**✅ Step 1: Backend Setup (Node.js + Express + MongoDB)**

1. **Create the backend folder** inside your project and initialize it as a Node.js project.
2. **Install all required libraries** like Express (for server), Mongoose (for database), JWT (for auth), Bcrypt (for password hashing), etc.
3. **Set up development scripts** so you can run the server easily using a shortcut command like npm run dev.
4. **Create a proper folder structure** to organize your code — like folders for models, routes, controllers, etc.
5. **Configure environment variables** using a .env file to securely store the MongoDB URI and JWT secret.
6. **Write basic server code** that starts the Express server and connects it to MongoDB.
7. **Run the server** to confirm that it's working and successfully connected to the database.
8. We'll build a secure system where users can **register and log in**, and you'll use **JWT tokens** to manage sessions.

 **Created a User model**  
Defined a schema to store user details like name, email, and hashed password in MongoDB.

 **Set up authentication routes**  
Added routes for /register and /login using Express, so users can sign up and sign in.

 **Built controller logic**

* On **register**, the password is hashed and the user is saved to the database.
* On **login**, the password is verified and a JWT token is issued if credentials are valid.

 **Connected the routes to the main server**  
Used middleware to mount the auth routes, making them active at /api/auth.

**🔹 3.1 – Task Model**

* Created a Mongoose schema (Task.js) to define what a task looks like:
  + title, description
  + status (Todo, In Progress, Done)
  + priority (Low, Medium, High)
  + assignedTo (references a user)
  + timestamps (auto-created/updated)

**🔹 3.2 – Task Routes**

* Created Express routes (taskRoutes.js) to handle:
  + POST /api/tasks → create a new task
  + GET /api/tasks → fetch all tasks
  + PUT /api/tasks/:id → update a task
  + DELETE /api/tasks/:id → delete a task

**🔹 3.3 – Task Controller**

* Added logic in taskController.js:
  + Create a task using data from request
  + Fetch all tasks and populate assigned user
  + Update tasks by ID
  + Delete tasks by ID

**🔹 3.4 – Connected the Task API to the App**

* Imported and registered taskRoutes in server.js
* All task operations now available via /api/tasks

**🎯 Outcome:**

You now have a **fully functional task API** in place — users can:

* Add tasks
* Assign them
* Edit status/priority
* Delete them
* ...and your database keeps it all organized!

**✅ Step 4: Real-Time Task Sync – Summary**

This step added **live collaboration** to your to-do board — so all users instantly see task changes (like Trello does).

**🔹 What You Did:**

1. **Enabled real-time communication using Socket.IO**  
   You updated your backend to support WebSocket connections so it can talk to connected clients (React apps) in real-time.
2. **Created a socket.js helper on the backend**  
   This allowed your backend controllers (like taskController.js) to emit real-time events like taskCreated, taskUpdated, and taskDeleted.
3. **Emitted events from the backend**  
   Whenever a task was created, updated, or deleted, you emitted a corresponding event to all connected clients.
4. **Set up Socket.IO in your React frontend**  
   You created a socket.js file in your React app that connects to your backend server.
5. **Listened for events in the browser**  
   You used useEffect in React to listen for those task events and log them in the browser console.

**🎯 Final Result:**

Now when a user performs an action like adding or updating a task:

* The backend saves the change to MongoDB
* Then emits a real-time event via WebSocket
* The frontend (anyone connected) instantly receives the update

This lays the foundation for building your **live-updating Kanban board** 🧠⚡

**✅ Step 5: Task Board UI – Summary**

In this step, you transformed your backend-powered app into a fully interactive, Trello-like frontend experience using React and real-time events.

**🔹 5.1 – Created Component Structure**

You designed the board layout using 3 reusable components:

* **TaskBoard** (main board)
* **Column** (represents Todo/In Progress/Done)
* **TaskCard** (represents a single task)

Each column displayed tasks based on their status.

**🔹 5.2 – Fetched Tasks from Backend**

* You used axios to fetch tasks from your Node.js backend.
* Tasks were stored in React state and rendered under the correct column based on their status.

**🔹 5.3 – Added Drag-and-Drop Functionality**

* You installed and used a drag-and-drop library.
* This allowed users to move tasks between columns.
* On every drag-and-drop, the task’s status was updated both in the **UI** and the **MongoDB database**.

**🔹 5.4 – Enabled Real-Time Sync for UI**

* You used Socket.IO in the frontend to listen for:
  + Task creation
  + Task updates (e.g., status change)
  + Task deletion
* The board automatically refreshed itself with the latest changes made by **any user in real time**, without needing a manual refresh.

**🎯 End Result:**

You now have a **real-time collaborative task board** that:

* Feels responsive and interactive
* Automatically syncs across multiple users
* Visually represents task progress (Todo → In Progress → Done)

**🎯 What is "Smart Assign"?**

Instead of choosing who to assign a task to, the system **automatically picks the best person**:

🔁 It assigns the task to the user who currently has the **fewest active (incomplete) tasks**.

**💡 How It Works (Behind the Scenes)**

When you click the “Smart Assign” button (or call the smart-assign API):

**✅ Step 1: It checks the users**

* It goes through every user in the system

**✅ Step 2: It counts each user's active tasks**

* For each user, it checks how many tasks they already have that are **not marked as Done**

**✅ Step 3: Picks the lightest workload**

* It selects the user with the **lowest task count**

**✅ Step 4: Assigns the task**

* The selected user is assigned to the task (just like setting assignedTo manually)

**✅ Step 5: Activity Log + Real-Time Update**

* A message is added to the **Activity Log** (e.g., “Task was smart-assigned to Gayathri”)
* The update is sent **live** to all connected users

**👀 Why It's Useful**

* Saves time — no need to choose manually
* Distributes work fairly
* Makes your app smarter — like an assistant doing the thinking
* Impresses interviewers — shows custom logic and teamwork thinking

**Challenge 1: Smart Assign**

**Goal:**  
Automatically assign a task to the user with the **least workload** (fewest active tasks).

**What you did:**

* When "Smart Assign" is triggered, the system:
  + Checks all users
  + Counts how many tasks each user has that are **not marked as Done**
  + Picks the user with the **least number of active tasks**
  + Assigns the task to that user
* It then emits a real-time event and logs the assignment.

**Why it's useful:**  
Distributes tasks fairly and reduces manual effort.

**✅ Challenge 2: Conflict Handling**

**Goal:**  
Prevent one user’s changes from overwriting another’s edits if they’re both working on the same task at the same time.

**What you did:**

* Each task carries a lastModified timestamp.
* When a user tries to update a task:
  + The system compares their version’s timestamp with the one in the database.
  + If their version is **older**, a **conflict is detected**.
  + The backend responds with both versions:
    - The **current task**
    - The **user’s outdated version**
* No update is saved until the conflict is resolved.

**Why it's useful:**  
Avoids accidental data loss in a real-time multi-user system.

**✅ Challenge 3: Validation**

**Goal:**  
Ensure task titles are meaningful and consistent.

**What you did:**

* When creating or updating a task:
  + The system checks that the title is **not already used** by another task.
  + It also checks that the title **does not match any reserved column names** like "Todo", "In Progress", or "Done".
* If either rule is violated, the system **rejects the request** with an appropriate message.

**Why it's useful:**  
Prevents confusion and keeps the board organized.