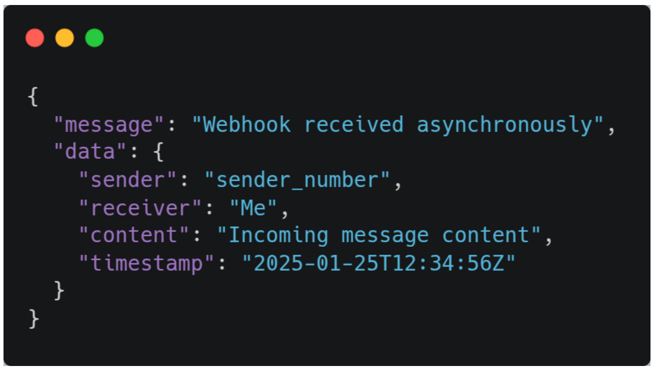
    e. **Status Code:** 201 Created

2. **Webhook**
    a. **URL:** /api/webhook/
    b. **Method:** POST
    c. **Payload:**



```
{
  "sender": "sender_number",
  "content": "Incoming message content"
}
```

    d. **Response:**



```
{
  "message": "Webhook received asynchronously",
  "data": {
    "sender": "sender_number",
    "receiver": "Me",
    "content": "Incoming message content",
    "timestamp": "2025-01-25T12:34:56Z"
  }
}
```

e. **Status Code:** 200 OK

Conclusion

This Django application provides a simple and extensible foundation for integrating WhatsApp messaging into a customer support system. It demonstrates key concepts such as asynchronous message processing, RESTful API design, logging, and the use of Django's admin interface. While basic, the project is designed to be easily extended with additional features, such as message queues, delivery status tracking, and authentication.