Annual Planner Application CPA Griffith College Dublin

Assignment 2

Rohin Mehra
Student Id:3082862
Department: Big Data Analysis and Management

16 April 2023

Application Code Functions and Operations Description

def createUserInfo(claims)

This code defines a function called createUserInfo that creates a new entity in the datastore to store information about a user. The function takes one argument, claims, which is a dictionary containing information about the user, such as their email and name.

The function first creates a key for the new entity using the user's email address as the key ID. Then it creates a new datastore entity object and initializes it with the email and name data from the claims dictionary.

Finally, the function uses the datastore_client object to store the new entity in the datastore.

def retrieveUserInfo(claims)

This code defines a function named retrieveUserInfo, which retrieves the user information from the datastore based on the claims provided.

The function first creates a key for the user info using the email address provided in the claims dictionary. It then uses the get() method to retrieve the user info entity from the datastore.

If the entity does not exist, the function creates a new user info entity, initializes its properties with default values (an empty list for events and calendars, and an empty list for calendar keys), and stores it in the datastore using the put() method.

Finally, the function returns the user info entity, whether it was retrieved or created.

def add_calendar(user_info, calendar_name, event_id, calendar_id=None, calendar_shared=False, calendar_shared_id=None)

The add_calendar function adds a new calendar entity to the datastore for the user specified in user_info.

The function takes in several arguments: user_info which is a dictionary containing information about the user, calendar_name which is the name of the new calendar, event_id which is the ID of the event to be added to the new calendar, calendar_id which is an optional unique ID for the new calendar, calendar_shared which is a boolean flag indicating if the calendar is shared with others, and calendar_shared_id which is the ID of the shared status for the new calendar.

The function first checks if a calendar with the same name already exists in the user's info and raises a ValueError if it does. If a unique calendar_id is not provided, the function generates a unique ID for the new calendar using uuid.uuid4(). It then creates a new calendar entity in the datastore and updates it with the provided data such as calendar_name, event_id, calendar_shared, etc.

The function then gets the list of calendars from the user's info and adds the new calendar to it if it doesn't already exist. It also updates the user's info with the new calendar key and stores it in the datastore. Finally, the user's info is stored in the datastore using datastore_client.put().

def add_event(user_info, day, month, year, event_name, event_start, event_end, event_description, event_shared, event_shared_with, event_id)

This code defines a function add_event that adds a new event to the user's event_list in the Datastore.

The function takes the following parameters:

user_info: A dictionary that contains the user's information including the event_list.

day, month, year: The day, month, and year of the event.

event_name: The name of the event.

event_start, event_end: The start and end time of the event.

event_description: The description of the event.

event_shared: A boolean indicating whether the event is shared or not.

event_shared_with: The user's email ID with whom the event is shared, if applicable.

event_id: The unique id of the event.

The function then creates a new event entity in the Datastore with the specified properties and puts it into the Datastore. It also updates the user_info dictionary with the new event entity's key and stores the updated user_info in the Datastore.

Note that the code also generates unique ids for the day_id, event_start_id, event_end_id, and event_name_id properties using the provided day, month, year, event_name, event_start, and event_end values.

def share_calendar(user_info, calendar_id, recipient_email)

The code above is a Python function named share_calendar that allows users of the app to share their calendar with other users by adding them to the calendar_shared and calendar_shared_id fields of the calendar entity. The function takes three parameters:

user_info: a dictionary that contains the user's information, including their email and a list of their calendars.

calendar_id: a string that represents the ID of the calendar to share.

recipient_email: a string that represents the email of the user to share the calendar with.

The function starts by checking if the owner is trying to share the calendar with themselves, and if they are, it raises a ValueError with an appropriate message.

Next, the function loops through the list of calendars in the user's info to find the calendar with the given calendar_id. If the calendar is found, the function checks if it is already shared with the recipient email. If it is not already shared, the function updates the calendar_shared and calendar_shared_id fields of the calendar to include the recipient email.

The function then retrieves the recipient's information from the datastore using their email address. If the recipient exists in the datastore, the function adds the shared calendar to their calendar_list and stores their updated information in the datastore. If the recipient does not exist in the datastore, the function returns a message indicating that the recipient was not found.

Finally, the function returns a message indicating whether the calendar was shared successfully or if an error occurred.

def compare_schedules(user_info, selected_calendar_ids)

This is a function that finds common time slots between selected calendars for a given user.

The function takes two inputs - user_info, a dictionary containing the user's information including the list of calendars, and selected_calendar_ids, a list of calendar IDs that the user wants to compare.

The function first creates a list of calendar objects based on the provided selected_calendar_ids. It then initializes an empty list of common_time_slots. It loops through all possible pairs of calendars to compare, and for each pair, it loops through all possible pairs of events between the two calendars. It then checks if the two events overlap in time and if they do, it adds the common time slot to the list of common_time_slots.

Finally, the function returns the list of common_time_slots found between the selected calendars.

def recent_events(user_info, days=7)

This code defines a function recent_events that takes two parameters, user_info and days, with days defaulting to 7. The purpose of the function is to find the events that occurred in the last days number of days.

The function starts by getting the list of calendar keys from the user_info dictionary. It then gets the current time using datetime.now() and calculates the start time by subtracting days number of days from the end time using timedelta.

A list named recent_events_list is initialized as an empty list. The function then loops through each calendar key and retrieves the calendar entity from the datastore using the datastore_client.get() method. The function then

loops through each event in the calendar and retrieves the start time of the event.

If the start time of the event falls between the start and end times calculated earlier, the event is added to the recent_events_list. Finally, the function returns the list of recent events.

@app.route('/login', methods=['GET', 'POST'])

This code defines a Flask route for a login page. The route can handle GET and POST requests. In the GET request, the function renders the login page, while in the POST request, it gets the username and password entered by the user in the form, and checks if they are valid. If the username and password are incorrect, the function sets an error message. If they are correct, the function sets a session variable to indicate that the user is logged in, and displays a flash message to the user. Finally, it redirects the user to the index page. The function returns the rendered login page, along with any error message that was set.

@app.route('/logout')

This code defines a route for the /logout page and a corresponding function logout(). When a user visits this page, the function removes the logged_in key from the user's session (if it exists), displays a flash message saying the user has been logged out, and redirects the user to the index page using the redirect() function with the url_for() function to generate the URL for the index page.

@app.route('/')

This code defines a function that handles the home page of a web application. The function uses the Flask framework to define a route at the root URL ("/").

When a user visits the home page, the function retrieves the user's token from cookies and attempts to authenticate the user. If the authentication is successful, the function retrieves the user's information from the datastore, or creates a new user record if this is the user's first time logging in.

The function then renders an HTML template (index.html) and passes in the user's claims, user_info, and any error messages that were generated during the authentication process. The rendered template is returned as an HTTP response.

@app.route('/update_event/<event_id>', methods=['GET', 'POST'])

This is a Flask route and function that handles updating an event in the user's calendar. The route is '/update_event/<event_id>' which means that the function will handle HTTP GET and POST requests for the URL '/update_event/' followed by the event ID. The event ID is passed as an argument to the function.

The function first retrieves the form data from the request which includes the event name, date, start time, end time, and description. It then verifies the user's authentication using the Firebase ID token obtained from the cookies.

Next, the function retrieves the user's information from the datastore and gets the list of events. It then loops through the list of events to find the event with the matching event ID. Once the event is found, it updates its properties with the new values from the form fields.

Finally, the function updates the user's info with the updated event list and stores it in the datastore. It then redirects the user to the 'update_event.html' page.

@app.route('/delete_event/<event_id>', methods=['POST', 'GET'])

The code defines a Flask route and function to handle deleting an event from the user's info in the datastore. It first gets the user's info from the datastore by verifying the Firebase token in the cookies. Then, it removes the event with the given event_id from the event_list in the user's info. Finally, it updates the user's info in the datastore with the updated event list and redirects the user to the delete event page. The route accepts both POST and GET requests, and the event_id is passed in the URL.

@app.route('/share_calendar', methods=['GET', 'POST'])

This code defines a Flask route to handle sharing a calendar. When a user submits the share calendar form, the handle_share_calendar() function is called. It retrieves the calendar ID and recipient email from the form, along with the user's authentication token. It then calls the share_calendar() function, passing in the user's information, calendar ID, and recipient email. If sharing the calendar is successful, the function displays a success message using the flash() function and redirects the user to the share calendar page. If an error occurs during the sharing process, the function displays an error message and redirects the user to the share calendar page.

@app.route('/share_calendar', methods=['GET', 'POST'])

This code defines a route /share_calendar with the HTTP methods GET and POST, and a function share_calendar_page that renders the share_calendar.html template when the user visits the /share_calendar URL.

The template likely includes a form where the user can input a calendar ID and recipient email, and submit the form using a POST request to share the calendar. When the user submits the form, the handle_share_calendar function is called, which retrieves the calendar ID and recipient email from the form, verifies the user's authentication token, retrieves the user's information from the datastore, and calls the share_calendar function to share the calendar with the recipient email. If the calendar is shared successfully, a success message is flashed to the user and they are redirected back to the share calendar page. If there is an error sharing the calendar, an error message is flashed to the user and they are redirected back to the share calendar page.

@app.route('/create_event', methods=['GET', 'POST'])

This code defines a route and a function that handles requests to create a new event. If the request method is POST, it gets the data from the form fields. It validates the data and splits the date into day, month, and year. Then, it verifies the user's token and retrieves the user's information from the datastore. Finally, it calls the add_event function with the information provided and redirects to the create event page. If the request method is GET, it renders the create event page.

@app.route('/edit_event/<event_id>', methods=['GET', 'POST'])

This code defines a Flask route and function for editing an event. It expects a GET request to display the edit event form and a POST request to submit the changes made to the event.

The route is '/edit_event/<event_id>', where <event_id> is a dynamic parameter that represents the ID of the event being edited.

The function takes the event_id as a parameter and retrieves the event data from the user_info entity in the datastore. If the event is not found, it returns a 404 error.

It then retrieves the calendar_id from the form data, and passes it along with the event and event_id to the 'edit_event.html' template.

@app.route('/compare_schedules', methods=['GET', 'POST'])

The code defines a Flask route for a web page that allows users to compare the schedules of multiple calendars. The route handles both GET and POST requests. In the POST request, the function retrieves the selected calendar IDs from the form data, retrieves the user's info from the datastore using the Firebase token, calls the compare_schedules function to find the common time slots between the selected calendars, and renders a template with the common time slots. The template is an HTML page called compare_schedules.html.

@app.route('/delete_calendar/<calendar_id>', methods=['POST', 'GET'])

This is a Flask route definition that defines the /delete_calendar/<calendar_id> route. The route accepts both GET and POST requests.

The calendar_id variable in the route is a dynamic variable that can be any value, and it is passed as an argument to the delete_calendar function.

The function starts by getting the user's information from the datastore, retrieving the calendar list from the user's info, and removing the calendar with the specified calendar_id from the list. The user's info is then updated with the new calendar list, and the updated info is stored back into the datastore.

Finally, the function redirects the user to the delete_calendar.html page.

@app.route('/create_calendar', methods=['POST', 'GET'])

This code defines a route and function for creating a new calendar. If the request method is POST, it gets the calendar name from the form data and checks if it's empty. If it's not empty, it verifies the user's authentication token and retrieves the user's information from the datastore. It generates a unique identifier for the new calendar, then calls the add_calendar function to add the calendar to the user's information in the datastore. Finally, it redirects to the index page. If the request method is not POST, it renders the create calendar page.

@app.route('/edit_calendar/<calendar_id>', methods=['GET', 'POST'])

The code defines a Flask route /edit_calendar/<calendar_id> that handles GET and POST requests. When a GET request is received, the function retrieves the user's information from the datastore, looks for the calendar with the specified calendar_id in the user's calendar list, and renders the edit_calendar.html template with the calendar's information. When a POST request is received, the function gets the updated calendar information from the form data, updates the user's calendar list with the updated information, and stores the updated user's information in the datastore. The function then redirects to the index page.

@app.route('/update_calendar/<calendar_id>', methods=['GET', 'POST'])

The code defines a Flask route /update_calendar/<calendar_id> that handles updating the name of a calendar identified by the calendar_id.

In the function update_calendar, the calendar name is retrieved from the form data and the user's info is retrieved from the datastore using the Firebase ID token stored in the cookie. The calendar list is then updated by searching for the calendar with the specified calendar_id and updating its name. Finally, the updated user's info is saved to the datastore, and the route is redirected to the update_calendar.html page.

@app.route('/search_events', methods=['GET'])

This is a Flask route that handles the search events page. The user enters a search query in the form and submits it. The function retrieves the search query from the form, gets the user's info from the datastore using the Firebase ID token, retrieves the event list from the user's info, and filters the events based on the search query. It then renders the search results page with the matching events.

@app.route('/recent_events', methods=['GET', 'POST'])

This is a Flask route function that handles requests to the '/recent_events' URL path. It allows users to view a list of recent events that have occurred in the past week.

The function first retrieves the search query parameter from the URL, and the user's information from the datastore using the Firebase ID token retrieved from the user's browser cookies.

The function then iterates through the user's calendar keys and retrieves the events from each calendar. It filters the retrieved events based on the search query and also filters for events that occurred in the past week.

Finally, it renders the 'recent_events.html' template with the matching events and recent events lists as arguments to be displayed on the page.