Formula One Application
CPA
Griffith College Dublin

Assignment 1

Rohin Mehra
Student Id:3082862
Department: Big Data Analysis and Management
26 March 2023

Assignment 1 1

createUserInfo(claims) 4

retrieveUserInfo(claims) 4

retrieveDriverList(first_name) 4

retrieveTeamList(name) 5

updateDriver(first_name, last_name, nationality, number, wins, age) 5

updateTeam(name, year_founded, race_wins, world_titles, team_principal, team_base, championships) 5

search_driver(first_name, last_name, age, points) 6

SearchTeam(team_name) 6

add_driver(user_info, first_name, last_name, nationality, birth_date, team_id, number, wins, age) 6

add_team(name, year_founded, race_wins, world_titles, team_principal, team_base, championships) 7

delete_driver(user_info, first_name) 7

delete_team(name) 8

compare_drivers(driver1, driver2) 8

compare_teams(race_wins, world_titles) 8

def login() 9

def logout() 10

def root() 10

def retrieveTeamList() 11

def retrieveDriverList() 11

def SearchDriver() 12

def SearchTeam() 12

def add_driver() 13

def add_team(): 13

def updatedriver() **14**

def updateTeam() **14**

def delete_driver() **15**

def delete_team() **16**

def compare_teams() **16**

Application Code Functions

and

Operations Description

createUserInfo(claims)

This function creates a user info object in the Google Cloud Datastore using the provided user claims data, such as email and name. The entity key is based on the user email and the entity is updated with the user email and name before being stored in the datastore. We will use it to create our users that are creating new accounts to log in to our Formula One Application.

retrieveUserInfo(claims)

This function retrieves the user info from the datastore by using the email from the claims. It creates a key for the user info, gets the user info from the data store using the key, and returns it. The user info includes the email and name of the user. We will use this to retrieve the already-created users for our Formula One Application.

retrieveDriverList(first_name)

The function retrieves a list of drivers by their first_name parameter. It gets the key of a team, brings the team from the datastore, creates an empty list to store the keys of drivers, gets the list of drivers from the data store using the driver keys, and then returns the list of drivers.

retrieveTeamList(name)

The code defines a function retrieveTeamList that takes in a name parameter and retrieves a list of teams from the Datastore. It creates a key for the team entity and then creates an empty list to store the keys of the drivers. Finally, it gets the list of drivers from Datastore and returns it.

updateDriver(first_name, last_name, nationality, number, wins, age)

The code defines a function to update driver information in the datastore. It creates a key for the driver entity, retrieves the driver information from the data store, appends new driver information to the list of drivers, updates the user's info with the new list of drivers, and finally stores the driver entity in the data store.

updateTeam(name, year_founded, race_wins, world_titles, team_principal, team_base, championships)

This code defines a function called updateTeam that takes seven arguments representing the updated information of a team: name, year_founded, race_wins, world_titles, team_principal, team_base, and championships. The function first gets the key of the team entity by using the datastore_client.key() method and passing in the kind of entity ("Team") and the name of the team. It then creates a new entity object for the team using this key.

The function then updates the properties of the team entity with the new information passed in as arguments. Finally, it stores the updated team entity in the datastore using the datastore_client.put() method.

Overall, this function updates the information of a team entity in the datastore by creating a new entity object, updating its properties, and storing it in the data store.

search_driver(first_name, last_name, age, points)

This is a function called search_driver that searches for drivers in the data store based on certain criteria. It takes in four parameters: first_name, last_name, age, and points, which are used to filter the results of the query. The function starts by creating a new query object for the driver kind of entity. If none of the search parameters is provided, it redirects the user to the home page. Then, it adds filters to the query for each search parameter that is provided. Finally, it runs the query and returns the results.

SearchTeam(team_name)

This is a function called search_team which takes a team_name parameter as input. First, the function uses the datastore_client.key method to getting the key of the team. It passes in 'team' as the kind and team_name as the name of the team. Then, the function uses the datastore_client.get method to retrieve the team from the data store based on the key. If the team is not found, the function returns a redirect to the home page. Otherwise, it returns the team.

add_driver(user_info, first_name, last_name, nationality, birth_date, team_id, number, wins, age)

This code is a function called add_driver that adds a new driver to a user's list of drivers. It takes in the user's information and the details of the new driver such as first name, last name, nationality, birth date, team id, number, wins, and age. First, the function retrieves the list of drivers from the user's information. If the user doesn't have any drivers yet, it creates an empty list. Then, the function creates a new dictionary object called the driver that contains all the details of the new driver. The driver dictionary is then appended to the drivers_list. Next, the user's information is updated with the new drivers_list. Finally, the updated user's information is stored in the datastore using the put() method.

add_team(name, year_founded, race_wins, world_titles, team_principal, team_base, championships)

This code defines a function called add_team that adds a team to a user's list of teams. The function takes in the following parameters:
user_info: a dictionary containing information about the user.
name: a string representing the name of the team.year_founded: an integer representing the year the team was founded.race_wins: an integer representing the number of race wins the team has.world_titles: an integer representing the number of world titles the team has.team_principal: a string representing the name of the team's principal.team_base: a string representing the location of the team's base. championships: a list of dictionaries representing the championships the team has won. The function first gets the list of teams from the user's user_info dictionary using the get() method with a default value of an empty list in case the teams_list key does not exist yet. It then creates a new dictionary called team with the information about the new team. The team dictionary is then appended to the teams_list. The function then updates the user_info dictionary with the new teams_list and stores it in the datastore using the put() method.

delete_driver(user_info, first_name)

This code defines a function delete_driver that takes in two arguments: user_info and first_name. It is intended to delete a driver from the user's list of drivers. The code retrieves the list of drivers from the user's info, and then loops through the list to find the driver with the matching first name. Once the driver is found, the code gets the key of the driver to delete from the datastore, deletes the driver from the datastore, removes the driver from the list of drivers, updates the user's info with the new list of drivers, and stores the user's info in the data store.

delete_team(name)

This code deletes a team from the user's list of teams in the datastore. It first retrieves the list of teams from the user's info, then loops through the list of teams to find the team with the specified name. If the team is found, its key is retrieved and used to delete the team from the datastore. The team is also removed from the list of teams and the user's info is updated with the new list of teams. Finally, the updated user's info is stored back into the datastore.

compare_drivers(driver1, driver2)

The function called compare_drivers takes two dictionaries representing the stats of two drivers as arguments (driver1 and driver2). The function then compares the drivers' stats in the categories of "wins" and "age" and returns a list of tuples containing the comparison results. Each tuple contains three elements: the name of the stat being compared (i.e. 'wins' or 'age'), the higher value of the two drivers in that category, and the string 'green' indicating that the driver with the higher value should be highlighted in green in a comparison table or chart. The function first initializes an empty list called comparison to store the comparison results. It then loops through each stat in the stats list (i.e. 'wins' and 'age') and gets the corresponding value for each driver using the get method of the dictionary. If the value of val1 (driver1's value) is greater than val2 (driver2's value), then the function appends a tuple to the comparison list with the stat name, driver1's value, and the string 'green'. If val2 is greater than val1, then the function appends a tuple with the stat name, driver2's value, and the string ‘green'. Finally, the function returns the comparison list of tuples.

compare_teams(race_wins, world_titles)

This code defines a function compare_teams that compares two F1 teams based on their performance statistics and returns a list of tuples containing the comparison results. The function takes two arguments, race_wins and

world_titles, which are the default values for the respective statistics in case they are not provided for the teams being compared. The function accesses the datastore to retrieve the teams being compared based on their team_id keys. It then iterates over a list of statistics to compare, namely race_wins and world_titles. For each statistic, the function retrieves the corresponding value for each team using the get method, which returns the default value if the statistic is not present for a team. The function then compares the values and appends a tuple to the comparison list with the statistic name, the higher value, and the colour green. After comparing the individual statistics, the function then compares the team principals and team bases separately. If one team has more race wins than the other, the tuple (race_wins, value, green) is appended to the comparison list. If not, the tuple (world_titles, value, red) is appended instead. Similarly, if one team has more world titles than the other, the tuple (world_titles, value, green) is appended. Otherwise, the tuple (race_wins, value, red) is appended. Finally, the function returns the comparison list. However, there is a syntax error in the last line where the variable name comparison_team is used instead of comparison.

def login()

This code defines a Flask route for a login page. It listens for both GET and POST requests. If the request method is POST, the function retrieves the username and password from the request form and checks if they are valid. If the credentials are correct, the function sets a session variable indicating that the user is logged in, displays a flash message, and redirects the user to the index page. If the credentials are incorrect, the function sets an error message to be displayed on the login page. If the request method is GET, the function simply renders the login page template with an optional error message.

def logout()

This code defines a Flask route for the '/logout' URL. When a user navigates to this URL, the Flask server will call this function. The function first removes the 'logged_in' key from the user's session using the session.pop() method. This effectively logs the user out by removing their authentication information from the session. Next, the function displays a message to the user using the flash() method. This message will be shown on the next page the user visits since it is stored in the Flask message queue. Finally, the function redirects the user to the index page using the redirect() method and the url_for() function, which constructs the URL for the specified endpoint (in this case, the 'index' function).

def root()

This code defines a route for the home page of a Flask web application. The route is specified using the @app.route('/') decorator, which maps the route to the root() function. The root() function gets the id_token from the cookies and attempts to authenticate the user using the google.oauth2.id_token.verify_firebase_token() function, passing in the id_token and a Firebase request adapter. If the token is valid, the object of the claim is set to the user's claims data, and the function retrieves the user's info from the data store using the retrieveUserInfo() function. If the user info does not exist, it means that the user is logging in for the first time, so the createUserInfo() function is called to create the user info in the data store. If the id_token is invalid, the ValueError exception is caught and the error_message variable is set to the exception message. Finally, the function renders the index.html template, passing in the claims object and error_message variable as context variables.

def retrieveTeamList()

This code defines a route for the "list_teams" page and handles the GET and POST requests.
It first tries to retrieve the user's token from the cookies, which is used to authenticate the user. If the token exists, it verifies it using the "verify_firebase_token" method from the Google OAuth2 library, passing in the token and a "firebase_request_adapter" object.
If the token is valid, the user's claims object is obtained from the token. The code then checks if the necessary keys 'name' and 'email' are present in the object of the claim. If these keys are present, the user's information is retrieved from the datastore using the "retrieveUserInfo" function. If the user information is not present in the data store, it is created using the "createUserInfo" function and then retrieved.
Finally, the "list_teams.html" template is rendered with the error message (if any) and the user's claims object.

def retrieveDriverList()

This code defines a route in a Flask web application for the "/list_driver" URL. The function associated with this route is called retrieveDriverList().
When a request is made to the "/list_driver" URL, the function first gets the Firebase ID token from the cookies of the request. If the token exists, the function tries to authenticate the user by verifying the token using the Firebase API. If the verification is successful, the function retrieves the user information from the datastore using the retrieveUserInfo() function, and if the user information does not exist, it creates it using the createUserInfo() function. Then, the function retrieves the list of drivers associated with the user from the datastore using the retrieveDriverList() function. Finally, the function renders the "list_drivers.html" template, passing in the error_message and claims variables to be displayed on the page.

Overall, this code allows authorized users to retrieve a list of their drivers from the datastore and display it on a webpage.

def SearchDriver()

This code is a Flask route for a search driver page. When the user accesses the page, the SearchDriver() function is called. The function begins by attempting to retrieve the user's token from the cookies, which are used to authenticate the user. If the token exists, the function verifies the token using the verify_firebase_token() method from google.oauth2.id_token library, which returns the object of a claim containing information about the user. If the necessary keys ('name' and 'email') are present in claims, the function retrieves the user's info from the data store. If the user info does not exist, the function creates it in the data store and retrieves it. The function then calls the search driver () function to search the data store for drivers. Any errors encountered during the authentication or retrieval process are stored in the error_message variable. Finally, the function renders the driver_results.html template, passing in the error_message and claims variables.

def SearchTeam()

This code defines a route for a web page that displays team results. It starts with a decorator @app.route('/team_results', methods=['GET', 'POST']) which specifies the URL path for the page and the HTTP methods that are allowed for this endpoint.Inside the function SearchTeam(), the code first gets the id_token from the request cookies. If the token exists, it means that the user is logged in, and the code tries to authenticate the user using the verify_firebase_token function from the google.oauth2.id_token module.
If the authentication is successful, the user's claims object is retrieved from the token, and if the necessary keys (in this case, name) are present in claims, the retrieveUserInfo() function is called to retrieve the user's info from the

data store. Otherwise, an error message is set. Finally, the function renders the team_results.html template and passes the error_message variable to it.

def add_driver()

This code defines a Flask route for adding a driver to a user's information. The route is '/add_driver' and accepts both GET and POST requests.
When a POST request is made, the function retrieves the user's information from the datastore and checks if it exists. If not, it creates new user info in the data store. It then checks if the necessary keys, 'name' and 'email', are present in the object of the claim, which is obtained from the Firebase ID token. If these keys are present, it calls the 'add_driver' function, passing in the driver's information. Otherwise, it sets an error message indicating that the user information is missing. If the Firebase ID token is invalid or missing, the function sets an error message. The function then renders the 'add_driver.html' template with the error message, or without one if there was no error. It's worth noting that the 'add_driver' function is not defined in the code snippet provided, so it's unclear what it does.

def add_team():

This is a Flask route that handles the adding of a new team to a web application. Here's a breakdown of the code:
@app.route('/add_team', methods=['GET', 'POST']): This is a decorator that specifies the URL path for this route. In this case, it's '/add_team'. It also specifies that this route can handle both GET and POST requests.
error_message = None: This creates a variable called 'error_message' and sets its initial value to None. This variable will be used later to store any error messages that may occur during the process of adding a new team.
id_token = request.cookies.get("token"): This retrieves the token from the cookies, if it exists. This token is used to authenticate the user.

if id_token: ...: This checks if a token exists. If it does, the user is assumed to be logged in and the code proceeds to try to authenticate the user. claims = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter): This authenticates the user by verifying the token with the Firebase authentication service. If the token is valid, the user's information is stored in the 'claims' variable. user_info = retrieveUserInfo(claims): This retrieves the user's information from the datastore.if user_info is None: ...: This checks if the user's information does not exist in the datastore. If it doesn't, it means that the user is logging in for the first time and the code proceeds to create the user's information in the datastore.add_team(name, year_founded, race_wins, world_titles, team_principal, team_base, championships): This adds the new team to the user's account.return render_template('add_team.html', error_message=error_message): This returns the add team page, along with any error message that may have occurred during the process.

def updatedriver()

This is a Flask route for the update driver page that handles GET and POST requests. The function is named "updatedriver". The code first checks if the user is logged in by getting the token from the cookies and verifying it using Firebase authentication. If the user is not logged in, the code sets an error message. If the user is logged in, the code retrieves the user's information from the datastore and checks if the necessary information is present in the claims object. If the necessary information is present, the code updates the driver's information with the given parameters. Otherwise, the code sets an error message. Finally, the function renders the "edit_driver.html" template with the error message.

def updateTeam()

This code defines a route for updating team information. The route is accessible via both GET and POST methods. When a user sends a GET request to the route, the edit_team.html template is rendered with an error message if there is any.
When a user sends a POST request to the route, the function attempts to authenticate the user by verifying the Firebase token stored in the cookie. If the token exists and is valid, the user's information is retrieved from the datastore using retrieveUserInfo() function. If the user information is not found, it means the user is logging in for the first time and thus, their information is created in the datastore using the createUserInfo() function.
Once the user's information is retrieved or created, the function attempts to update the team information using the updateTeams() function, passing the team's name, year founded, race wins, world titles, team principal, team base, and championships. If any necessary user information is missing, the error message "Missing user information" is returned.
The update_team.html template is rendered at the end of the function with any error messages passed along as a parameter.

def delete_driver()

This code defines a Flask route for deleting a driver. The route is defined with the URL "/drivers" and the HTTP method "POST". When a user makes a POST request to this route, the "delete_driver" function is called.
The function first checks if the user is logged in by retrieving the Firebase ID token from the user's cookies and verifying it with Firebase. If the token is valid, the user's information is retrieved from the datastore, and if it does not exist, the function creates the user's information in the datastore.
The function then calls the "delete_driver" function, passing in the user's claims (i.e., the user's information retrieved from the ID token) as an argument.

Finally, the function returns a rendered template of the "index.html" page, with any error messages that may have occurred during the process.

def delete_team()

This code is defining a route for a web application using Flask, a popular web framework in Python. Specifically, the route is for handling HTTP POST requests sent to the '/teams' URL.

The function associated with the route is called 'delete_team', which appears to handle the deletion of a team.

The code first checks if the user is logged in by checking for the presence of a token in the request cookies. If the token exists, the code tries to authenticate the user by verifying the token with the Firebase authentication service.

If the user is authenticated, the code checks if the necessary user information (name and email) is present in the token claims. If the information is present, the user's info is retrieved from the datastore and if it does not exist, the user's information is created in the datastore.

Finally, the code renders an HTML template called 'index.html', passing any error messages to the template to be displayed if necessary.

def compare_teams()

This code defines a Flask route named '/team_details' that can handle both GET and POST requests. When a GET request is received on this route, it will render a template named 'team_details.html'. When a POST request is received on this route, it will execute the 'compare_teams' function.

The 'compare_teams' function initializes a variable called 'comparison' to None, which will be used to compare two teams. It then retrieves data from a Google Cloud Datastore using the 'datastore_client.get()' method to obtain the race wins and world titles of two teams, and assigns them to 'team_1', 'team1', 'team_2', and 'team2' respectively.

If a token exists in the cookies, the code attempts to authenticate the user by verifying the token using the 'google.oauth2.id_token.verify_firebase_token()' method. If successful, the user's information is retrieved from the datastore and a comparison is made between the two teams. The comparison is stored in the 'comparison' variable, which is later passed to the template.
If the user information does not exist in the datastore, the code creates a new user information and retrieves it. Then, it proceeds to perform the comparison between the two teams and appends the result to the 'comparison' list.
If the token is invalid, an error message is displayed. Finally, the 'team1', 'team2', 'comparison', and 'error_message' variables are passed to the 'team_details.html' template using the 'render_template()' method.