

## 5-ba



I have the following task at hand can you help me guide through the steps of creating the necessary code to be able to fully finalise it: In this task you are expected to create an HTML form that allows you to add new tickets on the platform.

Tickets have the following representation:

•••ID

Artist

Location

Price

However, you are only required to add 3 input fields as the ID is automatically generated as a unique

identifier. In addition, there should be 2 buttons present, one to reset (clear) the input fields and one to

submit the ticket.

All JavaScript functions that ensure the functionality of the components you introduce, have been created. In order to ensure this functionality you must use the following IDs for your components:

- Artist input field ID: artist-input
- Location input field ID: location-input
- Price input field ID: price-input
- Reset button ID: ticket-add-reset-btn
- Submit button ID: ticket-add-submit-btn

After finishing your implementation you can test it by introducing the following ticket:

- Artist: Artist-Test
- Location: Location-Test
- Price: 999

You will be graded on:

1. Functionality
2. Page Appearance (you are expected to make the content introduced fit the page)
3. Written code

Below is a step-by-step guide plus a complete example of how to build the “Add Ticket” form. You’ll get:

1. **HTML** markup with the required IDs
2. **Basic CSS** to make it “fit” on the page
3. **JavaScript wiring** (you’ll just need to drop in your existing handler functions)

### 1. HTML Structure

```
html

<form id="ticket-add-form">
  <div class="form-group">
    <label for="artist-input">Artist</label>
    <input type="text" id="artist-input" name="artist" required />
  </div>
</form>
```

```

</div>

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

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

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

```

- The form itself has an ID (**ticket-add-form**) so you can easily select it as a container.
- Each input has exactly the IDs you need:
  - **artist-input**
  - **location-input**
  - **price-input**
- The two buttons are **<button>** tags with IDs:
  - **ticket-add-reset-btn** (type="button" so it won't submit by default)
  - **ticket-add-submit-btn** (type="submit")

## 2. Basic CSS

Add this to your stylesheet (or inside a **<style>** block) to give it a clean, centered look:

```

css

#ticket-add-form {
  max-width: 400px;
  margin: 2rem auto;
  padding: 1.5rem;
  border: 1px solid #ccc;
  border-radius: 8px;
  font-family: sans-serif;
  background: #fafafa;
}

.form-group {
  margin-bottom: 1rem;
}

.form-group label {
  display: block;
  margin-bottom: 0.3rem;
  font-weight: bold;
}

.form-group input {
  width: 100%;
  padding: 0.5rem;
  border: 1px solid #aaa;
  border-radius: 4px;
  box-sizing: border-box;
}

```

```
.button-group {
  display: flex;
  justify-content: space-between;
}

.button-group button {
  padding: 0.5rem 1.2rem;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

#ticket-add-reset-btn {
  background: #ddd;
}

#ticket-add-submit-btn {
  background: #28a745;
  color: white;
}
```

### 3. JavaScript Wiring

Assuming you already have two functions:

- `resetTicketForm()` — clears the inputs
- `submitNewTicket(artist, location, price)` — handles the creation

Wire them up like this:

```
html

<script>
  document.addEventListener('DOMContentLoaded', () => {
    const form = document.getElementById('ticket-add-form');
    const resetBtn = document.getElementById('ticket-add-reset-btn');
    const artistInput = document.getElementById('artist-input');
    const locationInput = document.getElementById('location-input');
    const priceInput = document.getElementById('price-input');

    // Reset button
    resetBtn.addEventListener('click', () => {
      // your provided function to clear fields
      resetTicketForm();
      // or, if you need it inline:
      // artistInput.value = '';
      // locationInput.value = '';
      // priceInput.value = '';
    });

    // Form submission
    form.addEventListener('submit', event => {
      event.preventDefault();
      const artist = artistInput.value.trim();
      const location = locationInput.value.trim();
      const price = parseFloat(priceInput.value);

      // call your existing handler
      submitNewTicket(artist, location, price);

      // Optionally clear after submit:
      resetTicketForm();
    });
  });
</script>
```

### Note:

- If `resetTicketForm()` and `submitNewTicket()` are global functions in another JS file, just ensure that file is loaded **before** this script block.
- If you want instant validation feedback, you could also check `.checkValidity()` on the form.

## 4. Testing

### 1. Fill in

- Artist → "Artist-Test"
- Location → "Location-Test"
- Price → 999

2. Click **"Submit"** → verify your JS handler ran and the ticket shows up (in your table/list).

3. Click **"Reset"** → all fields should clear.

## Summary

- **Functionality:** Inputs and buttons wired to your handlers
- **Appearance:** Centered form with spaced, styled fields
- **Code quality:** IDs match spec, semantic HTML, unobtrusive JS

Feel free to drop this into your project and let me know if you hit any bumps!



where should I put the HTML code

You'll want to put that form markup into whatever HTML document (or template) is responsible for rendering your "Add Ticket" screen or panel. Here are the most common spots:

### 1. Standalone HTML page

- If you have an `add-ticket.html` (or similar) file, drop the `<form>` block inside its `<body>` where you want it to appear.
- Just make sure to include your CSS (either via `<link href="...your-styles.css">` or a `<style>` block) and that your JS file(s) are loaded **after** the form (so the IDs exist when the script runs).

```
html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Add Ticket</title>
  <link rel="stylesheet" href="styles.css" />
</head>
```

```
<body>
  <!-- PUT THE FORM HERE -->
  <form id="ticket-add-form">
    ...your fields/buttons...
  </form>

  <script src="your-ticket-handlers.js"></script>
  <script src="ticket-add-form.js"></script>
</body>
</html>
```

## 2. Within a larger page or dashboard

- If your application has a main `index.html` or a `dashboard.html`, find the section (e.g. a `<main>` or a panel `<div id="content">`) where you want the “Add Ticket” interface.
- Paste the `<form>` there.

## 3. In a template (e.g. server-side or SPA framework)

- **Server-rendered (JSP, Thymeleaf, EJS, PHP, etc.)**
  - Locate the template file (e.g. `add-ticket.ejs` or `tickets.jsp`) and insert the form snippet in the body where your “new ticket” section goes.
- **Single-Page App (React/Vue/Angular)**
  - If you’re in React, you’d convert that form into JSX inside your `AddTicket` component:

```
jsx

export default function AddTicket() {
  return (
    <form id="ticket-add-form">
      { /* ...fields/buttons... */ }
    </form>
  );
}
```

- And then import/use `<AddTicket />` in the relevant view.

## Key things to watch

- **Order of assets:** CSS can load in `<head>`, but your JS that queries `document.getElementById(...)` must load after the form exists (either at the end of `<body>` or in a `DOMContentLoaded` callback).
- **File structure:** Keep your form’s HTML in the same directory as related views, or in a `partials/` folder if you’re reusing it.

Let me know what your project structure looks like, and I can point more precisely to the right file!



This is the code where I should implement the form, where exactly should this be inserted:

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

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

```

<title>E-Ticket</title>
<link rel="icon" href="../../assets/images/favicon.png" type="image/png">

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

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

</head>

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

</body>

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

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

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

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

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

      <table id="tickets-table" class="table table-hover align-middle" aria-hidden="true">
        <thead>
          <tr>
            <th scope="col">Artist</th>
            <th scope="col">Location</th>

```

```

        <th scope="col">Price</th>
        <th scope="col" class="text-end">Operation</th>
      </tr>
    </thead>
    <tbody id="tickets-table-body">
    </tbody>
  </table>
</div>

```

<!-- Add new ticket content -->

```

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

```

Important:

To ensure proper 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>

```

You should drop your `<form>` (and all of its inputs/buttons) right inside the inner wrapper of the **Add Ticket** panel—i.e. replace (or immediately below) the TASK 1 comment. Concretely, change this:

```

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 proper page functionality you are encouraged to write code strictly
            within the bounds of this component.

            Do not delete this comment!
        -->

    </div>
</div>

```

into something like this (you can leave the comment in place or remove it once you're comfortable):

```

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 ... Do not delete this comment! -->

    <form id="ticket-add-form" class="p-3">
      <div class="mb-3">
        <label for="artist-input" class="form-label">Artist</label>
        <input type="text" class="form-control" id="artist-input" name="artist" required>
      </div>

      <div class="mb-3">
        <label for="location-input" class="form-label">Location</label>
        <input type="text" class="form-control" id="location-input" name="location" required>
      </div>

      <div class="mb-3">
        <label for="price-input" class="form-label">Price</label>
        <input type="number" class="form-control" id="price-input" name="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="submit" id="ticket-add-submit-btn" class="btn btn-primary">
          Submit
        </button>
      </div>
    </form>

  </div>
</div>

```

## Why here?

- **Keeps TASK 1 bounds:** You're writing only inside that `<div>` so you won't accidentally break the surrounding layout or JS.
- **Bootstrapped:** Leverages your already-loaded Bootstrap CSS for spacing and form styles.
- **IDs match:** `artist-input`, `location-input`, `price-input`, `ticket-add-reset-btn`, `ticket-add-submit-btn` — exactly what your JavaScript expects.

Once that's in place, when you click "Add Ticket" in the navbar, `updatePage()` will show this div (with the form), and your click-handlers (already wired at the bottom of the file) will find the inputs/buttons by ID and do the right thing.



Task 1 is done now I have a new task give me a step by step guide on how can I solve this task, give clear explanation as I have very low coding experience: 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 "analyzed" 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

Here's a step-by-step walkthrough—no prior experience assumed—of how to wire up “Delete” on the front end and back end. We'll assume you're using a Node.js/Express server (but the concepts carry over to any web framework).

## 1. Client-Side: Hooking up the “Delete” button

You already have this function stub in your script:

```
js

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

### 1.1 Grab the ticket ID

Inside that listener, `this.id` is the ticket's unique ID (you set `<button id="${ticket.id}">...`).

### 1.2 Call your API

Use the Fetch API to send an HTTP **DELETE** request to the server:

```
js

button.addEventListener('click', async function () {
  const ticketId = this.id; // 1. get the ID
  const resp = await fetch(`/tickets/${ticketId}`, {
    method: 'DELETE'
  });
  if (!resp.ok) {
    return alert('Failed to delete ticket.');// 2. error handling
  }
  // 3. Refresh the table
  await updateTicketsTable();
});
```

- `fetch('/tickets/123', { method: 'DELETE' })` sends a request to **DELETE** `/tickets/123`.

- If the server replies with a non-2xx status (`resp.ok` false), we show an alert.
- Otherwise we call your existing `updateTicketsTable()` to re-draw the list.

## 2. Server-Side: Implementing the DELETE endpoint

Below is a minimal Express example. If your project uses a different structure or language, adapt the same ideas.

### 2.1 Install Express (if you haven't already)

```
bash

npm install express body-parser
```

### 2.2 Create or update `server.js`

```
js

const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const PORT = 3000;

// pretend "database":
let tickets = [
  // { id: '1', artist: 'Foo', location: 'Bar', price: 100 },
];

// helper to generate unique IDs:
const { v4: uuidv4 } = require('uuid');

app.use(bodyParser.json());

// GET all tickets
app.get('/tickets', (req, res) => {
  res.json(tickets);
});

// POST a new ticket
app.post('/tickets', (req, res) => {
  const { artist, location, price } = req.body;
  const newTicket = { id: uuidv4(), artist, location, price };
  tickets.push(newTicket);
  res.status(201).json(newTicket);
});

// — YOUR DELETE ROUTE —
app.delete('/tickets/:id', (req, res) => {
  const id = req.params.id;
  const before = tickets.length;

  // filter out the one with matching ID
  tickets = tickets.filter(t => t.id !== id);

  if (tickets.length === before) {
    // nothing was removed → invalid ID
    return res.status(404).json({ error: 'Ticket not found' });
  }

  // success!
  res.status(204).end();
});
//
```

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

## Why it works:

### 1. `app.delete('/tickets/:id', ...)`

- Listens for HTTP **DELETE** requests at `/tickets/ANY_ID_HERE`.

### 2. `req.params.id`

- Grabs the `:id` part of the URL.

### 3. `tickets.filter()`

- Rebuilds your array without the deleted ticket.

### 4. Status codes

- **204 No Content** means “success, nothing to return.”
- **404 Not Found** if that ID didn’t exist.

## 3. Wire your front end to the back end

### 1. Serve your static files from Express (if not already):

```
js
app.use(express.static('public'));
```

- Put your HTML/JS/CSS under **public/**.
- Now `fetch('/tickets/...')` will hit your Express routes.

### 2. Confirm CORS (if front end and back end are on different hosts/ports). For local dev on the same port with static files, you won’t need extra CORS setup.

## 4. Testing your delete

### 1. Start the server

```
bash
node server.js
```

### 2. Add some tickets via your form.

### 3. In the browser, click **Add Ticket** → fill in “Artist-Delete” / “Location-Delete” / -1 → **Submit**.

### 4. Switch to “View Tickets” and find the row with Artist-Delete.

### 5. Click the red “Delete” button on that row.

- You should see that row vanish instantly.

After you confirm the delete works on the test ticket, repeat clicking **Delete** on **all** of your test entries so your database is clean.

- **Client:** Attach `fetch(..., { method: 'DELETE' })` to each delete button, then re-draw the table.
- **Server:** Implement an Express `app.delete('/tickets/:id', ...)` route that removes the ticket and returns a 204.



Replace the `// TASK 2` comment with exactly the code that:

- Here's the full snippet to drop in:

Printed using [ChatGPT to PDF](#), powered by PDFCrowd [HTML to PDF API](#).



```

    }

    // 4. On success, re-fetch and redraw the table
    await updateTicketsTable();
  } catch (err) {
    // Network or other unexpected error
    console.error('Delete request failed:', err);
    alert('✖ Network error while deleting. Check console.');
```

## Step-by-step breakdown

1. `const ticketId = this.id;`

- Each delete button's `id` was set to the ticket's unique ID when you built the table rows.

2. `fetch(`/tickets/${ticketId}`, { method: 'DELETE' })`

- Sends an HTTP DELETE to your server endpoint (e.g. `DELETE /tickets/123`).

3. `if (!response.ok)`

- If the server responds with a status like 404 or 500, we show an alert.

4. `await updateTicketsTable();`

- Calls your existing function that reloads the tickets list from the server and redraws the `<tbody>`.

Just paste that in place of `// TASK 2`, and you'll have the full "delete" flow on the client side.



I received this what is this: `DELETE /tickets/6 HTTP/1.1" 404`

That line is just your server's access log telling you:

1. **What the client did:**

- It sent an HTTP **DELETE** request to the path `/tickets/6`
- Using HTTP version 1.1

2. **What your server replied:**

- It returned status code **404** ("Not Found")

## What "404" means here

A **404 Not Found** means your server didn't find any resource at `/tickets/6` to delete. In the context of our DELETE handler, it usually indicates one of two things:

1. **The route wasn't matched**

- Maybe your Express route isn't actually mounted at `/tickets/:id`
- Or you have middleware reordering that's catching the request first

## 2. The ticket ID didn't exist

- Your in-memory `tickets` array (or database) doesn't have an entry whose `id` equals "6", so your handler deliberately returned **404**

# How to debug

## 1. Check your DELETE route is defined correctly

```
js
app.delete('/tickets/:id', (req, res) => {
  // ...
});
```

Make sure that line appears *before* any `app.use(express.static(...))` or catch-all routes.

## 2. Log inside the handler

Put a `console.log('Deleting', req.params.id)` at the top of the function to confirm whether the handler is ever entered.

## 3. Verify your tickets data

Right before you filter, log out the `tickets` array. Do you see an object with `id`: "6"?

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

## 4. Ensure ID formats match

If your tickets use UUIDs (e.g. "a3f4-..."), but your client is sending a numeric ID like 6, they won't match. Make sure the button's `id` attribute is exactly the same string your server expects.

## 5. Test with an existing ID

- In your browser console or via `curl`, try deleting something you know exists:

```
sql
curl -X DELETE http://localhost:3000/tickets/<some-real-id>
```

- See if that also yields 404 or succeeds.

Once you get your route firing and your IDs aligned, the **404** should go away and your delete button will actually remove the ticket and return a 204 or 200 instead.



so this is the code where I made the task 2 changes but the delete button is not yet working receiving the error mentioned above, what seems to be the problem: <!DOCTYPE html>  
<html lang="en">  
  
<head>

```

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

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

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

</head>

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

</body>

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

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

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

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

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

      <table id="tickets-table" class="table table-hover align-middle" aria-hidden="true">
        <thead>
          <tr>

```

```

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

```

<!-- Add new ticket content -->

```

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

```

Important:

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

Do not delete this comment!

-->

```

<form id="ticket-add-form">
    <div class="form-group">
        <label for="artist-input">Artist</label>
        <input type="text" id="artist-input" name="artist" required />
    </div>

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

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

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

```

```

</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
      // 1. Get the ID from this button's id attribute
      const ticketId = this.id;

      try {
        // 2. Send DELETE /tickets/:id
        const response = await fetch(`/tickets/${ticketId}`, {
          method: 'DELETE'
        });

        // 3. If server returned an error status, alert and stop
        if (!response.ok) {
          alert('Could not delete ticket. Please try again.');
```

```

    } catch (err) {
      // Network or other unexpected error
      console.error('Delete request failed:', err);
      alert(' Network error while deleting. Check console.');
```

```

    }
  });
});
}

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>

```

A **404** there means “I looked for a DELETE handler at `/tickets/6` and didn’t find one.” In other words, your client-side code is firing correctly, but your server never matched that URL and returned **Not Found**.

## 1. Verify your server has a DELETE route

If you’re using Express, somewhere in your server setup you must have:

```

js

app.delete('/tickets/:id', (req, res) => {
    // ...
});

```

Make sure:

- It’s defined **before** any “catch-all” or static-file middleware (so it actually gets a chance to run).
- The path matches exactly `/tickets/:id` (no extra `/api` prefix, no version number, etc.).

### Quick debug

Inside that handler, add:

```
js

console.log('DELETE handler hit for ID:', req.params.id);
```

Then try deleting again. If you don't see that log, the route isn't being reached at all.

## 2. Check your ticket store

If your route is firing but still returns 404, your code probably does something like:

```
js

const before = tickets.length;
tickets = tickets.filter(t => t.id !== req.params.id);
if (tickets.length === before) {
  return res.status(404).json({ error: 'Ticket not found' });
}
res.status(204).end();
```

In that case, log out your in-memory array just before filtering:

```
js

console.log('Before delete:', tickets);
```

Make sure there actually is an object whose `.id` exactly equals the string you're deleting. Sometimes IDs are UUIDs (e.g. "a1b2-c3d4") but your buttons are numeric (6), so they never match.

## 3. Match your client path to the server path

You're calling:

```
js

await fetch(`/tickets/${ticketId}`, { method: 'DELETE' });
```

If your backend actually listens under a different base path (for instance `/api/tickets/:id`), you'll need to change that to:

```
js

await fetch(`/api/tickets/${ticketId}`, { method: 'DELETE' });
```

Or—better yet—centralize your API URLs in `api_script.js`:

```
js

// api_script.js
export async function deleteTicket(id) {
  const resp = await fetch(`/tickets/${id}`, { method: 'DELETE' });
  if (!resp.ok) throw new Error('Delete failed');
}
```

Then on the client:

```
js

await deleteTicket(ticketId);
await updateTicketsTable();
```

## 4. Put it all together

1. **Ensure** your server has:

```
js

app.delete('/tickets/:id', (req, res) => {
  console.log('DELETE handler hit for:', req.params.id);
  // filter out and send 204 or 404...
});
```

2. **Restart** your server, watch for that console message when you click “Delete”.

3. **Confirm** the IDs line up (log your `tickets` array).

4. **Adjust** the client `fetch` path if your API lives under a prefix.

Once your server acknowledges the DELETE, your client’s `updateTicketsTable()` will pull the new list—sans the deleted ticket—and the row will disappear.



what is API

An **API** (Application Programming Interface) is simply a set of rules and endpoints that lets one piece of software talk to another. Think of it like a restaurant menu:

1. **Menu = API specification**

- It lists all the “dishes” (operations) you can order, and tells you what ingredients (parameters) you need.

2. **You (the client)**

- You decide what you want, pick it off the menu, and send your order.

3. **Kitchen (the server)**

- The kitchen prepares exactly what you asked for and brings it back.

In web development, your browser (or JavaScript code) is the **client**, and your server (Node.js/Express, for example) is the **kitchen**.

## How it works in our ticket app

- **Endpoint:** DELETE `/tickets/:id`

- This is like the menu item “Delete a ticket by its ID.”
- **Parameters:**
  - The `:id` in the path is the only “ingredient” you need.
- **Request:**
  - Your front-end code says “Hey server, please delete ticket #6” by doing:

```
js
fetch('/tickets/6', { method: 'DELETE' });
```

- **Response:**
  - The server checks its database (or array), removes ticket #6, and replies:
    - **204 No Content** → success (kitchen brought it back, but there’s nothing to show)
    - **404 Not Found** → no such ticket (kitchen says “I don’t have that on the menu”)

## Why APIs matter

- **Separation of concerns:**
  - Front end doesn’t need to know how the server stores data. It just follows the menu.
- **Reusability:**
  - Many different clients (web, mobile, other services) can all use the same API.
- **Security & Validation:**
  - The server can check inputs, enforce permissions, and control exactly what operations are allowed.

## Real-world analogy

- When you use your phone’s “weather app,” that app is calling a **weather API** somewhere out on the Internet, asking “What’s the forecast for New York City right now?” The server responds with a data structure (JSON) that tells your app the temperature, humidity, etc., and then your app displays it nicely.

**In short:** An API is a formal, documented way for one program (client) to ask another program (server) to do something—like create, read, update, or delete tickets—without either one having to know the inner workings of the other.



I have this code in API how should I use this info to solve my problem: 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

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

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)

```

Below is how you wire up your existing Flask API and your front-end code so that clicking “Delete” on a ticket actually removes it from the database and refreshes the table.

## 1. Server-side: implement /delete\_ticket

In your Flask app, you already have a stub:

```

python

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

```

Replace it with:

```

python

@app.route('/delete_ticket', methods=['DELETE'])
def delete_ticket():
    tm = TicketManager()
    try:
        # 1. Parse the JSON body to get the ticket ID
        data = request.get_json()
        ticket_id = data.get('id')

        # 2. Ask the TicketManager to delete it
        tm.delete_ticket(ticket_id)

        # 3. Return a success response
        return jsonify({'message': 'OK'}), 200

    except KeyError:
        # If no 'id' in JSON
        return jsonify({'message': 'Missing ticket id'}), 400

    except Exception as e:
        # If TicketManager throws (e.g. ticket not found)
        return jsonify({'message': str(e)}), 404

```

## How it works

#### 1. `request.get_json()`

- Reads the JSON the client sent.

#### 2. `ticket_id = data.get('id')`

- Grabs the `id` field from that JSON.

#### 3. `tm.delete_ticket(ticket_id)`

- You must have (or add) a method in `TicketManager` that knows how to delete a ticket by its ID from your database.

#### 4. Return codes

- **200 OK** on success
- **400 Bad Request** if the client forgot to send an `id`
- **404 Not Found** if `delete_ticket` in your manager throws because that ID doesn't exist

## 2. Business layer: implement deletion logic

In `business/ticket_manager.py` add something like:

python

```
class TicketManager:
    # ... your existing methods ...

    def delete_ticket(self, ticket_id):
        # 1. Load tickets from your DB (or file, or memory)
        tickets = self.retrieve_tickets()

        # 2. Find & remove the one with matching ID
        updated = [t for t in tickets if t['id'] != ticket_id]

        if len(updated) == len(tickets):
            # no ticket was removed → wrong ID
            raise Exception(f'Ticket {ticket_id} not found')

        # 3. Save the updated list back to the DB
        self._save_all_tickets(updated)
```

- Replace `self._save_all_tickets` with whatever persistence you use (database call, file write, etc.).

## 3. Client-side: call your new DELETE endpoint

In your HTML/JS page, you already have:

js

```
// TASK 2
const response = await fetch(`/tickets/${ticketId}`, { method: 'DELETE' });
```

Change that to point at your Flask route and send JSON:

```
js

button.addEventListener('click', async function () {
  const ticketId = this.id;

  try {
    // 1. Send DELETE to /delete_ticket with JSON body { id: ticketId }
    const response = await fetch('/delete_ticket', {
      method: 'DELETE',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ id: ticketId })
    });

    // 2. Check for errors
    if (!response.ok) {
      const err = await response.json();
      alert('Error deleting: ' + err.message);
      return;
    }

    // 3. Refresh the table
    await updateTicketsTable();
  } catch (err) {
    console.error(err);
    alert('Network error—could not delete.');
```

## Why these changes?

### 1. URL

- Your Flask route is `/delete_ticket`, not `/tickets/:id`.

### 2. JSON body

- Flask's `request.get_json()` needs a JSON payload.

### 3. Headers

- **Content-Type:** `application/json` tells the server “I’m sending JSON.”

## 4. Testing

1. **Restart your Flask server** so it picks up the new route.

2. **Create** a test ticket in your UI with

- Artist: **Artist-Delete**
- Location: **Location-Delete**
- Price: **-1**

3. **Switch to “View Tickets”**, find that row, and click its **Delete** button.

4. **Observe**

- The row disappears immediately.
- In your Flask console you should see no 404s.

5. **Check persistence**

- Refresh the entire page or re-load tickets via API: that ticket should be gone.



Once all this is wired up, your “Delete” operation will work end-to-end: clicking the button sends a DELETE to your Flask API, the API calls your business logic to remove it from the database, and the front end refreshes the table to reflect the change.



```
in business ticket manager I have the following coe: 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))

    @staticmethod
    def delete_ticket(data):
        pass
```

You just need to grab the **id** out of the incoming **data** dict and forward it to your **SqlDataService**’s delete method. Assuming your **SqlDataService** has a **delete\_ticket(ticket\_id)** method, here’s how to fill in the stub:

```
python

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))

@staticmethod
def delete_ticket(data):
    # 1. Pull the ticket ID out of the JSON payload
    ticket_id = data.get('id')
    if ticket_id is None:
        raise KeyError("Missing 'id' in delete request")

    # 2. Call your data layer to delete
    data_service = SqlDataService()
    deleted_rows = data_service.delete_ticket(ticket_id)

    # 3. If nothing was deleted, raise so the API can return 404
    if deleted_rows == 0:
        raise Exception(f"Ticket {ticket_id} not found")

    # 4. Otherwise, return success (you can return deleted_rows if you like)
    return deleted_rows

```

## What this does

1. `data.get('id')`  
Reads the "id" field from the JSON you sent in the DELETE request.
2. `data_service.delete_ticket(ticket_id)`  
Calls down into your SQL layer to execute something like

```

sql

DELETE FROM tickets WHERE ticket_id = ?

```

### 3. Error handling

- If no `id` was provided, we throw a **KeyError** (your Flask route will catch this and issue a 400).
- If `delete_ticket` returns `0` (no rows affected), we throw a generic **Exception**; your Flask route will catch it and issue a 404.

## Next steps

1. **Implement (or verify) `delete_ticket` in `SqlDataService`** so that it executes the SQL **DELETE** and returns the number of affected rows.
2. **Restart your Flask app** and test again by clicking the Delete button in your UI.

With this in place, your business layer can now delete a ticket by ID, and your API route will return a 200 on success or a 404 if the ticket didn't exist.



I have added the above mentioned code in the business ticket manager file, then i had an indentation problem at the last `@staticmethod`. I pressed tab and then I received the following message in the displaying tab :

Unable to connect

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.

That browser message just means your front-end tried to talk to `localhost:5000` but nothing was listening there—your Flask server isn't running (or it crashed on startup). Here's how to get it back online:

## 1. Fix the indentation error in TicketManager

Make sure your `delete_ticket` method is indented exactly like this (no mix of tabs and spaces):

```
python

class TicketManager:

    @staticmethod
    def retrieve_tickets():
        ...

    @staticmethod
    def insert_ticket(data):
        ...

    @staticmethod
    def delete_ticket(data):
        ticket_id = data.get('id')
        if ticket_id is None:
            raise KeyError("Missing 'id' in delete request")

        data_service = SqlDataService()
        deleted_rows = data_service.delete_ticket(ticket_id)
        if deleted_rows == 0:
            raise Exception(f"Ticket {ticket_id} not found")

        return deleted_rows
```

1. Make sure **all** four spaces before each line inside `delete_ticket` are spaces (not a tab).
2. Make sure the `@staticmethod` above it lines up with the other decorators (no leading tab).

Save the file.

## 2. Restart your Flask server

In your terminal, cd into your API project directory and run:

```
bash

python path/to/your_api_file.py
```

(or if you use the Flask CLI)

```
bash

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

Watch the terminal:

- You should see **\* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit).**
- If there's a Python traceback instead, read the first error line—it'll tell you exactly which file and line is mis-indented or otherwise broken.

## 3. Verify the DELETE endpoint is up

Once **flask** reports it's running, open a separate terminal and try:

```
bash

curl -i -X DELETE http://127.0.0.1:5000/delete_ticket \
-H "Content-Type: application/json" \
-d '{"id": "some-test-id"}'
```

You should get back a **200 OK** (or a **400/404** with a JSON error). If you instead get “Connection refused” or no response, the server still isn't up.

## 4. Reload your front end

Once the API is successfully listening:

1. Reload the page in your browser.
2. Navigate to **View Tickets** so the front end re-binds your delete buttons.
3. Click **Delete** again.

Now you should no longer see the “Can't establish a connection” error, and your delete flow will work end-to-end.

### Quick summary

- **Indentation errors** crash Flask at startup—fix them first.
- **Restart** Flask after every code change.
- **Test** with **curl** before trying the UI.

That should get your server back online so your delete button can finally talk to it.



how do I check if I am in the right directory in the terminal ?

You can verify where you are and what's in that folder with just a couple of commands:

## 1. Check your current directory

- **macOS / Linux / WSL**

```
bash
```

```
pwd
```

Prints the "present working directory," e.g.:

```
bash
```

```
/home/your-user/projects/ticket-api
```

- **Windows PowerShell**

```
powershell
```

```
Get-Location
```

or simply

```
powershell
```

```
pwd
```

## 2. List the files in that directory

Once you know where you are, make sure your Flask app file (e.g. `app.py` or `server.py`) lives there:

- **macOS / Linux / WSL**

```
bash
```

```
ls
```

or for more detail:

```
bash
```

```
ls -l
```

- **Windows PowerShell**

```
powershell
```

```
dir
```

You should see something like:

```
python-repl

app.py
business/
data/
model/
requirements.txt
...
```

### 3. Change directories if you're in the wrong place

If `pwd` shows you're not in the folder containing your Flask file, navigate there:

```
bash

cd path/to/your/api/folder
```

For example, if you're currently in `/home/you` but your project is in `/home/you/projects/ticket-api`, do:

```
bash

cd projects/ticket-api
```

Then re-run `pwd` and `ls` to confirm you're in the right spot before starting Flask:

```
bash

python app.py
```

(or whichever command you use to launch your server)



Ok I have finalised the mentioned steps but when I press the delete button I have a new error: Error deleting: 'str' object has no attribute 'get'. What does this mean and how can I solve it?

That message means you passed a **string** into your `TicketManager.delete_ticket(data)` method, and inside that method you did `data.get('id')`—but strings don't have a `.get()` method, so Python blows up with:

```
'str' object has no attribute 'get'
```

### Why it happened

In your Flask route you probably did something like:

```
python

data = request.get_json()
ticket_id = data.get('id')
tm.delete_ticket(ticket_id)  # <-- passing a string, not a dict
```

Then in your manager you have:

```
python

@staticmethod
def delete_ticket(data):
    ticket_id = data.get('id')  # <-- here data is a string!
    ...
```

So your **data** parameter is actually the string **ticket\_id**, not the JSON dict.

## Two ways to fix it

### Option A) Change the route to pass the whole dict

Keep your manager signature as **delete\_ticket(data)** expecting a dict:

```
python

@app.route('/delete_ticket', methods=['DELETE'])
def delete_ticket():
    data = request.get_json()  # e.g. {'id': '6'}
    tm = TicketManager()
    tm.delete_ticket(data)  # pass the dict
    return jsonify({'message': 'OK'}), 200
```

### Option B) Change the manager to accept ticket\_id directly

Update your **TicketManager** so that **delete\_ticket** takes the string ID:

```
python

class TicketManager:

    @staticmethod
    def delete_ticket(ticket_id):  # now expects a string
        if ticket_id is None:
            raise KeyError("Missing 'id' in delete request")

        data_service = SqlDataService()
        deleted_rows = data_service.delete_ticket(ticket_id)

        if deleted_rows == 0:
            raise Exception(f"Ticket {ticket_id} not found")

        return deleted_rows
```

And in your route do:

```
python
```

```
@app.route('/delete_ticket', methods=['DELETE'])
def delete_ticket():
    payload = request.get_json()
    ticket_id = payload.get('id')
    TicketManager.delete_ticket(ticket_id)
    return jsonify({'message': 'OK'}), 200
```

## Which to choose?

- **Option A** keeps your manager method signature the same (it takes a dict), but makes sure you actually pass the dict from the route.
- **Option B** is arguably cleaner: a manager method that deletes by ID shouldn't need a whole dict, just the ID string.

## Quick checklist

1. **Pick one** of the above options.
2. **Make sure your Flask route and your manager signature match** (both expect the same type).
3. **Restart your server** and test again.

Once they line up, you'll no longer see the `'str' object has no attribute 'get'` error.



Ok I solved this problem now whenb I try ti delete I have a new error: Error deleting: SqlDataService.delete\_ticket() takes 1 positional argument but 2 were given

This error is telling you that your `SqlDataService.delete_ticket` method is currently defined to take only the implicit `self` parameter, but you're calling it with two arguments (Python passes `self` automatically when you call an instance method, so your one extra argument becomes the second positional parameter).

## 1. Fix the method signature

Open `data/sql_data_service.py` and find your `delete_ticket` stub. It probably looks something like:

```
python
```

```
class SqlDataService:
    # ...

    def delete_ticket(self):
        # placeholder
        pass
```

Change it to accept the ticket ID:



python

```
class SqlDataService:
    # ...

    def delete_ticket(self, ticket_id):
        """
        Deletes the ticket with the given ticket_id from the database.
        Returns the number of rows affected (0 if none, 1 if deleted).
        """
        # ...implementation goes here...
```

## 2. Implement the deletion in SQL

Here's a typical SQLite-based example. Adapt to your actual database connection logic:

python

```
import sqlite3
import os

class SqlDataService:
    def __init__(self):
        # adjust path to your DB file
        db_path = os.path.join(os.path.dirname(__file__), 'tickets.db')
        self.conn = sqlite3.connect(db_path, check_same_thread=False)
        self.conn.row_factory = sqlite3.Row

    def delete_ticket(self, ticket_id):
        cursor = self.conn.cursor()
        # Use your actual table and column names
        cursor.execute(
            "DELETE FROM tickets WHERE ticket_id = ?",
            (ticket_id,)
        )
        self.conn.commit()
        return cursor.rowcount
```

- `(ticket_id,)`: a single-element tuple for the parameter.
- `cursor.rowcount`: how many rows were deleted (0 if none matched).

## 3. Re-test your delete flow

1. **Restart** your Flask server so it picks up the updated method signature.
2. In your browser UI, click **View Tickets**, then click the red **Delete** on one of the rows.
3. If everything's wired up, you should see no errors, and the row will disappear.

### Why this fixes it

- **Before:** `delete_ticket(self)` accepted only `self`, so calling `delete_ticket(ticket_id)` passed two arguments (`self` and `ticket_id`), triggering the mismatch error.
- **After:** `delete_ticket(self, ticket_id)` matches the call, and your SQL logic actually removes the row.

Let me know if you hit any other bumps!



my delete code is missing thios is what I have tell me what I need to add in order to make this owrk:

```
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

def delete_ticket(self):
    pass
```

You need to give your `delete_ticket` method a `ticket_id` parameter and actually execute a **DELETE** against the `tickets` table, then return how many rows were affected. Here's a drop-in implementation for your `SqlDataService`:

```
python

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)
    raise
finally:
    if cursor:
        cursor.close()
    if conn:
        conn.close()

# ... your retrieve_tickets and insert_ticket stay the same ...

def delete_ticket(self, ticket_id):
    """
    Deletes a ticket by its ID.
    Returns the number of rows deleted (0 if none matched, 1 if success).
    """
    sql = (
        f"DELETE FROM {self.table_tickets} "
        f"WHERE {self.field_tickets_id} = ?"
    )

    # Open a fresh connection so we can inspect rowcount
    conn = sqlite3.connect(self.path)
    cursor = conn.cursor()
    try:
        cursor.execute(sql, (ticket_id,))
        conn.commit()
        return cursor.rowcount
    finally:
        cursor.close()
        conn.close()

```

## What changed

### 1. Signature

python

```
def delete_ticket(self, ticket_id):
```

Now you accept the ID you want to delete.

### 2. SQL DELETE

sql

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

Uses your table/name constants so if you rename fields it still works.

### 3. Row count

We open a dedicated connection, execute the delete, commit, and read `cursor.rowcount` to see if anything was actually deleted.

### 4. Return value

- 0 → no rows matched that ID
- 1 → one ticket deleted

## Next steps

- Make sure your **business** layer calls `data_service.delete_ticket(ticket_id)` (not passing a dict).
- Restart your Flask app, and try deleting again. Now your manager will see `deleted_rows` and raise a 404 if it's zero, or return success otherwise.

This should resolve the "takes 1 positional argument but 2 were given" error and let your delete flow complete end-to-end.



Great job my friend! Task 2 is done. Now for the final step we need to do task 3. I will give you the instructions for it. First solve the business problem and then help me implement it in the code to fully finish the task: 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 than the base price of a standard ticket (1.0 multiplier).

You have the following formulas at your disposal:

1. Dynamic Price:
2. Dynamic Markup:

$$P_{\text{new}} = P_0 \times (1 + \text{markupdyn} + \text{markupfactor} \times \text{seat type})$$

$$\text{markupfactor} = c \times (1 - \frac{P_0}{P_{\text{max}}})$$

where:

- $P_0$  - Base price
- $\text{markupdyn}$  - Total markup
- $\text{seat type}$  - VIP/Standard
- $c$  - Max markup constant(%)

- $V_t$  - Current variable
- $V_0$  - Initial variable

First, let's work out the new prices step-by-step. We'll assume that the **current price** column in your **tickets** table is the "base price"  $P_0$ .

## 1. Compute the markups

We have two sources of markup:

1. **Time-based** (max 30 %):

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

2. **Availability-based** (max 20 %):

$$\text{markup}_{\text{avail}} = 0.20 \times \left(1 - \frac{\text{remaining}}{\text{initial}}\right)$$

Total dynamic markup

$$\text{markup}_{\text{dyn}} = \text{markup}_{\text{time}} + \text{markup}_{\text{avail}}$$

Finally,

$$P_{\text{new}} = P_0 \times (1 + \text{markup}_{\text{dyn}}) \times \text{seat\_multiplier}$$

where **seat\_multiplier** = 1.2 for VIP, 1.0 for Standard.

Ticket	days_left	remaining	initial	VIP?
A	3	100	500	Yes
B	10	1200-800=400	1200	No
C	15	200	1800	No

### 1. Ticket A

- $\text{markup}_{\text{time}} = 0.30 \times (1 - 3/30) = 0.30 \times 0.90 = \mathbf{0.27}$
- $\text{markup}_{\text{avail}} = 0.20 \times (1 - 100/500) = 0.20 \times 0.80 = \mathbf{0.16}$
- $\text{markup}_{\text{dyn}} = 0.27 + 0.16 = \mathbf{0.43}$
- **seat\_multiplier = 1.2**
- **total multiplier =  $(1 + 0.43) \times 1.2 = \mathbf{1.716}$**

### 2. Ticket B

- $\text{markup}_{\text{time}} = 0.30 \times (1 - 10/30) = 0.30 \times 0.6667 = \mathbf{0.20}$
- $\text{markup}_{\text{avail}} = 0.20 \times (1 - 400/1200) = 0.20 \times 0.6667 = \mathbf{0.1333}$
- $\text{markup}_{\text{dyn}} = 0.20 + 0.1333 = \mathbf{0.3333}$
- **seat\_multiplier = 1.0**

- total multiplier =  $(1 + 0.3333) \times 1 = \mathbf{1.3333}$

### 3. Ticket C

- markup\_time =  $0.30 \times (1 - 15/30) = 0.30 \times 0.5 = \mathbf{0.15}$
- markup\_avail =  $0.20 \times (1 - 200/1800) = 0.20 \times 0.8889 = \mathbf{0.1778}$
- markup\_dyn =  $0.15 + 0.1778 = \mathbf{0.3278}$
- seat\_multiplier = **1.0**
- total multiplier =  $(1 + 0.3278) \times 1 = \mathbf{1.3278}$

## 2. SQL UPDATE statements


Open your **DB Browser for SQLite**, switch to the **Execute SQL** tab, and run one statement per ticket. Adjust the **WHERE** clause to match however you identify those three rows (e.g. by **artist**). Here's an example using **artist = 'Artist A'**, etc., and rounding to two decimals:

```
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. Paste those three statements into the SQL editor.
2. Click **Execute SQL** (or the “play”  button).
3. Then click **Write Changes** (commit) so they persist.

### 3. Verify

- Switch back to **Browse Data**, select the **tickets** table, and confirm the **price** column for A, B, C now shows the updated values (rounded to 2 decimals).

You've now:

1. Correctly computed the dynamic markups and new prices.
2. Written and run the proper SQL **UPDATE** statements.

That completes Task 3. Great work!



what are the exact new prices calculated till the end

Since the new prices depend on each ticket's original base price  $P_0$  as stored in your **price** column, let's pull those values and compute the exact new prices in one go.

Assuming you're in the **Execute SQL** tab of DB Browser for SQLite, run:

```
sql

-- Show base price and compute new price for A, B, C
SELECT
  artist,
  price      AS base_price,
  ROUND(price * 1.716, 2) AS new_price_A,
  ROUND(price * 1.3333, 2) AS new_price_B,
  ROUND(price * 1.3278, 2) AS new_price_C
FROM tickets
WHERE artist IN ('Artist A', 'Artist B', 'Artist C');
```

That will give you a table like:

artist	base_price	new_price_A	new_price_B	new_price_C
Artist A	$P_{0A}$	$P_{0A} \times 1.716$	—	—
Artist B	$P_{0B}$	—	$P_{0B} \times 1.3333$	—
Artist C	$P_{0C}$	—	—	$P_{0C} \times 1.3278$

Then just read off the **new\_price\_...** column that corresponds to each artist. For example, if:

- Artist A's base price was **\$120**, you'll see  $\text{ROUND}(120 * 1.716, 2) \Rightarrow \text{\$205.92}$
- Artist B's base price was **\$80**, you'll see  $\text{ROUND}(80 * 1.3333, 2) \Rightarrow \text{\$106.67}$
- Artist C's base price was **\$50**, you'll see  $\text{ROUND}(50 * 1.3278, 2) \Rightarrow \text{\$66.39}$

(Substitute whatever your actual base prices are—those in the **base\_price** column.)

Once you've got those exact numbers, you can use the earlier **UPDATE** statements to write them back.



This are the initial prices :Artist C   Location C   50\$  
 Artist B   Location B   40\$  
 Artist A   Location A   30\$

Given your base prices:

- **Artist A** (VIP, \$30 base)

$$P_{\text{new}} = 30 \times 1.716 = 51.48$$

- **Artist B** (Standard, \$40 base)

$$P_{\text{new}} = 40 \times 1.3333 = 53.33$$



- **Artist C** (Standard, \$50 base)

$$P_{\text{new}} = 50 \times 1.3278 = 66.39$$

So your exact new prices (rounded to two decimal places) are:

Ticket	Base Price	New Price
Artist A	\$30.00	\$51.48
Artist B	\$40.00	\$53.33
Artist C	\$50.00	\$66.39

You can now run these SQL statements in your SQLite editor:

```
sql

UPDATE tickets
  SET price = 51.48
 WHERE artist = 'Artist A';

UPDATE tickets
  SET price = 53.33
 WHERE artist = 'Artist B';

UPDATE tickets
  SET price = 66.39
 WHERE artist = 'Artist C';
```

Remember to **Execute SQL** and then **Write Changes** (commit).



how does DB Browser for SQLite work? how can I do the task in it ?

**DB Browser for SQLite** is a free, open-source visual tool that lets you interact with SQLite database files through a graphical interface—no command-line SQL client needed. Here's how you'd use it to update those three ticket prices:

## 1. Download & launch

1. Go to <https://sqlitebrowser.org/> and download the installer for your OS (Windows/macOS/Linux).
2. Install and open **DB Browser for SQLite**.

## 2. Open your database file

1. Click **File** → **Open Database**.
2. Navigate to where your **database.db** (or whatever your SQLite file is named) lives in your project folder.
3. Select it and click **Open**.

## 3. Inspect the tickets table

1. Switch to the **Browse Data** tab.
2. In the drop-down at the top left, choose the **tickets** table.
3. You'll see all rows and columns—confirm you see *Artist A, B, C* with the old prices (30, 40, 50).

## 4. Execute your UPDATE statements


1. Click the **Execute SQL** tab.
2. In the large text box, paste the three **UPDATE** queries:

```
sql

UPDATE tickets
  SET price = 51.48
 WHERE artist = 'Artist A';

UPDATE tickets
  SET price = 53.33
 WHERE artist = 'Artist B';

UPDATE tickets
  SET price = 66.39
 WHERE artist = 'Artist C';
```

3. Click the  **Execute SQL** button (the green ►) or press **Ctrl+Enter**.
  - Below the editor, you'll see messages like "3 rows affected" (one for each statement).

## 5. Commit (write) your changes

1. **Important:** DB Browser uses a "transaction" model. Until you commit, your edits are only in memory.
2. Click **File** → **Write Changes** (or the floppy-disk icon).
3. You'll get a confirmation that the changes are saved to disk.

## 6. Verify the updates

1. Go back to **Browse Data** → **tickets**.
2. Check the **price** column for Artists A, B, C—you should now see 51.48, 53.33, 66.39.

### Quick tips

- **Revert:** If you make a mistake, before writing changes you can click **Revert Changes** to undo.
- **Backup:** It's good practice to make a copy of your **.db** file before running mass updates.
- **SQL History:** The SQL tab keeps a history of what you ran—handy to re-run or tweak your queries.

That's it! You've now used DB Browser for SQLite to update your ticket prices in a few clicks, without ever leaving a GUI.