Day 14 — Mini Project: To-Do List App (Add, Delete, Update)

This canvas explains *how* a simple To-Do app works, step-by-step, then gives you a practical implementation plan and a short exercise you can finish in 15–20 minutes. I'll be explicit about why we do each thing and add small comments inside code examples so nothing is mysterious.

---

1) What the app must do

- **Add** a task (from an input form).
- **List** tasks (render an array as a list). Use `key` for each item.
- **Delete** a task (remove from state).
- **Update/Edit** a task (edit text inline and save change).
- Bonus: mark tasks as **completed**, persist to `localStorage` so tasks survive page refresh.

---

2) How it works — data & UI flow (high level)

1. **Single source of truth**: Keep one state variable `tasks` in the parent (e.g. `TodoApp`). It's an array of objects like `{ id, text, completed }`.
2. **Controlled input**: The add form uses `useState` to control the input value. On submit, the parent creates a new task and appends it to `tasks`.
3. **Render list**: Parent maps `tasks` → renders a `TaskItem` component for each task. Each `TaskItem` receives the task data and handler functions as props (delete, startEdit, saveEdit, toggleComplete).
4. **Update & delete**: When an action happens in a child, it calls the function passed from the parent — parent updates state immutably, React re-renders, UI updates.

Key ideas: immutability (never mutate `tasks` in place), unique `id` for keys (use `Date.now()` or `crypto.randomUUID()`), and controlled inputs for edit mode.

---

3) Data shape suggestion

```
// tasks array example
const tasks = [
  { id: 1661234567890, text: 'Buy milk', completed: false },
  { id: 1661234567891, text: 'Build Todo app', completed: true }
]
```

---

4) Components & responsibilities

- **TodoApp (parent)**

- Holds `tasks` and `input` state.
- Functions: `addTask`, `deleteTask`, `updateTask`, `toggleComplete`.
- Renders the input form and maps `TaskItem`.
- **TaskItem (child)**
- Receives `task` and action handlers as props.
- Displays task text or an edit input when in edit-mode.
- Calls `onDelete(task.id)`, `onSave(task.id, newText)`, `onToggle(task.id)`.

Why this split? Parent owns the data and is the single source of truth; children are UI helpers.

---

5) Core functions explained (what they do and why)

- `addTask(text)`
- Create a new task object with unique id.
- Set new state: `setTasks(prev => [...prev, newTask])`.
- Reason: avoid mutating previous state.
- `deleteTask(id)`
- Filter the tasks: `setTasks(prev => prev.filter(t => t.id !== id))`.
- Reason: produce a new array so React knows state changed.
- `updateTask(id, newText)`
- Map tasks and replace the matched item: `setTasks(prev => prev.map(t => t.id===id ? {...t, text: newText} : t))`.
- Reason: keep immutability and only change the one item.
- `toggleComplete(id)`
- Similar pattern: map and flip `completed` boolean.

---

6) Example: Minimal code (parent + child) with comments

```
// TodoApp.js (parent)
import React, { useState, useEffect } from 'react';
import TaskItem from './TaskItem';

function TodoApp(){
  const [tasks, setTasks] = useState(() => {
    // try to load saved tasks from localStorage on first render
    const raw = localStorage.getItem('tasks');
    return raw ? JSON.parse(raw) : [];
  });
  const [input, setInput] = useState(''); // controlled input for new task

  // Persist tasks to localStorage whenever tasks change
  useEffect(() => {
    localStorage.setItem('tasks', JSON.stringify(tasks));
  }, [tasks]);

  function addTask(e){
    e.preventDefault(); // prevent form reload
    const text = input.trim();
    if(!text) return; // skip empty

    const newTask = { id: Date.now(), text, completed: false };
    setTasks(prev => [...prev, newTask]); // append immutably
    setInput(''); // clear input
  }

  function deleteTask(id){
    setTasks(prev => prev.filter(t => t.id !== id));
  }

  function updateTask(id, newText){
    setTasks(prev => prev.map(t => t.id === id ? { ...t, text: newText } :
t));
  }

  function toggleComplete(id){
    setTasks(prev => prev.map(t => t.id === id ? { ...t, completed: !
t.completed } : t));
  }

  return (
    <div>
      <h2>To-Do List</h2>
      <form onSubmit={addTask}>
        <input value={input} onChange={e=>setInput(e.target.value)}
placeholder="Add task..." />
        <button type="submit">Add</button>
      </form>

      {tasks.length === 0 ? (
        <p>No tasks yet — add one!</p>
      ) : (
        <ul>
          {tasks.map(task => (
            <TaskItem
              key={task.id}
              task={task}
              onDelete={deleteTask}
```

```
                onUpdate={updateTask}
                onToggle={toggleComplete}
              />
            ))}
          </ul>
        )}
      </div>
    );
}
export default TodoApp;
```

```
// TaskItem.js (child)
import React, { useState } from 'react';

function TaskItem({ task, onDelete, onUpdate, onToggle }){
  const [isEditing, setIsEditing] = useState(false);
  const [draft, setDraft] = useState(task.text);

  function save(){
    const trimmed = draft.trim();
    if(!trimmed) return; // don't save empty
    onUpdate(task.id, trimmed); // call parent handler
    setIsEditing(false);
  }

  return (
    <li>
      <input type="checkbox" checked={task.completed}
onChange={()=>onToggle(task.id)} />

      {isEditing ? (
        <>
          <input value={draft} onChange={e=>setDraft(e.target.value)} />
          <button onClick={save}>Save</button>
          <button onClick={()=>{ setIsEditing(false); setDraft(task.text); }}
>Cancel</button>
        </>
      ) : (
        <>
          <span style={{ textDecoration: task.completed ? 'line-through' :
'none' }}>{task.text}</span>
          <button onClick={()=>setIsEditing(true)}>Edit</button>
          <button onClick={()=>onDelete(task.id)}>Delete</button>
        </>
      )}
    </li>
  );
}
```

```
export default TaskItem;
```

Notes: - `TaskItem` manages its local `isEditing` and `draft` state: local UI state lives in the component using it. - `TodoApp` manages the tasks array: app state lives in parent. - All state updates are done immutably.

---

7) UX & edge cases (what to think about)

- **Empty input**: prevent adding blank tasks.
- **Duplicate text**: you may allow or prevent duplicates.
- **Edit empty**: prevent saving empty text on edit.
- **Confirm delete**: optional confirmation for destructive actions.
- **Performance**: for very large lists consider virtualization (not needed now).

---

8) Implementation checklist — step by step (exact steps to follow)

1. Create `TodoApp.js` and `TaskItem.js` files.
2. In `TodoApp`:
3. `useState` for `tasks` and `input`.
4. Add `addTask`, `deleteTask`, `updateTask`, `toggleComplete` functions.
5. Render `<form>` and map `TaskItem`.
6. In `TaskItem`:
7. `useState` for `isEditing` and `draft`.
8. Render either edit inputs or display with Edit/Delete/Checkbox.
9. Call parent handlers where appropriate.
10. Test add / delete / edit flows manually.
11. (Optional) Add `useEffect` to persist `tasks` to `localStorage` on change (see example above).
12. Style it with CSS modules or inline styles.

---

9) 15–20 minute exercise (what to build now)

Target: Implement the basic app with Add & Delete + Edit feature (no persistence required). Follow these steps and stop when all passing:

1. Build `TodoApp` that can add tasks (test add and clearing input).
2. Render the list using `map()` and confirm keys work (no console warnings).
3. Implement Delete (button removes the item).
4. Implement Edit: clicking "Edit" replaces text with an input and Save/Cancel options. Save updates the parent state.

Stretch (if time remains): - Add checkbox to toggle `completed` and style completed tasks with `line-through`. - Save tasks to `localStorage`.

10) Quick debugging tips

- If list does not update: check you are returning a *new array* from `setTasks` (not mutating).
- If keys warning appears: make sure `key` is present and unique, use `task.id` not `index`.
- If edit input shows old text after save: ensure you reset `isEditing` and `draft` states correctly.

When you finish, paste your `TodoApp.js` and `TaskItem.js` here and I'll review for correctness and edge cases. If you want, I can also generate starter files you can copy-paste into your project.