Demo for fetching data from an API

This demo shows how to fetch data from a separate API and renders an HTML table.

The front end is created with the express template and only the views are used (with routes to render them).

This demo is done in two ways:

1. Using Fetch API and standard DOM manipulation
2. Using Ajax and jQuery

Backend

A simple API that returns event data, as well as options to force specific states using a query string for testing purposes.

1. No query/mode=success: Returns 200 with event data
2. mode=fail: Returns 400 with a message and status: 'fail' in the response body.
3. mode=error: Returns 500 with a message and status: 'error' in the response body.

An example of a successful request can be seen below:


```
{
"status": "success",
"data": [
        {
            "id": 1,
            "name": "Spring Festival",
            "date": "2025-06-01"
        },
        {
            "id": 2,
            "name": "Tech Conference",
            "date": "2025-06-10"
        },
        {
            "id": 3,
            "name": "Music Gig",
            "date": "2025-07-05"
        }
    ]
}
```


An example of a failed request can be seen below:


```
{
    "status": "fail",
```

```
    "message": "Bad request while fetching events."
}
```

Front end

As mentioned, this is generated using the express generator.

It has a single view to see all the events, and when the view is loaded there is a API call made to the backend to fetch the data from the separate backend.

Once this data has been fetched and no error state was returned, the table is populated. If there is an error state, a log is made in the console and an alert is made to the user.

Front end development

Once the project has been generated, we need to create a route and view for our events.

1. Create `views/events.ejs`
2. Create `routes/events.js` with a singe route to render the events view.
3. Add `event` route to `app.js`
4. Remove unused views (index)
5. Remove unused routes (index, users)

Once we have verified that the view is being rendered correctly, we can add some functionality.

The first step is to create an `events.js` script in our `public/javascripts` folder. We place it in public since that is added to our express app via `app.use(express.static(path.join(__dirname, 'public')));`

In this script we want to asynchronously load our event data when the document loads. There are several small different approaches to this, the approach we will take is to define a function (`getEvents`) and then call that when the DOM loads.

Note: There will be a separate `events-ajax.js` script created to show the same process but using jQuery and Ajax.

Here is a skeleton of our `public/javascritps/events.js` script:

```
async function getEvents() {
    try {
        // fetch logic goes here
        alert('hello');
    } catch (err) {
        // This catch block handles unexpected errors like network issues or code
bugs,
        // not the 'fail' or 'error' statuses returned inside a successful JSON
response.
        console.error('Error fetching events:', err);
    }
}
```

```
/*
    Remember: Async is for things that take time to execute. Browser events aren't
something that takes time.
    So, browsers don’t expect event listener callbacks (like DOMContentLoaded) to
be async,
    and there's no built-in way to handle promise rejections from them.
    If an async function inside a listener throws an error and it isn't caught, it
can fail silently.
    By keeping the listener synchronous and calling an async function inside it,
    we make error handling explicit and easier to control.
*/
document.addEventListener('DOMContentLoaded', () => {
    getEvents();
})
```

And here is our skeleton of `views/events.ejs`:

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Events</title>
</head>

<body>
    <h1>Hello</h1>

    <!-- Load script after body content -->
    <script src="/javascripts/events.js"></script>
</body>

</html>
```

Once we have verified that we see the alert when we run the application and navigate to our view, we can start fetching data from our backend.

We can make a call and check its response using the following code:

```
try {
    const res = await fetch('http://localhost:5000/events')
    // This is the actual http status, not the status property in our response
object.
    console.log(res.status);
}
```

If you get an error in the console that looks something like this:

```
Access to fetch at 'http://localhost:5000/events' from origin
'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-
Origin' header is present on the requested resource.
```

This means the backend has not enabled CORS (Cross origin resource sharing). Essentially, if a request comes from outside the origin (localhost:5000 in the case of our backend) AND is not something like postman (since they dont initiate a pre-flight check and dont trigger cors) it will be blocked. This is a server-protection mechanism built into all servers.

To fix this, you need to enable cors in the **backend API**, you can allow any origin (what we will do) or whitelist specific IPs.

First install the cors package:

```
npm install cors
```

Then add this line in `app.js`:

```
// app.js
const cors = require('cors');
...
app.use(cors());
...
```

After relaunching the backend, you should now be able to see the status code of 200 logged.

Now that we have confirmed we can ping the server and get a success, we can start extracting the data we need.

```
const res = await fetch('http://localhost:5000/events')
console.log(res.status);
// Extract the body by converting it to JSON
const json = await res.json()
console.log(json);
```

This will log the status of the response itself (200) and the content of the response object (`{status: 'success', data: Array(3)}`).

To handle our error states, we have two options. We can filter by `res.status.ok` (200) or our status property in our response body. Since we have the status of success/fail/error there, we will just use that.

```
async function getEvents() {
    try {
```

```
        const res = await fetch('http://localhost:5000/events?mode=success')

        // Extract the body by converting it to JSON
        const json = await res.json()

        if(json.status === 'success') {
            // Render table
            renderEventsTable(json.data)

        } else if(json.status === 'fail') {
            // Log that something failed
            console.error(`There was a failure: ${json.message}`)
        } else {
            // Log there was an error
            console.error(`An error occured: ${json.message}`)
        }

    } catch (err) {
        // This catch block handles unexpected errors like network issues or code
bugs,
        // not the 'fail' or 'error' statuses returned inside a successful JSON
response.
        console.error('Error fetching events:', err);
    }
}

function renderEventsTable(events) {
    console.log(events);
}
...
```

Even though the fetch API will automatically log when something has gone wrong, we can still write our own messages based on the error states. Now that we have the API call made and the relevant data extracted to a function to render the table.

To actually render this table, we need to have a container for it:

```
<!-- views/events.ejs -->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Events</title>
    <link rel="stylesheet" href="stylesheets/style.css">
</head>

<body>
    <table id="eventTable">
        <thead>
```

```
            <tr>
                <th>Name</th>
                <th>Date</th>
                <th>Action</th>
            </tr>
        </thead>
        <tbody></tbody>
    </table>

    <!-- Load script after body content -->
    <script src="/javascripts/events.js"></script>
</body>

</html>
```

Here we have added a table where our event data will be rendered. We give it an ID so we can go and fetch it using DOM manipulation. We also loaded a CSS file to give our table some padding and a border:

```
table, th, td {
  border: 1px solid black;
  border-collapse: collapse;
  padding: 5px;
}
```

Now the next step is to write our function to render the data, this involves several steps:

1. Fetch the element by ID, and then select the tbody element.
2. Clear the existing HTML to avoid re-rendering data
3. Loop through our events data and for each event do the following:
    1. Create a row element (`document.createElement('tr')`)
    2. Set its inner html to be 3 td elements (one for each property and one for our action)
    3. Create a handler for the click event of the action (`onclick="handleAction(${event.id})`)
    4. Append the `tr` as a child to the `tbody` element
4. Create a `handleAction(eventId)` function to handle our click. Note: We dont use EJS syntax (`<%= event.id %>`) to pass the events Id since the data did not come from EJS, we fetched it separately.

This can be seen in the snippet below:

```
function renderEventsTable(events) {
    // Get the body of the table to add rows
    // https://www.w3schools.com/jsref/met_document_queryselector.asp
    // https://www.w3schools.com/cssref/css_selectors.php
    const tableBody = document.querySelector('#eventTable tbody');
    tableBody.innerHTML = ''; // Clear existing rows

    events.forEach(event => {
        const row = document.createElement('tr');
```

```
        row.innerHTML = `
            <td>${event.name}</td>
            <td>${event.date}</td>
            <td>
                <button onclick="handleAction(${event.id})">Action</button>
            </td>
        `;

        tableBody.appendChild(row);
    });
}

function handleAction(eventId) {
    alert(`You clicked on event ID: ${eventId}`)
}
```

At this point, you should have a rendered table showing event data and a button that alerts which event you have clicked on.

jQuery version

There is also a version of this using jQuery and Ajax, just switch the imports from:

```
<!-- Load script after body content -->
<!-- <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="/javascripts/events-ajax.js"></script> -->
<script src="/javascripts/events.js"></script>
```

To:

```
<!-- Load script after body content -->
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script src="/javascripts/events-ajax.js"></script>
<!-- <script src="/javascripts/events.js"></script>  -->
```