GRIFFITH COLLEGE DUBLIN

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	✓	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.

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:

5. The method builds the URL to request the forecast data using the location and the API_KEY.
6. Sends an HTTP GET request to OpenWeatherMap API.
7. If the request is successful (status code 200), it reads the response as a string.
8. The response string is parsed into a WeatherResponse object using the Gson library.
9. The method returns the WeatherResponse object containing the forecast data.
10. 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**:

1. Establishes an HTTP POST connection to the OpenAI API using HttpURLConnection.
2. Sets the necessary request headers:
    - Authorization (Bearer token with the API key)
    - Content-Type (application/json)
    - OpenAI-Project (specific project identifier)
3. 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.
4. Sends the JSON payload to the API.
5. Reads the response stream and parses it using Gson.
6. Extracts the AI's reply from the first element of the "choices" array in the JSON response.
7. 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**:

1. Sends a request to the Google Geocoding API using the provided place.
2. If the API returns one or more results, the method considers the place valid.
3. If the API throws an exception (e.g., invalid input, connection issue), the method catches it and prints an error message.
4. 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**:

1. 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).
2. Sends the request and maps the response body to a WeatherResponse object.
3. If the API call fails, retries the request up to 2 times with exponential backoff.
4. Catches specific HTTP response exceptions and logs a warning if any.
5. Catches general exceptions and logs an error if an unexpected issue occurs.
6. 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:**

1. **Extract Data from the Incoming Message:**
    - The conversation ID, user ID, and the content of the user's message are extracted from the incomingMessage.
2. **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.
3. **Generate Assistant's Reply:**

    - The tripChatService.chatWithGPT method generates a response from GPT based on the ongoing conversation (using the conversation ID).

4. **Save Assistant's Reply:**
    - The assistant's reply (generated by GPT) is saved to the conversation using the conversationService.addMessage method.
5. **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.
6. **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.
7. **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:**

1. **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.
2. **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).

TripInfo Class

Overview:

This class is a **Data Transfer Object (DTO)** that holds information about a trip. It contains two lists: one for the locations associated with the trip and another for the dates related to those locations. This record is used to represent the structured details of a trip.

Annotations:

- @param: Describes each field in the class.
    - locations: A list of strings representing the locations of the trip (e.g., cities, landmarks, or points of interest).
    - dates: A list of strings representing the dates associated with the trip (e.g., dates when the traveler will visit each location).

Fields:

- List<String> locations: A list of location names that are part of the trip. These locations could represent places like cities, tourist attractions, or any other points of interest the traveler plans to visit.

- List<String> dates: A list of dates corresponding to the trip. Each date represents when the traveler is planning to visit a specific location, and it must align with the entries in the locations list.

Purpose:

The TripInfo record is designed to encapsulate the key details about a trip, such as where the traveler is going and when they are going to each location. This record allows easy representation and transmission of trip-related information, making it suitable for various trip planning scenarios.

Key Concepts:

- **Record Type:** This is a Java record, which simplifies the creation of immutable data structures. The record automatically generates constructor, toString(), equals(), and hashCode() methods.
- **DTO (Data Transfer Object):** The class is used to transfer trip-related data, typically between different layers of an application (e.g., from a backend service to the frontend).
- **List Representation:** The use of two lists (one for locations and another for dates) ensures that the trip's details are structured in a flexible and ordered way.

TripResponse Class

Overview:

This class is a **Data Transfer Object (DTO)** that represents the response for a trip planning request. It contains details such as the user's name, recommendations for the trip, a list of checklist sections, and the PDF file name associated with the trip response.

Annotations:

- @JsonInclude(JsonInclude.Include.NON_NULL): This Jackson annotation ensures that null values are not included in the serialized JSON representation of the object. If any field in the TripResponse is null, it will be excluded from the JSON output.

Fields:

- String userName: The name of the user for whom the trip response is being generated. This field is optional and may be excluded if it's null.
- String recommendations: A string containing the trip recommendations. This could include suggestions for destinations, activities, or general tips for the trip. This field is also optional and may be excluded if it's null.
- List<TripChecklistSection> sections: A list of TripChecklistSection objects, representing the sections of the checklist related to the trip. Each section may contain different categories or tasks to be completed for the trip.
- String pdfFileName: The name of the PDF file that will be generated as part of the trip response, typically containing all trip details in a downloadable format.

Purpose:

The TripResponse record is used to encapsulate all relevant data generated as a response to a trip planning request. It allows for structured data transfer back to the client, including personalized recommendations, a checklist, and a PDF file name for download.

Key Concepts:

- **DTO (Data Transfer Object):** The class serves as a container for the data related to the trip response. It is typically used to transfer data between different parts of an application, such as the backend service and the frontend client.
- **JSON Serialization:** The @JsonInclude annotation ensures that only non-null fields are included in the JSON response, reducing unnecessary data transfer.
- **Checklist Sections:** The sections field represents the trip checklist, with each section potentially containing multiple tasks or categories to be completed.
- **PDF File:** The pdfFileName field represents the name of a PDF file that provides detailed trip information in a downloadable format for the user.

MessageMapper Interface

Overview:

This interface is responsible for converting Message entities (specifically MessageView objects) into OutgoingMessageDTO objects. It uses **MapStruct**, a code generator for bean mappings, to generate the implementation at compile time.

Annotations:

- @Mapper(componentModel = "spring"):
  - This annotation marks the interface as a MapStruct mapper. It tells MapStruct to generate the implementation of the interface and sets the componentModel to "spring", which allows Spring to manage the mapper as a Spring bean and inject it into other components.

Fields:

- MessageMapper INSTANCE:
  - This is a singleton instance of the MessageMapper interface, generated by MapStruct. It can be used directly for mapping operations, although in Spring applications, dependency injection is preferred.

Methods:

- **toDto(MessageView message)**:
  - This method converts a single MessageView entity into an OutgoingMessageDTO.
  - **Parameters:**

        - MessageView message: The MessageView entity that needs to be converted to a DTO.
    - **Return Type:**
        - OutgoingMessageDTO: A DTO that represents the message in a format suitable for sending to the client.
- **toDtoList(List<MessageView> messages)**:
    - This method converts a list of MessageView entities into a list of OutgoingMessageDTOs.
    - **Parameters:**
        - List<MessageView> messages: A list of MessageView entities to be converted.
    - **Return Type:**
        - List<OutgoingMessageDTO>: A list of DTOs representing the messages, which can then be serialized and sent to the client.

Purpose:

The MessageMapper interface is part of the data transformation layer of the application. It converts MessageView entities from the persistence layer into OutgoingMessageDTO objects, which are suitable for transmission via WebSocket or other protocols. This ensures that the data sent to the client is structured according to the expected format.

Key Concepts:

- **MapStruct:** MapStruct is a code generator for bean mappings. It provides a way to generate implementations for mapping methods automatically at compile time, which improves performance and reduces boilerplate code.
- **DTO (Data Transfer Object):** The interface is used to convert between entities and DTOs. This helps in separating the domain logic from the data representation needed by the client.
- **Spring Integration:** The componentModel = "spring" configuration ensures that the generated mapper is managed by the Spring container, enabling it to be injected where needed.

Conversation Class

Overview:

This class represents a **Conversation** entity in the application. It is mapped to a database table using **JPA** annotations and holds information about a conversation, including the user ID, timestamps, and associated messages.

Annotations:

- @Entity: Marks this class as a JPA entity, which means it will be mapped to a database table.
- @Getter: Lombok annotation that automatically generates getters for all fields.

- @Setter: Lombok annotation that automatically generates setters for all fields (except for fields that are marked as final).
- @Id: Specifies the primary key of the entity.
- @GeneratedValue(strategy = GenerationType.IDENTITY): Specifies that the ID field should be automatically generated by the database (using auto-increment or a similar strategy).
- @OneToMany(mappedBy = "conversation", cascade = CascadeType.ALL): Specifies a one-to-many relationship between Conversation and Message. The mappedBy attribute indicates that the Message entity is the owning side of the relationship. CascadeType.ALL means that operations like persist, merge, and remove on the Conversation entity will be cascaded to the associated Message entities.
- @JsonIgnore: Prevents the messages field from being serialized to JSON. This ensures that messages are not included in the API response, possibly to avoid circular references.

Fields:

- Long id: The unique identifier for the conversation, automatically generated by the database.
- String userId: The ID of the user involved in the conversation.
- LocalDateTime createdAt: The timestamp when the conversation was created. This field is automatically set to the current time when the entity is instantiated and cannot be changed afterward.
- List<Message> messages: A list of Message entities associated with this conversation. The messages are fetched through a one-to-many relationship.
- LocalDateTime startedAt: The timestamp when the conversation actually started. This field is set to the current time by default but can be modified later if necessary.
- String title: The title of the conversation, which may describe the purpose or context of the conversation.

Purpose:

The Conversation entity stores the core details of a conversation, including the user involved, the timestamp when the conversation was created, and the list of associated messages. It is a central part of the chat or messaging system, providing the structure needed to store conversations and their metadata in the database.

Message Class

Overview:

This class represents a **Message** entity in the application. It is part of a **Conversation** and holds information about an individual message, including the sender, content, and timestamp.

Annotations:

- @Entity: Marks this class as a JPA entity, meaning it will be mapped to a database table.
- @Getter: Lombok annotation that automatically generates getters for all fields.
- @Setter: Lombok annotation that automatically generates setters for all fields.
- @Id: Specifies the primary key for the entity.

- @GeneratedValue(strategy = GenerationType.IDENTITY): Indicates that the id field will be automatically generated by the database using an auto-increment strategy.
- @Column(columnDefinition = "TEXT"): Specifies that the content field should be stored as a TEXT column in the database, allowing for longer message content.
- @ManyToOne(fetch = FetchType.LAZY): Defines a many-to-one relationship between Message and Conversation, where many messages can belong to a single conversation. The fetch = FetchType.LAZY setting ensures that the conversation is fetched only when explicitly requested, optimizing performance.
- @JoinColumn(name = "conversation_id", nullable = false): Specifies the column that will store the foreign key reference to the Conversation entity.
- @JsonBackReference: Prevents the conversation field from being serialized when the Message object is converted to JSON. This helps avoid circular references when serializing entities with bidirectional relationships (e.g., Message → Conversation → Messages).

Fields:

- Long id: The unique identifier for the message, automatically generated by the database.
- String userId: The ID of the user who sent the message.
- String sender: The sender of the message, which can be a user or an assistant (e.g., "user" or "assistant").
- String content: The content of the message, which is stored as a TEXT field in the database. This field contains the actual message text.
- LocalDateTime timestamp: The timestamp of when the message was created. This field is automatically set to the current time when the Message is created.
- Conversation conversation: The Conversation entity that this message belongs to. This establishes a relationship between a message and its parent conversation.

Purpose:

The Message entity stores the details of a message within a conversation. It includes the user ID, the sender's identity, the message content, the timestamp, and a reference to the conversation to which it belongs. This structure allows messages to be stored in the database and associated with a specific conversation.

TripChecklist Class

Overview:

This class represents a checklist for a trip, categorizing items into **essential**, **recommended**, **optional**, and **accessory** categories. It is used to organize the packing and preparation items for a trip.

Fields:

- String[] essentialItems: An array of essential items for the trip. These are items that are necessary for the trip and should not be forgotten.

- String[] recommendations: An array of recommended items for the trip. These are items that may enhance the trip experience but are not strictly necessary.
- String[] optionalItems: An array of optional items for the trip. These are items that are nice to have but are not critical for the trip.
- String[] accessories: An array of accessory items for the trip. These are usually smaller or supplementary items that can add convenience or style.

Constructor:

- **TripChecklist(String[] essentialItems, String[] recommendations, String[] optionalItems, String[] accessories)**:
    - This constructor initializes a new TripChecklist object with the provided arrays for essential items, recommendations, optional items, and accessories.
    - **Parameters:**
        - String[] essentialItems: The array of essential items for the trip.
        - String[] recommendations: The array of recommended items for the trip.
        - String[] optionalItems: The array of optional items for the trip.
        - String[] accessories: The array of accessory items for the trip.

Methods:

- **getEssentialItems()**:
    - Returns the array of essential items for the trip.
- **getRecommendations()**:
    - Returns the array of recommended items for the trip.
- **getOptionalItems()**:
    - Returns the array of optional items for the trip.
- **getAccessories()**:
    - Returns the array of accessories for the trip.

Purpose:

The TripChecklist class is used to organize and categorize items that should be considered for a trip. This could be useful in the context of planning or packing for a journey, ensuring that the traveler does not forget essential items while also considering optional or recommended items.

TripChecklistSection Class

Overview:

This class represents a section of the trip checklist for a specific city. It includes details such as the city's name, the expected weather, and a list of items to pack, including clothing, accessories, and optional items.

Fields:

- String cityName: The name of the city for which the checklist section is being created.
- String weather: The expected weather for the city, which could help determine what items are needed (e.g., hot, cold, rainy).
- List<String> clothing: A list of clothing items recommended for the trip to the specific city based on its weather conditions.
- List<String> accessories: A list of accessories that might be useful or recommended for the trip to the city (e.g., hats, sunglasses).
- List<String> optionalItems: A list of optional items that the traveler might consider bringing, depending on personal preference or specific activities.

Constructors:

- **TripChecklistSection()**:
  - Default constructor for creating an empty TripChecklistSection object.
- **TripChecklistSection(String cityName, String weather, List<String> clothing, List<String> accessories, List<String> optionalItems)**:
  - This constructor initializes the TripChecklistSection with the city name, weather, clothing, accessories, and optional items.
  - **Parameters:**
    - String cityName: The name of the city.
    - String weather: The expected weather in the city.
    - List<String> clothing: The list of clothing items.
    - List<String> accessories: The list of accessories.
    - List<String> optionalItems: The list of optional items.

Getters and Setters:

- **getCityName()**: Returns the city name.
- **setCityName(String cityName)**: Sets the city name.
- **getWeather()**: Returns the weather description for the city.
- **setWeather(String weather)**: Sets the weather description for the city.
- **getClothing()**: Returns the list of clothing items for the trip.
- **setClothing(List<String> clothing)**: Sets the list of clothing items.
- **getAccessories()**: Returns the list of accessories for the trip.
- **setAccessories(List<String> accessories)**: Sets the list of accessories.
- **getOptionalItems()**: Returns the list of optional items for the trip.
- **setOptionalItems(List<String> optionalItems)**: Sets the list of optional items.

Purpose:

The TripChecklistSection class is designed to hold the packing checklist for a specific city in a trip. By organizing the checklist by city, the traveler can prepare more effectively for different weather conditions and activities that may vary between locations.

TripDetails Class

Overview:

This class represents the details of a trip, including the list of locations the traveler will visit and the month of travel. It is used to store and manage the essential information regarding a trip's itinerary.

Fields:

- List<String> locations: A list of strings representing the locations (cities, countries, or places) included in the trip. This allows for flexible and dynamic listing of multiple destinations.
- String travelMonth: A string representing the month of travel. This could be used to manage trip-related scheduling or packing based on seasonal considerations.

Methods:

- **getLocations()**:
  - Returns the list of locations associated with the trip.
- **setLocations(List<String> locations)**:
  - Sets the list of locations for the trip.
- **getTravelMonth()**:
  - Returns the month of travel.
- **setTravelMonth(String travelMonth)**:
  - Sets the month of travel for the trip.
- **toString()**:
  - Overrides the toString() method to provide a string representation of the TripDetails object, including the list of locations and the travel month. This method is useful for debugging or displaying the object in a human-readable format.

Purpose:

The TripDetails class is designed to hold and manage the basic details of a trip, such as the travel destinations and the month when the trip is scheduled. This could be used in a trip planning application to help organize and track the details of a planned trip.

TripRequest Class

Overview:

This class represents a request for trip planning. It holds the necessary information about the user's trip and determines whether a PDF should be generated as part of the response. It is typically used to send trip-related data from the client to the server.

Fields:

- String userName: The name of the user making the trip request.
- String tripDetails: A string containing the details of the trip (such as a brief description or objectives for the trip).
- String location: The location associated with the trip, such as the city or country (e.g., "Paris") for gathering relevant weather information or planning purposes.
- boolean generatePdf: A flag indicating whether a PDF file should be generated as part of the trip response. If true, a PDF with the trip details might be created.

Methods:

- **getUserName()**:
  - Returns the name of the user making the trip request.
- **setUserName(String userName)**:
  - Sets the name of the user making the trip request.
- **getTripDetails()**:
  - Returns the details of the trip.
- **setTripDetails(String tripDetails)**:
  - Sets the details of the trip.
- **getLocation()**:
  - Returns the location associated with the trip.
- **setLocation(String location)**:
  - Sets the location for the trip.
- **isGeneratePdf()**:
  - Returns a boolean indicating whether a PDF should be generated.
- **setGeneratePdf(boolean generatePdf)**:
  - Sets whether a PDF should be generated as part of the trip response.

Purpose:

The TripRequest class is used to encapsulate the details of a user's trip planning request, including the user's name, the trip description, the location, and whether a PDF should be generated. This data structure is typically used in a REST API to pass trip-related information from the client to the server for processing.

WeatherResponse Class

Overview:

This class represents the response from the OpenWeatherMap API, providing detailed weather data for a specific city. It contains various nested classes to represent specific weather-related information, including forecasts, temperature data, wind, and other weather parameters.

Fields:

- String cod: The status code returned by the OpenWeatherMap API (e.g., "200" for a successful response).
- int cnt: The number of items returned in the forecast list.
- List<Forecast> list: A list containing weather forecasts for a certain period.
- City city: A City object containing details about the city for which the weather data is retrieved.

Forecast (Nested Static Class)

This class represents a single weather forecast entry within the weather response. It contains data about the forecast's timestamp, weather conditions, and relevant weather details like temperature, wind speed, etc.

Fields:

- long dt: The timestamp of the forecast.
- MainData main: An instance of MainData containing the temperature and pressure details.
- List<WeatherInfo> weather: A list of WeatherInfo objects representing different weather conditions (like rain, snow, etc.).
- Clouds clouds: An instance of the Clouds class, representing cloudiness details.
- Wind wind: An instance of the Wind class, representing wind speed and direction.
- String dt_txt: A string representing the date and time of the forecast entry.

MainData (Nested Static Class)

This class holds the primary weather data like temperature, pressure, and humidity.

Fields:

- double temp: The temperature in the city at the forecasted time.
- double feels_like: The "feels-like" temperature, representing how the temperature feels to humans.
- double temp_min: The minimum temperature forecasted for that time period.
- double temp_max: The maximum temperature forecasted for that time period.
- int pressure: The atmospheric pressure.
- int humidity: The humidity level in percentage.

WeatherInfo (Nested Static Class)

This class provides detailed information about the weather conditions (e.g., clear sky, rain).

Fields:

- int id: An ID representing the type of weather.
- String main: A brief description of the weather condition (e.g., "Clear", "Rain").
- String description: A more detailed description of the weather.
- String icon: A reference to an icon representing the weather condition.

Clouds (Nested Static Class)

This class represents the cloudiness in the forecasted location.

Fields:

- int all: The percentage of cloud cover in the forecast.

Wind (Nested Static Class)

This class holds data regarding the wind at the forecasted location.

Fields:

- double speed: The wind speed in meters per second.
- int deg: The wind direction in degrees.

City (Nested Static Class)

This class represents the city for which the weather data is being returned.

Fields:

- String name: The name of the city.
- Coord coord: An instance of Coord, which holds the geographical coordinates (latitude and longitude) of the city.

Coord (Nested Static Class)

This class contains the geographical coordinates (latitude and longitude) for the city.

Fields:

- double lat: The latitude of the city.
- double lon: The longitude of the city.

Usage

This class is typically used to model the response returned by the OpenWeatherMap API when retrieving weather information. It provides a structured way to store and access weather data, including forecast details, temperature, wind speed, cloud cover, and other weather parameters, along with the city's name and coordinates.

ConversationRepository Interface

Overview:

This interface defines the repository for the Conversation entity, providing a way to interact with the database using Spring Data JPA.

Methods:

1. **findByUserId(String userId)**:
    - **Purpose**: This method is used to retrieve a list of Conversation entities associated with a specific user.
    - **Parameters**: A String representing the user ID (userId).
    - **Return Type**: A List<Conversation> containing all conversations that belong to the specified user.

Explanation:

- This interface extends JpaRepository<Conversation, Long>, which means it inherits standard CRUD operations for the Conversation entity, like save(), findAll(), deleteById(), etc., without needing explicit implementations.
- The findByUserId(String userId) method provides a custom query method that Spring Data JPA interprets to generate a query for finding all Conversation records associated with a specific user based on the userId field.

MessageRepository Interface

Overview:

This interface defines the repository for the Message entity, providing various methods to interact with the database using Spring Data JPA.

Methods:

1. **findByConversationIdOrderByTimestampAsc(Long conversationId)**:
    - **Purpose**: This method retrieves a list of messages associated with a specific conversation, ordered by the timestamp in ascending order.
    - **Parameters**: A Long representing the conversation ID (conversationId).
    - **Return Type**: A List<MessageView> containing messages for the specified conversation, ordered by timestamp.
2. **findByUserIdAndConversationId(String userId, Long conversationId)**:
    - **Purpose**: This method retrieves a list of messages based on both the user ID and the conversation ID.
    - **Parameters**:
        - A String representing the user ID (userId).

        - A Long representing the conversation ID (conversationId).
    - **Return Type**: A List<Message> containing messages for the specified user and conversation.
3. **findByConversationId(Long conversationId)**:
    - **Purpose**: This method retrieves a list of messages associated with a specific conversation.
    - **Parameters**: A Long representing the conversation ID (conversationId).
    - **Return Type**: A List<Message> containing messages for the specified conversation.

Explanation:

- The interface extends JpaRepository<Message, Long>, which provides standard CRUD operations for the Message entity without needing explicit implementations.
- The custom methods:
    - findByConversationIdOrderByTimestampAsc(Long conversationId): Retrieves all messages for a given conversation, sorted by timestamp.
    - findByUserIdAndConversationId(String userId, Long conversationId): Retrieves all messages for a specific user in a given conversation.
    - findByConversationId(Long conversationId): Retrieves all messages in a given conversation without sorting.

MessageView Interface

Overview:

The MessageView interface is a JPA projection used for read-only retrieval of selected message fields. Instead of fetching the entire Message entity from the database, which might contain unnecessary fields, this projection allows for optimized querying by only selecting the columns that are relevant to the chat flow (in this case, sender, content, and timestamp).

Methods:

1. **getSender()**:
    - **Purpose**: Retrieves the sender of the message.
    - **Return Type**: String.
2. **getContent()**:
    - **Purpose**: Retrieves the content (text) of the message.
    - **Return Type**: String.
3. **getTimestamp()**:
    - **Purpose**: Retrieves the timestamp when the message was sent.
    - **Return Type**: LocalDateTime.

Explanation:

- **JPA Projection**: In JPA, a projection allows for fetching only the necessary parts of an entity, rather than the entire entity, which can improve performance, especially when dealing with large datasets.
- **Use Case**: In your case, this projection is specifically designed to retrieve the essential fields (sender, content, and timestamp) that are needed to display a chat message in the chat flow.

ConversationService Class

Overview:

The ConversationService class handles the main business logic related to managing conversations, messages, and PDF generation. It interacts with the ConversationRepository, MessageRepository, and PdfGeneratorService to provide essential features for managing chat interactions.

Key Methods:

1. **startNewConversation(String userId)**:
   - **Purpose**: Creates and saves a new conversation for a user.
   - **Details**: This method sets the userId and initializes the conversation with a default title and the current timestamp as startedAt.
2. **getUserConversations(String userId)**:
   - **Purpose**: Retrieves all conversations associated with a particular user.
   - **Details**: Uses conversationRepository.findByUserId() to fetch all conversations.
3. **getConversationMessages(Long conversationId)**:
   - **Purpose**: Fetches the messages of a specific conversation.
   - **Cache**: The method is annotated with @Cacheable("conversations") to cache the conversation messages, improving performance by avoiding repeated database queries.
4. **addMessage(Long conversationId, String sender, String content, String userId)**:
   - **Purpose**: Adds a new message to a conversation.
   - **Details**: Creates a Message object, associates it with the conversation, and saves it to the database.
5. **getMessagesByUserAndConversation(String userId, Long conversationId)**:
   - **Purpose**: Retrieves messages from a conversation by a specific user.
   - **Details**: Uses messageRepository.findByUserIdAndConversationId() to find the messages by user and conversation ID.
6. **updateConversationTitle(Long conversationId, String newTitle)**:
   - **Purpose**: Updates the title of a conversation.
   - **Details**: Fetches the conversation by ID and updates its title in the database.
7. **deleteConversation(Long conversationId)**:

   - **Purpose**: Deletes a conversation by ID.
   - **Details**: Uses conversationRepository.deleteById() to remove the conversation.
8. **generateConversationPdf(Long conversationId)**:
   - **Purpose**: Generates a PDF document summarizing the conversation.
   - **Details**: It assembles all the messages into a user-friendly format (e.g., "User: <content>" and "Tripper: <content>") and then calls the pdfGeneratorService to generate the PDF.
9. **getConversationById(Long id)**:
   - **Purpose**: Retrieves a specific conversation by its ID.
   - **Details**: If the conversation is not found, it throws an exception.

Caching in getConversationMessages:

The method getConversationMessages is annotated with @Cacheable("conversations"), which means that Spring will cache the results of this method, so repeated requests for the same conversation's messages will be served from the cache, reducing the load on the database.

PDF Generation Logic in generateConversationPdf:

- It fetches the conversation and its messages.
- If no title is provided for the conversation, a default title ("Conversation #<id>") is used.
- The user ID is split and formatted for better readability.
- Each message is processed to differentiate between messages sent by the user and messages sent by the assistant (Tripper).
- Finally, the messages are passed to the PdfGeneratorService to generate a PDF from the formatted content.

General Flow:

1. **Conversation creation**: A user starts a conversation.
2. **Message addition**: Messages are exchanged within the conversation.
3. **PDF generation**: The conversation can be converted into a PDF document for later use or sharing.
4. **Conversation management**: Conversations and their messages can be updated or deleted as needed.

PdfGeneratorService Class

Overview:

The PdfGeneratorService class is responsible for generating PDF documents from text-based content, particularly used to create PDFs summarizing the conversation between the user and the assistant (Tripper). The service uses the iText library to create and manipulate PDF documents.

Key Method:

generatePdfFromText(String title, String content, String userName):

- **Purpose**: Converts a conversation into a PDF document with formatted text.
- **Parameters**:
    - title: The title of the PDF (e.g., "Conversation Summary").
    - content: The text content of the conversation, typically formatted as a dialogue between the user and the assistant.
    - userName: The user's name, used to differentiate messages sent by the user and the assistant.
- **Returns**: A byte array representing the PDF file.
- **Steps**:
    - **Document Setup**: A Document object is created, which represents the PDF. The document is set up to output into a ByteArrayOutputStream.
    - **Fonts**: Different fonts are defined for the title, user messages, and assistant messages:
        - titleFont: For the title of the PDF (bold and larger).
        - userFont: For the user's messages (blue and bold).
        - tripperFont: For the assistant's messages (dark gray and bold).
        - messageFont: For regular message text.
    - **Title**: The title of the conversation is added at the top of the PDF, centered and with spacing.
    - **Content Processing**:
        - The content of the conversation is split into lines.
        - Each line is checked to see if it's from the user or the assistant. Based on this, the font and indentation are set appropriately.
        - Each message is added to the PDF with an indentation and proper formatting.
    - **Error Handling**: If there is an issue during the PDF creation process, an exception is thrown.

Detailed Steps in generatePdfFromText:

1. **Document Initialization**:
    - A new Document is created, and the PDF writer is set to output to a ByteArrayOutputStream (to return the PDF as a byte array).
2. **Fonts**:
    - Different fonts are used to distinguish the user's messages from the assistant's.
    - Title font: Large and bold for the title.
    - User font: Blue and bold for the user's messages.
    - Assistant font: Dark gray and bold for the assistant's messages.
    - General message font: Regular for the content.
3. **Title**:

    - The title of the PDF is centered, styled with the title font, and given some spacing before adding it to the document.

4. **Content Processing**:
    - The content is split by line breaks (\n).
    - The method loops through each line and determines if it's from the user or the assistant. It checks if the line starts with the user's name or "Tripper".
    - Based on the speaker, the appropriate font and indentation are applied to each message.
    - If the line doesn't belong to either speaker, it's simply added with general message formatting.
5. **Closing**:
    - The document is closed, and the ByteArrayOutputStream containing the PDF data is returned as a byte array.

SentenceDetectionService Class

Overvirw:

The SentenceDetectionService class provides functionality to detect sentences in a paragraph using OpenNLP. This service uses the OpenNLP library, which provides pre-trained models for various natural language processing tasks, including sentence detection.

Key Components:

1. **SentenceDetectorME**:
    - This is the core class from OpenNLP that handles sentence detection. It uses a trained model to segment text into individual sentences.
2. **SentenceModel**:
    - The model used by SentenceDetectorME to perform sentence detection. It contains the rules and patterns required to identify sentence boundaries.
3. **ClassPathResource**:
    - This is used to load the sentence detection model from the classpath. The model file (opennlp-en-ud-ewt-sentence-1.2-2.5.0.bin) is expected to be placed in the resources/models directory.

Constructor:

- **Initialization**:
    - The constructor loads the sentence detection model (opennlp-en-ud-ewt-sentence-1.2-2.5.0.bin) from the classpath.
    - It uses SentenceModel to load the model and initializes the SentenceDetectorME object that performs the actual sentence detection.
    - If there's an error while loading the model, it catches the exception and logs the error message.

Key Method:

detectSentences(String paragraph):

- **Purpose**: Detects and returns an array of sentences from a given paragraph.
- **Parameters**:
  - paragraph: The input text in which sentences need to be detected.
- **Returns**: An array of sentences detected in the paragraph.
- **Steps**:
  - The method calls the sentDetect function of the SentenceDetectorME class, passing the paragraph as an argument. This function segments the paragraph into individual sentences.
  - It returns the detected sentences as an array of strings.

TripChatService Class

Overview:

The TripChatService is a service class responsible for interacting with the user through a conversational interface. It utilizes several other services, such as ChatGPTClient, ConversationService, TripInfoExtractionService, and WeatherService, to assist users with travel-related queries and provide personalized travel suggestions based on the context of the conversation.

Key Components:

1. **ChatGPTClient**:
   - A client that interacts with the ChatGPT API to generate responses based on the conversation context and the prompt constructed in this service.
2. **ConversationService**:
   - A service that handles conversations, including retrieving messages for a specific conversation and adding new messages.
3. **TripInfoExtractionService**:
   - This service is responsible for extracting useful trip-related information (like locations and dates) from the user's last message, which is then used to provide relevant responses.
4. **WeatherService**:
   - A service that fetches weather data for the locations provided by the user, allowing the chatbot to offer more personalized travel advice, including weather conditions for the trip.

Methods:

1. chatWithGPT(Long conversationId):

- **Purpose**: The primary method that integrates with ChatGPT, builds the prompt based on the conversation history, extracts travel-related information, and includes weather information before calling the ChatGPTClient to get a response.
- **Steps**:
  1. **Retrieve the conversation messages**: The method retrieves all messages for the given conversation ID using conversationService.getConversationMessages.
  2. **Build the conversation prompt**: It builds a prompt from the user's message history using the buildPromptFromMessages method.
  3. **Extract trip-related information**: The last user message is passed to tripInfoExtractionService.extract(), which extracts locations and dates.
  4. **Fetch weather data**: For each extracted location, the weatherService.getForecastData(city) is called to get weather data.
  5. **Build the prompt**: The method constructs a personalized prompt for ChatGPT, including weather details and relevant conversation context. It also provides guidance on how to respond based on whether dates have been provided.
  6. **Generate response**: The constructed prompt is passed to chatGPTClient.getChatResponse to generate a response from the assistant.

2. buildPromptFromMessages(List<MessageView> messages):

- **Purpose**: Constructs the conversation prompt by iterating through all the messages in a conversation and appending their content.
- **Returns**: A string containing all the conversation messages concatenated.

TripInfoExtractionService

Overview:

The TripInfoExtractionService is responsible for analyzing a given text to extract useful information related to travel, specifically **locations** and **dates**. It uses patterns, keywords, and external services (like Google Maps) to identify these elements. This service is useful for understanding user input in natural language, especially when the user provides a mix of information about locations and potential travel dates.

Key Components:

1. **GoogleMapsService**:
   - A service that helps validate locations, ensuring that they are actual places or recognizable locations. This can be used to verify if a specific location is valid based on the text input.
2. **MONTHS**:
   - A list of all month names in lowercase, used for detecting month names in the user input.

3. **DATE_PHRASES**:
    - A set of predefined date-related phrases that are common in natural language when talking about upcoming trips (e.g., "next week", "this month", etc.).
4. **DATE_PATTERN**:
    - A regex pattern used to identify possible date expressions like "next week", "March", "12/12", etc. This helps capture dates from user input even when they are not formatted conventionally.

Methods:

1. extract(String text):

- **Purpose**: Extracts locations and dates from the provided text.
- **Steps**:
    1. **Normalize and Split the Text**: The input text is split into words, and unnecessary words (like "to", "in", "at") are ignored during processing.
    2. **Location Extraction**:
        - The method checks combinations of words (using a "sliding window" approach) to identify potential locations.
        - It considers combinations of one, two, or three words at a time, checking if the resulting string forms a valid location using the googleMapsService.isValidLocation() method. This helps in identifying real places.
        - Stop words (e.g., "to", "the") are removed to ensure meaningful phrases are considered as location candidates.
    3. **Date Extraction**:
        - The method looks for specific date-related phrases (e.g., "next week", "this month") in the user input.
        - It also checks for month names and uses the DATE_PATTERN regex to detect possible date expressions in the text.
    4. **Returns**:
        - A TripInfo object containing two lists:
            - **locations**: The valid locations identified in the text.
            - **dates**: The potential dates or date-related phrases extracted from the text.

Example Workflow:

Consider user input: "I am planning to go to Paris next month, maybe in June or July."

1. **Location Extraction**:

    - The service will process the phrase "go to Paris" and identify "Paris" as a valid location (after validating it with Google Maps).
2. **Date Extraction**:
    - The phrase "next month" will be captured by the DATE_PHRASES list.
    - "June" and "July" will be identified as valid month names from the MONTHS list.
3. **Result**:
    - The extracted locations: ["Paris"]
    - The extracted dates: ["next month", "June", "July"]

Thus, the TripInfo object returned would have these values.

TripPlannerService

The TripPlannerService is responsible for processing a user's trip request and generating personalized travel recommendations using the following steps:

1. **Input Parsing**: It parses the user's input trip details (e.g., locations, dates) to understand the trip's context.
2. **Weather Information**: It gathers weather information for each destination city, using a weather service.
3. **ChatGPT Integration**: It constructs a structured prompt and sends it to the ChatGPT API for dynamic, personalized travel recommendations.
4. **Response Parsing**: It processes the ChatGPT response into a structured format, which includes recommendations for each city, including clothing, accessories, and other travel items.

This service integrates NLP (Natural Language Processing) for input parsing and external services for weather and chat generation.

Key Components:

1. **ChatGPTClient**:
    - A client used to interact with the ChatGPT API to generate personalized responses and travel recommendations based on the user's trip details.
2. **WeatherService**:
    - A service used to fetch weather forecasts for the cities the user plans to visit. This helps provide relevant weather information for travel advice.
3. **NLPInputParser**:
    - A utility that parses the user's input trip details (locations, dates) to extract relevant information for further processing.
4. **TripDetails**:

    - Contains information extracted from the user's trip request, such as cities, travel dates, and any other relevant details.
5. **TripResponse**:
    - The final response object returned by the service, which includes the user's name, a structured list of recommendations, and any additional information provided by ChatGPT.
6. **GPTChecklistParser**:
    - A utility that parses the structured response from ChatGPT into a list of **TripChecklistSection** objects, making it easier to display or use the recommendations.

Key Methods:

1. processTripRequest(TripRequest request):

This method handles the entire process of generating a trip plan for the user. It involves several steps:

1. **Parse Input**:
    - The nlpParser.parseTripDetails(request.getTripDetails()) method is used to extract locations (cities) from the trip details.
2. **Build Weather Section**:
    - For each city in the user's trip details, it fetches the weather information using the weatherService.getForecastData(city) method.
    - If the weather information is available, it is appended to a string, which will later be used to build a prompt for ChatGPT.
3. **Build ChatGPT Prompt**:
    - A strict format is defined in the prompt, which includes user-provided details (like cities and weather) and specific instructions on how ChatGPT should format its response.
    - The format includes the following sections for each city:
        - **Weather**: Description of the weather.
        - **Clothing**: Recommendations for clothing.
        - **Accessories**: Suggested accessories.
        - **Optional Items**: Extra items that might be useful.
4. **Get Dynamic Response from ChatGPT**:
    - The chatGPTClient.getChatResponse(conversationContext) sends the structured prompt to ChatGPT and retrieves a dynamic response with personalized travel recommendations.
5. **Parse Structured Response**:
    - The response from ChatGPT is parsed into a structured format using the GPTChecklistParser.parse(dynamicResponse) method, which turns it into a list of **TripChecklistSection** objects.

6. **Return TripResponse**:
    - The final TripResponse is returned, which includes the user's name, the original response from ChatGPT, and the structured checklist sections.

Example Workflow:

1. **User Input**:
    - User provides trip details like: "I am planning to visit Paris and London next month."
2. **Weather Information**:
    - The service fetches weather data for both Paris and London.
3. **ChatGPT Prompt**:
    - The system constructs a prompt that includes:
        - User's trip details.
        - Weather information for Paris and London.
        - Format instructions for ChatGPT to generate travel recommendations.
4. **ChatGPT Response**:
    - ChatGPT responds with a structured list for each city, including:
        - **Paris**:
            - **Weather**: 18°C, cloudy.
            - **Clothing**: Light jacket, comfortable shoes.
            - **Accessories**: Sunglasses, small umbrella.
            - **Optional Items**: Travel guidebook.
        - **London**:
            - **Weather**: 15°C, rainy.
            - **Clothing**: Raincoat, boots.
            - **Accessories**: Umbrella, camera.
            - **Optional Items**: Power bank.
5. **Structured Response**:
    - The response is parsed into a structured format, making it easy for the system to display or use the recommendations.
6. **Final Output**:
    - The TripResponse object is returned, containing the trip recommendations.

GPTChecklistParser Class

Overview:

The GPTChecklistParser class is designed to parse the structured response returned by ChatGPT, which provides city-based travel recommendations (e.g., weather, clothing, accessories, optional items) for different locations. The class processes the response to extract these recommendations into a more usable format, specifically as TripChecklistSection objects.

Key Components and Methods

1. parse(String gptText):

This method processes the entire GPT response, which contains multiple sections, each dedicated to a city. It splits the text by new lines and identifies the city headings (e.g., **RIO DE JANEIRO, BRAZIL**). For each city section, it accumulates the relevant lines and sends them to the parseCityBlock method for further breakdown.

- **City Detection**: The method looks for lines matching a pattern like **CITY NAME**. When a new city is detected, the previous city's accumulated lines are parsed and stored.
- **Storing Sections**: Once all city blocks are processed, the method returns a list of TripChecklistSection objects, each containing details for a specific city.

2. parseCityBlock(String cityName, List<String> lines):

Once a city is identified, the method processes its associated lines. It parses the lines based on specific headings such as "Weather", "Clothing", "Accessories", and "Optional Items", and stores them in the correct section.

- **Headings**: The method checks for headings like **Weather:**, **Clothing:**, etc., and allocates the relevant content to each heading.
- **Buffering**: The method uses a buffer to accumulate lines under each heading. When a new heading is encountered, the previous content is flushed to the appropriate field in the TripChecklistSection.

3. storeBufferInSection(TripChecklistSection section, String heading, List<String> buffer):

This method is responsible for storing the buffered lines into the appropriate fields of the TripChecklistSection based on the current heading.

- **Weather**: If the heading is "Weather", the lines are combined into a single string and stored as the weather information for the city.
- **Clothing, Accessories, Optional Items**: If the heading is "Clothing", "Accessories", or "Optional Items", the buffered lines are parsed into lists of items (i.e., bullet points are stored as individual items).
- **Bullet Points**: Lines that start with - or • are considered bullet points and are stored separately.
- **Plain Text**: Any other lines are treated as plain text and added to the respective section.

4. storeBufferInSection Detailed Parsing:

- The method ensures that the buffered items are parsed correctly based on the section heading.
  - For **Weather**, the lines are concatenated into a single text block.

    - For **Clothing, Accessories, and Optional Items**, lines are treated as bullet points and added to respective lists.

GPTResponseParser Class

Overview:

This class is responsible for parsing example travel responses and converting them into structured TripChecklistSection objects. Each section represents detailed travel information for a specific city, including weather conditions, clothing, accessories, and optional items.

Method:

- **parseFromExampleResponse()**:
    - This method creates example TripChecklistSection objects for two cities (Paris and Barcelona) by simulating a parsed response containing weather data, clothing suggestions, accessories, and optional items for each city.
    - **Returns**:
        - A list of TripChecklistSection objects, where each section contains the parsed trip details for one city.
    - **Purpose**:
        - This method is useful for generating a simulated response from a travel assistant, making it easier to test and verify the structure of trip checklists. It provides pre-defined examples for cities like Paris and Barcelona, with information about packing recommendations based on weather.

Example of Output:

The output of the parseFromExampleResponse() method is a list containing two TripChecklistSection objects for Paris and Barcelona, with the following attributes:

1. **Paris:**
    - **Weather**: August in Paris is warm and sunny, with occasional rain showers. Temperatures range from 17°C (63°F) to 26°C (79°F).
    - **Clothing**:
        - Lightweight clothes (t-shirts, shorts, breathable fabrics)
        - Light sweater or jacket for cooler evenings
        - Comfortable walking shoes
        - Umbrella or lightweight rain jacket for unexpected showers
    - **Accessories**:
        - Stylish scarf
        - Sunglasses

        - Crossbody bag
    - **Optional Items**:
        - Dressier outfit for a night out
        - Reusable water bottle
2. **Barcelona:**
    - **Weather**: August in Barcelona is hot and sunny, with temperatures ranging from 22°C (72°F) to 30°C (86°F).
    - **Clothing**:
        - Light, airy clothing (tank tops, shorts, cotton dresses)
        - Swimwear for the beach
        - Comfortable sandals or sneakers
        - Light cardigan or wrap for cooler evenings
    - **Accessories**:
        - Sun hat
        - Beach towel
        - Small backpack
    - **Optional Items**:
        - Beach accessories (beach bag, flip-flops, cover-up)
        - Guidebook or smartphone with a travel app

Purpose:

The purpose of the GPTResponseParser class is to simulate parsing a GPT-generated response for travel planning. It converts a free-form text response into structured data, breaking it down by city and categorizing items into clothing, accessories, and optional items. This can be particularly useful for testing and development when working with AI-based travel recommendation systems.

NLPInputParser Class

Overview:

This class is responsible for parsing the input text to extract key travel details such as locations and dates. It utilizes natural language processing (NLP) techniques, including sentence detection, tokenization, and part-of-speech tagging, to identify and categorize relevant information from the user input.

Constructor:

- **NLPInputParser()**:
    - This constructor initializes the NLP models required for processing input:
        - **Sentence Detection Model**: Detects sentences in the input text.
        - **Tokenizer Model**: Breaks sentences into tokens (words).

        - **POS Tagger Model**: Tags each token with a part-of-speech (e.g., noun, verb, etc.).
    - The models are loaded from resource files, and exceptions are thrown if the models cannot be found.

Method:

- **parseTripDetails(String input)**:
    - This method processes the provided input text to extract trip details, including locations and dates.
    - **Parameters**:
        - input (String): The raw text containing the trip details, such as city names and travel dates.
    - **Returns**:
        - A TripDetails object containing the extracted locations and travel month.
    - **Functionality**:
        - The method first splits the input text into sentences.
        - Each sentence is tokenized into individual words (tokens), and each token is tagged with a part-of-speech label.
        - The method identifies **locations** (proper nouns, e.g., city names) and **dates** (both explicit date formats and month names).
        - It collects and stores the locations and dates into lists and returns the TripDetails object, which encapsulates these details.

Output Example:

For an input like: "I'm planning a trip to Paris in June and Barcelona in July.", the output would be:

- **Locations**:
    - Paris
    - Barcelona
- **Travel Month**: June July

Purpose:

The purpose of the NLPInputParser class is to help extract meaningful details from a user's trip-related input, such as locations and dates. This is especially useful in travel planning applications where users might provide natural language text (e.g., "I want to visit Paris and Barcelona in the summer") and the system needs to process it into structured data for further analysis or recommendations.

TripChecklistGenerator Class

Overview:

The TripChecklistGenerator class is designed to generate a travel checklist based on the trip details, particularly focusing on the month or season of travel. It uses simple heuristics based on the travel month to provide relevant items that should be packed for the trip.

Method:

- **generateChecklist(TripDetails details)**:
    - This method generates a checklist of essential, recommended, optional, and accessory items for a trip based on the travel month contained in the TripDetails.
    - **Parameters**:
        - details (TripDetails): An object containing trip-related information, including the travel month.
    - **Returns**:
        - A TripChecklist object that contains arrays of recommended items for the trip, categorized into essential items, recommendations, optional items, and accessories.
    - **Functionality**:
        - The method checks the travel month from the TripDetails. If the month is recognized as a summer month (June, July, August) or contains the word "summer," a set of summer-related items is generated. Similarly, for winter months (December, January, February), it generates winter-related items. If the month is not recognized or it's a neutral month, a general list of items is returned.
        - The generated checklist contains arrays of essential items, recommended items, optional items, and accessories. Each list corresponds to the type of items that are appropriate for the specified travel month or season.

Purpose:

The primary purpose of the TripChecklistGenerator class is to assist travelers in packing for their trips by suggesting items that are relevant to the season or month of travel. It simplifies the packing process by categorizing items into essential, recommended, optional, and accessory items, based on basic seasonal guidelines.

Application Class

Overview:

The Application class serves as the main entry point for the **Tripper** application, a Spring Boot-based application. It is responsible for launching the application and enabling caching support. This class uses Spring Boot annotations to enable the necessary configurations and features for the application.

Annotations:

- **@SpringBootApplication**:

    - This is a convenience annotation that combines several other annotations such as @Configuration, @EnableAutoConfiguration, and @ComponentScan. It enables Spring Boot’s auto-configuration, component scanning, and configuration properties support, making it the primary configuration annotation for Spring Boot applications.
- **@EnableCaching**:
    - This annotation enables caching support in the Spring Boot application. By adding this annotation, the application is capable of using caching mechanisms to improve performance by storing and retrieving previously computed data instead of recalculating it.

Method:

- **main(String[] args)**:
    - The main method is the entry point of the Java application. It uses the SpringApplication.run() method to launch the Spring Boot application.
    - **Parameters**:
        - args (String[]): Command-line arguments passed to the application at runtime.
    - **Functionality**:
        - The method calls SpringApplication.run() to start the Spring Boot application and initialize the necessary components and services. This includes setting up the application context and loading configuration settings.

Purpose:

The Application class serves as the main configuration and entry point for the **Tripper** application. It is responsible for initiating the Spring Boot application and enabling caching to optimize performance.