Lab Assignment 2: Building a Modern Client-Side Kanban Board

TECNOLOGIA SETÚBAL
SETÚBAL POLYTECHNIC UNIVERSITY

Course: Programação Avançada para a Internet (Mestrado em Engenharia de Software)

Project Overview

You will build a "Kanban Board" web application to manage tasks. The application will be built entirely with client-side technologies (HTML, CSS, JavaScript) and will not require a backend server. All data will be saved in the browser's `localStorage`. This project will take you from the fundamentals of modern JavaScript to building a complete Progressive Web App (PWA).

Prerequisites

- Basic knowledge of HTML and CSS.
- Familiarity with fundamental JavaScript concepts (variables, functions, objects, arrays).
- A modern web browser (Chrome, Firefox, Edge).
- A local web server for testing (required for Levels 4+). You can use the `live-server` VS Code extension or use the Node.js `serve` package in your project folder.

Level 1: The Foundation - Page Structure and Modern JS

Objective: Create the basic structure of the application and use modern JavaScript syntax and modules to render a static list of tasks.

Key Topics: `class`, ES Modules (`import`/`export`), Arrow Functions, `let`/`const`.

Instructions:

1. **File Structure:**
    - `index.html`
    - `styles/main.css`
    - `scripts/main.js`
    - `scripts/Task.js`
2. **HTML (`index.html`):**
    - Create a basic HTML structure.
    - Inside the `<body>`, create a main container for the board.

   - Inside the container, create three columns with `id`s: `todo-column`, `inprogress-column`, and `done-column`. Give them titles like "To Do", "In Progress", and "Done".
   - Link your `main.css` in the `<head>`.
   - Link your `main.js` at the bottom of the `<body>` using `<script type="module" src="scripts/main.js"></script>`. Using `type="module"` is crucial.

3. **CSS (`css/main.css`):**

   - Add some basic styling to make the columns appear side-by-side (e.g., using Flexbox or Grid, or you may also use Bootstrap or Tailwind CSS).
   - Style the task cards to be visually distinct.

4. **Task Class (`scripts/Task.js`):**

   - Create a `Task` class.
   - The `constructor` should accept `id`, `text`, `status` (`todo`, `inprogress`, `done`), and optional `location = null` (for geolocation data as `{ lat, lng }`).
   - Add a getter `displayText` that returns the task text, appending location coordinates if available (e.g., `${this.text} (Lat: ${this.location.lat}, Lng: ${this.location.lng})`).
   - Add an instance method `toJSON()` that returns a plain object for JSON serialization (including all properties).
   - Add a static method `fromJSON(data)` that recreates a `Task` instance from a plain JSON object.
   - Use `export` to make this class available to other modules: `export class Task { ... }`.

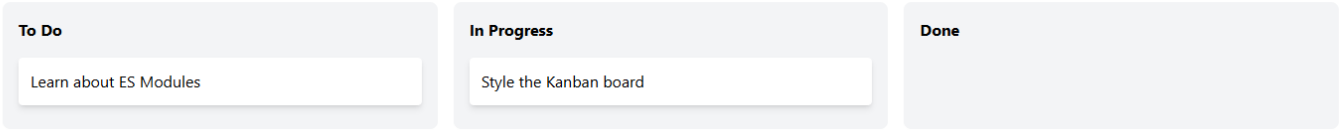
5. **Main Logic (`main.js`):**

   - `import` the `Task` class from `./scripts/Task.js`.
   - Create a static array of `Task` objects for testing.

     ```
     const tasks = [
       new Task(1, 'Learn about ES Modules', 'todo'),
       new Task(2, 'Style the Kanban board', 'inprogress'),
     ];
     ```

   - Create a function `renderTasks()` that:
     - Clears the content of all columns.
     - Loops through your `tasks` array.
     - For each task, creates an HTML element (a `div`) representing the task card.
     - Appends the card to the correct column based on its `status`.
   - Call `renderTasks()` when the script loads.

Goal: At the end of this level, you should see your static tasks rendered in the correct columns on the page.



To Do

Learn about ES Modules

In Progress

Style the Kanban board

Done

Level 2: Persistence - Adding and Saving Tasks

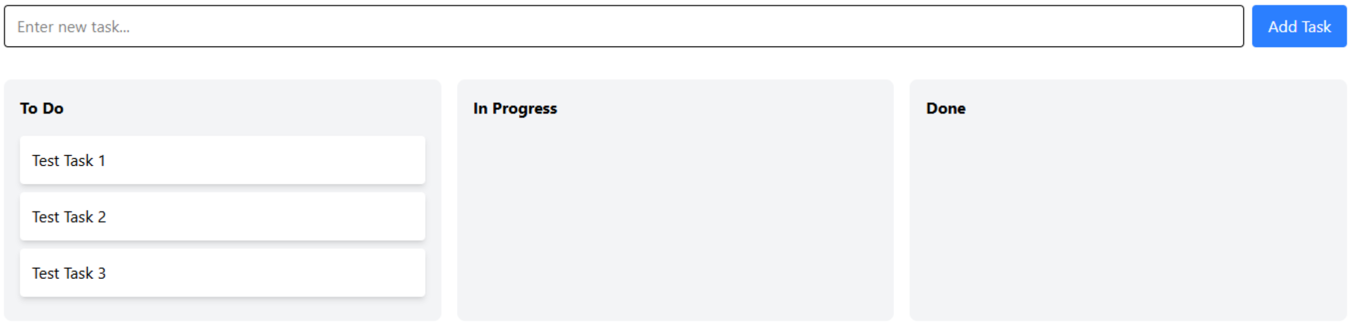
Objective: Allow users to add new tasks and persist them in the browser using the Web Storage API.

Key Topics: `localStorage`, DOM Manipulation, `JSON.stringify`, `JSON.parse`.

Instructions:

1. **HTML (`index.html`):**
    - Add a simple form above the board with a text input for the new task and an "Add Task" button.
2. **Main Logic (`main.js`):**
    - **Create a `tasks` array:** This will hold the state of your application. Initialize it from `localStorage` or with an empty array if storage is empty.
    - **`saveTasks()` function:** Create a function that takes the current `tasks` array, converts it to a JSON string using `JSON.stringify()`, and saves it to `localStorage`.
    - **`loadTasks()` function:** Create a function that reads the JSON string from `localStorage`, parses it with `JSON.parse()`, and populates the `tasks` array. **Important:** The parsed objects will be generic objects, not instances of your `Task` class. You'll need to map them back into `Task` instances.
    - **Event Listener for the form:**
        - When the form is submitted, prevent the default page reload.
        - Create a new `Task` instance with a unique ID (e.g., `Date.now()`), the text from the input, and a default status of `'todo'`.
        - Add the new task to your `tasks` array.
        - Call `saveTasks()`.
        - Call `renderTasks()` to update the UI.
    - **Modify initial load:** Instead of using the static array, call `loadTasks()` at the start and then `renderTasks()`.

Goal:: You can now add tasks, refresh the page, and your tasks will still be there.



Enter new task...

Add Task

To Do

Test Task 1

Test Task 2

Test Task 3

In Progress

Done

NOTE: Note that this only works **client-side** on the specific **browser** and **user profile** that you are using. If you clear your browser's cache, cookies, or site data, you will **lose** whatever is stored in `localStorage`. Therefore, this is **not persistent storage** for important or critical information. It is useful, however, to store data that must be remembered between page loads and across sessions. In contrast, **`sessionStorage`** is even more ephemeral and will only store information for the duration of the current running session (data is cleared when the tab is closed). For durable, secure, and globally accessible data, we must implement a server-side component that stores this information on a database, as we will see in later classes.

Level 3: Interactivity - Drag and Drop

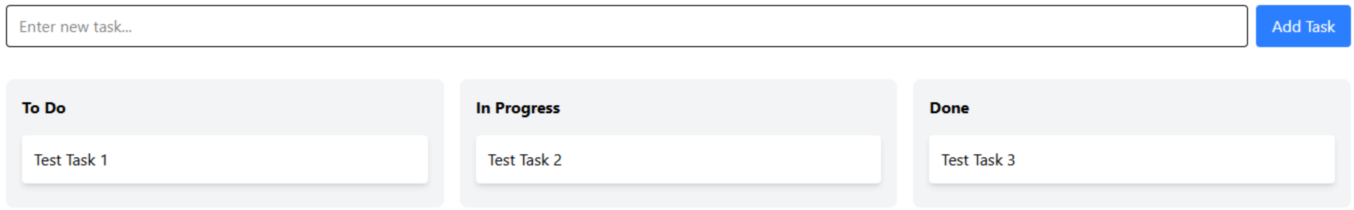
Objective: Implement drag-and-drop functionality to move tasks between columns.

Key Topics: Drag and Drop API (`draggable`, `dragstart`, `dragover`, `drop`).

Instructions:

1. **Rendering (`main.js`):**
   - Inside your `renderTasks` function, when creating a task card element, set its `draggable` attribute to `true`.
   - Add a `dragstart` event listener to the card. Inside the listener, use `event.dataTransfer.setData('text/plain', task.id)` to store the ID of the task being dragged.
2. **Column Event Listeners (`main.js`):**
   - For each column (`todo-column`, etc.), add a `dragover` event listener. Inside, you **must** call `event.preventDefault()` to allow a drop to occur.
   - Add a `drop` event listener to each column. Inside this listener:
     - Call `event.preventDefault()`.
     - Get the task ID using `event.dataTransfer.getData('text/plain')`.
     - Find the corresponding task in your `tasks` array.
     - Update the `status` of that task to match the column it was dropped into (e.g., if dropped in `inprogress-column`, set `status` to `'inprogress'`).
     - Call `saveTasks()`.
     - Call `renderTasks()` to reflect the change in the UI.

Goal: You can now click and drag a task card from one column and drop it into another, and the change will be saved.



Level 4: Handling Files - Canvas Image Loading and Drawing

Objective: Create a drop zone that accepts image files from the user's computer, loads them into a canvas, and allows drawing black lines by dragging the mouse over the canvas.

Key Topics: Drag and Drop API (with files), File API (`FileReader`), Canvas API (drawing images and paths).

Instructions:

1. **HTML (`index.html`):**

  - Below your Kanban board container, add a new section.
  - Inside, create a `div` with an `id` of `image-drop-zone`. Give it some text like "Drag & Drop an Image Here".
  - Add a `<canvas>` tag below the drop zone with an `id` of `displayed-canvas` and set attributes like `width="400"` and `height="300"`. Initially, it will be blank.

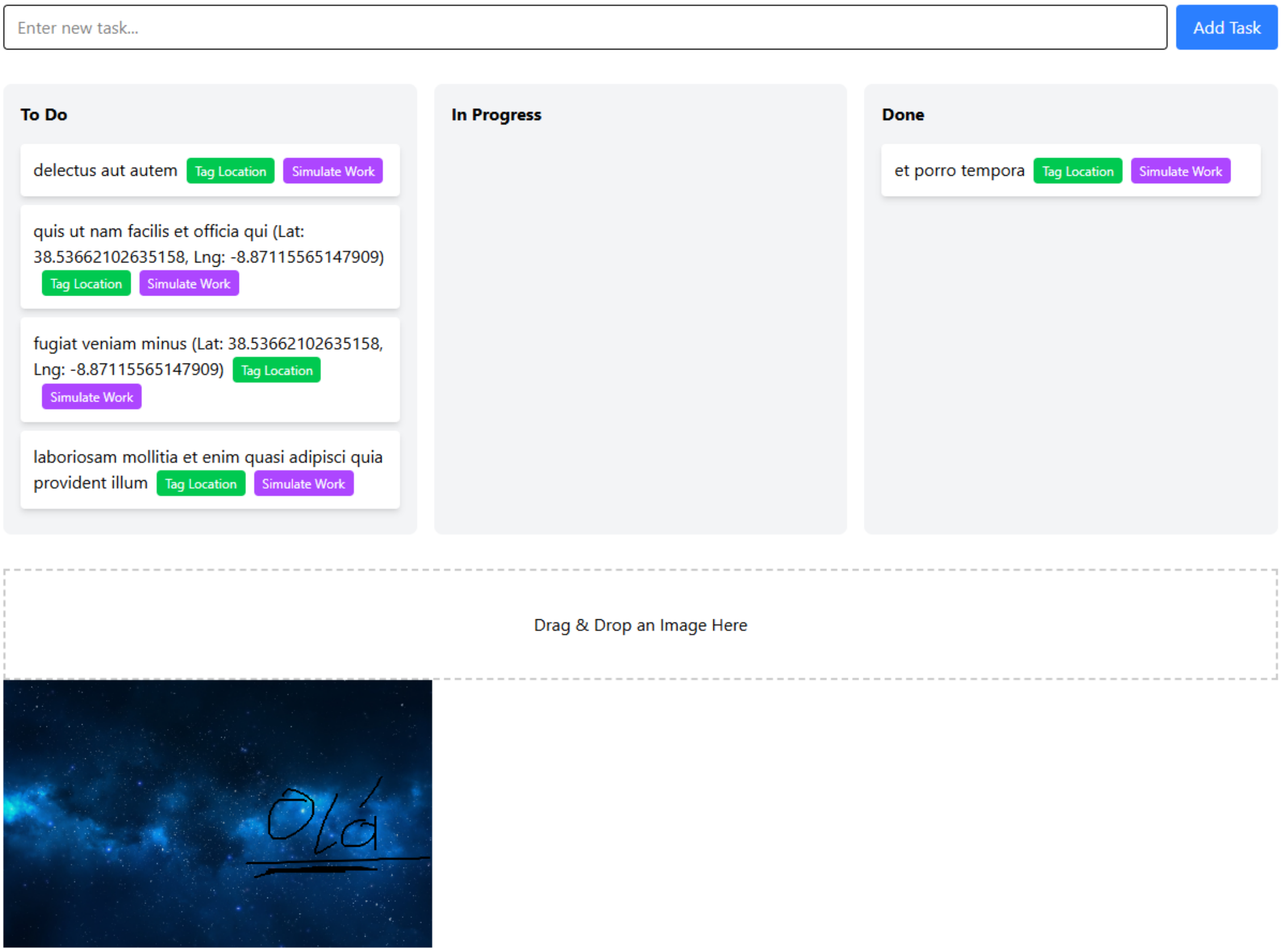
2. **CSS (`styles/main.css`):**

  - Style the `#image-drop-zone` with a border (e.g., `2px dashed #ccc`), padding, and a minimum height to make it a clear target.
  - Create a class named `.drag-over` that changes the background color or border color of the drop zone. This will provide visual feedback to the user.
  - Style the `#displayed-canvas` to have a maximum width and height so it doesn't break your layout (e.g., `max-width: 100%; max-height: 400px; border: 1px solid #ccc; cursor: crosshair;` for drawing feedback).

3. **Main Logic (`scripts/main.js`):**

  - Import the `CanvasDrawer` class from `./CanvasDrawer.js`.
  - Get a reference to the drop zone element.
  - Instantiate the drawer: `const drawer = new CanvasDrawer(document.getElementById('displayed-canvas'));`.
  - Add a `dragover` event listener to the drop zone. It must call `event.preventDefault()` and add the `.drag-over` class to the element.
  - Add a `dragleave` event listener to remove the `.drag-over` class.
  - Add a `drop` event listener:
    - Call `event.preventDefault()` and remove the `.drag-over` class.
    - Access the dropped files via `event.dataTransfer.files`.
    - Check if at least one file was dropped and if `files[0].type.startsWith('image/')`.
    - If it is an image file, create a new `FileReader()`.
    - Set up the `reader.onload` event handler. When the file is loaded, `reader.result` will contain a Base64 data URL of the image.
    - Inside `onload`, create a new `Image` object, set its `src` to `reader.result`, and in the image's `onload` handler, get the canvas and ctx (`const canvas = document.getElementById('displayed-canvas'); const ctx = canvas.getContext('2d');`), clear the canvas (`ctx.clearRect(0, 0, canvas.width, canvas.height);`), and draw the image proportionally (`ctx.drawImage(img, 0, 0, img.width * (canvas.width / img.width), img.height * (canvas.width / img.width));`).
    - Finally, call `reader.readAsDataURL(files[0])` to start the file reading process.
  - The `CanvasDrawer` class handles mouse events for drawing (already provided along with this Lab Assignment).

Goal: You can drag an image file from your computer, drop it onto the designated area to load it into the canvas (scaled proportionally), and then drag your mouse over the canvas to draw black lines on top of the image.



delectus aut autem

quis ut nam facilis et officia qui (Lat: 38.53662102635158, Lng: -8.87115565147909)

fugiat veniam minus (Lat: 38.53662102635158, Lng: -8.87115565147909)

laboriosam mollitia et enim quasi adipisci quia provident illum

et porro tempora

Level 5: Asynchronicity - Loading Sample Data

Objective: Use the Fetch API to load sample "to-do" items from a public API if the board is empty.

Key Topics: Fetch API, Promises, `async/await`.

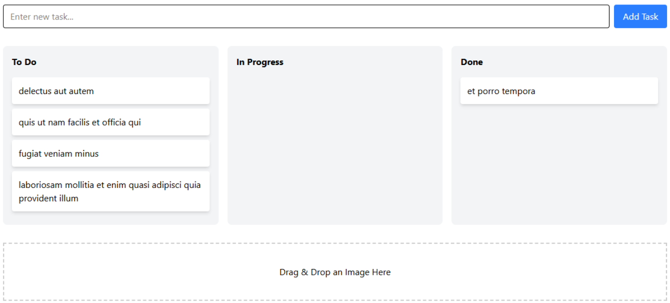
Instructions:

1. **Main Logic (`main.js`):**
   - Create a new `async` function named `fetchInitialData`.
   - Inside this function, use `await fetch('https://jsonplaceholder.typicode.com/todos?_limit=5')` to get five sample tasks.
   - Process the response to get the JSON data.
   - Map the fetched data to new `Task` instances. The fetched todos have a `title` property (use for `text`) and a `completed` property (you can use this to set the `status` to `'done'` or `'todo'`).
   - Return the array of new `Task` instances from the function.
   - **Modify initial load:** In your `loadTasks` function, check if `tasks.length === 0` after attempting to load from localStorage. If it is, assign the result of `await fetchInitialData()` to the `tasks` array and call `saveTasks()` to persist the data.
   - After calling `loadTasks()`, ensure `renderTasks()` is invoked to update the UI.

Goal: When a new user opens the app for the first time, it will be pre-populated with 5 sample tasks from an online API.

NOTE: Async/await and Promises are both JavaScript mechanisms for handling asynchronous operations, but they differ in syntax and usability: Promises use chained `.then()` and `.catch()` methods for a functional, callback-based flow that can become nested and hard to read in complex scenarios, while async/await acts as syntactic sugar over Promises, allowing code to resemble synchronous logic with `await` keywords inside `async` functions and `try/catch` for errors, making it more intuitive, easier to debug, and better suited for sequential tasks—though both support parallelism via `Promise.all()` and have similar performance, async/await is the preferred modern approach for most applications due to its readability.

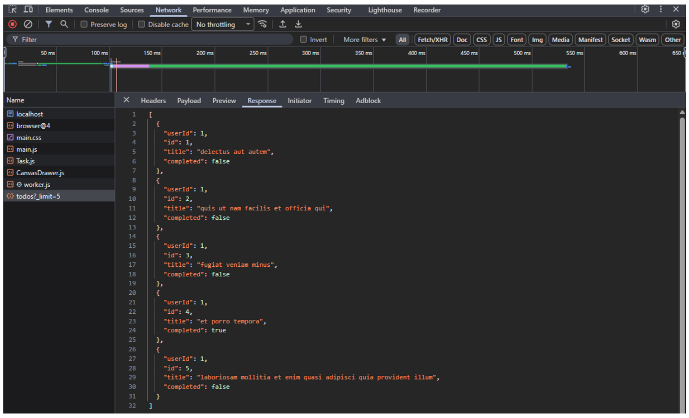
Challenge: Also implement this same logic using Promises instead of async/await and compare the approaches.



To Do
- delectus aut autem
- quis ut nam facilis et officia qui
- fugiat veniam minus
- laboriosam mollitia et enim quasi adipisci quia provident illum

In Progress

Done
- et porro tempora



```json
[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
  {
    "userId": 1,
    "id": 3,
    "title": "fugiat veniam minus",
    "completed": false
  },
  {
    "userId": 1,
    "id": 4,
    "title": "et porro tempora",
    "completed": true
  },
  {
    "userId": 1,
    "id": 5,
    "title": "laboriosam mollitia et enim quasi adipisci quia provident illum",
    "completed": false
  }
]
```

Level 6 (Bonus 1): Advanced APIs - Geolocation & Web Workers

Objective: Enhance tasks with location data and simulate a complex background process.

Key Topics: Geolocation API, Web Workers, Notification API.

Instructions:

1. **Geolocation:**

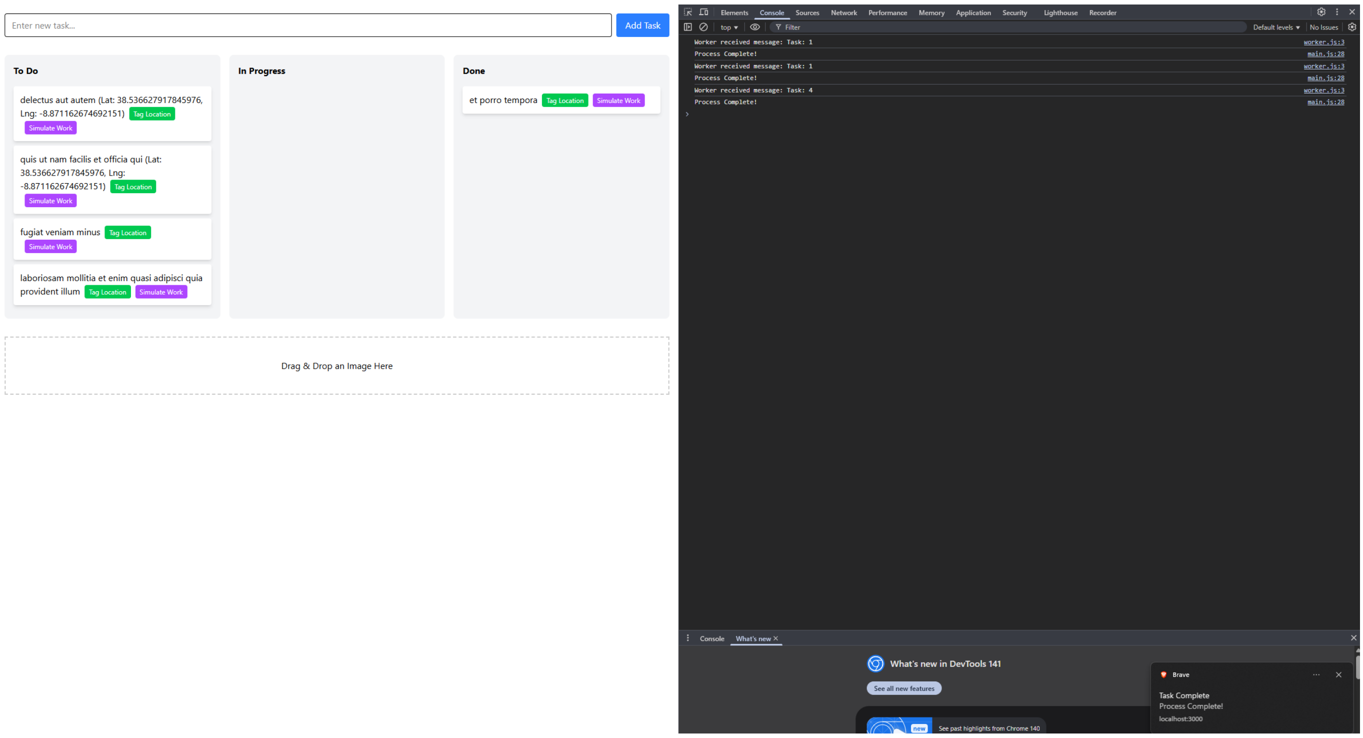
    - In your `renderTasks` function, add a "Tag Location" button to each task card.
    - Add an event listener to this button. On click, call `navigator.geolocation.getCurrentPosition()`.
    - In the success callback, get the `latitude` and `longitude` from the `position.coords` object.
    - Find the relevant task and add a `location` property to it, and update the rest of your code accordingly (e.g., you must add a `location` property to the `Task` class, and you must change the contructor to account for it and use it when loading information from the `localStorage`).
    - Update the task card's text to show the coordinates.
    - Don't forget to `saveTasks()`.

2. **Web Workers and Notification API:**

    - Create a new file: `worker.js`.
    - Inside `worker.js`, create a simple `onmessage` handler that simulates a heavy task (e.g., a loop that counts to a large number) and then uses `postMessage()` to send back a "Process Complete!" message.
    - In `main.js`, create a single `Worker` instance: `const worker = new Worker('worker.js');`.

    - Add a "Simulate Work" button to each task card.
    - When clicked, call `worker.postMessage()`.
    - Set up an `onmessage` listener for the worker instance to display a browser notification using the Notification API when the result comes back. Handle permission requests if necessary.

Goal: Tasks can be tagged with a location, and you can trigger a background process without blocking the main thread, with completion notified via a browser popup.



To Do

- delectus aut autem (Lat: 38.536627917845976, Lng: -8.871162674692151)
- quis ut nam facilis et officia qui (Lat: 38.536627917845976, Lng: -8.871162674692151)
- fugiat veniam minus
- laboriosam mollitia et enim quasi adipisci quia provident illum

In Progress

Done

- et porro tempora

Drag & Drop an Image Here

```
Worker received message: Task: 1                    worker.js:3
Process Complete!                                   main.js:28
Worker received message: Task: 1                    worker.js:3
Process Complete!                                   main.js:28
Worker received message: Task: 4                    worker.js:3
Process Complete!                                   main.js:28
```

Task Complete
Process Complete!
localhost:3000

Level 7 (Bonus 2): Real-Time Communication - WebSockets

Objective: Integrate WebSockets to send new tasks to an echo server and log the echoed response in the console for real-time feedback.

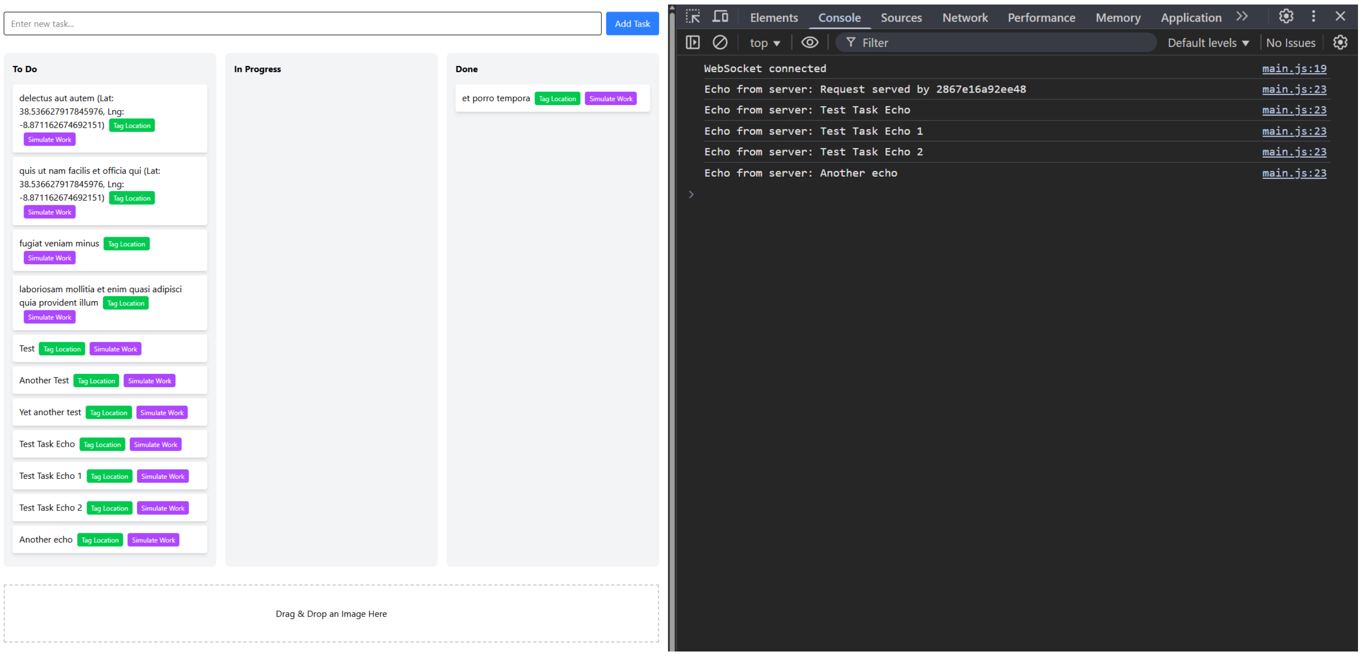
Key Topics: WebSockets API.

Instructions:

1. **Main Logic (`main.js`):**
    - Create a WebSocket connection to the public echo server: `const socket = new WebSocket('wss://echo.websocket.org');`.
    - Set up event listeners for the socket:
        - `onopen`: Log a message to the console indicating the connection is open (e.g., "WebSocket connected").
        - `onmessage`: Log the received data to the console (e.g., `console.log('Echo from server:', event.data);`).
        - Optionally, handle `onclose` and `onerror` for debugging.
    - **Modify the add task event listener:** After creating and adding the new task to the array (but before calling `saveTasks()` and `renderTasks()`), send the task text to the server via `socket.send(task.text);`. This will trigger an echo back, which will be logged in the console.

Goal: Whenever you add a new task, it is sent to the echo server, and the server echoes it back, which is logged in the browser console. This demonstrates real-time, bidirectional communication without blocking the UI.

NOTE: The echo server (`wss://echo.websocket.org`) is a public testing endpoint that simply reflects any message sent to it. In a real application, you would connect to your own WebSocket server for features like live collaboration or notifications. If the connection fails (e.g., due to network issues), the app should continue functioning normally—handle errors gracefully to avoid breaking task addition.



```
WebSocket connected                                          main.js:19
Echo from server: Request served by 2867e16a92ee48           main.js:23
Echo from server: Test Task Echo                             main.js:23
Echo from server: Test Task Echo 1                           main.js:23
Echo from server: Test Task Echo 2                           main.js:23
Echo from server: Another echo                               main.js:23
```

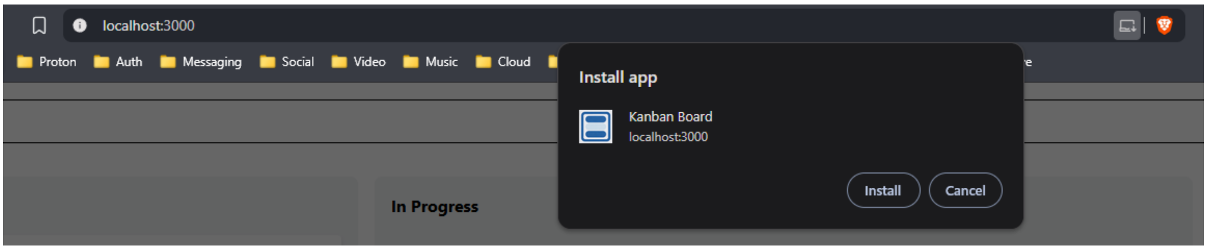
Level 8 (Bonus 3): The Final Goal - Creating a PWA

Objective: Convert your application into an installable Progressive Web App that works offline.

Key Topics: Service Worker, Web App Manifest.

Instructions:

1. **Web App Manifest:**
    - Create a `manifest.json` file in your root directory.
    - Add essential properties: `name`, `short_name`, `start_url`, `display` (`standalone`), and an `icons` array.
    - Create a simple 512x512 icon and save it in your project folder (or use the one provided with this Lab Assignment).
    - Link to the manifest from your `index.html`: `<link rel="manifest" href="manifest.json">`.

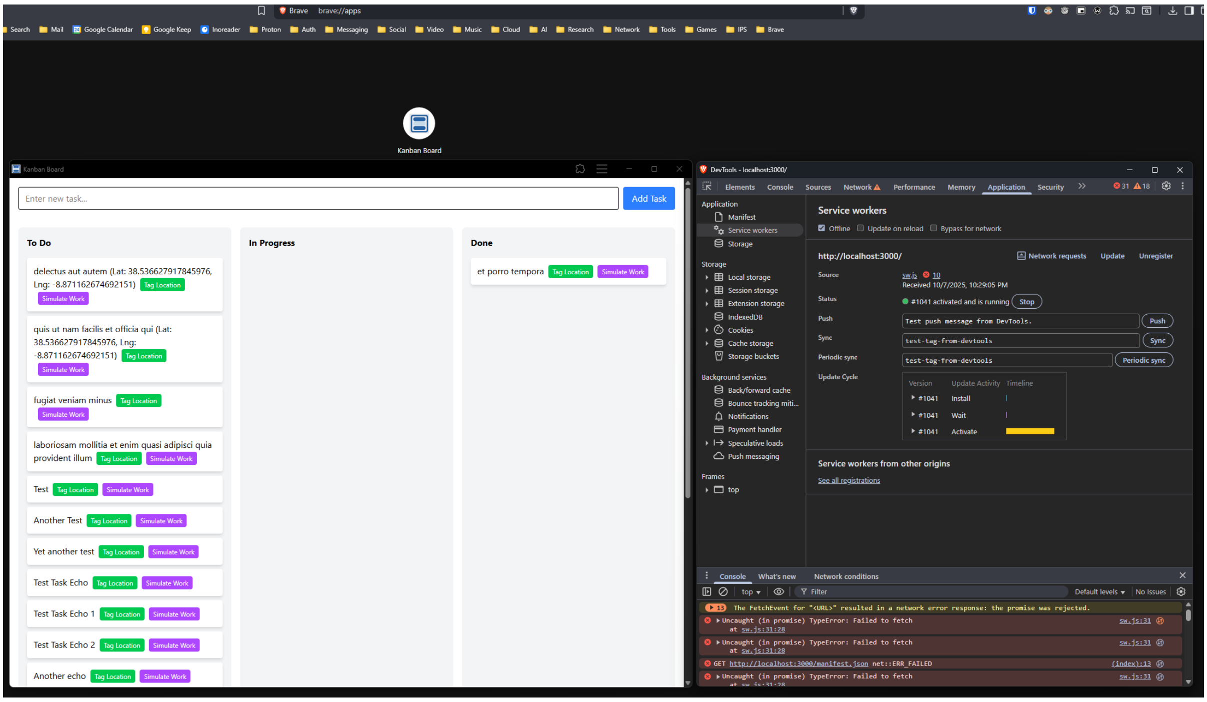


Install app

Kanban Board
localhost:3000

2. **Service Worker:**

   - Create a `sw.js` file in your root directory.
   - **`install` event:** Add an event listener for `install`. In it, open a cache and add all your core files (`index.html`, `style.css`, `main.js`, `js/Task.js`) to the cache.
   - **`fetch` event:** Add a listener for `fetch`. Implement a "cache-first" strategy: check if the requested resource is in the cache. If it is, serve it from the cache. If not, fetch it from the network.
   - **Register the Service Worker:** In `main.js`, add the code to check if `serviceWorker` is supported in the `navigator` and, if so, register your `sw.js` file.



Enter new task...

To Do

- delectus aut autem (Lat: 38.536627917845976, Lng: -8.871162674692151) Tag Location Simulate Work
- quis ut nam facilis et officia qui (Lat: 38.536627917845976, Lng: -8.871162674692151) Tag Location Simulate Work
- fugiat veniam minus Tag Location Simulate Work
- laboriosam mollitia et enim quasi adipisci quia provident illum Tag Location Simulate Work
- Test Tag Location Simulate Work
- Another Test Tag Location Simulate Work
- Yet another test Tag Location Simulate Work
- Test Task Echo Tag Location Simulate Work
- Test Task Echo 1 Tag Location Simulate Work
- Test Task Echo 2 Tag Location Simulate Work
- Another echo Tag Location Simulate Work

In Progress

Done

- et porro tempora Tag Location Simulate Work

Service workers

☑ Offline ☐ Update on reload ☐ Bypass for network

http://localhost:3000/

Source	sw.js 10 Received 10/7/2025, 10:29:05 PM
Status	#1041 activated and is running
Push	Test push message from DevTools.
Sync	test-tag-from-devtools
Periodic sync	test-tag-from-devtools

Update Cycle

Version	Update Activity
#1041	Install
#1041	Wait
#1041	Activate

Service workers from other origins

See all registrations

```
13 The FetchEvent for "<URL>" resulted in a network error response: the promise was rejected.
Uncaught (in promise) TypeError: Failed to fetch                                  sw.js:31
    at sw.js:31:28
Uncaught (in promise) TypeError: Failed to fetch                                  sw.js:31
    at sw.js:31:28
GET http://localhost:3000/manifest.json net::ERR_FAILED                       (index):13
Uncaught (in promise) TypeError: Failed to fetch                                  sw.js:31
    at sw.js:31:28
```

Goal: Your application can now be installed on desktops or mobile devices like a native app and will function offline, allowing users to manage tasks seamlessly even without an internet connection (with some features potentially limited, such as real-time updates).

NOTE: Beyond enabling offline access and installability in PWAs, service workers serve as a programmable proxy between your web app, the browser, and the network, allowing for advanced capabilities like intercepting and modifying network requests for custom caching strategies, delivering push notifications even when the app is closed, synchronizing data in the background (e.g., queuing actions for later upload), and boosting performance through intelligent resource precaching. These features make web apps feel more native and reliable, especially in low-connectivity scenarios.