Background Story

It was a rainy afternoon in Gloria, Oriental Mindoro - typical monsoon day when barangay hall operations slow to a crawl. Odessa was in her home office, sipping hot kamote tea, when her cousin from the barangay association called her in a panic.

“Ods, we have so many resident complaints, permit requests, and purok assignments. Every page refresh resets our form data. We need a tool that remembers our place—parang memory ng lolo’t lola natin!” 😅

Odessa smiled. She knew this was a perfect chance to practice application state management. She recalled her days at the startup, juggling user sessions, routing between login, dashboard, and settings screens—all without losing data. Now, she had to build a light barangay app that:

1. **Remembers** which screen the user is on (e.g., “New Complaint” vs. “View Residents”).
2. **Stores** drafts and form inputs as they type.
3. **Persists** data across refresh, so no lost work.

She sketched a simple flow: a global `state` object to hold `currentScreen`, `draftComplaint`, and a list of `residents`. She’d wrap it in a module (`stateManager.js`) with methods `getState()`, `setState()`, `saveState()`, and `loadState()`. For persistence, she’d use `localStorage`.

By evening, her code editor lit up with neatly organized files:

- `index.html` with three `<section>` screens
- `stateManager.js` exporting state functions
- `app.js` to wire UI events, call `showScreen()`

Within minutes, her cousin’s barangay app could switch screens without losing the draft complaint. Even after reload, the draft reappeared! Odessa leaned back, proud—she was thinking like a systems architect, designing clear state flows, preventing global namespace collisions, and ensuring data persisted like a true Pinoy talaarawan (journal).

As the rain lulled to a drizzle, Odessa imagined scaling this to a full municipal system. But first, she’d teach you how to master application state—one global object at a time. 🌧️ ➡️ 🔄

Theory & Lecture Content

Managing state means tracking the “current values” of your app: UI screen, form inputs, user data, etc.

1. Why State Matters

- Keeps UI in sync with data.
- Prevents lost inputs on navigation or refresh.
- Centralizes data for debugging & testing.

2. Global Variables & Namespacing

A single global object avoids polluting `window`:

```
// BAD: many globals
let currentScreen = "home";
let draft = {};

// BETTER: single namespace
const AppState = {};
```

Use a module pattern:

```
// stateManager.js
const AppState = {
  currentScreen: "home",
  draftComplaint: "",
  residents: [],
};
export default AppState;
```

3. Encapsulating State Access

Direct mutation is risky. Provide getters/setters:

```
// stateManager.js
const state = {
  currentScreen: "home",
  draftComplaint: "",
  residents: [],
};

export function getState(key) {
  return state[key];
}

export function setState(key, value) {
  state[key] = value;
}
```

4. Persisting State (localStorage)

```
export function saveState() {
  localStorage.setItem("appState", JSON.stringify(state));
}

export function loadState() {
  const json = localStorage.getItem("appState");
  if (json) {
    Object.assign(state, JSON.parse(json));
    }
  }
```

Call `loadState()` on startup, and `saveState()` on changes.

Reference:
https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

5. Screen Management

Show/hide sections based on `currentScreen`:

```
export function showScreen(name) {
  setState("currentScreen", name);
  saveState();
  document.querySelectorAll("section").forEach((sec) => {
    sec.style.display = sec.id === name ? "block" : "none";
  });
}
```

Exercises

Exercise 1: Build a State Manager

Problem Statement
Create `stateManager.js` with a private state object and functions: `getState`, `setState`, `saveState`, `loadState`.

TODOs

- Define private `state` with keys: `currentScreen`, `draft`, `items` (empty array).
- Export `getState(key)`, `setState(key, value)`.
- Export `saveState()` and `loadState()` using `localStorage`.

Starter Code (stateManager.js)

```
// TODO: implement stateManager

const state = {
  currentScreen: "home",
  draft: "",
  items: [],
};

// export functions here
```

Full Solution (stateManager.js)

```
const state = {
  currentScreen: "home",
  draft: "",
  items: [],
};

export function getState(key) {
  return state[key];
}

export function setState(key, value) {
  state[key] = value;
}

export function saveState() {
  localStorage.setItem("appState", JSON.stringify(state));
}

export function loadState() {
  const json = localStorage.getItem("appState");
  if (json) {
    Object.assign(state, JSON.parse(json));
  }
}
```

Exercise 2: Screen Navigation

Problem Statement

In `app.js`, use `stateManager` to load state on startup, then show the correct screen. Implement click handlers to navigate.

TODOs

- Import `getState`, `setState`, `saveState`, `loadState`.
- On DOMContentLoaded, call `loadState()` and `showScreen(getState('currentScreen'))`.
- Attach click events on nav buttons to call `showScreen('home')`, `showScreen('form')`, etc.

Starter Code (app.js)

```
import { getState, setState, saveState, loadState } from "./stateManager.js";

function showScreen(name) {
  // TODO: implement show/hide
}

document.addEventListener("DOMContentLoaded", () => {
  // TODO: loadState and showScreen
  // TODO: attach button handlers
});
```

Full Solution (app.js)

```
import { getState, setState, saveState, loadState } from "./stateManager.js";

function showScreen(name) {
  setState("currentScreen", name);
  saveState();
  document.querySelectorAll("section").forEach((sec) => {
    sec.style.display = sec.id === name ? "block" : "none";
  });
}

document.addEventListener("DOMContentLoaded", () => {
  loadState();
  showScreen(getState("currentScreen"));

  document
    .getElementById("btnHome")
    .addEventListener("click", () => showScreen("home"));
  document
    .getElementById("btnForm")
    .addEventListener("click", () => showScreen("form"));
  document
    .getElementById("btnList")
    .addEventListener("click", () => showScreen("list"));
});
```

Demonstration HTML (index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>State App</title>
  </head>
  <body>
    <nav>
      <button id="btnHome">Home</button>
      <button id="btnForm">Form</button>
      <button id="btnList">List</button>
    </nav>
    <section id="home">🏠 Welcome Home</section>
    <section id="form" style="display:none">
      <h2>📝 Draft Form</h2>
      <textarea id="draft"></textarea>
    </section>
    <section id="list" style="display:none">
      <h2>📋 Items List</h2>
    </section>
    <script type="module" src="app.js"></script>
  </body>
</html>
```

Exercise 3: Persist Draft Input

Problem Statement
Enhance the form screen so that the textarea's content is saved to state and `localStorage` on every keystroke and restored on load.

TODOs

- In `app.js`, get `#draft` textarea.
- On `input` event, call `setState('draft', value)` and `saveState()`.
- After `loadState()`, set `textarea.value = getState('draft')`.

Full Solution (app.js) — additions only

```
const draftEl = document.getElementById("draft");
draftEl.value = getState("draft");
draftEl.addEventListener("input", (e) => {
  setState("draft", e.target.value);
  saveState();
});
```

Test Cases

stateManager.test.js

```
import { getState, setState, saveState, loadState } from "./stateManager.js";

describe("stateManager", () => {
  beforeEach(() => {
    localStorage.clear();
    setState("currentScreen", "home");
    setState("draft", "");
    setState("items", []);
  });

  test("getState and setState work", () => {
    setState("draft", "Hello");
    expect(getState("draft")).toBe("Hello");
  });

  test("saveState and loadState persist data", () => {
    setState("currentScreen", "form");
    setState("draft", "Test");
    saveState();

    // mutate in-memory then reload
    setState("currentScreen", "home");
    setState("draft", "");
    loadState();

    expect(getState("currentScreen")).toBe("form");
    expect(getState("draft")).toBe("Test");
  });
});
```

app.test.js

```
/**
 * @jest-environment jsdom
 */
import fs from "fs";
import path from "path";
import { loadState, saveState, getState } from "./stateManager.js";

beforeAll(() => {
  document.body.innerHTML = fs.readFileSync(
    path.resolve(__dirname, "index.html"),
    "utf8"
  );
  require("./app.js");
});

describe("app screen navigation & draft persistence", () => {
  test("restores and saves screen state", () => {
    // initial is home
    expect(document.getElementById("home").style.display).toBe("block");

    document.getElementById("btnForm").click();
    expect(getState("currentScreen")).toBe("form");
    expect(document.getElementById("form").style.display).toBe("block");

    document.getElementById("btnHome").click();
    expect(getState("currentScreen")).toBe("home");
  });

  test("draft input persists across reload", () => {
    const textarea = document.getElementById("draft");
    textarea.value = "Barangay update";
    textarea.dispatchEvent(new Event("input"));
    // simulate reload
    document
      .querySelectorAll("section")
      .forEach((sec) => (sec.style.display = "none"));
    loadState();
    require("./app.js");

    const newTextarea = document.getElementById("draft");
    expect(newTextarea.value).toBe("Barangay update");
  });
});
```

Closing Story

As dusk fell over Gloria, Odessa watched her cousin’s barangay workers navigate the app effortlessly. Screens switched without losing drafts, resident lists loaded instantly, and every keystroke was saved—no more “Data lost!” panic. She closed her laptop, thinking: “An architect thinks in state flows and data persistence.”

Her next challenge loomed: orchestrating user interactions with event delegation and custom events, making UIs not just stateful but interactive. Tomorrow, she’d dive into listening for clicks at scale, bubbling events, and crafting custom signals between modules—like barangay tanod coordinating a smooth fiesta procession. 🎉

Odessa smiled, ready to design the next layer of her application’s intelligence. The state was managed—now it was time to make it responsive. 🚀