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

A picture containing text, outdoor, building, white

Description automatically generated

Software Development 2

BSCH-SD2

Chatbot Project  
Final Documentation

* **ChatGPTClient Class**

**Overview:**

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

**Dependencies:**

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

**Attributes:**

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

**Public Methods:**

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

**Overview:**

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

**Dependencies:**

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

**Attributes:**

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

**Constructor:**

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

**Public Methods:**

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

**Overview:**

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

**Dependencies:**

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

**Attributes:**

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

**Constructor:**

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

**Public Methods:**

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

**Error Handling:**

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

**Overview:**

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

**Dependencies:**

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

**Annotations Explained:**

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

**Attributes:**

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

**Overview:**

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

**Dependencies:**

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

**Annotations Explained:**

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

**Implemented Interface:**

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

**Overridden Methods:**

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

**Overview:**

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

**Dependencies:**

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

**Annotations Explained:**

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

**Implemented Interface:**

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

**Overridden Methods:**

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

**Overview:**

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

**Dependencies:**

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

**Annotations Explained:**

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

**Fields:**

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

**Public Endpoints:**

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

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

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

Retrieve the entire message history for a conversation.

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

Retrieve all conversations for a specific user.

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

Retrieve all messages for a user and conversation ID.

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

Update the title of a conversation.

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

Delete a conversation.

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

Export a conversation as a PDF file.

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