DESIGN DOCUMENT FOR HEPIK

Based on our previous document we have established the following:

User Identification and Greeting
Input: User enters the chat.
Action: System prompts for user's name.
Output: System stores the name and uses it in further interactions.

Trip Cities Input
Input: User provides a list of city names.
Validation: System checks if the provided names correspond to recognizable cities.
Output: System stores the cities for trip planning.

Trip Dates Input
Input: User enters dates corresponding to each city in a specific format.
Validation: System checks the format and validity of the dates (e.g., not past dates).
Output: System associates each date with the corresponding city and stores this information.

Trip Confirmation
Input: System summarizes the trip details and asks for user confirmation.
Action: User confirms or corrects the details.
Output: System finalizes the trip plan details on confirmation.

Trip Suggestions and Best Time
Input: With the trip details confirmed, the system processes the information.
Action: System generates suggestions for events, sites, and advises on the best time to visit.
Output: System presents the user with suggestions and advice.

Additional Trip Planning
Input: System inquires if the user wants to plan more trips.
Action: User decides whether to proceed with more planning or end the session.
Output: System either loops back to the trip cities input for more planning or ends the interaction.

This document serves the purpose of detailing on each of these steps based on what the system should do with the user input, validations and expected outcomes.

DATA STRUCTURES:

Introduction

This section outlines the primary data structures that will be used by HEPIK to manage and process user interactions. These structures are critical for handling the flow of information from initial user input, through external API interactions, to the final output presented to the user. Each structure has been designed to encapsulate relevant data and to facilitate the operations needed to deliver accurate travel and clothing recommendations based on weather forecasts.

UserSession
Purpose: Represents a user's interaction session, encapsulating all the details of their planned trips.
Properties:
userName: String to store the user's name.
cityTrips: List of CityWeatherTrip objects.
confirmationStatus: Boolean to track whether the user has confirmed the trip details.

CityWeatherTrip
Purpose: Represents the details of a planned trip to a single city, including the date of the visit and associated weather forecasts and clothing recommendations.
Properties:
cityName: String to store the name of the city.
visitDate: LocalDate to store the planned date of visit.
forecast: WeatherForecast object that holds weather data.
clothingRecommendation: ClothingRecommendation object that holds packing advice.
Relationships: Part of the cityTrips list within UserSession. The WeatherForecast and ClothingRecommendation are composed within it.

WeatherForecast
Purpose: Stores weather information retrieved from the Open-Meteo API, which is crucial for determining appropriate clothing recommendations.
Properties:
temperature: Double to store the forecasted temperature.
weatherCondition: String to store the textual weather conditions (e.g., Sunny, Rainy).
Source: Populated by making a call to the Open-Meteo API with the cityName and visitDate from CityWeatherTrip.

ClothingRecommendation
Purpose: Provides a list of recommended clothing items based on the temperature and weather conditions forecasted.
Properties:
itemsToPack: List of Strings, where each String is a recommended clothing item to pack.

Generation: Filled by a method that maps WeatherForecast data to clothing items.

Data Flow

The data flow begins with the user entering their name, followed by the names and dates for their intended trips. Each city and date is encapsulated in a CityWeatherTrip object, which is added to the cityTrips list within a UserSession object. As the user confirms their trip details, HEPIK calls the Open-Meteo API, populating WeatherForecast objects within each CityWeatherTrip. Subsequently, the clothingRecommendation is generated for each trip.

Persistence and Serialization

The persistence strategy will determine how user sessions are stored to ensure data is not lost between interactions. This could be through in-memory data stores, which are fast but volatile, or more permanent solutions like databases or file systems.

The process of serialization will be used to convert the in-memory instances of UserSession and related classes to a format that can be easily stored and retrieved.


```
{
  "userName": "Alice",
  "cityTrips": [
    {
     "cityName": "Cairo",
     "visitDate": "2024-05-14",
      "weatherForecast": {
        "temperature": 35.0,
        "weatherCondition": "Sunny"
      },
      "clothingRecommendation": {
        "itemsToPack": ["Lightweight clothing", "Sun hat", "Sunglasses"]
      }
    }
  ],
  "confirmationStatus": true
}
```


VALIDATION RULES:

User Name Validation

Rule: The name should only contain alphabetic characters, spaces, hyphens, and apostrophes.

Regex Pattern: "^[a-zA-Z\\s'-]+$"

Java Implementation: Use Pattern and Matcher from java.util.regex to validate the name.

Error Handling: Prompt the user to re-enter their name if the validation fails, with a message explaining the allowed characters.

City Names Validation
Rule: City names should be strings composed of alphabetic characters and may contain spaces and hyphens. Validation should also ensure the name corresponds to a real city, which can be checked using the Open-Meteo API or a predefined list of city names.
Regex Pattern: "^[a-zA-Z\\s-]+$"
Java Implementation: In addition to regex validation, perform an API call to Open-Meteo to verify the city name is recognized.
Error Handling: If the city is not recognized by the API, inform the user and ask for a new city name.

Dates Validation
Rule: Dates must be in the format DD/MM/YYYY and must not be in the past.
Regex Pattern: "^(0[1-9]|[12]\\d|3[01])/(0[1-9]|1[0-2])/\\d{4}$"
Java Implementation: After regex validation, use LocalDate and a DateTimeFormatter to parse the date and compare it to the current date to ensure it is not past.
Error Handling: If the date is in the past or the format is incorrect, prompt the user to enter a valid future date in the correct format.

Weather Data Validation
Rule: Temperature values should be within a reasonable range for Earth's climate, and weather conditions should be non-empty strings.
Java Implementation: Check the temperature range (e.g., -50°C to +50°C) and ensure the weather condition is not empty or null when processing Open-Meteo API responses.
Error Handling: If the API returns data outside of expected bounds, handle this gracefully, possibly with default advice or prompting the user for re-entry.

USER FEEDBACK:

Positive Feedback
Correct Input Acknowledgement: Whenever the user provides correct input, HEPIK should acknowledge it with a confirmation message, e.g., "Got it, you're planning to visit Cairo on 14/05/2024."
Progress Updates: Inform the user of what HEPIK is doing at each step, e.g., "Checking the weather for Cairo," to ensure they understand the process is ongoing.

Input Correction Guidance
Invalid Input: If a user input is invalid, provide a specific error message guiding them on how to correct it. For example, if the date format is incorrect, HEPIK

could respond with "I didn't understand the date. Please enter it in DD/MM/YYYY format, like 14/05/2024."
Unrecognized City Name: If a city name is not recognized, the feedback could be "I couldn't find that city. Could you please check the spelling and try again?"

Confirmation

Summary Confirmation: Before proceeding with data processing, HEPIK should summarize the trip details and ask the user to confirm. This ensures that the user agrees with the interpreted input before any processing takes place.

ERROR HANDLING:

Input-Related Errors
Regex Validation Failures: If input fails regex validation, prompt the user to re-enter the data in the correct format.
Past Date Entry: If a date entered is in the past, request the user to enter a future date.

API Errors
Open-Meteo API Failure: If there is an issue with the Open-Meteo API, such as a timeout or a server error, HEPIK should notify the user and may provide a default set of clothing recommendations based on average conditions for the time of year.
API Data Anomalies: If the weather data received is outside expected ranges or otherwise anomalous, notify the user that the weather information may be inaccurate and proceed with caution, possibly by offering generic advice.

System Errors
Unhandled Exceptions: In the case of an unexpected system error, present the user with a friendly error message, log the details for internal review, and, if possible, provide the user with the option to restart the interaction or contact support.

The combination of comprehensive user feedback and robust error handling contributes to a resilient system that prioritizes user experience and ensures the continuity of service, even when facing unexpected conditions or inputs.