INTERNATIONAL UNIVERSITY OF APPLIED SCIENCES

AI Use Case

Phase 2 (Documentation):

Hotel Room Booking ChatBot

Implementation and Reflection Phase

Author: Mostafa Saeedan

Matriculation number: 9210293

Tutor: Dr. Tim Schlippe

Feb 2025

Table of Contents

Objective of Phase 2

Goal:

In Phase 2 we are to implement the core functionality of the hotel booking chatbot. I'm focusing on transitioning the project from the conceptual design stage to an operational system, ensuring that the chatbot is functional, interactive, and capable of:

- Collecting booking details from users (e.g., name, check-in and check-out dates, number of guests).
- Validating and storing data in a structured database.
- Handling basic user interactions with defined intents and actions.
- Managing fallback scenarios and reset functionalities effectively.

Scope:

1. **Implementing Core Functionalities:**
   - Developing conversation flows using predefined intents and actions.
   - Creating and managing a SQLite database for storing booking information.
2. **Testing and Debugging:**
   - Ensuring proper responses for user queries.
   - Validating database updates for different booking scenarios.
3. **Integration and Setup:**
   - Connecting the Rasa framework with backend logic (e.g., custom Python scripts).
   - Providing a simple web interface for interaction, if applicable.

Deliverables

By the end of this phase, the following deliverables are expected:

- A fully functional chatbot capable of booking hotel rooms.
- A database (SQLite) that stores user bookings and relevant details.
- A well-documented testing and debugging report.
- Screenshots or transcripts demonstrating successful interaction and functionality.

Key Milestones

1. **Conversation Design:**
   - Completion of intents, entities, and slots in domain.yml.
   - Development of NLU training data and stories.
2. **Custom Actions:**
   - Implementation of Python scripts in actions.py for database interaction.
3. Database Management:
   - Auto-generation of the database (hotel_management.db) upon first run.
   - Validation of booking entries via SQL queries.

4. **Testing:**
    - Successful execution of test cases for all core functionalities.
    - Identification and resolution of errors encountered during execution.

Project Overview

Recap of the Conception

Phase 1 focused on defining the conceptual design and objectives of the hotel booking chatbot. Key deliverables from this phase included:

- **Objective Definition:**
    - Creating a chatbot capable of simplifying the hotel room booking process.
    - Automating basic queries to reduce workload for customer service representatives.
- **Users Identified:**
    - Potential hotel guests.
    - Hotel staff requiring access to bookings.
- **Key Features Outlined:**
    - Room booking functionality with user details.
    - Query handling and fallback management.

Focus in Phase 2

The primary focus of Phase 2 is the implementation and testing of the chatbot based on the conceptual design. Specific areas include:

- **Building Functionalities:**
- Implementing intents, entities, and actions in the Rasa framework.
    - Creating a database for storing and retrieving booking information.
- **Testing User Interaction:**
    - Ensuring seamless conversations and validating database transactions.
- **Debugging and Refinement:**
    - Addressing technical challenges encountered during the development process.

Technology Stack

- The project leverages the following technologies:
- Programming Language: Python
- Framework: Rasa (for chatbot development)
- Database: SQLite

Implementation

System Architecture:

The chatbot flow begins with the user's input, which is processed by the Rasa NLU module. The system identifies the user's intent (e.g., booking a room, asking for help) and extracts relevant entities (e.g., name, dates, number of guests). Based on this analysis, the conversation progresses through predefined stories and responses in the domain.yml file.

Key flow:

- User Input → Intent Recognition → Entity Extraction → Response Selection
- For database-related queries (e.g., storing bookings), the system triggers custom actions defined in actions.py to interact with the SQLite database (hotel_management.db).
- Custom actions validate inputs, update the database, and return results or confirmations to the user.

Example:

When a user provides booking details, the chatbot:

- Identifies the intent (book_room).
- Extracts entities like name, check-in date, check-out date, and number of guests.
- Validates the data and stores it in the database using a custom action.
- Responds with a confirmation message.

This architecture ensures seamless interaction, accurate data handling, and a smooth user experience.

Components

- **actions.py:** Handles database interactions and custom chatbot actions. This script includes logic for validating user inputs, storing data in the database, and providing feedback to users.
- **domain.yml:** Defines the intents, slots, entities, and responses that dictate the chatbot's behavior. It acts as the central configuration file for the chatbot.
- **nlu.yml:** Contains natural language training data to help the chatbot recognize intents and extract entities from user input effectively.
- **rules.yml:** Sets predefined rules for conversation paths, ensuring logical progression during interactions.
- **stories.yml:** Provides example conversations for training the chatbot. These help the bot learn how to handle different scenarios.
- **chatbot.html:** An optional web interface that allows users to interact with the chatbot through a browser.

Database

The SQLite database hotel_management.db is created automatically by the logic in actions.py when the chatbot runs for the first time. If the database does not exist, the system ensures it is initialized with the required schema. This database is integral to storing and managing hotel booking data, ensuring efficient interaction between the chatbot and backend.

Tables Contained in the Database

1. **hotels:**
   - Stores information about hotels, such as name, location, and star rating.
   - Columns: hotel_id (Primary Key), hotel_name, location, star_rating.
2. **rooms:**
   - Maintains details of rooms within hotels, including type, price, and availability status.
   - Columns: room_id (Primary Key), hotel_id (Foreign Key), room_type, price_per_night, availability_status.
3. **guests:**
   - Tracks guest information, such as full name and contact details.
   - Columns: guest_id (Primary Key), full_name, contact_info.
4. **bookings:**
   - Manages booking records, associating guests with rooms for specific dates and costs.
   - Columns: booking_id (Primary Key), guest_id (Foreign Key), room_id (Foreign Key), check_in_date, check_out_date, total_cost.

Interaction with the Chatbot

- **Custom Actions:**
  - The chatbot triggers actions in actions.py to execute database operations such as creating bookings, retrieving available rooms, and recording payments.
  - For example, when a user provides booking details, the create_booking function stores the data in the bookings table.

- **Validation:**
    - The chatbot checks the database to validate inputs like room availability and guest details before confirming a booking.
- **Real-Time Updates:**
    - Database records are updated dynamically to reflect changes such as room availability or payment status.

This automated and seamless database integration ensures that the chatbot can handle user queries effectively while maintaining data accuracy and consistency.

Testing and Debugging

Testing Scenarios

1. **Booking a Room:**
    - User initiates booking intent by providing relevant information like name, check-in date, check-out date, and number of guests.
    - Expected Outcome: The bot collects all details, validates them, and confirms the booking with a unique ID.
2. **Invalid Input Handling:**
    - User provides incorrect or incomplete information (e.g., missing dates or an invalid guest count).
    - Expected Outcome: The bot prompts the user to provide the missing or corrected details.
3. **Database Interaction:**
    - Bot validates if the room is available in the database and updates booking details accordingly.
    - Expected Outcome: The database reflects the correct booking information after interaction.

4. **Fallback Handling:**
   - User enters an unrelated or unclear message.
   - Expected Outcome: The bot provides a fallback response guiding the user back to valid inputs.
5. **Reset Functionality:**
   - User requests to reset the conversation.
   - Expected Outcome: All slots are cleared, and the bot restarts the interaction.

Challenges Encountered

1. **Database Integration Issues:**
   - Initial challenges in ensuring the database auto-generates and updates correctly.
   - Encountered mismatched data types during SQL queries, causing booking failures.
2. **Fallback Logic:**
   - Difficulty in setting up robust fallback responses for unclear user inputs.
   - Ensuring that the bot smoothly transitions back to the booking flow after fallbacks.
3. **Conversation Flow Validation:**
   - Handling edge cases where users skip required slots or provide incomplete information.
4. **Custom Action Errors:**
   - Debugging Python scripts in actions.py for proper data retrieval and updates.
   - Ensuring actions execute within the required timeframe to maintain responsiveness.
5. **Testing Complexity:**
   - Simulating various user inputs and edge cases to validate the bot's behavior.
   - Ensuring consistency between the chatbot's responses and the database state.

Resolutions:

1. **Database Integration Fixes:**
- Adjusted SQL queries to ensure data types align with the database schema.
- Added logic in actions.py to check database connection before executing queries.

2. **Improved Fallback Logic:**

- Enhanced fallback messages to provide clear guidance for correcting user inputs.
- Updated the fallback policy in rules.yml to allow smooth recovery of conversation flow.

3. **Refined Conversation Flows:**
   - Added extra validation for slots in domain.yml to handle incomplete or invalid user data.
   - Introduced additional stories in stories.yml to cover edge cases and unusual scenarios.
4. **Custom Action Debugging:**
   - Used logging in actions.py to identify errors in real-time.
   - Fixed timeout issues by optimizing database queries and Python scripts.
5. **Testing Enhancements:**
   - Developed a comprehensive set of test cases to validate every intent and action.
   - Utilized debugging tools to simulate different user interactions and scenarios.

Screenshots and Demonstration

Screenshots and Logs

Reflection

Challenges:

1. **Database Integration Issues:**
   - Description: Initial challenges arose while integrating the SQLite database with the chatbot. The system sometimes failed to create tables automatically or encountered data type mismatches during SQL queries.

- Impact: This led to booking failures and inconsistent data storage.

2. **Custom Action Errors:**
   - Description: Errors were encountered when executing custom actions in actions.py, particularly when handling database read/write operations.
   - Impact: The chatbot would fail to confirm bookings, leading to a poor user experience.
3. **Intent Misclassification:**
   - Description: The chatbot occasionally misclassified user intents, especially when the phrasing was slightly different from the training data.
   - Impact: This resulted in incorrect responses and disrupted conversation flows.
4. **Fallback Handling:**
   - Description: The fallback mechanism was not robust enough to handle unexpected or unclear user inputs.
   - Impact: Users were often left without proper guidance on how to continue the conversation.
5. **Slot Validation Issues:**
   - Description: The chatbot struggled with validating slots correctly, especially when users provided incomplete or invalid data.
   - Impact: This caused errors in booking confirmations and incomplete data entries in the database.

Resolutions:

1. Database Integration Fixes:

- Modified actions.py to include additional checks ensuring that the database is created if it doesn't exist.
  - Adjusted SQL queries to handle data type mismatches and ensure smooth database operations.

2. Improved Custom Actions:

- Added error-handling mechanisms in actions.py to manage exceptions during database interactions.
- Implemented logging to track errors, making it easier to debug and fix issues.

3. Enhanced Intent Classification:
   - Expanded the training data in nlu.yml with more varied examples to improve the chatbot's understanding of different user inputs.
   - Fine-tuned the NLU model to reduce misclassification rates.
4. Robust Fallback Logic:
   - Updated the fallback policy in rules.yml to provide more informative responses when the chatbot doesn't understand user input.
   - Created additional fallback stories in stories.yml to guide users back to the correct flow.
5. Slot Validation Improvements:

- Implemented custom validation functions in actions.py to ensure that user-provided data meets the required criteria.
  - Added conditional checks to prompt users for missing information before proceeding with bookings.

Future Enhancements

While the current version of the chatbot meets the basic requirements for hotel booking, there are several areas where it can be enhanced to provide a more comprehensive and user-friendly experience.

1. **Multilingual Support:**

• **Description:** Adding support for multiple languages would make the chatbot accessible to a broader audience, especially for international hotel guests.

• **Implementation Approach:**

• Utilize Rasa's built-in multilingual NLU capabilities to handle different languages.

• Train the NLU model with multilingual datasets and provide language-specific responses in domain.yml.

2. **Real-Time Room Availability Updates:**

- **Description:** Integrating real-time room availability would allow users to see the most current status of room bookings without delays.
- **Implementation Approach:**
- Connect the chatbot to an external API or hotel management system that updates room availability in real time.
    - Modify actions.py to fetch real-time data instead of relying solely on static database queries.

3. **Anaytics Dashboard for Hotel Admins:**

- **Description:** Creating an analytics dashboard would enable hotel staff to monitor booking trends, user interactions, and chatbot performance.
- **Implementation Approach:**
    - Develop a web-based dashboard using frameworks like Flask or Django to visualize booking data stored in the database.
    - Include features like booking statistics, user activity logs, and performance metrics.

4. **Enhanced Payment Integration:**

- **Description:** Adding secure online payment functionality would streamline the booking process.
- **Implementation Approach:**
    - Integrate payment gateways like Stripe or PayPal using APIs.
    - Update actions.py to handle payment processing, confirmation, and error handling securely.

5. **Voice Assistant Integration:**

- **Description:** Extending the chatbot to work with voice assistants (e.g., Alexa, Google Assistant) would enhance accessibility.
- **Implementation Approach:**
- Leverage Rasa's voice integration capabilities or third-party APIs to enable voice interactions.
    - Adapt the NLU model to handle voice inputs effectively.

Appendix

Full Code Snippets for Key Functionalities:

```python
# Importing necessary modules and classes for type hinting, date parsing, database operations,
and Rasa SDK functionality
from typing import Any, Text, Dict, List, Optional
from datetime import datetime  # For working with dates and times
import dateparser  # For parsing user-provided date text into a standard format
import sqlite3  # For database operations (e.g., storing booking data)
import re  # For regular expression operations (e.g., text processing)
from rasa_sdk import Action, Tracker  # For defining custom actions and tracking conversation
state
from rasa_sdk.executor import CollectingDispatcher  # For dispatching responses to the user
from rasa_sdk.events import SlotSet, EventType, FollowupAction, AllSlotsReset, Restarted  # For
managing events and slots in the conversation
import logging # For logging messages to the console
import os

logger = logging.getLogger(__name__) # For logging messages to the console


# -----------------------------------------------------------------------------
# 1) A helper function to parse user-provided date text to a standard format
#    (e.g., "dd-mm-yyyy"). Using dateparser for flexible date input.
# -----------------------------------------------------------------------------
def parse_date_basic(date_text: str) -> Optional[datetime]:
    """
    A simple dateparser-based function that returns a Python datetime
    if parse is successful, or None if it fails.
    This does NOT do any fancy future-year logic. We rely on other logic
    to interpret it relative to check-in, etc.
    """
    if not date_text:
        return None
    parsed = dateparser.parse(date_text)
    return parsed

def parse_checkout_relative(checkout_text: str, dt_checkin: datetime) -> Optional[datetime]:
    """
    Parses the user’s checkout text. If the text includes a year explicitly,
    dateparser will handle that. If not, dateparser might place it in the
    current year. If that ends up before dt_checkin, we try bumping to the
    same year as check-in or the next year.
    """
    if not checkout_text:
        return None

    dt_checkout = dateparser.parse(checkout_text)     # parse with dateparser ignoring year (if
user didn't specify)
    if not dt_checkout:
        return None

    if dt_checkout >= dt_checkin:     # If the parsed checkout is already >= checkin, great —
no changes
        return dt_checkout

    return dt_checkout

# --------------------------------------------------------------------------------
# 2) ActionValidateInputs
#    This action runs after the form is completed to validate or correct data.
#    Then it saves the data in a (dummy) database.
# --------------------------------------------------------------------------------
class ActionValidateInputs(Action):
    def name(self) -> Text:
        return "action_validate_inputs"

    async def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any]
    ) -> List[EventType]:
        """
        1. Retrieve slots from the form: name, checkin_date, checkout_date, number_of_guests
        2. Validate each:
            - name: not empty
            - checkin_date: not in the past
            - checkout_date: strictly after checkin_date
            - number_of_guests: integer, 1..20
        3. If invalid, re-prompt the user by clearing the relevant slot(s).
        4. If valid, store row in DB, return booking_id -> slot, so "utter_confirm_booking" can show it.
        """

        # ----------------------------------------------------------------------
        # Retrieve slots
        # ----------------------------------------------------------------------
        name_slot = tracker.get_slot("name") or ""
        checkin_slot = tracker.get_slot("checkin_date") or ""
        checkout_slot = tracker.get_slot("checkout_date") or ""
        guests_slot = tracker.get_slot("number_of_guests") or ""

        # ----------------------------------------------------------------------
        # Validate / parse name (simple capitalization)
        # ----------------------------------------------------------------------
        if not name_slot.strip():
            dispatcher.utter_message(
                text="Please provide a valid name."
            )
            # Clearing the slot to force the bot to re-ask
            return [SlotSet("name", None), FollowupAction("hotel_booking_form")]


        formatted_name = " ".join(word.capitalize() for word in name_slot.split())

        # ----------------------------------------------------------------------
        # Validate / parse check-in & check-out dates
        # ----------------------------------------------------------------------
        # Parse checkin
        dt_checkin = parse_date_basic(checkin_slot)
        if not dt_checkin:
            dispatcher.utter_message(
                text="Please provide a valid check-in date (e.g. '25 January 2025')."
            )
            return [
                SlotSet("checkin_date", None),
                FollowupAction("hotel_booking_form")
            ]


        # If checkin in the past, re-prompt
        now = datetime.now()
        if dt_checkin < now:
            dispatcher.utter_message(
                text=(
                    "Your check-in date is in the past!\n"
                    "If you meant next year, please specify the year.\n"
                )
            )
            return [
                SlotSet("checkin_date", None),
                FollowupAction("hotel_booking_form")
            ]

        # Parse checkout in a "relative" manner
        dt_checkout = parse_checkout_relative(checkout_slot, dt_checkin)
        if not dt_checkout:
            logger.warning("Check-out date is invalid or unparseable.")
            dispatcher.utter_message(
                text="Please provide a valid check-out date (e.g. '26 January 2025')."
            )
            return [
                SlotSet("checkout_date", None),
                FollowupAction("hotel_booking_form")
            ]

        # Now if the final dt_checkout is STILL <= dt_checkin, fail
        if dt_checkout <= dt_checkin:
            logger.warning("Check-out date is earlier than or equal to check-in date.")
            dispatcher.utter_message(
                text=(
                    "Check-out date must be after your check-in date.\n"
                    "If you meant a date next year, please specify that explicitly.\n"
                )
            )
            return [
                SlotSet("checkout_date", None),
                FollowupAction("hotel_booking_form")
            ]

        # At this point, we have validated checkin in the future, checkout > checkin
        # Format them as dd-mm-yyyy strings
        parsed_checkin_str = dt_checkin.strftime("%d-%m-%Y")
        parsed_checkout_str = dt_checkout.strftime("%d-%m-%Y")

        # ------------------------------------------------------------------------
        # Validate / parse number_of_guests
        # ------------------------------------------------------------------------
        # e.g. try to extract an integer from guests_slot
        import re
        found_digits = re.findall(r"\d+", guests_slot)
        if not found_digits:
            logger.warning("Number of guests is invalid or unparseable.")
            dispatcher.utter_message(
                text="Please provide a valid number of guests (e.g. '3')."
            )
            return [
                SlotSet("number_of_guests", None),
                FollowupAction("hotel_booking_form"),
            ]

        # We take the first digit we find as the guest number
        guests_count = int(found_digits[0])
        if guests_count < 1 or guests_count > 20:
            logger.warning(f"Guests count {guests_count} is outside the valid range (1–20).")
            dispatcher.utter_message(
                text="Please provide a valid number of guests between 1 and 20."
            )
            return [
                SlotSet("number_of_guests", None),
                FollowupAction("hotel_booking_form"),
            ]

        # ---------------------------------------------------------------------
        # 3) Check availability in a dummy database & "save" the booking
        # ---------------------------------------------------------------------
        #  using local SQLite.can be Adjusted for real database.

        # 3a. Connect to DB (or create if not exist)
        #     In production, will connect to a remote DB or use SQLAlchemy.

        os.makedirs("database", exist_ok=True)
        conn = sqlite3.connect("database/hotel_bookings.db")
        cursor = conn.cursor()

        # 3b. Create a table if not exists
        cursor.execute(
            """
            CREATE TABLE IF NOT EXISTS bookings (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT,
                checkin_date TEXT,
                checkout_date TEXT,
                guests INTEGER
            )
            """
        )

        # 3c. Here normally check if there's an available room.
        #     We'll assume there's always availability for simplicity.
        #     If not available, we'll do something like:
        # dispatcher.utter_message(text="No rooms available for those dates!")
        # return []
        # 3d. Insert the booking row
        cursor.execute(
            """
            INSERT INTO bookings (name, checkin_date, checkout_date, guests)
            VALUES (?, ?, ?, ?)
            """,
            (
                formatted_name,
                parsed_checkin_str,  # dd-mm-yyyy
                parsed_checkout_str, # dd-mm-yyyy
                guests_count
            ),
        )

        conn.commit()

        # we retrieve the newly inserted booking ID or other info
        booking_id = cursor.lastrowid

        # Close the DB connection
        conn.close()


        # Convert booking_id to string for the slot
        str_booking_id = str(booking_id)

        # ----------------------------------------------------------------------
        # 4) Return a final "summary" to user or rely on utter_confirm_booking
        #    The rule has "action_validate_inputs" -> "utter_confirm_booking",
        #    which uses the domain's template. We'll just set the final, validated
        #    slots here for the template.
        # ----------------------------------------------------------------------
        # Example: "Your booking for Mikel from 25-01-2023 to 26-01-2023 for 3 guests is
confirmed!"
        # The domain's "utter_confirm_booking" uses {name}, {checkin_date}, etc.

        # Store final validated values in the slots so "utter_confirm_booking" sees them
        return [
            SlotSet("name", formatted_name),
            SlotSet("checkin_date", parsed_checkin_str),
            SlotSet("checkout_date", parsed_checkout_str),
            SlotSet("number_of_guests", str(guests_count)),
            SlotSet("booking_id", str_booking_id),
        ]

# ------------------------------------------------------------------------------
# 3) ActionResetSlots
#    This action clears all slots when the user says 'reset'.
# ------------------------------------------------------------------------------
class ActionResetSlots(Action):
    """Clears all slots when user says 'reset'."""

    def name(self) -> Text:
        return "action_reset_slots"

    async def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any]
    ) -> List[EventType]:
        dispatcher.utter_message(text="Resetting all your data. Let's start fresh!")
        # Option 1: Clear all slots:
        return [AllSlotsReset(), FollowupAction("action_listen")]
        # Option 2: If prefer to do something else after resetting,
        # we can also return [AllSlotsReset(), FollowupAction("action_listen")]
        # or a new greeting, etc.

# ------------------------------------------------------------------------------
# 4) ActionRestartSession
#    Fully restarts the conversation with a brand-new session.
# ------------------------------------------------------------------------------
class ActionRestartSession(Action):
    """Fully restarts the conversation (new session)."""

    def name(self) -> Text:
        return "action_restart_session"

    async def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any]
    ) -> List[EventType]:
        dispatcher.utter_message(text="Starting a completely new conversation now!")
        # The "Restarted()" event instructs Rasa Core to create a brand-new session
        # so your entire flow restarts from scratch
        return [AllSlotsReset(), Restarted()]

# ------------------------------------------------------------------------------
# 5) ActionDefaultFallback
#   This action is called when the user's input is not recognized by Rasa's NLU.
# ------------------------------------------------------------------------------
class ActionDefaultFallback(Action):
    """Called when the user’s input is not recognized by Rasa’s NLU"""
    def name(self) -> Text:
        return "action_default_fallback"

    async def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any]
    ) -> List[EventType]:
        dispatcher.utter_message(
            text="I'm sorry, I didn't understand that. Type 'help' for instructions or 'reset'
to start over."
        )
        return []
```

Resources and References

- **Rasa Documentation:** https://rasa.com/docs/
- **SQLite Official Documentation:** https://www.sqlite.org/docs.html
- **Python Official Documentation:** https://docs.python.org/3/
- **Flask (for API integration):** https://flask.palletsprojects.com/