Name: Yathurshen Muralitharan
Date: 13/06/2021
Linkedin: https://www.linkedin.com/in/yathurshenmuralitharan/
Github repo: https://github.com/yatu360/ticketmanagement-python

1. Assignment goal:

The goal of this assignment is to demonstrate knowledge of Python, Flask, Restful APIs and database management techniques.

2. Scope:

To create a rudimentary event ticket management solution. The overall goal is to be able to create events and manage the number of people accessing each event.

3. Requirements:

-The application should have a web page to create events. Each event should have a Name, Date and initial number of tickets.

- Assume that this app will only have one user (so no need to implement authentication).

- From the event page, the user should be able to see the total number of tickets and how many have been redeemed. The page should also contain a button to refresh the counters.

- Each ticket can be represented with a unique token.

- The user should be able to add more tickets to an event.

- The application should have a page where the user can check the status of a ticket. A ticket can either be redeemed or ok.

- The application should expose an endpoint to redeem a ticket.

4. Assumptions:

- The app will only have one user.
- The app is to be used for low to medium traffic events.

5. Implementation:

The following tools and techniques were utilised in the design and creation of the ticket management app:

Front end: HTML, Jinja2 and CSS.

Backend: Python, Flask, SQLAlchemy

Database: SQLite

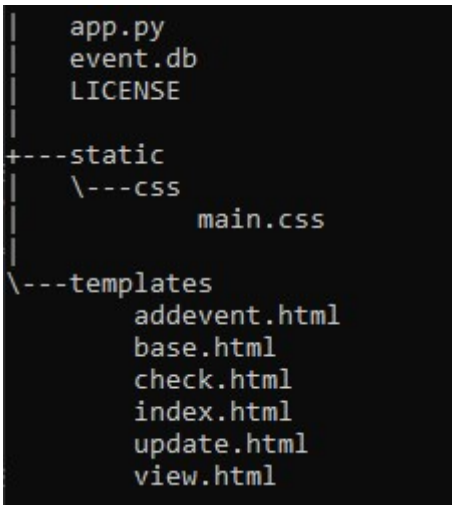
Reasons for choosing SQLAlchemy:

- Easier to setup as opposed to raw database
- Allows working with python objects
- Ability to work with different database management systems.

Despite the above listed advantages, SQLAlchemy does have its limitations such as any object nested in the database will no longer be mutable as SQLAlchemy will not detect the change and its processing ability is slower than using the database management system directly. Though for this assignment, SQLAlchemy proves sufficient.

Main reason for choosing SQLite as database system is that is generally fast and works great for most low to medium traffic requests.

5.1 File Structure Tree:



app.py – The main program file containing the backend services.

event.db – The database file used in the program

5.2 Tables:

The application consists of two tables with one-to-many implementation.

The first table is 'Event' which is also the parent table. The parent table 'Event' contains the following columns:

name: Stores the name of the event of string type and accepts up to 200 character. This is also set as the primary key for the able hence no duplication of events will be allowed. This is also a mandatory field.

```
name = db.Column(db.String(200), primary_key=True, nullable =False)
```

init_ticket: Stores the initial number of tickets for the given event. This column has field type integer and is a mandatory field.

```
init_ticket = db.Column(db.Integer, nullable =False)
```

date: Stores the date of the event, with field type string, accepts up to 10 characters. This column is also mandatory.

```
date = db.Column(db.String(10), nullable=False)
```

available: Stores the available tickets for the event with field type integer.

```
available = db.Column(db.Integer)
```

redeemed_ticket: Stores the redeemed tickets for the event with field type integer.

```
redeemed_ticket = db.Column(db.Integer)
```

The second table is 'Tickets' which is the child table. This table consists of the following columns:

id: Stores the unique ticket identifier for each event. This is the primary key of this table and has field type integer.

```
id=db.Column(db.Integer, primary_key=True, nullable =False)
```

event_name: Stores the event name of each ticket; entries to this column is linked to the name column of the 'Event' table.

```
event_name = db.Column(db.String(200), db.ForeignKey('event.name'))
```

redeemed: Stores true or false depending on if the ticket has been redeemed or not. True if redeemed, false is available.

```
redeemed = db.Column(db.Boolean)
```

5.3 Application methods:

I have chosen to create different routes for html and for api/json for simplicity and to make intentions clear. This also makes unit testing easier.

5.3.1 API Request Methods:

-**delete_api():**
URL: /api/delete/
Deletes the database and creates a new one, thereby deleting all the entries.

-**addticket_api(name):**
URL: /api/addticket/<name>
This method exposes an endpoint where it takes in the event name as a parameter and adds a ticket its available number of tickets. Returns '200: OK' if it successfully added the ticket.

-**redeem_api(id):**
URL: /redeem/<id>
This method exposes an endpoint where it takes in the ticket id as a parameter and redeems the stated ticket, if the ticket redemption is successful returns'200: OK', else it returns '410: GONE' if the ticket has already been redeemed.

-**view_api(name):**

URL: /api/view/<name>
This method exposes an endpoint where it takes in the event name as a parameter and returns json serialised container with event name, total tickets, available tickets, and redeemed tickets.

-**check_api(id):**
URL: /api/check/<id>
This method exposes an endpoint where it takes in the unique ticket id as a parameter and returns 'The ticket is ok (available)' if the ticket is available or 'The ticket has been redeemed' if the ticket is redeemed.

5.3.2 Web Request Methods

-**Index():**
URL: root
Loads the front page of the application

-**delete():**
URL: /delete1/
Deletes the database and creates a new one, thereby deleting all the entries.

-**check():**
Receives ticket id from the page's form and displays 'The ticket is ok (available)' if the ticket is available or 'The ticket has been redeemed' if the ticket is redeemed.

-**add():**
Receives event name, date of event and the number of initial tickets from form located in addevent.html page. Processes the information by initialising the available and redeemed tickets; available = initial tickets and redeemed tickets = 0. Finally processes ticket IDs for N tickets of the event and stores them in the Tickets table; this is done via for-loop.

-**view(name):**
URL: /view/<name>
Takes in name parameter then queries the name to retrieve the information from the database. This information is then past to the view template where the information is displayed to the user.

-**reset(name):**
URL: /reset/<name>
Takes in name as a parameter then queries to retrieve the event information, using which it deletes all the tickets and reinitialises the event to its original information.

-**redeemticket(name):**
URL: /redeemticket/<name>
Takes in name as a parameter, then queries to retrieve the event information, using which it sets the first available ticket of the event to be redeemed. The method also changes the value of the redeemed section of 1 ticket from the Ticket table linked to the event to be 'True'.

-**addticket(name):**
URL: /addticket/<name>

Takes in name as a parameter, then queries to retrieve the event information, using which it adds one ticket to the available data of the event. The method also adds one ticket to the ticket table and links it to the event as well as assigning it with the next increment of ticket ID.

5.4 Web page templates:

The application utilises four templates all of which inherits their settings from base.html. The templates are index.html, addevent.html, check.html and view.html.

In all templates, Jinja2 syntaxes are used to process frontend services.