Mini Project 2
PirateEase Chatbot

Brysen Pfingsten

Monday 31st March, 2025

Dr. Shajina Anand, CSAS 2124, Seton Hall University

Contents

1 Introduction

The focus of this project was to explore how object-oriented design patterns can be used to build a chatbot that is both dynamic in its interaction with users and easily extensible for future upgrades. The chatbot is designed to handle 40 predefined queries, along with at least 3 additional flexible queries.

To achieve this, I applied object-oriented principles and design patterns to create a full-stack application. The frontend allows users to interact with the chatbot through the command line, while the backend is responsible for retrieving data from JSON files and generating dynamic responses. This approach gives the chatbot the feel of a real large language model (LLM) utilizing retrieval-augmented generation (RAG).

2 Creational Design Patterns

2.1 Singleton

The `Singleton` pattern is used extensively throughout this project to ensure that only one instance of a given class is created during the program's lifecycle. To streamline this process, I created a base Singleton class that other classes could inherit from. This class overrides the `__new__` method and stores instances in a shared dictionary, allowing for efficient and consistent access to single instances across the application.

```python
class Singleton:
    # Dictionary mapping classes to their instances
    _instances: dict = {}

    def __new__(cls):
        if cls not in cls._instances:
            instance = super().__new__(cls)
            instance._initialized = False
            cls._instances[cls] = instance
        return cls._instances[cls]
```

When applied to backend services, the `Singleton` pattern ensures consistent and synchronized data as the user interacts with PirateEase. For example, the `OrderService` class inherits from the `Singleton` base class and, upon its first and only initialization, loads JSON data from a file and structures it into a list of `Order` objects. This approach improves efficiency by reducing memory usage and load times, while guaranteeing that any updates to the order data are reflected consistently throughout the application. This pattern is applied uniformly across all backend services to maintain coherence and performance.

```python
class OrderService(Singleton):
```

```python
    def __init__(self):
        # If already initialized, skip
        if self._initialized:
            return
        # Load orders from DB
        with open('Databases/orders.json', 'r', encoding='utf-8') as f:
            raw_orders: dict = json.load(f)
            self.__orders: dict[int, Order] = {
                int(order_id): Order(
                    id=int(order_id),
                    customer_name=data["customer_name"],
                    order_date=data["order_date"],
                    eta_hours=data["eta_hours"],
                    item=data["item"],
                    quantity=data["quantity"],
                    refunded=False
                )
                for order_id, data in raw_orders.items()
            }
        # Mark as initialized
        self._initialized = True

    def retrieve_order(self, order_id: str) -> str:
        # Get the order from the DB
        order: Order = self.__orders.get(int(order_id))
        if order is not None:  # If an order matched
            return str(order)  # Return its string representation
        else:  # No order matched the order ID
            return ''  # Return an empty string
```

Additionally, the `Singleton` pattern is applied to the `SessionManager` to ensure that its state and history remain consistent and accessible throughout the entire program. This allows the application to reliably track user interactions—such as appending to the session history or retrieving session state—while handling input in the `UserInterface` class and across all `QueryHandler` components.

```python
# In PirateEase.Utils.session_manager
class SessionManager(Singleton):
    def __init__(self):
        if self._initialized:
            return
        self.__state: dict[str, str] = {}
        self.__history: list[str] = []
        self._initialized = True

```

```python
# In PirateEase.Utils.user_interface
class UserInterface:

    ...

    @classmethod
    def get_refund_reason(cls):
        message: str = 'PirateEase: ' + ResponseFactory.get_response('
            refund_reason')
        slow_print(message)
        cls.session.append_history(message)
        refund_reason: str = input('User: ').strip()
        cls.session.append_history(f'User: {refund_reason}')
        return refund_reason

# In PirateEase.Queries.order_tracking_handler
class OrderTrackingHandler(QueryHandler):
    def handle(self, query: str) -> str:
        if 'order_id' in self._session:
            order_id: str = self._session.get('order_id')
            return self._backend.process_request('order', order_id)

        order_id: str = UserInterface.get_order_id()
        response = self._backend.process_request('order', order_id)
        while not response:
            not_found_response: str = 'PirateEase: ' + ResponseFactory.
                get_response('order_not_found').format(order_id=order_id)
            self._session.append_history(not_found_response)
            slow_print(not_found_response)
            order_id = UserInterface.get_order_id()
            response = self._backend.process_request('order', order_id)

        self._session.set('order_id', order_id)
        return response
```

2.2 Factory

Another creational pattern used in this project is the `Factory` pattern. This pattern encapsulates the object creation logic, allowing for greater flexibility and decoupling in how new instances are constructed. A key example is the `ResponseFactory`, which is responsible for generating dynamic responses based on the specified query category. It also includes predefined response templates with runtime variable substitution, enabling more natural and context-aware replies.

```python
class ResponseFactory:
    with open('Databases/responses.json', 'r', encoding='utf-8') as f:
        responses: dict[str, list[str]] = json.load(f)

    @classmethod
    def get_response(cls, category: str) -> str:
        return random.choice(cls.responses[category])
```

Some examples of categories and responses are below.

```
"product_available": [
    "{item} is in stock at {price}! Only {quantity} remaining.",
    "{item} can be yours today for {price}! We got {quantity} left.",
    "{item} is up for grabs at {price}! {quantity} available now.",
    "{item} is ready to ship for just {price}! Only {quantity} in stock.",
    "Good news! {item} is available for {price} - {quantity} left!",
    "Act fast! {quantity} units of {item} remaining at {price} each.",
    "Yes! {item} is here at {price}, with {quantity} in stock.",
    "Get your {item} now for {price}! Only {quantity} left.",
    "{item} just hit the shelves for {price}! {quantity} up for grabs."
  ],
  "not_available": [
    "{item} is currently out of stock.",
    "Sorry, we're fresh out of {item}.",
    "Looks like {item} is temporarily unavailable.",
    "Ahoy! {item} is all gone for now.",
    "No more {item} at the moment-check back soon!",
    "We have run out of {item}, unfortunately.",
    "{item} is out of commission for the time being.",
    "Heads up! {item} is not available right now."
  ]
```

Additionally, the `Factory` pattern is employed when dynamically retrieving `QueryHandler` instances based on intent types. The `QueryManager` class encapsulates a dictionary that maps intent strings to their corresponding `QueryHandler` classes. It exposes a `get_handler()` interface, which returns the appropriate handler instance for a given intent, streamlining the routing of user queries to the correct logic.

```python
class QueryManager:
    def __init__(self):
        self.handlers: dict[str, QueryHandler] = {
            "order": OrderTrackingHandler(),
            "refund": RefundHandler(),
            "inventory": ProductAvailabilityHandler(),
```

```
            "live_agent": LiveAgentHandler(),
            "exit": ExitHandler(),
            "db": QueryDatabase(),
            "unknown": DefaultHandler()
        }

    def get_handler(self, query_type: str) -> QueryHandler:
        return self.handlers.get(query_type.lower())
```

3 Structural Design Patterns

3.1 Facade

The `Facade` pattern is used in this project to coordinate various subsystems and components through a simplified interface that abstracts away internal complexity. For example, the `ChatBot` class acts as a facade by managing interactions between session history, backend services, intent recognition, and the query manager—all through a single `process_query()` interface.

```
class ChatBot:

    ...

    def process_query(self, query: str) -> str:
        # Add the query to the history
        self.__session_manager.append_history('User: ' + query)
        # Setup up global variables to be used later
        response: str = ''
        db_response: str = self.__query_manager.get_handler('db').handle(
            query)
        # If negative sentiment is detected
        if self.__sentiment_analyzer.negative_sentiment_detected(query):
            negative_sentiment_response: str = ResponseFactory.
                get_response('negative')
            live_agent_connection_response: str = \
            self.__query_manager.get_handler('live_agent').handle(query).
                split('\n', 1)[1]
            response = negative_sentiment_response + '\n' +
                live_agent_connection_response
        # Else if the database returned a response
        elif db_response:
            response = db_response
        # Else determine the intent and route it to a handler.
        else:
```

```
            intent: str = self.__intent_recognizer.recognize_intent(query)
            handler: QueryHandler = self.__query_manager.get_handler(
                intent)
            response = handler.handle(query.lower().strip())
        # Add the response to the history and return it
        self.__session_manager.append_history(f'PirateEase: {response}')
        return response
```

Additionally, the `Facade` pattern is applied in the `BackendManager` to centralize and streamline communication with the underlying backend services. By routing requests and data to the appropriate services, it simplifies access to backend functionality and promotes modularity. This design also makes it easy to extend the system—new services can be integrated by simply registering them with the manager, without altering the rest of the codebase.

```
class BackendManager(Singleton):
    @staticmethod
    def process_request(request_type: str, data: str = ''):
        if request_type == "order":
            return OrderService().retrieve_order(data)
        elif request_type == "refund":
            return RefundService().refund_past_order(data)
        elif request_type == "inventory":
            return InventoryService().check_availability(data)
        elif request_type == "agent":
            return LiveAgentService().get_available_agent()
        elif request_type == "exit":
            return ExitService().get_exit_response()
```

While not exact examples of the `Facade` pattern, both the `UserInterface` and `QueryHandler` classes serve to encapsulate and coordinate functionality from other subsystems such as the `SessionManager` and `ResponseFactory`. This abstraction simplifies the overall logic and promotes separation of concerns, allowing these components to serve as clean entry points for user input and query processing.

3.2 Proxy

Multiple `QueryHandler` instances demonstrate the `Proxy` pattern by acting as intermediaries between user-submitted queries and the backend services. For example, the `OrderTrackingHandler` and `RefundHandler` are responsible for validating that the provided order ID exists in the system before returning the backend response. Additionally, they check the session state to determine whether an order ID or refund ID has already been supplied in a previous interaction. Once a valid identifier is confirmed, the handler forwards the request to the appropriate backend service via the `BackendManager` and returns the resulting response. This ensures that backend services receive only valid, preprocessed data, while preserving a consistent interface for handling queries.

```python
class RefundHandler(QueryHandler):
    def handle(self, query: str) -> str:
        refund_id: str = ''
        if 'refund_id' in self._session:
            refund_id = self._session.get('refund_id')
            return self._backend.process_request('refund_id', refund_id)
        else:
            refund_id = UserInterface.get_order_id()
            response = self._backend.process_request('refund', refund_id)
            while not response:
                not_found_response: str = 'PirateEase: ' + ResponseFactory
                    .get_response('order_not_found').format(order_id=
                    refund_id)
                self._session.append_history(not_found_response)
                slow_print(not_found_response)
                refund_id = UserInterface.get_order_id()
                response = self._backend.process_request('refund',
                    refund_id)
            self._session.set('refund_id', refund_id)
            UserInterface.get_refund_reason()
            return response
```

4 Behavioral Design Patterns

4.1 Observer

The `Observer` pattern is used in the live agent support feature of the application to handle dynamic agent availability and notification. In this implementation, the `LiveAgentNotifier` acts as the subject, while individual `Agent` instances serve as observers. Agents can subscribe to or unsubscribe from the notifier based on their availability, which may change as they clock in/out or when they are connected/disconnected with a customer.

When a user requests live assistance, the notifier broadcasts the chat history to all subscribed agents via the `notify_agents()` method. This decouples the notification logic from the main application flow and promotes a scalable, extensible architecture—new agent types or alerting behaviors can be added without altering the core logic. The pattern also ensures that only available agents receive alerts, maintaining a consistent and efficient system state.

```python
class Agent:
    def __init__(self, name: str, available: bool):
        self.name: str = name
```

```python
        self.available: bool = available

    def alert(self, history: list[str]):
        return f'{self.name} was alerted!\nHistory: {history}'


class LiveAgentNotifier:
    # All the agents subscribed to this notifier
    observers: list[Agent] = []

    @staticmethod
    def add_observer(agent: Agent) -> None:
        LiveAgentNotifier.observers.append(agent)

    @staticmethod
    def remove_observer(agent: Agent) -> None:
        LiveAgentNotifier.observers.remove(agent)

    @staticmethod
    def notify_agents(history: list[str]) -> None:
        # For each subscribed agent
        for agent in LiveAgentNotifier.observers:
            agent.alert(history)  # Alert them w/ history


class LiveAgentService(Singleton):

    ...

    def get_available_agent(self) -> str:
        agent: Agent = random.choice([a for a in self.__agents if a.
            available])  # Choose a random agent
        agent.available = False  # Mark them as unavailable
        LiveAgentNotifier.remove_observer(agent) # Stop observing
        return ResponseFactory.get_response('connecting_agent').format(
            agent=agent.name)  # Return dynamic response
```

4.2 Chain of Responsibility

The application demonstrates the Chain of Responsibility pattern through the flow of user queries as they pass through multiple layers of logic and services. The architecture models this pattern by distributing responsibility across loosely coupled components, each with a distinct role.

For example, when a user submits a query:

1. The input is check for negative sentiment by the `SentimentAnalyzer`. If a negative sentiment is detected, a live agent is requested and hanlded through the `LiveAgentService`.
2. If that did not take place, the query is matched against a database of predefined user queries and responses through the `QueryDatabase`.
3. Finally, if none of the previous two conditions were met, the query is passed through the `IntentRecognizer`.
4. Based on the identified intent, the appropriate `QueryHandler` is selected by the `QueryManager`.
5. The handler may gather additional user input via the `UserInterface`, such as an order ID, item name, or refund reason.
6. Once sufficient data is collected and validated, it is sent to the `BackendManager`.
7. The `BackendManager` routes the request to the correct backend service.
8. The final result is returned up the chain and displayed to the user.

This design allows each component to focus on its own responsibility—intent resolution, input handling, validation, service routing, and response formatting—without being tightly coupled to the others. It also makes the system easier to extend, as new types of queries or services can be added without disrupting the existing flow.

5 OOP Principles

This project leverages fundamental Object-Oriented Programming (OOP) principles to create a robust, maintainable, and extensible application. Each component in the PirateEase chatbot is structured to promote modularity and scalability.

5.1 Encapsulation

Encapsulation involves bundling related data and behavior into classes while hiding internal implementation details. In PirateEase:

- `LiveAgentService` encapsulates the logic for agent availability, exposing only essential methods such as `get_available_agent()`.
- Each `QueryHandler` encapsulates the validation and processing logic unique to its specific query type.

5.2 Abstraction

Abstraction simplifies complex interactions by exposing high-level interfaces while concealing intricate details. For instance:

- The abstract class `QueryHandler` defines a unified interface `handle()` for processing queries, allowing clients to remain unaware of underlying complexities.
- The `BackendManager` provides a simplified interface for routing data and requests to backend services without exposing their internal implementations.

5.3 Inheritance

Inheritance allows classes to reuse common functionality, establishing clear hierarchical relationships:

- Concrete implementations like `OrderTrackingHandler` inherit common behavior from the abstract `QueryHandler`.
- The `LiveAgentService` inherits from a common `Singleton` superclass, reusing logic to enforce a single instance.

5.4 Polymorphism

Polymorphism enables interchangeable use of classes through shared interfaces, enhancing system flexibility:

- Different `QueryHandler` implementations are interchangeable, allowing the `QueryManager` to delegate handling dynamically at runtime.
- Backend services accessed through the `BackendManager` use polymorphism to process diverse query types through a consistent interface.

6 SOLID Principles

- **Single Responsibility**: Each class has one clear, focused responsibility (e.g., `Agent` manages agent state, `BackendManager` routes requests).
- **Open/Closed**: New functionality (such as additional handlers or backend services) can be introduced via subclassing or adding new classes, without modifying existing code.
- **Liskov Substitution**: Subclasses of abstract base classes (e.g., handlers derived from `QueryHandler`) can replace the parent class seamlessly in the system, preserving correct behavior.

- **Interface Segregation**: Interfaces are kept minimal and specific. Components like `SessionManager` and `BackendManager` expose only methods relevant to their consumers.
- **Dependency Inversion**: In this particular project, the Dependency Inversion Principle is not employeed to a large degree due to the relatively controlled scope and complexity of the application. While absraction and polymorphism are used, many of the composed classes are accessed directly as singletons rather than through interfaces.

7 UML Class Diagram

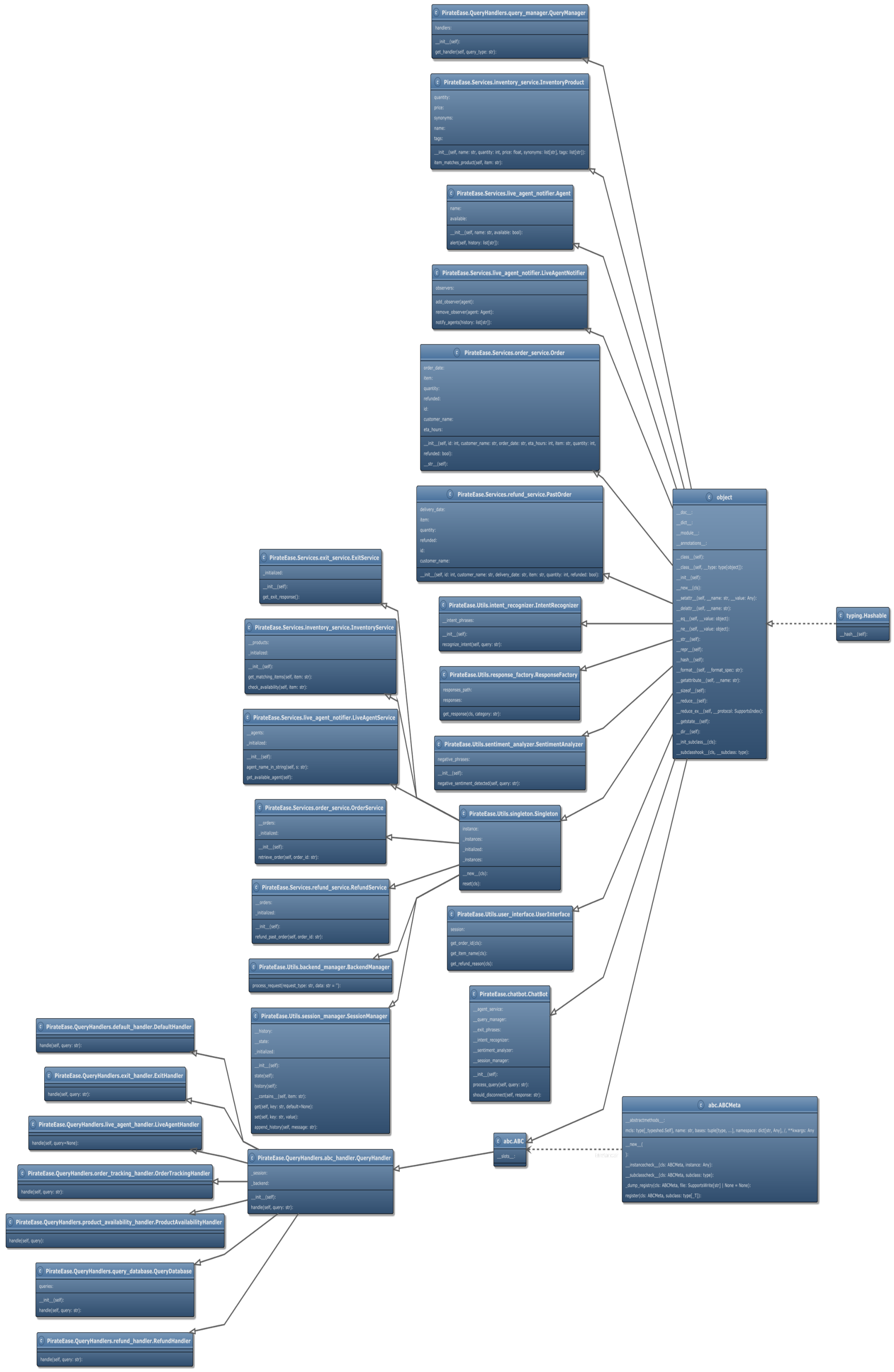


Figure 1: UML Class Diagram

Starting from the right side of the architecture diagram, several classes inherit from the `Hashable` interface. These classes are organized into clear, logical categories:

- **Data Storage Objects**
  - `InventoryProduct`
  - `Agent`
  - `Order`
  - `PastOrder`
- **Utility Classes**
  - `IntentRecognizer`
  - `ResponseFactory`
  - `SentimentAnalyzer`
  - `Singleton`
  - `UserInterface`
- **Notifier**
  - `LiveAgentNotifier`
- **Chatbot Core**
  - `ChatBot`

Additionally, the following classes explicitly implement the `Singleton` pattern:

- `ExitService`
- `InventoryService`
- `LiveAgentService`
- `OrderService`
- `RefundService`
- `BackendManager`
- `SessionManager`

Finally, the query-handling logic is encapsulated within distinct `QueryHandler` implementations, each derived from the abstract `QueryHandler` interface:

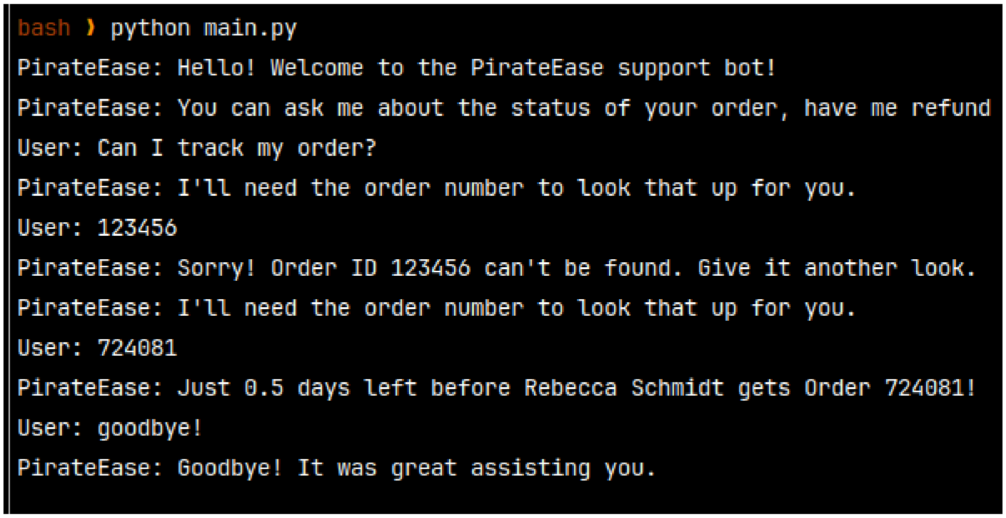
- `DefaultHandler`
- `ExitHandler`
- `LiveAgentHandler`
- `OrderTrackingHandler`
- `ProductAvailabilityHandler`
- `RefundHandler`
- `QueryDatabase`

8 Test Coverage

File ▲	statements	missing	excluded	coverage
PirateEase__init__.py	0	0	0	100%
PirateEase\chatbot.py	35	0	0	100%
PirateEase\QueryHandlers__init__.py	0	0	0	100%
PirateEase\QueryHandlers\default_handler.py	8	0	0	100%
PirateEase\QueryHandlers\exit_handler.py	5	0	0	100%
PirateEase\QueryHandlers\live_agent_handler.py	10	0	0	100%
PirateEase\QueryHandlers\order_tracking_handler.py	20	0	0	100%
PirateEase\QueryHandlers\product_availability_handler.py	8	0	0	100%
PirateEase\QueryHandlers\query_database.py	13	0	0	100%
PirateEase\QueryHandlers\query_manager.py	14	0	0	100%
PirateEase\QueryHandlers\refund_handler.py	22	0	0	100%
PirateEase\Services__init__.py	0	0	0	100%
PirateEase\Services\exit_service.py	9	0	0	100%
PirateEase\Services\inventory_service.py	36	0	0	100%
PirateEase\Services\live_agent_notifier.py	42	0	0	100%
PirateEase\Services\order_service.py	28	0	0	100%
PirateEase\Services\refund_service.py	28	0	0	100%
PirateEase\Utils__init__.py	0	0	0	100%
PirateEase\Utils\backend_manager.py	20	0	0	100%
PirateEase\Utils\intent_recognizer.py	11	0	0	100%
PirateEase\Utils\response_factory.py	10	0	0	100%
PirateEase\Utils\sentiment_analyzer.py	8	0	0	100%
PirateEase\Utils\session_manager.py	23	0	0	100%
PirateEase\Utils\singleton.py	11	0	0	100%
PirateEase\Utils\slow_print.py	6	0	0	100%
PirateEase\Utils\user_interface.py	37	0	0	100%
Total	**404**	**0**	**0**	**100%**

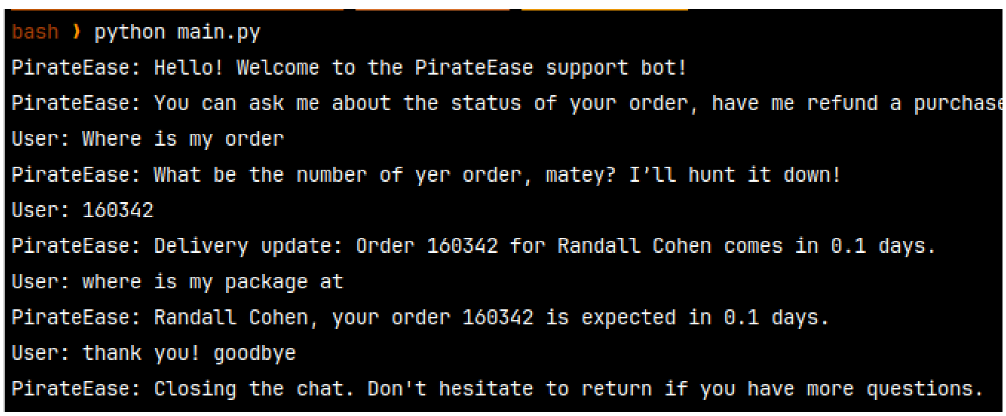
Figure 2: Test Coverage

9 Screenshots



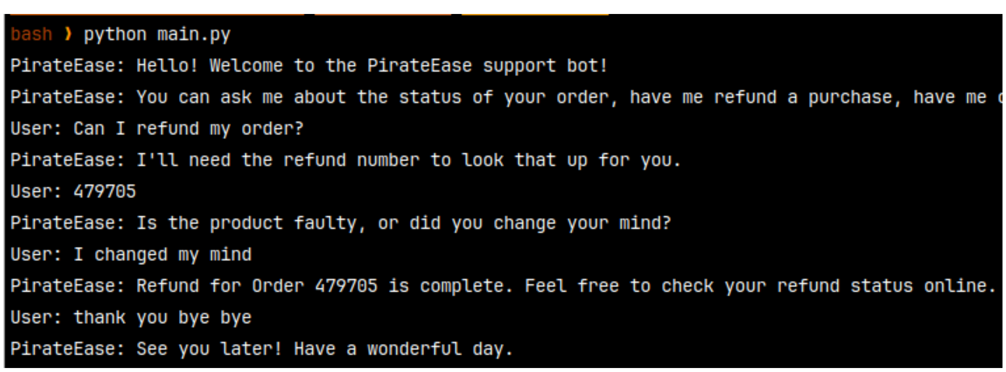
```
bash ❯ python main.py
PirateEase: Hello! Welcome to the PirateEase support bot!
PirateEase: You can ask me about the status of your order, have me refund
User: Can I track my order?
PirateEase: I'll need the order number to look that up for you.
User: 123456
PirateEase: Sorry! Order ID 123456 can't be found. Give it another look.
PirateEase: I'll need the order number to look that up for you.
User: 724081
PirateEase: Just 0.5 days left before Rebecca Schmidt gets Order 724081!
User: goodbye!
PirateEase: Goodbye! It was great assisting you.
```

Figure 3: User enters wrong order ID followed by correct order ID



```
bash ❯ python main.py
PirateEase: Hello! Welcome to the PirateEase support bot!
PirateEase: You can ask me about the status of your order, have me refund a purchase
User: Where is my order
PirateEase: What be the number of yer order, matey? I'll hunt it down!
User: 160342
PirateEase: Delivery update: Order 160342 for Randall Cohen comes in 0.1 days.
User: where is my package at
PirateEase: Randall Cohen, your order 160342 is expected in 0.1 days.
User: thank you! goodbye
PirateEase: Closing the chat. Don't hesitate to return if you have more questions.
```

Figure 4: Remembers order ID



```
bash ❯ python main.py
PirateEase: Hello! Welcome to the PirateEase support bot!
PirateEase: You can ask me about the status of your order, have me refund a purchase, have me
User: Can I refund my order?
PirateEase: I'll need the refund number to look that up for you.
User: 479705
PirateEase: Is the product faulty, or did you change your mind?
User: I changed my mind
PirateEase: Refund for Order 479705 is complete. Feel free to check your refund status online.
User: thank you bye bye
PirateEase: See you later! Have a wonderful day.
```

Figure 5: Successful Refund

```
bash ❯ python main.py
PirateEase: Hello! Welcome to the PirateEase support bot!
PirateEase: You can ask me about the status of your order, have me refund a purchase, ha
User: is the jolly roger available
PirateEase: Let me know which product you're asking about and I'll check availability.
User: eyepatch
PirateEase: We've run out of Eyepatch, unfortunately.
User: ok thank you bye
PirateEase: Goodbye! It was great assisting you.
```

Figure 6: Inventory out of Stock

```
bash ❯ python main.py
PirateEase: Hello! Welcome to the PirateEase support bot!
PirateEase: You can ask me about the status of your order, have me refund a purchase, have me check
User: Do you ship to PO Boxes
PirateEase: Yes, we ship to PO boxes via standard delivery. Expedited options may not be available.
User: thank you. bye
PirateEase: Bye for now! Always here when you need help.
```

Figure 7: Response from Database

```
bash ❯ python main.py
PirateEase: Hello! Welcome to the PirateEase support bot!
PirateEase: You can ask me about the status of your order, have me refund a purchase, have me check
User: Can I get all your products for free?
PirateEase: I'll need to pass this along to someone who knows more. Can I help with anything else?
User: This is the worst service ever!
PirateEase: I'm really sorry this happened. Let me get someone to assist you right away.
Bringing in Charles Vane now...
```

Figure 8: Live Agent Escalation