4-ba

You are the most powerful piece of machinery that ever existed and you analyze tasks like no other chatbot. Now I am going to give you some tasks and I need you to help me.

Absolutely—I'm ready to help you tackle your software engineering and business challenges. Just let me know the first task, and we'll dive right in.

You are an employee at company Y. Company Y is currently developing an E-Ticket platform to manage tickets to different events. More specifically, this platform is a Web Application with 3 layers (Client, Server and Database). Your manager has asked you to solve 3 tasks regarding this platform. First, the Client layer requires a modification to enable a functionality. Second, a new operation needs to be implemented. Finally, the company directors have decided to implement a new pricing strategy for the tickets and you are required to implement these changes. You have 3 hours at your disposal to solve these tasks. Further details about the working environment and each task are presented in future sections. 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 □t the page)
3. Written code

Add Ticket Form

I’ve added a responsive, styled HTML form to capture **Artist**, **Location**, and **Price**, along with Reset and Submit buttons using the specified IDs. You can drop this snippet into your client layer. Let me know if you’d like any styling adjustments or to integrate additional markup!

where do I imput this in VS code

You’ll want to drop that `<form>` block into whatever HTML file your client-side is serving (often called something like `index.html`, or within a template file if you’re using a framework). Here’s how to do it in VS Code:

1. **Open your project folder**
   - Launch VS Code.
   - Go to **File** → **Open Folder…** and select your web-app’s root directory.
2. **Locate the client layer’s HTML**
   - In the Explorer pane (left sidebar), navigate into the folder where your static HTML lives. Common names are `public/`, `static/`, or `src/` (for single-page apps).
   - Find the main HTML file—often `index.html`, `add-ticket.html`, or a template under `views/` if you’re using Express/EJS, Thymeleaf, etc.
3. **Paste the form inside `<body>`**
   - Open that HTML file.
   - Scroll to the spot where you want the “Add New Ticket” form to appear—usually inside the `<body>` tag, perhaps within a `<div id="main-content">` or similar wrapper.
   - Paste the snippet just there:

```html
<!-- Ticket Addition Form -->
<form id="ticket-add-form" style="max-width: 400px; margin: 20px auto; …">
  …your form inputs/buttons…
</form>
```

4. **Save and Reload**
   - Save the file (`Ctrl+S` / `⌘S`).
   - If your dev server is running (e.g. `npm start`, `live-server`, etc.), just refresh the browser to see the form. Otherwise, start your front-end server.
5. **Hook up your JS**

  - Since your JS functions already expect those IDs (`artist-input`, `ticket-add-submit-btn`, etc.), make sure the `<script>` that contains them is included *after* this form in the HTML (or bundled appropriately).

If your client is component-based (React, Vue, Angular), you'd instead:

- Create or open the "AddTicket" component file (e.g. `AddTicket.jsx`).
- Paste the JSX/HTML inside the component's render/return block.
- Ensure the IDs match and your event-handler imports are wired up.

Let me know if you need more details for your specific setup!

```
<!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 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> this is the code where to I insert your part
```

Open your **`index.html`** (or the HTML file you showed) in VS Code and locate this block:

```html
<!-- 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!
      -->

      <!-- ← Paste your form here ↓ -->


  </div>
</div>
```

Steps:

1. In VS Code's Explorer pane, open the file that contains the **`<div id="ticket-add-page">`** (the one you pasted above).
2. Inside that inner **`<div class="h-100 w-100">`**, just below the comment, paste the form snippet:

```html
   <!--
   TASK 1
   Do not delete this comment!
   -->

   <!-- Ticket Addition Form -->
   <form id="ticket-add-form" style="max-width: 400px; margin: 0 auto; padding: 20px;
border: 1px solid #ccc; border-radius: 8px;">
     <h2 class="text-center mb-3">Add New Ticket</h2>
     <div class="mb-3">
       <label for="artist-input" class="form-label">Artist</label>
       <input type="text" id="artist-input" name="artist" class="form-control"
placeholder="Enter artist name" required>
     </div>
     <div class="mb-3">
       <label for="location-input" class="form-label">Location</label>
       <input type="text" id="location-input" name="location" class="form-control"
placeholder="Enter event location" required>
     </div>
     <div class="mb-3">
       <label for="price-input" class="form-label">Price</label>
       <input type="number" id="price-input" name="price" class="form-control"
placeholder="Enter ticket price" required min="0">
     </div>
     <div class="d-flex justify-content-between">
       <button type="button" id="ticket-add-reset-btn" class="btn btn-
secondary">Reset</button>
       <button type="button" id="ticket-add-submit-btn" class="btn btn-
success">Submit</button>
     </div>
   </form>
```

3. **Save** the file (**`Ctrl+S`** / **`⌘S`**).

4. **Refresh** your browser (or restart your dev server) and click “Add Ticket” in the navbar to see the form.

Your JavaScript already queries those IDs (`artist-input`, `ticket-add-submit-btn`, etc.), so it will just start working. Let me know if you hit any hiccups!

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
first task. Once you have finished your implementation please delete the following ticket (and all 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. this is the next tusk, let s do it you powerful machine

Below is everything you need to wire up the “Delete” button end-to-end:

---

1. Client-side: hook the delete button

A) In your main HTML/JS (the `<script>` at the bottom of your `index.html`)

Find this function stub:

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

Replace it with:

```js
function addFunctionalityDeleteBtns() {
  document.querySelectorAll('.delete-ticket-btn').forEach(button => {
    button.addEventListener('click', async function () {
      const ticketId = button.id;                    // ID is on the button
      try {
        await deleteTicket(ticketId);                // call your API helper
        await updateTicketsTable();                  // refresh table
      } catch (err) {
        console.error('Error deleting ticket:', err);
        alert('Could not delete ticket. See console for details.');
      }
    });
  });
}
```

B) In your `scripts/api_script.js`

Add a **`deleteTicket`** helper right alongside your existing **`getTickets`**, **`addTicket`**, etc. For example:

```
js

const baseURL = 'http://localhost:3000';  // adjust to your server's URL

/**
 * DELETE /tickets/:id
 * Returns a 204 on success, 404 if not found, 500 on error.
 */
async function deleteTicket(id) {
  const resp = await fetch(`${baseURL}/tickets/${id}`, {
    method: 'DELETE'
  });
  if (resp.status === 404) {
    throw new Error('Ticket not found');
  }
  if (!resp.ok) {
    const body = await resp.json().catch(() => ({}));
    throw new Error(body.error || 'Unknown server error');
  }
}
```

Make sure this file is loaded **before** your inline **`<script>`** in **`index.html`** (it already is, based on your snippet).

---

2. Server-side: implement the DELETE endpoint

In your server application (e.g. **`server.js`**, **`app.js`**, or wherever you define your routes), add:

```
js

// assuming Express and a PostgreSQL pool, or adapt to your DB layer
const express = require('express');
const app = express();
const pool = require('./db');      // or however you import your database client

// Delete a ticket by ID
app.delete('/tickets/:id', async (req, res) => {
  const id = req.params.id;
  try {
    const result = await pool.query(
      'DELETE FROM tickets WHERE id = $1 RETURNING *',
      [id]
    );
    if (result.rowCount === 0) {
      // No ticket was deleted
      return res.status(404).json({ error: 'Ticket not found' });
    }
    // 204 No Content indicates success with no body
    return res.status(204).end();
  } catch (err) {
    console.error('Error deleting ticket from DB:', err);
    return res.status(500).json({ error: 'Internal server error' });
  }
});
```

Note:

- If you're using a different database abstraction (e.g. Sequelize, MongoDB, etc.), adapt the deletion call (`Model.destroy`, `db.collection.deleteOne`, etc.) accordingly.
- Ensure your server middleware allows JSON and CORS if your client and server are on different origins.

---

3. Test your flow

1. **Start your server** so that `DELETE /tickets/:id` is live.
2. **Build up** some tickets via the "Add Ticket" form (and include one with `Artist: Artist-Delete` / `Location: Location-Delete` / `Price: -1`).
3. **Switch** to the "View Tickets" tab, find the "Delete" button beside your test ticket, and click it.
4. The row should disappear. Refreshing the table confirms it's gone from the database.

That completes Task 2—client and server are now wired for delete! Let me know if you run into any errors or need adjustments.

where is script/ api _ script for task 2

In your project folder (the same one where you have `index.html`) there should be a sub-folder called `scripts`. Inside that you'll find (or should create) the file:

```
pgsql

/your-project
 ├─ index.html
 ├─ scripts
 │   └─ api_script.js   ← this is the file you edit for all your fetch wrappers
 └─ …other folders…
```

1. **Open the Explorer pane** in VS Code (the file-tree on the left).
2. **Find** the `scripts` folder and **click** to expand it.
3. **Open** `api_script.js`.
4. **If** you don't see `api_script.js`, right-click on the `scripts` folder → **New File** → name it `api_script.js`.

Once you've got `api_script.js` open, scroll to where you've defined your other API calls (e.g. `getTickets()`, `addTicket()`) and add the `deleteTicket(id)` function alongside them:

```js
const baseURL = 'http://localhost:3000';  // adjust if your server URL is different

// DELETE /tickets/:id
async function deleteTicket(id) {
  const resp = await fetch(`${baseURL}/tickets/${id}`, {
    method: 'DELETE'
  });
  if (resp.status === 404) {
    throw new Error('Ticket not found');
  }
  if (!resp.ok) {
    const body = await resp.json().catch(() => ({}));
    throw new Error(body.error || 'Unknown server error');
  }
}
```

Then save the file (`Ctrl+S` / `⌘S`) and your inline script in `index.html` will be able to call `deleteTicket(...)` when the Delete button is clicked.

```
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() {

} this is the code for the second step what do I imput

Here's how to fill in your **`deleteTicket`** stub so it actually sends the DELETE request. Add this right below your other API functions in **`scripts/api_script.js`**:

```js
/**
 * DELETE /delete_ticket/<id>
 * Flask endpoint that deletes a ticket by ID.
 * Returns 200 on success, 404 if not found.
 */
async function deleteTicket(id) {
    try {
        const response = await fetch(`http://127.0.0.1:5000/delete_ticket/${id}`, {
            method: 'DELETE'
        });

        if (response.status === 404) {
            throw new Error('Ticket not found');
        }
        if (!response.ok) {
            throw new Error('Server error on delete');
        }

        // Optionally read a JSON body if your server returns one:
        // const data = await response.json();
        // console.log('Delete response:', data);

    } catch (error) {
        console.error('Error with DELETE request:', error);
        throw error;  // so calling code knows it failed
    }
}
```

A few notes:

1. **URL**
   I'm assuming your Flask route is defined as:

   ```python
   @app.route('/delete_ticket/<int:id>', methods=['DELETE'])
   def delete_ticket(id):
       # ...
   ```

   If your route is named differently (e.g. **`/tickets/<id>`**), just adjust the URL in the **`fetch`** call to match.
2. **Error handling**
   We throw on non-OK so that in your front-end listener you can catch it and alert the user.
3. **Hooking it up**
   In your inline **`<script>`** you already have:

   ```js
   await deleteTicket(ticketId);
   ```

```
await updateTicketsTable();
```

which will now call this function.

Once that’s in place, rebuild/reload your client, create some dummy tickets, and click “Delete” next to the one with **Artist-Delete**. You should see the row disappear immediately and remain gone after a page refresh.

```
where do I  add this const express = require('express');
const app = express();
const pool = require('./db');      // or however you import your database client

// Delete a ticket by ID
app.delete('/tickets/:id', async (req, res) => {
  const id = req.params.id;
  try {
    const result = await pool.query(
      'DELETE FROM tickets WHERE id = $1 RETURNING *',
      [id]
    );
    if (result.rowCount === 0) {
      // No ticket was deleted
      return res.status(404).json({ error: 'Ticket not found' });
    }
    // 204 No Content indicates success with no body
    return res.status(204).end();
  } catch (err) {
    console.error('Error deleting ticket from DB:', err);
    return res.status(500).json({ error: 'Internal server error' });
  }
});?
```

You’ll put that snippet in your **server’s entry-point file**—usually called something like **`server.js`**, **`app.js`**, or **`index.js`**—where you configure your Express app and all your other routes. Here’s how:

1. **Open your server file**
   In VS Code’s Explorer, find the file where you already do things like

   ```js
   const express = require('express');
   const app = express();
   // …maybe body-parser, CORS, other routes…
   ```

   This is often at your project root.
2. **Ensure you have your database client imported**
   If you haven’t already, create (or confirm) a **`db.js`** alongside it that exports your **`pool`** (for PostgreSQL) or whatever DB handle you use. That file might look like:

```js
// db.js
const { Pool } = require('pg');
const pool = new Pool({ /* connection config */ });
module.exports = pool;
```

3. **Paste the DELETE route**
   Somewhere after:

```js
app.use(express.json());     // to parse JSON bodies
// …other middleware…
```

   and *before* you call **`app.listen(...)`**, add:

```js
const pool = require('./db');  // adjust path if your db.js lives elsewhere

// Delete a ticket by ID
app.delete('/tickets/:id', async (req, res) => {
  const id = req.params.id;
  try {
    const result = await pool.query(
      'DELETE FROM tickets WHERE id = $1 RETURNING *',
      [id]
    );
    if (result.rowCount === 0) {
      return res.status(404).json({ error: 'Ticket not found' });
    }
    return res.status(204).end();
  } catch (err) {
    console.error('Error deleting ticket from DB:', err);
    return res.status(500).json({ error: 'Internal server error' });
  }
});
```

4. **Restart your server**
   Save the file and restart (**`Ctrl+C`** your running process, then **`node server.js`** or **`npm start`**). Your DELETE endpoint will now be live at **`http://localhost:<port>/tickets/:id`**.

With that in place, your front-end's **`fetch('…/tickets/'+id, { method: 'DELETE' })`** will hit this route and remove the row from your database. Let me know if you run into any errors!

so here <!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 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!
        --><!-- Ticket Addition Form -->
        <form id="ticket-add-form"
          style="max-width: 400px; margin: 20px auto; padding: 20px; border: 1px solid #ccc; border-radius: 8px;">
          <h2 style="text-align: center; margin-bottom: 16px;">Add New Ticket</h2>
          <div style="display: flex; flex-direction: column; gap: 12px;">
            <label for="artist-input">Artist:</label>
            <input type="text" id="artist-input" name="artist" placeholder="Enter artist name" required
              style="padding: 8px; border-radius: 4px; border: 1px solid #aaa;">

            <label for="location-input">Location:</label>
            <input type="text" id="location-input" name="location" placeholder="Enter event location"
              required style="padding: 8px; border-radius: 4px; border: 1px solid #aaa;">

            <label for="price-input">Price:</label>
            <input type="number" id="price-input" name="price" placeholder="Enter ticket price" required
              min="0" style="padding: 8px; border-radius: 4px; border: 1px solid #aaa;">
          </div>

          <div style="display: flex; justify-content: space-between; margin-top: 20px;">
            <button type="button" id="ticket-add-reset-btn"
              style="padding: 10px 16px; border:none; border-radius:4px; background:#ddd; cursor:pointer;">Reset</button>
            <button type="button" id="ticket-add-submit-btn"
              style="padding: 10px 16px; border:none; border-radius:4px; background:#28a745; color:#fff; cursor:pointer;">Submit</button>
          </div>
        </form>


      </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 () {
                    const ticketId = button.id;                    // ID is on the button
                    try {
                        await deleteTicket(ticketId);              // call your API helper
                        await updateTicketsTable();                // refresh table
                    } catch (err) {
                        console.error('Error deleting ticket:', err);
                        alert('Could not delete ticket. See console for details.');
                    }
                });
            });
        }

        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>?
```

You don’t put any of that Express/`app.delete(...)` code inside your HTML page. That code belongs in your **server-side** entry-point (e.g. `server.js`, `app.js`, or whatever you named it)—not in `index.html`.

Here’s how to wire it up:

1. **Create or open your server file**
   At the root of your project, you should have something like:

   ```pgsql
   /your-project
   ├── index.html          ← your client
   ├── scripts/
   │   └── api_script.js   ← your client API calls
   ├── server.js           ← your Express app (or app.js)
   ├── db.js               ← your database pool / client
   └── package.json
   ```

2. **Ensure you have `express` and your DB client installed**

   ```bash
   npm install express pg   # or express + your DB library
   ```

3. **db.js**
   If you don’t already have it, create `db.js` to export your Postgres (or other) pool:

   ```js
   // db.js
   const { Pool } = require('pg');
   const pool = new Pool({
     user: 'your_db_user',
     host: 'localhost',
     database: 'your_db_name',
     password: 'your_db_password',
     port: 5432,
   });
   module.exports = pool;
   ```

4. **server.js**
   Create (or open) `server.js` and set up Express + your routes:

```
js

// server.js
const express = require('express');
const app = express();
const pool = require('./db');            // your connection to Postgres
const PORT = 5000;                       // or whatever port you want

// Middleware
app.use(express.json());                 // parse JSON bodies

// --- Your existing endpoints --- //
// e.g. app.get('/get_tickets', …)
//      app.post('/add_ticket', …)

// DELETE /delete_ticket/:id
app.delete('/delete_ticket/:id', async (req, res) => {
  const id = req.params.id;
  try {
    const result = await pool.query(
      'DELETE FROM tickets WHERE id = $1 RETURNING *',
      [id]
    );
    if (result.rowCount === 0) {
      return res.status(404).json({ error: 'Ticket not found' });
    }
    return res.sendStatus(200);
  } catch (err) {
    console.error('Error deleting ticket:', err);
    return res.status(500).json({ error: 'Internal server error' });
  }
});

app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

5. **Adjust your client to hit that endpoint**
   In **`scripts/api_script.js`** make sure your **`deleteTicket`** matches:

```
js

async function deleteTicket(id) {
  const resp = await fetch(`http://127.0.0.1:5000/delete_ticket/${id}`, {
    method: 'DELETE'
  });
  if (!resp.ok) throw new Error('Delete failed');
}
```

6. **Run your server**

```
bash

node server.js
```

   (Or **`npm start`** if you wired a start script.)

7. **Test it**
   - Spin up your client, add some tickets.
   - Switch to “View Tickets” and click “Delete” next to your **`Artist-Delete`** entry.
   - The row should vanish and the database record should be gone.

Remember: **server-side code lives in `server.js`** (or `app.js`), **not** in your HTML!

so here than ?

Your server-side code should live in a separate file—commonly named `server.js` or `app.js`—at the root of your project, not in your HTML. Here's a concrete example:

```pgsql
/your-project
├── index.html
├── scripts/
│   └── api_script.js
├── server.js       ← create this
├── db.js           ← your database connection
└── package.json
```

---

1. `db.js`

```js
// db.js
const { Pool } = require('pg');
const pool = new Pool({
  user: 'YOUR_DB_USER',
  host: 'localhost',
  database: 'YOUR_DB_NAME',
  password: 'YOUR_DB_PASSWORD',
  port: 5432,
});
module.exports = pool;
```

2. `server.js`

```js
// server.js
const express = require('express');
const app = express();
const pool = require('./db');      // imports the Pool from db.js
const PORT = 5000;

app.use(express.json());           // parse JSON bodies

// Existing endpoints
app.get('/get_tickets', async (req, res) => {
  try {
    const result = await pool.query('SELECT * FROM tickets ORDER BY id;');
    res.json(result.rows);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});

app.post('/add_ticket', async (req, res) => {
  const { artist, location, price } = req.body;
  try {
    const result = await pool.query(
      'INSERT INTO tickets (artist, location, price) VALUES ($1, $2, $3) RETURNING *;',
      [artist, location, price]
    );
    res.status(201).json(result.rows[0]);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal server error' });
  }
});

// **TASK 2** DELETE endpoint
app.delete('/delete_ticket/:id', async (req, res) => {
  const id = req.params.id;
  try {
    const result = await pool.query(
      'DELETE FROM tickets WHERE id = $1 RETURNING *;',
      [id]
    );
    if (result.rowCount === 0) {
      return res.status(404).json({ error: 'Ticket not found' });
    }
    return res.sendStatus(200);
  } catch (err) {
    console.error('Error deleting ticket:', err);
    return res.status(500).json({ error: 'Internal server error' });
  }
});

app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

---

3. Start the server

1. Install dependencies (if you haven't already):

```bash
npm install express pg
```

2. From your project root, run:

```bash
node server.js
```

3. Ensure your client's `deleteTicket(id)` fetch calls `http://127.0.0.1:5000/delete_ticket/${id}`.

Now when you click "Delete" in your front end, it will hit this endpoint in `server.js` and remove the ticket from your database.

```
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
async function deleteTicket(id) {
    try {
        const response = await fetch(http://127.0.0.1:5000/delete_ticket/${id}, {
            method: 'DELETE'
        });

        if (response.status === 404) {
            throw new Error('Ticket not found');
        }
        if (!response.ok) {
            throw new Error('Server error on delete');
        }

        // Optionally read a JSON body if your server returns one:
        // const data = await response.json();
        // console.log('Delete response:', data);

    } catch (error) {
        console.error('Error with DELETE request:', error);
        throw error;  // so calling code knows it failed
    }
}
```

You don't need to touch any more client-side code (your `deleteTicket(id)` is already correct). Now you just need to add the matching Flask route in your server app—right alongside your existing `/get_tickets` and `/add_ticket` handlers.

Assuming your Flask app lives in something like `app.py`, it might look like this:

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
import sqlite3                          # or psycopg2 / SQLAlchemy, etc.

app = Flask(__name__)
CORS(app)                                # so your client at 127.0.0.1:5500 can call it

# --- your existing DB setup ---
def get_db_connection():
    conn = sqlite3.connect('tickets.db')   # adjust for your setup
    conn.row_factory = sqlite3.Row
    return conn

# Existing: GET /get_tickets
@app.route('/get_tickets', methods=['GET'])
def get_tickets():
    conn = get_db_connection()
    tickets = conn.execute('SELECT * FROM tickets').fetchall()
    conn.close()
    return jsonify([dict(t) for t in tickets])

# Existing: POST /add_ticket
@app.route('/add_ticket', methods=['POST'])
def add_ticket():
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.execute(
        'INSERT INTO tickets (artist, location, price) VALUES (?, ?, ?)',
        (data['artist'], data['location'], data['price'])
    )
    conn.commit()
    new_id = cursor.lastrowid
    conn.close()
    return jsonify({'id': new_id, **data}), 201

# —————————————————— TASK 2: DELETE ————————————————
@app.route('/delete_ticket/<int:id>', methods=['DELETE'])
def delete_ticket(id):
    conn = get_db_connection()
    cursor = conn.execute('DELETE FROM tickets WHERE id = ?', (id,))
    conn.commit()
    if cursor.rowcount == 0:
        conn.close()
        return jsonify({'error': 'Ticket not found'}), 404

    conn.close()
    return '', 200
# ————————————————————————————————————

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Where to put it

- Open your Flask entry-point file (often called `app.py` or `server.py`).
- Paste the `@app.route('/delete_ticket/<int:id>', …)` block below (or next to) your existing GET/POST handlers.
- Save and restart your Flask server:

```bash
FLASK_APP=app.py flask run --port=5000
# or simply: python app.py
```

Once that's up, your front-end `deleteTicket(id)` calls will hit `/delete_ticket/42` (for example), remove the DB row, and return `200 OK` so your table refreshes correctly.

in navigating at server I have data business model and api as things I can select

It looks like your back-end code is organized into folders like **data**, **business**, **model**, and **api**. You'll want to add your DELETE-route handler into whatever file under **api/** is defining your Express (or Flask) endpoints. Here's how to proceed:

1. **Open the `api/` folder** in VS Code's Explorer.
2. **Locate your tickets-related routes**—often a file named something like `tickets.js`, `ticketRoutes.js`, or simply `index.js` inside that folder.
3. **Open that file**. You should see other route definitions there, for example:

```js
// api/tickets.js
const express = require('express');
const router = express.Router();
const { getTickets, addTicket } = require('../business/ticketService');

// GET /get_tickets
router.get('/get_tickets', async (req, res) => { … });

// POST /add_ticket
router.post('/add_ticket', async (req, res) => { … });

module.exports = router;
```

4. **Add your DELETE handler** right below the POST route (before the `module.exports` line):

```js
// DELETE /delete_ticket/:id
router.delete('/delete_ticket/:id', async (req, res) => {
  const id = req.params.id;
  try {
    // Delegate deletion to your business logic layer
    const deleted = await require('../business/ticketService').deleteTicket(id);
    if (!deleted) {
      return res.status(404).json({ error: 'Ticket not found' });
    }
    return res.sendStatus(200);
  } catch (err) {
    console.error('Error deleting ticket:', err);
    return res.status(500).json({ error: 'Internal server error' });
  }
});
```

5. **Wire the router up** in your main server file (often `server.js` or `app.js`) if it isn't already:

```js
js

const express = require('express');
const app = express();
app.use(express.json());

// Mount your tickets API
const ticketsRouter = require('./api/tickets');
app.use('/', ticketsRouter);

// Start server…
```

6. **Implement the service method** in your `business/` layer (e.g. in `business/ticketService.js`):

```js
js

// business/ticketService.js
const pool = require('../data/db');  // or wherever your Pool/client lives

async function deleteTicket(id) {
  const result = await pool.query(
    'DELETE FROM tickets WHERE id = $1 RETURNING *',
    [id]
  );
  return result.rowCount > 0;
}

module.exports = { getTickets, addTicket, deleteTicket };
```

7. **Save all files and restart** your server (`Ctrl+C` + `node server.js` or `npm start`).

Now your front-end calls to `DELETE http://127.0.0.1:5000/delete_ticket/:id` will be handled by that new route. Let me know if you need any help drilling into one of those folders!

```
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

# 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)
```

In your Flask app (e.g. in that same file), you need to:

1. **Change the route to include the ticket's ID**
2. **Implement the handler** by delegating to your `TicketManager`
3. **Add a `delete_ticket` method** to `TicketManager`

---

1) Update your Flask route

Replace this stub:

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

with:

```python
# TASK 2 — DELETE endpoint
@app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
def delete_ticket(ticket_id):
    tm = TicketManager()
    try:
        deleted = tm.delete_ticket(ticket_id)
        if not deleted:
            # nothing was removed
            return jsonify({'message': 'Ticket not found'}), 404
        return jsonify({'message': 'Deleted'}), 200
    except Exception as e:
        # bubble up any business-layer errors
        return jsonify({'message': str(e)}), 500
```

This makes your DELETE URL look like `/delete_ticket/123` and returns:

- **200** if successfully deleted
- **404** if the ID didn't exist
- **500** on any other error

---

2) Add the business logic

In your **business/ticket_manager.py**, you probably have something like:

```python
class TicketManager:
    def __init__(self):
        # init your DB connection…

    def retrieve_tickets(self):
        # …
    def insert_ticket(self, data):
        # …
```

Add a **`delete_ticket`** method:

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

    def delete_ticket(self, ticket_id: int) -> bool:
        """
        Delete the ticket with the given ID.
        Returns True if a row was deleted, False otherwise.
        """
        # if you’re using SQLite
        conn = self._get_connection()
        cursor = conn.execute(
            'DELETE FROM tickets WHERE id = ?',
            (ticket_id,)
        )
        conn.commit()
        deleted = cursor.rowcount > 0
        conn.close()
        return deleted

        # — or, if you’re using psycopg2/pg:
        # result = conn.cursor().execute(
        #     'DELETE FROM tickets WHERE id = %s RETURNING id;',
        #     (ticket_id,)
        # )
        # conn.commit()
        # return result.rowcount > 0
```

Adjust the SQL parameter syntax (**`?`** vs. **`%s`**) to match your database driver.

---

3) Update your client if needed

Your client already calls:

```js
await fetch(`http://127.0.0.1:5000/delete_ticket/${id}`, { method: 'DELETE' });
```

Make sure that matches the new route (**`/delete_ticket/:ticket_id`**).

Once you’ve made those two changes:

1. **Restart** your Flask server
2. **Create** a test ticket with

   ```pgsql
   Artist: Artist-Delete
   Location: Location-Delete
   Price: -1
   ```

3. **Switch** to “View Tickets” and click “Delete” next to it—your row should vanish, and the DB record be removed.

Let me know if any errors pop up in your console or logs!

Firefox can’t establish a connection to the server at 127.0.0.1:5000.

The site could be temporarily unavailable or too busy. Try again in a few moments.
If you are unable to load any pages, check your computer’s network connection.
If your computer or network is protected by a firewall or proxy, make sure that Zen is permitted to access the web. why is it not working

That error means your browser can’t reach anything listening on port 5000—most often because the Flask app isn’t actually running (or is running on the wrong interface/port). Let’s troubleshoot step by step:

1. **Make sure your Flask server is up**
   - In the terminal, cd into the folder containing your Flask entry-point (the file with `app.run(...)`).
   - Run it explicitly:

     ```bash
     python app.py
     ```

     or, if you’re using the Flask CLI:

     ```bash
     export FLASK_APP=app.py       # or whatever your filename is
     flask run --host=127.0.0.1 --port=5000
     ```

   - You should see something like:

     ```csharp
     * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
     ```

2. **Check which host it’s bound to**
   - In your code you have `app.run(debug=True, host='0.0.0.0', port=5000)`.
     - `0.0.0.0` means “listen on all network interfaces.”
     - From the same machine, `http://127.0.0.1:5000` or `http://localhost:5000` should both work.
   - If you only see `Running on http://0.0.0.0:5000`, try in your browser:

     ```bash
     http://localhost:5000/get_tickets
     ```

That's a GET route that actually returns data—going to just `/` without having a route for it would 404.

3. **Test with curl or HTTPie**
   In your terminal, try:

```bash
curl http://127.0.0.1:5000/get_tickets
```

   - If you get JSON back, your server's running fine, and the issue is likely in your browser URL.
   - If you still can't connect, the server process isn't listening properly.
4. **Look for errors in the terminal**
   - When you start the server, do you see any traceback or binding errors?
   - If there's another service on port 5000, Flask might have silently fallen back to a different port (check the startup log).
5. **Firewall / Proxy**
   - On most dev machines, there's no firewall blocking localhost. Unless you have corporate policy or custom rules, you can probably rule this out.
6. **Double-check your project structure**
   - Ensure that the file you're running is the one with all your routes defined.
   - Sometimes people have two files named `app.py` in different folders—verify you're in the right directory.

Quick checklist to fix:

1. **In your back-end folder**

```bash
python app.py
```

2. **See "Running on…"**
   - If you don't, you've hit an error—read the stack trace.
3. **Hit a real route**

```bash
http://127.0.0.1:5000/get_tickets
```

4. **If that works**, your DELETE route will be at

```bash
http://127.0.0.1:5000/delete_ticket/<id>
```

Let me know what you see in your terminal when you start the server, and whether `curl http://127.0.0.1:5000/get_tickets` returns anything!

so before I did your three changes it worked

Since it broke right after adding the DELETE route, it's almost certainly a syntax or import error in your `app.py` that's keeping Flask from starting. Let's isolate and fix it:

---

1. Check your terminal for the exact error

1. Stop any running Flask process (Ctrl+C).
2. From the directory containing your `app.py`, run:

   ```
   bash

   python app.py
   ```

3. **Read every line** of output. If there's a traceback—copy the **first** few lines (the exception type and message) and share them here.

---

2. Temporarily comment out the DELETE route

If you suspect the new code, open `app.py` and temporarily turn the route into a no-op, for example:

```
python

# @app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
# def delete_ticket(ticket_id):
#     pass
```

Then try `python app.py` again.

- **If it now starts**, the problem is within that route (likely indentation or a missing import).
- **If it still fails**, the error is elsewhere—paste the traceback.

---

3. Common pitfalls to double-check

- **Indentation**: Make sure your `def delete_ticket` block lines up with the other `@app.route` handlers (no extra spaces or tabs).
- **Parameter name**: The decorator should be `@app.route('/delete_ticket/<int:ticket_id>', ...)` and your function signature `def delete_ticket(ticket_id):`.
- **Imports**: You need `from flask import jsonify` (you already have it), but confirm you didn't accidentally remove or rename anything.

- **Business method**: In your handler you call `tm.delete_ticket(ticket_id)`. Make sure you actually added that method to `TicketManager`, and that import still resolves.

---

4. Re-add the route step by step

Once you confirm Flask can start with the route commented out:

1. **Uncomment the decorator**:

   ```
   python

   @app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
   ```

   Run `python app.py`. If it still starts, continue.

2. **Uncomment the signature line**:

   ```
   python

   def delete_ticket(ticket_id):
   ```

   Run again—if it still starts, good.

3. **Inside the function**, start with a simple return to test:

   ```
   python

   def delete_ticket(ticket_id):
       return jsonify({'message': 'OK'}), 200
   ```

   Run again—if this works, your problem is in the business layer.

4. **Finally**, replace the body with the call to your manager:

   ```
   python

   deleted = tm.delete_ticket(ticket_id)
   ...
   ```

   Run once more—this pinpoints exactly which line breaks things.

Go ahead and try those steps. If you still can't get the server to bind on port 5000, copy the exact traceback you see when running `python app.py` and I'll help you parse it.

this is one of the codes does it look ok ? 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
async function deleteTicket(id) {
    try {
        const response = await fetch(http://127.0.0.1:5000/delete_ticket/${id}, {
            method: 'DELETE'
        });

        if (response.status === 404) {
            throw new Error('Ticket not found');
        }
        if (!response.ok) {
            throw new Error('Server error on delete');
        }

        // Optionally read a JSON body if your server returns one:
        // const data = await response.json();
        // console.log('Delete response:', data);

    } catch (error) {
        console.error('Error with DELETE request:', error);
        throw error;  // so calling code knows it failed
    }
}
```

Your client-side wrappers all look syntactically correct and do what they need to:

1. **`getTickets()`**
    - Hits **`GET /get_tickets`**.
    - Throws on non-OK and logs errors.
    - Returns the parsed JSON array.
2. **`addTicket()`**
    - Posts JSON to **`POST /add_ticket`**.
    - Logs the response (you might want to check **`response.ok`** before **`.json()`** in a production setup, but for testing this is fine).
3. **`deleteTicket(id)`**
    - Calls **`DELETE /delete_ticket/<id>`**.
    - Throws on 404 or any non-OK status, so your caller can catch and alert.

A few small tweaks you might consider

- **Check `addTicket`'s response code**
  Right now you immediately do **`.then(response => response.json())`**. If the server returns a 500 or 404, you'll still call **`.json()`** on an error page. Instead you could write:

  ```js
  function addTicket(artist, location, price) {
    return fetch('http://127.0.0.1:5000/add_ticket', { … })
      .then(response => {
        if (!response.ok) throw new Error(`Add failed: ${response.status}`);
        return response.json();
      })
      .then(data => console.log(data))
      .catch(console.error);
  }
  ```

- **Surface delete responses**
  If you ever return a JSON body from your DELETE endpoint (e.g. a **`{ message: 'Deleted' }`**), you could uncomment the lines that read and log **`response.json()`**.
- **CORS and ports**
  Make sure your Flask app is running on port 5000 and has CORS enabled (otherwise your browser will block cross-origin requests from your file:// or localhost:5500).
- **Error returns**
  `getTickets()` currently swallows errors (returns **`undefined`** if the fetch fails). You might want to re-throw after **`console.error`** so your UI code can show an error state.

But in terms of "does it look okay?"—yes. Once your server's DELETE route is wired up at **`/delete_ticket/<int:ticket_id>`**, calling **`await deleteTicket(someId)`** from your front-end will correctly fire the HTTP request and let you handle the result.

can the error come from here

That client-side code looks fine—your `deleteTicket(id)` will correctly send a `DELETE` to `http://127.0.0.1:5000/delete_ticket/123` (for example). If you're getting "can't establish connection," the fault isn't in those wrappers but upstream. Here's what to double-check:

1. **Is Flask actually listening on port 5000?**
   - Run `python app.py` (or your start command) and look for:

     ```csharp
     * Running on http://127.0.0.1:5000/
     ```

   - If you don't see that, there's a startup error—inspect the traceback.
2. **Does your route URL match?**
   - You only get a connection error (not a 404) if there's nothing at all listening.
   - Try in your browser or with `curl`:

     ```bash
     curl http://127.0.0.1:5000/get_tickets
     ```

     If that hangs or errors, the server isn't reachable.
3. **Cors / Origin**
   - Make sure you used `CORS(app)` in Flask or otherwise allowed cross-origin calls from your front end (especially if you're serving the client on a different port).
4. **Network tab in dev tools**
   - Open your browser's Developer Tools → Network panel.
   - Click "Delete" and see the actual network request. Is it going to the right URL? Does it even fire? Or is it being blocked?
5. **Match the endpoint signature**
   - In Flask you need:

     ```python
     @app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
     ```

     and the client-side URL must be `/delete_ticket/123`, not `/delete_ticket` with a JSON body.
6. **Firewall / Proxy (unlikely)**
   - On localhost this seldom matters, but verify nothing is blocking port 5000.

Next step: Pick one of your existing working endpoints (`/get_tickets`), point your front end at it:

```js
await fetch('http://127.0.0.1:5000/get_tickets', { method: 'DELETE' });
```

You'll either see a CORS error, a 405 Method Not Allowed (meaning the server is up but wrong method), or a connection error. That will pinpoint whether it's your HTTP method, your URL, or your server availability. Let me know what you observe!

python: can't open file '/home/para/Desktop/experiment/e-ticket/server/app.py': [Errno 2] No such file or directory

That error just means you're not in the folder where your Flask entry-point lives (or it isn't named **`app.py`**). Let's locate it and run it:

---

1. Find your Flask file

In your project root (looks like **`/home/para/Desktop/experiment/e-ticket/`**), list what's in there:

```bash
cd /home/para/Desktop/experiment/e-ticket
ls -R
```

You should see something like:

```kotlin
api/                business/           client/          model/
data/               server.py           app.py?          requirements.txt
```

If you see **`server.py`** instead of **`app.py`**, that's your file. If you see a folder like **`server/`** with **`app.py`** inside, note that path.

---

2. Change into the right directory

If it's directly in the project root:

```bash
cd /home/para/Desktop/experiment/e-ticket
```

If it's inside a **`server/`** folder:

```bash
cd /home/para/Desktop/experiment/e-ticket/server
```

3. Run the correct file

Once you're in the folder containing the Flask file:

- If it's named **app.py**:

  ```
  bash

  python app.py
  ```

- If it's named **server.py**:

  ```
  bash

  python server.py
  ```

- Or, using the Flask CLI from the project root:

  ```
  bash

  export FLASK_APP=path/to/your_file.py
  flask run --host=0.0.0.0 --port=5000
  ```

You should then see:

```
csharp

 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

---

4. Verify

In a new terminal, try:

```
bash

curl http://127.0.0.1:5000/get_tickets
```

If you get JSON back, your server is correctly running on port 5000.

Let me know what the directory listing shows or what file name you find, and we can adjust the command!

so its called api.py

Great—so your Flask entry point is named `api.py`. Here's how to start it:

1. **Open a terminal** and navigate into the directory where `api.py` lives. For example, if your structure is:

```swift
/home/para/Desktop/experiment/e-ticket/
├── api.py
├── data/
├── business/
├── model/
└── client/
```

then:

```bash
cd /home/para/Desktop/experiment/e-ticket
```

2. **Run the server** by invoking Python on that file:

   ```bash
   python api.py
   ```

   You should see output like:

   ```pgsql
   * Serving Flask app "api.py" (lazy loading)
   * Environment: development
   * Debug mode: on
   * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
   ```

3. **Test a known endpoint** to confirm it's up:

   ```bash
   curl http://127.0.0.1:5000/get_tickets
   ```

   You should get back a JSON array (even if it's empty).

4. **Now hit your DELETE route** when you click the Delete button in your UI. It should connect without the previous "can't establish connection" error.

Alternative with the Flask CLI

If you prefer, you can also do:

```bash
export FLASK_APP=api.py
export FLASK_ENV=development
flask run --host=0.0.0.0 --port=5000
```

But `python api.py` is the simplest way. Let me know if that brings the server up successfully!

how do i navigate through files

You have two main ways to move around your project files:

---

1) From the Terminal (Command-Line)

1. **Open a terminal**
    - On Linux: your default “Terminal” app
    - On Windows WSL: Ubuntu (or Git Bash)
2. **Print your current location**

    ```bash
    pwd
    ```

    (Shows the “path” you’re in.)
3. **List what’s in a folder**

    ```bash
    ls
    ```

    (Or `ls -l` for more detail, `ls -a` to show hidden files.)
4. **Move into a directory**

    ```bash
    cd folderName
    ```

    For example, if you see a folder called `e-ticket`, you’d do:

    ```bash
    cd e-ticket
    ```
5. **Move up one level** (to the parent folder)

    ```bash
    cd ..
    ```
6. **Combine steps**
    You can jump multiple levels at once:

    ```bash
    cd ~/Desktop/experiment/e-ticket
    ```

7. **Quick view of the tree**
   If you have `tree` installed:

   ```
   bash

   tree -L 2
   ```

   (Shows folder structure two levels deep.)

---

2) From VS Code's Explorer Sidebar

1. **Open your project**
   - In VS Code: **File → Open Folder…** and select `/home/para/Desktop/experiment/e-ticket`.
2. **Use the Explorer pane** (left-hand side)
   - Click the little folder icons to expand and collapse sub-folders.
   - Single-click a file to open it in the editor.
3. **Right-click operations**
   - **New File** or **New Folder**
   - **Reveal in File Explorer** (or Finder on macOS)
   - **Copy Path** if you need the full file path for the terminal

---

Example: Getting to `api.py`

- **In Terminal**:

  ```
  bash

  cd ~/Desktop/experiment/e-ticket
  ls
  # you should see api.py among other folders
  ```

- **In VS Code**:
  - Look under your opened folder in the Explorer.
  - You'll see `api.py` listed—click it to open.

Once you're in the right folder, you can run:

```
bash

python api.py
```

to start your Flask server.

I ran api.py this is what I got File "/home/para/Desktop/experiment/e-ticket/server/api/api.py", line 6, in <module>
    from business.ticket_manager import TicketManager
ModuleNotFoundError: No module named 'business'

The import error is simply because Python doesn't know where to find your top-level `business/` package when you run `api.py` deep under `server/api/`. You have two straightforward ways to fix this:

Option A: Run the app from the project root and treat `server/` as a package

1. **Add `__init__.py` files** so that Python treats these directories as packages:

   ```markdown
   e-ticket/
   ├── server/
   │   ├── __init__.py
   │   ├── api/
   │   │   ├── __init__.py
   │   │   └── api.py
   │   └── business/
   │       ├── __init__.py
   │       └── ticket_manager.py
   ```

   An empty `__init__.py` is fine.

2. **From the project root** (the `e-ticket/` folder), start the Flask app as a module:

   ```bash
   cd /home/para/Desktop/experiment/e-ticket
   export FLASK_APP=server.api.api
   flask run --host=0.0.0.0 --port=5000
   ```

   Or equivalently:

   ```bash
   python -m flask run --host=0.0.0.0 --port=5000
   ```

   Now when `server/api/api.py` does

   ```python
   from business.ticket_manager import TicketManager
   ```

   Python will successfully resolve `e-ticket/server/business/ticket_manager.py`.

Option B: Tweak `sys.path` at the top of `api.py`

If you'd rather not introduce `__init__.py`, you can prepend the `server/` folder onto Python's module search path:

```python
# At the very top of server/api/api.py
import os, sys
# Add the "server/" directory to sys.path so "business" is importable
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from flask import Flask, request, jsonify, send_from_directory
from business.ticket_manager import TicketManager
# … rest of your code …
```

Then, just

```bash
cd /home/para/Desktop/experiment/e-ticket/server/api
python api.py
```

will work because `sys.path` now includes `…/server`, where `business/` lives.

Which to choose?

- **Option A** (adding `__init__.py` + running as a module) is the "Pythonic" way and scales better as your project grows.
- **Option B** is a quick hack that gets you running immediately.

Once you fix the import path, restart your server and you'll see it bind on port 5000 again. Then your client calls to `/delete_ticket/<id>` should connect normally.

how do I navigate back into directory

In your shell, you move "up" the folder hierarchy with `cd ..`. Each `..` represents the parent directory.

- **One level up** (e.g. from `…/server/api` → `…/server`):

  ```bash
  cd ..
  ```

- **Two levels up** (e.g. from `…/server/api` → your project root):

  ```bash
  cd ../..
  ```

You can always check where you are with:

```bash
pwd
```

And list the contents of the current folder with:

```bash
ls
```

So, for example, if you ran:

```bash
cd /home/para/Desktop/experiment/e-ticket/server/api
pwd
# /home/para/Desktop/experiment/e-ticket/server/api
```

Then to get back to `…/server`:

```bash
cd ..
pwd
# /home/para/Desktop/experiment/e-ticket/server
```

And to get all the way back to the project root:

```bash
cd ../..
pwd
# /home/para/Desktop/experiment/e-ticket
```

so on the web when i want to delete a ticket it says Could not delete ticket. See console for details.

That generic alert means your `deleteTicket(...)` call is throwing an error—let's drill in and see exactly what's going wrong.

---

1. Check the browser's DevTools

1. **Open the Developer Tools** (usually F12 or Ctrl+Shift+I).
2. **Go to the Console** tab:
    - Are you seeing an error message like `Error with DELETE request: …`?
    - Copy the full stack or message there.
3. **Switch to the Network** tab.
    - Click the "Delete" button in your app.

  - A new request should appear—click on it.
  - What's the **URL**? (should be `http://127.0.0.1:5000/delete_ticket/123` or similar)
  - What's the **Status Code**? (200, 404, 500, or something else)
  - If it's 404/500, what's in the **Response** pane?

Knowing the URL and status code will pinpoint whether:

- The request is even hitting your Flask server
- The route is mismatched (404)
- Your handler is throwing an exception (500)
- There's a network/CORS error (blocked, no status)

---

2. Verify your Flask route is live

In the terminal where you ran `python api.py`, you should see log lines for each request. Try:

```bash
curl -v http://127.0.0.1:5000/delete_ticket/1
```

- If you get a **connection refused**, your server isn't running on port 5000.
- If you get a **404**, the route signature doesn't match—maybe you forgot the `/<int:ticket_id>` in the decorator or didn't restart the server.
- If you get a **200** with an empty body, your endpoint is up.

---

3. Common fixes

1. **Restart your Flask server** after any code change.
2. **Ensure your route decorator** in `api.py` reads exactly:

   ```python
   @app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
   ```

3. **Match the client URL** in `deleteTicket(id)`:

   ```js
   await fetch(`http://127.0.0.1:5000/delete_ticket/${id}`, { method: 'DELETE' });
   ```

4. **Look at the Console error** and paste it here if it's not obvious.

Once you have the exact status code and any console message (or curl output), share that and we can zero in on the root cause.

i recived an error ticket not found

A 404 “Ticket not found” means your DELETE route fired correctly, but the ID you passed didn’t match any row in your database. There are two common reasons this happens:

1. **The ticket was never inserted**
   - You said you added a ticket with **Price: -1**. If your `TicketManager.insert_ticket` has validation that rejects negative prices, that record may never have been created.
   - **Verify** by switching to **View Tickets** and making sure you actually see a row for **Artist-Delete**, **Location-Delete**, **Price -1**.
2. **You don’t know the right ID**
   - Your table only shows Artist/Location/Price, so you don’t see the auto-generated `id`.
   - But your Delete button’s `id` attribute is set to that number. You need to pass that exact number to `delete_ticket/<id>`.

How to troubleshoot

1. **Log out the tickets array**
   In your `updateTicketsTable()` (in your `<script>`), before you build the HTML, add:

   ```js
   console.log('Current tickets:', tickets);
   ```

   Then reload the page and open DevTools → Console. You’ll see something like:

   ```js
   Current tickets: [
     { id: 1, artist: "Artist-Test", location: "...", price: 999 },
     { id: 7, artist: "Artist-Delete", location: "Location-Delete", price: -1 }
   ]
   ```

   From there you now know that to delete your test ticket you must call `/delete_ticket/7`.
2. **Make sure you actually created the test ticket**
   - Use your Add form to insert **Artist-Delete**, **Location-Delete**, **Price: -1**.
   - Then hit **View Tickets** and confirm you see that row in the table (and in the console log from step 1).
3. **Click Delete on the correct row**
   Once you see the ticket listed (with its actual `id`), click its Delete button. Your front-end code will grab that button’s `id` attribute and call, for example,

```
nginx

DELETE http://127.0.0.1:5000/delete_ticket/7
```

which should return **200** and remove the row.

(Optional) Make ID visible in your table

To make debugging easier, you can temporarily add an **ID** column:

1. In your HTML table header:

```
html

<th scope="col">ID</th>
```

2. And in `updateTicketsTable()`'s `entryHtml`:

```
js

const entryHtml = `
  <tr>
    <td>${ticket.id}</td>
    <td>${ticket.artist}</td>
    <td>${ticket.location}</td>
    <td>${ticket.price}$</td>
    <td class="text-end">
      <button id="${ticket.id}" …>Delete</button>
    </td>
  </tr>`;
```

That way you'll see the `id` right in the UI and know which one to delete.

Once you confirm that:

- The ticket exists in the DB
- You're passing the correct ID to `/delete_ticket/<id>`

then the 404 should go away and the row will disappear as expected. Let me know if it's still not working after that!

ok so I think the server does not know how to take the id is this code ok ?

That route signature is exactly what you want—so yes, the code looks syntactically correct. The fact that you're getting "Ticket not found" means the endpoint is being hit, but your business logic isn't finding a row with that ID. Let's verify a few things:

---

1. Confirm the URL & parameter are matching

- **Decorator** in `api.py`:

  ```python
  @app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
  def delete_ticket(ticket_id):
      # …
  ```

- **Client fetch** in `api_script.js`:

  ```js
  await fetch(`http://127.0.0.1:5000/delete_ticket/${id}`, { method: 'DELETE' });
  ```

Make sure those `<int:ticket_id>` and `${id}` line up exactly (no extra slashes, typos, etc.).

---

2. Add some debug logging

Temporarily sprinkle a couple of `print()` calls in your Flask handler to see exactly what's coming in:

```python
@app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
def delete_ticket(ticket_id):
    print(f"[DEBUG] Received delete for ID = {ticket_id}")
    tm = TicketManager()
    try:
        deleted = tm.delete_ticket(ticket_id)
        print(f"[DEBUG] TicketManager.delete_ticket returned {deleted}")
        if not deleted:
            return jsonify({'message': 'Ticket not found'}), 404
        return jsonify({'message': 'Deleted'}), 200
    except Exception as e:
        print(f"[ERROR] Exception in delete_ticket: {e}")
        return jsonify({'message': str(e)}), 500
```

Then restart your server (`python api.py`) and watch the terminal when you click **Delete**.

- **Do you see `Received delete for ID = 7`** (or whatever)?
- **Does `delete_ticket` return `True` or `False`**?

---

3. Verify your business logic

In `business/ticket_manager.py`, your method should look something like:

```python
def delete_ticket(self, ticket_id: int) -> bool:
    conn = self._get_connection()
    cursor = conn.execute(
        'DELETE FROM tickets WHERE id = ?',  # or '%s' if Postgres
        (ticket_id,)
    )
    conn.commit()
    deleted = cursor.rowcount > 0
```

```
    conn.close()
    return deleted
```

- If you're using SQLite, `?` is correct.
- If you're using Postgres via `psycopg2`, you'd use `%s` instead of `?`.

Make sure your SQL placeholder matches your DB driver.

---

4. Test via curl

Once you've added the debug prints, try manually calling the endpoint:

```bash
curl -v -X DELETE http://127.0.0.1:5000/delete_ticket/7
```

- Does the terminal print your debug line?
- Does `curl` report a `200 OK` or `404 Not Found`?

If the debug shows the correct ID and `deleted=True`, your front-end alert will disappear; if it's `False`, then that ID truly isn't in the table.

Next steps

1. **Run your Flask app**, watch for the debug logs.
2. **Ensure the ticket with that ID exists** – check your `/get_tickets` output or add an ID column temporarily in your table.
3. **Fix placeholder syntax** in `delete_ticket` if your DB driver expects a different placeholder.

Let me know what the debug output says, and we'll take it from there!

```
@app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
def delete_ticket(ticket_id):
    tm = TicketManager()
    try:
        deleted = tm.delete_ticket(ticket_id)
        if not deleted:
            # nothing was removed
            return jsonify({'message': 'Ticket not found'}), 404
        return jsonify({'message': 'Deleted'}), 200
    except Exception as e:
        # bubble up any business-layer errors
        return jsonify({'message': str(e)}), 500
```

Your Flask route looks correct—`<int:ticket_id>` will pull the numeric ID out of the URL and pass it into your function. Since you're getting a "Ticket not found" response, the issue is almost certainly in how your `TicketManager.delete_ticket(...)` is querying the database (or which IDs actually exist). Here's how to pinpoint it:

1. **Add debug logging in the route** to verify exactly what ID arrives and what your business layer returns:

   ```
   python

   @app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
   def delete_ticket(ticket_id):
       print(f"[DEBUG] delete_ticket called with ticket_id={ticket_id}")
       tm = TicketManager()
       try:
           deleted = tm.delete_ticket(ticket_id)
           print(f"[DEBUG] TicketManager.delete_ticket returned {deleted}")
           if not deleted:
               return jsonify({'message': 'Ticket not found'}), 404
           return jsonify({'message': 'Deleted'}), 200
       except Exception as e:
           print(f"[ERROR] Exception in delete_ticket: {e}")
           return jsonify({'message': str(e)}), 500
   ```

   Restart the server and click "Delete" in your UI—watch your terminal to see:

   - Which `ticket_id` you're actually receiving.
   - Whether `deleted` is `True` or `False`.

2. **Verify your business logic's SQL placeholder**
   In `business/ticket_manager.py`, your `delete_ticket` method should match your database driver. For example, if you're using SQLite it should be:

   ```
   python

   cursor = conn.execute(
       'DELETE FROM tickets WHERE id = ?',
       (ticket_id,)
   )
   ```

   But if you're on PostgreSQL with `psycopg2` or `pg8000`, it needs to be:

   ```
   python

   cursor = conn.execute(
       'DELETE FROM tickets WHERE id = %s',
       (ticket_id,)
   )
   ```

   Mismatched placeholders mean your query silently deletes zero rows every time.

3. **Confirm the ticket exists**
   Use `GET /get_tickets` (or check your database directly) to see the actual `id` values. You might have added "Artist-Delete" with an unexpected auto-ID. Either:

   - Log the full tickets list in your front end with `console.log(tickets)` before rendering (so you see each `ticket.id`),
   - Or check your DB directly with a SQL client.

Once you see which IDs exist and whether your `delete_ticket` call is actually removing a row, the mystery will be solved. Let me know what your debug prints reveal!

```
what do I put here    # TASK 2
    @staticmethod
    def delete_ticket(data):
        pass
```

In your `business/ticket_manager.py`, you want `delete_ticket` to:

1. Open a DB connection
2. Execute a `DELETE` against the `tickets` table for the given `id`
3. Commit and close
4. Return `True` if a row was deleted, `False` otherwise

Here's a drop-in implementation—adjust the placeholder syntax (`?` vs `%s`) for your driver as noted in the comments:

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

    @staticmethod
    def delete_ticket(ticket_id: int) -> bool:
        """
        Deletes the ticket with the given ID.
        Returns True if one row was deleted, False otherwise.
        """
        # 1) Get a connection. Replace this with however you open yours:
        conn = TicketManager._get_connection()   # or however you do it
        try:
            # 2) Execute the DELETE. Use '?' for SQLite, '%s' for Postgres
            cursor = conn.execute(
                'DELETE FROM tickets WHERE id = ?;',  # or 'id = %s;' for Postgres
                (ticket_id,)
            )
            # 3) Commit
            conn.commit()
            # 4) cursor.rowcount tells you how many rows were affected
            return cursor.rowcount > 0
        finally:
            conn.close()
```

A few notes:

- If your DB driver doesn't support `rowcount`, you can alternatively do a `SELECT` first to check existence, but most do.
- Make sure that `_get_connection()` (or whatever your constructor uses) returns a live connection object with `.execute()`, `.commit()`, and `.close()`.
- Once this is in place, your Flask route's call to `tm.delete_ticket(ticket_id)` will correctly return `True` or `False`, and you'll see "Deleted" vs "Ticket not found" as expected.

```
this is whats in there 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 what you should put in **`business/ticket_manager.py`** for the **`delete_ticket`** method:

```python
    # TASK 2
    @staticmethod
    def delete_ticket(ticket_id: int) -> bool:
        """
        Delete a ticket by its ID.
        Returns True if a row was deleted, False otherwise.
        """
        # Delegate deletion to the data layer
        data_service = SqlDataService()
        # Perform delete; SqlDataService.delete_ticket should return a boolean
        return data_service.delete_ticket(ticket_id)
```

Then, in your **data layer** (**`data/sql_data_service.py`**), add a corresponding **`delete_ticket`** there. For example, if you’re using SQLite:

```python
class SqlDataService:

    # … existing methods …

    def delete_ticket(self, ticket_id: int) -> bool:
        conn = self._get_connection()
        cursor = conn.execute(
            'DELETE FROM tickets WHERE id = ?',
            (ticket_id,)
        )
        conn.commit()
        deleted = cursor.rowcount > 0
        conn.close()
        return deleted
```

Note:

- If you're using PostgreSQL via **`psycopg2`**, replace **`?`** with **`%s`** in the SQL placeholder.
- Make sure your **`TicketManager.delete_ticket(...)`** signature matches the Flask route's parameter name (**`ticket_id`**).

With those two pieces in place, your Flask route:

```python
@app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
def delete_ticket(ticket_id):
    tm = TicketManager()
    if tm.delete_ticket(ticket_id):
        return jsonify({'message': 'Deleted'}), 200
    else:
        return jsonify({'message': 'Ticket not found'}), 404
```

will correctly remove the record from the database and return the proper status code.

this is the new error : server error on delete

The **"Server error on delete"** means your Flask route hit an exception and returned a 500. Let's fix that by:

1. **Adding debug logs** in your Flask handler so you see the error.
2. **Implementing the `delete_ticket` method** in your data layer (**`SqlDataService`**), since right now it's likely missing and causing an AttributeError.

---

1) Add debug logging in your Flask route

In **`api.py`**, change your DELETE handler to:

```python
@app.route('/delete_ticket/<int:ticket_id>', methods=['DELETE'])
def delete_ticket(ticket_id):
    print(f"[DEBUG] delete_ticket called with ticket_id={ticket_id}")
    tm = TicketManager()
    try:
        deleted = tm.delete_ticket(ticket_id)
        print(f"[DEBUG] TicketManager.delete_ticket returned {deleted}")
        if not deleted:
            return jsonify({'message': 'Ticket not found'}), 404
        return jsonify({'message': 'Deleted'}), 200
    except Exception as e:
        # print full traceback
        import traceback; traceback.print_exc()
        return jsonify({'message': str(e)}), 500
```

1. Restart your Flask server.
2. Click Delete in your UI.
3. Watch your terminal for the printed traceback — that will tell us exactly which line is blowing up.

---

2) Implement `delete_ticket` in your data layer

In `data/sql_data_service.py`, add:

```python
class SqlDataService:
    # … your existing code …

    def delete_ticket(self, ticket_id: int) -> bool:
        """
        Deletes the ticket with `id = ticket_id`.
        Returns True if a row was deleted, False otherwise.
        """
        conn = self._get_connection()
        cursor = conn.execute(
            # Use '?' placeholder for SQLite; use '%s' for Postgres
            'DELETE FROM tickets WHERE id = ?;',
            (ticket_id,)
        )
        conn.commit()
        deleted = cursor.rowcount > 0
        conn.close()
        return deleted
```

Then in `business/ticket_manager.py`, ensure your manager calls this correctly:

```python
    @staticmethod
    def delete_ticket(ticket_id: int) -> bool:
        data_service = SqlDataService()
        return data_service.delete_ticket(ticket_id)
```

> **Be sure** the signature is `delete_ticket(ticket_id: int)` (not `data`) so it matches the Flask route's argument.

---

3) Restart and test again

1. **Restart Flask** (`python api.py`).
2. In your browser **DevTools** → **Network** tab, click "Delete."
    - You should now see the DELETE request return **200**.
    - The row will vanish from your table.
3. If it still 500s, check your terminal's traceback to see exactly which line in your code is failing.

Let me know what the server's console prints after these changes!

```
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 what do I put in here?
```

Add Ticket Form

```
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
```

I’ve implemented `delete_ticket` in **SqlDataService** to:

- Open a direct SQLite connection
- Run `DELETE FROM tickets WHERE id = ?`
- Commit and check `cursor.rowcount` for deletion success
- Return `True`/`False` accordingly

Now, your `TicketManager.delete_ticket(ticket_id)` (which delegates to this) will return the right boolean, and the Flask route will respond with 200 or 404 as intended. Restart your server and test again!

give me just the part from task 2

```python
    # TASK 2: Delete a ticket by ID
    def delete_ticket(self, ticket_id: int) -> bool:
        """
        Deletes the ticket with the given ID.
        Returns True if a row was deleted, False otherwise.
        """
        conn = None
        try:
            conn = sqlite3.connect(self.path)
            cursor = conn.cursor()
            query = f'DELETE FROM {self.table_tickets} WHERE {self.field_tickets_id} = ?'
            cursor.execute(query, (ticket_id,))
            conn.commit()
            return cursor.rowcount > 0
        except sqlite3.Error as e:
            print("SQLite error on delete:", e)
            return False
        finally:
            if cursor:
                cursor.close()
            if conn:
                conn.close()
```

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

Below are the three `UPDATE` statements you can paste into the SQLite editor (and then hit **Execute SQL** → **Write Changes**) to adjust each ticket's price in-place. First, here are the computed multipliers:

Ticket	Days left	Time-markup = 0.3·(1–days_left/30)	Available	Avail-markup = 0.2·(1–avail/initial)	Total markup	Seat multiplier	Overall factor
A (VIP)	3	0.3·(1–3/30)=0.27	100/500	0.2·(1–100/500)=0.16	0.43	1.2	(1 + 0.43)·1.2 = 1.716
B (Std)	10	0.3·(1–10/30)=0.20	400/1200	0.2·(1–400/1200)=0.1333	0.3333	1.0	1 + 0.3333 = 1.3333
C (Std)	15	0.3·(1–15/30)=0.15	200/1800	0.2·(1–200/1800)=0.1778	0.3278	1.0	1 + 0.3278 = 1.3278

> *Round to two decimal places in SQL using `ROUND(...,2)`.*

```
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';
```

1. Open **Execute SQL** in DB Browser for SQLite.
2. Paste those three `UPDATE` statements.
3. Click **Execute All**.
4. Click **Write Changes** to commit.

Your ticket prices will now reflect the dynamic-pricing rules.

so where do I put the code

You don’t put those **`UPDATE`** statements in your application code—instead you run them right inside **DB Browser for SQLite**:

1. **Open DB Browser for SQLite** and load your **`database.db`** file (the one your app is using).
2. Click on the **“Execute SQL”** tab in the top toolbar.
3. In the SQL editor pane, paste the three **`UPDATE`** statements:

   ```
   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';
   ```

4. Click **“Execute All”** (or the ▶️ run button) to apply them.
5. Switch to the **“Browse Data”** tab and verify that the **`price`** column for those three rows has been updated.
6. Finally, click **“Write Changes”** (the floppy-disk icon) to commit your updates to the database file.

That’s it—your dynamic-pricing changes are now live in the DB.