|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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: | Project Final 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. | | | | | | |

A picture containing text, outdoor, building, white

Description automatically generated

Software Development 2

BSCH-SD2

Chatbot Project  
Final Documentation

**WEB APPLICATION**

**ChatGPTClient Class**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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**

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.

**MOBILE APPLICATION**

* **ChatGPTClient Class**

**Overview:**

The ChatGPTClient class is a Spring @Component that handles communication with the OpenAI ChatGPT API. It sends a user-provided conversation context to the API and receives a generated response from the AI model. The class uses **Spring's @Value annotation** to inject the API key and API URL from the application's configuration file (e.g., application.properties or application.yml).

**Dependencies:**

* **Spring Framework**:
  + @Component for defining a Spring-managed bean.
  + @Value for injecting property values.
* **Google Gson**:
  + Used for building and parsing JSON payloads.
* **Java Standard Libraries**:
  + HttpURLConnection for HTTP communications.
  + InputStreamReader, BufferedReader, and OutputStream for I/O operations.
  + URL and networking tools for setting up API requests.
  + StandardCharsets for character encoding.

**Attributes:**

* API\_KEY (String): Injected from the property ${openai.api.key}, used for authenticating with the OpenAI API.
* API\_URL (String): Injected from the property ${openai.api.url}, defines the endpoint to which requests are sent.

**Public Methods:**

* **String getChatResponse(String conversationContext)**
  + **Parameters**:
    - conversationContext: A string containing the user's input or context for the conversation.
  + **Returns**:
    - A String containing the response generated by the ChatGPT model.
    - Returns an error message if the request fails.
  + **Process Flow**:
    - Establishes an HTTP POST connection to the OpenAI API using HttpURLConnection.
    - Sets the necessary request headers:
      * Authorization (Bearer token with the API key)
      * Content-Type (application/json)
      * OpenAI-Project (specific project identifier)
    - Builds the request payload:
      * Defines the model as gpt-3.5-turbo.
      * Adds a system message that instructs the assistant to act as Tripper, a helpful travel clothing planner.
      * Adds the user-provided conversation context as a user message.
    - Sends the JSON payload to the API.
    - Reads the response stream and parses it using Gson.
    - Extracts the AI's reply from the first element of the "choices" array in the JSON response.
    - If successful, returns the AI's message; otherwise, returns a default error message.
  + **Error Handling**:
    - Catches any exceptions during the request or parsing process.
    - Returns a generic error message containing the exception details.
* **GoogleMapsService Class**

**Overview:**

The GoogleMapsService class is a Spring @Service that interacts with the **Google Maps Geocoding API**.  
Its main purpose is to validate whether a given place name corresponds to a real geographic location by sending requests to the Geocoding API. The API key is securely injected from the application's configuration using Spring's @Value annotation.

**Dependencies:**

* **Spring Framework**:
  + @Service to declare the class as a Spring-managed service bean.
  + @Value for property injection (Google API key).
* **Google Maps Java Client Library**:
  + GeoApiContext for building the API context.
  + GeocodingApi for accessing geocoding services.
  + GeocodingResult for handling API responses.

**Attributes:**

* context (GeoApiContext):
  + Holds the configuration for connecting to the Google Maps API, including the API key.

**Constructor:**

* **GoogleMapsService(@Value("${google.api.key}") String apiKey)**
  + **Parameters**:
    - apiKey: The Google Maps API key injected from the application configuration.
  + **Behavior**:
    - Initializes a GeoApiContext with the provided API key.
    - Prepares the service to make authorized requests to Google Maps APIs.

**Public Methods:**

* **boolean isValidLocation(String place)**
  + **Parameters**:
    - place: The name of the place (e.g., "Paris", "Times Square", "Mount Everest") to validate.
  + **Returns**:
    - true if the location is recognized by the Google Geocoding API.
    - false if the location is not found or an error occurs.
  + **Process Flow**:
    - Sends a request to the Google Geocoding API using the provided place.
    - If the API returns one or more results, the method considers the place valid.
    - If the API throws an exception (e.g., invalid input, connection issue), the method catches it and prints an error message.
    - Returns false if no results are found or an exception is raised.
* **WeatherService Class**

**Overview:**

The WeatherService class is a Spring @Service that fetches weather forecast data from the **OpenWeather API**. It uses **WebClient** for asynchronous HTTP requests and is configured with a base URL and an API key loaded from application properties (OpenWeatherProperties). It also leverages caching to optimize repeated lookups for the same location.

**Dependencies:**

* **Spring Framework**:
  + @Service for defining a service bean.
  + @Cacheable for caching responses automatically.
* **Lombok**:
  + @Slf4j to automatically create a logger.
* **WebClient** (Spring WebFlux):
  + For reactive, non-blocking HTTP client functionality.
* **Reactor**:
  + Retry to apply automatic retry policies for failed API calls.

**Attributes:**

* webClient (WebClient):
  + The configured WebClient instance used to perform HTTP requests to the OpenWeather API.
* props (OpenWeatherProperties):
  + Contains configuration values such as the API URL and API key.

**Constructor:**

* **WeatherService(WebClient.Builder webClientBuilder, OpenWeatherProperties props)**
  + **Parameters**:
    - webClientBuilder: A WebClient builder to create a configured WebClient.
    - props: An instance containing the OpenWeather API configuration (URL and Key).
  + **Behavior**:
    - Initializes the webClient with the base URL from properties.

**Public Methods:**

* **Optional<WeatherResponse> getForecastData(String location)**
  + **Parameters**:
    - location: A String representing the location for which to retrieve the weather forecast.
  + **Returns**:
    - An Optional<WeatherResponse> object containing the weather forecast if found.
    - An empty Optional if an error occurs or no data is returned.
  + **Annotations**:
    - @Cacheable(value = "weather", key = "#location.toLowerCase()", unless = "#result.isEmpty()")
      * Caches the forecast for a location (lowercased) unless the result is empty.
  + **Process Flow**:
    - Builds a GET request to the OpenWeather API with query parameters:
      * q: the requested location.
      * appid: the API key.
      * units: set to "metric" (for Celsius).
    - Sends the request and maps the response body to a WeatherResponse object.
    - If the API call fails, retries the request up to 2 times with exponential backoff.
    - Catches specific HTTP response exceptions and logs a warning if any.
    - Catches general exceptions and logs an error if an unexpected issue occurs.
    - Returns the forecast wrapped in an Optional.

**Error Handling:**

* If the OpenWeather API responds with an error status, it logs a **warning** with the status and the message.
* If a general error occurs (e.g., network issue, parsing error), it logs an **error** message.
* **OpenWeatherProperties Class**

**Overview:**

The OpenWeatherProperties class is a Spring-managed configuration class used to bind and store external properties related to the OpenWeather API. It holds the **API key** and the **base URL** required to perform API calls. By using Spring Boot’s @ConfigurationProperties, the fields in this class are automatically populated from configuration files (e.g., application.properties or application.yml).

**Dependencies:**

* **Spring Framework**:
  + @Component to register the class as a Spring bean.
  + @ConfigurationProperties for automatic property binding.
* **Lombok**:
  + @Getter to auto-generate getter methods for all fields.
  + @Setter to auto-generate setter methods for all fields.

**Annotations Explained:**

* @Setter: Lombok annotation that generates all setter methods for the fields.
* @Getter: Lombok annotation that generates all getter methods for the fields.
* @Component: Marks the class as a Spring-managed bean.
* @ConfigurationProperties(prefix = "weather.api"): Binds all properties prefixed with weather.api (e.g., weather.api.key, weather.api.url) from the configuration file to this class.

**Attributes:**

* **key** (String):
  + Stores the API key required to authenticate requests to the OpenWeather API.
* **url** (String):
  + Stores the base URL endpoint for the OpenWeather API.
* **WebConfig Class**

**Overview:**

The WebConfig class is a **Spring configuration class** that customizes the behavior of the Spring MVC framework. In this case, it specifically configures **CORS (Cross-Origin Resource Sharing)** settings, allowing the frontend application to communicate with the backend server. It implements the WebMvcConfigurer interface, which provides callback methods for customizing Spring MVC configuration.

**Dependencies:**

* **Spring Framework**:
  + @Configuration to mark the class as a source of bean definitions.
  + WebMvcConfigurer to customize Spring MVC settings.
  + CorsRegistry for CORS mappings.

**Annotations Explained:**

* @Configuration: Marks the class as a configuration class that Spring will pick up and process during application startup.

**Implemented Interface:**

* **WebMvcConfigurer**:
  + Interface provided by Spring MVC to allow configuration of web-related settings, such as CORS mappings, interceptors, formatters, etc.

**Overridden Methods:**

* **addCorsMappings(CorsRegistry registry)**
  + **Parameters**:
    - registry: A CorsRegistry object used to configure allowed origins, methods, headers, and other CORS settings.
  + **Behavior**:
    - Configures CORS settings globally for all endpoints (/\*\*).
    - Allows requests only from the origin https://tripper-frontend.vercel.app.
    - Permits the following HTTP methods: "GET", "POST", "PUT", "DELETE", "PATCH", and "OPTIONS".
    - Allows all headers ("\*").
    - Enables sending credentials (such as cookies or authorization headers) in cross-origin requests (allowCredentials(true)).
* **WebSocketConfig Class**

**Overview:**

The WebSocketConfig class is a **Spring configuration class** that sets up WebSocket communication in the application. It enables STOMP (Simple Text Oriented Messaging Protocol) over WebSocket and configures endpoints and message brokers for real-time messaging between clients and the server. The class implements WebSocketMessageBrokerConfigurer, allowing full customization of WebSocket behavior.

**Dependencies:**

* **Spring Framework**:
  + @Configuration to declare the class as a Spring-managed configuration bean.
  + @EnableWebSocketMessageBroker to enable WebSocket message handling backed by a message broker.
* **WebSocket and STOMP Support**:
  + WebSocketMessageBrokerConfigurer to configure endpoints and brokers.
  + StompEndpointRegistry for endpoint registration.
  + MessageBrokerRegistry for broker configuration.

**Annotations Explained:**

* @Configuration: Indicates that this class provides Spring configuration.
* @EnableWebSocketMessageBroker: Enables support for handling WebSocket messages using a message broker (STOMP protocol).

**Implemented Interface:**

* **WebSocketMessageBrokerConfigurer**:
  + Provides callback methods to customize WebSocket message handling.

**Overridden Methods:**

* **registerStompEndpoints(StompEndpointRegistry registry)**
  + **Parameters**:
    - registry: The StompEndpointRegistry used to register WebSocket endpoints.
  + **Behavior**:
    - Registers an endpoint at /ws-chat that clients can connect to for WebSocket communication.
    - Allows connections from any origin (setAllowedOriginPatterns("\*")) to facilitate flexibility across different domains.
    - Enables **SockJS** fallback, ensuring clients that do not support WebSocket can still connect using alternative transports.
* **configureMessageBroker(MessageBrokerRegistry registry)**
  + **Parameters**:
    - registry: The MessageBrokerRegistry used to configure the message broker.
  + **Behavior**:
    - Enables a **simple in-memory broker** that routes messages to destinations prefixed with /topic.
    - Sets /app as the **application destination prefix** for messages that are routed to message-handling methods on the server side.
* **ChatController Class**

**Overview:**

The ChatController class is a **REST controller** responsible for managing conversation flows between users and the chatbot (TripChatService). It exposes HTTP endpoints for starting conversations, sending messages, retrieving conversation histories, updating conversation titles, deleting conversations, and exporting conversations as PDFs. This controller acts as a bridge between the frontend and the service layer.

**Dependencies:**

* **Spring Framework**:
  + @RestController for defining RESTful web endpoints.
  + @RequestMapping, @PostMapping, @GetMapping, @PatchMapping, @DeleteMapping for route mappings.
  + ResponseEntity for flexible HTTP response handling.
* **Lombok**:
  + @RequiredArgsConstructor to automatically generate a constructor for final fields.
* **Project classes**:
  + ConversationService for managing conversation state and persistence.
  + TripChatService for interacting with ChatGPT to generate replies.
  + Message, Conversation, and MessageView for representing chat data.

**Annotations Explained:**

* @RestController: Marks this class as a REST controller, automatically serializing return values to JSON.
* @RequestMapping("/chat"): Maps all routes inside the controller under /chat.
* @RequiredArgsConstructor: Lombok annotation that generates a constructor injecting required dependencies (final fields).

**Fields:**

* **DEFAULT\_USER\_ID** (String): Default user identifier ("anonymous") used when no user ID is provided.
* **conversationService** (ConversationService): Handles conversation logic, message storage, and retrieval.
* **tripChatService** (TripChatService): Interacts with ChatGPT to generate automated assistant responses.

**Public Endpoints:**

* **POST /chat/start**  
  Start a new conversation.
  + **Input**: JSON body with optional userId.
  + **Output**: JSON containing the conversationId.
* **POST /chat/{conversationId}/message**

Send a user message and receive the assistant's reply.

* + **Input**: URL path parameter conversationId and JSON body with userId and message.
  + **Output**: Updated list of messages (List<MessageView>) in the conversation.
* **GET /chat/{conversationId}/messages**

Retrieve the entire message history for a conversation.

* + **Input**: conversationId (path).
  + **Output**: List of messages (List<MessageView>).
* **GET /chat/user/{userId}**

Retrieve all conversations for a specific user.

* + **Input**: userId (path).
  + **Output**: List of conversations (List<Conversation>).
* **GET /chat/history**

Retrieve all messages for a user and conversation ID.

* + **Input**: Query parameters userId and conversationId.
  + **Output**: List of messages (List<Message>).
* **PATCH /chat/{conversationId}/title**

Update the title of a conversation.

* + **Input**: conversationId (path) and JSON body with title.
  + **Output**: HTTP 200 OK if successful.
* **DELETE /chat/{conversationId}**

Delete a conversation.

* + **Input**: conversationId (path).
  + **Output**: HTTP 204 No Content if successful.
* **GET /chat/{conversationId}/export/pdf**

Export a conversation as a PDF file.

* + **Input**: conversationId (path).
  + **Output**: A downloadable PDF file containing the conversation transcript.
  + **Error Handling**: Returns HTTP 500 Internal Server Error if export fails.

**ChatSocketController Class**

**Overview:**

This class is a Spring MVC controller that handles WebSocket communication for a chat feature, facilitating real-time message exchanges between a user and an assistant (powered by GPT).

**Annotations:**

* @Controller: This annotation marks the class as a Spring MVC controller, making it eligible to handle web requests.
* @RequiredArgsConstructor: This Lombok annotation generates a constructor with required arguments (for all final fields), reducing boilerplate code.

**Dependencies (Injected via Constructor):**

* ConversationService: Handles operations related to conversations, such as adding messages and retrieving message history.
* TripChatService: Facilitates interaction with GPT, generating assistant replies based on conversation history.
* SimpMessagingTemplate: A Spring component that sends messages over WebSocket.
* MessageMapper: A utility to convert between domain objects (like MessageView) and Data Transfer Objects (DTOs) like OutgoingMessageDTO.

**Method: handleChat(@Payload MessageDTO incomingMessage)**

* **Annotation: @MessageMapping("/chat.send")**
  + Maps incoming WebSocket messages (from the /chat.send destination) to this method. The method is triggered whenever a new message is sent from the client-side chat interface.
* **Parameters:**
  + @Payload MessageDTO incomingMessage: The incoming message is automatically deserialized into a MessageDTO object, which contains data sent by the user, such as the conversation ID, user ID, and message content.
* **Method Flow:**
  + **Extract Data from the Incoming Message:**
    - The conversation ID, user ID, and the content of the user's message are extracted from the incomingMessage.
  + **Save User's Message:**
    - The conversationService.addMessage method is called to save the user's message to the database, associating it with the conversation ID and marking it as a message from the user.
  + **Generate Assistant's Reply:**
    - The tripChatService.chatWithGPT method generates a response from GPT based on the ongoing conversation (using the conversation ID).
  + **Save Assistant's Reply:**
    - The assistant's reply (generated by GPT) is saved to the conversation using the conversationService.addMessage method.
  + **Fetch Updated Conversation History:**
    - The complete conversation history for the conversation ID is fetched using the conversationService.getConversationMessages method. This ensures that both the user's message and the assistant's reply are included in the conversation history.
  + **Convert to DTO Format:**
    - The list of messages (MessageView objects) is mapped to OutgoingMessageDTO objects using the messageMapper.toDtoList method. This conversion is necessary because WebSocket messages need to be in a specific DTO format for proper communication with the frontend.
  + **Send Updated Conversation to Client:**
    - The messagingTemplate.convertAndSend method sends the list of converted messages to the client-side chat interface over the WebSocket channel. The messages are sent to the /topic/chat/{conversationId} destination, allowing the frontend to receive the updated conversation.

**TripPlannerController Class**

**Overview:**

This class is a Spring MVC REST controller that provides an endpoint for trip planning. It allows the client to send a trip planning request and receive a response containing the details of the planned trip.

**Annotations:**

* @RestController: This annotation marks the class as a Spring MVC controller that returns data (rather than views), specifically in JSON or XML format, as part of a REST API.
* @RequestMapping("/trip-planner"): This annotation defines the base URL path (/trip-planner) for all the endpoints in this controller.
* @RequiredArgsConstructor: This Lombok annotation generates a constructor with required arguments (for all final fields), reducing boilerplate code.

**Dependencies (Injected via Constructor):**

* TripPlannerService: This service is responsible for processing the trip request and generating the corresponding response.

**Method: handleTripPlanning(@RequestBody TripRequest request)**

* **Annotation: @PostMapping**
  + This annotation indicates that the method handles HTTP POST requests. It maps to the URL /trip-planner and is used to submit the trip planning request.
* **Parameters:**
  + @RequestBody TripRequest request: The method takes a TripRequest object, which is automatically populated from the JSON payload in the HTTP request body.
* **Method Flow:**
  + **Process Trip Request:**
    - The tripPlannerService.processTripRequest(request) method is called with the provided TripRequest. This method processes the request and generates a TripResponse based on the request details.
  + **Return the Response:**
    - A ResponseEntity containing the TripResponse is returned with an HTTP status code of 200 (OK), indicating the request was successfully processed.

**Flow Summary:**

1. The client sends a POST request to the /trip-planner endpoint with a TripRequest in the request body.
2. The handleTripPlanning method processes the request using the TripPlannerService.
3. A TripResponse is generated and returned to the client as the response body.

**Key Concepts:**

* **REST API Endpoint:** The controller provides a RESTful endpoint that processes trip planning requests.
* **Request and Response Body:** The TripRequest is sent in the request body, and the TripResponse is returned in the response body.
* **Service Layer:** The TripPlannerService processes the business logic related to trip planning.

**MessageDTO Class**

**Overview:**

This class is a **Data Transfer Object (DTO)** that represents an incoming message. It is used to transfer messages from the client to the server in a structured format, containing details about the conversation, user, and the message content.

**Annotations:**

* @param: Describes each field in the class.
  + conversationId: The ID of the conversation that this message belongs to.
  + userId: The ID of the user who sent the message.
  + content: The content of the message that the user is sending.

**Fields:**

* Long conversationId: The unique identifier for the conversation. This field helps in associating the message with a specific conversation.
* String userId: The identifier for the user sending the message.
* String content: The actual message content that the user is sending.

**Purpose:**

This MessageDTO record is used to encapsulate the details of an incoming message, which is sent from the client (e.g., a chat interface) to the server. The record ensures that the necessary details (conversation ID, user ID, and message content) are passed together in a well-defined structure.

**Key Concepts:**

* **Record Type:** The class is a Java record, which is a special kind of class introduced in Java 14 that is meant for immutable data objects. It automatically generates constructors, toString(), equals(), and hashCode() methods.
* **DTO (Data Transfer Object):** The purpose of this class is to transfer data, particularly for communication between the frontend (client) and backend (server).
* **Message Content:** The content field represents the actual text or data the user is sending in a message.

**OutgoingMessageDTO Class**

**Overview:**

This class is a **Data Transfer Object (DTO)** that represents an outgoing message. It is used to transfer messages from the server to the client, containing details such as the sender, message content, and timestamp.

**Annotations:**

* @param: Describes each field in the class.
  + sender: The sender of the message (e.g., user or assistant).
  + content: The content of the message that is being sent.
  + timestamp: The timestamp when the message was sent.

**Fields:**

* String sender: The sender of the message (could be a user or an assistant).
* String content: The content of the message being sent to the client.
* String timestamp: The timestamp when the message was sent, in string format.

**Methods:**

* **from(Message message)**: A static method that converts a Message object (likely a domain model object) into an OutgoingMessageDTO. This method extracts the sender, content, and timestamp from the Message and returns a new OutgoingMessageDTO.
  + **Parameters:**
    - Message message: The source Message object containing the data to be transferred.
  + **Return Value:**
    - A new instance of OutgoingMessageDTO with data extracted from the Message object.

**Purpose:**

The OutgoingMessageDTO is used to send message data from the server to the client. It encapsulates the sender's identity, the message content, and the timestamp, ensuring that the client receives the necessary information for displaying the message in the chat interface.

**Key Concepts:**

* **Record Type:** The class is a Java record, a type that simplifies the creation of immutable data classes. The record automatically generates toString(), equals(), hashCode(), and other utility methods.
* **DTO (Data Transfer Object):** The class is designed to transport data between the backend (server) and the frontend (client).
* **Message Conversion:** The static method from() converts a domain model object (Message) into a DTO, making it suitable for transmission over the network (e.g., via WebSocket or HTTP).