# 6-ba

helo\

Hello there! How can I assist you today with software engineering or business-related questions?

help me make this:In this task you are expected to create an HTML form that allows you to add new tickets on the platform.
Tickets have the following representation:
••••ID
Artist
Location
Price
However, you are only required to add 3 input fields as the ID is automatically generated as a unique
identifier. In addition, there should be 2 buttons present, one to reset (clear) the input fields and one to
submit the ticket.
All JavaScript functions that ensure the functionality of the components you introduce, have been created. In order to ensure this functionality you must use the following IDs for your components:
• Artist input field ID: artist-input
• Location input field ID: location-input
• Price input field ID: price-input
• Reset button ID: ticket-add-reset-btn
• Submit button ID: ticket-add-submit-btn
After finishing your implementation you can test it by introducing the following ticket:
• Artist: Artist-Test
• Location: Location-Test
• Price: 999
You will be graded on:
1. Functionality
2. Page Appearance (you are expected to make the content introduced fit the page)
3. Written code

Here's a self-contained HTML snippet you can drop into your project. It defines a simple, responsive form with the exact IDs your existing JavaScript expects. Feel free to tweak the styles to match your site's look and feel.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>Add Ticket</title>
    <style>
      /* Center the form on the page */
      body {
        font-family: Arial, sans-serif;
        background: #f5f5f5;
        display: flex;
        align-items: center;
        justify-content: center;
        height: 100vh;
        margin: 0;
      }
      .ticket-form {
        background: #fff;
        padding: 2rem;
        border-radius: 8px;
        box-shadow: 0 2px 8px rgba(0,0,0,0.1);
        width: 100%;
        max-width: 400px;
      }
      .ticket-form h2 {
        margin-top: 0;
        margin-bottom: 1rem;
        font-size: 1.5rem;
        text-align: center;
      }
      .ticket-form .form-group {
        margin-bottom: 1rem;
      }
      .ticket-form label {
        display: block;
        margin-bottom: 0.3rem;
        font-weight: bold;
      }
      .ticket-form input {
        width: 100%;
        padding: 0.5rem;
        font-size: 1rem;
        border: 1px solid #ccc;
        border-radius: 4px;
        box-sizing: border-box;
      }
      .ticket-form .button-group {
        display: flex;
        justify-content: space-between;
        margin-top: 1.5rem;
      }
      .ticket-form button {
        flex: 1;
        padding: 0.7rem;
        font-size: 1rem;
        border: none;
        border-radius: 4px;
        cursor: pointer;
      }
      .ticket-form button + button {
        margin-left: 1rem;
      }
      #ticket-add-reset-btn {
        background: #e0e0e0;
      }
      #ticket-add-submit-btn {
        background: #4CAF50;
        color: #fff;
      }
    </style>
  </head>
  <body>

    <form class="ticket-form" id="ticket-add-form" novalidate>
      <h2>Add New Ticket</h2>

      <div class="form-group">
        <label for="artist-input">Artist</label>
        <input
          type="text"
          id="artist-input"
          name="artist"
          placeholder="e.g. Artist-Test"
          required
        />
      </div>

      <div class="form-group">
        <label for="location-input">Location</label>
        <input
          type="text"
          id="location-input"
          name="location"
          placeholder="e.g. Location-Test"
          required
        />
      </div>

      <div class="form-group">
        <label for="price-input">Price</label>
        <input
          type="number"
          id="price-input"
          name="price"
          placeholder="e.g. 999"
          min="0"
          required
        />
      </div>

      <div class="button-group">
        <button
          type="button"
          id="ticket-add-reset-btn"
        >
          Reset
        </button>
        <button
          type="button"
          id="ticket-add-submit-btn"
        >
          Submit
        </button>
      </div>
    </form>

    <!-- Your existing JS functions will hook into these IDs -->
    <script src="path/to/your/ticket-functions.js"></script>
  </body>
  </html>
```

## Notes

1. **IDs**
    - **`artist-input`**
    - **`location-input`**
    - **`price-input`**
    - **`ticket-add-reset-btn`**
    - **`ticket-add-submit-btn`**
2. **Buttons** are **`type="button"`** so they won’t trigger a native form submission or reset — your JavaScript should listen for clicks on these IDs and run the appropriate logic.

3. **Styling**
    - A simple, mobile-friendly centered card
    - Inputs stretch to fill the width
    - Clear visual distinction between Reset and Submit

---

**To test:**

1. Enter **Artist-Test** in the Artist field
2. Enter **Location-Test** in the Location field
3. Enter **999** in the Price field
4. Click **Submit** and verify your JS adds the ticket
5. Click **Reset** to clear the form

Let me know if you need any tweaks or if your JavaScript functions have specific callback names we should wire up!

lets start again. i will send you all the info first and then i will send the instructions

Sure thing! I'm ready—go ahead and send over all the info, and then the instructions.

this is the first part: You have access to the following tools to achieve your goals:
• VSCode: to edit and run the code
• ChatGPT: to converse with the AI model with the aim of completing your tasks
• Postman: to test server requests without client implementation
• DB Browser for SQlite: to visualize database data and run SQL queries.

Got it—tools noted. What's next?

second: Below is a preview of the programming languages / technologies used to develop the platform:
• Client layer: HTML, CSS, JavaScript and Bootstrap
• Server layer: Python (Flask)
• Database layer: SQL
• Observation: The Graphical User Interface (GUI) follows the Single-Page Application (SPA) model.

Thanks for the tech stack overview. What's next?

and now the first task: A.1
Task 1: Add ”Insert” Ticket HTML form
In this task you are expected to create an HTML form that allows you to add new tickets on the platform.
Tickets have the following representation:
••••ID
Artist
Location
Price
However, you are only required to add 3 input fields as the ID is automatically generated as a unique
identifier. In addition, there should be 2 buttons present, one to reset (clear) the input fields and one to
submit the ticket.
All JavaScript functions that ensure the functionality of the components you introduce, have been created. In order to ensure this functionality you must use the following IDs for your components:
• Artist input field ID: artist-input
• Location input field ID: location-input
• Price input field ID: price-input
• Reset button ID: ticket-add-reset-btn
• Submit button ID: ticket-add-submit-btn
After finishing your implementation you can test it by introducing the following ticket:
• Artist: Artist-Test
• Location: Location-Test
• Price: 999
You will be graded on:
1. Functionality
2. Page Appearance (you are expected to make the content introduced fit the page)
3. Written code

Here’s a Bootstrap-based form snippet you can drop into your SPA’s HTML (e.g. in your main template or a dedicated “Add Ticket” component). It uses exactly the IDs your existing JS needs, includes Reset and Submit buttons, and is styled to fit neatly into a Bootstrap layout.

```html
<!-- AddTicketForm.html -->
<div class="container my-4">
  <div class="card shadow-sm">
    <div class="card-header bg-primary text-white">
      <h5 class="mb-0">Add New Ticket</h5>
    </div>
    <div class="card-body">
      <form id="ticket-add-form" novalidate>
        <!-- Artist -->
        <div class="mb-3">
          <label for="artist-input" class="form-label">Artist</label>
          <input
            type="text"
            id="artist-input"
            class="form-control"
            placeholder="e.g. Artist-Test"
            required
          />
        </div>
        <!-- Location -->
        <div class="mb-3">
          <label for="location-input" class="form-label">Location</label>
          <input
            type="text"
            id="location-input"
            class="form-control"
            placeholder="e.g. Location-Test"
            required
          />
        </div>

        <!-- Price -->
        <div class="mb-3">
          <label for="price-input" class="form-label">Price</label>
          <input
            type="number"
            id="price-input"
            class="form-control"
            placeholder="e.g. 999"
            min="0"
            required
          />
        </div>

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

## Explanation & Next Steps

1. **IDs for JS hooks**
    - **`artist-input`**
    - **`location-input`**
    - **`price-input`**
    - **`ticket-add-reset-btn`**
    - **`ticket-add-submit-btn`**
2. **Bootstrap classes**
    - `.container my-4` to give spacing
    - `.card` to wrap the form in a neat panel
    - `.form-label` and `.form-control` for consistent styling
    - Button classes for visual hierarchy

3. **Button types**
   Both buttons are `type="button"` so they won't trigger native form submit/reset; your existing JS listeners should handle their click events.
4. **Testing**
   1. Load your SPA and navigate to wherever you include this form.
   2. Fill in "Artist-Test", "Location-Test", and "999".
   3. Click **Submit** → your JS should pick up the values and call the backend.
   4. Click **Reset** → fields should clear.

Let me know if you need help wiring up the event listeners or integrating this into your Flask templates!

now give step by step instructions

Here's a clear, step-by-step guide to add your "Insert Ticket" form into the SPA:

---

# 1. Locate the Right Template/Component File

1. In your project's **client layer**, identify where new page content lives.
   - If you're using a single HTML file (e.g. `index.html`), you'll drop it there.
   - If you've modularized with partials or components (e.g. an `AddTicket.html` or a JavaScript-rendered view), open that.

# 2. Ensure Bootstrap Is Loaded

Make sure your `<head>` (or your main layout template) includes the Bootstrap CSS and JS:

```html
<!-- in <head> -->
<link
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.4.0/dist/css/bootstrap.min.css"
  rel="stylesheet"
/>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.4.0/dist/js/bootstrap.bundle.min.js">
</script>
```

# 3. Insert the Form Markup

Inside your chosen container (for example, within `<div id="app"></div>`), paste the form snippet:

```html
<div class="container my-4">
  <div class="card shadow-sm">
    <div class="card-header bg-primary text-white">
      <h5 class="mb-0">Add New Ticket</h5>
    </div>
    <div class="card-body">
      <form id="ticket-add-form" novalidate>
        <!-- Artist -->
        <div class="mb-3">
          <label for="artist-input" class="form-label">Artist</label>
          <input
            type="text"
            id="artist-input"
            class="form-control"
            placeholder="e.g. Artist-Test"
            required
          />
        </div>

        <!-- Location -->
        <div class="mb-3">
          <label for="location-input" class="form-label">Location</label>
          <input
            type="text"
            id="location-input"
            class="form-control"
            placeholder="e.g. Location-Test"
            required
          />
        </div>

        <!-- Price -->
        <div class="mb-3">
          <label for="price-input" class="form-label">Price</label>
          <input
            type="number"
            id="price-input"
            class="form-control"
            placeholder="e.g. 999"
            min="0"
            required
          />
        </div>

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

## 4. Wire Up Your JavaScript

You said all JS functions already exist. Now connect them:

1. In your main JS file (e.g. **`app.js`** or wherever you listen for events), add:

```js
document
  .getElementById("ticket-add-reset-btn")
  .addEventListener("click", clearTicketForm);

document
  .getElementById("ticket-add-submit-btn")
  .addEventListener("click", submitNewTicket);
```

2. Ensure those functions—**`clearTicketForm`** and **`submitNewTicket`** (or whatever names your project uses)—reference the inputs by ID:

```js
function clearTicketForm() {
  document.getElementById("artist-input").value = "";
  document.getElementById("location-input").value = "";
  document.getElementById("price-input").value = "";
}

function submitNewTicket() {
  const artist = document.getElementById("artist-input").value;
  const location = document.getElementById("location-input").value;
  const price = +document.getElementById("price-input").value;
  // ...your existing fetch()/axios call to POST to Flask backend...
}
```

## 5. Integrate with Flask (Server Layer)

1. **If** your form submits via AJAX/fetch: ensure your Flask route exists, e.g.:

```python
@app.route("/api/tickets", methods=["POST"])
def add_ticket():
    data = request.json
    # insert into SQLite...
    return jsonify(success=True, ticket_id=new_id)
```

2. **Else** if you plan to do a traditional form POST, change the button to **`type="submit"`**, set the **`<form action="/tickets/add" method="post">`**, and handle the route accordingly.

*(But since you said JS functions exist, you're likely using AJAX.)*

## 6. Test Your Form

1. Run your Flask server (**`flask run`** or however you launch it).
2. Open your SPA in the browser.
3. Fill in:
   - **Artist**: **`Artist-Test`**
   - **Location**: **`Location-Test`**
   - **Price**: **`999`**

4. Click **Submit** → watch your Network tab (Postman or browser DevTools) to confirm a successful POST.
5. Click **Reset** → verify all input fields clear.

## 7. Style Tweaks (Optional)

- Adjust margins, colors, or typography in your CSS or inline `<style>` to perfectly match your app's design system.
- If you need to hide/show this form as part of your SPA navigation, wrap it in a `<div id="add-ticket-panel" style="display:none">` and toggle visibility in your routing logic.

---

That's it! You now have a fully integrated "Insert Ticket" HTML form with the required IDs, hooked into your existing JavaScript and Flask backend. Let me know if you need help with any specific step.

where should i insert the markup in the code

You'll drop the form's HTML into whatever top-level template or "view container" your SPA renders into. Here are two common patterns—and where to put it in each:

---

## 1. Single `index.html` SPA

If your whole frontend lives in one file, something like:

```html
<!DOCTYPE html>
<html lang="en">
<head> …</head>
<body>
  <!-- ← Your existing navbar/header -->
  <nav>…</nav>

  <!-- ← Main app mount point -->
  <div id="app">
    <!-- ← INSERT THE FORM HERE -->
  </div>

  <!-- ← Your existing footer -->
  <footer>…</footer>

  <script src="…your bundle…"></script>
</body>
</html>
```

**What to do:**
Paste the `<div class="container my-4">…</div>` form snippet directly inside the `<div id="app">`. For example:

```html
<div id="app">
  <!-- Add Ticket Form -->
  <div class="container my-4">
    <!-- …the card + form markup… -->
  </div>
  <!-- Other SPA sections/components -->
</div>
```

---

## 2. Flask + Jinja2 Templates

Often you'll have a base layout (`base.html`) and individual page templates:

**`templates/base.html`**

```html
<!DOCTYPE html>
<html lang="en">
<head> …common CSS/JS… </head>
<body>
  {% include 'navbar.html' %}
  <main class="py-4">
    <!-- This is where child templates put their page content: -->
    {% block content %}{% endblock %}
  </main>
  {% include 'footer.html' %}
</body>
</html>
```

**`templates/tickets.html`** (or your "add ticket" page)

```html
{% extends 'base.html' %}
{% block content %}
  <!-- ← INSERT THE FORM HERE -->
  <div class="container my-4">
    <!-- …the card + form markup… -->
  </div>
{% endblock %}
```

Then your Flask route would render `tickets.html`:

```python
@app.route('/tickets')
def tickets_page():
    return render_template('tickets.html')
```

---

### Checklist

1. **Locate**:
    - **SPA index**: find `<div id="app">`
    - **Flask/Jinja**: find the `{% block content %}` in your child template

2. **Paste** the entire `<div class="container my-4"> … </div>` snippet inside that spot.
3. **Save** and **reload**. The form should now appear where you expect in your running app.

Let me know if your project uses a different structure—happy to point you to the exact file!

he first task is complete.

Great! Glad the form is in place and working. What would you like to tackle next?

i will now send instructions for the second task

Perfect—please go ahead and share the instructions for the second task.

In this task you are expected to implement the logic behind the □delete□ ticket operation. Once the delete button has been clicked, the platform needs to send a request to the server to delete the
ticket
from the database. Once the request has been finalized the table needs to refresh. You are required to implement both client and server side logic.
To test your implementation you can introduce more false tickets using the form implemented in the
the
first task. Once you have finished your implementation please delete the following ticket (and all other
other
tickets created for testing):
• Artist: Artist-Delete
17
APPENDIX A. EXPERIMENT INSTRUCTIONS
• Location: Location-Delete
• Price: -1
You will be graded on:
1. Functionality
2. Written code

Here's a concrete, step-by-step plan to wire up the “Delete” operation end-to-end. I'm assuming you already have:

- A JS function that fetches and renders your tickets table (e.g. `loadTickets()`),
- A Flask/SQLite setup for storing tickets,
- And that each rendered row in your table has a “Delete” button placeholder (we'll add that, too).

---

# 1. Database Model / Helper

1. In whichever module you handle DB operations (e.g. **`models.py`** or a **`db_helpers.py`**), add a function to delete by ID:

```
python

def delete_ticket(ticket_id):
    conn = sqlite3.connect('tickets.db')
    cur = conn.cursor()
    cur.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
    conn.commit()
    conn.close()
```

2. Make sure your tickets table's primary key column is named **`id`**.

---

## 2. Server-Side: Flask Endpoint

1. In your Flask app (e.g. **`app.py`**), import the helper and add:

```
python

from flask import jsonify

@app.route('/api/tickets/<int:ticket_id>', methods=['DELETE'])
def api_delete_ticket(ticket_id):
    try:
        delete_ticket(ticket_id)
        return jsonify(success=True), 200
    except Exception as e:
        return jsonify(success=False, error=str(e)), 500
```

2. If you're not using DELETE verbs, you can also do a POST route like **`/api/tickets/delete`** that reads JSON **`{ id: ... }`**.

---

## 3. Client-Side: Render "Delete" Buttons

1. In your table rendering code (e.g. inside **`loadTickets()`**), for each ticket row inject a button:

```
html

<button
  class="btn btn-sm btn-outline-danger ticket-delete-btn"
  data-id="{{ticket.id}}"
>
  Delete
</button>
```

2. If you're using template literals in JS:

```
js

rows += `
  <tr>
    <td>${t.id}</td>
    <td>${t.artist}</td>
    <td>${t.location}</td>
    <td>${t.price}</td>
    <td>
```

```
        <button
          class="btn btn-sm btn-outline-danger ticket-delete-btn"
          data-id="${t.id}"
        >
          Delete
        </button>
      </td>
    </tr>`;
```

---

## 4. Client-Side: Hook Up the Click Handler

1. After you finish injecting the rows into your table's **`<tbody>`**, run:

```js
document.querySelectorAll('.ticket-delete-btn').forEach(btn => {
  btn.addEventListener('click', () => {
    const id = btn.getAttribute('data-id');
    deleteTicket(id);
  });
});
```

2. Implement **`deleteTicket(id)`**:

```js
async function deleteTicket(id) {
  if (!confirm('Are you sure you want to delete this ticket?')) return;

  const resp = await fetch(`/api/tickets/${id}`, {
    method: 'DELETE'
  });
  const json = await resp.json();
  if (json.success) {
    // remove from DOM or reload the table:
    loadTickets();
  } else {
    alert('Delete failed: ' + (json.error || 'unknown error'));
  }
}
```

---

## 5. Table Refresh

- You've called **`loadTickets()`** after a successful delete.
- Ensure **`loadTickets()`** clears out the old **`<tbody>`** contents before reinjecting rows.

---

## 6. Test Your Implementation

1. **Create** a test ticket via your "Add Ticket" form:
    - Artist: **`Artist-Delete`**
    - Location: **`Location-Delete`**
    - Price: **`-1`**
2. **Open** your tickets view; confirm it appears in the table.

3. **Click** the Delete button next to that row.
4. **Confirm** the browser prompt.
5. **Observe** the row disappear and/or a console/network check showing a `DELETE /api/tickets/<id>` → `200`.
6. **Verify** in DB Browser for SQLite that no record with that ID remains.

---

Once you've gone through these steps, your delete-ticket flow will be fully wired up client-to-server and back. Let me know if you run into errors or need more detail on any specific piece!

how to make sure that the tickets primary key colum is named id

To ensure your tickets table uses a column named `id` as its primary key in SQLite, you have two paths depending on whether you're creating the table from scratch or need to update an existing one.

---

# 1. Creating the Table (or Re-creating It)

Use a DDL statement that explicitly defines `id` as an integer primary key with auto-increment:

```sql
CREATE TABLE IF NOT EXISTS tickets (
  id       INTEGER PRIMARY KEY AUTOINCREMENT,
  artist   TEXT    NOT NULL,
  location TEXT    NOT NULL,
  price    REAL    NOT NULL
);
```

- **`id INTEGER PRIMARY KEY AUTOINCREMENT`**
    - Declares `id` as the primary key.
    - Guarantees it's unique and automatically increments on each insert.

Run this in DB Browser for SQLite (Execute SQL tab) or from your Flask startup script.

---

# 2. Verifying the Column Name

If the table already exists, you can check its schema:

```sql
PRAGMA table_info('tickets');
```

It will return rows like:

| cid | name | type | notnull | dflt_value | pk |
|---|---|---|---|---|---|
| 0 | id | INTEGER | 1 | NULL | 1 |
| 1 | artist | TEXT | 1 | NULL | 0 |
| … | … | … | … | … | … |

- Look for the row where `name = 'id'` and `pk = 1`.
- If you see a different primary key column (e.g. `ticket_id`), you'll need to migrate.

---

## 3. Migrating an Existing Table

SQLite doesn't support renaming columns or changing primary keys in place. You'll need to:

1. **Create a new table** with the correct schema:

   ```
   sql

   CREATE TABLE tickets_new (
     id       INTEGER PRIMARY KEY AUTOINCREMENT,
     artist   TEXT    NOT NULL,
     location TEXT    NOT NULL,
     price    REAL    NOT NULL
   );
   ```

2. **Copy the data** over:

   ```
   sql

   INSERT INTO tickets_new (artist, location, price)
     SELECT artist, location, price
     FROM tickets;
   ```

3. **Drop the old table** and rename the new one:

   ```
   sql

   DROP TABLE tickets;
   ALTER TABLE tickets_new RENAME TO tickets;
   ```

4. **Re-verify** with `PRAGMA table_info('tickets')`.

---

## 4. Integrate in Your Flask App

If you run your DDL as part of your application startup, you can add to `app.py` (before handling requests):

```
python

import sqlite3

def init_db():
    conn = sqlite3.connect('tickets.db')
    conn.execute('''
      CREATE TABLE IF NOT EXISTS tickets (
```

```
        id       INTEGER PRIMARY KEY AUTOINCREMENT,
        artist   TEXT    NOT NULL,
        location TEXT    NOT NULL,
        price    REAL    NOT NULL
      );
    ''')
    conn.commit()
    conn.close()

if __name__ == '__main__':
    init_db()
    app.run(debug=True)
```

This way, every time you start your Flask server, it ensures the **`tickets`** table exists with an **`id`** primary key column.

---

With these steps you'll guarantee that your tickets table always has an **`id`** column serving as its primary key.

i need a bit more help with task 2. lets start over with more, clear instructions from you

Let's break Task 2 ("Delete Ticket") down into a very explicit, file-by-file, step-by-step process. I'll assume your server code lives in **`app.py`**, your database helpers in **`db_helpers.py`**, and your client code in **`static/js/app.js`**, with your HTML table in **`templates/tickets.html`**. Adjust paths as needed.

---

# 1. Confirm Your DB Schema Has `id` as PK

1. Open **`db_helpers.py`** (or wherever you initialize your DB).
2. Ensure you have a function like this to create your table:

   ```
   python

   import sqlite3

   def init_db():
       conn = sqlite3.connect('tickets.db')
       conn.execute('''
         CREATE TABLE IF NOT EXISTS tickets (
           id       INTEGER PRIMARY KEY AUTOINCREMENT,
           artist   TEXT    NOT NULL,
           location TEXT    NOT NULL,
           price    REAL    NOT NULL
         );
       ''')
       conn.commit()
       conn.close()
   ```

3. If you haven't already, call **`init_db()`** at server startup (see step 3 below).

---

# 2. Add a "Delete" Helper in `db_helpers.py`

Still in **`db_helpers.py`**, add:

```python
def delete_ticket(ticket_id):
    conn = sqlite3.connect('tickets.db')
    cur = conn.cursor()
    cur.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
    conn.commit()
    conn.close()
```

---

## 3. Wire Up the Flask DELETE Endpoint in `app.py`

1. At the top of **`app.py`**, import your new helper:

   ```python
   from db_helpers import init_db, delete_ticket
   ```

2. Ensure your DB is initialized when the app starts:

   ```python
   if __name__ == '__main__':
       init_db()
       app.run(debug=True)
   ```

3. Below your other **`@app.route`** handlers, add:

   ```python
   from flask import jsonify

   @app.route('/api/tickets/<int:ticket_id>', methods=['DELETE'])
   def api_delete_ticket(ticket_id):
       try:
           delete_ticket(ticket_id)
           return jsonify(success=True), 200
       except Exception as e:
           return jsonify(success=False, error=str(e)), 500
   ```

   - This listens at **`DELETE /api/tickets/123`** and removes that row.

---

## 4. Add “Delete” Buttons in Your Tickets Table

Open **`templates/tickets.html`** (or whatever template renders your table). Locate the loop that prints each **`<tr>`** for a ticket, for example:

```html
<tbody id="tickets-table-body">
  {% for t in tickets %}
  <tr data-id="{{ t.id }}">
    <td>{{ t.id }}</td>
    <td>{{ t.artist }}</td>
    <td>{{ t.location }}</td>
    <td>{{ t.price }}</td>
    <td>
```

```
      <!-- add this Delete button: -->
      <button class="btn btn-sm btn-outline-danger ticket-delete-btn">
        Delete
      </button>
    </td>
  </tr>
  {% endfor %}
</tbody>
```

- We'll attach a click listener to every `.ticket-delete-btn`.

---

## 5. Hook Up Client-Side Logic in `static/js/app.js`

1. Ensure you have a function to reload your tickets table:

```js
async function loadTickets() {
  const res = await fetch('/api/tickets');
  const data = await res.json();
  const tbody = document.getElementById('tickets-table-body');
  tbody.innerHTML = '';                // clear existing rows
  data.forEach(t => {
    const tr = document.createElement('tr');
    tr.dataset.id = t.id;
    tr.innerHTML = `
      <td>${t.id}</td>
      <td>${t.artist}</td>
      <td>${t.location}</td>
      <td>${t.price}</td>
      <td>
        <button class="btn btn-sm btn-outline-danger ticket-delete-btn">
          Delete
        </button>
      </td>`;
    tbody.appendChild(tr);
  });
  attachDeleteHandlers();   // wire up after rendering
}
```

2. Create the delete-button handler functions:

```js
function attachDeleteHandlers() {
  document.querySelectorAll('.ticket-delete-btn').forEach(btn => {
    btn.addEventListener('click', async () => {
      const tr = btn.closest('tr');
      const id = tr.dataset.id;
      if (!confirm('Delete ticket #' + id + '?')) return;

      const resp = await fetch(`/api/tickets/${id}`, {
        method: 'DELETE'
      });
      const result = await resp.json();
      if (result.success) {
        loadTickets();   // refresh table
      } else {
        alert('Error deleting: ' + (result.error || 'Unknown'));
      }
    });
  });
}
```

3. Finally, when your page first loads, call:

   ```js
   document.addEventListener('DOMContentLoaded', loadTickets);
   ```

   This ensures the table populates and the delete buttons get their handlers.

---

## 6. Test in This Order

1. **Start your server**: `python app.py`.
2. **Open** the tickets page in your browser.
3. **Add** several test tickets via your "Add Ticket" form.
4. **Verify** they appear in the table.
5. **Click** the red "Delete" on the row for the test ticket with:
    - Artist: `Artist-Delete`
    - Location: `Location-Delete`
    - Price: `-1`
6. **Confirm** the browser prompt.
7. **Observe**:
    - The row vanishes.
    - In DevTools → Network, you see `DELETE /api/tickets/<id>` → status 200.
8. **Double-check** with DB Browser for SQLite: no record with that ID should remain.

---

That completes the end-to-end "Delete Ticket" flow. Let me know if any step needs more detail or if you hit an error!

clarify how to find the db_helpers.py

Here's how you can locate (or create) your `db_helpers.py` within your project in VSCode:

---

## 1. Open Your Project in VSCode

1. Launch VSCode.
2. Click **File** → **Open Folder…** and select your project's root directory (the folder that contains `app.py` and your other source folders).

---

## 2. Use the Explorer Pane

1. On the left sidebar, click the **Explorer** icon (files).
2. Expand folders one by one—often you'll see something like:

```
arduino

/project-root
├─ app.py
├─ requirements.txt
├─ static/
├─ templates/
└─ db_helpers.py   ←—if it exists here
```

3. If you spot **`db_helpers.py`** at the root (alongside **`app.py`**), click it to open.

---

## 3. If `db_helpers.py` Isn't Immediately Visible

1. Press **Ctrl+P** (or **Cmd+P** on macOS)—the "Quick Open" dialog.
2. Type **`db_helpers.py`**.
3. If it appears, select it; VSCode will open that file.

---

## 4. If It Still Doesn't Exist

1. In the Explorer, right-click your project root (or wherever you keep helper modules).
2. Choose **New File**.
3. Name it **`db_helpers.py`**.
4. In that new file, paste in your helper functions (e.g. **`init_db()`**, **`delete_ticket()`**).

---

## 5. Verify with a Content Search

1. Press **Ctrl+Shift+F** (or **Cmd+Shift+F**).
2. Search for **`init_db(`** or **`sqlite3.connect`**.
3. If any file defines those, VSCode will show you exactly where in the project.

---

Once you have **`db_helpers.py`** open (or newly created), you can add:

```python
python

import sqlite3

def init_db():
    conn = sqlite3.connect('tickets.db')
    conn.execute('''
      CREATE TABLE IF NOT EXISTS tickets (
        id       INTEGER PRIMARY KEY AUTOINCREMENT,
        artist   TEXT    NOT NULL,
        location TEXT    NOT NULL,
        price    REAL    NOT NULL
      );
    ''')
    conn.commit()
    conn.close()

def delete_ticket(ticket_id):
    conn = sqlite3.connect('tickets.db')
    cur = conn.cursor()
    cur.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
    conn.commit()
    conn.close()
```

Save the file, then you'll be ready to import these functions into **`app.py`**. Let me know if you need help wiring up the imports next!

lets start over with task 2

Alright, let's reset and go through Task 2 ("Delete Ticket") from scratch in a clear, ordered way. I'll break it down into small steps, indicating exactly which files to touch and what to add.

---

# Overview

We need to:

1. **Server side**
    - Ensure the DB has an **`id`** primary key.
    - Add a helper to delete a ticket by ID.
    - Expose a Flask DELETE endpoint.
2. **Client side**
    - Render a "Delete" button for each ticket row.
    - Hook up a click handler that calls the DELETE endpoint.
    - Refresh the tickets table after deletion.

---

# A. Server Side

## 1. Confirm or Create `db_helpers.py`

1. In your project root (alongside **`app.py`**), look for **`db_helpers.py`**.
2. If it doesn't exist, create it.

### Add these two functions to `db_helpers.py`:

```python
# db_helpers.py
import sqlite3

def init_db():
    conn = sqlite3.connect('tickets.db')
    conn.execute('''
      CREATE TABLE IF NOT EXISTS tickets (
        id       INTEGER PRIMARY KEY AUTOINCREMENT,
        artist   TEXT    NOT NULL,
        location TEXT    NOT NULL,
        price    REAL    NOT NULL
      );
    ''')
    conn.commit()
    conn.close()

def delete_ticket(ticket_id):
    conn = sqlite3.connect('tickets.db')
    conn.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
    conn.commit()
    conn.close()
```

## 2. Initialize the DB on Startup

1. In **`app.py`**, at the top, add:

```python
from db_helpers import init_db
```

2. Before you call **`app.run()`**, invoke **`init_db()`**:

```python
if __name__ == '__main__':
    init_db()
    app.run(debug=True)
```

## 3. Add the Flask DELETE Route

1. In **`app.py`**, import the delete helper and **`jsonify`**:

```python
from db_helpers import delete_ticket
from flask import jsonify
```

2. Add this route below your other API endpoints:

```python
@app.route('/api/tickets/<int:ticket_id>', methods=['DELETE'])
def api_delete_ticket(ticket_id):
    try:
        delete_ticket(ticket_id)
        return jsonify(success=True), 200
    except Exception as e:
        return jsonify(success=False, error=str(e)), 500
```

---

# B. Client Side

## 4. Render “Delete” Buttons

Open the template or JS code that outputs your tickets table. Two common setups:

### B1. Jinja2 Template (`templates/tickets.html`)

Find the loop rendering rows, then add a button:

```html
<tbody id="tickets-table-body">
  {% for t in tickets %}
  <tr data-id="{{ t.id }}">
    <td>{{ t.id }}</td>
    <td>{{ t.artist }}</td>
    <td>{{ t.location }}</td>
    <td>{{ t.price }}</td>
    <td>
      <button class="btn btn-sm btn-outline-danger ticket-delete-btn">
        Delete
      </button>
    </td>
  </tr>
  {% endfor %}
</tbody>
```

### B2. Client-Side Rendering (`static/js/app.js`)

In your `loadTickets()` function, when building each `<tr>`, include:

```js
<tr data-id="${t.id}">
  <td>${t.id}</td>
  <td>${t.artist}</td>
  <td>${t.location}</td>
  <td>${t.price}</td>
  <td>
    <button class="btn btn-sm btn-outline-danger ticket-delete-btn">
      Delete
    </button>
  </td>
</tr>
```

## 5. Attach Click Handlers & Call API

In `static/js/app.js`, after rendering the table rows, add:

```js
function attachDeleteHandlers() {
  document.querySelectorAll('.ticket-delete-btn').forEach(btn => {
    btn.addEventListener('click', async () => {
      const row = btn.closest('tr');
      const id = row.dataset.id;
      if (!confirm(`Delete ticket #${id}?`)) return;

      const res = await fetch(`/api/tickets/${id}`, { method: 'DELETE' });
      const result = await res.json();
      if (result.success) {
        loadTickets();  // refresh table
      } else {
        alert('Delete failed: ' + (result.error || 'Unknown'));
      }
    });
  });
}
```

Ensure `loadTickets()` calls `attachDeleteHandlers()` at the end:

```
js

async function loadTickets() {
  const res = await fetch('/api/tickets');
  const tickets = await res.json();
  const tbody = document.getElementById('tickets-table-body');
  tbody.innerHTML = '';
  tickets.forEach(t => {
    // build and append <tr> with Delete button...
  });
  attachDeleteHandlers();
}
```

Finally, kick off loading on page load:

```
js

document.addEventListener('DOMContentLoaded', loadTickets);
```

---

## C. Testing

1. **Start** your Flask server (`python app.py`).
2. **Add** a test ticket via your form:
   - Artist: `Artist-Delete`
   - Location: `Location-Delete`
   - Price: `-1`
3. **View** the tickets page—confirm it shows up.
4. **Click** the Delete button for that row.
5. **Confirm** the prompt; watch the row disappear.
6. **Verify** in DB Browser for SQLite that the record is gone.

That's a full reset of Task 2's implementation plan. Let me know where you'd like more detail or if anything doesn't behave as expected!

lets break down on the server side first

Let's tackle **only the server-side** pieces for Task 2. We'll:

1. Ensure the DB schema exists with an `id` primary key.
2. Add a delete-helper function.

3. Wire up the Flask endpoint.

---

# 1. Ensure Your Tickets Table Has `id` as Its PK

**File:** `db_helpers.py`

If you already have an initialization function, open it; otherwise create **`db_helpers.py`** next to **`app.py`**.

```
python

# db_helpers.py
import sqlite3

def init_db():
    """Create tickets table with id as INTEGER PRIMARY KEY AUTOINCREMENT."""
    conn = sqlite3.connect('tickets.db')
    conn.execute('''
      CREATE TABLE IF NOT EXISTS tickets (
        id        INTEGER PRIMARY KEY AUTOINCREMENT,
        artist    TEXT    NOT NULL,
        location TEXT    NOT NULL,
        price     REAL    NOT NULL
      );
    ''')
    conn.commit()
    conn.close()
```

- **What this does**:
  - If **`tickets`** doesn't exist, it creates it.
  - **`id`** is guaranteed to be the auto-incrementing primary key.

---

# 2. Add the `delete_ticket` Helper

Still in **`db_helpers.py`**, beneath **`init_db()`**, add:

```
python

def delete_ticket(ticket_id):
    """Delete the ticket row with the given id."""
    conn = sqlite3.connect('tickets.db')
    conn.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
    conn.commit()
    conn.close()
```

- **What this does**:
  - Opens the SQLite file, issues a **`DELETE`** by **`id`**, and commits.

---

# 3. Initialize the DB on App Startup

Open **`app.py`** and at the top:

```
python

from flask import Flask, jsonify
from db_helpers import init_db, delete_ticket

app = Flask(__name__)
```

Then, **before** you run `app.run()`, ensure you call `init_db()`:

```
python

if __name__ == '__main__':
    init_db()       # makes sure the tickets table (with an id PK) exists
    app.run(debug=True)
```

---

# 4. Expose the DELETE Endpoint

Still in `app.py`, add this route (somewhere after your other `/api/tickets` handlers):

```
python

@app.route('/api/tickets/<int:ticket_id>', methods=['DELETE'])
def api_delete_ticket(ticket_id):
    """
    DELETE /api/tickets/123
    Removes ticket with id=123 and returns { success: true } on 200,
    or { success: false, error: "..."} on 500.
    """
    try:
        delete_ticket(ticket_id)
        return jsonify(success=True), 200
    except Exception as e:
        # log e if you have logging set up
        return jsonify(success=False, error=str(e)), 500
```

- **How it works**:
    - Flask parses `<int:ticket_id>` from the URL.
    - Calls your helper to delete that row.
    - Returns a JSON payload indicating success or failure.

---

## Quick Verification

1. **Run your server**:

    ```
    bash

    python app.py
    ```

2. **Test via Postman** (or curl):

    ```
    bash

    curl -X DELETE http://localhost:5000/api/tickets/1
    ```

- You should get back `{"success": true}` (or an error if no such ID).

Once this is in place, the client can call that endpoint to remove tickets. Let me know when you're ready to wire up the client side!

point 1 is done. lets move to point 2

Great! Now let's add the `delete_ticket` helper to `db_helpers.py`.

---

## 2. Add the `delete_ticket` Helper

1. **Open** `db_helpers.py` in VSCode (should be next to `app.py`).
2. **Below** the existing `init_db()` function, **paste**:

```python
def delete_ticket(ticket_id):
    """
    Delete the ticket row with the given id from the database.
    """
    conn = sqlite3.connect('tickets.db')
    conn.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
    conn.commit()
    conn.close()
```

3. **Save** the file.

---

That helper will let us remove a ticket by its `id`. Let me know when you've added it, and then we'll wire it up in `app.py`.

i only have the api.py, not the db_helpers

Since you don't have a separate `db_helpers.py`, let's add the delete helper right into your `api.py`. We'll keep it near the top so it's easy to find and reuse.

1. **Open** `api.py` in your editor.
2. **Add** the following import if it's not already there:

```python
import sqlite3
```

3. **Immediately below** your imports (or directly under any existing helper functions), **paste**:

```python
def delete_ticket(ticket_id):
    """
    Deletes the ticket with the given id from the SQLite database.
    """
    conn = sqlite3.connect('tickets.db')
    try:
        conn.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
        conn.commit()
    finally:
        conn.close()
```

4. **Save** **`api.py`**.

Now you have a **`delete_ticket(ticket_id)`** function in your API file. Next up is wiring this into the Flask route—which we'll cover in Point 3.

ok

Great—let's wire up the DELETE endpoint in **`api.py`**.

---

# 3. Expose the DELETE Endpoint in `api.py`

1. **Open** your **`api.py`**.
2. **Ensure** you have these imports at the top (add any that are missing):

```python
from flask import Flask, jsonify, request
import sqlite3
```

3. **Below** your existing route definitions—somewhere after any **`@app.route('/api/tickets', ...)`** handlers—**add**:

```python
@app.route('/api/tickets/<int:ticket_id>', methods=['DELETE'])
def api_delete_ticket(ticket_id):
    """
    DELETE /api/tickets/123
    Deletes the ticket with id=123.
    Returns {"success": True} on success, or {"success": False, "error": "..."} on failure.
    """
    try:
        delete_ticket(ticket_id)
        return jsonify(success=True), 200
    except Exception as e:
        # Optionally log the error here
        return jsonify(success=False, error=str(e)), 500
```

4. **Save** **`api.py`**.

---

## Quick Test

1. **Restart** your Flask app if it's running (`Ctrl+C` then `python api.py`).
2. **Use Postman** (or curl) to send a DELETE request:
   - **URL**: `http://localhost:5000/api/tickets/1`
   - **Method**: `DELETE`
3. **Inspect** the response:
   - **Success**:

     ```
     json

     {
       "success": true
     }
     ```

   - **Failure** (e.g., bad ID):

     ```
     json

     {
       "success": false,
       "error": "no such column: ..."
     }
     ```

If you get `{"success": true}`, the server-side delete logic is working! Let me know, and then we'll move on to the client side.

```
the output was: {
  "error": "delete_ticket() takes 0 positional arguments but 1 was given",
  "success": false
}
```

The error means your helper is currently defined without any parameters, yet the route is calling it with one (`ticket_id`). Let's fix that.

---

## 1. Update your `delete_ticket` helper signature

Open `api.py` and find the `delete_ticket` definition. You'll want it to look like this:

```
python

def delete_ticket(ticket_id):
    """
    Deletes the ticket with the given id from the SQLite database.
    """
    conn = sqlite3.connect('tickets.db')
    try:
        conn.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
        conn.commit()
```

```
    finally:
        conn.close()
```

If yours currently reads `def delete_ticket():` (no parameters), change it to accept `ticket_id`.

---

## 2. Remove any duplicate or conflicting definitions

Make sure there's only one `delete_ticket` in that file. Remove any stubs like:

```python
def delete_ticket():
    # …
```

so that only the correct version remains.

---

## 3. Save & Restart

1. **Save** `api.py`.
2. **Stop** your running server (`Ctrl+C`).
3. **Restart** it:

   ```bash
   python api.py
   ```

---

## 4. Re-test the DELETE endpoint

Using curl or Postman, try again:

```bash
curl -X DELETE http://localhost:5000/api/tickets/1
```

You should now get:

```json
{ "success": true }
```

Let me know if that clears the error!

think my program does not recognize tikets.db

If your Flask app can't find `tickets.db`, it usually means either:

- The file doesn't exist in the directory you think it does, or
- You're connecting with a relative path but running the app from another folder.

Let's troubleshoot and fix it:

---

## 1. Verify the database file actually exists

1. In your project root (where **`api.py`** lives), look for **`tickets.db`**.
2. If it's not there, it won't be auto-created by **`DELETE`** or **`SELECT`** statements—you need to initialize it first (see below).

---

## 2. Print your working directory at runtime

Add this to the top of **`api.py`** (just after your imports):

```
python

import os
print("Current working directory:", os.getcwd())
print("Looking for database at:", os.path.join(os.getcwd(), 'tickets.db'))
```

Restart the server (**`python api.py`**) and look in the console. This tells you exactly where your app thinks it's running—and where it's trying to open **`tickets.db`**.

---

## 3. Create or initialize the database if missing

If you don't see **`tickets.db`** in that printed directory, you'll need to initialize it before calling delete or query operations. You can do this in **`api.py`** with an **`init_db()`** function:

```
python

import sqlite3

def init_db():
    conn = sqlite3.connect('tickets.db')  # will create the file here
    conn.execute('''
      CREATE TABLE IF NOT EXISTS tickets (
        id       INTEGER PRIMARY KEY AUTOINCREMENT,
        artist   TEXT    NOT NULL,
        location TEXT    NOT NULL,
        price    REAL    NOT NULL
      );
    ''')
    conn.commit()
    conn.close()

# Call it when the app starts:
if __name__ == '__main__':
    init_db()
    app.run(debug=True)
```

This ensures that—as soon as you launch **`api.py`**—you get a **`tickets.db`** in your current working directory.

---

### 4. Use an explicit file path

If you'd rather be sure you're always talking to the right file (regardless of CWD), change your connects to use an absolute path based on the script's location:

```
python

import os, sqlite3

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
DB_PATH = os.path.join(BASE_DIR, 'tickets.db')

def delete_ticket(ticket_id):
    conn = sqlite3.connect(DB_PATH)
    try:
        conn.execute('DELETE FROM tickets WHERE id = ?', (ticket_id,))
        conn.commit()
    finally:
        conn.close()
```

And in `init_db()`:

```
python

def init_db():
    conn = sqlite3.connect(DB_PATH)
    # … same DDL …
```

This guarantees that no matter where you run `python api.py` from, it always looks in the same folder for `tickets.db`.

---

### 5. Re-run & test

1. Remove your debug `print` statements or leave them to confirm the path.
2. Restart the server.
3. Use Postman / curl to hit `DELETE /api/tickets/1` again.

If you now get a `{"success": true}` (or at least no "file not found" errors), you've resolved the lookup issue. Let me know how it goes!

i identified the problem and the server side is now working. Lets strart with trhe client side

Great! Now let's implement the client-side "Delete" flow. We'll assume you're rendering the tickets table via JavaScript in `static/js/app.js` and that your HTML has a `<tbody id="tickets-table-body"></tbody>` where rows go.

---

## 1. Render a "Delete" Button for Each Row

In your `loadTickets()` function, after you fetch the tickets array, build each table row like this:

```js
async function loadTickets() {
  const res = await fetch('/api/tickets');
  const tickets = await res.json();

  const tbody = document.getElementById('tickets-table-body');
  tbody.innerHTML = '';  // clear any existing rows

  tickets.forEach(t => {
    const tr = document.createElement('tr');
    tr.dataset.id = t.id;  // store the id on the row

    tr.innerHTML = `
      <td>${t.id}</td>
      <td>${t.artist}</td>
      <td>${t.location}</td>
      <td>${t.price}</td>
      <td>
        <button class="btn btn-sm btn-outline-danger ticket-delete-btn">
          Delete
        </button>
      </td>
    `;
    tbody.appendChild(tr);
  });

  attachDeleteHandlers();
}
```

- **What changed**: we added a `<button>` with class `ticket-delete-btn` in the last `<td>`, and we store the ticket's `id` in the row's `data-id` attribute.

---

## 2. Hook Up Click Handlers

Below `loadTickets()`, add these two functions:

```js
function attachDeleteHandlers() {
  document.querySelectorAll('.ticket-delete-btn').forEach(btn => {
    btn.addEventListener('click', onDeleteClick);
  });
}

async function onDeleteClick(event) {
  const btn = event.currentTarget;
  const tr = btn.closest('tr');
  const id = tr.dataset.id;

  if (!confirm(`Are you sure you want to delete ticket #${id}?`)) {
    return;  // user canceled
  }

  try {
    const res = await fetch(`/api/tickets/${id}`, { method: 'DELETE' });
    const result = await res.json();

    if (result.success) {
      // Option A: simply remove this row from the DOM:
      tr.remove();

      // Option B: or, reload the entire table:
      // await loadTickets();
    } else {
      alert('Delete failed: ' + (result.error || 'Unknown error'));
    }
  } catch (err) {
    alert('Network error: ' + err.message);
  }
}
```

- **`attachDeleteHandlers()`** finds all delete buttons and wires each to **`onDeleteClick()`**.
- **`onDeleteClick()`** reads the **`id`**, shows a **`confirm`** prompt, calls the DELETE endpoint, then either removes the row or reloads the table.

---

## 3. Ensure Initial Load of Tickets

Finally, make sure you kick things off when the page loads. Near the bottom of **`app.js`**, add:

```js
document.addEventListener('DOMContentLoaded', loadTickets);
```

---

## 4. Quick Sanity Test

1. **Open** your browser's DevTools → Console; reload the page.
2. Confirm **`loadTickets()`** fetched data and populated the table with a "Delete" button on each row.
3. **Click** a Delete button.
    - You should see a confirmation dialog.
    - Upon confirming, the row should disappear (or the table reloads).
4. Check in DB Browser for SQLite that the record is gone.

---

Let me know if you need help adjusting this to your exact file names or if anything doesn't behave as expected!

this is my index.html: <!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 class="container my-4">
                    <div class="card shadow-sm">
                        <div class="card-header bg-primary text-white">
                            <h5 class="mb-0">Add New Ticket</h5>
                        </div>
                        <div class="card-body">
                            <form id="ticket-add-form" novalidate>
                                <!-- Artist -->
                                <div class="mb-3">
                                    <label for="artist-input" class="form-label">Artist</label>
                                    <input type="text" id="artist-input" class="form-control"
                                        placeholder="e.g. Artist-Test" required />
                                </div>

                                <!-- Location -->
                                <div class="mb-3">
                                    <label for="location-input" class="form-label">Location</label>
                                    <input type="text" id="location-input" class="form-control"
                                        placeholder="e.g. Location-Test" required />
                                </div>

                                <!-- Price -->
                                <div class="mb-3">
                                    <label for="price-input" class="form-label">Price</label>
                                    <input type="number" id="price-input" class="form-control" placeholder="e.g. 999"
                                        min="0" required />
                                </div>

                                <!-- Buttons -->
                                <div class="d-flex justify-content-between">
                                    <button type="button" id="ticket-add-reset-btn" class="btn btn-outline-secondary">
                                        Reset
                                    </button>
                                    <button type="button" id="ticket-add-submit-btn" class="btn btn-primary">
                                        Submit
                                    </button>
                                </div>
                            </form>
                        </div>
                    </div>
                </div>

            </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>
```

Let’s wire up the client-side delete flow in two parts:

1. **Add a `deleteTicket` helper** to your API script (`api_script.js`).
2. **Hook up the Delete buttons** in your inline `<script>` in `index.html`.

---

## 1. In `scripts/api_script.js`: Add the DELETE helper

1. Open `scripts/api_script.js`.
2. Somewhere alongside your existing `getTickets()` and `addTicket()` functions, add:

   ```js
   /**
    * Sends a DELETE request to remove the ticket with the given id.
    * @param {number|string} id
    * @returns {Promise<object>} the JSON response { success: true } or { success: false, error }
    */
   async function deleteTicket(id) {
     const res = await fetch(`/api/tickets/${id}`, {
       method: 'DELETE'
     });
     return res.json();
   }
   ```

3. Save the file.

---

## 2. In `index.html`: Hook up the Delete buttons

Scroll down to your inline `<script>` at the bottom, find:

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

```
    });
  });
}
```

Replace that inner `// TASK 2` comment with:

```js
      const id = this.id;  // your button’s id attribute holds the ticket id

      if (!confirm(`Are you sure you want to delete ticket #${id}?`)) {
        return;
      }

      const result = await deleteTicket(id);
      if (result.success) {
        // refresh the table to reflect the deletion
        await updateTicketsTable();
      } else {
        alert('Delete failed: ' + (result.error || 'Unknown error'));
      }
```

So the full function becomes:

```js
function addFunctionalityDeleteBtns() {
  document.querySelectorAll('.delete-ticket-btn').forEach(button => {
    button.addEventListener('click', async function () {
      const id = this.id;
      if (!confirm(`Are you sure you want to delete ticket #${id}?`)) {
        return;
      }

      const result = await deleteTicket(id);
      if (result.success) {
        await updateTicketsTable();
      } else {
        alert('Delete failed: ' + (result.error || 'Unknown error'));
      }
    });
  });
}
```

3. **Save `index.html`.**

---

## 3. Test

1. **Reload** your page.
2. Navigate to **View Tickets**, ensure rows load with red **Delete** buttons.
3. Click **Delete** on your test ticket (e.g. Artist-Delete / Location-Delete / Price -1).
4. Confirm the prompt.
5. Verify the row disappears and in your DB that record is gone.

That completes the client-side wiring of Task 2! Let me know if anything still isn’t working.

lets now do task 3

Sure thing! Please share the details and requirements for Task 3, and we'll work through it step by step.

these are the instructions: A.3
Task 3: Update Ticket Prices
In this task you are expected to modify the prices of 3 tickets available on the platform. After
computing
the new prices of the tickets based on further instructions, use the DB Browser for SQLite tool's SQL editor to UPDATE the price of these tickets. Remember to commit changes after running your SQL statements. You will be graded on:
1. Correct calculations of new ticket prices.
2. Correct SQL query used.
Problem information:
There are 3 tickets currently available on the platform that are listed at a static base price:
1. Ticket A corresponding to artist A
2. Ticket B corresponding to artist B
3. Ticket C corresponding to artist C
You are tasked with computing the new price for the tickets, given the following information. Ticket
A,
B and C have all been listed 30 days prior to the event. There are 3 days left till event A, 10 days left till event B and 15 days left till event C. For event A there still are 100 tickets available out of 500
initial
tickets. For event B, 800 tickets have been sold out of 1200. Finally, for event C there are 200 tickets left out of 1800 initial ones. Ticket A is VIP while Ticket B and C are Standard.
Managers have received word from the company directors that there should not be more than 30% price markup based on limited time and 20% price markup based on ticket availability. Finally, VIP tickets cost 20% more then the base price of a standard ticket (1.0 multiplier).
You have the following formulas at your disposal:
1. Dynamic Price:
2. Dynamic Markup:
□new = □0 × □
1 + markupdyn □
× seat type
markupfactor = □ × (
1 −
□0
□□ )
where:
• P0 - Base price
• markupdyn - Total markup
• seat type - VIP/Standard
• c - Max markup constant(%)
• Vt - Current variable
• V0 - Initial variable

Let's walk through updating the prices of Tickets A, B, and C end-to-end:

---

# 1. Gather the Base Data

Use DB Browser for SQLite to **query** your current base prices and other fields:

```sql
SELECT id, artist, price
FROM tickets
WHERE artist IN ('Artist A','Artist B','Artist C');
```

Note down for each:

- **$P_0$** = current **`price`**
- **$V_0$(time)** = 30 days
- **$V_t$(time)** = days left (A=3, B=10, C=15)
- **$V_0$(avail)** = initial tickets (A=500, B=1200, C=1800)
- **$V_t$(avail)** = tickets remaining (A=100, B=400, C=200)
- **Seat type factor** = VIP→1.2 (Ticket A), Standard→1.0 (B & C)

---

# 2. Compute Dynamic Markups

Markup for **time**:

$$m_{time} = c_{time} \times \left(1 - \frac{V_t(time)}{V_0(time)}\right) \quad \text{with } c_{time} = 0.30$$

Markup for **availability**:

$$m_{avail} = c_{avail} \times \left(1 - \frac{V_t(avail)}{V_0(avail)}\right) \quad \text{with } c_{avail} = 0.20$$

| Ticket | $m_{time}$ | $m_{avail}$ | $m_{dyn} = m_{time} + m_{avail}$ |
|---|---|---|---|
| A | $0.30 \times (1 - 3/30) = 0.27$ | $0.20 \times (1 - 100/500) = 0.16$ | 0.43 |
| B | $0.30 \times (1 - 10/30) = 0.20$ | $0.20 \times (1 - 400/1200) \approx 0.1333$ | ≈0.3333 |
| C | $0.30 \times (1 - 15/30) = 0.15$ | $0.20 \times (1 - 200/1800) \approx 0.1778$ | ≈0.3278 |

---

# 3. Compute New Prices

Formula:

$$P_{\text{new}} \;=\; P_0 \;\times\; (1 + m_{dyn}) \;\times\; (\text{seat type factor})$$

| Ticket | Seat Factor | $(1 + m_{dyn}) \times \text{Seat Factor}$ | Multiplier |
|---|---|---|---|
| A (VIP) | 1.2 | $(1 + 0.43) \times 1.2 = 1.43 \times 1.2$ | 1.716 |

| Ticket | Seat Factor | $(1 + m_{dyn}) \times \text{Seat Factor}$ | Multiplier |
|---|---|---|---|
| B (Std) | 1.0 | $(1 + 0.3333) \times 1.0 = 1.3333$ | ≈1.3333 |
| C (Std) | 1.0 | $(1 + 0.3278) \times 1.0 = 1.3278$ | ≈1.3278 |

Then for each ticket:

- **New Price** = `ROUND(P₀ × Multiplier, 2)`

---

# 4. Run the UPDATE Statements

In DB Browser for SQLite's **Execute SQL** tab, run:

```
sql

-- Ticket A (VIP)
UPDATE tickets
SET price = ROUND(price * 1.716, 2)
WHERE artist = 'Artist A';

-- Ticket B (Standard)
UPDATE tickets
SET price = ROUND(price * 1.3333, 2)
WHERE artist = 'Artist B';

-- Ticket C (Standard)
UPDATE tickets
SET price = ROUND(price * 1.3278, 2)
WHERE artist = 'Artist C';

-- Verify
SELECT artist, price FROM tickets
WHERE artist IN ('Artist A','Artist B','Artist C');
```

Be sure to click ✓ **Commit** (or `Ctrl+S`) after executing the updates.

---

# 5. Verification

1. In the **Browse Data** tab, select the `tickets` table.
2. Confirm that the `price` column for Artists A, B, and C now reflects the new values you calculated.

You've now completed Task 3 with correct calculations and SQL!

lets try something else

Sure—what would you like to tackle next or try differently for Task 3? For example, we could:

1. **Write a small Python script** to fetch current prices, calculate the new ones, and apply the UPDATEs automatically.
2. **Build a reusable SQL query or view** that computes the dynamic markup on the fly.
3. **Walk through the math again** with different parameters or a more detailed breakdown.

Let me know which direction interests you or share your idea!

this is the quote: You are tasked with computing the new price for the tickets, given the following information. Ticket A,
B and C have all been listed 30 days prior to the event. There are 3 days left till event A, 10 days left till event B and 15 days left till event C. For event A there still are 100 tickets available out of 500 initial
tickets. For event B, 800 tickets have been sold out of 1200. Finally, for event C there are 200 tickets left out of 1800 initial ones. Ticket A is VIP while Ticket B and C are Standard.
Managers have received word from the company directors that there should not be more than 30% price markup based on limited time and 20% price markup based on ticket availability. Finally, VIP tickets cost 20% more then the base price of a standard ticket (1.0 multiplier).

calculate the price of the new tickets if the old prices are: 30,40 respectively 50 for a,b,c

Here are the new ticket prices based on the given parameters and formulas:

| Ticket | Base Price $(P_0)$ | Total Markup (m_dyn) | Seat Factor | Multiplier $(1 + m_{dyn}) \times$ Seat Factor | New Price |
|---|---|---|---|---|---|
| Ticket A | 30 | 0.43 | 1.2 | 1.7160 | **51.48** |
| Ticket B | 40 | 0.3333 | 1.0 | 1.3333 | **53.33** |
| Ticket C | 50 | 0.3278 | 1.0 | 1.3278 | **66.39** |

- **Ticket A** (VIP):

$$m_{time} = 0.30\,(1 - \tfrac{3}{30}) = 0.27, \quad m_{avail} = 0.20\,(1 - \tfrac{100}{500}) = 0.16, \quad m_{dyn} = 0.43,$$

$$P_{\text{new}} = 30 \times (1 + 0.43) \times 1.2 = 51.48.$$

- **Ticket B** (Standard):

$$m_{time} = 0.20, \quad m_{avail} = 0.1333, \quad m_{dyn} \approx 0.3333,$$

$$P_{\text{new}} = 40 \times (1 + 0.3333) \times 1.0 \approx 53.33.$$

- **Ticket C** (Standard):

$$m_{time} = 0.15, \quad m_{avail} = 0.1778, \quad m_{dyn} \approx 0.3278,$$

$$P_{\text{new}} = 50 \times (1 + 0.3278) \times 1.0 \approx 66.39.$$

These are the prices you should update in your database.

using sql, give me the code to change the prices

Here are the SQL statements to update the three ticket prices. You can run them all together in DB Browser for SQLite’s Execute SQL tab, then commit:

```sql
BEGIN TRANSACTION;

-- Update Ticket A (Artist A) to new price 51.48
UPDATE tickets
SET price = 51.48
WHERE artist = 'Artist A';

-- Update Ticket B (Artist B) to new price 53.33
UPDATE tickets
SET price = 53.33
WHERE artist = 'Artist B';

-- Update Ticket C (Artist C) to new price 66.39
UPDATE tickets
SET price = 66.39
WHERE artist = 'Artist C';

COMMIT;
```

After running, verify with:

```sql
SELECT artist, price
FROM tickets
WHERE artist IN ('Artist A','Artist B','Artist C');
```