

**ICT 239**

**Web Application Development**

**TMA January 2023**

|  |  |
| --- | --- |
| Name | Vinteres Leariza Castor |
| T-Group | T02 |
| SUSS PI No | W22\*\*\*\*\* |
| Date Submitted | 01 May 2023 |
| I confirm that the work in this submission is wholly mine. | |

**Instructions:**

1. **Use** **the provided solution template** to answer each question part to avoid missing any requirement.
   * Show only relevant code segments related to your explanation.

Do not copy large chunks of code as it does not show understanding.

1. In your app folder in Vocareum Lab
   * **Update requirements.txt** if you install more packages
   * **Provide instructions** in a file named instructions.txt or instructions.md on how to execute your application if you did not use start.sh to run.
2. Record your runs for each question and place your recordings in the subfolder recordings under the folder /home/labuser/MyWork.
   * The recordings should **include the steps to start** the application running.
   * Show the runs are **carried out on Vocareum Lab**.
3. Copy your app folder and your solution document into the folder /home/labuser/MyWork. Ensure that your solution document uses the naming convention according to canvas submission convention. Rename it at this point if otherwise.
4. Refer to page 19, Assignment Submission of Student Guide for ICT239 v1.0 and demo video on how to create an archive for submission and to get your code copied into the appendix of your solution document as well as details for step 6 and 7. Carry out the instructions.
5. Please note Vocareum Submission deadline rules on page 26. Same rule applies to Canvas Submission. **Click submit** **on Vocareum** to submit your work.
6. Your solution document with the generated appendix can be found in the folder: TO\_BE\_DOWNLOADED\_AFTER\_PACKAGING/

Download the document and **submit** **on canvas**.

1. **Failure to comply may result in unnecessary mark deductions.**

**Question 1 a**

Note: I used BLUE highlighter to highlight codes that are relevant in some of the questions

|  |  |
| --- | --- |
| (i) | Name of changed HTML file: **base.html**  Code segment changed:  {% if not current\_user.is\_authenticated %}  <li class="nav-item"><a href="login" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-home text-light fa-lg mr-3"></i>Login</a></li>  <li class="nav-item"><a href="register" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-user text-light fa-lg mr-3"></i>Register</a></li>  <li class="nav-item"><a href="newlink" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-user text-light fa-lg mr-3"></i>New Link</a></li> |
| (ii) | Name of changed CSS file: custom.css  Code segment that is commented out for styling the links in the sidebar:  /\*.sidebar-link:hover {      background-color: #444;      border-radius: 5px;  }\*/  Code segment that is commented out for styling the Login card:  /\*.card-common:hover {      box-shadow: 2px 3px 15px #999;      transform: translateY(-1px);  }\*/ |

**Question 1 b**

|  |  |
| --- | --- |
| (i)  and  (ii) | **Tracing:**  The LOGIN view in Figure Q1.1(a) will transit to the DASHBOARD view in Figure Q1.2 (a) when the user login as admin@abc.com.  From **auth.py** file, refer to the highlighted code below,  @auth.route('/login', methods=['GET', 'POST'])  @auth.route('/')  def login():      # if current\_user.is\_authenticated == True:      #     return redirect(url\_for('dashboard.render\_dashboard'))      form = RegForm()      if request.method == 'POST':          print(request.form.get('checkbox'))          if form.validate():              check\_user = User.getUser(email=form.email.data)              if check\_user:                  if check\_password\_hash(check\_user['password'], form.password.data):                      login\_user(check\_user)                      return redirect(url\_for('dashboard.render\_dashboard'))                      #return redirect(url\_for('dashboard'))                  else:                      form.password.errors.append("User Password Not Correct")              else:                  form.email.errors.append("No Such User")  after user is authenticated, the user is redirected to the dashboard which is handled by the **dashboard.py** file.  But in this case, the [admin@abc.com](mailto:admin@abc.com) has additional options on the sidebar, namely an “upload” and “Log2” button and a dropdown menu. This is handled by the code found in **base.html**. Refer to the highlighted code below:  {% else %}  <li class="nav-item"><a href="dashboard" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-chart-area text-light fa-lg mr-3"></i>Dashboard</a></li>  <li class="nav-item"><a href="log" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Log</a></li>  {% if current\_user.email == "admin@abc.com" %}  <li class="nav-item"><a href="upload" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-cloud-upload-alt text-light fa-lg mr-3"></i>Upload</a></li>  <li class="nav-item"><a href="log2" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Log2</a></li>  {% block userDropdown %}  {% endblock %}  {% endif %}  Next, the DASHBOARD view in Figure Q1.2 (a) will transit to the LOGGING BMI 2 view in Figure Q1.2 (b) when the [admin@abc.com](mailto:admin@abc.com) user clicks on the Log2 button on the sidebar. From the code highlighted above from **base.html**, we can see that this line would reference the “Log2” button with the **log2.html**:  <li class="nav-item"><a href="log2" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Log2</a></li>  **Flask routing:**  Since the LOGIN view in Figure Q1.1(a) will transit to the DASHBOARD view in Figure Q1.2 (a) when the user login as [admin@abc.com](mailto:admin@abc.com), the flask route responsible is from **auth.py** file, refer to the highlighted code below,  @auth.route('/login', methods=['GET', 'POST'])  @auth.route('/')  def login():      # if current\_user.is\_authenticated == True:      #     return redirect(url\_for('dashboard.render\_dashboard'))      form = RegForm()      if request.method == 'POST':          print(request.form.get('checkbox'))          if form.validate():              check\_user = User.getUser(email=form.email.data)              if check\_user:                  if check\_password\_hash(check\_user['password'], form.password.data):                      login\_user(check\_user)                      return redirect(url\_for('dashboard.render\_dashboard'))                      #return redirect(url\_for('dashboard'))                  else:                      form.password.errors.append("User Password Not Correct")              else:                  form.email.errors.append("No Such User")  As highlighted above, when the user enters their email and password in the LOGIN page, the HTTP request is the **POST** request which triggers the login() function:  @auth.route('/login', methods=['GET', 'POST'])  @auth.route('/')  def login():  This is the HTTP response after the **POST** request:    As we can see in the HTTP response above, after the **POST** request, it triggers the HTTP **GET** request to the server after validation to retrieve the dashboard page. The code segment responsible for this is:  if check\_password\_hash(check\_user['password'], form.password.data):                      login\_user(check\_user)                      return redirect(url\_for('dashboard.render\_dashboard'))                      #return redirect(url\_for('dashboard'))  After the HTTP **GET** request, the user is then redirected to the dashboard page. The flask route responsible for this is from **dashboard.py** file which triggers the render\_dashboard() function. Refer to the highlighted code below,  # Only GET, /dashboard only produces the dashboard view    @dashboard.route('/dashboard')  @login\_required  def render\_dashboard():      return render\_template('dashboard.html', name=current\_user.name, panel="Dashboard")  Next, when the user clicks the Log2 button, the Flask route responsible to get to the LOGGING BMI 2 view in Figure Q1.2 (b) is found in the **bmi.py** file, log2() function:  # This following functions is for GET /log2 and POST /process via log2.html and log2.js  @bmi.route('/log2')  def log2():      all\_users = User.objects()      return render\_template('log2.html', name=current\_user.name, panel="Logging BMI 2", user\_list=all\_users)  The default method for route is the **GET** method. So, when the user clicks on the Log2 button, a **GET** request is sent to retrieve the **log2.html** file and the HTTP response is 200 which means that the server was able to process the request:    So once the **GET** request is sent for the **log2.html** file, we can see that the script reference is **log2.js file**:  {% extends "base.html" %}  {% block userDropdown %}  {% if panel == "Logging BMI 2" %}  <select  name="parameter" id="selected\_parameter" >    <option value="all">Select One</option>    {% for user in user\_list %}    <option value="{{user.email}}">{{user.email}}</option>    {% endfor %}  </select>  {% endif %}  <script src="{{ url\_for('static', filename='js/log2.js') }}"></script>  {% endblock %}  This triggers the HTTP **GET** request to retrieve log2.js and the server responds with 304: |
| (iii) | **Jinja: (for the second transition only)**  The Jinja codes that help to render the LOGGING BMI 2 view in Figure. Q1.2 (b) when a user action is applied on the DASHBOARD view starts in the first line of code in the **log2.html** and **dashboard.html** files, where they inherit from the "**base.html**" template and will include all of its content.:  {% extends "base.html" %}  In the **base.html** file we can see that for the Log2 option to appear, the user has to login as [admin@abc.com](mailto:admin@abc.com). This is handled by the following Jinja code found in **base.html**:  {% if current\_user.email == "admin@abc.com" %}  <li class="nav-item"><a href="upload" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-cloud-upload-alt text-light fa-lg mr-3"></i>Upload</a></li>  <li class="nav-item"><a href="log2" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Log2</a></li>  {% block userDropdown %}  Once it is authorized that the user [admin@abc.com](mailto:admin@abc.com) is logged in, the following Jinja code will be executed in **log2.html** file when the user presses the Log2 button which will render the LOGGING BMI 2 view in Figure Q1.2 (b):  {% extends "base.html" %}  {% block userDropdown %}  {% if panel == "Logging BMI 2" %}  <select  name="parameter" id="selected\_parameter" >    <option value="all">Select One</option>    {% for user in user\_list %}    <option value="{{user.email}}">{{user.email}}</option>    {% endfor %}  </select>  {% endif %}  <script src="{{ url\_for('static', filename='js/log2.js') }}"></script>  {% endblock %}  {% block mainblock %}      <div class="card-header">          <h3>BMI Calculator</h3>      </div>      <div class="card-body">          <div class="container-fluid col-sm-12">              <form action="/process2" method="post">              <!-- <input name="user\_email" type="hidden" id="user\_email" value=""> -->                  <div class="form-group row">                    <label for="email" class="col-sm-3 col-form-label">Email</label>                    <div class="col-sm-5">                      <input class="form-control" id="user\_email" placeholder="email" name="user\_email" required>                    </div>                  </div>                  <div class="form-group row">                    <label for="weight" class="col-sm-3 col-form-label">Weight</label>                    <div class="col-sm-5">                      <input class="form-control" id="weight" placeholder="Weight (kg)" name="weight" required>                    </div>                  </div>                  <div class="form-group row">                    <label for="height" class="col-sm-3 col-form-label">Height</label>                    <div class="col-sm-5">                      <input class="form-control" id="height" placeholder="Height (cm or m)" name="height" required>                    </div>                  </div>                   <div class="form-group row">                    <label for="spce" class="col-sm-3 col-form-label"></label>                      <div class="col-sm-5">                          <input type="radio" id="m" name="unit" value="m" checked>                          <label for="meter">m</label>                          <input type="radio" id="cm" value="cm" name="unit">                          <label for="centimeter">cm</label>                      </div>                  </div>                  <div class="form-group row">                      <label for="space" class="col-sm-3 col-form-label">Date & Time</label>                      <div class="col-sm-5">                        <input type="datetime-local" class="form-control" id="datetime" name="date" required>                      </div>                    </div>                  <div class="form-group row form-check">                          <button type="submit" class="offset-sm-3 btn btn-primary">Log BMI</button>                  </div>                  </form>              </div>      </div>  {% endblock %} |
| (iv) | **Flask Blueprint:**  The blueprints are that are responsible for the transition from Fig. Q1.2 (a) and then to Fig. Q1.2 (b) are found in:  **dashboard.py**:  dashboard = Blueprint('dashboard', \_\_name\_\_)  **bmi.py**:  bmi = Blueprint('bmi', \_\_name\_\_)  @bmi.route('/log2')  def log2():      all\_users = User.objects()      return render\_template('log2.html', name=current\_user.name, panel="Logging BMI 2", user\_list=all\_users)  @dashboard.route('/dashboard')  @login\_required  def render\_dashboard():      return render\_template('dashboard.html', name=current\_user.name, panel="Dashboard")  The functionality provided for each blueprint is that the Flask Blueprint is a way to organize a Flask application into reusable modules or components.  So, all relevant functions of authentication, BMI calculation and dashboard are put into separate files, **dashboard.py** and **bmi.py**. They are then all recalled into the **app.py** and when the Flask Blueprint is recalled, it **extend**s the functions in the separate python files. They are recalled in the **app.py** as follows:  # register blueprint from respective module  app.register\_blueprint(dashboard)  app.register\_blueprint(auth)  app.register\_blueprint(bmi)  From **auth.py**, we can see that the @auth.route decorator is used. This will associate the login() function to the ‘login’ and ‘/’ route:  @auth.route('/login', methods=['GET', 'POST'])  @auth.route('/')  def login():      # if current\_user.is\_authenticated == True:      #     return redirect(url\_for('dashboard.render\_dashboard'))      form = RegForm()      if request.method == 'POST':          print(request.form.get('checkbox'))          if form.validate():              check\_user = User.getUser(email=form.email.data)              if check\_user:                  if check\_password\_hash(check\_user['password'], form.password.data):                      login\_user(check\_user)                      return redirect(url\_for('dashboard.render\_dashboard'))                      #return redirect(url\_for('dashboard'))                  else:                      form.password.errors.append("User Password Not Correct")              else:                  form.email.errors.append("No Such User")  From the above code, we can also see that after successful authentication, the user will be redirected to the dashboard. This is executed with the highlighted code above.  The following render\_dashboard() function is then accessed in the **dashboard.py file** using the dashboard blueprint:  @dashboard.route('/dashboard')  @login\_required  def render\_dashboard():      return render\_template('dashboard.html', name=current\_user.name, panel="Dashboard")  This will trigger the transition from the login page to the page shown in Fig Q1.2 (a).  For the transition to Fig Q1.2(a), when the user clicks on the Log2 button the user is routed to /log2. The following blueprint is employed which is found in bmi.py:  @bmi.route('/log2')  def log2():      all\_users = User.objects()      return render\_template('log2.html', name=current\_user.name, panel="Logging BMI 2", user\_list=all\_users)  The @bmi.route decorator is used. This will associate the log2() function to the /log2 route which renders the **log2.htm**l template. This will trigger the transition to the page shown in Fig Q1.2 (b). |

**Question 1 c**

|  |  |
| --- | --- |
| (i) | **BEFORE** revision  Screenshot for failed Registration Session    Screenshot for failed Login Session |
| (ii) | Name of source file(s) revised: **auth.py**  The part of the code that was producing the error is highlighted below:  @auth.route('/register', methods=['GET', 'POST'])  def register():      form = RegForm()      if request.method == 'POST':          if form.validate():              existing\_user = User.getUser(email=form.email.data)              if existing\_user is None:                  hashpass = generate\_password\_hash(form.password.data, method='sha256')                  User.createUser(email=form.email.data,password=hashpass, name=form.name.data)                  return redirect(url\_for('auth.login'))              else:                  form.email.errors.append("User already existed")                  render\_template('register.html', form=form, panel="Register")        return render\_template('register.html', form=form, panel="Register")  The code is an if else statement that if the user being registered is not in the data base, it will not return an error. Else, if the user is already existing, it will throw the error “User already existed”  I will then change the error message to:  form.email.errors.append("User is already registered.")  Name of source file(s) revised: **auth.py**  The part of the code that was producing the error is highlighted below:  @auth.route('/login', methods=['GET', 'POST'])  @auth.route('/')  def login():      # if current\_user.is\_authenticated == True:      #     return redirect(url\_for('dashboard.render\_dashboard'))      form = RegForm()      if request.method == 'POST':          print(request.form.get('checkbox'))          if form.validate():              check\_user = User.getUser(email=form.email.data)              if check\_user:                  if check\_password\_hash(check\_user['password'], form.password.data):                      login\_user(check\_user)                      return redirect(url\_for('dashboard.render\_dashboard'))                      #return redirect(url\_for('dashboard'))                  else:                      form.password.errors.append("User Password Not Correct")              else:                  form.email.errors.append("No Such User")  The code is an if else statement that if the user logging in enters the correct password, it will not return an error. Else, if the user enters a wrong password, it will throw the error “User Password not correct error”  I will then change the error message to:  form.password.errors.append("User Password entered is incorrect.") |
| (iii) | **AFTER** revision  Screenshot for failed Registration Session    Screenshot for failed Login Session |

**Question 1 d**

|  |  |
| --- | --- |
| (i) | Jinja and JavaScript  Name of file that provides the available choices to the html page: **log2.js**  Show the relevant code:  $("#selected\_parameter").change(function(){        debugger        let parameter=$(this).val();        // Set the hidden parameter by the user's email whose BMI is logged        var hiddenElement = document.getElementById("user\_email");        hiddenElement.setAttribute("value", parameter)    })  Inside this function, there is an event listener that waits for the "change" event to occur on an HTML element with the ID "selected\_parameter".  When this event occurs, the code inside the event handler function is executed. First, the val() function is called on the $(this) object, which refers to the element that triggered the event (in this case, the "selected\_parameter" element). This retrieves the value of the selected option from a dropdown list and assigns it to the variable parameter.  The code then sets the value of HTML input element with the ID "user\_email" to the value of parameter. This is achieved by getting a reference to the element using document.getElementById(), and then calling the setAttribute() method to set its value.  The html page where the dropdown list is being populated: **log2.html**  Show the relevant code:  {% extends "base.html" %}  {% block userDropdown %}  {% if panel == "Logging BMI 2" %}  <select  name="parameter" id="selected\_parameter" >    <option value="all">Select One</option>    {% for user in user\_list %}    <option value="{{user.email}}">{{user.email}}</option>    {% endfor %}  </select>  {% endif %}  <script src="{{ url\_for('static', filename='js/log2.js') }}"></script>  {% endblock %}  The {% block userDropdown %} line defines a block of content that can be overridden in the child template. This Jinja code is inherited in the **base.html** file where the dropdown menu will appear only if the [admin@abc.com](mailto:admin@abc.com) user is logged in:  {% if current\_user.email == "admin@abc.com" %}  <li class="nav-item"><a href="upload" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-cloud-upload-alt text-light fa-lg mr-3"></i>Upload</a></li>  <li class="nav-item"><a href="log2" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Log2</a></li>  {% block userDropdown %}  Referring back to **log2.html**, the {% if panel == "Logging BMI 2" %} line is a conditional statement that checks whether a variable in the side panel is equal to the string "Logging BMI 2". If it is, then the content inside the block will be displayed.  Inside the block, there is an HTML select element that creates a dropdown list with a name "parameter" and an ID "selected\_parameter". The options in the dropdown list are populated from a loop that iterates over a list of "user" objects from the "user\_list" variable. The value of each option is set to the email address of the user, and the label is also set to their email.  When the user interacts with the dropdown list by selecting an option, the JavaScript code in "log2.js" is executed.  The JavaScript code listens for changes to the "selected\_parameter" element using the $("#selected\_parameter").change() function and retrieves the selected value using $(this).val(). It then sets the value of a hidden input element called "user\_email" to the selected value.  When the user submits the form, the value of the "user\_email" input element is included in the BMI Calculator form. |
| (ii) | Flask routing:  From the **log2.html**, we can see the form element that specifies the action and method for submitting form data to a server.  {% block mainblock %}      <div class="card-header">          <h3>BMI Calculator</h3>      </div>      <div class="card-body">          <div class="container-fluid col-sm-12">              <form action="/process2" method="post">  The action attribute specified the URL where the form data will be sent when the form is submitted. In this case, it is set to "/process2.  The method attribute specifies the HTTP method to use when submitting the form data. In this case, the form data will be sent as part of an HTTP POST request.  When a user submits the form by clicking a submit button or pressing Enter in a form field, the browser will send an HTTP POST request to the URL specified in the action attribute, with the form data included in the request body. The server can then handle the request by processing the form data and generating an appropriate response.  The HTTP request made when the user clicks on the Log BMI button is POST.  From **bmi.py,** we can see the Python Flask route that handles a POST request to the '/process2' endpoint. When a client submits a form with method 'POST' to this endpoint, the data from the form is sent to the server as part of the request body.:  @bmi.route('/process2',methods= ['POST'])  def process2():      # Get the parameters posted by form in log2.html      weight  = float(request.form['weight'])      height = float(request.form['height'])      unit = request.form['unit']      user\_email = request.form['user\_email']      date = request.form['date']      datetime\_object = datetime.strptime(date, '%Y-%m-%dT%M:%S')      date\_object= datetime\_object.date()  The http response is 302 Found. |
| (iii) | Model:  Identify the TWO models related to the data/results submitted in Fig. 1.3(b).  Model 1: bmilog  Model 2: bmidaily   |  |  | | --- | --- | | **Name of Model** | **Source file** | | bmilog | bmilog.py  From bmi.py, we can see:  bmilogObject = BMILOG.createBMILOG(user=existing\_user, datetime=datetime\_object, weight=weight, height=height, unit=unit, bmi=0.0)  This code creates a new BMILOG object with the user's information and stores it in the BMILOG model. | | bmidaily | bmidaily.py  From bmi.py, we can see:  BMIDAILY.createBMIDAILY(existing\_user, date\_object, 1, bmilogObject.bmi)  This line creates a new BMIDAILY object with the user and date provided and stores it in the BMIDAILY model. | |

**Question 1 e**

|  |  |
| --- | --- |
| (i) | Flask routing Before AJAX call:  The HTTP request made when the “Click to BMI chart (backend)” link is clicked is the **GET** request before the AJAX call was made.  The following code in **dashboard.py** is relevant for making the HTTP **GET** request.  # chart2 GET and POST act in tandum, POST done via myChart\_CSV2.js  @dashboard.route('/chart2', methods=['GET', 'POST'])  def chart2():      if request.method == 'GET':          #I want to get some data from the service          return render\_template('bmi\_chart2.html', name=current\_user.name, email\_id=current\_user.email, panel="BMI Chart")    #do nothing but to show index.html  The function chart2() executes after a GET or POST request is sent to the URL endpoint '/chart2'.  Since the request method is GET, it will render the **bmi\_chart2.html** template with the name, email\_id, and panel variables passed to it using the render\_template() function, as highlighted above. The name and email\_id variables are obtained from the current\_user object, which is assumed to be available if the user is logged in. The panel variable is set to "BMI Chart".  The HTTP response associated to this HTTP **GET** request is 200: |
| (ii) | AJAX: call  Name of JavaScript file: **myChart\_CSV2.js**  The purpose of this AJAX call is to send a HTTP POST request to the /chart2 endpoint and receive a response containing data that will be used to render a chart.  The success function is called when the HTTP request is successful. It receives the data returned from the server, along with the status and xhr objects:  success: function(data, status, xhr  The code then extracts relevant data from the data object, which contains a chartDim property and a labels property.  var chartDim = data.chartDim;      var xLabels = data.labels;  The chartDim property contains an object whose keys are labels, such as 'usr\_1', 'usr\_2', etc., and whose values are arrays of 2-element arrays, each containing a timestamp (as a string) and a numerical value. The labels property is an array of labels that correspond to the chartDim object.  The code then creates a new Chart object using the Chart.js library, passing in a ctx object.  var myChart = new Chart(ctx, {        data: {        // labels: xLabels,        datasets: []        },  The options parameter specifies that the x-axis should display dates and the y-axis should display numerical values.  scales: {            x: {              type: 'time',              time: {                // "parser": "MM/DD/YYYY HH:mm",                parser: 'yyyy-MM-dd',              },              scaleLabel: {                display: true,                labelString: 'Date'              }            },            y: {              scaleLabel: {                display: true,                labelString: 'value'              }            }          }  The code then loops through each label in chartDim, creating a new dataset for each one and pushing it to the Chart object. Each dataset is a line graph with the label property set to the label, a randomly generated border color, a background color, and an array of data objects, where each data object is an object with x and y properties representing a timestamp and a numerical value, respectively.  for (i= 0; i < vLabels.length; i++ ) {        myChart.data.datasets.push({        label: vLabels[i], // Flight#        type: "line",        // borderColor: '#'+(0x1ff0000+Math.random()\*0xffffff).toString(16).substr(1,6),        borderColor: '#'+(0x1100000+Math.random()\*0xffffff).toString(16).substr(1,6),        backgroundColor: "rgba(249, 238, 236, 0.74)",        data: vData[i],        spanGaps: true        });  Finally, the Chart object is updated to display the new datasets on the canvas element.  myChart.update(); |
| (iii) | Flask routing on AJAX call and after AJAX call returns:  The HTTP request made when the “Click to BMI chart (backend)” link is clicked is the **POST** request after the AJAX call was made.  The following code in **dashboard.py** is relevant for making the HTTP **POST** request.  @dashboard.route('/chart2', methods=['GET', 'POST'])  def chart2():      if request.method == 'GET':          #I want to get some data from the service          return render\_template('bmi\_chart2.html', name=current\_user.name, email\_id=current\_user.email, panel="BMI Chart")    #do nothing but to show index.html      elif request.method == 'POST':            # Retrieve data from AJAX POST          res = request.get\_data("data")          d\_token = json.loads(res)          email\_id = d\_token['email\_id']            # if it is admin, all BMIDAILY records are to be charted          if email\_id == "admin@abc.com":              email\_id = None            chartDim, labels = getChartDim(user\_email=email\_id)            return jsonify({'chartDim': chartDim, 'labels': labels})  The function chart2() executes after a GET or POST request is sent to the URL endpoint '/chart2'.  Since the request method is POST, the function retrieves some data from the AJAX POST request, which is passed as a parameter to the request.get\_data() method. The retrieved data is expected to be a JSON object that contains an email ID.  The function then uses this email ID to retrieve data from the service using the getChartDim() function. The getChartDim() function returns the chart dimension and labels based on the email ID. If the email ID is "admin@abc.com", then all records are charted.  Finally, the function returns a JSON response that contains the chart dimension and labels. The chart dimension is an array of arrays that represents the data points of the chart, and the labels represent the x-axis labels of the chart.  The HTTP response associated to this HTTP POST **request** is 200: |

**Question 2**

|  |  |
| --- | --- |
| a. | Which function are you making the changes for?  (specify either Login/Register/Logout function)  Frontend change #1: From the **custom.css** file, I changed the following highlighted code to match the image and gradient of the **sidebar** similar to FigureQ2.1 (a),  /\* navbar \*/  .sidebar {      height: 100vh;      background: linear-gradient(rgba(136, 175, 252, 0.548), rgba(0, 0, 0, .9)), url(../img/golf\_swing\_stock\_1920.jpg);      background-position: center;      background-repeat: no-repeat;      background-size: cover;      box-shadow: 5px 7px 25px #999;  }  Frontend change #2: From the **custom.css** file, I changed the following highlighted code to match the image and gradient of the **top navigation bar** similar to FigureQ2.1 (a),  .top-navbar {      position: static;      background: linear-gradient(rgba(170, 187, 222, 0.548), rgba(98, 98, 192, 0.5)), url(../img/golf\_swing\_stock\_1920.jpg);;  }  Frontend change #3: From the **base.html** file, the following highlighted code are removed as they are no longer needed as they do not appear in the sidebar as shown in Figure Q2.2 (a) and (b),  {% if not current\_user.is\_authenticated %}                  <li class="nav-item"><a href="login" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-home text-light fa-lg mr-3"></i>Login</a></li>                  <li class="nav-item"><a href="register" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-user text-light fa-lg mr-3"></i>Register</a></li>                  <!--<li class="nav-item"><a href="newlink" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-user text-light fa-lg mr-3"></i>New Link</a></li>-->                  {% else %}                  {% if current\_user.email == "admin@abc.com" %}                  <li class="nav-item"><a href="upload" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-cloud-upload-alt text-light fa-lg mr-3"></i>Upload</a></li>                  <!--{% block userDropdown %}                  {% endblock %}-->                  {% endif %}                  <!--<li class="nav-item"><a href="dashboard" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-chart-area text-light fa-lg mr-3"></i>Dashboard</a></li>-->                  <!--<li class="nav-item"><a href="log" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Log</a></li>-->                  <li class="nav-item"><a href="swing" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Swing</a></li>                  <li class="nav-item"><a href="swing\_chart" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Swing Chart</a></li>                {% endif %}  From here, I added some codes as highlighted below to match the ‘Upload’, ‘Swing’ and ‘Swing Chart’ options on the side bar as shown in Figure Q2.2 (a) and (b). I also rearranged the codes to match the order of the options:  {% if not current\_user.is\_authenticated %}                  <li class="nav-item"><a href="login" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-home text-light fa-lg mr-3"></i>Login</a></li>                  <li class="nav-item"><a href="register" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-user text-light fa-lg mr-3"></i>Register</a></li>                  {% else %}                  {% if current\_user.email == "admin@abc.com" %}                  <li class="nav-item"><a href="upload" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-cloud-upload-alt text-light fa-lg mr-3"></i>Upload</a></li>                  {% endif %}                  <li class="nav-item"><a href="log2" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Swing</a></li>                  {% block userDropdown1 %}                  {% endblock %}                  {% block userDropdown2 %}                  {% endblock %}                  <li class="nav-item"><a href="/swingchart" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-chart-area text-light fa-lg mr-3"></i>Swing Chart</a></li>                  {% endif %}  Backend change #1: After login, the user will be redirected to the dashboard page. Since we no longer need the contents of the dashboard, in the **dashboard.py file**, I removed the title at the top panel for now:  @dashboard.route('/dashboard')  @login\_required  def render\_dashboard():      return render\_template('dashboard.html', name=current\_user.name, panel="")  Frontend change #4: I removed the following highlighted code from the **dashboard.html** file, since we do not want to display anything on the /dashboard route after logging in for now:  {% extends "base.html" %}  <body>  {% block mainblock %}      <!--<div class="card-header">          <h1>Dashboard</h1>      </div>      <div class="card-body">          <p>You are currently logged in as {{ name }}</p>          <div class="linkChart">              <ul>                  <li>Frontend Chart: <a href="/chart">Click to BMI Chart (Frontend)</a></li>                  <li>Backend Chart: <a href="/chart2">Click to BMI Chart (Backend)</a> </li>                  <li>Average Chart: <a href="/chart3">Click to Average BMI Chart (Backend)</li>              </ul>          </div>      </div>-->  {% endblock %}  </body>  </html> |
| bi. | Name of file: **golfsetData.py**   |  |  | | --- | --- | | Name of class/model | Purpose | | ClubHead(db.Document) | 1. **‘meta = {'abstract' : True, 'allow\_inheritance': True}’**: This sets some metadata about the class. ‘**abstract=True’** means that this class is abstract, and cannot be instantiated directly. Instead, it serves as a parent class for other classes that will inherit from it. ‘**allow\_inheritance=True’** means that any subclass can be saved to the database in a collection that contains all of the subclasses and the parent class. 2. **‘loft = db.FloatField()’:** This creates a ‘**loft’** field in the database collection for this class, and sets its type to a ‘**float’** value. This means that when an object of this class (or any subclass) is saved to the database, the ‘**loft’** value will be stored as a floating-point number. 3. **‘weight = db.IntField()’:** This creates a ‘**weight**’ field in the database collection for this class, and sets its type to an integer value. This means that when an object of this class (or any subclass) is saved to the database, the ‘**weight’** value will be stored as an integer. 4. ‘**def getLoft(self):**’: This is a method that returns the ‘**loft’** value of an object of this class (or any subclass). 5. ‘**def getMaterialClass(self):**’: This is a method that returns the name of the class of the object that calls it. | | WoodHead(ClubHead) | 1. The ‘**meta’** attribute defines metadata about how the model should be stored in a database, and the ‘**createWoodHead’** method is a static method that creates new instances of the ‘**WoodHead’** class. 2. ‘**WoodHead’** is a subclass of ‘**ClubHead’**, which means it inherits all the fields and methods from ‘**ClubHead’**, but adds an additional field ‘**size’** to the model. 3. The ‘**size’** field is defined using ‘**db.IntField()**’, which means it is an integer field in the database. 4. The ‘**createWoodHead’** method is a static method that takes three arguments: ‘**loft’**, ‘**weight**’, and ‘**size**’. It first tries to find an existing ‘**WoodHead**’ object with the same ‘**loft**’ and ‘**weight**’ values in the database. If it finds one, it returns that object. If it doesn't find one, it creates a new ‘**WoodHead**’ object with the given ‘**loft**’, ‘**weight’**, and ‘**size’** values, and saves it to the database. It then returns the newly created object. 5. The ‘**getSize’** method is an instance method that returns the value of the ‘**size** field for a given ‘**WoodHead’** object. | | IronHead(ClubHead) | 1. ‘**IronHead’** is a subclass of ‘**ClubHead’**, which means it inherits all the fields and methods from ‘**ClubHead’**, but adds an additional field ‘**material’** to the model. 2. The ‘**material’** field is an attribute of the ‘**IronHead’** class, which indicates the type of material used to make the clubhead. It is stored as a ‘**StringField’** in the database. 3. The ‘**meta’** field is a dictionary that contains metadata about the ‘**IronHead’** class. In this case, it specifies the name of the collection in the database where instances of this class will be stored. 4. The ‘**createIronHead’** method is a static method that creates a new instance of the ‘**IronHead’** class, if one does not already exist in the database, with the specified ‘**loft’**, ‘**weight’**, and ‘**material’** properties. If an instance already exists with the specified ‘**loft’** and ‘**weight’**, it returns that instance. | | PutterHead(ClubHead) | 1. ‘**PutterHead’** is a subclass of ‘**ClubHead’**, which means it inherits all the fields and methods from ‘**ClubHead’**, but adds an additional field ‘**style’** to the model. 2. The ‘**meta’** field is a dictionary that contains metadata about the ‘**PutterHead’** class. In this case, it specifies the name of the collection in the database where instances of this class will be stored. 3. The ‘**style’** field is an attribute of the ‘**PutterHead’** class. It uses the ‘**StringField’** type which is stored in the database. 4. ‘**createPutterHead’** is a method of the ‘**PutterHead’** class that creates a new ‘**PutterHead’** object with the specified ‘**loft’**, ‘**weight’**, and ‘**style’** values. If an instance already exists with the specified ‘**loft’** and ‘**weight’**, it returns that instance. 5. ‘**getStyle’** method of the ‘**PutterHead’** class that returns the ‘**style’** value of a ‘**PutterHead’** object. | | Shaft(db.Document) | 1. The ‘**meta’** attribute is a dictionary that defines the name of the MongoDB collection where the ‘**Shaft’** documents will be stored. In this case, the collection is called **'shafts'**. 2. The class has four data fields: ‘**length’**, ‘**weight’**, ‘**material’**, and ‘**flex’**. These fields correspond to the physical properties of a golf club shaft. 3. Each field is defined using a specific type of **db.Field** (e.g., **FloatField**, **IntField**, **StringField**). This ensures that the data stored in each field has the appropriate data type. 4. The ‘**createShaft’** method is a static method that creates a new ‘**Shaft’** document in the database. It takes four arguments (‘**length’**, ‘**weight’**, ‘**material’**, and ‘**flex’**) and returns the created document. It first checks if a document with the same values already exists in the database by querying the ‘**Shaft’** collection with the ‘**length’**, ‘**weight’**, ‘**material’**, and ‘**flex’** fields. If a document is found, the method returns it. If no document is found, the method creates a new ‘**Shaft’** document with the given values and saves it to the database. 5. The ‘**getLength’** method is an instance method that returns the ‘**length’** field of a ‘**Shaft’** document. | | Grip(db.Document) | 1. The ‘**meta’** attribute is a dictionary that defines the name of the MongoDB collection where the ‘**Shaft’** documents will be stored. In this case, the collection is called **'grip'**. 2. The ‘**diameter’**, ‘**weight’**, and ‘**material’** fields define the schema for the ‘**Grip’** model. Each of these fields has a different data type and is used to store different types of information about the grip objects. 3. In this case, the ‘**createGrip’** method is defined to take three parameters: ‘**diameter’**, ‘**weight’**, and ‘**material’**. A query is executed to find an existing ‘**Grip’** object with the specified ‘**diameter’**, ‘**weight’**, and ‘**material’** values. If a document is found, the method returns it If no matching object is found, a new ‘**Grip’** object is created and saved to the database. | | Club(db.Document) | 1. This code defines a model called ‘**Club’**. The model contains fields for ‘**label’**, ‘**head’**, ‘**shaft’**, and ‘**grip’**, all of which are reference fields that point to the other models. 2. The ‘**meta’** attribute is a dictionary that defines the name of the MongoDB collection where the ‘**Shaft’** documents will be stored. In this case, the collection is called **'clubs'**. 3. **‘label’** field: a string field that stores the label of the club. 4. **‘head’** field: a reference field that points to an instance of the ‘**ClubHead’** model. 5. **‘shaft’** field: a reference field that points to an instance of the ‘**Shaft’** model. 6. **‘grip’** field: a reference field that points to an instance of the ‘**Grip’** model 7. **‘createClub’** is a method that creates a new ‘**Club’** instance from a list of data. This method first checks if a ‘**Club’** instance with the same label already exists in the database. If it does not exist, it creates new instances of ‘**WoodHead’**, ‘**IronHead’**, or ‘**PutterHead’** (depending on the type of club), ‘**Shaft’**, and ‘**Grip’**, and then creates a new ‘**Club’** instance with these objects as fields. If a ‘**Club’** instance with the same label already exists, it returns that instance instead. 8. **‘getClubsize()’**: a method that retrieves the size of the club's head by calling the ‘**getSize()’** method on the ‘**ClubHead’** instance that the ‘**head’** field points to. 9. **‘getClubstyle()’**: a method that retrieves the style of the club's head by calling the ‘**getStyle()’** method on the ‘**ClubHead’** instance that the ‘**head’** field points to. 10. **‘getShaftlength()’**: a method that retrieves the length of the club's shaft by calling the ‘**getLength()’** method on the ‘**Shaft’** instance that the ‘**shaft’** field points to. 11. **‘getClubHeadloft()’**: a method that retrieves the loft of the club's head by calling the ‘**getLoft()’** method on the ‘**ClubHead’** instance that the ‘**head’** field points to. 12. **‘getClubMaterial()’**: a method that retrieves the material of the club's head by calling the ‘**getMaterialClass()’** method on the ‘**ClubHead’** instance that the ‘**head’** field points to. | | GolfSet(db.Document) | 1. This code defines a model called ‘**GolfSet’**. The model contains fields for ‘**golfer’**, which is an instance of the **User** class, and ‘**clubs’** which is a dictionary. 2. The ‘**meta’** attribute is a dictionary that defines the name of the MongoDB collection where the ‘**GolfSet’** documents will be stored. In this case, the collection is called **'golfSets'**. 3. **‘createGolfSet’** method takes an email address as an argument and uses it to find an instance of the ‘**User’** class. If a user is found, it checks if that user already has a ‘**GolfSet’** document in the database. If not, it creates a new ‘**GolfSet’** instance and saves it to the database. The method then returns the ‘**GolfSet’** instance that was found or created. 4. **‘addClub’** is a method which takes a ‘**club’** object as an argument, which is an instance of ‘**Club’** class. If the ‘**club.label’** is already in the ‘**clubs’** dictionary, the method returns **False**. Otherwise, it adds the ‘**club’** object to the ‘**clubs’** dictionary with ‘**club.label’** as the key. It then saves the modified ‘**GolfSet’** instance to the database and returns **True**. 5. **‘getClub’** is a method that takes a ‘**label’** string as an argument and returns the value associated with that key in the **clubs** dictionary. If the key is not found, it returns **None**. 6. **‘getAllClubs’** is a method that returns the entire ‘**clubs’** dictionary. 7. **‘getGolfSetByEmail’** is a method that takes an email address as an argument and uses it to find an instance of the ‘**User’** class. If a user is found, it returns the ‘**GolfSet’** instance associated with that user. If no user is found, it returns **None**. |   # Python code in text format for the model(s):  from app import db  from models.users import User  import math  class ClubHead(db.Document):      meta = {'abstract' : True, 'allow\_inheritance': True}      loft = db.FloatField()      weight = db.IntField()      def getLoft(self):          return self.loft      def getMaterialClass(self):          return type(self).\_\_name\_\_  class WoodHead(ClubHead):      meta = {'collection' : 'woodheadClub'}      size = db.IntField()        @staticmethod      def createWoodHead(loft, weight, size):          woodHead = WoodHead.objects(loft=loft, weight=weight).first()          if not woodHead:               woodHead = WoodHead(loft=loft, weight=weight, size=size).save()          return woodHead      def getSize(self):          return self.size  class IronHead(ClubHead):      meta = {'collection' : 'ironheadClub'}      material = db.StringField()        @staticmethod      def createIronHead(loft, weight, material):          ironHead = IronHead.objects(loft=loft, weight=weight).first()          if not ironHead:               ironHead = IronHead(loft=loft, weight=weight, material=material).save()          return ironHead  class PutterHead(ClubHead):      meta = {'collection' : 'putterheadClub'}      style = db.StringField()        @staticmethod      def createPutterHead(loft, weight, style):          putterHead = PutterHead.objects(loft=loft, weight=weight).first()          if not putterHead:               putterHead = PutterHead(loft=loft, weight=weight, style=style).save()          return putterHead      def getStyle(self):          return self.style    class Shaft(db.Document):      meta = {'collection': 'shafts'}      length = db.FloatField()      weight = db.IntField()      material = db.StringField()      flex = db.StringField()      @staticmethod      def createShaft(length, weight, material, flex):          shaft = Shaft.objects(length=length, weight=weight, material=material, flex=flex).first()          if not shaft:              shaft = Shaft(length=length, weight=weight, material=material, flex=flex).save()          return shaft      def getLength(self):          return self.length  class Grip(db.Document):      meta = {'collection': 'grip'}      diameter = db.FloatField()      weight = db.IntField()      material = db.StringField()      @staticmethod      def createGrip(diameter, weight, material):          grip = Grip.objects(diameter=diameter, weight=weight, material=material).first()          if not grip:              grip = Grip(diameter=diameter, weight=weight, material=material).save()          return grip  class Club(db.Document):      meta = {'collection': 'clubs'}      label = db.StringField()      head = db.ReferenceField(ClubHead)      shaft = db.ReferenceField(Shaft)      grip = db.ReferenceField(Grip)      @staticmethod      def createClub(dataList):          club = Club.objects(label=dataList[0]).first()          if not club:              if dataList[1].strip() == "Wood":                  head = WoodHead.createWoodHead(float(dataList[2]), int(dataList[3]), int(dataList[4]))              elif dataList[1].strip() == "Iron":                  head = IronHead.createIronHead(float(dataList[2]), int(dataList[3]), str(dataList[4]))              elif dataList[1].strip() == "Putter":                  head = PutterHead.createPutterHead(float(dataList[2]), int(dataList[3]), str(dataList[4]))              shaft = Shaft.createShaft(float(dataList[5]), int(dataList[6]), str(dataList[7]), str(dataList[8]))              grip = Grip.createGrip(float(dataList[9]), int(dataList[10]), str(dataList[11]))              club = Club(label=dataList[0], head=head, shaft=shaft, grip=grip).save()            return club      def getClubsize(self):          return self.head.getSize()      def getClubstyle(self):          return self.head.getStyle()      def getShaftlength(self):          return self.shaft.getLength()      def getClubHeadloft(self):          return self.head.getLoft()      def getClubMaterial(self):          return self.head.getMaterialClass()  class GolfSet(db.Document):      meta = {'collection': 'golfSets'}      golfer = db.ReferenceField(User)      clubs = db.DictField()        @staticmethod      def createGolfSet(email):          golfer = User.objects(email=email).first()          if golfer:              golfSet = GolfSet.objects(golfer=golfer).first()              if not golfSet:                  golfSet = GolfSet(golfer=golfer, clubs={}).save()              return golfSet      def addClub(self,club):          if club.label in self.clubs: #if club is found in the clubs dictionary              return False          self.clubs[club.label] = club          self.save()          return True      def getClub(self,label):          return self.clubs.get(label)      def getAllClubs(self):          return self.clubs      @staticmethod      def getGolfSetByEmail(email):          golfer = User.getUser(email)          if golfer:              return GolfSet.objects(golfer=golfer).first()          else:              return None |
| ii | Name of file: **golfsetData.py**   |  |  | | --- | --- | | **Name of class/model** | **Purpose** | | Swing(db.Document) | 1. The ‘**meta’** attribute is a dictionary that defines the name of the MongoDB collection where the ‘**Swing’** documents will be stored. In this case, the collection is called **‘swings’**. 2. **‘golfer’** is a reference field, which is used to store a reference to the ‘**User’** model representing the golfer who performed the swing. 3. **‘swing\_datetime’** is a DateTimeField, which is used to store the date and time when the swing was performed. 4. **‘club’** is a reference field, which is used to store a reference to the ‘**Club’** model representing the club used to perform the swing. 5. **‘swingSpeed’** is a FloatField, which is used to store the swing speed in miles per hour. 6. **‘distance’** is a FloatField, which is used to store the estimated distance of the swing. 7. **‘createSwing’** is a method used to create a new ‘**Swing’** document in the MongoDB collection. It takes the parameters representing the ‘**golfer’**, ‘**date and time’** of the swing, ‘**club’** used, ‘**swing speed’**, and ‘**estimated distance’** of the shot, and creates a new ‘**Swing’** document with these values if it does not already exist. 8. **computeDistance** is a method used to calculate the estimated distance of a shot based on the club used and swing speed. It takes the ‘**Club’** model document representing the club used and the swing speed in miles per hour as input, and returns the estimated distance of the shot as per the given formula in the question. |   # Python code in text format for the model(s):  class Swing(db.Document):      meta = {'collection': 'swings'}      golfer = db.ReferenceField(User)      swing\_datetime = db.DateTimeField(['%Y-%m-%d %h:%M'])      club = db.ReferenceField(Club)      swingSpeed = db.FloatField()      distance = db.FloatField()        @staticmethod      def createSwing(golfer, swing\_datetime, club, swingSpeed, distance):          swing = Swing.objects(golfer=golfer, swing\_datetime=swing\_datetime, club=club, swingSpeed=swingSpeed, distance=distance).first()          if not swing:              swing = Swing(golfer=golfer, swing\_datetime=swing\_datetime, club=club, swingSpeed=swingSpeed, distance=distance).save()              print("added")          return swing      @staticmethod      def computeDistance(club,swingSpeed):          print(club.getClubMaterial())          #compute club\_head\_height          if club.getClubMaterial() == "WoodHead":              club\_head\_height = club.getClubsize() / 400          elif club.getClubMaterial() == "IronHead":              club\_head\_height = 1          elif club.getClubMaterial() == "PutterHead":              if club.getClubstyle() == "Blade":                  club\_head\_height = 1              else:                  club\_head\_height = 0.5          else:              return False          #compute club\_length          club\_length = club\_head\_height + club.getShaftlength()            #compute estimated\_distance          club\_head\_loft = club.getClubHeadloft()          estimated\_distance = (280 - abs(48- int(club\_length))\*10 - abs(int(club\_head\_loft) - 10)\*1.25) \* int(swingSpeed)/96          return estimated\_distance |
| iii | **Upload function**  # **Frontend** component code segment #1 – In the **upload.html** file, I changed the header to display “Upload Recordings” with this line of code:  <h2>Upload recordings</h2>  # **Frontend** component code segment #2 – In the **upload.html** file, I added a dropdown menu where the user can select the golfer. The first option in the dropdown menu has no value and the text "Select One User", indicating to the user that they need to choose a user from the list. Here is snippet for that code:  <!--drop down menu to choose Golfer-->                  <label>Upload for Golfer:</label>                  <select name="users" id="userSelect">                      <option value = ""> Select One User</option>                      {%for aUser in Users%}                      <option value="{{aUser.email}}">{{aUser.name}} </option>                      {%endfor%}                  </select>  From the code above, the ‘for loop’ iterates over the list of users. For each user, an option element is created with a value equal to the user's email and the text equal to the user's name. The option elements are then added to the dropdown menu.  # **Frontend** component code segment #2 – In the **upload.html** file, I added a dropdown menu where the user can select the file type to upload. The first option in the dropdown menu has a value of "GolfSet", indicating to the user that they can choose a file containing data about golf sets. The second option in the dropdown menu has a value of "Swings", indicating to the user that they can choose a file containing data about swings. This option can then be used to determine which type of file the user wants to upload and how the data in that file should be processed. Here is the code:  <!--drop down menu to choose file type-->                  <label>with data type of:</label>                  <select name="datatype" id="datatypeSelect">                      <option value = "GolfSet"> Golf Set</option>                      <option value = "Swings"> Swings</option>                  <label>Choose a car for <b id="chosen\_email"></b>:</label>                  </select>  # **Frontend** component code segment #3 – In the **upload.html** file, I added a paragraph information about the data that should be included in the file. Similar to what was shown in Figure Q2.3(b). Here is the code snippet:  <!--paragraph information-->                  <p>                  Golf Set Line:<br>                  label,clubtype,loft,weight,size/material/style,length,weight,material,flex,diameter,weight,material<br><br>                    Swing Line:<br>                  swing\_time,swing\_speed,club\_label<br>                  </p>  # **Frontend** component code segment #4 – In the **upload.html** file, the file input element is created with an id of "upload", a name of "file", a type of "file", and an "accept" attribute that restricts the file types to CSV and TXT files. The "required" attribute ensures that the user must select a file before submitting the form.  The input element with type "submit" is created with a value of "Upload" and a type of "Upload". When the user clicks this button, the form will be submitted, and the selected file will be uploaded to the server. Here is the code snippet:  !--Choose file and upload-->                  <label for='upload'>Upload CSV or txt file</label>                  <input class="upload" id='upload' name='file' type='file' accept='.csv, .txt' required>              </div>              <div>                  <input type="submit" value="Upload" type="Upload"/>  # **Backend** component code segment **#1** – In the **app.py** file, I created a route to the upload page. The GET request renders the **upload.html** template with the **name** and **panel** variables passed in for display purposes. The **Users** variable is also passed in for the Jinja template to populate a dropdown list of available users. Here is the code snippet:  def upload():      if request.method == 'GET':          return render\_template("upload.html", name=current\_user.name, panel="Upload", Users=User.getAllUsers()) #added Users=User.getAllUsers() for Jinja reference in upload.html  # **Backend** component code segment **#2** – Continuing from the code above, in the **app.py** file, The POST request is triggered when the user submits the form. It first retrieves the selected data type (either "GolfSet" or "Swings") and the uploaded file from the request. Then, it creates a **GolfSet** object for the selected user using the **createGolfSet()** method. Here is the code snippet:  elif request.method == 'POST':          type = request.form.get('type')          file = request.files.get('file')          data = file.read().decode('utf-8')            aDataType = request.form.get("datatype")          aEmail = request.form.get("users")            #create GolfSet          golfset = GolfSet.createGolfSet(aEmail)  If the selected type is "upload", it reads the file using **csv.reader()** or **csv.DictReader()** depending on the data type. For each row in the file, it creates a **Club** object or a **Swing** object depending on the data type, and adds the club to the golf set or saves the swing data to the database.  The **Club** and **Swing** objects are created using methods defined in their respective classes. For example, **Club.createClub()** creates a **Club** object using the data in a row of the file. While the **Swing.createSwing()** creates a **Swing** object. This will send the objects created to the respective clubs and swings database. The uploaded file is read and parsed using the **csv** module, and the data is converted to appropriate types as needed. Finally, the **upload.html** template is rendered again with the **name**, **panel**, and **Users** variables passed in. Below is the code snippet:  elif type == 'upload':              #if data type is GolfSet              if aDataType == "GolfSet":                  #readfile                  dict\_reader = csv.reader(io.StringIO(data), delimiter=',', quotechar='"')                  file.close()                    for item in dict\_reader: #returns list of each row in .txt file                      print(item)                      #create club in database                      club = Club.createClub(item)                      #add club to golf set                      golfset.addClub(club)              #if data type is Swings              elif aDataType == "Swings":                  #readfile                  dict\_reader = csv.DictReader(io.StringIO(data), delimiter=',', quotechar='"')                  file.close()                  for item in list(dict\_reader): #returns list of each row in .txt file                      print(item)                      golfer = User.getUser(aEmail)                      date\_time = item['swing\_time'].replace('"', '')                      swing\_datetime = datetime.strptime(date\_time, "%Y-%f-%dT%I:%M")                      club = golfset.getClub(item['club\_label'])                      swingSpeed = item['swing\_speed']                      distance = Swing.computeDistance(club,swingSpeed)                      print(distance)                      swing = Swing.createSwing(golfer, swing\_datetime, club, swingSpeed, distance)            return render\_template("upload.html", name=current\_user.name, panel="Upload", Users=User.getAllUsers()) |

**Question 3 SWING function**

|  |  |
| --- | --- |
| a | # **Frontend** component code segment **#1** – In the **base.html** file, I added 2 dropdown lists under the ‘Swing’ option on the side bar which is a Jinja variable that will be configured in the log2.html file. Here is the code snippet:  <li class="nav-item"><a href="log2" class="nav-link text-white p-3 mb-2 sidebar-link"><i class="fas fa-address-card text-light fa-lg mr-3"></i>Swing</a></li>                  {% block userDropdown1 %}                  {% endblock %}                  {% block userDropdown2 %}                  {% endblock %}  # **Frontend** component code segment **#2** – In the **log2.html** file, it inherits the code from base.html file. The block "userDropdown1" is used to render a dropdown list of user emails for the Swing page. The conditional statement **{% if panel == "Swing" %}** checks if the current page being rendered is the "Swing" page. If it is, then a dropdown list is rendered with the id "selected\_email" and an onchange event is triggered when an option is selected, which calls the function "myFunction()" defined in the "log2.js" script.  Within the dropdown list, there is an initial option that reads "Select One User(by email)" and then, depending on whether the user is an admin or not, the dropdown list is populated with either all registered users or just the current user's email.  The script tag at the end of the block loads the "log2.js" script, which defines the "myFunction()" function that is called when the dropdown list changes.  Here is the code snippet:  {% extends "base.html" %}  <!--DROP DOWN LIST FOR EMAIL-->  {% block userDropdown1 %}  {% if panel == "Swing" %}  <select name="email" id="selected\_email" onchange="myFunction()">    <option value="all">Select One User(by email)</option>    {% if current\_user.email == "admin@abc.com" %} <!--admin has access to all registered users-->      {% for user in user\_list %}      <option value="{{user.email}}">{{user.email}}</option>      {% endfor %}    {% endif %}    {% if current\_user.email != "admin@abc.com" %} <!--non-admin only has access to logged in-->    <option value="{{current\_user.email}}">{{current\_user.email}}</option>    {% endif %}  </select>  {% endif %}  <script src="{{ url\_for('static', filename='js/log2.js') }}"></script>  {% endblock %}  # **Frontend** component code segment **#3** – Continuing from the code above in the **log2.html** file, another block named **userDropdown2** is used to render a dropdown list of clubs. The dropdown list is only shown when the **panel** variable passed to the template is equal to "Swing". The **select** element in the dropdown list has a **name** attribute set to "club" and an **id** attribute set to "selected\_club". There is one default **option** element that prompts the user to select a club. The **onchange** attribute is set to call a JavaScript function **myFunction()** defined in an external JS file. When a user selects a club from the dropdown list, the **myFunction()** function is called. It performs some actions based on the selected club, which are defined in the JS file. Here is the code snippet:  <!--DROP DOWN LIST FOR CLUB-->  {% block userDropdown2 %}  {% if panel == "Swing" %}  <select name="club" id="selected\_club" onchange="myFunction()"> <!--myFunction() is from JS file-->    <option value = ""> Select One Club (by label)</option>  </select>  {% endif %}  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>  <script src="{{ url\_for('static', filename='js/log2.js') }}"></script>  {% endblock %}  # **Frontend** component code segment **#4** – Continuing from the code above in the **log2.html** file, the code snippet below shows the main content block of the web page. It contains a form that allows the user to input their email, selected club, swing speed, and date and time. The form is submitted by clicking the "Swing" button.   * Email: The user is required to enter their email address, which will be used to identify the user and their swings in the database. * Club: The user is required to enter the label of the club they will be using for their swing. * Swing Speed: The user is required to enter the speed (in mph) of their swing. * Date & Time: The user is required to enter the date and time when the swing will take place. This is entered using a datetime-local input field.   Once the user fills out the form and clicks the "Swing" button, the form data is submitted to the /process2 route on the server for processing.\  Here is the code snippet:  {% block mainblock %}      <div class="card-header">          <h3>Golf Distance Calculator</h3>      </div>      <div class="card-body">          <div class="container-fluid col-sm-12">              <form action="/process2" method="post">              <!-- <input name="user\_email" type="hidden" id="user\_email" value=""> -->                  <div class="form-group row">                    <label for="email" class="col-sm-3 col-form-label">Email</label>                    <div class="col-sm-5">                      <input class="form-control" id="user\_email" placeholder="email" name="user\_email" required>                    </div>                  </div>                  <div class="form-group row">                    <label for="club" class="col-sm-3 col-form-label">Club</label>                    <div class="col-sm-5">                      <input class="form-control" id="club" placeholder="club" name="club" required>                    </div>                  </div>                  <div class="form-group row">                    <label for="swingspeed" class="col-sm-3 col-form-label">Swing Speed</label>                    <div class="col-sm-5">                      <input class="form-control" id="swingspeed" placeholder="speed (in mph)" name="swingspeed" required>                    </div>                  </div>                  <div class="form-group row">                      <label for="space" class="col-sm-3 col-form-label">Date & Time</label>                      <div class="col-sm-5">                        <input type="datetime-local" class="form-control" id="datetime" name="date" required>                      </div>                    </div>                  <div class="form-group row form-check">                          <button type="submit" class="offset-sm-3 btn btn-primary">Swing</button>                  </div>                  </form>              </div>      </div>  {% endblock %} |
| b | # **Backend** component code segment **#1** – From the **log2.js** file, ‘**myFunction’** is called when the user selects an email address from the ‘**selected\_email’** dropdown list. It gets the selected email value, sets it as the value of the ‘**user\_email’** input field, and then calls the ‘**getClubs’** function passing the selected email as a parameter. Here is the code snippet:  function myFunction(){    var x = document.getElementById("selected\_email").value;    //alert(x)    document.getElementById("user\_email").value = x;    //populate AJAX drop downlist to show car for selected email    getClubs(x)    var y = document.getElementById("selected\_club").value;    document.getElementById("club").value = y;  }//end of myFunction  # **Backend** component code segment **#2** – Continuing from the code above, in the **log2.js** file, **‘getClubs’** is an AJAX call that sends a POST request to the server with the selected email as data. When the request is successful, it retrieves the list of clubs associated with the selected email and populates the ‘**selected\_club’** dropdown list with the retrieved data using the ‘**populate’** function. Here is the code snippet:  function getClubs(aEmail){    $.ajax({        url: "/getClubs", //route in app.py        type: "POST",        data: {            aEmail: aEmail        },        error: function(){            alert("Error");        }, //end of error        success: function(data, status, xhr){            var club\_list=[];            for (var label of data.myClubs){                club\_list.push(label)            }            populate("#selected\_club",club\_list)        } //end of success    }) //end of ajax call  }// end of getClubs  # **Backend** component code segment **#3** – Continuing from the code above, in the log2.js file, **‘populate’** is a function used to populate a dropdown list with data. It clears the existing options in the dropdown list and appends new options based on the data passed to it. In this case, it adds the retrieved club labels to the ‘**selected\_club’** dropdown list. Here is the code snippet:  function populate(selector, club\_list){      $(selector).empty();    $(selector).append("<option value=\"\">Select One Club (by label)</option>");    if (club\_list.length != 0) {        for (var i=0; i<club\_list.length; i++) {            let select\_str = "<option value=\"" + club\_list[i] + "\">" + club\_list[i] + "</option>";            $(selector).append(select\_str);        }    }  }//end of populate |

**Question 4 SWING CHART function**

|  |  |
| --- | --- |
| a | # **Frontend** component code segment **#1** – From the **swingchart.html** file, this file inherits from the ‘**base.html’** Jinja template. The chart is displayed inside a div with class "chart-container" and a canvas element with id "swingChart". The canvas element has a custom attribute named "email\_id" which contains an email ID used to retrieve data for the chart. When the page loads, the **swingchart.js** file is executed, which retrieves data using the email ID and renders the chart on the canvas element. Here is the code snippet:  {% extends "base.html" %}  {% block mainblock %}  <div class="card-header">      <h3>Swing Chart</h3>  </div>  <div class="card-body">          <!-- Create a div where the graph will take place -->          <div class="chart-container" style="position: relative; width: 100%; height: 80vh;">              <canvas id="swingChart" width="400" height="300" email\_id="{{email\_id}}"></canvas>          </div>  </div>  <script src="{{ url\_for('static', filename='js/swingchart.js') }}"></script>  {% endblock %} |
| b | # **Backend** component code segment – From the **swingchart.js**, this code creates the chart using Chart.js library to display swing data. The chart is created in the canvas element with ID 'swingChart' in **swingchart.html file**. Here is the code snippet:  var ctx = document.getElementById('swingChart').getContext('2d');  // Retrieve email id from element with id 'swingChart'  var email\_id = $("#swingChart").attr("email\_id")  The code then initiates an AJAX request to the "/swingchart" URL using the HTTP POST method. The request sends the "email\_id" value in JSON format as data. If the request is successful, the function specified in the "success" parameter will be called with the response data as an argument. If there is an error, the function specified in the "error" parameter will be called. Here is the code snippet:  $.ajax({    url:"/swingchart",    type:"POST",    contentType: 'application/json;charset=UTF-8',    data: JSON.stringify({'email\_id': email\_id}),    error: function() {        alert("Error");    },    success: function(data, status, xhr) {  The chart is created using the following steps:   * Create a new Chart object with the canvas element as the first parameter and options as the second parameter. * Parse the data returned from the server and create a list of labels for the x-axis. The x-axis is configured as a time axis. * Loop through each data series (each series represents a user) and create a list of data points for the y-axis. These data points are the distance of the swings of the golfer. * For each data series, create a new dataset object with a unique label and add it to the chart. * Finally, update the chart to display the data.   Here is the code snippet for these steps:  var chartDim = {};        var chartDim = data.chartDim;      var xLabels = data.labels;      // # New Output      // # var chartDim = data.chartDim;      // # {'usr\_1': [[datetime1, 600], [datetime2, 600], ...], {'hotel\_2': [[],[], ...]}  ...}      // # var xLabels = data.labels;      // # // []      //debugger      var vLabels = [];      // ['usr\_1', 'usr\_2', ...]      var vData = [];      // [ [{'x': datetime\_1, 'y':666}, {'x': datetime\_2, 'y':1200} ...]      for (const [key, values] of Object.entries(chartDim)) {        vLabels.push(key);        let xy = [];        for (let i = 0; i < values.length; i++) {          debugger          let d = new Date(values[i][0]);          let year = d.getFullYear();          let month = ('' + (d.getMonth()+1)).padStart(2, '0');          let day = ('' + d.getDate()).padStart(2, '0');          let hour = ('' + d.getHours()).padStart(2, '0');          let mins = ('' + d.getMinutes()).padStart(2, '0');          aDateTime = year + '-' + month + '-' + day + ' ' + hour + ':' + mins          xy.push({'x': aDateTime, 'y': values[i][1]});        }        vData.push(xy);      }  The following code creates a new Chart object using the canvas context and specifies that there are no datasets yet. Here is the code snippet:  var swingChart = new Chart(ctx, {        data: {        // labels: xLabels,        datasets: []        },        options: {            responsive: true,            maintainaspectratio: false,          scales: {            x: {              type: 'time',              time: {                parser: 'yyyy-MM-dd HH:mm',              },              scaleLabel: {                display: true,                labelString: 'Date'              }            },            y: {              scaleLabel: {                display: true,                labelString: 'value'              }            }          }        }      });  The following code loops through the arrays of labels and data and adds a new dataset to the chart for each one. The chart is then updated to reflect the new datasets. Here is the code snippet:  for (i= 0; i < vLabels.length; i++ ) {        swingChart.data.datasets.push({        label: vLabels[i],        type: "line",        borderColor: '#'+(0x1100000+Math.random()\*0xffffff).toString(16).substr(1,6),        backgroundColor: "rgba(249, 238, 236, 0.74)",        data: vData[i],        spanGaps: true        });        swingChart.update(); |

Appendix: (rtf contents here as **text**)