A picture containing text, outdoor, building, white

Description automatically generated

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Student name: | Student 1: Karen Ferreira Magalhaes  Student 2: Thales Campos  Student 3: Vitor Freitas | | | | | |
| Student number: | Student 1: 3146094  Student 2: 3151261  Student 3: 3152612 | | | | | |
| Faculty: | Computing Science | | | | | |
| Course: | BSCH/BSCO/EXCH | | | Stage/year: | 2 | |
| Subject: | Software Development 2 | | | | | |
| Study Mode: | Full time | Icon  Description automatically generated |  | Part-time |  |  |
| Lecturer Name: | Haseeb Younis/ Muhammad Shoaib | | | | | |
| Assignment Title: | Prototype Documentation | | | | | |
| Date due: | 27/04/2025 | | |  | | |
| Date submitted: | 23/03/2025 | | |  | | |
| **Plagiarism disclaimer:**  *I understand that plagiarism is a serious offence and have read and understood the college policy on plagiarism. I also understand that I may receive a mark of zero if I have not identified and properly attributed sources which have been used, referred to, or have in any way influenced the preparation of this assignment, or if I have knowingly allowed others to plagiarise my work in this way.*  *I hereby certify that this assignment is my own work, based on my personal study and/or research, and that I have acknowledged all material and sources used in its preparation. I also certify that the assignment has not previously been submitted for assessment and that I have not copied in part or whole or otherwise plagiarised the work of anyone else, including other students.*  **Signed: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | | | | | | |
| **Please note:** Students **MUST** retain a hard / soft copy of **ALL** assignments as well as a receipt issued and signed by a member of Faculty as proof of submission. | | | | | | |

**Documentation – Milestone 01**

**ChatGPTClient Class Documentation**

Overview: The ChatGPTClient class handles communication with the OpenAI ChatGPT API. It sends user messages, receives AI-generated responses, and processes the API response using the Gson library.

Package: main.java.com.tripper

Dependencies:

* java.io.\* (For input/output operations)
* java.net.\* (For handling HTTP connections)
* com.google.gson.\* (For JSON processing)

Constants:

* String API\_URL: The endpoint URL for OpenAI's ChatGPT API.
* String API\_KEY: The API key used for authentication (should be kept secret and not hardcoded in production).

Methods:

1. String getChatResponse(String conversationContext):
   * Sends a user message to the OpenAI ChatGPT API and retrieves the AI-generated response.
   * Parameters:
     + conversationContext: The user's input message to be processed by the AI.
   * Returns:
     + A string containing the AI-generated response.
   * Process:
     + Establishes an HTTP connection with the OpenAI API.
     + Constructs a JSON payload containing the request data.
     + Sends the request and reads the API response.
     + Parses the JSON response to extract the AI's reply.
     + Returns the processed response or an error message if the API call fails.

Exception Handling:

* Catches general exceptions during the API request and response handling.
* Prints stack traces for debugging and returns an error message when an issue occurs.

Security Considerations:

* The API key should not be stored in the source code; it should be loaded from environment variables or secure storage in production.
* Ensure proper exception handling to avoid exposing sensitive information in error messages.

**ConversationController Class Documentation**

Overview: The ConversationController class manages the chatbot's interaction flow, handling user input, processing responses, and guiding the conversation through predefined states.

Package: main.java.com.tripper

Dependencies:

* java.util.Scanner (For user input handling)

Attributes:

* ConversationState state: Stores conversation progress and user data.
* ConversationManager conversationManager: Manages chatbot responses.
* ChatGPTClient chatGPTClient: Communicates with OpenAI's API.
* Scanner scanner: Handles user input from the console.

Methods:

1. void run():
   * Manages the chatbot's state-based conversation flow.
   * Guides the user through different states: GREETING, COLLECT\_TRIP\_DETAILS, CONFIRM\_DETAILS, GENERATE\_RECOMMENDATIONS, OFFER\_PDF, END.
   * Integrates responses from ChatGPTClient.
   * Handles user inputs for trip details and PDF generation.
2. void typePrint(String message, int delayMs):
   * Simulates a typing effect when displaying chatbot messages.
   * Parameters:
     + message: The text to display.
     + delayMs: Delay per character in milliseconds.
3. String prompt(String message):
   * Displays a message and collects user input.
   * Parameters:
     + message: The prompt for the user.
   * Returns:
     + User input as a trimmed string.

Exception Handling:

* Ensures smooth conversation flow by handling user input errors.
* Uses try-catch to handle interruptions in the typing effect.

Functionality:

* Starts the chatbot with a greeting message.
* Collects and processes user trip details.
* Requests recommendations from ChatGPTClient.
* Offers an option to generate a PDF checklist.
* Ends the conversation with a farewell message.

**ConversationManager Class Documentation**

Overview: The ConversationManager class handles the chatbot's predefined responses to user input. It generates friendly and engaging messages to guide users through the interaction process.

Package: main.java.com.tripper

Methods:

1. String getGreeting(String userName):
   * Generates a personalized greeting for the user.
   * Parameters:
     + userName: The name of the user.
   * Returns:
     + A friendly welcome message including the user's name.
2. String askForTripDetails():
   * Prompts the user to provide details about their trip.
   * Returns:
     + A request message asking the user for trip information.
3. String friendlyResponse(String dynamicResponse):
   * Formats and returns a friendly response incorporating dynamically generated recommendations.
   * Parameters:
     + dynamicResponse: The AI-generated travel recommendations.
   * Returns:
     + A structured response including the recommendations.

Functionality:

* Ensures a smooth and engaging chatbot experience.
* Provides user-friendly prompts and structured responses to enhance interaction.
* Acts as an intermediary between user input and AI-generated recommendations.

**ConversationState Class Documentation**

Overview: The ConversationState class maintains the state of a chatbot conversation. It stores user-specific data such as name, trip details, and confirmation status to ensure a smooth interaction flow.

Package: main.java.com.tripper

Dependencies:

* java.util.ArrayList (For handling dynamic lists)
* java.util.List (For managing location storage)

Attributes:

* String userName: Stores the user's name.
* List locations: Holds the list of travel locations provided by the user.
* String tripDetails: Contains user-supplied details about the trip.
* boolean detailsConfirmed: Indicates whether the trip details have been confirmed by the user.

Methods:

1. ConversationState():
   * Constructor that initializes the locations list and sets detailsConfirmed to false.
2. String getUserName():
   * Retrieves the user's name.
   * Returns:
     + The name of the user.
3. void setUserName(String userName):
   * Sets the user's name.
   * Parameters:
     + userName: The name of the user.
4. List getLocations():
   * Retrieves the list of locations.
   * Returns:
     + A list of location names.
5. void addLocation(String location):
   * Adds a new location to the list.
   * Parameters:
     + location: The name of the location to add.
6. String getTripDetails():
   * Retrieves the trip details provided by the user.
   * Returns:
     + The trip details as a string.
7. void setTripDetails(String tripDetails):
   * Sets the trip details.
   * Parameters:
     + tripDetails: A string containing trip information.
8. boolean isDetailsConfirmed():
   * Checks if the trip details have been confirmed by the user.
   * Returns:
     + True if confirmed, false otherwise.
9. void setDetailsConfirmed(boolean detailsConfirmed):
   * Updates the confirmation status of the trip details.
   * Parameters:
     + detailsConfirmed: Boolean value indicating confirmation status.

Functionality:

* Tracks user input and progress throughout the chatbot conversation.
* Stores key trip-related information for personalized recommendations.
* Ensures state persistence for an improved chatbot experience.

**InputParser Class Documentation**

Overview:  
The InputParser class processes and extracts travel-related information from user input. It uses regular expressions and string manipulation to identify the travel month and potential locations provided by the user.

Package:  
main.java.com.tripper

Dependencies:

* java.util.\* (For handling collections and arrays)
* java.util.regex.\* (For regular expression matching)

Constants:

* Pattern monthPattern:  
  A regular expression pattern to match any month name (case-insensitive).

Methods:

1. TripDetails parseTripDetails(String input):
   * Parameters:
     + input: The user-provided string containing trip details.
   * Returns:
     + A TripDetails object containing the extracted travel month and locations.
   * Process:
     + Uses regex to identify a month in the input and sets the travel month in the TripDetails object.
     + Splits the input into tokens and identifies capitalized words (excluding month names) as potential locations.
     + Removes punctuation from the tokens and adds valid locations to the list.

Functionality:

* Identifies and extracts the travel month and location information from a free-form user input string.
* Uses regular expressions for month matching and simple string checks for locations.
* Ensures proper location parsing even if months and locations are mixed in the input.

**NLPInputParser Class Documentation**

Overview:

The NLPInputParser class is responsible for processing input text using natural language processing (NLP) techniques. It uses OpenNLP tools to perform sentence detection, tokenization, part-of-speech (POS) tagging, and optional lemmatization. The primary goal is to parse trip details, such as locations and travel dates, from user input.

Dependencies:

* OpenNLP Library (tools for sentence detection, tokenization, POS tagging, and lemmatization).

Attributes:

* SentenceDetectorME sentenceDetector: Detects sentence boundaries in the input text.
* TokenizerME tokenizer: Tokenizes the text into individual words.
* POSTaggerME posTagger: Tags each token with its part-of-speech.
* DictionaryLemmatizer lemmatizer (optional): Lemmatizes tokens to their base forms.

Constructor:

* Initializes models for sentence detection, tokenization, POS tagging, and lemmatization (if available).
* Loads model files from the local file system for OpenNLP tools.

Methods:

1. parseTripDetails(String input):
   * Parameters: String input – The input text containing trip details.
   * Returns: TripDetails – A TripDetails object containing locations and the travel month parsed from the input.
   * Description:
     + Detects sentences in the input.
     + Tokenizes sentences into words.
     + Performs POS tagging on the tokens.
     + Identifies proper nouns (NNP/NNPS tags) as potential location names.
     + Identifies date-like tokens (e.g., "12/12/2025" or month names).
     + Collects and returns the locations and travel month as part of the TripDetails object.

Security Considerations:

* Ensure the model files are securely stored and not exposed to unauthorized access.
* Handle any exceptions that might occur during model loading or processing.

**PDFGenerator Class Documentation**

Overview:

The PDFGenerator class is responsible for creating a custom PDF document containing a checklist for travel preparation. It includes multiple sections such as trip details, essential items, recommendations, optional items, and accessories. The class uses the Apache PDFBox library to generate the PDF.

Dependencies:

* Apache PDFBox (used for PDF creation and manipulation)

Methods:

1. generateChecklist(String fileName, ConversationState state, String tripDetails, String[] essentialItems, String[] recommendations, String[] optionalItems, String[] accessories):
   * Parameters:
     + fileName: The name of the output PDF file.
     + state: The ConversationState object holding user details such as name.
     + tripDetails: A string representing trip details (e.g., locations).
     + essentialItems: An array of essential clothing items.
     + recommendations: An array of recommended items for the trip.
     + optionalItems: An array of optional items.
     + accessories: An array of accessory items for the trip.
   * Description:
     + Generates a custom PDF checklist with sections for the trip details, essential items, recommendations, optional items, and accessories.
     + The document is saved to the provided file path (fileName).
   * Returns: None.
2. addSectionHeader(PDPageContentStream contentStream, String header, float margin, float yPosition, PDType1Font font, int fontSize):
   * Parameters:
     + contentStream: The stream used to write content to the PDF page.
     + header: The title of the section (e.g., "Trip:", "Essential:", etc.).
     + margin: The left margin for positioning the header.
     + yPosition: The current y-coordinate for positioning.
     + font: The font to use for the header.
     + fontSize: The font size to use for the header.
   * Description:
     + Adds a section header to the PDF.
     + Adjusts the y-position for the next section.
   * Returns: The updated y-position after adding the header.
3. addBulletList(PDPageContentStream contentStream, String[] items, float xPosition, float yPosition, PDType1Font font, int fontSize, float leading):
   * Parameters:
     + contentStream: The stream used to write content to the PDF page.
     + items: An array of items to be listed as bullets.
     + xPosition: The x-coordinate for positioning the list.
     + yPosition: The current y-coordinate for positioning the list.
     + font: The font to use for the list items.
     + fontSize: The font size to use for the list items.
     + leading: The line height or spacing between list items.
   * Description:
     + Adds a bullet-point list to the PDF.
     + Adjusts the y-position after adding each item.
   * Returns: The updated y-position after adding the list.
4. addParagraph(PDPageContentStream contentStream, String text, float xPosition, float yPosition, PDType1Font font, int fontSize, float maxWidth, float leading):
   * Parameters:
     + contentStream: The stream used to write content to the PDF page.
     + text: The text to be added as a paragraph.
     + xPosition: The x-coordinate for positioning the paragraph.
     + yPosition: The current y-coordinate for positioning the paragraph.
     + font: The font to use for the text.
     + fontSize: The font size to use for the text.
     + maxWidth: The maximum width for the paragraph (used for text wrapping).
     + leading: The line height or spacing between lines of text.
   * Description:
     + Adds a paragraph to the PDF with automatic word wrapping.
     + Adjusts the y-position after adding the text.
   * Returns: The updated y-position after adding the paragraph.

**Example Usage:**

*String fileName = "TripChecklist.pdf";*

*ConversationState state = new ConversationState();*

*state.setUserName("Thales Campos");*

*String tripDetails = "Trip to Brazil, Italy, and Spain.";*

*String[] essentialItems = {"T-shirts", "Shorts", "Sunglasses"};*

*String[] recommendations = {"Comfortable Shoes", "Camera"};*

*String[] optionalItems = {"Swimwear", "Hat"};*

*String[] accessories = {"Backpack", "Travel Pillow"};*

*PDFGenerator.generateChecklist(fileName, state, tripDetails, essentialItems, recommendations, optionalItems, accessories);*

This will generate a PDF titled "TripChecklist.pdf" containing a personalized checklist for Thales Campos, detailing trip information and packing essentials.

**Security Considerations:**

* Ensure that any user-generated input (e.g., tripDetails, essentialItems) is sanitized to prevent injection attacks.
* The file generation process should be done in a secure location to avoid unauthorized access.

**TerminalChatbot Class Documentation**

Overview:

The TerminalChatbot class serves as the entry point for the chatbot application. It initializes the ConversationController and starts the interaction by invoking the run() method. This setup is typically used for a terminal-based interface where the chatbot engages with the user in a command-line environment.

Dependencies:

* ConversationController: The class responsible for controlling the flow of the conversation, processing user inputs, and managing the chatbot's state.

Methods:

1. main(String[] args):
   * Parameters:
     + args: Command-line arguments (if any).
   * Description:
     + This is the entry point of the application. It creates an instance of the ConversationController and starts the conversation by calling its run() method.
   * Returns: None.

Example Usage:

To run the chatbot, the user simply needs to execute the TerminalChatbot class in a terminal environment.

java main.java.com.tripper.TerminalChatbot

This command will start the chatbot, which will interact with the user by controlling the conversation flow through the ConversationController.

Flow:

1. Initialization:
   * The main() method initializes a ConversationController object.
2. Conversation Start:
   * The run() method of the ConversationController is invoked, starting the chatbot's interaction with the user.

This class does not handle direct user interaction but relies on the ConversationController to manage the logic and state of the conversation.

**TripChecklist Class Documentation**

Overview:

The TripChecklist class holds and organizes the various categories of items for a travel checklist. It categorizes the items into essential items, recommendations, optional items, and accessories, which are useful for generating a detailed packing list for the user.

Fields:

* essentialItems: An array of essential items needed for the trip (e.g., passport, tickets).
* recommendations: An array of recommended items (e.g., sunscreen, camera).
* optionalItems: An array of optional items that might be useful but are not strictly necessary (e.g., a book, extra shoes).
* accessories: An array of accessory items (e.g., hats, sunglasses, scarves).

Constructor:

* TripChecklist(String[] essentialItems, String[] recommendations, String[] optionalItems, String[] accessories)
  + Parameters:
    - essentialItems: An array containing items deemed essential for the trip.
    - recommendations: An array of recommended items for the trip.
    - optionalItems: An array of optional items that can be included for the trip.
    - accessories: An array of accessory items for the trip.
  + Description: This constructor initializes the four categories of items, providing a structured way to manage the checklist for the trip.

Getter Methods:

* getEssentialItems(): Returns the array of essential items.
* getRecommendations(): Returns the array of recommended items.
* getOptionalItems(): Returns the array of optional items.
* getAccessories(): Returns the array of accessory items.

Example Usage:

*// Example of how to create a TripChecklist and access the items*

*String[] essentials = {"Passport", "Flight tickets", "Travel Insurance"};*

*String[] recommendations = {"Camera", "Sunscreen", "Guidebook"};*

*String[] optional = {"Book", "Extra shoes"};*

*String[] accessories = {"Hat", "Sunglasses", "Scarf"};*

*TripChecklist checklist = new TripChecklist(essentials, recommendations, optional, accessories);*

*// Accessing the items*

*String[] essentialItems = checklist.getEssentialItems();*

*String[] recommendations = checklist.getRecommendations();*

*String[] optionalItems = checklist.getOptionalItems();*

*String[] accessories = checklist.getAccessories();*

Use Case:

This class is typically used to store and retrieve different categories of items that need to be packed for a trip. It can be passed to other components like PDFGenerator to create packing lists, or to be used within the ConversationController to provide personalized recommendations.

**TripChecklistGenerator Class Documentation**

Overview:

The TripChecklistGenerator class is responsible for generating a set of checklist items (essential items, recommendations, optional items, and accessories) based on the details of the trip, particularly the travel month. It uses simple heuristics to categorize items based on whether the trip is likely to be in summer, winter, or a neutral season.

Method:

* generateChecklist(TripDetails details)
  + Parameters:
    - details: An instance of TripDetails that contains information about the user's travel month.
  + Returns: A TripChecklist object containing categorized items (essential, recommendations, optional, and accessories).
  + Description: This method generates checklist arrays based on the travel month in the TripDetails object. It uses the travel month to determine which items should be included for the trip, following a set of heuristics:
    - Summer months (June, July, August) or mentions of "summer" lead to lighter clothing items.
    - Winter months (December, January, February) or mentions of "winter" lead to warmer clothing and accessories.
    - If the travel month is unclear, a neutral set of items is chosen.

Logic:

* Summer Travel (e.g., June, July, August):
  + Essential Items: Light T-shirt, Shorts, Comfortable walking shoes, Sunglasses.
  + Recommendations: Hat, Sunscreen, Umbrella (for potential rain).
  + Optional Items: Light Sweater, Extra pair of Socks.
  + Accessories: Crossbody Bag, Travel Adapter, Power Bank.
* Winter Travel (e.g., December, January, February):
  + Essential Items: Warm Jacket, Thermal Wear, Gloves, Scarf, Beanie.
  + Recommendations: Boots, Extra Socks.
  + Optional Items: Lip Balm, Hand Warmers.
  + Accessories: Backpack, Travel Adapter.
* Neutral Travel (if no specific month or season is clear):
  + Essential Items: Versatile T-shirt, Jeans, Comfortable Shoes.
  + Recommendations: Light Jacket, Umbrella.
  + Optional Items: Hat, Sunglasses.
  + Accessories: Backpack, Portable Charger.

Example Usage:

*java*

*// Example of how to generate a checklist based on travel month*

*TripDetails tripDetails = new TripDetails();*

*tripDetails.setTravelMonth("June");*

*TripChecklist checklist = TripChecklistGenerator.generateChecklist(tripDetails);*

*// Accessing checklist items*

*String[] essentialItems = checklist.getEssentialItems();*

*String[] recommendations = checklist.getRecommendations();*

*String[] optionalItems = checklist.getOptionalItems();*

*String[] accessories = checklist.getAccessories();*

Use Case:

This class is typically used in the context of a travel planning application or a chatbot. After determining the user's travel month, the TripChecklistGenerator can create a personalized packing list to help the user prepare for their trip. The generated checklist can be used in various formats (e.g., displayed on the user interface, saved as a PDF, etc.).

**TripDetails Class Documentation**

Overview:

The TripDetails class holds information about the user's trip. It includes a list of locations (places the user intends to visit) and the travel month (the month during which the user is planning their trip).

Fields:

1. locations (List<String>):
   * A list of locations that the user intends to visit. This could represent cities, countries, or specific landmarks.
2. travelMonth (String):
   * The month when the user is planning to travel, represented as a string (e.g., "June", "December"). This helps in generating a personalized packing list based on seasonal factors.

Methods:

* getLocations():
  + Description: Returns the list of locations for the trip.
  + Return Type: List<String>
* setLocations(List<String> locations):
  + Description: Sets the list of locations for the trip.
  + Parameters: A List<String> containing location names (e.g., "Paris", "Venice").
* getTravelMonth():
  + Description: Returns the travel month for the trip.
  + Return Type: String
* setTravelMonth(String travelMonth):
  + Description: Sets the travel month for the trip.
  + Parameters: A String representing the month of travel (e.g., "June").
* toString():
  + Description: Returns a string representation of the TripDetails object, which includes the list of locations and the travel month.
  + Return Type: String

Example Usage:

*java*

*// Creating a TripDetails instance*

*TripDetails tripDetails = new TripDetails();*

*// Setting the travel month and locations*

*tripDetails.setTravelMonth("June");*

*tripDetails.setLocations(List.of("Brazil", "Italy", "Spain"));*

*// Accessing the trip details*

*String travelMonth = tripDetails.getTravelMonth(); // "June"*

*List<String> locations = tripDetails.getLocations(); // ["Brazil", "Italy", "Spain"]*

*// Printing the trip details*

*System.out.println(tripDetails.toString());*

*// Output: TripDetails [locations=[Brazil, Italy, Spain], travelMonth=June]*

Use Case:

The TripDetails class is used to store essential trip information, such as the destinations the user is visiting and when they are traveling. This information is used by other classes, like TripChecklistGenerator, to create personalized checklists and planning resources for the user.

**WeatherService Class Documentation**

Overview:

The WeatherService class is responsible for fetching weather forecast data from the OpenWeatherMap API. The class uses HTTP requests to fetch weather data for a specific location and returns the data as a WeatherResponse object.

Fields:

* API\_KEY (String):
  + A constant holding the API key required to authenticate requests to the OpenWeatherMap API. *(Note: For security reasons, make sure to keep the API key secret.)*
* BASE\_URL (String):
  + A constant holding the base URL for the OpenWeatherMap API endpoint.

Method:

* getForecastData(String location):
  + Description: This method fetches the weather forecast for a specified location using the OpenWeatherMap API.
  + Parameters:
    - location (String): The name of the location for which the forecast is needed (e.g., "Paris").
  + Return Type: WeatherResponse
    - Returns a WeatherResponse object containing the parsed forecast data.
  + Throws:
    - Catches and prints exceptions related to network requests or data parsing.

Steps:

* + The method builds the URL to request the forecast data using the location and the API\_KEY.
  + Sends an HTTP GET request to OpenWeatherMap API.
  + If the request is successful (status code 200), it reads the response as a string.
  + The response string is parsed into a WeatherResponse object using the Gson library.
  + The method returns the WeatherResponse object containing the forecast data.
  + In case of any errors (e.g., failed request or invalid data), it prints an error message and returns null.

Example Usage:

*java*

*// Create a WeatherService instance*

*WeatherService weatherService = new WeatherService();*

*// Fetch weather forecast for a given location*

*WeatherResponse forecast = weatherService.getForecastData("Paris");*

*if (forecast != null) {*

*// Process the forecast data*

*System.out.println("Weather forecast: " + forecast);*

*} else {*

*System.out.println("Unable to retrieve weather data.");*

*}*

Notes:

1. The API\_KEY is stored as a private constant within the class. Be sure to keep it secret, and avoid sharing it in public repositories.
2. The WeatherResponse class is assumed to be a data model for parsing the JSON response from the OpenWeatherMap API. This class would typically contain fields for temperature, weather conditions, etc.
3. Make sure to handle any potential issues with rate limits or incorrect API keys when using this service.