The Transition Document for the Web Application Team

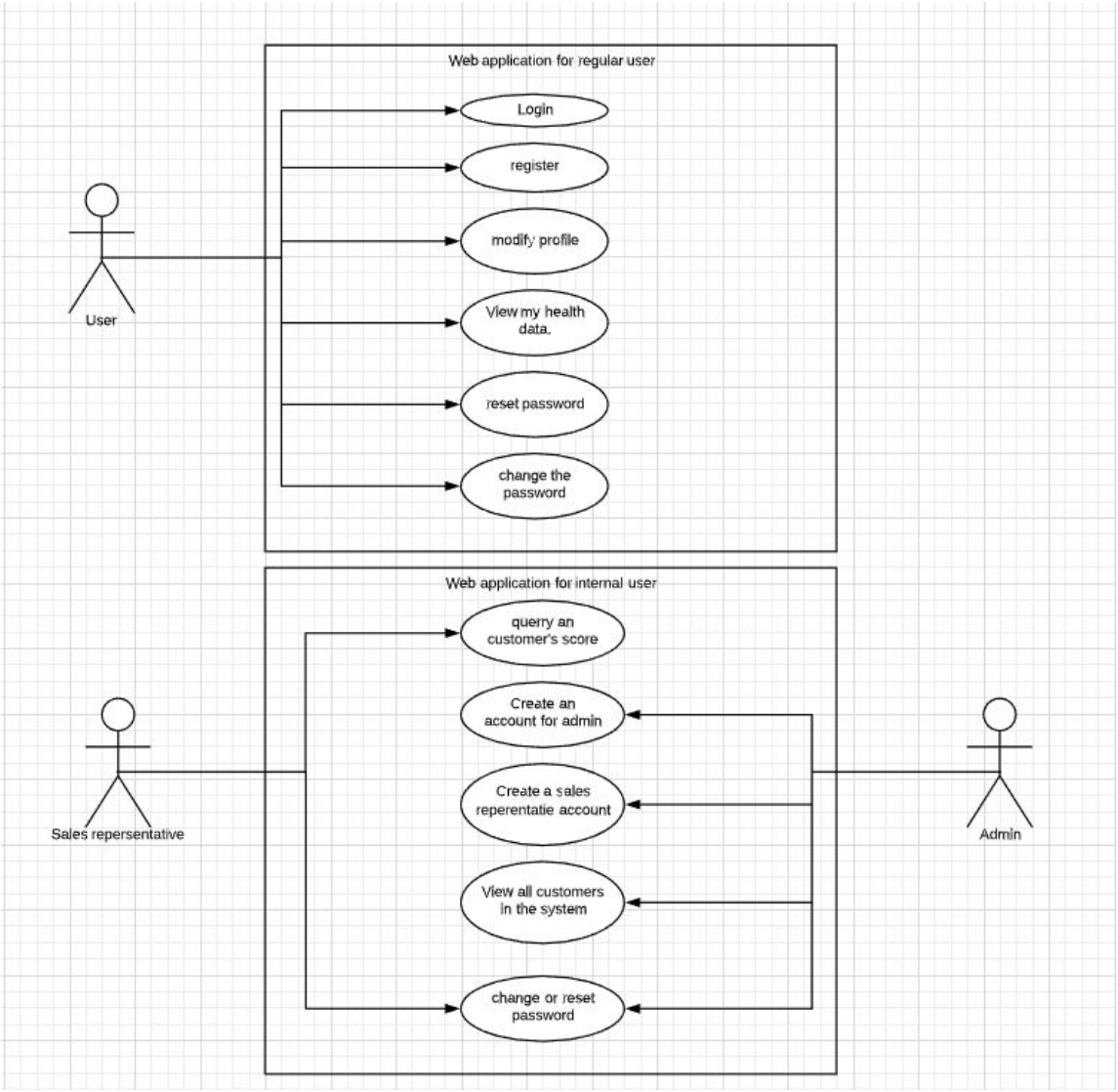
1 About the design of the web application

The web application is a subsystem of the project. The major goal is to create a web service to allow users to access their accounts through a browser. In May, we had an initial design of the web application and started to build our prototype. In July, after Devi showed his demo of the ML, we rebuilt the application since the database and the design changed.

1.1 The Initial design

1.1.1 Use case

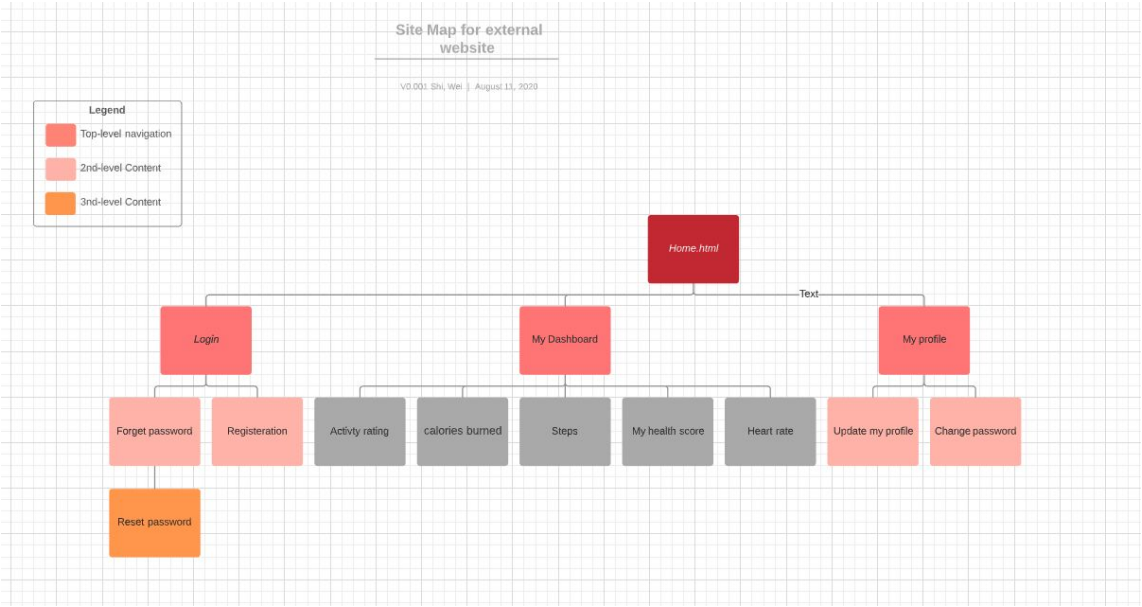
In the beginning, we decided to add a web service for the company employees, a default internal user “Admin” can create an account for other administrators and sales representatives. Different types of users can access different services. Our web application includes two parts, one is the “customer” part, the other is the “admin” part.



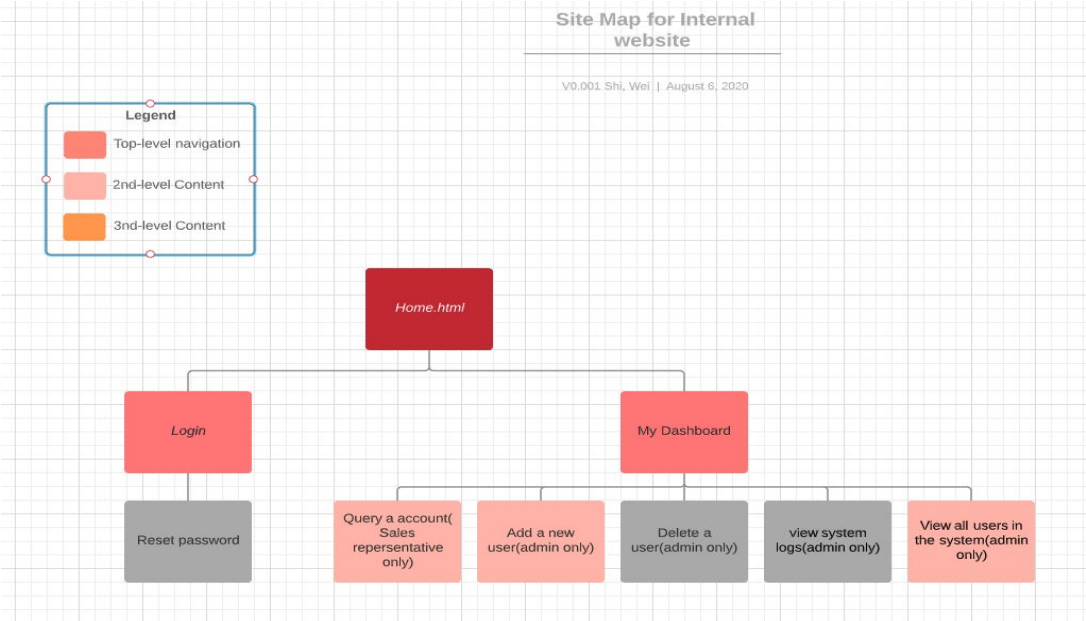
You also can use this link to access the use case diagram.

1.1.2 Site map

The website for the customers



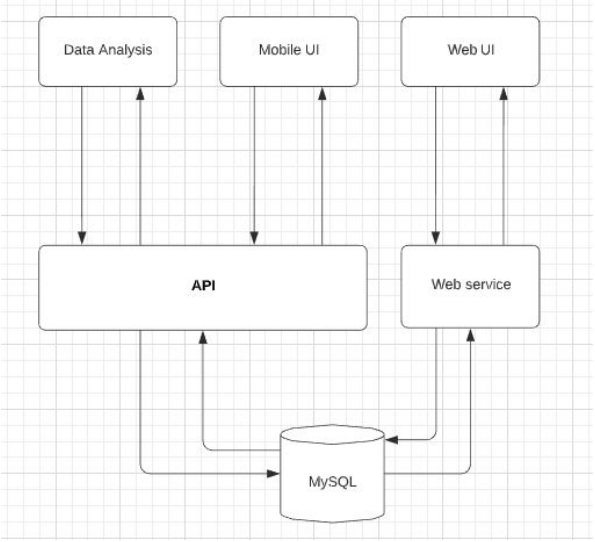
The website for the company employees, we also called the admin part of the web app.



You also can use this link to access the sitemap.

1.1.3 The relationship with the other subsystem

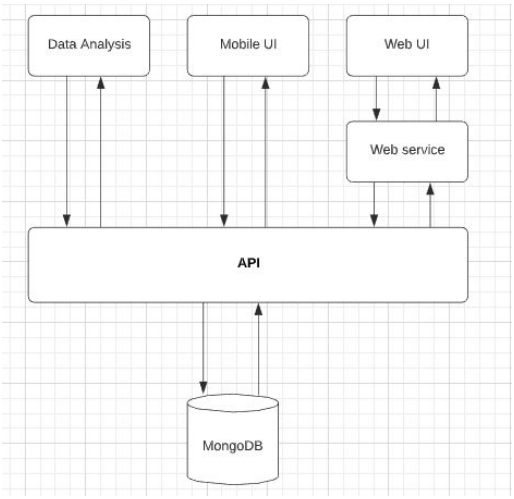
My opinion is that the three subsystems are independent. Once the database schema is decided, each team can work on the subsystem without waiting for other teams. Later on, we changed the architecture design and the database we use.

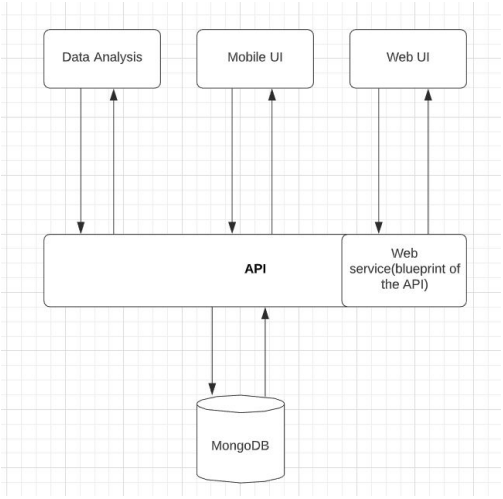


1.2 The changes in the design.

We changed the design around July 5th since Devi believed that if our endpoints register a user into the database will lead the issue in the data analysis subsystem.

The major change is about architecture design. Devi wants one API to provide three services for the web app, the mobile app, and the data analysis website. But the problem is Only one flask-login instant can be used in a flask project. We decided that our endpoints call the API to access the database to avoid the problems(which the left picture shows). Devi created user login, log out and registration in the API, we need to create the other endpoints we want at the API and call them from our web service. Later on, we found that we can use flask-session to replace the flask-login. **Our web service became a blueprint of the API now (the right picture shows)**. You can use this link to access the ads.





Other changes include that we **changed the database from Mysql to MongoDB** and **abandoned the Admin part of the web application** since Devi's website has the same idea. We only need to focus on the customers' website.

2 About the implementation of the web application

We choose **python and the flask framework** to create our web application. We create the models in models.py to interact with the database, build the endpoints in routes.py as the controller, and construct templates as the views.

2.1 Admin web app, the RESTful API and the demo

If you are familiar with python and web application development, you may skip this section and start from section 2.2. Before the changes in the design, I created an admin web application and a RESTful API. Edwin Also made a demo by integrating his customer registration page into my admin part of the web application, another approach for one web service providing login support for two web applications.

2.1.1 Admin web app and the Restful API

The purpose of building the RESTful API is that I tried to help the mobile application team to use a MySQL database to store the user registration info and user authentication instead of using AWS Cognito. This small project allows a mobile application to register a user, request a token, and revoke a token. It also helps you to understand the differences between the backend of a web application and a mobile application.

The RESTful API is a blueprint of the admin web service.

Download it: https://github.com/shiweiwei168/Web-admin

Install and run the service:

```
apt-get install python3-venv

python3 -m venv flask    //Create a venv called flask.
source flask/bin/activate  //activate the venv
pip install -r requirements.txt
run "FLASK_APP=admin.py"
run "flask run"
```

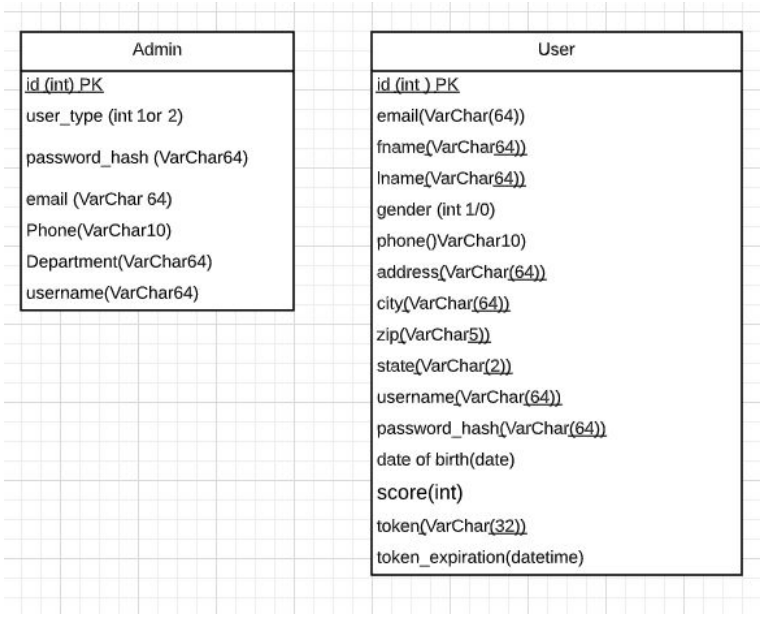
Test the admin website:

Use your browser to access 127.0.0.1:5000
The system has a default admin account {"admin","admin"}.

Test the RESTful API:

I include a readme.docx under the app/api folder.
https://github.com/shiweiwei168/Web-admin/blob/master/app/api/Readme.docx

The database schema



You can access the file by this link: [the schema]
You also can review the models.py in the project.

Major Flask modules

Flask-SQLalchemy //SQL database support
flask-migrate. //help you to migrate your database easily
werkzeug. // password hash
Flask-login //user authenticate
Blueprint. //The RESTful API is a blueprint of the admin web app.
Base64. // generate the token.

Switch the database

one advantage of using a SQL database is that you can use SQLite when you develop the application on your computer. You can switch the database to the MySQL database when you deploy the application to AWS.
I include a document about switching the database in the project.
https://github.com/shiweiwei168/Web-admin/blob/master/Switch%20database.txt

Deploy your web application on the AWS

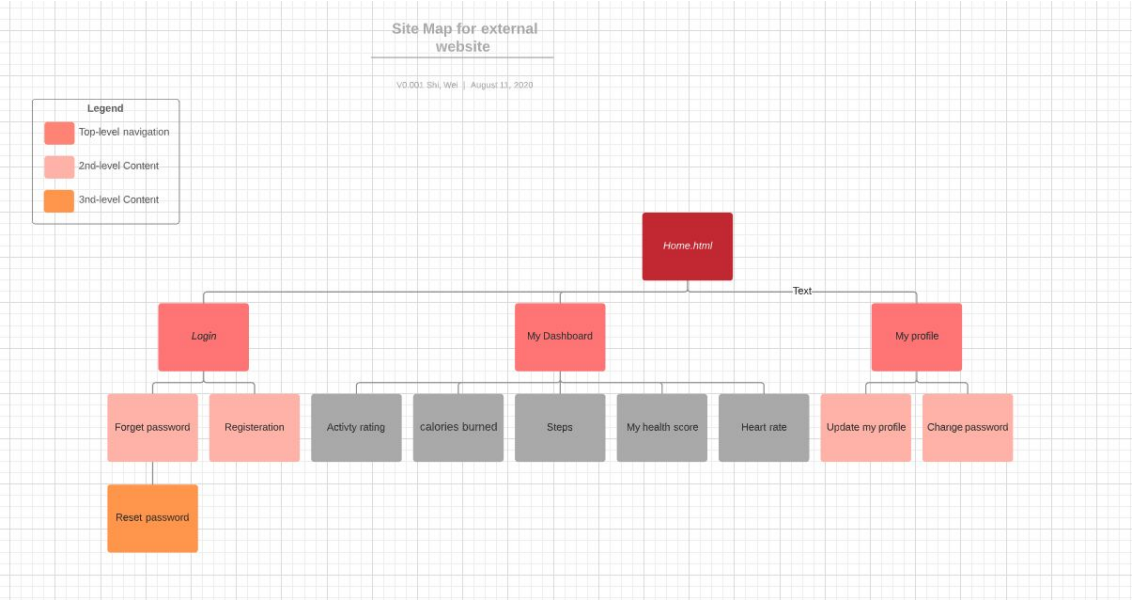
https://github.com/shiweiwei168/Web-admin/blob/master/deploy%20to%20AWS.txt

2.1.2 The Demo

The original demonstration of the website had a very different design then the current version. Initially we had one login page that both the users and admins would use to log in. The login request would check if the account was in the database as an admin or not, then redirect you to their corresponding pages. After more thought, we decided it was safer to separate the admins and users into different pages and databases. Next we migrated from an SQL database to using mongoDB as it was easier to implement with AWS and the mobile team.

2.2 The current version of the web application

The current version of the web application is a blueprint of the API. Our goal focuses on building a customer website.



*My Dashboard will show users today's detailed data, including steps, calories burned, heart rate, and activity rating.

*My profile will show the user's profile and allow users to upload a picture as a user's avatar, edit their profile and change the password.

2.2.1 Install and run the service:

download or clone it: https://github.com/statefarmuta/api_ml_admin

Devi provides a document and two videos that help you install and run the services (include our service).

Here is a link to the document.

https://github.com/statefarmuta/api_ml_admin/blob/master/README.md

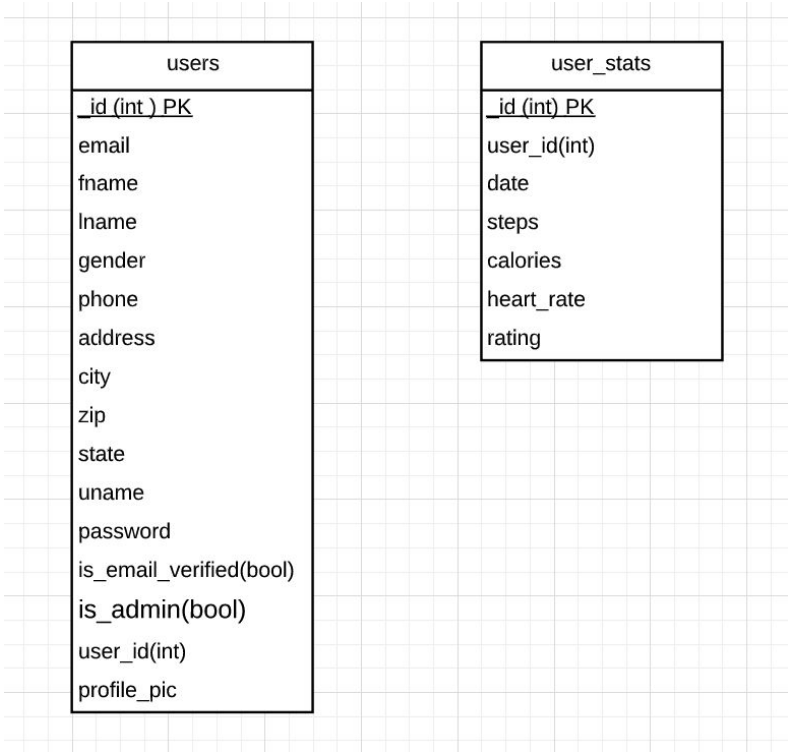
After installing the service on your computer, you should check if the **collection "user_stats"** exists. If the collection doesn't exist, you need to create one. database.txt

you can test our web service at http://0.0.0.0:5000/web/.

After you register the first user on the website, you may notice that the profile picture is empty. In order to avoid this situation, you should upload a picture "default" in the database.

You can login a user first, access my profile page, upload the picture “default” under “\app\web\doc” folder to the system. This picture will become a default picture for all users.

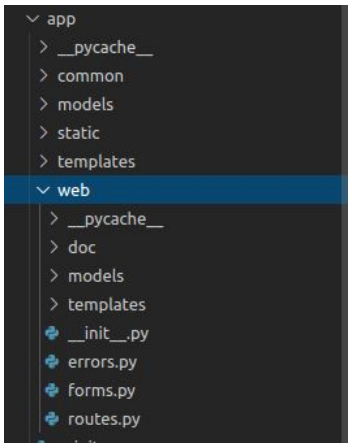
2.2.2 Database schema



You can access the file by this link: the schema
*user_stats table is still unclear at the end of our project. You should read Devi’s Document and talk with your teammates to make sure of the attributes of the table.

2.2.3 Application Structure

As I mentioned before, our web service is a blueprint of the API. The Blueprint under app/web/



We added the codes in the API’s __init__.py to register a blueprint, Pymongo, and Session.

```python
# __init__.py
from flask                  import Flask
from flask_sqlalchemy       import SQLAlchemy
from flask_login            import LoginManager
from flask_bcrypt           import Bcrypt
from app.common.database    import Database
from flask_mail             import Mail, Message

#### blueprint
from flask import Blueprint
from flask_session import Session
from flask import session
from flask_pymongo import PyMongo
#### blueprint

# Redis
import redis
from rq import Queue

# Grabs the folder where the script runs.
basedir = os.path.abspath(os.path.dirname(__file__))

app = Flask(__name__)
mail= Mail(app)

####blueprint
app.secret_key = 'fegrweiugwoibgpiw40pt8940gtbuorwbgo408bg80pw4'
app.config['SESSION_TYPE'] = 'filesystem'
sess = Session()
sess.init_app(app)
app.config['MONGO_DBNAME'] = 'novutree'
app.config['MONGO_URI'] = 'mongodb://localhost:27017/novutree'
mongo = PyMongo(app)
from app.web import bp as web_bp
app.register_blueprint(web_bp,url_prefix='/web')
####blueprint
```

Three of our endpoints, user registration, login, and log out call the API's endpoints to get the job done to avoid causing the issue for the data analysis subsystem. In the API's user registration, Devi also provides the email verification support. You may read his documentation to learn his approach.

```python
#user register, called the API.
@bp.route('/register', methods=['GET','POST'])
def register():
    if 'user' in session:
        return redirect(url_for('web.mydashboard'))
    form = RegistrationForm()
    jsonPayload = None
    if form.validate_on_submit():

        jsonPayload = {
                'username':form.username.data,
                'email':form.email.data,
                'password':form.password.data,
                'phone':form.phone.data,
                'name':form.fname.data,
                'lname':form.lname.data,
                'gender':form.gender.data,
                'address':form.address.data,
                'city':form.city.data,
                'zipcode':form.zipcode.data,
                'state':form.state.data
            }
        #print(jsonPayload)
        result = requests.post('http://0.0.0.0:5000/auth/register', json=jsonPayload)

        result = result.json()
        if result['status'] == 'success':
            flash('Congratulations, you registered an account! Please verify your email first.')
            return redirect(url_for('web.login'))
        flash('The username or email have already registered.')
    return render_template('/register.html', title='Register', form=form)
```

Bhupendra created the endpoints in the API and calls those endpoints from our endpoint to complete the password reset. He has a section (2.2.9) to talk about it. The rest endpoints of the web service can directly access the database to complete the task.

2.2.4 Database access

We tried two approaches to access the database.

The first one is Devi's approach. He uses it in the API.

```
from app.common.database import Database
```

```
Database.update_one(collection="users", query=[{'user_id':session['user'].uid}, \
    {"$set":{"uname":form.username.data,"fname":form.fname.data,"lname":form.lname.data,"bio":for
        ,"phone":form.phone.data,"address":form.address.data,"city":form.city.data \
            ,"zipcode":form.zipcode.data,"state":form.state.data}}])
```

The second one is Pymongo

```
app.config['MONGO_DBNAME'] = 'novutree'
app.config['MONGO_URI'] = 'mongodb://localhost:27017/novutree'
mongo = PyMongo(app)
```

```
from app import mongo
```

```
        #upload the pic
        mongo.save_file(profile_pic.filename, profile_pic)
        #get previous filename
        user = User.get_by_username(session['user'].name)
        previousFilename = user.profile_pic
        #update the profile
        query = { 'uname':session['user'].name}
        updates = { "$set": { "profile_pic": profile_pic.filename } }
        mongo.db.users.update_one(query, updates)

        # delete the old file if it isn't default
        if previousFilename != 'default':
            # get fs.files _id for previousfile #find_one doesn't work.
            fileobject = mongo.db.fs.files.find({'filename':previousFilename})
            #print(fileobject[0]['_id'])
            #delete the old file chunks from fs.chunks
            mongo.db.fs.chunks.remove({ 'files_id' : fileobject[0]['_id'] })
            #delete the old file record from fs.files
            mongo.db.fs.files.remove({ '_id' : fileobject[0]['_id'] })
    else:
        flash('The filename has been used. Please choice a different file name.')
```

If you upload a file to the database, you need to use pymongo.

2.2.5 Session

We used Session-based user authentication instead of flask_login. The server stores session info when the user login.

In the application, we have a session user model under app\web\models folder.

sessionuser.py

app > web > models > sessionuser.py

```python
from datetime import datetime
#from app import login
from flask_login import UserMixin
from werkzeug.security import generate_password_hash, check_password_hash

class Session_User():

    def __init__(self, username, hash, uid):
        self.name = username
        self.hash = hash
        self.uid = uid
```

When a user login, create a session user and store the info in the server.

```python
        result = result.json()
        if result['status'] != 'fail':
            user = Session_User(form.username.data,result['auth_token'],result['user']['user_id'])

            session['user'] = user
```

We can check the session user to protect the endpoint.

```python
if 'user' in session:
    userProfile = User.get_by_id(session['user'].uid)
    return render_template('/myprofile.html', user=userProfile)
else:
    return redirect(url_for('web.login'))
```

2.2.6 Dashboard

The dashboard presents today's data to the user. If the data didn't exist, the endpoint will return all 0 Dict.

```python
@bp.route('/mydashboard', methods=['GET'])
def mydashboard():
    if 'user' in session:
        today = date.today()
        todayData = mongo.db.user_stats.find_one({'user_id':session['user'].uid,'date':today.strftime("%Y-%m-%d")})
        #print(today.strftime("%Y-%m-%d"))
        #print(todayData)
        #print(todayData['steps'])
        #print(todayData['steps'][1])
        #if today's data doesn't exist
        if todayData is None:
            todayTotal=[0,0,0,0]
            todayData ={'steps':[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],\
                'calories':[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],\
                'heart_rate':[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],\
                'rating':[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]}

        else:
            totalsteps=0
            totalcalories=0
            totalheart_rate=0
            totalrating=0
            for steps in todayData['steps']:
                totalsteps +=  steps
            for calories in todayData['calories']:
                totalcalories +=  calories
            for heart_rate in todayData['heart_rate']:
                totalheart_rate += heart_rate
            for rating in todayData['rating']:
                totalrating += rating
            todayTotal =[totalsteps,totalcalories,totalheart_rate,totalrating]

        return render_template('/mydashboard.html',totalToday= todayTotal,todayData=todayData)
    else:
        return redirect(url_for('web.login'))
```

2.2.7 Profile_pic

The tricky part of the profile related endpoints is profile_pic upload.
We use GrifFS of MongoDB to store the profile picture files. The details
Although MongoDB doesn't prohibit users upload files that have the same filename, I decided that the filename should be unique. Therefore, if the filename has been used, I will ask the user to choose another one. Once the user uploads a new picture, I will delete the old one from the database except "default", a default user profile_pic. (don't forget to upload a picture "default" to the database before you run the service.)
In the template "myprofile.html":

```
<img src="{{url_for('web.request_file',filename=user['profile_pic'])}}" style="width:160px;height:160px;">
<form action = "http://0.0.0.0:5000/web/uploader" method = "POST"
    enctype = "multipart/form-data">
    <input type = "file" name = "profile_pic" />
    <input type = "submit" value ="Upload"/>
</form>
```

Endpoint:

```
#upload a pic to the database
@bp.route('/uploader', methods=['GET','POST'])
def upload_file():
    if 'user' in session:
        if 'profile_pic' in request.files:
            #make sure that the filename is unique
            profile_pic = request.files['profile_pic']
            files = mongo.db.fs.files.find({'filename':profile_pic.filename}).count()
            #print(files)
            if files ==0: # if files is not empty
                #upload the pic
                mongo.save_file(profile_pic.filename, profile_pic)
                #get previous filename
                user = User.get_by_username(session['user'].name)
                previousFilename = user.profile_pic
                #update the profile
                query = { 'uname':session['user'].name}
                updates = { "$set": { "profile_pic": profile_pic.filename } }
                mongo.db.users.update_one(query, updates)

                # delete the old file if it isn't default
                if previousFilename != 'default':
                    # get fs.files _id for previousfile #find_one doesn't work.
                    fileobject = mongo.db.fs.files.find({'filename':previousFilename})
                    #print(fileobject[0]['_id'])
                    #delete the old file chunks from fs.chunks
                    mongo.db.fs.chunks.remove({ 'files_id' : fileobject[0]['_id'] })
                    #delete the old file record from fs.files
                    mongo.db.fs.files.remove({ '_id' : fileobject[0]['_id'] })
            else:
                flash('The filename has been used. Please choice a different file name.')

            return redirect(url_for('web.myprofile'))
    else:
        return redirect(url_for('web.index'))
```

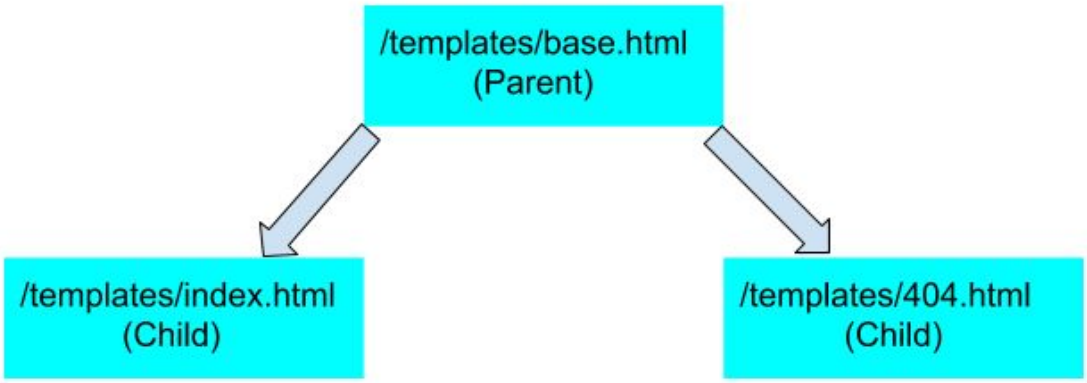
If you don't like this design, you can change it. For example, store a file's "_id" in the attribute profile_pic instead of "filename".

2.2.8 UI ()

The web application utilizes flask, jinja2 templates, HTML, CSS, as well as chart.js and a jquery plugin to display the data to the user.

Within the templates directory you will find all the HTML files:

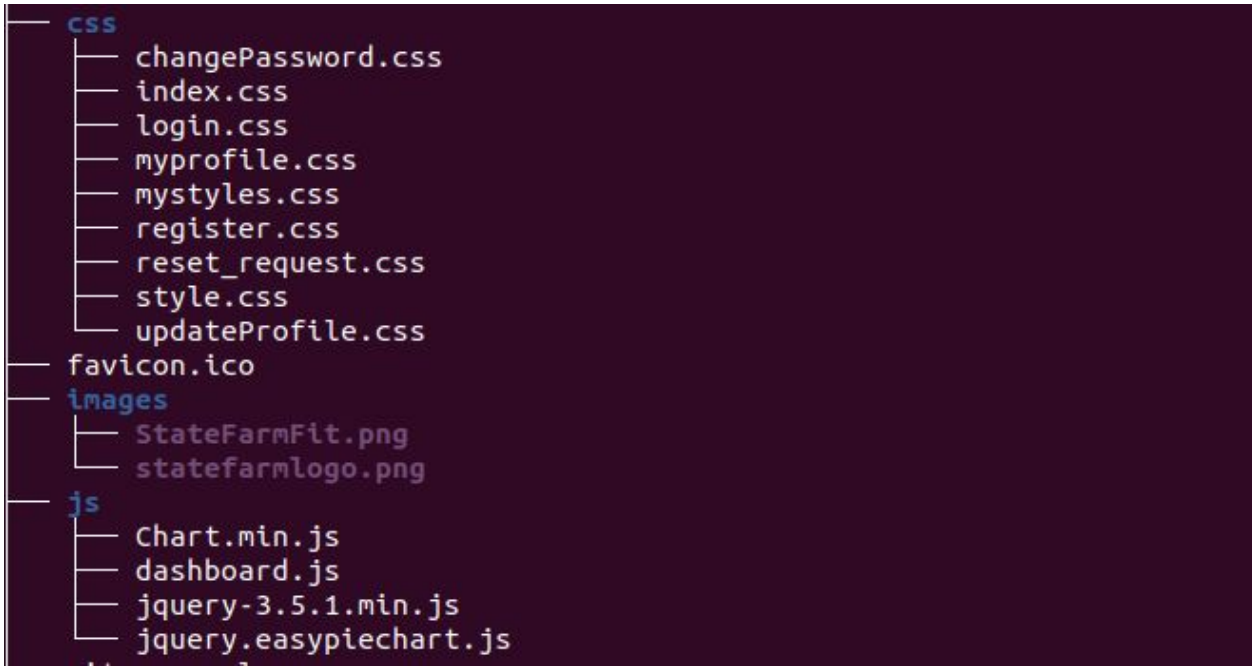
```
cxm0094@cxm0094-virtual-machine:~/senior_design/api_ml_admin/app/web$ tree
.
├── errors.py
├── forms.py
├── __init__.py
├── models
│   ├── __pycache__
│   │   ├── sessionuser.cpython-38.pyc
│   │   └── user.cpython-38.pyc
│   ├── sessionuser.py
│   └── user.py
├── __pycache__
│   ├── errors.cpython-38.pyc
│   ├── forms.cpython-38.pyc
│   ├── __init__.cpython-38.pyc
│   └── routes.cpython-38.pyc
├── routes.py
├── templates
│   ├── 404.html
│   ├── 500.html
│   ├── base.html
│   ├── changePassword.html
│   ├── index.html
│   ├── login.html
│   ├── mydashboard.html
│   ├── myprofile.html
│   ├── register.html
│   ├── reset_request.html
│   ├── updateProfile.html
│   └── view.html
└── user_stats.txt

4 directories, 25 files
```



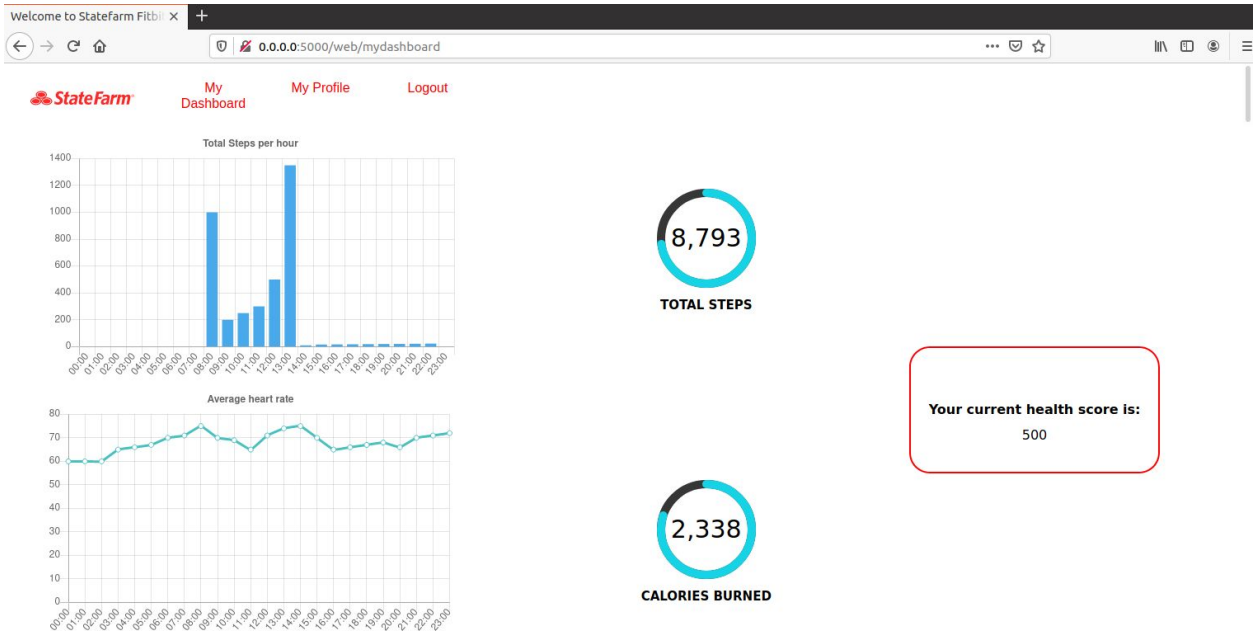
Structure of Template Inheritance

The base.html file contains the styling for the navbar which all other pages have in common. All other HTML files inherit from the base.

One directory up, in the app directory, there is a static directory. The static directory is where flask looks for images, css, and js files.



All the images, css, and js files used in the web application can be found in the static directory.



For the line graph and bar chart on the dashboard page chart.js was used. For the circular progress bars a jquery plugin was used, which can be found here: https://github.com/rendro/easy-pie-chart.

Here are some youtube videos I found helpful when coding the front end:

- Animated Circular Progress Bar Using Easy Pie Chart Plugin - Create a Progress Bar With Javascript
  - https://www.youtube.com/watch?v=BOgc1KRYk30
- Customized Animated Circular Progress Bar - Part 2 - Easy Pie Chart.js Simple jQuery Plugin Tutorial
  - https://www.youtube.com/watch?v=rOtNHrEDBbM&t=170s
- Beautiful Charts with JavaScript - Chart.JS Tutorial
  - https://www.youtube.com/watch?v=f-7uQXGur2o&t=217s
- Flask Forms and Styling
  - https://www.youtube.com/watch?v=6LvmaAtuwfU&t=1686s

2.2.9 Password reset

This section will explain about the password reset for web application (Statefarm Fitbit). Password reset is most important in any application. This part will explain the working of password reset features for those who forget their password.

Web/templates/login.html
This html file has a link for password reset. When you click this link it will redirect you to reset_request where you should provide email.

Web/templates/reset_reuest.html
This is a html file where you can fill up the related email when you want to change the password or forgot the password.

```
{% extends "base.html" %}

{% block content %}
    <h1>Reset Password</h1>

    <form action="" method="post">
        {{ form.hidden_tag() }}

        <p>It looks like you have trouble logging in. Enter your Email below and we will send you a link to reset your
        <br>
        <p>
            {{ form.email.label }}<br>
            {{ form.email(size=64) }}<br>
            {% for error in form.email.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
    <a href="{{ url_for('web.register') }}">SignUp.<br><br></a>
{% endblock %}
```

Web/forms.py
This is the corresponding form for request_reset.html that allows you to enter your email field. When you enter a valid email and click on submit button, it will trigger the endpoint reset_password_request.

```python
class RequestResetForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    submit = SubmitField('Request Password Reset')
```

Web/routes.py

```python
@bp.route("/reset_password_request",methods=['GET','POST'])
def reset_request():
    #if current_user.is_authenticated:
        #return redirect(url_for('index'))
    form = RequestResetForm()
    if form.validate_on_submit():
        jsonPayload={
            'email':form.email.data,
        }
        result = requests.post('http://0.0.0.0:5000/auth/reset_password_request', json=jsonPayload)
        result = result.json()
        if result['status'] == 'success':
            flash('Email has been sent to reset your password','_info_')
            return redirect(url_for('web.login'))

        flash('We did not find that email address in our records. Please check and re-enter it or register
    return render_template('/reset_request.html', title='Reset Password', form=form)

@bp.route('/mydashboard', methods=['GET'])
```

This will provide a route that will link to the API endpoint. Here we are running locally so the payload will be post to that endpoint i.e. 'http://0.0.0.0:5000/auth/reset_password_request'. If you are trying to use aws, you need to have link for ec2 which is something like http://ec2-34-212-133-20.us-west-2.compute.amazonaws.com/auth/request_password_request' This will redirect into that endpoint which is at view.py

app > views.py > auth_reset_password

```python
@app.route('/auth/reset_password_request',methods=['POST'])
def auth_reset_password():
 # get the post data
    post_data = request.get_json()
    if post_data is  None:
        post_data = request.form

    try:
        user = User.get_by_email(post_data.get('email'))
        if user:
            token = user.get_reset_token()
            msg = Message('Password Reset Request',
                          sender='ingenuity.senior@gmail.com',
                          recipients=[user.email])

            msg.body = f'''To reset your password, visit the following link:
                    {url_for('reset_token', token=token, _external=True)}
            If you did not make this request then simply ignore this email and no changes will be made.
            Sincerely,
            StateFarm
            '''
            mail.send(msg)
            responseObject = {
                    'status': 'success',
                    'message': 'Reset link sent.'
                }
            return make_response(jsonify(responseObject)), 201
        else:
            responseObject = {
            'status': 'fail',
            'message': 'Some error occurred with database. Please try again.'
            }
            return make_response(jsonify(responseObject)), 500
    except Exception as e:
        print(e)
        responseObject = {
            'status': 'fail',
```

Here if email does not exist in the database, it will provide some error. If exist, it will send a reset link to that email. But the token expires at 600 seconds. For this
token = user.get_reset_token() will use user_id for token expiry.

app > models > user.py > User > get_reset_token

```python
from werkzeug.security import generate_password_hash, check_password_hash

class User(UserMixin, object):

    # the main variable in user model
    def __init__(self, fname, lname, gender, phone, address, city, zipcode, state, uname, email, password, prof
        self.fname = fname
        self.lname = lname
        self.gender = gender
        self.phone = phone
        self.address = address
        self.city = city
        self.zip = zipcode
        self.state = state
        self.uname = uname
        self.email = email
        self.password = password
        self.profile_pic = 'default' if profile_pic is None else profile_pic
        self.bio = 'Hey this is ' + 'fname' if bio is None else bio
        self.user_id = int(str(uuid.uuid4().int)[:6]) if user_id is None else user_id
        self.is_admin = is_admin
        self.is_email_verified = is_email_verified

    def set_password(self, password):
        self.password_hash = generate_password_hash(self.password)

    def get_reset_token(self, expires_in=600):
        return jwt.encode(
            {'reset_token': self.user_id, 'exp': time() + expires_in},
            app.config['SECRET_KEY'], algorithm='HS256').decode('utf-8')

```

After user receive mail and click on the reset token link. It will be trigger towards another end point i.e. reset_token. Here first user_id is verified with verify_reset_token which is at app/models/users.py

```python
    def get_reset_token(self, expires_in=600):
        return jwt.encode(
            {'reset_token': self.user_id, 'exp': time() + expires_in},
            app.config['SECRET_KEY'], algorithm='HS256').decode('utf-8')

    @staticmethod
    def verify_reset_token(token):
        try:
            user_id = jwt.decode(token, app.config['SECRET_KEY'],
                                 algorithms=['HS256'])['reset_token']
        except:
            return
        #return User.get_id(user_id)
        return User.get_by_id(user_id)

```

If the token is invalid, it will redirect again to reset_request.html to enter email again and start process. If token is valid, the endpoint direct you to the form where you can enter the new password which is app/templates/page/reset_token.html. The corresponding .html and forms are shown below:

app > templates > pages > reset_token.html > ...

```
{% block content %}

    <h1>Reset Your Password</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

app > forms.py > RegisterForm

```
# -*- encoding: utf-8 -*-
"""
License: MIT
Copyright (c) 2019 Devi
"""

from flask_wtf          import FlaskForm
from flask_wtf.file     import FileField, FileRequired
from wtforms            import StringField, TextAreaField, SubmitField, PasswordField
from wtforms.validators import InputRequired, Email, DataRequired

class LoginForm(FlaskForm):
    username    = StringField  (u'Username'        , validators=[DataRequired()])
    password    = PasswordField(u'Password'        , validators=[DataRequired()])

class RegisterForm(FlaskForm):
    name        = StringField  (u'Name'      , validators=[DataRequired()])
    lname       = StringField  (u'Last Name' , validators=[DataRequired()])
    gender      = StringField  (u'Gender', validators=[DataRequired()])
    phone       = StringField  (u'Phone', validators=[DataRequired()])
    address     = StringField  (u'Address', validators=[DataRequired()])
    city        = StringField  (u'City', validators=[DataRequired()])
    zipcode     = StringField  (u'Zipcode', validators=[DataRequired()])
    state       = StringField  (u'State' , validators=[DataRequired()])
    username    = StringField  (u'Username'  , validators=[DataRequired()])
    password    = PasswordField(u'Password'  , validators=[DataRequired()])
    email       = StringField  (u'Email'     , validators=[DataRequired(), Email()])


class ResetPasswordForm(FlaskForm):
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Reset password')
```

```
@app.route("/reset_token/<token>",methods=['GET','POST'])
def reset_token(token):
    user=User.verify_reset_token(token)
    if user is None:
        flash('An invalid token','warning')
        return redirect(url_for('web.reset_request'))
    form = ResetPasswordForm()
    if form.validate_on_submit():
        pw_hash = form.password.data
        Database.update_one(collection="users", query=[{'user_id':user.user_id}, {"$set": { "password": pw_hash }} ])
        flash('Your password has been updated! you are now able to login')
        return redirect(url_for('web.login'))
    return render_template('pages/reset_token.html', title='Reset password', form=form)
```

Here in this end point the database will be updated and sends you with the message of successful password reset and redirect to the login page. Now you can enter your username and new password to login into your account.

3 Contact Information

“May, Chelsea” chelsea.may@mavs.uta.edu If you have questions about the UI section.
"Ramdam, Bhupendra" bhupendra.ramdam@mavs.uta.edu if you have questions about the password reset section.
"Popaja, Edwin E" edwin.popaja@mavs.uta.edu if you have questions about the Demo section.
"Shi, Wei" wei.shi2@mavs.uta.edu 972-987-7802 if you have questions about all other sections.