Javascript Task Sheet 1

Date: 21 May, 2025

✅ Task 1: Rebuild `map`, `filter`, `reduce` — Without Native Methods

What to Build:
Recreate JavaScript's `.map()`, `.filter()`, and `.reduce()` functions by writing your own versions from scratch. Do not use any native array methods.

```
myMap(array, callback)
myFilter(array, callback)
myReduce(array, callback, initialValue)
```

Constraints:

- ❌ No `.map`, `.filter`, `.reduce`, `.forEach`, or `.flatMap`
- ✅ Use only `for`, `while`, or recursion

Why This Matters:
These three functions are the foundation of functional programming in JS. You'll discover how callbacks are invoked, how accumulators work, and how iteration transforms data. This makes future React data rendering logic intuitive.

AI Trap: AI will give you working code, but won't help you **explain it, optimize it**, or **design multiple implementations with constraints**. You'll write 3 different styles (loop-based, recursion-based, and mutation-based) and compare them.

✅ Task 2: Fake Terminal CLI in the Browser

What to Build:
Simulate a terminal/command-line UI inside a web browser. Commands to support:

- `add task <text>`
- `list tasks`
- `clear tasks`
- `delete task <id>`

Display command input/output in a styled `<div>`, not `<input>`.

Why This Matters:
You'll learn to:

- Parse strings
- Build internal state (like a task manager)
- Dynamically create and update DOM like a terminal
  This builds a strong mental model for interpreting inputs and building custom renderers (a key React skill).

AI Trap: AI might help generate a basic UI, but parsing commands, handling edge cases, and building a versioned internal state cannot be easily copy-pasted.

✅ Task 3: Tab System Without `classList` or `style.display`

What to Build:
Create a tab component with three tabs (e.g., Home, About, Contact). Switching tabs should:

- Hide other sections
- Show only selected content
  But: You cannot use `.classList` or inline styles to do this.

Why This Matters:
You'll be forced to think creatively:

- How else can we show/hide elements?
- Can you reorder the DOM? Use attributes? Custom logic?
  This builds problem-solving skills when libraries are unavailable or break.

AI Trap: AI will usually suggest `.classList.toggle()` or `style.display = 'none'`. You must deliberately bypass this with custom logic.

✅ Task 4: Manual ToDo App with Debug State Panel

What to Build:
A complete ToDo app with add, delete, toggle-complete functionality.

- A floating state panel that:
- Shows the current task array
- Logs every change (add/delete) with timestamp

Why This Matters:
This is how `Redux DevTools` works. You'll learn:

- Manual state handling
- DOM re-rendering
- Logging state changes for debugging

AI Trap: The task includes building a *debugger*, not just the app — AI won't know what you mean by this unless you do the thinking.

✅ Task 5: Stopwatch & Timer — With Artificial Lag Injection

What to Build:

- Stopwatch (start, stop, lap, reset)
- Countdown timer (user inputs time)
- Digital clock (real time)

Inject **delays or race conditions** by wrapping logic inside random timeouts:

```
setTimeout(() => updateTimer(), Math.random() * 1000)
```

Why This Matters:
Teaches timing issues, event loop, and controlling concurrent intervals — essential for React `useEffect`, Node's async behavior, and debugging bugs caused by unintentional async behavior.

AI Trap: You're deliberately making things **break** and fixing them — something AI isn't good at doing blindly.

✅ Task 6: Modal with Keyboard Accessibility & Focus Trap

What to Build:
Build a modal that:

- Auto-focuses the first input
- Disallows focus leaving the modal when opened
- Closes on:
  - Escape key
  - Clicking outside modal

Why This Matters:
Manual DOM control like this is core to accessibility and UI frameworks. This teaches you what `focus trap` libraries do under the hood.

AI Trap: AI won't implement a correct focus trap without you understanding `document.activeElement`, `tabindex`, and keyboard listeners.

✅ Task 7: Live Preview + Rule-Driven Validation Engine

What to Build:
Form fields:

- Name, email, password, bio
- Profile image upload
- Display live preview next to form

Also:

- Create a `validateField()` that takes rules from JSON like:

```
{ name: ['required'], password: ['min:8'] }
```

Why This Matters:
You'll build your own form engine — a crucial abstraction used in every enterprise app.

AI Trap: Copy-pasting won't teach you how to separate rules, evaluate them, and show proper UX.

✅ Task 8: Manual Hash-Based Router with History Stack

What to Build:
Single Page App with 3 pages (`#home`, `#about`, `#contact`) using hash routing. Maintain:

- A manual navigation stack
- Back/forward tracking
- Scroll position restoration per view

Why This Matters:
You'll understand how React Router or Next.js routing works under the hood, and why SPA routing is harder than it looks.

AI Trap: AI won't generate a full-stack-aware version of history + scroll + state unless you design it.

✅ Task 9: Component System with `createComponent()`

What to Build:
Create a function `createComponent(tag, props, children)`
Use this to build:

- Button
- Avatar
- Card
- Input

Then implement a system to:

- Re-render only when props change
- Compare old and new props

Why This Matters:
You’re simulating React’s **component + virtual DOM + diffing** model.

AI Trap: AI will generate a simple factory function — but reconciling changes, props diffing, and selective DOM updating must be reasoned through.

✅ Task 10: Sortable Table with Debug Panel

What to Build:
A data table:

- Sortable by age/name/score
- Filter by text
- A floating panel shows:
    - Current filters
    - Current sort direction
    - Visible row count

Why This Matters:
This is what **dashboard tables and admin panels** are built on. State reflection is core to debugging production apps.

AI Trap: AI can help sort, but won’t tell you **why your table didn’t re-render**, or why a filter broke the sort order.

✅ Task 11: Virtual DOM Diff Engine + Visual Logs

What to Build:

- Define 2 virtual DOM JSON objects
- Write `diff(oldTree, newTree)` to:
    - List additions, removals, updates
    - Apply patch to DOM
- Log:

    - `+` for new nodes
    - `~` for changed nodes
    - `-` for removed nodes

Why This Matters:
This is how React performs DOM updates efficiently. You'll simulate reconciliation manually.

AI Trap: You can't solve this without **understanding tree comparison**, keying, and mutation.

✅ Task 12: Fetch with Retry + Backoff Strategy

What to Build:
Use `fetch()` to hit OpenWeatherMap API:

- On failure, retry after:
    - 1s → 2s → 4s → Stop after 5 tries
- Show loading, success, error states
- Add a retry button

Why This Matters:
Resilience, retries, and exponential backoff are real-world practices in all frontend/backend apps.

AI Trap: You'll have to *design a strategy*, not just write a `fetch()` function.

✅ Task 13: Drag & Drop Kanban Board — With Keyboard Support

What to Build:

- A **Kanban board** with 3 columns:
  🟡 To Do | 🟠 In Progress | 🟢 Done
- Each task card should be:
    - Draggable using native `dragstart`, `dragover`, `drop` events

    - Droppable into another column

Required Features:

- Manual state management (object with column arrays)
- Persist state in `localStorage`
- Support **keyboard movement**:
    - Focus on task → Use `← ↑ → ↓` or `WASD` to move tasks between columns

Why This Matters:

- You'll learn complex **DOM event coordination**
- Simulates advanced UI used in apps like Trello, Jira
- Keyboard support teaches **accessibility**, important for real users and compliance

AI Trap:
AI can help with drag-and-drop basics, but:

- Won't handle **keyboard accessibility**
- Won't track full column logic
- Won't persist state properly unless you design it

✅ Task 14: Lazy-loaded Image Gallery with Cache Detection

What to Build:

- Grid of 20+ images (with categories like "Nature", "People", "Architecture")
- Filter images by tag (e.g., buttons or dropdown)
- Lazy-load images only when visible (IntersectionObserver or scroll logic)

- On click, show modal preview

Advanced Twist:

- Use `performance.getEntriesByType('resource')` to check if image was loaded from **cache** or **network**
- Log this in a debug panel (e.g., “13 images loaded from cache”)

Why This Matters:

- Teaches **performance awareness**, image loading strategies
- Introduces you to browser performance APIs
- Bridges frontend design with **browser internals**

AI Trap:
AI won't think of **cache detection**, will likely skip **IntersectionObserver**, and won't explain **why certain images load faster** unless you design the logic.

✅ Task 15: Notepad Lite – With Rich Text + Undo/Redo Stack

What to Build:

- A text editor (like a mini Notepad) in the browser
- Features:
  - Bold / Italic / Underline buttons
  - Word and character count
  - Download as `.txt` file

Advanced Twist:

- Implement **Undo/Redo functionality**
  - Use a stack-based system: every keystroke is a change

    - Provide keyboard shortcuts (Ctrl+Z, Ctrl+Y)

Why This Matters:

- You'll learn to manage **complex internal state**
- Undo systems are core in collaborative apps like Google Docs, Figma

AI Trap:
AI may write a basic editor, but building **undo history**, applying operations in the right sequence, and syncing button states requires you to design a state machine.

✅ Task 16: Analog Clock + Timezone Switching UI

What to Build:

- A **real-time analog clock** using CSS + JS
- Render:
    - Hour, minute, and second hands that move every tick
- Add:
    - Dropdown with 5 timezones (UTC offsets)
    - On change, update the clock to reflect the new timezone

Why This Matters:

- Reinforces **DOM transforms**, time math, and `setInterval`
- Teaches how timezones affect date/time calculations
- Foundation for **server/client rendering consistency**

AI Trap:
AI may get the clock running, but timezone conversions require **manual Date math**, and adjusting for local offset is a reasoning-heavy task.

✅ Task 17: Custom Carousel with Infinite Loop + Smooth Animations

What to Build:

- A responsive carousel with:
  - 5 visible images (or slides)
  - Next/Prev buttons
  - Dot indicators
  - Auto-play every 3 seconds
- Implement:
  - Smooth scroll transitions
  - Infinite loop behavior (rewind to start seamlessly)

Advanced Twist:

- Inject 1000 slides/images
- Use **virtualized rendering** (only render 3–5 visible slides)

Why This Matters:

- Reinforces **scroll, animation, and state control**
- Virtualization is key for building **high-performance apps** (e.g., tables, image viewers)

AI Trap:
Basic carousels are AI-friendly, but **loop logic**, **performance optimization**, and **virtualization design** are too contextual for AI to do well.

✅ Task 18: Toast Notification System with Priority Queue

What to Build:

- Create your own `toast()` notification system (like in Slack, Gmail)

Call API like:

```
toast.success('Message sent')
toast.error('Network error')
```

-

System Rules:

- Max 3 visible at a time
- Auto-dismiss after 5s
- Stack top to bottom
- Queue up remaining toasts
- Prioritize by type:
  - error > warning > success > info

Why This Matters:

- You'll simulate a **priority queue + lifecycle system**
- This builds engineering muscle needed for stateful systems like **job queues, alerts, or event logs**

AI Trap:
AI will likely hard-code display logic. **Queuing and prioritization**, combined with **dismiss logic**, requires real architectural design.

✅ Task 19: JSON-Based Dynamic Form Engine + Export

What to Build:
Given a config like:

```
[
  { type: 'text', label: 'Email', required: true },
  { type: 'password', label: 'Password', rules: ['min:8'] },
  { type: 'select', label: 'Gender', options: ['M', 'F', 'Other'] }
]
```

- Render a dynamic form based on the config
- Handle:
  - Validations from `rules`
  - Error messages
  - Labeling
- On submit → Return JSON object of input values

Export Feature:

- Export this form engine as a function: `generateForm(config, containerId)`

Why This Matters:
You'll build a **form engine**, not just a form. This sets you up to build **low-code tools**, **CMS**, and **admin panels** in the real world.

AI Trap:
AI will render a static form — won't help you write a reusable engine unless you plan the API, handle configs, and abstract input logic.

✅ Task 20: Redux-Like State Store with Time Travel Debugger

What to Build:
Create a Redux-inspired store:

```
const store = createStore(reducer, initialState)
store.subscribe(render)
store.dispatch({ type: 'INCREMENT' })
```

Features:

- `getState`, `dispatch`, `subscribe`
- Actions and reducer pattern
- Add:
    - Visual log of past states
    - Time-travel buttons: Prev ⏮️ | Next ⏭️

Why This Matters:
You'll understand:

- What Redux does
- How `useReducer` and `useContext` are powered
- Debugging via **immutable state chains**

AI Trap:
AI might mimic Redux API, but you must:

- Design reducer logic
- Track full state history
- Write visual debugger and hook it into DOM
- Handle edge cases like jumping beyond array bounds
