ft_transformers

Day 1

Phase 1: The Setup

i install Node.js (LTS Version) and "Native" Docker Engine

i was facing a problem installing Docker Desktop on my endeavouros VM because Docker Desktop tries to create *its own* VM to run containers. since i am *already* inside a VM, my computer is blocking this virtualization. and one more thing is Native Docker runs directly on my Linux kernel. It is lighter, faster, and uses less RAM than Docker Desktop. Summary: on Windows or Mac, developers *need* Docker Desktop to get a Linux environment. On Linux, i am the environment. i don't need the Desktop app.

Phase 2: Create my Service

Step 1: Generate the Project

i didn't just open a blank file and start typing. i commanded a tool (the CLI) to build a complete, working skeleton for me instantly

`npm i -g @nestjs/cli`

What it is: CLI stands for Command Line Interface

a specific tool set made by the NestJS team. This tool knows exactly how to build NestJS applications. i installed it "globally" (`-g`), which means this tool is now available everywhere on my computer, not just in this folder

`nest new kanban-service`

It will ask which package manager to use. we Choose npm

NPM stands for Node Package Manager
NPM lets me download code written by other people (like NestJS, React, or Prisma) so i don't have to write everything from scratch
The CLI automatically created a standard folder structure for me. It made the `src`

folder, the `test` folder (for checking errors), and configuration files. This ensures my project looks exactly like every other NestJS project in the world

i Generated "Boilerplate" Code. Boilerplate is the standard, boring code that every project needs just to start running. The CLI wrote the first few files for me automatically:

- `main.ts` : The Entry Point
- `app.module.ts` : The Root Module (the brain that organizes the code).
- `package.json` : The Manifest (the ID card that lists what our project needs).

i installed the Dependencies (bundles of code written by other people that my project needs to work). At the very end of the setup, the CLI ran a hidden command to download thousands of small code files and put them in a folder called `node_modules` it was downloading the "engine" parts so i don't have to build them myself.

Step 2: Change the Port

change the port from 3000 to 3003 inside `kanban-service/src/main.ts`

by doing this i changed the Port to give my application a unique door number (3003). This prevents a Port Conflict with other running applications, ensuring my Kanban service has its own dedicated space to listen for requests.
to run it just type `npm run start:dev` in the terminal.

Phase 3: Dockerize It (The Contract)

Step 4: Create the `Dockerfile`

```
# Start with a lightweight Node.js operating system
FROM node:20-alpine

# Install OpenSSL (The missing library)
RUN apk add --no-cache openssl

# Create a folder inside the container
WORKDIR /app
```

```
# Copy the "shopping list" (package.json) wedo this to so we don't re-installin
g the libraries next time
COPY package*.json ./

# reads the package.json file and downloads all the libraries into the node_mo
dules folder inside the container
RUN npm install

# Copy the rest of our code
COPY . .

# Generate the Prisma Client inside the container
RUN npx prisma generate

# Build the app (Node.js cannot run TypeScript files directly so we translate T
ypeScript to JavaScript inside a dist folder).
RUN npm run build

# Tell the container to expose port 3003
EXPOSE 3003

# The command to start the app (Run my code inside disk folder)
CMD ["npm", "run", "start:prod"]
```

i use my computer to *create and edit* the files because it's easier. i use the container only to *run* the files
We need to tell Docker: *"Ignore my local trash files."* This stops it from freezing at Step 5.

1. Create a new file in our root folder called `.dockerignore`
2. Paste this inside:

```
node_modules
dist
Dockerfile
```

```
.git
.env
```

Step 5: Create the `docker-compose.yml`

```
version: '3.8'

services:
  kanban-app:
    build: .
    ports:
      - "3003:3003"
    environment:
      - DATABASE_URL=postgresql://user:password@kanban_db:5432/kanban_
db?schema=public
    depends_on:
      - kanban_db

  kanban_db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: kanban_db
    ports:
      - "5434:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

Phase 4: The Database

- Before Phase 4: If i started my NestJS app and created a task, it would exist in the computer's RAM (short-term memory). If i restarted the server, the task would vanish.
- After Phase 4: my application will have "Long-Term Memory." i can create a task today, turn off my computer, come back next week, and the task will still be there.

before start explaining what we are doing exactly on this phase i should explain what is exactly postgresql (or postgres)

- Technically: It is a software program that saves data into tables (rows and columns), just like Excel.
- Practically: When our users create a Task, the code sends it to Postgres to put it on a shelf. When we turn off the computer, Postgres keeps that data safe on our hard drive.

Step 6: Install Prisma

Prisma play the translator role. my code speaks TypeScript, but my database speaks SQL. Prisma translates between them so i don't have to write raw SQL queries like `SELECT * FROM ...`

- `npm install prisma --save-dev`

It downloads the Prisma CLI (Command Line Interface). i use it to create database tables and generate files. i use `--save-dev` because we use prisma cli to create database tables and generate files. we need it *while we are coding*. It tells the computer: *"Use this tool to build the project, but when we go to production (the finished app), throw it away to save space." (if we sell a chair we* do not put the electric drill,the glue, the screws,... inside the box,The customer just wants the chair, they don't need the tools used to build it*)*

- `npx prisma init`

this command automatically created two critical things in the project that didn't exist before:

1. a folder named `prisma` containing a file called `schema.prisma`.
2. a file named `.env` in our root folder.

`npx` stands for Node Package Execute. It allows us to run the Prisma tool we just installed locally without needing to install it globally on the entire computer. `npm install -g prisma` this installs it on the *entire computer*. we can run it from any folder. but `npm install prisma --save-dev` installs it ONLY inside the `node_modules` folder of the current project. If we open a new terminal in a different folder, the command `prisma` will not work. without `npx` we should run this command `./node_modules/.bin/prisma init` to tell the terminal were is prisma. With `npx`: we type `npx prisma init` . npx automatically looks inside `node_modules`, finds the hidden `prisma` tool, and runs it. It is a "finder" helper.

A. `prisma/schema.prisma` (The Blueprint)

This is the most important file in the entire database phase.

- What it is: The Declarative Schema.
- Function: It is where we define our "Models" (e.g., "I want a User", "I want a Task").
- Why we need it: This is the Source of Truth. Prisma reads this file to know how to build our SQL tables automatically. Without it, Prisma is blind.

B. `.env` (The Secrets)

- What it is: Environment Configuration.
- Function: It holds sensitive variables, specifically the `DATABASE_URL`.
- Why we need it: Hard coding passwords inside the code is a security risk. the `.env` file keeps the password safe on the computer. Prisma looks here to find the address of the database (the port `5434` we set up earlier).

Step 7: Define the "Task" Model

in this step we setup prisma. it take `schema.prisma` and translate it into SQL code so postgreSQL understand it and translate it into typescript so my code understand it. `schema.prisma` is the Universal Translator. we are writing in PSL (Prisma Schema Language) inside it. it is not a programming language it is a DSL (Domain Specific Language) It was invented by the Prisma team specifically to be Human Readable.

How does Prisma "read" this file ?

1. Prisma reads my `model Task` and converts it into SQL commands (`CREATE TABLE...`) to send to the PostgreSQL container. This physically builds the table in the database.
2. our code runs in TypeScript. It has no idea that a database exists. It doesn't know that a table named "Task" exists. It doesn't know that a "Task" has a "title". Prisma reads `model Task` and writes TypeScript code into the `node_modules` folder.
   - It creates a TypeScript type called `Task`.
   - It creates functions like `createTask`, `findTask`, `deleteTask`.

   So, this one file is used to **build our database** AND **write my code** automatically (no SQL needed inside TypeScript code)

this is our data structure inside prisma/schema.prisma:

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model Task {
  id          Int      @id @default(autoincrement()) // "id (PK)"
  project_id  Int      // "project_id" (No relation, just a number)
  title       String   // "title"
  description String?  // "description" (The ? means it can be empty)
  status      String   // "status" (TODO, DONE)
  position    Int      // "position"
  assigned_to Int?     // "assigned_to" (User ID)
  created_by  Int      // "created_by" (User ID)
  due_date    DateTime? // "due_date"
  created_at  DateTime @default(now())
  updated_at  DateTime @updatedAt
}
```

▼ Part 1: The Setup (Connecting the Lines)

- `datasource db` : defining a **Data Source** and named it `db`.
- `provider = "postgresql"` : This tells Prisma which language to speak. Since we are using a Postgres database, Prisma needs to speak the "Postgres dialect" of SQL. If we were using MySQL or MongoDB, we would change this word here.
- `url = env("DATABASE_URL")` : This is the **Phone Number** of our database.
    - `env(...)` : This function tells Prisma, *"Don't look here for the password. Go look in the secure `.env` file."*
    - It grabs the URL (which contains the username, password, port `5434`, and database name) so it knows where to connect.

▼ Part 2: The Tool Maker (The Generator)

- `generator client` : telling Prisma, *"I want we to build me a tool."*
- `provider = "prisma-client-js"` : we are specifying *which* tool. *"Build me a JavaScript/TypeScript client."*
    - This is the instruction that tells Prisma to read our model and write that helpful TypeScript code into `node_modules` folder (the "Menu" we talked about earlier).

▼ Part 3: The Blueprint (The Task Model)

- `model Task` : This defines a new **Entity**. In our code, it will be a Class/Object. In the database, it will be a Table.

A. The Primary Key

- `id        Int     @id @default(autoincrement())`
    - `id` : The name of the column.
    - `Int` : The data type (Integer/Number).

    - `@id`: (symbol starting with `@`) marks this field as the **Primary Key**. It means this is the unique ID card for every task.
    - `@default(autoincrement())`: This is an automation rule.
        - `default`: If we don't provide a value, use this rule.
        - `autoincrement()`: Start at 1, then 2, then 3... we never have to manually number our tasks.

B. The Data Fields

- `project_id Int`: Just a number. This links the task to a specific project (Project #1, Project #2).
- `title String`: Text. The name of the task.
- `description String?`:
    - `String`: Text.
    - `?` **(Question Mark)**: This means **Nullable** (Optional). A task *can* exist without a description. If i remove the `?`, the database will force me to write a description every time.
- `status String`: Text. e.g., "TODO", "IN_PROGRESS".
- `position Int`: A number used for sorting. E.g., The top task is position `0`, the next is `100`.

C. Foreign Keys

- `assigned_to Int?`: A number representing a User ID. It is **Optional (`?`)** because a task might not be assigned to anyone yet.
- `created_by Int`: A number representing a User ID. It is **Required** because *someone* must have created the task.

D. The Time Travel Fields

- `due_date DateTime?`: A calendar date and time. It is **Optional (`?`)** because not all tasks have deadlines.
- `created_at`: Records when the task was born.

    - **`@default(now())`**: The moment the row is created, the database looks at its own clock and stamps the time. we never touch this field.
- **`updated_at`**: Records when the task was last changed.
    - **`@updatedAt`**: This is a special Prisma trigger. If we change the `title` of a task a week from now, Prisma will automatically update this field to that new time.

now we wrote the `schema.prisma` file, but we didn't tell Prisma to generate the actual TypeScript code from it. The "header file" (`@prisma/client`) doesn't exist yet.

1. Install the runtime library (The Object Code):

   we installed the "Dev" tools (`prisma`), but we forgot the "Runtime" library that runs when the app is live. we install the specific version that works perfectly with standard code

   ```
   npm install prisma@5.22.0 @prisma/client@5.22.0 --save-exact
   ```

2. Generate the Code (The Compiler):

   This reads our `schema.prisma` file and *writes* the TypeScript files for us automatically

   ```
   npx prisma generate
   ```

Step 8: Connect the Wires

int this step i change the DATABASE_URL inside .env file to match the docker password. when i edited this file, i was configuring the Database Connection String. This single line of text is the "bridge" that allows my NestJS application (running on my laptop) to find and talk to the PostgreSQL database (running inside the Docker container).

```
DATABASE_URL="postgresql://user:password@localhost:5434/kanban_db?schema=public”
```

1. `postgresql://` (The Protocol)
    - **Technical Term:** Protocol Scheme.
    - **Explanation:** This tells my application *how* to talk. It is like choosing a language. It says: "We are not talking to a website (`http://`), we are talking to

a Postgres database."

2. `user:password` (The Credentials)
   - **Technical Term:** Authentication Token.
   - **Explanation:** When the app knocks on the database connection, the database checks these words. If they don't match what is in the `docker-compose.yml`, the database will reject the connection with an "Authentication Failed" error.
3. `@localhost` (The Host)
   - **Technical Term:** Hostname.
   - **Explanation:** This tells the app *where* the database lives.
     - Since i are running the app on my own computer (EndeavourOS), `localhost` means "this computer."
4. `:5434` (The Port)
   - **Technical Term:** Port Forwarding Target.
   - **Explanation:**
     - my app lives on my laptop, so it must call the "laptop number" (**5434**) to reach the container
5. `/kanban_db` (The Database Name)
   - **Technical Term:** Database Instance.
   - **Explanation:** A single Postgres server can hold many different projects (Kanban, User, Chat). This specifies exactly *which* project folder to open.

Day 2

In the early days, different languages (like a C++ server and a Java client) couldn't talk to each other easily because they stored data in memory differently (e.g., how many bytes is an `int`?). JSON was invented as a universal language for data. It is just a text string that looks like this:

```
{
  "title": "Fix the server",
  "status": "PENDING",
  "priority": 1
}
```

Every language (C++, Python, JavaScript) has a library to read this text and convert it into variables.

so it is purely a text format used to store and transport data. It is not a programming language. It cannot calculate `1 + 1`. It just sits there, holding information.

JavaScript was the original language built to run inside web browsers (like Chrome). JavaScript is loosely typed (we don't declare `int` or `string`), which makes it very annoying. as web apps got huge (like Facebook or Google Docs), the lack of types in JavaScript became a nightmare. Microsoft invented TypeScript to add Static Typing (like C++) to JavaScript. TypeScript is just like C++. It has loops, functions, and most importantly, Types (`string`, `number`, `boolean`). matter fact, the computer cannot run TypeScript directly. It compiles the TypeScript code into JavaScript, just like `gcc` compiles C code into assembly/machine code.

Step 1: Define the "Header File" (The DTO)

In C++, before we write a function, we define the `struct` or `class` in a `.h` file so the compiler knows what the data looks like. in NestJS, we call this a DTO (Data Transfer Object). We need to tell the code: *"Expect the user to send a Title and a Description."*

1. we create a file: `src/tasks/dto/create-task.dto.ts`.
   in this step i had one big question "If JSON is just data (text), why do we need a TypeScript file (`create-task.dto.ts`) for it?"

   so i try to link it with c++ because it is my closest language. in C++, if we expect data from a user, we must define a `struct` or a `class` first so the compiler knows how much memory to allocate. the `.ts` file (DTO) is exactly this `struct`

When the user sends that JSON text to our server, our code doesn't know what it contains yet. It's just a text. We write the DTO (Data Transfer Object) in TypeScript to tell the program: "When we receive that JSON text, I expect it to look like this Class"

so in summary in `create-task.dto.ts`, we are writing TypeScript code that defines the shape we expect the JSON data to have.

2. inside `create-task.dto.ts` we have:

```
export class CreateTaskDto {
  title: string;

  description?: string;

  projectId: number;
}
```

Step 2: Write the Logic (The Service)

we gonna create the function that create a TODO task for us so when the browser send a request for making a TODO task we call this function and it speak with the DataBase using prisma and create a task data inside the database.

in C++, we would write a function like `void createTask(Task t) { database.save(t); }`.
In NestJS, we put these functions inside a Service Class.

in this step, we are writing a single **Function Definition**.

Technically, we are defining a method inside a class. When this function is eventually called (at Runtime), the following chain of events happens in the CPU and Memory:

1. **Function Call:** The function `create()` starts executing.
2. **Argument Passing:** It receives the `CreateTaskDto` (the object we just defined) as an argument.

3. **Database Driver Invocation:** The function calls `this.prisma.task.create()`. This is the "driver" that knows how to talk to the database.
4. **Serialization:** The driver converts our TypeScript object (`{ title: "Hi" }`) into a database command (`INSERT INTO tasks...`).
5. **I/O Wait (Async):** The function pauses and waits for the hard drive to write the data.
    a. **Async/Await:**
        - Sending data to a database is an **I/O Operation** (Input/Output). It is slow compared to CPU processing.
        - In C++, a standard function would block (freeze) the thread until the database confirms the save.
        - In Node.js/TypeScript, we use `async/await`. We tell the CPU: *"Send this data to the DB, and while we wait for the confirmation, go handle other users' requests."*
6. **Return:** Once the database confirms the write, the function returns the newly created data (including the generated `id`) to the caller.

1. Run this command to generate the files automatically (inside the project folder)

   `nest g service tasks`

   The command `nest g service tasks` does **two** things.

    1. It creates the file `tasks.service.ts` with the boilerplate code
    2. It Registers the service in the `tasks.module.ts`
2. and we change `src/tasks/tasks.service.ts` into:

```
import { Injectable } from '@nestjs/common';
import { PrismaClient } from '@prisma/client';
import { CreateTaskDto } from './dto/create-task.dto';

const prisma = new PrismaClient();

@Injectable()
```

```
export class TasksService {
  async create(createTaskDto: CreateTaskDto) {
    return await prisma.task.create({
      data: {
        title: createTaskDto.title,
        description: createTaskDto.description,
        project_id: createTaskDto.projectId,
        status: 'TODO',
        position: 1,
        created_by: 1
      },
    });
  }
}
```

Step 3: The Entry Point (The Controller)

In a web server, a Controller is a class responsible for handling incoming HTTP requests and sending back HTTP responses. we are creating a Controller that will listen for a specific type of request.

1. **The Trigger:** A user (or browser) sends an **HTTP POST** request to the URL `http://localhost:3000/tasks`.
2. **The Match:** The Framework (NestJS) looks at all our Controllers. It sees that `TasksController` is assigned to listen to `/tasks`.
3. **The Handler:** Inside the Controller, we have a specific function decorated with `@Post()`. This function "catches" the request.
4. **The Extraction:** The Controller looks inside the request **Body** (the JSON data) and converts it into a `CreateTaskDto` object.
5. **The Delegation:** The Controller calls `this.tasksService.create(dto)`. (It passes the work to the Service we just wrote).
6. **The Response:** When the Service finishes, the Controller receives the result and sends it back to the user with a **201 Created** status code.

we gonna do this by modifying `src/tasks/tasks.controller.ts` we will tell it:

1. **Listen** for a `POST` request at `/tasks`.
2. **Expect** the data to look like `CreateTaskDto`.
3. **Call** the `tasksService.create()` function when that happens.

1. Run this command:

   `nest g controller tasks`

   The command automates two steps:

   1. It makes the `tasks.controller.ts` file.
   2. It opens `src/tasks/tasks.module.ts` and adds `TasksController` to the `controllers: [...]` list.
2. open `src/tasks/tasks.controller.ts`

```
import { Controller, Post, Body } from '@nestjs/common';
import { TasksService } from './tasks.service';
import { CreateTaskDto } from './dto/create-task.dto';

@Controller('tasks')
export class TasksController {
  constructor(private readonly tasksService: TasksService) {}

  @Post()
  create(@Body() createTaskDto: CreateTaskDto) {
    return this.tasksService.create(createTaskDto);
  }
}
```

Step 4: Test It

we don't need a frontend to test this. we can use `curl`

1. Make sure our app is running: `npm run start:dev`
2. Make sure our Docker database is running: `docker-compose up -d`
3. Step 3: Push the Tables (Create the Furniture)

`docker compose exec kanban-app npx prisma db push`

4. paste this command:

   `curl -X POST http://localhost:3003/tasks -H "Content-Type: application/json" -d '{"title": "I am connected!", "projectId": 1, "project_id": 1}'`

If the JSON response like this: `{"id": 1, "title": "Learn TypeScript", ...}` we just built our first REST API endpoint

Day 3

In a normal website (HTTP), the Client (the browser) asks a question, the Server (NestJS) answers, and then they stop talking.

In a WebSocket, we perform a **Handshake**. This is a special "Hello" that keeps the connection open. Instead of hanging up the phone, they stay on the line forever. This way, the Server can send a Payload (a packet of data) to the Client whenever it wants, without being asked first. This is called Real-Time Bi-directional Communication.

The goal for this Day is to transform our server from a standard "request-response" model into a "live" system, ensuring that when one user moves a task on the Kanban board, everyone else sees it happen instantly.

step 1: Install the "Walkie-Talkie" (The Libraries)

To make our server talk in real-time, we need to add two specific sets of code called Libraries. We use `npm` (Node Package Manager) to download them into our project.
use this command:

`npm install @nestjs/websockets @nestjs/platform-socket.io socket.io`

By running this command, we are adding the Dependencies to our `package.json` file. This tells the computer: *"From now on, this project is capable of keeping a line of communication open at all times."*

1. @nestjs/websockets: This is the Module. By default, NestJS doesn't know how to do WebSockets. This library gives NestJS the instructions it needs to understand how to handle persistent connections.

2. socket.io: This is the Engine. While "WebSocket" is the name of the technology, `socket.io` is the actual machine that does the heavy lifting. It manages the tiny details of how data travels back and forth between the computer and the browser.
3. @nestjs/platform-socket.io: This is the Adapter. It allows the NestJS "brain" to talk perfectly to the `socket.io` "engine." It makes sure they speak the exact same language.

Step 2: Create the Gateway

In NestJS, we don't use Controllers for WebSockets. Instead, we use a **Gateway**.

In the world of WebSockets, a Gateway is a special type of Class decorated with `@WebSocketGateway()`. While a Controller handles standard web addresses (URLs), a Gateway handles Socket Connections.

1. The Decorator (`@WebSocketGateway`): In code, a "Decorator" is like a Label. When we put this label on a file, we are telling the NestJS system: *"This specific file is not for normal web pages. it is the boss of the WebSocket connections."* It tells the server to open a specific Port to listen for people trying to connect. When the NestJS compiler sees `@WebSocketGateway()`, it automatically writes the extra low-level code needed to turn our simple class into a network server.
2. The Instance: When the server starts, it creates an Instance of this Gateway. This is a living piece of memory that stays awake as long as the server is running. It sits there and waits.
3. The Server Member: Inside this Gateway, we define a Server Variable. This is the actual Socket.io Server. It is the "Master Controller" that has a list of every single person (Client) who is currently connected to our app.
4. Events: The Gateway is designed to handle Events. An Event is a specific message with a name. For example, if a user moves a task, the browser sends an event named `taskMoved`. The Gateway is the thing that "hears" that name and decides what code to run next.

we do this by running this command:

`nest g gateway tasks`

This creates `src/tasks/tasks.gateway.ts` file. We create this file so the server has a central place to manage "Events." Without a Gateway, the server wouldn't know which messages to listen for or how to send information back to the users instantly.

Step 3: Write the Connection Code

```
import {
  WebSocketGateway,
  WebSocketServer,
  OnGatewayConnection,
  OnGatewayDisconnect
} from '@nestjs/websockets';
import { Server, Socket } from 'socket.io';

// 1. We open the Gateway on Port 3003 (same as the app)
// 2. "cors: origin *" means "Allow anyone to connect" (Crucial for developme
nt!)
@WebSocketGateway({ cors: { origin: '*' } })
export class TasksGateway implements OnGatewayConnection, OnGatewayDi
sconnect {

  // This gives us access to the "Big Microphone" to shout to everyone
  @WebSocketServer()
  server: Server;

  // When a user connects (opens the website)
  handleConnection(client: Socket) {
    console.log(`Client connected: ${client.id}`);
  }

  // When a user closes the tab
  handleDisconnect(client: Socket) {
    console.log(`Client disconnected: ${client.id}`);
  }
}
```

This code is the "Receptionist." It welcomes users when they connect and prints a log message.

Step 4: Plug in the Radio

our server started the API (the Mailbox), but it completely ignored the WebSocket (the Walkie-Talkie). It doesn't even know it exists. This usually happens because when we generated the gateway, it wasn't automatically added to our Module file. We need to tell the `TasksModule`: *"Hey, we have a new Gateway. Please turn it on."*

1. open `src/app.module.ts`
2. Add the Gateway to "providers"

```
import { Module } from '@nestjs/common';
import { TasksService } from './tasks.service';
import { TasksController } from './tasks.controller';
import { TasksGateway } from './tasks.gateway'; // ⟵ Import this!

@Module({
  controllers: [TasksController],
  providers: [
    TasksService,
    TasksGateway // ⟵ Add this line!
  ],
})
export class TasksModule {}
```

Step 5: The Test

we can't use `curl` for WebSockets. We need a "Fake Frontend."

1. Create a file named `client.html`
2. Paste this code inside:

```html
<!DOCTYPE html>
<html>
<head>
   <title>WebSocket Test</title>
   <script src="https://cdn.socket.io/4.7.2/socket.io.min.js"></script>
</head>
<body>
   <h2>Status: <span id="status" style="color:red">Disconnected</span>
</h2>
   <script>
      // Connect to our NestJS server
      const socket = io('http://localhost:3003');

      socket.on('connect', () => {
         document.getElementById('status').innerText = "Connected! 🟢";
         document.getElementById('status').style.color = "green";
         console.log("I am connected with ID:", socket.id);
      });

      socket.on('disconnect', () => {
         document.getElementById('status').innerText = "Disconnected 🔴";
         document.getElementById('status').style.color = "red";
      });
   </script>
</body>
</html>
```

Step 6: The Moment of Truth

1. Make sure our local server is running.

   `npm run start:dev` (Make sure Docker is only running the DB in this case, or we'll get a port conflict. for Docker to runs only the DB: `docker compose up kanban_db -d`).

2. Double-click `client.html` to open it in Chrome/Firefox.

we should see:

- **In the Browser:** The text turns **Green**.
- **In our Terminal:** we see `Client connected: xxxxxxxx`.

Step 7: The Broadcast

Right now, the connection is open, but nobody is saying anything. we need to make it so that when we create a task via `curl` (HTTP), the server instantly tells the Browser (WebSocket) about it.

a. give our Gateway a function that sends a message to everyone. so we should add a function inside `src/tasks/tasks.gateway.ts` :

```
broadcastTaskCreated(task: any) {
    this.server.emit('task:created', task);
  }
```

b. Connect the Controller to the Gateway. now we need to tell the Controller (which handles the POST request) to use the Gateway. open `src/tasks/tasks.controller.ts`. we need to change the constructor to "Inject" the gateway, and then call it.

```
import { Controller, Post, Body } from '@nestjs/common';
import { TasksService } from './tasks.service';
import { CreateTaskDto } from './dto/create-task.dto';
import { TasksGateway } from './tasks.gateway'; // ⟵ 1. Import this

@Controller('tasks')
export class TasksController {
  constructor(
    private readonly tasksService: TasksService,
    private readonly tasksGateway: TasksGateway // ⟵ 2. Inject the
Gateway
  ) {}

  @Post()
  async create(@Body() createTaskDto: CreateTaskDto) {
    // 1. Save to Database (The Old Way)
    const task = await this.tasksService.create(createTaskDto);

    // 2. Shout to everyone (The New Way)
    this.tasksGateway.broadcastTaskCreated(task);

    return task;
  }
}
```

c. Update the fake frontend

```
// ... socket.on('connect') ...

        socket.on('task:created', (data) => {
            console.log("New Task Received:", data);

            const msg = document.createElement('div');
            msg.innerText = `🆕 New Task: ${data.title} (ID: ${data.id})`;
            document.body.appendChild(msg);
        });
```

Recap for Day 1, Day 2 and Day 3

we built a real time, containerized microservice.

- **Day 1: The Setup (The Garage)**
  - [x] ~~Installed NestJS, Docker.~~
  - [x] ~~Designed the Database Schema (User, Project, Task) using Prisma.~~
- **Day 2: The Logic (The Engine)**
  - [x] ~~Created the DTOs (The Header Files) to validate input.~~
  - [x] ~~Built the Service (The Logic) to talk to the Database.~~

- [x] ~~Built the Controller (The Receptionist) to handle HTTP requests.~~

- **Day 3: The Voice (Real-Time)**
  - [x] ~~Installed WebSockets (The Walkie-Talkie).~~
  - [x] ~~Built a Gateway to shout "Task Created!" to all connected users.~~
  - [x] ~~Connected a "Fake Frontend" (`client.html`) to prove it works.~~

another think is.right now if we upload our project to GitHub, **everyone in the world** will see our password (`password`). This is called "Hardcoding Secrets

so we gonna use `.env` file (this file stays on the computer because it is in `.dockerignore` and we gonna add it in `.gitignore`)

1. Update our .env file:

```
# Database Secrets
POSTGRES_USER=myuser
POSTGRES_PASSWORD=mypassword
POSTGRES_DB=kanban_db

# The Connection Strings
# For Local App (running on host) → Uses port 5434
DATABASE_URL="postgresql://myuser:mypassword@localhost:5434/kanban_db?schema=public"

# For Docker App (running inside container) → Uses port 5432
DATABASE_URL_DOCKER="postgresql://myuser:mypassword@kanban_db:5432/kanban_db?schema=public"
```

2. update `docker-compose.yml` to use Variables

```
version: '3.8'

services:
  kanban-app:
    build: .
    ports:
      - "3003:3003"
    environment:
      # We use the specific Docker URL variable here
      - DATABASE_URL=${DATABASE_URL_DOCKER}
    depends_on:
      - kanban_db

  kanban_db:
    image: postgres:15
    environment:
      # Read these from the .env file
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - "5434:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

to check if everything is working as expected run `docker-compose config` this will shows we how Docker "sees" the file

3. cleaning up

   let's wipe the slate clean and start fresh with our new secure settings.

   a. Kill the Zombie (Free Port 3003)

      `fuser -k 3003/tcp`

   b. Delete the Old Volume

      `docker-compose down -v`

   c. Start Fresh

      `docker-compose up -d`

   d. push the tables one last time

`npx prisma db push`

Day 4

we aren't just storing data anymore. we are controlling *how* it changes.

Target 1: The Bouncer

right now, if I send a task with `"status": "TEST"`, our database accepts it. This will break our frontend later. We need to force it to only accept specific values.

1. **define the Rules**

   create a file: `src/tasks/dto/task-status.enum.ts`

   ```
   export enum TaskStatus {
     TODO = 'TODO',
     IN_PROGRESS = 'IN_PROGRESS',
     REVIEW = 'REVIEW',
     DONE = 'DONE',
   }
   ```

2. **Enforce the Rules**

   first Install the Missing Tools by running this command:

   `npm install class-validator class-transformer @nestjs/mapped-types`

   we gonna update `src/tasks/dto/create-task.dto.ts`

   ```
   import { IsString, IsNotEmpty, IsOptional, IsInt, IsEnum } from 'class-valida
   tor';
   import { TaskStatus } from './task-status.enum';

   export class CreateTaskDto {
     @IsString()
     @IsNotEmpty()
     title: string;

     @IsOptional()
   ```

```
  @IsString()
  description?: string;

  @IsOptional()
  @IsEnum(TaskStatus)
  status?: TaskStatus;

  @IsInt()
  projectId: number;
}
```

we installed the tools (`class-validator`) and we added the rules (`@IsEnum`), but we never turned the validation system ON. In NestJS, validation is not automatic. we have to explicitly tell the app: *"Hey, check every request against the DTO rules before letting it in."*. so we need to enable the `ValidationPipe` globally in our `main.ts` file.

open `src/main.ts`

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.useGlobalPipes(new ValidationPipe({
    whitelist: true,
    transform: true
  }));

  app.enableCors();

  await app.listen(process.env.PORT ?? 3003);
}
bootstrap();
```

Target 2: The Mover

we gonna allow users to move tasks (`TODO` → `DONE`)

1. **the Update Ticket (DTO)**

   Create a file: `src/tasks/dto/update-task.dto.ts`

   ```
   import { PartialType } from '@nestjs/mapped-types';
   import { CreateTaskDto } from './create-task.dto';

   export class UpdateTaskDto extends PartialType(CreateTaskDto) {}
   ```

   that single line `extends PartialType(...)` saves us from rewriting all the properties and manually adding `@IsOptional()` to every single one

2. **The Logic (Service)**

   now we need to write the function inside `TasksService` that actually modifies the database.
   In raw SQL, we would write: `UPDATE task SET title = 'New' WHERE id = 1;` in Prisma (our ORM), this looks like a function call. It takes two main arguments:

   - `where`: The unique identifier (the `id`) to find the specific row.
   - `data`: The object containing the new values (our `UpdateTaskDto`).

   we gonna do by updating `src/tasks/tasks.service.ts`

   ```
   // ... existing create() function ...

    async update(id: number, updateTaskDto: UpdateTaskDto) {
     return await this.prisma.task.update({
       where: { id: id },
       data: {
         title: updateTaskDto.title,
         description: updateTaskDto.description,
         status: updateTaskDto.status,
       },
   ```

```
    });
  }
```

3. **The Endpoint (Controller)**

   We have the DTO (the ticket) and the Service (the logic). Now we need the Controller (the receptionist) to accept the request from the outside world. so we gonna handle `PATCH /tasks/:id` inside `src/tasks/tasks.controller.ts`

```
import { Controller, Get, Post, Body, Patch, Param, Delete, ParseIntPipe
} from '@nestjs/common';
import { TasksService } from './tasks.service';
import { CreateTaskDto } from './dto/create-task.dto';
import { TasksGateway } from './tasks.gateway';
import { UpdateTaskDto } from './dto/update-task.dto';

@Controller('tasks')
export class TasksController {
  constructor(
    private readonly tasksService: TasksService,
    private readonly tasksGateway: TasksGateway
  ) {}

  @Post()
  async create(@Body() createTaskDto: CreateTaskDto) {
    const task = await this.tasksService.create(createTaskDto);
    this.tasksGateway.broadcastTaskCreated(task);
    return task;
  }

  @Patch(':id')
  async update(
    @Param('id', ParseIntPipe) id: number,
    @Body() updateTaskDto: UpdateTaskDto
  ) {
    const updatedTask = await this.tasksService.update(id, updateTaskD
to);
    this.tasksGateway.server.emit('task:updated', updatedTask);
    return updatedTask;
  }
}
```

Target 3: The Cleaner (DELETE)

in C++, `delete` frees up memory that is no longer needed. In a database, `DELETE` removes a row permanently.

1. **Step 3.1: The Logic (Service)**

   We will add a `remove` function to our Service

   Prisma Delete: It works exactly like `update`, but it doesn't need `data`. It only needs `where`.
   `DELETE FROM task WHERE id = 5;`

   open `src/tasks/tasks.service.ts` file and add this function:

   ```
   async remove(id: number) {
       return await this.prisma.task.delete({
         where: { id: id },
       });
     }
   ```

2. **Step 3.2: The Endpoint (Controller)**

   We need to map the HTTP `DELETE` verb to our service logic by adding the `remove` function inside `src/tasks/tasks.service.ts` we gonna some more update to avoid some errors:

   - **Import** the `UpdateTaskDto`.
   - **Import** the `PrismaService` (assuming we have one, or `PrismaClient`).
   - **Add a Constructor** to inject Prisma so `this.prisma` works.

   ```
   import { Injectable } from '@nestjs/common';
   import { CreateTaskDto } from './dto/create-task.dto';
   ```

```
import { UpdateTaskDto } from './dto/update-task.dto';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class TasksService {
  prisma = new PrismaClient();

  async create(createTaskDto: CreateTaskDto) {
    return await prisma.task.create({
      data: {
        title: createTaskDto.title,
        description: createTaskDto.description,
        project_id: createTaskDto.projectId,
        status: 'TODO',
        position: 1,
        created_by: 1
      },
    });
  }

  async update(id: number, updateTaskDto: UpdateTaskDto) {
    return await this.prisma.task.update({
      where: { id: id },
      data: {
        ...updateTaskDto,
      },
    });
  }

  async remove(id: number) {
    return await this.prisma.task.delete({
      where: { id: id },
    });
  }
}
```

The Grand Test

1. Run this command to start just the database

   `docker-compose up kanban_db -d`

2. Start the server:

   `npm run start:dev` (Make sure Docker App is stopped first: `docker stop kanban-service-kanban-app-1`)

3. Test 1: Create a Task (To get an ID)

```
curl -X POST http://localhost:3003/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Test Task", "projectId": 1, "project_id": 1}'curl -X POST http://localhost:3003/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Test Task", "projectId": 1, "project_id": 1}'
```

4. **Test 2: Try to break it (Validation):** Try to set status to "BANANA". It should fail (400 Bad Request).

```
curl -X PATCH http://localhost:3003/tasks/1 \
  -H "Content-Type: application/json" \
  -d '{"status": "BANANA"}'
```

5. **Test 3: Move the Task (Update):** Set status to "DONE"

   for this test we gonna made some change on the `client.html` file

```
<!DOCTYPE html>
<html>
<head>
    <title>Kanban Real-Time Test</title>
    <script src="https://cdn.socket.io/4.7.2/socket.io.min.js"></script>
    <style>
        body { font-family: monospace; padding: 20px; background: #222; color: #eee; }
```

```
      .log { margin-bottom: 5px; padding: 5px; border-radius: 4px; }
      .green { background: #1b5e20; } /* Created */
      .blue { background: #0d47a1; }  /* Updated */
      .red { background: #b71c1c; }   /* Deleted */
    </style>
</head>
<body>
    <h2>Status: <span id="status" style="color:gray">Connecting...</span
></h2>
    <div id="messages"></div>

    <script>
      const socket = io('http://localhost:3003');

      // 1. Connection Status
      socket.on('connect', () ⇒ {
        document.getElementById('status').innerText = "Connected! 🟢";
        document.getElementById('status').style.color = "#4caf50";
        console.log("Connected to server");
      });

      socket.on('disconnect', () ⇒ {
        document.getElementById('status').innerText = "Disconnected 🔴";
        document.getElementById('status').style.color = "#f44336";
      });

      // 2. Listen for NEW Tasks (Green)
      socket.on('task:created', (data) ⇒ {
        console.log("Created:", data);
        addMessage(`🆕 New Task: ${data.title} (ID: ${data.id})`, 'green');
      });

      // 3. Listen for UPDATED Tasks (Blue) ⟵ This was likely missing!
      socket.on('task:updated', (data) ⇒ {
        console.log("Updated:", data);
        addMessage(`🔄 Updated Task ${data.id}: Status is now ${data.sta
tus}`, 'blue');
        });

        // 4. Listen for DELETED Tasks (Red)
        socket.on('task:deleted', (data) => {
            console.log("Deleted:", data);
            addMessage(`🗑️ Deleted Task ID: ${data.id}`, 'red');
        });

        function addMessage(text, className) {
            const div = document.createElement('div');
            div.className = `log ${className}`;
            div.innerText = text;
            document.getElementById('messages').appendChild(div);
        }
    </script>
</body>
</html>
```

Create a Task (to get a fresh ID, likely ID #4):

```
curl -X POST http://localhost:3003/tasks \
  -H "Content-Type: application/json" \
  -d '{"title": "Sleep Time", "projectId": 1, "project_id": 1}'
```

Update the Task (Change ID to match the one we just created, e.g., 4):

```
curl -X PATCH http://localhost:3003/tasks/4 \
  -H "Content-Type: application/json" \
  -d '{"status": "DONE"}'
```

6. The Cleaner (DELETE)

   The Delete Command Run this to delete Task #4 (the one we just marked as DONE)

```
curl -X DELETE http://localhost:3003/tasks/4
```

To prove it is really gone, try to update it again. we should get an "Internal Server Error" (because the database will say "I can't find ID 1").

```
curl -X PATCH http://localhost:3003/tasks/1 \
  -H "Content-Type: application/json" \
  -d '{"status": "DONE"}'
```

here we gonna get an error because we say “Hey Database, please update Task #4 to DONE." Database: "I looked everywhere. Task #1 does not exist. I cannot update a ghost.”

why it says "500 Internal Server Error" instead of "404 Not Found". Right now, our code assumes the ID always exists. When it doesn't, Prisma panics and throws an exception (Crash), which NestJS reports as a 500 Error.

The Fix: Catch the Crash

1. Open `src/tasks/tasks.service.ts` and replace `update` and `remove` functions to this:

```
import { Injectable, NotFoundException } from '@nestjs/common'; // <-
-- 1. Import NotFoundException
// ... other imports ...

// ... inside the class ...

  async update(id: number, updateTaskDto: UpdateTaskDto) {
    try {
      return await this.prisma.task.update({
        where: { id: id },
        data: { ...updateTaskDto },
      });
    } catch (error) {
      // P2025 is Prisma's code for "Record not found"
      if (error.code === 'P2025') {
        throw new NotFoundException(`Task with ID ${id} not found`);
      }
      // If it's another error, re-throw it so we know something else broke
      throw error;
    }
  }
  async remove(id: number) {
    try {
      return await this.prisma.task.delete({
        where: { id: id },
      });
    } catch (error) {
      if (error.code === 'P2025') {
        throw new NotFoundException(`Task with ID ${id} not found`);
      }
      throw error;
    }
  }
```

2. Test It:
    a. Restart the server.
    b. Run the error command again:

    ```
    curl -X PATCH http://localhost:3003/tasks/4 \
      -H "Content-Type: application/json" \
      -d '{"status": "DONE"}'
    ```

Day 4 Graduation

we have successfully transitioned from "Configuration" to "Coding".

- [x] ~~**Validation:** our blocked "BANANA" statuses.~~
- [x] ~~**Update:** we moved a task from TODO to DONE.~~
- [x] ~~**Delete:** we removed a task permanently.~~
- [x] ~~**Real-Time:** All of these actions sent WebSocket messages to the browser~~