2-se

I needd your help with a project with python backend and html css js frontend that uses api to consume the backend. first of all I will forward you the page code.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>E-Ticket</title>
    <link rel="icon" href="../assets/images/favicon.png" type="image/png">

    <!-- Personal CSS -->
    <link rel="stylesheet" href="../style/style.css">
    <link rel="stylesheet" href="../style/ticket.css">

    <!-- Bootstrap 5 -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.3/font/bootstrap-icons.css">

</head>

<!-- Bootstrap 5 JS Bundle with Popper -->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"></script>

</body>

<body class="d-flex justify-content-between align-items-center" style="background-color: #00466a;">

    <div class="container container-wrapper d-flex flex-column justify-content-center align-items-center h-50 w-50 p-4">

        <div class="container mb-4">
            <h1 class="m-0">Ticket Selling Platform</h1>
        </div>

        <!-- Navbar -->
        <div class="container d-flex bg-dark mb-4 py-3" style="border-radius: 48px;">
            <button id="view-tickets-nav-btn"
                class="btn btn-dark menu-btn mx-2 d-flex align-items-center justify-content-center">
                View Tickets
            </button>
            <button id="add-ticket-nav-btn"
```

```
      class="btn btn-dark menu-btn mx-2 d-flex align-items-center justify-content-center">
      Add Ticket
    </button>
  </div>

  <!-- View tickets content -->
  <div id="ticket-view-page" class="container justify-content-center align-items-start"
    style="display: none; height: 65%; overflow-y: auto; scrollbar-width: none;">

    <table id="tickets-table" class="table table-hover align-middle" aria-hidden="true">
      <thead>
        <tr>
          <th scope="col">Artist</th>
          <th scope="col">Location</th>
          <th scope="col">Price</th>
          <th scope="col" class="text-end">Operation</th>
        </tr>
      </thead>
      <tbody id="tickets-table-body">
      </tbody>
    </table>
  </div>

  <!-- Add new ticket content -->
  <div id="ticket-add-page" class="container"
    style="display: none; height: 65%; overflow-y: auto; scrollbar-width: none;">
    <div class="h-100 w-100">
      <!--
      TASK 1

      Important:
      To ensure propper page functionality you are encouraged to write code strictly
      within the bounds of this component.

      Do not delete this comment!
      -->

    </div>
  </div>

</div>

<!-- Page Design -->
<div class="h-100 pe-5 d-flex justify-content-around" style="width: 30%;">
  <div class="ticket-row h-100 d-flex flex-column justify-content-center">
    <div class="ticket-orange">
      <h3 style="transform: rotate(270deg);">Ticket</h3>
    </div>
    <div class="ticket-yellow">
      <h3 style="transform: rotate(270deg);">Ticket</h3>
    </div>
```

```
            <div class="ticket-blue">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-orange">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-yellow">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
        </div>
        <div class="ticket-row h-100 d-flex flex-column justify-content-center">
            <div class="ticket-blue">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-orange">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-yellow">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-blue">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
        </div>
        <div class="ticket-row h-100 d-flex flex-column justify-content-center">
            <div class="ticket-yellow">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-blue">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-orange">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-yellow">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
            <div class="ticket-blue">
                <h3 style="transform: rotate(270deg);">Ticket</h3>
            </div>
        </div>

    </div>

    <!-- Bootstrap 5 JS Bundle with Popper -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js">
</script>
    <!-- Personal JS -->
    <script src="../scripts/api_script.js"></script>

    <script>
```

```
// Session functions
let pageNumber = 0;

function resetVariables() {
    pageNumber = 0;
}

function saveSessionData() {
    const data = {
        pageNumber: pageNumber,
    }
    sessionStorage.setItem('sessionData', JSON.stringify(data));
}

// Nav buttons
const viewTicketsNavBtn = document.getElementById('view-tickets-nav-btn');
const addTicketNavBtn = document.getElementById('add-ticket-nav-btn');

// Content divs
const viewTicketsContent = document.getElementById('ticket-view-page');
const addTicketsContent = document.getElementById('ticket-add-page');

// Tickets table
const ticketsTableBody = document.getElementById('tickets-table-body');

// Add Ticket Inputs
const inputArtist = document.getElementById('artist-input');
const inputLocation = document.getElementById('location-input');
const inputPrice = document.getElementById('price-input');

// Add Ticket buttons
const submitTicket = document.getElementById('ticket-add-submit-btn');
const resetTicket = document.getElementById('ticket-add-reset-btn');

// Nav buttons functionality
viewTicketsNavBtn.addEventListener('click', function () {
    pageNumber = 0;
    saveSessionData();
    updatePage();
});

addTicketNavBtn.addEventListener('click', function () {
    pageNumber = 1;
    saveSessionData();
    updatePage();
});

// Add Ticket buttons functionality
submitTicket.addEventListener('click', async function () {
    const artist = inputArtist.value;
```

```
            const location = inputLocation.value;
            const price = parseInt(inputPrice.value);

            inputArtist.value = '';
            inputLocation.value = '';
            inputPrice.value = '';

            await addTicket(artist, location, price);
            pageNumber = 0;
            saveSessionData();
            updatePage();
        });

        resetTicket.addEventListener('click', function () {
            inputArtist.value = '';
            inputLocation.value = '';
            inputPrice.value = '';
        })

        // Delete Ticket buttons functionality
        function addFunctionalityDeleteBtns() {
            document.querySelectorAll('.delete-ticket-btn').forEach(button => {
                button.addEventListener('click', async function () {
                    // TASK 2
                });
            });
        }

        async function updateTicketsTable() {
            ticketsTableBody.innerHTML = "";

            const tickets = await getTickets();
            for (const ticket of tickets) {
                const entryHtml =
                    <tr>
                        <td>${ticket.artist}</td>
                        <td>${ticket.location}</td>
                        <td>${ticket.price}$</td>
                        <td class="text-end">
                            <button id="${ticket.id}" type="button" class="btn btn-danger menu-btn delete-ticket-btn">Delete</button>
                        </td>
                    </tr>
                ;
                ticketsTableBody.innerHTML += entryHtml;
            }
            addFunctionalityDeleteBtns();
        }

        async function updatePage() {
            let sessionData = sessionStorage.getItem('sessionData');
            if (sessionData) {
                const storedData = JSON.parse(sessionStorage.getItem('sessionData'));
                pageNumber = storedData.pageNumber;
            }

            // View Tickets page
            if (pageNumber == 0) {
                // Update navbar
                viewTicketsNavBtn.classList.add('active');
                addTicketNavBtn.classList.remove('active');

                // Update page
                viewTicketsContent.style.display = 'flex';
                addTicketsContent.style.display = 'none';

                // Update ticket table
                await updateTicketsTable();

                return;
            }

            // Add Tickets page
            if (pageNumber == 1) {
                // Update navbar
                viewTicketsNavBtn.classList.remove('active');
                addTicketNavBtn.classList.add('active');

                // Update page
                viewTicketsContent.style.display = 'none';
                addTicketsContent.style.display = 'flex';
            }
        }

        // Document loaded
        window.onload = function () {
            updatePage();
        }

    </script>

</body>
```

</html> after that, I will send you the logic for the api, and we will decide together the best way to go through this. first of all, just learn about the current page

Here’s what I see in the “E-Ticket” page you’ve shared:

---

1. Overall Layout

- **Wrapper container** (`.container-wrapper`) centered, 50% width/height, with the page title ("Ticket Selling Platform") at the top.
- **Two "pages"** inside it, shown/hidden via inline `display` styles:
  - **View Tickets** (`#ticket-view-page`)
  - **Add Ticket** (`#ticket-add-page`)
- A **vertical decorative "ticket" sidebar** on the right.

---

2. Navigation

- Two buttons in a dark-background navbar:
  - **View Tickets** (`#view-tickets-nav-btn`)
  - **Add Ticket** (`#add-ticket-nav-btn`)
- Clicking either sets `pageNumber` (0=view, 1=add), saves it to `sessionStorage`, and calls `updatePage()`, which toggles visibility + active styling.

---

3. "View Tickets" Page

- A `<table>` with headers **Artist, Location, Price, Operation**.
- Body (`#tickets-table-body`) is populated by `updateTicketsTable()` which:
  1. Calls `getTickets()` (your API client).
  2. Builds one `<tr>` per ticket, injecting `ticket.artist`, `ticket.location`, `ticket.price`.
  3. Inserts a Delete button with class `.delete-ticket-btn` and `id="<ticket.id>"`.
  4. Hooks up click handlers in `addFunctionalityDeleteBtns()` (currently empty, marked **TASK 2**).

---

4. "Add Ticket" Page

- Currently empty aside from a comment block marked **TASK 1**.
- Your script references three inputs by ID:
  - `#artist-input`
  - `#location-input`
  - `#price-input`
- And two buttons:
  - `#ticket-add-submit-btn`
  - `#ticket-add-reset-btn`
- The submit handler reads values, calls `addTicket(artist, location, price)`, then switches back to View Tickets.

---

5. JavaScript Flow

1. **On load** → `updatePage()`
2. **View Tickets** mode → shows table, calls `updateTicketsTable()`.
3. **Add Tickets** mode → shows the (empty) add form.
4. **Submit** → reads inputs, clears them, calls `addTicket()`, then goes back to page 0.
5. **Reset** → clears inputs.
6. **Delete** → placeholder in `addFunctionalityDeleteBtns()` (TASK 2).

---

6. Where to Inject Code

- **TASK 1** (in the Add Ticket page's container) is where you'll need to add the HTML form:

  ```html
  <div>
    <label>Artist: <input id="artist-input" … /></label>
    <!-- etc. -->
    <button id="ticket-add-submit-btn">Submit</button>
    <button id="ticket-add-reset-btn">Reset</button>
  </div>
  ```

- **TASK 2** (in the delete-button click handler) is where you'll call your `deleteTicket(id)` API helper and then refresh the table.

Next Steps

1. **Show me the API client logic** (the contents of `api_script.js` or equivalent), so we can see how to call `getTickets()`, `addTicket()`, and `deleteTicket()`.
2. Decide how to structure error handling, form validation, and re-rendering after add/delete.

Whenever you're ready, send over the backend/API code and we'll map out the implementation!

our first task is to create a form that allow to add new tickets. tickets have ID; Artyist, Location and Price. ID is autogenerated so no worries. also, there should be one button with text "clear" to resrt the input fields and one to submit the ticket. you need to know the id's of each field and button so that the js connected to this proejct can contnue wotrking properly. Artist input field ID: artist-input, Location field ID: location-input, Price field ID: price-input; Reset button ID: ticket-add-reset-btn; Submit button ID: ticket-add-reset-btn; Submit button iD: ticket-add-submit-btn. this instead is the current js implementation: async function getTickets() {

```
    try {
        const response = await fetch(http://127.0.0.1:5000/get_tickets);

        if (!response.ok) {
            throw new Error('error');
```

```
        }

        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error with GET request:', error);
    }
}

function addTicket(artist, location, price) {
    return fetch('http://127.0.0.1:5000/add_ticket', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ 'artist': artist, 'location': location, 'price': price })
    })
        .then(response => response.json())
        .then(data => {
            console.log(data);
        })
        .catch(error => {
            console.error('Error:', error);
        });
}

// TASK 2
function deleteTicket() {

} and these are the python endpoints: import os

from flask import Flask, request, jsonify, send_from_directory
from flask_cors import CORS

from business.ticket_manager import TicketManager

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
PROJECT_ROOT = os.path.abspath(os.path.join(BASE_DIR, '..', '..'))
CLIENT_DIR = os.path.join(PROJECT_ROOT, 'client')

app = Flask(__name__, static_folder=os.path.join(CLIENT_DIR, 'web'))

@app.route('/')
def serve_index():
    return send_from_directory(os.path.join(CLIENT_DIR, 'web'), 'index.html')

@app.route('/assets/<path:filename>')
def serve_assets(filename):
    return send_from_directory(os.path.join(CLIENT_DIR, 'assets'), filename)

@app.route('/style/<path:filename>')
```

```
def serve_style(filename):
    return send_from_directory(os.path.join(CLIENT_DIR, 'style'), filename)

@app.route('/scripts/<path:filename>')
def serve_scripts(filename):
    return send_from_directory(os.path.join(CLIENT_DIR, 'scripts'), filename)

# API Route example
@app.route('/hello_world', methods=['GET'])
def hello_world():
    data = {'message': 'hello world'}
    resp = jsonify(data)
    return resp

@app.route('/get_tickets', methods=['GET'])
def get_tickets():
    tm = TicketManager()
    try:
        data = tm.retrieve_tickets()
        resp = jsonify(data)
        return resp

    except Exception as e:
        data = {'message': e.args[0]}
        resp = jsonify(data)
        return resp

@app.route('/add_ticket', methods=['POST'])
def add_ticket():
    data = request.json
    tm = TicketManager()
    try:
        tm.insert_ticket(data)
        data = {'message': 'OK'}
        resp = jsonify(data)
        return resp

    except Exception as e:
        data = {'message': e.args[0]}
        resp = jsonify(data)
        return resp

    resp = jsonify(None)
    return resp

# TASK 2
@app.route('/delete_ticket', methods=['DELETE'])
def delete_ticket():
    pass

if __name__ == '__main__':
```

app.run(debug=True, host='0.0.0.0', port=5000). focus only on giving me the necessary code to implement this new feature and task use the same style and similar css as the one used in the current implementation:

```
body {
    position: fixed;
    top: 0px;
    left: 0;
    width: 100%;
    height: 100vh;
}

.container-wrapper {
    width: 100%;
    max-height: 100%;
    border-radius: 24px;
    overflow: hidden;
    border: 2px solid black;
    box-shadow: 0 4px 6px 2px rgba(0, 0, 0, 0.2);
    background-color: white;
}

.menu-btn {
    border-radius: 24px !important;
}

.active {
    background-color: #fea000 !important;
    color: black !important;
}

#tickets-table thead th {
    position: sticky;
    top: 0;
    z-index: 2;
    box-shadow: 0 2px 0 black;
} .ticket-yellow {
    position: relative;
    width: 110px;
    height: 220px;
    background-image:
        radial-gradient(circle at top left, transparent 17px, #fea000 17px),
        radial-gradient(circle at top right, transparent 17px, #fea000 17px),
        radial-gradient(circle at bottom left, transparent 17px, #fea000 17px),
        radial-gradient(circle at bottom right, transparent 17px, #fea000 17px);
    background-size: 50% 50%;
    background-repeat: no-repeat;
    background-position: top left, top right, bottom left, bottom right;
    margin-top: 10px;
    margin-bottom: 10px;
    display: flex;
    justify-content: center;
    align-items: center;
}

.ticket-orange {
    position: relative;
    width: 110px;
    height: 220px;
    background-image:
        radial-gradient(circle at top left, transparent 17px, #f36500 17px),
        radial-gradient(circle at top right, transparent 17px, #f36500 17px),
        radial-gradient(circle at bottom left, transparent 17px, #f36500 17px),
        radial-gradient(circle at bottom right, transparent 17px, #f36500 17px);
    background-size: 50% 50%;
    background-repeat: no-repeat;
    background-position: top left, top right, bottom left, bottom right;
    margin-top: 10px;
    margin-bottom: 10px;
    display: flex;
    justify-content: center;
    align-items: center;
}

.ticket-blue {
    position: relative;
    width: 110px;
    height: 220px;
    background-image:
        radial-gradient(circle at top left, transparent 17px, #5bc4d0 17px),
        radial-gradient(circle at top right, transparent 17px, #5bc4d0 17px),
        radial-gradient(circle at bottom left, transparent 17px, #5bc4d0 17px),
        radial-gradient(circle at bottom right, transparent 17px, #5bc4d0 17px);
    background-size: 50% 50%;
    background-repeat: no-repeat;
    background-position: top left, top right, bottom left, bottom right;
    margin-top: 10px;
    margin-bottom: 10px;
    display: flex;
    justify-content: center;
    align-items: center;
}
```

Below is the HTML you can drop into the **`<!-- TASK 1 … -->`** area of your Add-Ticket page. It uses Bootstrap form controls and your existing **`.menu-btn`** class to match the rest of the UI. Just replace the comment block with this:

```html
<!-- TASK 1
     Do not delete this comment! -->
<form id="ticket-add-form" class="w-100 h-100 d-flex flex-column justify-content-between">
  <div>
    <div class="mb-3">
      <label for="artist-input" class="form-label">Artist</label>
      <input
        type="text"
        class="form-control"
        id="artist-input"
        placeholder="Enter artist name"
        required
      />
    </div>
    <div class="mb-3">
      <label for="location-input" class="form-label">Location</label>
      <input
        type="text"
        class="form-control"
        id="location-input"
        placeholder="Enter location"
        required
      />
    </div>
    <div class="mb-3">
      <label for="price-input" class="form-label">Price ($)</label>
      <input
        type="number"
        class="form-control"
        id="price-input"
        placeholder="Enter price"
        min="0"
        required
      />
    </div>
  </div>

  <div class="d-flex justify-content-between">
    <button
      type="button"
      id="ticket-add-reset-btn"
      class="btn btn-secondary menu-btn"
    >
      Clear
    </button>
    <button
      type="submit"
      id="ticket-add-submit-btn"
      class="btn btn-primary menu-btn"
    >
      Submit
    </button>
  </div>
</form>
```

Notes:

- The form is stretched full-height (`h-100`) so your reset/submit buttons stay pinned at the bottom.
- IDs match your JS:
  - **Artist** → `artist-input`
  - **Location** → `location-input`
  - **Price** → `price-input`
  - **Clear** → `ticket-add-reset-btn`
  - **Submit** → `ticket-add-submit-btn`
- We wrapped the Submit button in a `<form>` with `type="submit"`. Your existing JS listener on the submit button will still fire—but if you prefer to prevent native form submission, you can add:

```js
document.getElementById('ticket-add-form').addEventListener('submit', e => {
  e.preventDefault();
  submitTicket.click();
});
```

Otherwise the button's click handler will suffice.

- Uses Bootstrap's spacing (`mb-3`) and your `.menu-btn` for rounded styling.

this was a very good job .please keep it on the same line and method. I like it. now the task 2 is to implement the Delete ticket function. please identify all missing components and parts of code and send them to me so that I can effectively copy these pieces of code in the right place in the files and it works exactly as expected.

Below are **all** the pieces you'll need to drop into the right places so that "Delete" works end-to-end, front to back. Just copy each snippet into the indicated file/section.

---

1. Front-end (`client/web/scripts/api_script.js`)

a) Add the `deleteTicket` helper below your existing `addTicket`:

```js
// ----------- After addTicket(...) -----------
async function deleteTicket(id) {
    try {
        const response = await fetch('http://127.0.0.1:5000/delete_ticket', {
            method: 'DELETE',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ id: id }),
        });

        if (!response.ok) {
            throw new Error('Network response was not ok');
        }

        const data = await response.json();
        console.log('Deleted:', data);
    } catch (error) {
        console.error('Error deleting ticket:', error);
    }
}
```

b) Fill in TASK 2 in your existing `addFunctionalityDeleteBtns()`:

```js
function addFunctionalityDeleteBtns() {
    document.querySelectorAll('.delete-ticket-btn').forEach(button => {
        button.addEventListener('click', async function () {
            const id = this.id;                     // the ticket's id
            await deleteTicket(id);               // call the DELETE API
            await updateTicketsTable();           // refresh the table view
        });
    });
}
```

---

2. Back-end (`server/app.py`)

a) Implement the `/delete_ticket` route (replace the `pass`):

```python
@app.route('/delete_ticket', methods=['DELETE'])
def delete_ticket():
    data = request.json
    ticket_id = data.get('id')
    tm = TicketManager()
    try:
        tm.delete_ticket(ticket_id)
        return jsonify({'message': 'OK'})
    except Exception as e:
        return jsonify({'message': str(e)}), 400
```

---

3. Business Logic (`server/business/ticket_manager.py`)

If you don't already have one, add this method to your `TicketManager` class:

```python
class TicketManager:
    # … existing methods …

    def delete_ticket(self, ticket_id):
        conn = self._get_connection()      # or however you open your DB
        cursor = conn.cursor()
        cursor.execute("DELETE FROM tickets WHERE id = ?", (ticket_id,))
        conn.commit()
        cursor.close()
        conn.close()
```

> **Note:**
>
> - Adjust `_get_connection()` to whatever you call to open your SQLite/Postgres/etc.
> - Make sure your table is named `tickets` and `id` is the primary key.

Recap of where to drop each snippet:

File	Location / What to Paste
`api_script.js`	After `addTicket()`; and replace the `// TASK 2` block in `addFunctionalityDeleteBtns()`
`server/app.py`	In the `@app.route('/delete_ticket', methods=['DELETE'])` handler (replace `pass`)

File	Location / What to Paste
`server/business/ticket_manager.py`	Inside `class TicketManager`, add `delete_ticket(self, ticket_id)`

Once that's in place, clicking **Delete** will call your Flask endpoint, remove the record, and immediately refresh the table. Let me know if you run into any errors or need tweaks!

part 1 and 2 are perfect, but please rewrite part 3 considering this code for the logic: import os

```
import sqlite3
from model.ticket import Ticket

class SqlDataService:
    def __init__(self):
        # Setting database path dynamically
        base_dir = os.path.dirname(os.path.abspath(__file__))
        project_root = os.path.abspath(os.path.join(base_dir, '..', '..'))
        self.path = os.path.join(project_root, 'database.db')

        # Ticket Constants
        self.table_tickets = 'tickets'
        self.field_tickets_id = 'id'
        self.field_tickets_artist = 'artist'
        self.field_tickets_location = 'location'
        self.field_tickets_price = 'price'

    def execute_query(self, query, values = None):
        conn = None
        try:
            conn = sqlite3.connect(self.path)
            cursor = conn.cursor()

            if values:
                cursor.execute(query, values)
            else:
                cursor.execute(query)

            if cursor.description is not None:
                results = cursor.fetchall()
                return results
            else:
                conn.commit()
                return 0

        except sqlite3.Error as e:
            print("SQLite error:", e)
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()

    # Ticket Functions
    def retrieve_tickets(self):
        query = f'SELECT * FROM {self.table_tickets} ORDER BY {self.field_tickets_id} DESC'
        print(query)

        entries = self.execute_query(query)
        results = [Ticket(e[1], e[2], e[3], e[0]) for e in entries]
        return results

    def insert_ticket(self, new_ticket):
        query = f'INSERT INTO {self.table_tickets} ({self.field_tickets_artist}, {self.field_tickets_location}, {self.field_tickets_price}) VALUES (?, ?, ?)'
        values = (new_ticket.artist, new_ticket.location, new_ticket.price)
        print(query)

        result = self.execute_query(query, values)
        return result

    # TASK 2
    def delete_ticket(self):
        pass from model.ticket import Ticket
from data.sql_data_service import SqlDataService

class TicketManager:

    @staticmethod
    def retrieve_tickets():
        data_service = SqlDataService()
        tickets = data_service.retrieve_tickets()
        return [{'id': t.ticket_id, 'artist': t.artist, 'location': t.location, 'price': t.price} for t in tickets]

    @staticmethod
    def insert_ticket(data):
        artist = data.get('artist')
        location = data.get('location')
        price = data.get('price')

        data_service = SqlDataService()
        return data_service.insert_ticket(Ticket(artist, location, price))

    # TASK 2
    @staticmethod
    def delete_ticket(data):
        pass
```

Here’s the **updated Part 3** using your existing `SqlDataService` and `TicketManager` structure.

3a. `data/sql_data_service.py`

Add the **`delete_ticket`** method to your **`SqlDataService`** class (replace the **`pass`**):

```python
    # TASK 2
    def delete_ticket(self, ticket_id):
        query = f'DELETE FROM {self.table_tickets} WHERE {self.field_tickets_id} = ?'
        values = (ticket_id,)
        # execute_query will commit and return 0 on non-SELECT
        return self.execute_query(query, values)
```

3b. `business/ticket_manager.py`

Implement the static **`delete_ticket`** in **`TicketManager`** (replace the **`pass`**):

```python
    # TASK 2
    @staticmethod
    def delete_ticket(data):
        # Expecting data = {'id': <ticket_id>}
        ticket_id = data.get('id')
        if ticket_id is None:
            raise ValueError("No ticket ID provided for deletion.")

        data_service = SqlDataService()
        # returns 0 on success (per execute_query contract)
        result = data_service.delete_ticket(ticket_id)
        return result
```

Summary of changes

File	Change
`sql_data_service.py`	Add **`delete_ticket(self, ticket_id)`** method using **`DELETE`** SQL
`ticket_manager.py`	Implement **`delete_ticket(data)`** to extract **`id`** & call service

With these in place, your front-end's **`deleteTicket(id)`** → Flask endpoint → **`TicketManager.delete_ticket(...)`** → **`SqlDataService.delete_ticket(...)`** chain will remove the row from your SQLite **`tickets`** table.

please verify that eveyrthing should work correctly and eventually fix it: // Delete Ticket buttons functionality

```
    function addFunctionalityDeleteBtns() {
        document.querySelectorAll('.delete-ticket-btn').forEach(button => {
            button.addEventListener('click', async function () {
                const id = this.id;                  // the ticket's id
                await deleteTicket(id);              // call the DELETE API
                await updateTicketsTable();          // refresh the table view
            });
        });
        } async function updateTicketsTable() {
            ticketsTableBody.innerHTML = "";

            const tickets = await getTickets();
            for (const ticket of tickets) {
                const entryHtml =
                    <tr>
                        <td>${ticket.artist}</td>
                        <td>${ticket.location}</td>
                        <td>${ticket.price}$</td>
                        <td class="text-end">
                            <button id="${ticket.id}" type="button" class="btn btn-danger menu-btn delete-ticket-btn">Delete</button>
                        </td>
                    </tr>
                ;
                ticketsTableBody.innerHTML += entryHtml;
            }
            addFunctionalityDeleteBtns();
        } // TASK 2
async function deleteTicket(id) {
    try {
        const response = await fetch('http://127.0.0.1:5000/delete_ticket', {
            method: 'DELETE',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ id: id }),
        });

        if (!response.ok) {
            throw new Error('Network response was not ok');
        }

        const data = await response.json();
        console.log('Deleted:', data);
    } catch (error) {
        console.error('Error deleting ticket:', error);
    }
} # TASK 2
@app.route('/delete_ticket', methods=['DELETE'])
def delete_ticket():
    data = request.json
    ticket_id = data.get('id')
    tm = TicketManager()
    try:
        tm.delete_ticket(ticket_id)
        return jsonify({'message': 'OK'})
    except Exception as e:
        return jsonify({'message': str(e)}), 400 def delete_ticket(data):
        # Expecting data = {'id': <ticket_id>}
        ticket_id = data.get('id')
        if ticket_id is None:
            raise ValueError("No ticket ID provided for deletion.")

        data_service = SqlDataService()
        # returns 0 on success (per execute_query contract)
        result = data_service.delete_ticket(ticket_id)
        return result     # TASK 2
    def delete_ticket(self, ticket_id):
        query = f'DELETE FROM {self.table_tickets} WHERE {self.field_tickets_id} = ?'
        values = (ticket_id,)
        # execute_query will commit and return 0 on non-SELECT
        return self.execute_query(query, values)
```

I walked through the full delete-path and spotted one mismatch: your Flask route is calling `TicketManager.delete_ticket(ticket_id)` but your `TicketManager.delete_ticket` as written expects a **data dict**. Let's align them so everything flows smoothly:

1. `business/ticket_manager.py`

Make `delete_ticket` take the raw `ticket_id` instead of a dict:

```python
from data.sql_data_service import SqlDataService

class TicketManager:

    # … retrieve_tickets / insert_ticket …

    # TASK 2 — now takes ticket_id directly
    @staticmethod
    def delete_ticket(ticket_id):
        if ticket_id is None:
            raise ValueError("No ticket ID provided for deletion.")
        data_service = SqlDataService()
        return data_service.delete_ticket(ticket_id)
```

2. `data/sql_data_service.py`

(unchanged from before)

```python
    # TASK 2
    def delete_ticket(self, ticket_id):
        query = f'DELETE FROM {self.table_tickets} WHERE {self.field_tickets_id} = ?'
        values = (ticket_id,)
        return self.execute_query(query, values)
```

3. `server/app.py`

Your route can remain as-is, since it now matches the manager's signature:

```python
@app.route('/delete_ticket', methods=['DELETE'])
def delete_ticket():
    data = request.json
    ticket_id = data.get('id')
    tm = TicketManager()
    try:
        tm.delete_ticket(ticket_id)
        return jsonify({'message': 'OK'})
    except Exception as e:
        return jsonify({'message': str(e)}), 400
```

Verify end-to-end

1. **Click "Delete"** → JS picks up button **`id`**, calls

   ```js
   await deleteTicket(id);
   ```

2. **`deleteTicket(id)`** (in **`api_script.js`**) →

   ```js
   fetch('/delete_ticket', { method: 'DELETE', body: JSON.stringify({ id }) })
   ```

3. **Flask route** → **`ticket_id = data['id']`** →

   ```python
   TicketManager.delete_ticket(ticket_id)
   ```

4. **TicketManager** → calls

   ```python
   SqlDataService.delete_ticket(ticket_id)
   ```

5. **SQL** →

   ```sql
   DELETE FROM tickets WHERE id = ?
   ```

6. JS refreshes **updateTicketsTable()** → table re-renders without that row.

With those three snippets in place you should have a fully working delete flow. Let me know if any errors pop up!

that was perfect. now let's continue with task 3 in which we need to update the values of the ticket prices using sql queries on a sqllite database. the current content of the database is: 1 Artist A Location A 30
2 Artist B Location B 40
3 Artist C Location C 50
8 Artist-Test Location-Test 999 and the task is described as follows: Ticket A corresponds to Artist A, Ticket B to Artist b, ticket c to artist c. Ticket A,b and c have been listed 30 days prior to the event. there are 3 days left till event A, 10 days till event B, and 15 dats till event C. For event A there are 100 tickets available out of 500, for event B, 800 were sold out of 1200; for event C, 200 were not sold out of 1800. Ticket A is VIP, while Tickets B and C are Standard tickets. Managers have received word from the company directors that there should not be more than 30% tickets cost 20% more than the base price of a standard ticket (1.0 multiplier). the formula for the dynamic price is = newPrice = P0 * (1+markupDynamic) * seatType. the markupDynamic factor is = c * (1 - (Vt/Vo)) where : P0 is bace price, seat type is either VIP or standard, c is the markup constant %; Vt is the current variable and V0 is the initial variable. please think this thourgh because it is not an easy task and you need to focus very much on understanding the request of the ask. before proceeding, ask me a few questions to make sure to understood correctly what to do next

Before I dive into writing the SQL to recalculate and update each ticket's price, I want to make absolutely sure I've captured all the parameters correctly. A few questions:

1. **Markup Constant (c)**
    - You mention "managers … that there should not be more than 30% tickets cost 20% more than the base price of a standard ticket."
        - Does that imply **c = 0.20** (20%)?
        - Or is **c = 0.30** (30%)?
2. **Seat-Type Multiplier**
    - For **VIP** tickets: what multiplier should we use? (e.g. 1.2×, 1.5×, etc.)
    - For **Standard** tickets: I assume multiplier = 1.0, correct?
3. **V_0 and V_t** (initial vs. current variable)
    - Should V_0 be "total tickets allocated for the event" (e.g. 500 for A, 1200 for B, 1800 for C)?
    - And V_t the number **sold so far** (i.e. 400 sold for A, 1200 sold for B, 1600 sold for C)?
      (You gave "100 available out of 500" → so 400 sold; "800 were sold out of 1200" → that sounds like 800 sold, 400 remaining; "200 were not sold out of 1800" → so 1600 sold.)
4. **Which rows to update**
    - We have a fourth row (`id=8`, "Artist-Test"). Should that one be skipped, or should we treat it as one of the three events?
5. **Persisting new prices**
    - Do we simply `UPDATE tickets SET price = <newPrice>` for each applicable row?
    - Or would you prefer adding a new column (e.g. `dynamic_price`) instead of overwriting the original?

Once I have these details, I'll craft the precise `UPDATE` statements (and any helpers) to recalculate and store each ticket's new price.

> 1. there should not be more than 30% price markup based on limited time and 20% price markup based on ticket availability. this is only the information I have, think it through to understand it and realise it. 2. VIP tickets cost 20% more than standard. 3. yes, and yes 4. skip it 5. update the current price, do not create new columns

Based on your clarifications, here's the approach I'll take:

1. **Two separate dynamic-markup factors**
    - **Time-based**:

$$\text{time\_mark} = 0.30 \times \left(1 - \frac{\text{days\_left}}{30}\right)$$

    - **Availability-based**:

$$\text{avail\_mark} = 0.20 \times \left(1 - \frac{\text{available}}{\text{initial\_total}}\right)$$

2. **Seat-type multiplier**
    - **VIP**: 1.2
    - **Standard**: 1.0
3. **New price formula**

$$\text{newPrice} = P_0 \times \left(1 + \text{time\_mark} + \text{avail\_mark}\right) \times \text{seat\_multiplier}$$

4. **Skip** the `id=8` test row and **overwrite** the existing `price` column.

Here are the **three** SQL `UPDATE` statements you can run—either directly in your SQLite client or via your Python data-service—to recalculate & persist the new prices for the three real events:

```sql
-- EVENT A (Artist A, VIP)
--  30 days total window, 3 days left, 100 available of 500
UPDATE tickets
SET price = ROUND(
    price
    * (1
       + (0.30 * (1 - 3.0/30))
       + (0.20 * (1 - 100.0/500))
      )
    * 1.2
  , 2)
WHERE artist = 'Artist A';


-- EVENT B (Artist B, Standard)
--  30 days window, 10 days left, 400 available of 1200
UPDATE tickets
```

```
SET price = ROUND(
    price
    * (1
       + (0.30 * (1 - 10.0/30))
       + (0.20 * (1 - 400.0/1200))
      )
    * 1.0
  , 2)
WHERE artist = 'Artist B';


-- EVENT C (Artist C, Standard)
--  30 days window, 15 days left, 200 available of 1800
UPDATE tickets
SET price = ROUND(
    price
    * (1
       + (0.30 * (1 - 15.0/30))
       + (0.20 * (1 - 200.0/1800))
      )
    * 1.0
  , 2)
WHERE artist = 'Artist C';
```

How to apply in code

If you'd rather invoke this from Python, you could add a method to your `SqlDataService`:

```python
def update_dynamic_prices(self):
    stmts = [
        # (copy the three UPDATEs above, as strings)
    ]
    for sql in stmts:
        self.execute_query(sql)
```

and then call that from a new `TicketManager.update_prices()` endpoint or a one-off script.

Let me know if you'd like the Python wrapper too—or if anything needs tweaking!