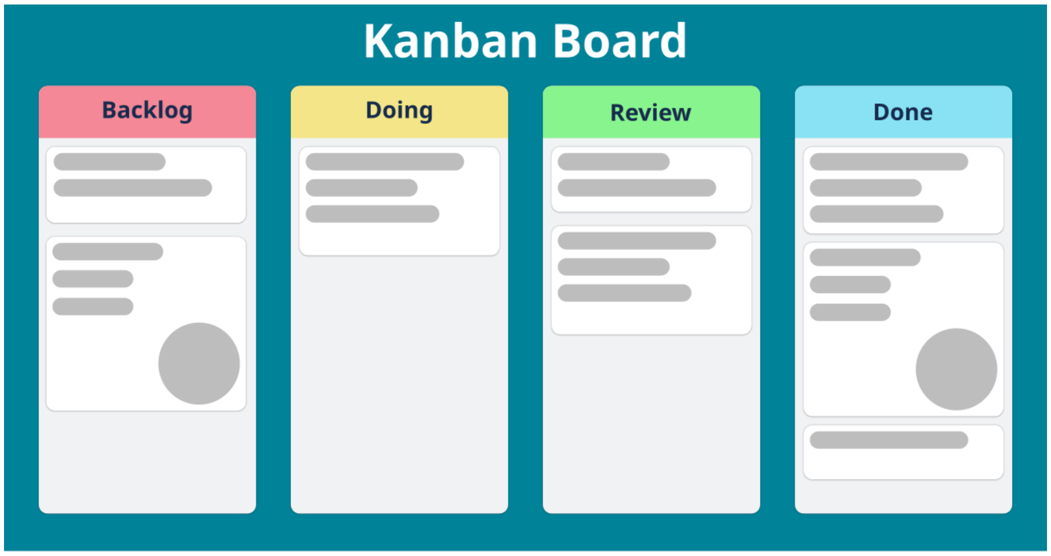
University of Science and Technology
Communications & Information Engineering Program
SW302 – User interface development

Assignment 3
React

- ## Assignment Title: Kanban board using React

  Objective: Create a collaborative Kanban board single-page application using the React library



- ## Requirements

  1) Assignment Setup & Tooling
  - Create a Vite React project using JavaScript & Tailwind
  - Configure ESLint with strict rules, including eslint-plugin-react and eslint-plugin-jsx-a11y.
  - Configure Prettier for formatting consistency.
  - Set up Jest + React Testing Library for unit and integration tests.
  - Set up Playwright or Cypress for one end-to-end test.
  - Mandatory package.json scripts: dev, build, lint, test, test:coverage, e2e.
  - Linting must produce zero errors.

  2) Core UI — Kanban Board, Lists, Cards
  - Implement a Kanban board using the following required React components: App, Board, ListColumn, Card, CardDetailModal, Header/Toolbar, BoardProvider (Context + Reducer), and utility components such as ConfirmDialog and InlineEditor. All these components must appear in your project structure.
  - Users must be able to add, rename, and archive lists (shown as columns, see the Figure above). All list data must be stored inside the global board state managed by useReducer + Context, and must be persisted in IndexedDB or localStorage for offline use.
  - Users must be able to add, edit, delete cards (task cards).
  - Each task card must support title, description, and tags.
  - Implement drag & drop for:
    - rearranging cards within the same list columns

   - moving cards between lists
- Drag and drop must use HTML5 Drag & Drop API or @dnd-kit/core. No prebuilt board components.
- Memoize Card components using React.memo, and wrap handlers in useCallback/useMemo whenever:
  - the parent component re-renders frequently,
  - card props do not change on most renders,
  - callbacks (e.g., onEdit, onMove, onDelete) are passed down to many Card components,
  - or rendering a Card involves non-trivial UI (modals, tag rendering, icons, etc.).
- IDs for cards/lists must be stable (UUID v4 acceptable).
- Your project must follow this required folder structure:

```
src/
        components/
                Board.jsx
                ListColumn.jsx
                Card.jsx
                CardDetailModal.jsx
                Header.jsx
                Toolbar.jsx
                ConfirmDialog.jsx
        context/
                BoardProvider.jsx
                boardReducer.js
        hooks/
                useBoardState.js
                useOfflineSync.js
                useUndoRedo.js
        services/
                api.js          // mock server logic + sync endpoints
                storage.js      // IndexedDB or localStorage handlers
        utils/
                validators.js
                helpers.js
        styles/
                global.css
                components.css
        App.jsx
        main.jsx
```

3) State Management — useReducer + Context

- All board data must be stored in a global state using useReducer + Context.
- All updates must be performed through reducer actions (no direct mutation).
- Introduce persistence through IndexedDB or localStorage.
- Must work offline: The application must fully load and function without an internet connection. All lists, cards, edits, moves, and deletions must be saved locally (IndexedDB or localStorage), and any changes made while offline must be queued and automatically synced with the server once the user reconnects.
- Implement optimistic UI updates:
  - UI updates instantly
  - then sync with a mock server. The mock server must be implemented using either Mock Service Worker (MSW) or a small Node/Express server. It must support endpoints for creating, updating, deleting, and moving cards/lists, and must be able to intentionally introduce delays or failures to test optimistic updates and conflict handling.
  - on failure: revert UI + display error

4) Syncing + Conflict Resolution

- Track lastModifiedAt and version for each list/card.
- When syncing:
  - If the server version is newer, perform a three-way merge. The server refers to the required mock server (MSW or Node/Express) that stores the authoritative version of lists and cards. Your merge logic must compare the base version, the local offline version, and the server version to decide how to resolve conflicts.
  - If merge fails, show a merge resolution UI where the user chooses what to keep
- Implement background sync: The application must automatically synchronize queued offline changes by using both of the following mechanisms:
  - on reconnect: detect when the browser becomes online using window.addEventListener("online", ...) and immediately sync all pending changes.
  - periodic sync timer: run a recurring timer (e.g., every 30–60 seconds) that attempts to sync any unsent updates and fetch the latest server data.

5) Custom Hooks

- Create three custom hooks, each fully documented & tested:
- useBoardState — wraps reducer actions for easier use.
- useOfflineSync — handles persistence, sync queue, retry logic, server interactions.
- useUndoRedo — supports multi-level undo/redo for board operations.

6) Performance Optimization

- App must remain smooth with 500+ cards. You must generate this dataset using a local data-seeding script (e.g., a small JavaScript script that creates at least 500 dummy cards and inserts them into IndexedDB or localStorage). This dataset must be used during performance testing and profiling.

- Use virtualization/windowing where necessary: You must virtualize the rendering of card lists using a library such as react-window or by implementing manual windowing logic. Any list displaying more than 30 cards must use virtualization to ensure smooth scrolling and rendering performance.
- Include React.memo, useMemo, useCallback to reduce re-renders.
- Provide performance profiling evidence: You must include a React Profiler export or a Chrome Performance trace (JSON or screenshot), along with a written analysis (150–250 words) explaining render times, component bottlenecks, and how your optimizations improved performance.

7) Code Splitting & Suspense

- Lazy-load heavy components (e.g., card detail modal).
- Use React.Suspense with a custom informative fallback.
- Show evidence of bundle splitting from the Vite build output.

8) Accessibility (A11y)

- Full keyboard operation for:
  - adding cards
  - editing cards
  - moving cards between lists
- Modal dialogs must trap focus and close with ESC.
- All interactive elements must have ARIA labels/roles/states.
- Must meet WCAG AA color contrast: You must verify color contrast using tools such as the axe-core browser extension, Lighthouse accessibility audits, or the WebAIM Contrast Checker, and fix any violations found.
- Generate an axe-core report and fix all critical/serious issues.

9) Testing Requirements

- Unit tests: hooks + components.
- Integration tests: reducer logic + offline syncing.
- One end-to-end test covering:
  - creating lists & cards
  - moving cards
  - performing offline changes
  - syncing after reconnect
- Minimum 80% test coverage (lines).

10) Summary Documentation

Create a docs/ directory with the following:

- Short essays: You must submit five essays, each 150–300 words, covering the following topics:
  - architecture choices: explain your overall system design, including component hierarchy, state ownership, data flow, folder structure, and the reasoning behind these decisions.

  - optimistic updates: describe the full sequence of events from user action → immediate UI update → local queueing → server sync → success/failure handling.
  - conflict resolution approach: detail your three-way merge method and how the UI handles unresolved conflicts.
  - performance issues found + solutions implemented: document real bottlenecks you observed and the exact optimizations you applied.
  - accessibility choices + testing: explain the steps taken to ensure keyboard navigation, ARIA compliance, color contrast, and how you validated accessibility with testing tools.
- At least two essays must include personal debugging anecdotes.
- Essays must reference actual file names and line numbers from your project.:

• Submission Package Requirements

1. **Full Git Repository:**
   - Must contain meaningful, atomic commits (no giant commits).
     Commit messages must reflect the implemented tasks.
2. **Source Code:**
   - Entire React project with JSX-only code.
   - All reducer logic, hooks, components, styles (Using Tailwind).
3. **Documentation Folder (docs/)**
   - Required essays (all mandatory).
   - Accessibility report.
   - Profiling report with screenshots/traces.
4. **README.md**
   - Clear project setup instructions.
   - How to run unit, integration, and e2e tests.
   - A 200–400 word architectural summary.

Evaluation Criteria (100%)

- Understanding — 20%
- Project setup, build, lint — 7%
- Core UI + drag & drop — 14%
- State management + persistence/offline — 10%
- Optimistic updates + conflict resolution — 11%
- Custom hooks + test coverage — 8%
- Performance optimization + profiling — 8%
- Accessibility compliance — 6%
- Unit, integration, e2e testing — 8%
- Documentation quality + commit hygiene — 8%

Restrictions

- No external UI component libraries (e.g., Material UI, Ant Design).
- No drag-and-drop board libraries — you must implement your own logic using HTML5 DnD or @dnd-kit/core.
- No CSS frameworks that provide prebuilt components (utility frameworks like Tailwind are allowed).
- AI-generated code copy/paste is strictly not allowed; the assignment is intentionally designed to require original reasoning.
- All essays must include authentic project-specific context (fake or generic text will not be accepted).

The problem is the system design here, it's really difficult