Tutorial #3 – Create.. Update.. Delete Inventory

In this tutorial, you will learn how to add, update and delete items, using the Node.js, Express, EJS platform.

Prelude

In your CRUD project, you are building tutorial 3 on top of tutorial 2, which was build on tutorial 1. Just in case you are starting a new project each time, make sure that you are using the correct imports and middleware to support the HTTP requests and the route handler request and response objects. Here is what you should be importing and bringing in as middleware via app.use, in your app.js server file, following.

```
// express is used to create the server
import express from "express";

// body-parser is middleware for Node.js, commonly used with Express.js, to parse incoming request bodies before handlers process them. Makes request data available from req.body.
import bodyParser from 'body-parser';

// imports needed to resolve ES6 file and directory references
import path from "path";
import { fileURLToPath } from "url";

// import to include  method-override middleware, to use HTTP methods like DELETE
import methodOverride from 'method-override';

//import needed to establish and access the database
import { setupDatabase, getDbConnection } from './database.js';

// Set up server and port
const app = express();
const port = 3000;

// Resolve path format and name of current directory
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Middleware is utility code that can access the request object and the response object
//
// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: true }));

// Middleware to serve files
app.use(express.static('public'));
// ... make the public folder the default one for express, or
app.use(express.static(__dirname + "/public"));

// Middleware to
// ...parse incoming requests with JSON payloads, based on body-parser. Makes parsed data available in req.body
app.use(express.json());
// ... parse incoming requests with URL-encoded payloads (like form submit), also based on body-parser.
// extended: true option lets objects and arrays to be encoded into the URL-encoded format so they can be passed.
app.use(express.urlencoded({ extended: true }));
// ... allows use of HTTP verbs PUT, DELETE. Looks for a _method query parameter in the request, overrides HTTP method so routing parameters (like meetingID) get to route handler.
app.use(methodOverride('_method'));


// Set up the database
setupDatabase().catch(console.error);

app.set("view engine", "ejs");
```

... and here is a more tenable version!

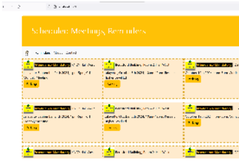
```
// express is used to create the server
import express from "express";

// body-parser is middleware for Node.js, commonly used with Express.js, to parse incoming request bodies before handlers process them. Makes request data available from req.body.
import bodyParser from 'body-parser';
```

```
// imports needed to resolve ES6 file and directory references
import path from "path";
import { fileURLToPath } from "url";

// import to include  method-override middleware, to use HTTP methods like DELETE
import methodOverride from 'method-override';

//import needed to establish and access the database
import { setupDatabase, getDbConnection } from './database.js';

// Set up server and port
const app = express();
const port = 3000;

// Resolve path format and name of current directory
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

// Middleware is utility code that can access the request object and the response object
//
// Middleware to parse request bodies
app.use(bodyParser.urlencoded({ extended: true }));

// Middleware to serve files
app.use(express.static('public'));
// ... make the public folder the default one for express, or
app.use(express.static(__dirname + "/public"));

// Middleware to
// ...parse incoming requests with JSON payloads, based on body-parser. Makes parsed data available in req.body
app.use(express.json());
// ... parse incoming requests with URL-encoded payloads (like form submit), also based on body-parser.
// extended: true option lets objects and arrays to be encoded into the URL-encoded format so they can be passed.
app.use(express.urlencoded({ extended: true }));
// ... allows use of HTTP verbs PUT, DELETE. Looks for a _method query parameter in the request, overrides HTTP
method so routing parameters (like meetingID) get to route handler.
app.use(methodOverride('_method'));

// Set up the database
setupDatabase().catch(console.error);

// Specify EJS for the views
app.set("view engine", "ejs");
```

Make sure to check your 'setup' code in app.js, for these imports and associations.

Finalizing CRUD – Create, Update, Delete

Over the past 2 tutorials, you have seen how to



- ***Create a Node.js/Express application, listening for requests on localhost***
- ***Add routes to the server, to correspond to specific pages (like home, about, contact)***
- ***Interface with the SQLite database***
- ***Retrieve data from a SQLite database, and present this information***
- ***Add local styling to the content***
- ***Integrate the Bootstrap library to elevate the UI***

A company typically has a requirement to ***list*** products and to ***manage*** inventory, and that could be services as well as tangible items like coffee or clothing. Business applications revolve around representing and manipulating this data. Can you think of any business that does not concern itself with managing data?

Data represents the company's offerings, the reason that the business is in operation. Businesses are dynamic - new products or services are added, existing products or services might be removed or modified.

You have an application that reads data from a database, but this data has been seeded by the application – it does not represent a product or service that has been added, because the ability to extend product inventory has not been implemented. You do not have the ability to delete or remove an item that is no longer a part of the business offering. You cannot update an item to reflect new considerations.

In your job as a software developer, you may be tasked with the creation of a ***CRUD*** application. A CRUD application is a key building block for software engineers, and concerns data management.

CRUD is an acronym for

Create – the process of adding a product or service

Read – getting the details about a product or service

Update – modifying the details of a product or service

Delete – removing a product or service from the company's offerings

So far, you have the Read covered. Now let's tackle the other 3!

CRUD Tutorial #3, Adding Items To Inventory, Maintaining Inventory

The menu bar has a typical Home – About – Contact set of options. The About is not playing a major role, as this is a simple application that lists meetings and the header on the home page states just that.

Let's hi-jack the about page and turn it into an administration panel.

Why would you need an administration panel?

Authorization to view meetings might be a general permission, but the authorization required to make changes to a company's data should not be something that anyone can have. By placing access to higher authority operations (like changing data) in a specific view, you can add permissions to that view alone, or remove the path to this view from the menu bar, if the user does not have the required access authority.

How would you design a panel for administration of the data entities, the meetings?

This panel has responsibility for providing a way to ***Create***, ***Update*** and ***Delete***. The design of this view is open to interpretation, and there are many designs possible. In this tutorial, we will use an approach that allows the user to select items to modify (delete or update) on one side, and allows the user to add an item on the other side. What do you think about this design? Would you do anything different?

You might use a view, termed Administration (admin.ejs) to represent the view that enables Create, Update and Delete, as above.

```
<%- include("../partials/head") %>
<body>
  <div class="container">
     <div class="mt-4 p-5 bg-warning text-white rounded">
       <h1><%= title %></h1>
    </div>
   <%- include("../partials/menu") %>

  <p>
    Meeting Administration.
  </p>

  <div class="container">
    <div class="row">
    <div class="container col-sm-12 col-md-4 col-lg-5">
     <h1>Meetings</h1>
      <ul class="list-group">
        …..
      </ul>
  </div>

  <div class="container  col-sm-12 col-md-4 col-lg-5">
    <h2>Add a New Meeting</h2>
    <form action="/add_meeting" method="POST" id="meetingForm">
         …            …..
    </form>
  </div>
  </div>
  </div>
  <%- include("../partials/footer") %>
</body>
</html>
```

The route for this can be added to app.js as follows. You can pass in a message, or other information, like any notifications, titles etc.

```
app.get("/admin", async (req, res) => {
  try {
    const db = await getDbConnection();
    const rows = await db.all('SELECT * FROM meetings');
    res.render("pages/admin", { data: rows, title: "Administer Meetings", notification:
true, message: "--------" });
  }
  catch (err) {
```

```
    console.error(err);
    res.status(404).send('An error occurred while getting the data to manage');
  }
});
```

Here, the list of meetings can be displayed, with prompts (buttons, links) for modification, and a panel for adding a new item.

localhost:3000/admin

Administer Meetings

Reminders Administration Contact

Meeting Administration.

Meetings

Topic: Team Building Event
Date & Time: 2024-09-20T14:00:00
Location: Outdoor Park
Delete
Make Changes!

Topic: Client Presentation
Date & Time: 2024-09-25T09:00:00
Location: Main Auditorium
Delete
Make Changes!

Topic: Purdue Marketing
Date & Time: Oct 12th at 3pm
Location: WL
Delete
Make Changes!

Add a New Meeting

Topic

Mandatory

Date & Time

Location

Parking

Add Meeting

Purdue Pete Reminders, Tutorials

So, this should be linked and referenced from the menu, menu.ejs,, as we now have a new plan for an administration view.

```
<nav class="navbar navbar-expand-lg bg-light">
  …….
    <div class="collapse navbar-collapse" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item">
          <a class="nav-link active" aria-current="page" href="/">Reminders</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/admin">Administration</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/contact">Contact</a>
```

```
        </li>
…….
```

Test, make sure that you can open the administration page, before proceeding.

Now, let's add the form part, for collecting the data for the new item to be added to the database. This form is in admin.ejs, and corresponds to the ***/add_meeting*** route, as this is the route that is rendered when the form is submitted.

```
  <form action="/add_meeting" method="POST" id="meetingForm">
      <div class="form-group">
        <label for="topic">Topic</label>
        <input type="text" class="form-control" id="topic" name="topic" required>
      </div>
      <div class="form-group form-check">
        <label for="mandatory">Mandatory</label>
        <input type="checkbox" class="form-control form-check-input" id="mandatory"
name="mandatory">
      </div>

      <div class="form-group">
        <label for="dateTime">Date & Time</label>
        <input type="text" class="form-control" id="dateTime" name="dateTime" required>
      </div>
      <div class="form-group">
        <label for="location">Location</label>
        <input type="text" class="fom-control" id="location" name="location" required>
      </div>
      <div class="form-group">
        <label for="parking">Parking</label>
        <input type="text" class="form-control" id="parking" name="parking" required>
      </div>
       <hr>
       <button type="submit" class="btn btn-primary">Add Meeting</button>

     </form>
```

Let's finish up by adding the code for listing the meetings, with the options for delete and modify. The meetings are rendered as a list of items, each of which has a delete button and an edit link for

update. Why not two buttons? You can implement the edit action as a button, if you want. Both of these possibilities are here to demonstrate the possibilities.

Important!!! The code for delete and modify needs to target ***a specific item***, so we use special parameters, ***route parameters***, to pass this information to the route handler.

This code,

```
<form action="/delete/<%= meeting.id %>?_method=DELETE" method="POST">
```

and

```
<a class="nav-link" href="/edit/<%= meeting.id %>">Make Changes!</a>
```

… passes the meeting id to the route handler in the case of delete, and to the view in the case of modify, so that the required object can be targeted. Remember, the meeting id here is the value that uniquely identifies the item in the database table.

```
<h1>Meetings</h1>
   <ul class="list-group">
     <% data.forEach((meeting, index) => { %>
       <li class="list-group-item" id="<%= index %>">
         <strong>Topic:</strong> <%= meeting.topic %><br>
         <strong>Date & Time:</strong> <%= meeting.dateTime %><br>
         <strong>Location:</strong> <%= meeting.location %><br>
         <form action="/delete/<%= meeting.id %>?_method=DELETE" method="POST">
           <input type="hidden" name="_method" value="DELETE">
           <button type="submit">Delete</button>
       </form>
       <a class="nav-link" href="/edit/<%= meeting.id %>">Make Changes!</a>
       </li>
     <% }) %>
   </ul>
```

We have kept the sequential index value, in the forEach. You can use the index as a complement to the items to incorporate organization, or you can remove it.

The important thing to remember for delete or modify is that the highlighted code uses route parameters, these are values passed to the route handler. In this case, the route parameters indicate the item id of the item to be deleted or modified.

Create, Add A Meeting

What is necessary in order to add a new meeting obligation, or add an item to the company's inventory?

Assume the use of a database.

We know what form this operation should take. Anything involving manipulating company data should be on a view that is accessed only by an administrator. As stated, a common approach is to display items in one view, and use an authority object to grant access to any modification views – like a view that enables adding a new item, deleting an item, updating an item. That way the panel can be accessed only by an admin type role. That way, the admin role can choose from operations to complete in one place. The design plan is often created by user requirements and UX/UI designers.

While we are not integrating authority or permissions now, you should recognize that keeping this control on one view is a plus for adding access checks.

A view responsible for collecting data to be added as a new item might include a form with fields corresponding to that new item's attribute. The meeting object attributes include Topic, whether or not the meeting is Mandatory, Date & Time, Location, Parking.

Hopefully, you have implemented this in the administration view.

Step 1 : Add the route(s)

The first route is the view that allows for the data to be changed, as per access permissions. We will not implement any such access permissions here, but hopefully you can see that an authentication component can be added, and access implemented according to company policy.

Given a new page, admin - Administration,

```
app.get("/admin", async (req, res) => {
......
});
```

Step 2: Design and add a view for administration.

Add a form or other mechanism for getting information about the new item, to the view. Forms would be a typical way to request more than a single piece of information, and are the standard way of getting data to be added to a repository (like a database).

Add a New Meeting

Topic

Mandatory

Date & Time

Location

Parking

Add Meeting

A form action attribute, and method,

```
<form action="/add_meeting" method="POST" id="meetingForm">
```

have a direct relationship with the submit button in the form,

```
<button type="submit" class="btn btn-primary">Add Meeting</button>
```

The combination above results in control being passed to the add_meeting route, as a POST request. Do you see how?

```
<form action="/add_meeting" method="POST" id="meetingForm">
   <div class="form-group">
     <label for="topic">Topic</label>
     <input type="text" class="form-control" id="topic" name="topic" required>
   </div>
   <div class="form-group form-check">
     <label for="mandatory">Mandatory</label>
     <input type="checkbox" class="form-check-input" id="mandatory" name="mandatory">
```

```
      </div>
      <div class="form-group">
        <label for="dateTime">Date & Time</label>
        <input type="text" class="form-control" id="dateTime" name="dateTime" required>
      </div>
      <div class="form-group">
        <label for="location">Location</label>
        <input type="text" class="fom-control" id="location" name="location" required>
      </div>
      <div class="form-group">
        <label for="parking">Parking</labelr>
        <input type="text" class="form-control" id="parking" name="parking" required>
      </div>
      <hr>
      <button type="submit" class="btn btn-primary">Add Meeting</button>
    </form>
```

The other elements in the form correspond to formatting (labels) and getting input (form fields, input).

As part of forms processing, when the submit button is clicked, the action taken is to route to /add_meeting. This request is sent with the form's data, which can be retrieved from the request object, the ***req*** parameter. In the route handler, ***req.bod***y contains all of the information from the form – this is accessed so that the correct values can be used in the database insert call.

Step 3: Create a POST request process and a route for inserting the new item

Once submit is clicked on the form, a POST request to route \add_item is generated.

```
app.post('/add_meeting', async (req, res) => {
 …..
});
```

What happens here?

Well, inside this route handler, try doing

console.log(req.body)

The data from the form has been sent with the request header, to the /add_meeting route, and this is what you should see as a result of the console.log.

So, when this data is routed to \add_item, the data is available and ready to be inserted into the database, which brings you to the next step.

Step 4: Link to the database, as you will need to add a new item to the table

Find the database reference link …

```
try {
    const db = await getDbConnection();
  ….
}
catch (err) {
  console.error(err);
  res.status(404).send('An error occurred while accessing the data');
}
```

Step 5: Get the data from the request object body

From the console.log in the last step, you saw that the form data (for the new item) has been passed to the route handler, as an object. We need the values in this object, individually, so that we can use them in the SQL INSERT command.

Let's talk about destructuring – this is a technique for assigning values from object attributes. Instead of assigning each attribute individually from the object, as in

```
const topic = req.body.topic;
```

… the object can be destructured, to neatly assign en masse,

```
const { topic, mandatory, dateTime, location, parking } = req.body;
```

That way, all of the values are transferred in a single (not too difficult to understand, hopefully) line of code.

The only issue is the format for the mandatory field, as this is a Boolean value, which is stored in the database as a 1 or a 0. Use of some well-placed logic, like an if or a ternary operator, makes the conversion so that the database INSERT is successful.

```
app.post('/add_meeting', async (req, res) => {
  console.log(req.body);
  const { topic, mandatory, dateTime, location, parking } = req.body;
```

```
  let is_mandatory = req.body.mandatory ? 1 : 0;
```

Step 6: Add the item to the database

Now, we can add of the fields, as an entity, to the database table.

```
app.post('/add_meeting', async (req, res) => {
  const { topic, mandatory, dateTime, location, parking } = req.body;
  let is_mandatory = req.body.mandatory ? 1 : 0;
  try {
    const db = await getDbConnection();
    await db.run('INSERT INTO meetings (topic, mandatory, dateTime, location, parking)
VALUES (?, ?, ?, ?, ?)', [topic, is_mandatory, dateTime, location, parking]);
    });
```

Now what?

Typically, users should be given an indication that the operation was successful. This can be handled with a message, or by redirecting to another page, like the home page, showing that the new item is now listed as a meeting. You can add a message too, but proving that the item has been added by revealing the new list of items, is considered best practice.

Step 7: List the updated inventory to show that the item has been added

You *should* do this by default, to give instant feedback to the user. This reinforces the addition more than a message stating 'Add Successful'. The typical response here is a return to the home page via res.redirect, as opposed to opening up a new route. If you use res.render here instead of res.redirect, you will be brought to pages/index...add-item.

The call to redirect will return to the home page, as this is what is specified as the route. Any failures should use 500, Internal server Error.

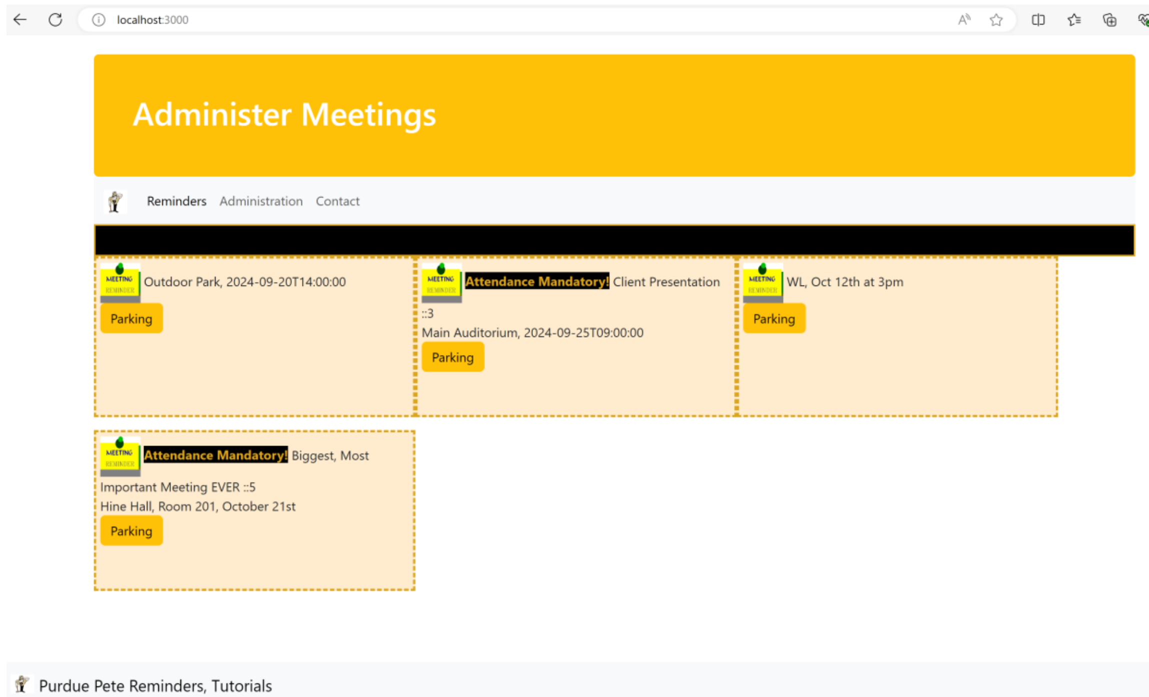
```
// Route to handle form submission
app.post('/add_meeting', async (req, res) => {
  const { topic, mandatory, dateTime, location, parking } = req.body;
  let is_mandatory = req.body.mandatory ? 1 : 0;
console.log(`IN ADD MTG - topic ${topic}, mandatory ${is_mandatory}, dateTime ${dateTime},
location ${location}, parking ${parking}`);
  try {
    const db = await getDbConnection();
    await db.run('INSERT INTO meetings (topic, mandatory, dateTime, location, parking)
VALUES (?, ?, ?, ?, ?)', [topic, is_mandatory, dateTime, location, parking]);
```

```
    // redirect to home route
    res.redirect('/'); // Redirect back to home route
  } catch (err) {
    console.error(err);
    res.status(500).send('An error occurred while submitting the form');
  }
});
```

Often, the following status codes are used to indicate success or failure.

- **201 Created**: If the item is successfully added, the server responds with a 201 status code, indicating that a new resource has been created.
- **500 Internal Server Error**: If there's an error during the process, the server responds with a 500 status code.

When the Add Meeting button is clicked, you should see the home page, with all of the meetings listed. Check the end – this is the default position for the new item, as you have added it to the database table as the last entry. You can change this – reverse the list, add the item to a special notification area, or other.



Update

What is necessary in order to modify the details of an item?

You should add a link or button, to indicate that the user can modify an item. Hopefully you did this already in the first part of this tutorial.

The biggest nuance here is that you need to target a specific item to update. How have we targeted specific items? If we are dealing with an array or collection, we use an index. If we are dealing with a database, we use the id of the item as assigned by the database management system.

The general rule is to decide on the mechanism to identify the item to be modified. This item will have to be retrieved (a get request), so that the current values can be displayed to the user, prior to modification. The updates will then be saved to the database (a post request). This means that there should be 2 route handlers – one for the get request and another for the post request.

The alternative is to use the current object, as in the object at the index of the item to be modified - but this is risky in terms of data integrity as it would require a mapping between the index and the object id. Many others have access to the database, and may potentially have updated this record. It is best to retrieve current information.

Step 1 : Add the route

This step encompasses the need to identify the item, and uses route parameters.

There is a special syntax for conveying these route parameters, from the form (and it is typical that a form be used to collect updated information about the item to be modified). Let's examine the get request first, use the proposed form, and then examine the post request.

```
// Edit route
app.get('/edit/:id', async (req, res) => {
  …
});
```

Here, the syntax identifies route parameters, or at least a single route parameter, the id of the item being modified. This id is passed to the view containing the form, the form that facilitates the modification.

Step 2: Add a form or other mechanism for the update

```
<%- include('../partials/head.ejs') %>

<body>

<div class="mt-4 p-5 bg-warning text-white rounded">
    <h1><%= title %></h1>
  </div>
<form action="/edit/<%= data.id %>" method="POST">
    <div class="form-group">
        <label for="topic">Topic:</label>
        <input type="text" id="topic" name="topic" value=<%= data.topic %>>
    </div>
    <div class="form-group form-check">
        <label for="mandatory">Mandatory</label>
        <input type="checkbox" class="form-control form-check-input" id="mandatory"
name="mandatory">
    </div>
    <div class="form-group">
        <label for="datetime">Date and Time:</label>
        <textarea id="datetime" name="datetime"><%= data.datetime %></textarea>
    </div>
    <div class="form-group">
        <label for="location">Location:</label>
        <textarea id="location" name="location"><%= data.location %></textarea>
    </div>
    <div class="form-group">
        <label for="parking">Parking:</label>
        <textarea id="parking" name="parking"><%= data.parking %></textarea>
```

```
    </div>
    <button type="submit">Save</button>
</form>
</body>
```

This should be very similar to the add form.

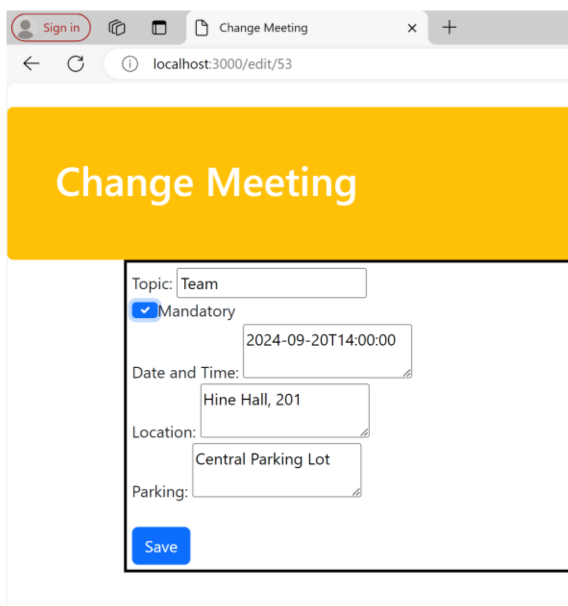
Step 3: Link to the database

Step 4: Get the current data for the item to be modified

Step 5: Open the edit view, with the data from the database, and any other information required.

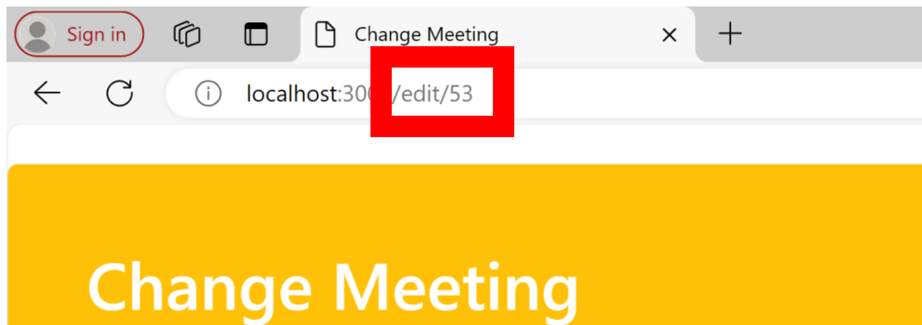
```
// Edit route
app.get('/edit/:id', async (req, res) => {
  const db = await getDbConnection();
  const sql = `SELECT * FROM meetings WHERE id = ?`;
  const row = await db.get(sql, req.params.id);
  res.render('pages/edit', { data: row, title: "Change Meeting", notification: true, message: "Meeting being modified" });
});
```

You might see something like this.



CRUD Tutorial #3, Adding Items To Inventory, Maintaining Inventory

Note the route parameter in the url – this uniquely identifies the meeting, it is the meeting id.



An edit view with such a form should hold the fields/labels that represent the database entity – the meeting. You can include this edit.ejs in your pages folder.

```
<%- include('../partials/head.ejs') %>
<body>
<div class="mt-4 p-5 bg-warning text-white rounded">
    <h1><%= title %></h1>
  </div>
  <div class="container">
<form action="/edit/<%= data.id %>" method="POST" style="border: 3px solid black;
padding:5px;">
    <div class="form-group">
        <label for="topic">Topic:</label>
        <input type="text" id="topic" name="topic" value=<%= data.topic %>>
    </div>
    <div class="form-group form-check">
        <label for="mandatory">Mandatory</label>
        <input type="checkbox" class="form-control form-check-input" id="mandatory"
name="mandatory">
    </div>
    <div class="form-group">
        <label for="datetime">Date and Time:</label>
        <textarea id="datetime" name="datetime"><%= data.dateTime %></textarea>
    </div>
    <div class="form-group">
        <label for="location">Location:</label>
        <textarea id="location" name="location"><%= data.location %></textarea>
    </div>
    <div class="form-group">
        <label for="parking">Parking:</label>
        <textarea id="parking" name="parking"><%= data.parking %></textarea>
```

```
      </div>
      <br>
    <button type="submit" class="btn btn-primary">Save</button>
</form>
</div>
</body>
```

Step 4: Add a route for updating

Step 5: Implement an UPDATE operation with the new item

```
app.post('/edit/:id', async (req, res) => {
  const db = await getDbConnection();
  let { topic, mandatory, datetime, location, parking } = req.body;
  let is_mandatory = mandatory == undefined ? 0 : 1;
  const sql = `UPDATE meetings SET topic = ?, mandatory = ?, dateTime = ?, location = ?,
parking = ? WHERE id = ?`;
  await db.run(sql, [topic, is_mandatory, datetime, location, parking, req.params.id]);
});
```

Step 5: List the updated inventory to show that the item has been modified, redirect to the home page

```
app.post('/edit/:id', async (req, res) => {
  const db = await getDbConnection();
  let { topic, mandatory, datetime, location, parking } = req.body;
  let is_mandatory = mandatory == undefined ? 0 : 1;
  const sql = `UPDATE meetings SET topic = ?, mandatory = ?, dateTime = ?, location = ?,
parking = ? WHERE id = ?`;
  await db.run(sql, [topic, is_mandatory, datetime, location, parking, req.params.id]);
  res.redirect('/'); // Redirect back to home route
});
```

Delete

What is necessary in order to delete an item?

Well, apart from the obvious processes – target the specific item, and remove, there is an issue that we have to fix, concerning the parking. Spend a few minutes thinking about this. We used dummy

data in an array, when we built the parking button into the app. Without delete functionality, the index of the item in the array worked just fine to target the meeting object. However, now ..... there is an issue. What do you think this issue is?

(hint – removing an item causes shifts in the index we have used to access it)

Let's go through the steps to add the delete functionality, and address the parking modal after this.

Make sure to include in your server file (like app.js here)

// import to include method-override middleware, to use HTTP methods like DELETE
import methodOverride from 'method-override';

... and

// ... allows use of HTTP verbs PUT, DELETE. Looks for a _method query parameter in the request, overrides HTTP method so routing parameters (like meetingID) get to route handler.
app.use(methodOverride('_method'));

Step 1 : Add the route

We added a delete button to each meeting listed in the admin view.

Topic: Curriculum Development
Date & Time: October 11th
Location: Conference Room A
Delete
Make Changes!

```
<form action="/delete/<%= meeting.id %>?_method=DELETE" method="POST">
    <input type="hidden" name="_method" value="DELETE">
    <button type="submit">Delete</button>
</form>
```

So, the delete route should be declared as /delete/, with the meeting id appended, so as to pass this to the route handler as a route parameter.

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

Step 2: Use the route parameter to target the specific meeting to delete

Step 3: Link to the database

Step 4: Implement a DELETE operation to remove the item

```
app.delete('/delete/:id', async (req, res) => {
  try {
      const db = await getDbConnection();
      await db.run('DELETE FROM meetings WHERE id = ?', req.params.id);
  } catch (error) {
      console.error(error);
      res.status(500).send('Error deleting item');
  }
});
```

Step 5: List the updated inventory to show that the item has been removed

```
app.delete('/delete/:id', async (req, res) => {
  try {
      const db = await getDbConnection();
      await db.run('DELETE FROM meetings WHERE id = ?', req.params.id);
      res.redirect('/'); // Redirect back to home route
  } catch (error) {
      console.error(error);
      res.status(500).send('Error deleting item');
  }
});
```

If you experiment with delete, and then open the parking modal, you may not get what you might expect – as we can no longer rely on the index of the data array. Instead, the procedure in place for the parking modal has to change.

We can either pull the meeting from the database again, with the meeting id, or we can pass the meeting object from the button to the modal. The latter is the technique used here.

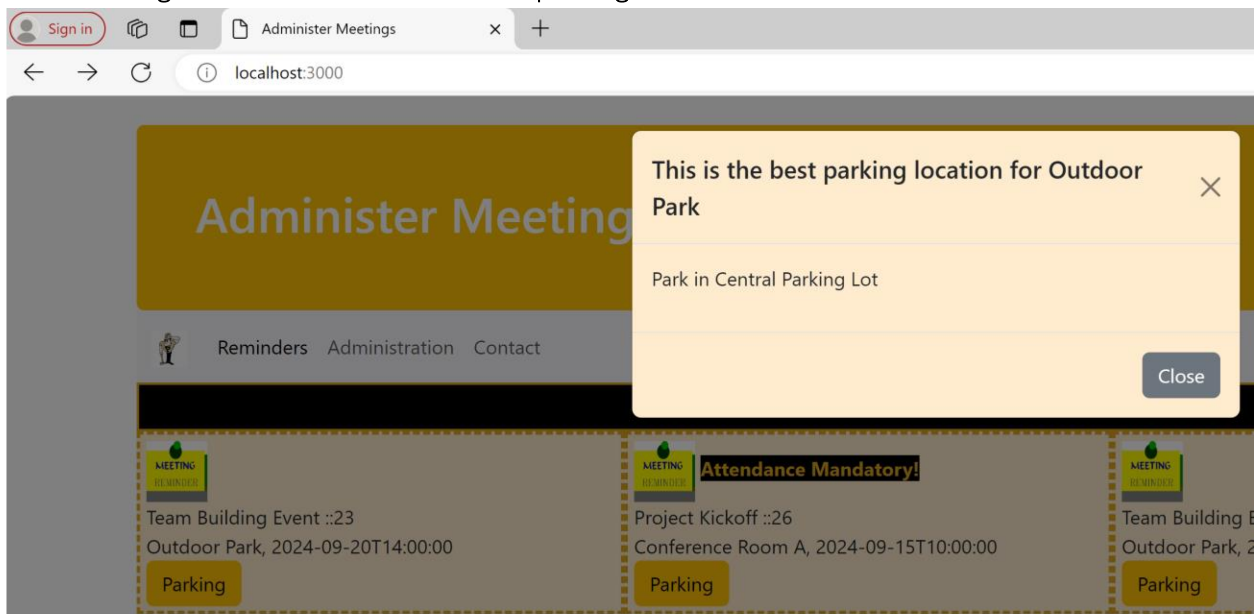
Change the button markup in index.ejs. Serialize (means change to string format) the meeting object first so that it can be reliably retrieved from the button object in the click handler.

```
<!-- Button trigger modal -->
 <% let meetingAsStr=JSON.stringify(meeting); %>
<button type="button" class="parking btn btn-warning" data-bs-toggle="modal"
data-bs-target="#parkingModal" id = "<%=meeting.id %>" data-meeting = "<%=
meetingAsStr %>">
  Parking
</button>
```

Now parse the meeting into a JavaScript object, so that the attributes can be accessed,

```
<script>
   let allParkingButtons = document.querySelectorAll('.parking');
   allParkingButtons.forEach((parkingButton)=>
{parkingButton.addEventListener('click', function(event) {
            const parkingButton = event.target;
            // Access to an object from the button requires .getAttribute!
            let meetingData = parkingButton.getAttribute('data-meeting');
            // The object got serialized for the button, so now convert to object
            let meetingObj = JSON.parse(meetingData);

            let parking_info_ref = document.querySelector('#parking_info');
            parking_info_ref.textContent = "Park in " + meetingObj.parking;
            let parking_modal_title_ref =
document.querySelector("#parking_for_meeting");
            parking_modal_title_ref.textContent = "This is the best parking
location for " + meetingObj.location;
        })});
</script>
```

This should give the correct details for the parking.



Congrats on your full-stack, *rounded* CRUD application!