E-commerce Sales Chatbot Documentation

1. System Architecture

1.1 Technology Stack

- **Frontend**: React.js – for building the user interface.
- **Backend**: Flask (Python) – for building the REST API and handling backend logic.
- **Database**: SQLite – for storing user data, product information, chat logs, and session data.
- **Authentication**: JWT (JSON Web Token) – for session management, ensuring secure authentication of users.
- **API**: RESTful architecture – for communication between the frontend and backend.

1.2 Component Overview

The system is structured as follows:

E-commerce Chatbot

├── Frontend (React)

│ ├── Chat Interface

│ ├── Home Page

│ └── Login/register

├── Backend (Flask)

│ ├── API Routes

│ ├── Database Models

│ └── Chat Logic

└── Database (SQLite)

├── Users

├── Products

├── Sessions

└── ChatLogs

2. Implementation Details

2.1 Database Schema

- **Products Table**: Stores product details for the e-commerce platform.

```
CREATE TABLE products (
        id INTEGER NOT NULL,
        name VARCHAR(100) NOT NULL,
        price FLOAT NOT NULL,
        description TEXT NOT NULL,
        category VARCHAR(50) NOT NULL,
        PRIMARY KEY (id)
);
```

- **Users Table**: Stores user data such as email and password (hashed).

```
CREATE TABLE users (
        id INTEGER NOT NULL,
        email VARCHAR(100) NOT NULL,
        password_hash TEXT NOT NULL,
        PRIMARY KEY (id),
        UNIQUE (email)
);
```

- **Sessions Table**: Tracks user sessions for authentication.

```
CREATE TABLE sessions (
        session_id VARCHAR NOT NULL,
        user_id VARCHAR,
        created_at DATETIME,
        PRIMARY KEY (session_id)
)
```

- **Chat Logs Table**: Stores the conversation history between the user and chatbot.

```
CREATE TABLE chat_logs (
    id INTEGER NOT NULL,
    session_id VARCHAR NOT NULL,
    message TEXT NOT NULL,
    sender VARCHAR NOT NULL,
```

timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,

PRIMARY KEY (id),

FOREIGN KEY(session_id) REFERENCES sessions (session_id) ON DELETE CASCADE

);

2.2 API Endpoints

- **Authentication Endpoints**:
  - **POST /api/register**: Register a new user.
    - Payload: {email, password}
  - **POST /api/login**: User login.
    - Payload: {email, password}
    - Returns: {session_id, message}
- **Product Endpoints**:
  - **GET /products**: Fetch all products with optional category filter.
    - Query Params: category
  - **GET /products/<product_id>**: Fetch details of a specific product.
- **Chat Endpoints**:
  - **POST /api/chat**: Processes chat messages from the user.
    - Payload: {session_id, message}
    - Returns: {status, bot_response}

2.3 Chatbot Features

- **Product Search and Filtering**: Allows users to search for products by category and price range.
- **Category Browsing**: Users can browse products based on categories.
- **Price Range Queries**: Users can ask for products within specific price ranges.
- **Session Persistence**: User sessions are maintained across multiple interactions with the bot.
- **Chat History Logging**: Each conversation is logged in the database for future reference.

3. Frontend Implementation

3.1 Chat Interface Components

Key components include:

- **Chat Container**: Displays the entire chat interface.
- **Message Display**: Renders individual messages from the bot and the user.
- **Input Form**: Allows the user to input messages.
- **Session Management**: Handles user session (authentication and session ID).
- **Loading States**: Indicates when the chatbot is processing a request.
- **Error Handling**: Displays error messages when something goes wrong.

3.2 State Management

React's useState hook is used to manage the state:

```
const [messages, setMessages] = useState([]); // Chat messages
const [sessionId, setSessionId] = useState(null); // Session ID
const [loading, setLoading] = useState(false); // Loading state
const [error, setError] = useState(null); // Error state
```

4. Backend Implementation

4.1 Flask Application Structure

The Flask app is structured as follows:

```
app/
├── app.py         # Initializes Flask app
├── models.py          # Contains database models (Product, User, etc.)
├── populate_db.py.py          # contains 100 mock e-commerce entries
├── extensions.py       # Contains external extensions like database and CORS
└── instance/
    └── ecommerce.db    # SQLite database file
```

4.2 Database Models

Flask SQLAlchemy is used to define the database models:

- **Product Model**:

```
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    price = db.Column(db.Float)
    description = db.Column(db.Text)
```

category = db.Column(db.String(50))

- **User Model**:

class User(db.Model):

id = db.Column(db.Integer, primary_key=True)

email = db.Column(db.String(120))

password_hash = db.Column(db.String(128))

5. Security Measures

5.1 Authentication

- **Password Hashing**: Passwords are hashed using Werkzeug's secure hashing mechanism.
- **Session Management**: JWT tokens are used for secure session handling.
- **CORS Protection**: CORS is implemented to prevent unauthorized API access.

5.2 Data Protection

- **Input Validation**: Ensures that user inputs are sanitized.
- **SQL Injection Prevention**: Parameterized queries are used to prevent SQL injection attacks.
- **XSS Protection**: The frontend sanitizes user inputs to avoid cross-site scripting attacks.

6. Testing

6.1 API Testing

- **/test-db**: Endpoint for testing the database connection.
- **/check-db-file**: Confirms the existence of the database file.
- **/add-test-product**: Adds a test product to the database.

6.2 Chat Testing

- **Test cases include**:
    - **Product Queries**: Testing if the chatbot can correctly fetch and display products.
    - **Category Browsing**: Ensuring the chatbot can filter products by category.
    - **Price Inquiries**: Verifying the chatbot responds accurately to price range requests.
    - **Error Handling**: Ensuring proper error responses when invalid inputs are provided.

7. Challenges and Solutions

7.1 Session Management

- **Challenge**: Maintaining user context across chat sessions.
- **Solution**: Implemented UUID-based session tracking, with sessions stored in the database.

7.2 Chat Response Processing

- **Challenge**: Handling different types of queries (product search, price range, etc.).
- **Solution**: Implemented a keyword-based routing system to handle various queries and return structured responses.

7.3 Database Performance

- **Challenge**: Ensuring efficient product searching.
- **Solution**: Indexed frequently queried columns (e.g., product name, category) and optimized SQL queries for faster retrieval.

8. Future Improvements

- **NLP Integration**: Integrate Natural Language Processing (NLP) to understand more complex user inputs.
- **Product Recommendations**: Use user interaction data to suggest products.
- **Order Processing**: Add functionality for order creation and payment processing.
- **Multi-language Support**: Expand the chatbot to support multiple languages.
- **Analytics Dashboard**: Create a dashboard for tracking user interactions, sales, and chatbot performance.
- **Chat History Export**: Provide users with the ability to export their chat history.

9. Deployment Guidelines

1. Set up environment variables for API keys and sensitive data.
2. Configure the database connection in the Flask app.
3. Set up CORS policies for frontend-backend communication.
4. Initialize the database with mock data for testing purposes.
5. Test all API endpoints to ensure they function as expected.
6. Monitor error logs and optimize code if necessary.

10. Maintenance

1. Regular database backups to ensure data safety.
2. Log rotation to manage log files and prevent overflow.

3. Continuous performance monitoring for bottlenecks or slow queries.
4. Regular security updates to address potential vulnerabilities.
5. Collect user feedback for continuous improvement and feature additions.