Todo List Web App – Requirements

# Requirements

- store a list

- add items

- mark items as done

- remove items

- edit items

Tech requirements:

- only update frontend if backend updates succeeded

**Tech discovery**

**Frontend (React)**

1. **React components**:
   * **TodoList**: Displays the list of tasks.
   * **TodoItem**: Represents individual items (editable, markable as done, removable).
   * **AddTodo**: Form to add new tasks.
   * **EditTodo**: Allows editing task details (optional).
2. **State management**: Use React's useState or useReducer to manage the state of tasks (e.g., completed or not, item list).
3. **Event handling**: Handle user actions like adding, editing, marking as done, and deleting tasks through React event listeners.

**Backend (TypeScript)**

1. **API with Express**:
   * **POST** /todos: Add a new todo item.
   * **GET** /todos: Retrieve all tasks.
   * **PUT** /todos/:id: Edit a todo item (mark as done, update details).
   * **DELETE** /todos/:id: Remove a todo item.
2. **Database**:
   * Use a simple database like PostgreSQL if scaling.
   * Define a **Todo** model with fields like id, text, isDone, and createdAt.
3. **Data validation**: Use libraries like zod or Joi to ensure valid data for requests.
4. **API testing**: Tools like Postman for testing backend APIs.
5. **Deployment**: You can deploy on services like Vercel (for frontend) and Fly.io/Heroku (for backend).

# [Done] Steps 1-3

To get started, here are the first three steps:

1. **Set Up the Frontend (React)**
   * Install Node.js if you don’t have it.
   * Create a React app:

npx create-react-app todo-app --template typescript

* + This will set up a React project with TypeScript support.

1. **Set Up the Backend (TypeScript)**
   * Initialize a Node.js + TypeScript backend:

mkdir todo-backend

cd todo-backend

npm init -y

npm install express typescript @types/express ts-node nodemon

* + Initialize TypeScript configuration:

npx tsc --init

* + Create a basic Express server in index.ts and set it up to listen on a port.

1. **Link Frontend and Backend**
   * Set up CORS in your Express backend to allow cross-origin requests:

npm install cors

* + In index.ts:

import cors from 'cors';

app.use(cors());

These steps will get your development environment ready. You can then start implementing features (task management, API endpoints) step by step.

# Steps 4-6

1. **Create the Todo Model and Database Setup**:
   * [Done] Choose a database (e.g., SQLite or PostgreSQL).
     + PostgreSQL as it is used at Klarna
   * [Done] Install PostgreSQL
   * [Done] Create todo\_app database
   * [Done] Set up database connection
   * [Done] Create Sequelize file
   * [Done] Create a Todo model/schema with fields: id, text, isDone, and createdAt.
   * [Done] Sync model with database
   * [Done] Test database connection
   * [Done] Verify table in postgresql
2. **Implement API Endpoints**:
   * Set up your Express server with the following routes:
     + [Done] POST /todos to add a new todo item.
     + [Done] GET /todos to retrieve all tasks.
     + [Did not work - Skipped] PUT /todos/:id to edit a todo item.
     + [Did not work - Skipped] DELETE /todos/:id to remove a todo item.
   * Ensure you handle requests and responses correctly.
3. **Build Frontend Components**:
   * Create React components for:
     + TodoList to display the list of todos.
     + TodoItem to show each item with options to mark as done, edit, or delete.
     + AddTodo form to submit new tasks.
   * Use useEffect to fetch todos from the backend when the app loads.

This will set a solid foundation for your todo app!

# [Done] Steps 7-9

**High-Level Overview**

1. **Organize Component Structure:** Create separate files for your main components (TodoList, TodoItem, AddTodo) to keep your code modular.
2. **Implement TodoList Component:** Build a component that fetches the todo list from your backend (using useEffect) and renders each item using TodoItem.
3. **Implement AddTodo Component:** Create a form component with event handling (using useState) to add new todos via a POST request.

**Detailed Steps**

1. **[Done] Organize Component Structure**
   * **What:**
     + Inside your src folder, create a subfolder (e.g., components).
     + Create files: TodoList.tsx, TodoItem.tsx, and AddTodo.tsx.
   * **Why:**
     + Separating components improves readability, reusability, and maintainability.
     + Modular structure makes debugging and future enhancements easier.
2. **[Done] Implement TodoList Component**
   1. **What:**
      1. In TodoList.tsx, import React, useState, and useEffect.
      2. Define a Todo interface (e.g., id, text, isDone, createdAt).
      3. Use useEffect to fetch the todo list from your backend endpoint (GET /todos).
      4. Map over the fetched list to render a TodoItem for each todo.
   2. **Why:**
      1. Fetching todos on component mount ensures your UI reflects the current state from the backend.
      2. Displaying each todo using a dedicated component (TodoItem) makes it easier to later add functionality (mark done, edit, delete).
3. **[Done] Implement AddTodo Component**
   1. **What:**
      1. In AddTodo.tsx, create a form with an input field and a submit button.
      2. Use useState to manage the input value.
      3. On form submission, prevent the default behavior and send a POST request to your backend (POST /todos) with the new todo details.
      4. Optionally, update the parent component’s state or trigger a re-fetch to display the new todo.
   2. **Why:**
      1. A dedicated AddTodo component encapsulates the functionality for creating new tasks.
      2. Handling events (input change and form submit) ensures a responsive UI and keeps your state synchronized with backend data.

These steps provide a clear path to build your frontend components while integrating the basic required functionalities.

# Steps 10-12

**High-Level Steps**

1. **Complete the Missing API Endpoints:**  
   – Finish implementing the PUT (edit) and DELETE (remove) routes on your backend so every change is first confirmed by the server.
2. **Improve Frontend–Backend Integration & Error Handling:**  
   – Update your React components to only change state when the backend confirms success and display errors otherwise.
3. **Deploy to Production:**  
   – Build production bundles for both frontend and backend and deploy them (e.g., React on Vercel and Node/Express on Heroku/Fly.io) with proper environment settings.

**Low-Level Detailed Sub-Steps**

1. **Complete Missing API Endpoints (PUT & DELETE):**
   * In your Express server (e.g., in your index.ts or dedicated routes file):
     + **PUT /todos/:id:**  
       • Validate incoming data (consider using a library like zod or Joi).  
       • Update the matching record in the database (using Sequelize).  
       • Return a JSON response indicating success or a detailed error.
     + **DELETE /todos/:id:**  
       • Check that the provided ID exists.  
       • Remove the record from your database and respond with a success message.
   * Use a tool like Postman to manually test these endpoints before integrating with the frontend.
2. **Improve Frontend–Backend Integration & Error Handling:**
   * In your React components (like TodoItem.tsx and AddTodo.tsx):
     + Modify your fetch calls so that you check the response’s status (e.g., if (!res.ok) throw new Error(...)).
     + Only update your UI (or state) when the backend confirms success.
     + Optionally, display user-friendly error messages if an API call fails.
   * This ensures that your frontend state only changes when the backend update actually succeeds.
3. **Deploy to Production:**
   * **Backend Deployment:**
     + Choose a platform (for example, Heroku or Fly.io).
     + Prepare your project for production (set environment variables, configure CORS appropriately, add a Procfile if needed).
     + Deploy your built Node.js/TypeScript server following the chosen platform’s guide.
   * **Frontend Deployment:**
     + Run a production build (npm run build) in your React app.
     + Deploy the build folder to a hosting provider like Vercel.
     + Ensure the frontend is correctly configured to point to your live backend (using environment variables or a config file).

Following these steps will get you from a locally tested app to a live production version where both the backend and frontend work together reliably.

# Appendix

## Appendix A – Sequelize as an Object-Relational Mapping

Object-Relational Mapping (ORM) = programming technique that allows developers to interact with a relational database using object-oriented programming languages

Sequelize is an ORM, but not an object database. Key difference:

* Sequelize: ORM for relational databases.
* Object Database: Directly stores data as objects, with no conversion to tables.

Nice side-node: Some folks at Klarna use Sequelize to construct PostgreSQL queries ([Klarna Engineering Blog](https://engineering.klarna.com/node-js-integration-testing-with-ease-fab5f8d29163)).

## Appendix B: Box & Remote Control Metaphor for useState<Todo[]>([])

**useState<Todo[]>([])**

Imagine you have a **box** where you keep a list of sticky notes (your to-do list).

🟦 **Box (todos)** → Holds all the sticky notes (the list of todos).  
🎛 **Remote (setTodos)** → Lets you **add, remove, or update** the sticky notes inside the box.  
📦 **Starting Condition (useState<Todo[]>([]))** → The box **starts empty** (no sticky notes).

Plus:

### **🗂 The Compartment Organizer (**Todo[]**) → Defines What Fits Inside**

* Just like a **storage organizer** only fits specific-sized items (e.g., sticky notes, books, or tools), Todo[] ensures that only objects with the correct shape (id, text, isDone, createdAt) are stored inside the state.