

Overview of the 5 Telecom Projects

1. Project 1: AssetTrack - Telecom Asset Management Dashboard

- **Core Task:** Track telecom equipment (towers, routers) on a map, manage maintenance schedules, and view asset details.
- **Key Tech:** React, Leaflet (Maps), Prisma, PostgreSQL.

2. Project 2: SupportDesk - Customer Service Portal & Ticketing System

- **Core Task:** A platform for customers to submit support tickets and chat in real-time with support agents.
- **Key Tech:** React, Socket.io (Real-time chat), Prisma, PostgreSQL, JWT (for customer/agent roles).

3. Project 3: CoverageMapper - Network Coverage Mapping Tool

- **Core Task:** Visualize network coverage (4G, 5G) on an interactive map. Users can check coverage at a specific address.
- **Key Tech:** React, Leaflet (Maps with overlays/heatmaps), Recharts (for coverage stats), Prisma, PostgreSQL.

4. Project 4: BillFlow - Telecom Billing & Plan Management System

- **Core Task:** An admin dashboard to manage customer billing cycles, generate invoices, and manage subscription plans.
- **Key Tech:** React, React Query, Recharts (for billing analytics), Prisma, PostgreSQL.

5. Project 5: ProcureNet - Vendor & Equipment Procurement Portal

- **Core Task:** A portal for managing vendors, creating purchase orders for equipment, and handling multi-step approval workflows.
- **Key Tech:** React, React Query (for complex state), Prisma, PostgreSQL, JWT (for different user roles like manager, employee).

Day 1: Frontend Foundations with React & Tailwind CSS

Summary: Today, we're jumping straight into building modern user interfaces. You'll set up a professional development environment, learn the core concepts of React like JSX and components, and style them efficiently with Tailwind CSS.



Learning Objectives

- Scaffold a new React project using Vite.
- Understand JSX syntax and its role in React.
- Create functional components with props for reusability.
- Manage component state using the `useState` hook.
- Set up and use Tailwind CSS for rapid, utility-first styling.
- Build and style a basic UI component.

Prerequisites

- **Node.js & npm/pnpm:** Node.js v18+ installed.
- **Code Editor:** VS Code with ESLint, Prettier, and Tailwind CSS IntelliSense extensions.
- **Git & GitHub:** Basic knowledge of `git` and a GitHub account.

Agenda

- **09:00 - 09:30:** Welcome & Setup Check 
- **09:30 - 10:30:** Topic: Intro to React, Vite, and JSX.
- **10:30 - 11:30:** Topic: Functional Components, Props, and State (`useState`).
- **11:30 - 12:30:** Hands-on Lab 1: Building a Static `UserProfile` Card.
- **12:30 - 13:30:** Lunch Break 
- **13:30 - 14:30:** Topic: Introduction to Tailwind CSS.
- **14:30 - 16:00:** Hands-on Lab 2: Styling the `UserProfile` Card with Tailwind.
- **16:00 - 17:30:** Team Project Work: Initial Project Setup & Component Scaffolding.
- **17:30 - 18:00:** Daily Demos & Wrap-up.

Topics & Teaching Content

(This section is identical to the previous response, covering React, Vite, JSX, Components, State, and Tailwind setup fundamentals. The core teaching remains the same.)

Hands-on Lab: Build a PlanCard Component

(This lab is identical to the previous response, as it provides a solid, generic foundation for component building before students tackle their specific projects.)

Team Project Work

Goal: Each team sets up their frontend repository and creates the main layout and one core static component relevant to their project.

- **All Projects:**
 1. Create a new Vite project (react-ts).
 2. Initialize a Git repository and push it to a new GitHub repo.
 3. Set up Tailwind CSS.
 4. Create a basic layout with a Navbar and a Footer component in src/components/layout.
- **Project 1 (AssetTrack):** Create a static AssetSummaryCard (/components/AssetSummaryCard.tsx) to show details like Asset Name, Type (e.g., Tower, Router), Status (Online/Offline), and Location (text for now).
- **Project 2 (SupportDesk):** Create a static TicketPreview component (/components/TicketPreview.tsx) to display a ticket's subject, customer name, status (Open/Closed), and last update time.
- **Project 3 (CoverageMapper):** Create a CoverageSearch component (/components/CoverageSearch.tsx) with a styled input field for an address and a "Check Coverage" button.
- **Project 4 (BillFlow):** Create a static InvoiceRow component (/components/InvoiceRow.tsx) that displays an Invoice ID, Customer Name, Amount, Due Date, and Status (Paid/Overdue) in a table-like format using divs.
- **Project 5 (ProcureNet):** Create a static PurchaseOrderCard (/components/PurchaseOrderCard.tsx) showing an Order ID, Vendor Name, Total Cost, and Status (Pending Approval/Approved).

Daily Demo Checklist

The team must demonstrate:

- [] The Vite development server is running.
- [] The GitHub repository is set up with the initial commit.
- [] The main `App.tsx` renders the `Navbar`, `Footer`, and the team's core static component.
- [] The core component is styled with Tailwind CSS and looks professional.

Homework

1. **Refine:** Add hover effects and responsive classes (`md:`, `lg:`) to your team's core component.
2. **Explore:** Create one more simple, static component for your project (e.g., a styled button, a status badge).
3. **Read:** Go through the official React "Thinking in React" guide.

Short Quiz

1. **Question:** What is the main difference between `props` and `state`?
 - **Answer:** `props` are passed from a parent and are immutable in the child. `state` is internal data managed by the component itself and can be changed.
2. **Question:** What command initializes Tailwind CSS configuration files?
 - **Answer:** `npx tailwindcss init -p`
3. **Question:** Why do we use `className` instead of `class` in JSX?
 - **Answer:** Because JSX is JavaScript, and `class` is a reserved keyword.

Resources

- **React:** [Official React Docs](#)
- **Tailwind CSS:** [Official Tailwind CSS Docs](#)

Instructor Notes

- **Pacing:** The setup part can be slow. Ensure everyone has Node/Git installed beforehand.
- **Troubleshooting:** Tailwind CSS not applying styles is the most common issue. Double-check the `content` path in `tailwind.config.js` and the `@tailwind` directives in `index.css`.

Day 2: React State, Routing, and Data Fetching

Summary: Today, we'll make our React apps interactive and dynamic. You'll learn how to handle user events, manage side effects, implement client-side routing to create a multi-page feel, and master modern data fetching with React Query.

Learning Objectives

- Handle user events like clicks and form submissions.
- Understand the `useEffect` hook for managing side effects.
- Implement client-side navigation using React Router.
- Grasp the core concepts of server state vs. client state.
- Fetch data from an API using React Query (`useQuery`).
- Mutate data on the server using React Query (`useMutation`).

Prerequisites

- Completion of Day 1 content.

Agenda

- **09:00 - 09:15:** Daily Standup & Homework Review.

- **09:15 - 10:15:** Topic: Handling Events & `useEffect` Hook.
- **10:15 - 11:30:** Topic: Client-Side Routing with React Router.
- **11:30 - 12:30:** Hands-on Lab 1: Creating a Multi-Page App.
- **12:30 - 13:30:** Lunch Break 🍷
- **13:30 - 14:00:** Topic: Intro to Server State & React Query.
- **14:00 - 15:30:** Topic: Fetching (`useQuery`) & Mutating (`useMutation`) Data.
- **15:30 - 17:30:** Team Project Work: Implementing Routes and Mock Data Fetching.
- **17:30 - 18:00:** Daily Demos & Wrap-up.

Topics & Teaching Content

(This section is identical to the previous response, covering Events, `useEffect`, React Router, and React Query fundamentals.)

Hands-on Lab: Create a Pokedex App

(This lab is identical to the previous response, as it's a perfect, self-contained exercise for practicing routing and data fetching with a public API before students connect to their own backends.)

Team Project Work

Goal: Structure the app with pages and routes, and use React Query with a mock API to fetch initial data.

- **All Projects:**
 1. Create a `src/pages` directory.
 2. Define the main pages for your app.
 3. Set up React Router in `App.tsx` and a `Navbar` with `<Link>` tags.
 4. Set up React Query (`QueryClientProvider`).

5. For one page, use `useQuery` to fetch data from a mock source like [JSONPlaceholder](#) or a local JSON file to populate your components from Day 1.
- **Project 1 (AssetTrack):** Create `DashboardPage` and `AssetListPage`. The `AssetListPage` should use `useQuery` to fetch a mock array of asset objects and render an `AssetSummaryCard` for each.
 - **Project 2 (SupportDesk):** Create `DashboardPage` (for agents) and `MyTicketsPage` (for customers). `MyTicketsPage` should use `useQuery` to fetch a mock list of tickets and render a `TicketPreview` for each.
 - **Project 3 (CoverageMapper):** Create `MapPage` and `StatsPage`. The `MapPage` will be the main focus. For now, use `useQuery` on the `StatsPage` to fetch mock network statistics (e.g., `{ 5g_towers: 150, 4g_towers: 800 }`).
 - **Project 4 (BillFlow):** Create `InvoicesPage` and `CustomersPage`. The `InvoicesPage` should use `useQuery` to fetch a mock list of invoices and render an `InvoiceRow` for each.
 - **Project 5 (ProcureNet):** Create `OrdersPage` and `VendorsPage`. The `OrdersPage` should use `useQuery` to fetch mock purchase orders and render a `PurchaseOrderCard` for each.

Daily Demo Checklist

- [] The app has at least two navigable pages using React Router.
- [] The `Navbar` navigates between pages without a full page reload.
- [] React Query's `QueryClientProvider` is set up.
- [] At least one component successfully uses `useQuery` to fetch and display mock data.
- [] Loading and error states from `useQuery` are handled in the UI.

Homework

1. **Add a Detail Page:** Create a new page that shows more details about a single item (asset, ticket, etc.) when you click on it from the list. Use the `useParams` hook.
2. **Add a Form:** Create a simple form for creating a new item. Manage the input values with `useState`.
3. **Read:** Skim the React Query docs on "Important Defaults."

Short Quiz

- 1. Question:** In React Router, what component renders a component based on the URL?
 - **Answer:** The `<Route>` component, inside `<Routes>`.
- 2. Question:** What is the purpose of the array passed as the first argument to `useQuery`?
 - **Answer:** It's the `queryKey`, used by React Query to cache and manage the data.
- 3. Question:** When would you use `useEffect` instead of React Query?
 - **Answer:** For non-server-state side effects, like adding a `window` event listener or setting a timer.

Resources

- **React Router:** [Official Docs](#)
- **React Query:** [Official Docs](#)

Instructor Notes

- **Pacing:** The concept of server state vs. client state is key. Use analogies. Server state is data that lives on your server; you just have a temporary copy. Client state is data that only exists in the browser.
- **Troubleshooting:** `QueryClientProvider` must wrap the entire app. Ensure the `queryFn` returns a promise (e.g., the result of an `axios` or `fetch` call).

Day 3: Building a Backend with Node.js & Express

Summary: It's time to build the server! Today you'll learn how to create a powerful backend API using Node.js and the Express framework. You'll define RESTful endpoints, handle incoming requests, and structure a scalable backend application.

Learning Objectives

- Understand the role of Node.js and the event loop.
- Set up a new Node.js project with TypeScript.
- Create a basic web server using Express.
- Understand and implement RESTful API principles (GET, POST, PUT, DELETE).
- Use Express middleware for common tasks like parsing JSON bodies.
- Structure a backend project with routes, controllers, and services.

Prerequisites

- Completion of Day 1-2 content.

Agenda

- **09:00 - 09:15:** Daily Standup & Homework Review.
- **09:15 - 10:15:** Topic: Intro to Node.js, npm, and the Backend.
- **10:15 - 11:30:** Topic: Setting up an Express Server with TypeScript.
- **11:30 - 12:30:** Topic: Routing and Controllers.
- **12:30 - 13:30:** Lunch Break 🍽️
- **13:30 - 14:30:** Topic: Middleware and Handling Requests (body, params, query).
- **14:30 - 16:00:** Hands-on Lab: Building a "Task List" API.
- **16:00 - 17:30:** Team Project Work: Scaffolding the Backend and Creating First Endpoints.
- **17:30 - 18:00:** Daily Demos & Wrap-up.

Topics & Teaching Content

(This section is identical to the previous response, covering Node.js, Express setup, Routing, Controllers, and Middleware fundamentals.)

Hands-on Lab: Building a "Task List" API

(This lab is identical to the previous response. It provides a crucial, isolated environment for students to practice creating a full CRUD API before applying the concepts to their more complex projects.)

Team Project Work

Goal: Each team will set up their backend repository and create the initial, in-memory (no database yet) CRUD endpoints for their primary resource.

- **All Projects:**
 1. Create a new `server` directory, initialize a Node/TypeScript project, and install Express.
 2. Set up the basic server structure with folders for `routes` and `controllers`.
 3. Create an in-memory array to store data.
 4. Implement `GET` (all) and `GET` (by ID) endpoints for your main data model.
- **Project 1 (AssetTrack):** Create endpoints for assets.
 1. `GET /api/assets`
 2. `GET /api/assets/:id`
- **Project 2 (SupportDesk):** Create endpoints for tickets.
 1. `GET /api/tickets`
 2. `GET /api/tickets/:id`
- **Project 3 (CoverageMapper):** Create endpoints for coverage data points (e.g., a point on a map with a signal strength).
 1. `GET /api/coverage-points`
 2. `GET /api/coverage-points/:id`
- **Project 4 (BillFlow):** Create endpoints for invoices.
 1. `GET /api/invoices`
 2. `GET /api/invoices/:id`
- **Project 5 (ProcureNet):** Create endpoints for purchase orders.
 1. `GET /api/purchase-orders`
 2. `GET /api/purchase-orders/:id`

Daily Demo Checklist

- [] The Node.js server starts without errors using `pnpm run dev`.
- [] The GitHub repository for the server is set up.
- [] The project has a clear folder structure (`routes`, `controllers`).
- [] The team can demonstrate fetching all items (`GET /api/...`) using Postman/Insomnia.
- [] The team can demonstrate fetching a single item by its ID (`GET /api/.../:id`).
- [] The API correctly returns JSON data with appropriate status codes.

Homework

1. **Implement More Endpoints:** Add the `POST`, `PUT`, and `DELETE` endpoints for your project's main resource using the in-memory array.
2. **Add Validation:** Add simple validation to your `POST` endpoint (e.g., check if required fields are present in `req.body`).
3. **Read:** Read about REST API status codes (200, 201, 400, 404, 500).

Short Quiz

1. **Question:** What is the purpose of `express.json()` middleware?
 - **Answer:** It parses incoming requests with JSON payloads and makes the data available on `req.body`.
2. **Question:** In Express, how do you access a URL parameter like the `id` in `/items/:id`?
 - **Answer:** Via `req.params.id`.
3. **Question:** What script in `package.json` allows for automatic server restarts on file changes?
 - **Answer:** A script using `nodemon`, like `"dev": "nodemon src/index.ts"`.

Resources

- **Express.js:** [Official Website](#)
- **REST API Status Codes:** [MDN HTTP status codes](#)

Instructor Notes

- **Pacing:** This is a big context switch. Use diagrams of the request-response cycle.
- **Troubleshooting:** Common issues: forgetting `express.json()` leads to `req.body` being `undefined`; CORS errors (ensure `cors()` is used); typos in route paths. Using an API client like Postman is essential for debugging.

Day 4: Databases with PostgreSQL & Prisma

Summary: Today we replace our in-memory arrays with a real, persistent database. You'll learn how to model your data, interact with a PostgreSQL database using the powerful Prisma ORM, and run database migrations to keep your schema in sync with your code.


Learning Objectives

- Understand the basics of relational databases and SQL.
- Set up a free PostgreSQL database on the cloud (e.g., Supabase, Neon).
- Install and configure Prisma ORM in a Node.js project.
- Define data models using the Prisma schema language.
- Run database migrations to create and update your database tables.
- Perform CRUD (Create, Read, Update, Delete) operations using the Prisma Client.

Prerequisites

- Completion of Day 3 content.
- A free account on a cloud database provider like [Supabase](#) or [Neon](#).

Agenda

- **09:00 - 09:15:** Daily Standup & Homework Review.
- **09:15 - 10:15:** Topic: Intro to Relational Databases & PostgreSQL.
- **10:15 - 11:30:** Topic: Setting up Prisma and Defining Schemas.
- **11:30 - 12:30:** Topic: Migrations with `prisma migrate`.
- **12:30 - 13:30:** Lunch Break 
- **13:30 - 15:00:** Topic: CRUD Operations with Prisma Client.
- **15:00 - 16:00:** Hands-on Lab: Connecting the Task List API to a Real Database.
- **16:00 - 17:30:** Team Project Work: Modeling Data and Migrating the Database.
- **17:30 - 18:00:** Daily Demos & Wrap-up.

Topics & Teaching Content

1. Introduction to Prisma ORM

- **Explanation:** An Object-Relational Mapper (ORM) like Prisma allows you to interact with your database using your programming language (TypeScript) instead of writing raw SQL. You define your models in a schema file, and Prisma generates a type-safe client to query your data. This speeds up development and reduces errors.
- **Key Concepts:** Schema-first workflow, migrations, type-safe client.
- **Installation & Setup:**
`Bash`

```
# In your 'server' directory
```

- `pnpm add prisma @prisma/client`
- `pnpm add -D @types/node`
-

- `# Initialize Prisma`
- `npx prisma init`
-

This creates a `prisma` folder with a `schema.prisma` file and a `.env` file for your database connection string.

2. Defining Schemas and Running Migrations

- **Explanation:** The `schema.prisma` file is the single source of truth for your database schema. You define models, fields, types, and relations here. Once you change your schema, you run a **migration** to apply those changes to your actual database.
- **Code Example (`schema.prisma`):**
Code snippet

`// This is your Prisma schema file,`

- `// learn more about it in the docs: https://pris.ly/d/prisma-schema`
-
- `generator client {`
- `provider = "prisma-client-js"`
- `}`
-
- `datasource db {`
- `provider = "postgresql"`
- `url = env("DATABASE_URL")`
- `}`
-
- `model User {`
- `id Int @id @default(autoincrement())`
- `email String @unique`
- `name String? // Optional field`
- `posts Post[] // Relation to the Post model`
- `}`
-

- `model Post {`
- `id Int @id @default(autoincrement())`
- `title String`
- `content String?`
- `published Boolean @default(false)`
- `author User @relation(fields: [authorId], references: [id])`
- `authorId Int`
- `}`
-

- **Migration Commands:**
Bash

`# Create a new migration based on schema changes and apply it`

- `npx prisma migrate dev --name init`
-
- `# After running migrate, you need to generate the client`
- `npx prisma generate`
-

3. CRUD with Prisma Client

- **Explanation:** Prisma generates a `PrismaClient` that you can use to programmatically query your database in a type-safe way.
- **Code Example (Using the client in a controller):**
`TypeScript`

```

import { PrismaClient } from '@prisma/client';
import { Request, Response } from 'express';

const prisma = new PrismaClient();

// GET /api/posts
export const getAllPosts = async (req: Request, res: Response) => {
  const posts = await prisma.post.findMany({
    include: { author: true }, // Include the related author
  });
  res.json(posts);
};

// POST /api/posts
export const createPost = async (req: Request, res: Response) => {
  const { title, content, authorId } = req.body;
  const newPost = await prisma.post.create({
    data: { title, content, authorId },
  });
  res.status(201).json(newPost);
};

```

Hands-on Lab: Refactor Task List API with Prisma

Objective: Modify yesterday's in-memory Task List API to use Prisma and a real PostgreSQL database.

1. **Setup:** Create a new PostgreSQL database on Supabase/Neon and get the connection string.
2. **Initialize Prisma:** Run `npx prisma init` in your lab project. Add the connection string to the `.env` file.
3. **Define Schema:** Create a `Task` model in `schema.prisma`.
Code snippet


```

4.   id      Int    @id @default(autoincrement())
5.   title   String
6.   completed Boolean @default(false)
7.   createdAt DateTime @default(now())
8. }
9.

```

10. **Migrate:** Run `npx prisma migrate dev --name add-tasks-table`.
11. **Refactor Controllers:** Replace all the in-memory array logic in your controllers with PrismaClient calls (`prisma.task.findMany()`, `prisma.task.create()`, etc.).
12. **Test:** Use Postman to verify that your API now creates, reads, updates, and deletes tasks from the actual database.

Team Project Work

Goal: Define the core data models for your project, run the initial database migration, and refactor your `GET` endpoints to fetch data from the database.

- **All Projects:**
 1. Set up a shared PostgreSQL database for the team on Supabase/Neon.
 2. Initialize Prisma in the server project.
 3. Define the primary models in `schema.prisma`. Don't worry about all relations yet, focus on the main entities.
 4. Run `npx prisma migrate dev --name init` to create the tables.
 5. Instantiate PrismaClient and refactor your `GET` (all) and `GET` (by ID) endpoints to use it.
- **Project 1 (AssetTrack):** Define `Asset` and `MaintenanceLog` models. The `Asset` should have fields like `name`, `type`, `status`, `latitude`, `longitude`.
- **Project 2 (SupportDesk):** Define `Ticket` and `User` models. A `Ticket` should have `subject`, `description`, `status`, and relations to a customer `User` and an agent `User`.

- **Project 3 (CoverageMapper):** Define a `CoveragePoint` model with `latitude`, `longitude`, `signalStrength` (integer), and `networkType` (e.g., "5G", "4G").
- **Project 4 (BillFlow):** Define `Customer`, `Plan`, and `Invoice` models. An `Invoice` should relate to a `Customer` and have fields like `amount`, `dueDate`, `status`.
- **Project 5 (ProcureNet):** Define `Vendor`, `PurchaseOrder`, and `LineItem` models. A `PurchaseOrder` should relate to a `Vendor` and have many `LineItems`.

Daily Demo Checklist

- [] The `.env` file contains a valid `DATABASE_URL`.
- [] The `schema.prisma` file shows the core models for the project.
- [] The team can show that `npx prisma migrate dev` ran successfully.
- [] Using a database GUI (like Supabase's table editor) or `psql`, the team can show the created tables.
- [] The `GET /api/...` endpoints now return data from the database (even if it's empty) instead of the old in-memory array.

Homework

1. **Seed the Database:** Create a `seed.ts` script to populate your database with some initial sample data. Use Prisma Client within the script. See Prisma docs for "Seeding".
2. **Refactor All Endpoints:** Convert the remaining `POST`, `PUT`, and `DELETE` endpoints to use Prisma Client.
3. **Explore Relations:** Read the Prisma docs on one-to-many and many-to-many relationships. Try to add one simple relation to your schema (e.g., a `User` has many `Tickets`).

Short Quiz

1. **Question:** What is the "single source of truth" for your database structure when using Prisma?
 - **Answer:** The `schema.prisma` file.
2. **Question:** What command applies your schema changes to the database?
 - **Answer:** `npx prisma migrate dev`

3. **Question:** How do you fetch a record and its related records in a single Prisma query?

- **Answer:** By using the `include` option in methods like `findUnique` or `findMany`.
E.g., `prisma.user.findUnique({ where: { id: 1 }, include: { posts: true } })`.

Resources

- **Prisma Docs:** [Official Website](#)
- **PostgreSQL:** [Official Website](#)
- **Supabase:** [Postgres Hosting](#)

Instructor Notes

- **Pacing:** This day is heavy on new concepts (SQL, ORMs, migrations). It's okay if teams don't finish refactoring all endpoints. The priority is a successful first migration and understanding the workflow.
- **Troubleshooting:** Database connection errors are common. Double-check the `DATABASE_URL` in `.env`. Ensure any firewalls (corporate networks) are not blocking the database port. A common mistake is forgetting to run `npx prisma generate` after a schema change, which can lead to type errors.

Here is the continuation of the 8-day plan, followed by all the requested supplementary artifacts.

Day 5: Full-Stack Integration

Summary: This is the day it all comes together. You will connect your React frontend to your Express backend, replacing all mock data with live data from your own API and database. This is a huge milestone in becoming a full-stack developer.

Learning Objectives


- Understand and troubleshoot Cross-Origin Resource Sharing (CORS) issues.
- Configure the frontend to make API calls to the backend server.
- Use frontend environment variables to manage the API base URL.
- Refactor React Query hooks to fetch data from your own API.
- Implement "mutations" with React Query to create, update, and delete data from the frontend.
- Handle API loading and error states gracefully in the UI.

Prerequisites

- Completion of Day 4 content.
- A running React application and a running Express/Prisma backend.

Agenda

- **09:00 - 09:15:** Daily Standup & Homework Review.
- **09:15 - 10:00:** Topic: CORS and Connecting Frontend to Backend.
- **10:00 - 11:30:** Topic: Full CRUD with React Query (`useQuery`, `useMutation`).

- **11:30 - 12:30:** Hands-on Lab: Connecting the React Pokedex to a Real Backend.
- **12:30 - 13:30:** Lunch Break 
- **13:30 - 17:30:** Team Project Work: Full-Stack CRUD Implementation.
- **17:30 - 18:00:** Daily Demos & Wrap-up.

Topics & Teaching Content

1. CORS and Frontend Configuration

- **Explanation:** For security reasons, browsers restrict HTTP requests to a different "origin" (domain, protocol, or port). Since your React app (e.g., `localhost:5173`) and your Express server (e.g., `localhost:3001`) are on different ports, they are different origins. **CORS** is a mechanism that uses additional HTTP headers to tell browsers to permit a web application running at one origin to access selected resources from a server at a different origin. We already installed the `cors` package, now we'll see it in action.
- **Key Concepts:** Origin, Preflight Request (`OPTIONS`).
- **Backend `cors` setup (review):**
TypeScript

```
// server/src/index.ts
```

- `import cors from 'cors';`
- `// ...`
- `app.use(cors({`
- `origin: 'http://localhost:5173', // Or your frontend's URL`
- `methods: ['GET', 'POST', 'PUT', 'DELETE'],`
- `}));`
-

- Frontend Environment Variable:
Create a .env.local file in the root of your React project (client folder).
client/.env.local
- VITE_API_BASE_URL=http://localhost:3001/api
-

Now you can access this in your code as `import.meta.env.VITE_API_BASE_URL`.

2. Full CRUD with React Query

- **Explanation:** We've used `useQuery` for reading data. For creating, updating, or deleting data, we use `useMutation`. A mutation is typically used for any server side-effect. `useMutation` gives you a `mutate` function to call, along with status indicators like `isLoading` and `isSuccess`. A key feature is its ability to automatically refetch or update your queries upon success.
- **Code Example (Creating a post):**
TypeScript

```
import { useMutation, useQueryClient } from '@tanstack/react-query';
```

- ```
import axios from 'axios';
```
- 

```
const apiClient = axios.create({ baseURL:
import.meta.env.VITE_API_BASE_URL });
```

- 
- ```
// Function that performs the API call
```
- ```
const createPost = async (newPost: { title: string; content: string }) => {
```
- ```
  const { data } = await apiClient.post('/posts', newPost);
```
- ```
 return data;
```
- ```
};
```
-

```
const PostForm = () => {  
  const queryClient = useQueryClient();
```

- ```
 const mutation = useMutation({
```

- mutationFn: createPost,
- onSuccess: () => {
- // Invalidate and refetch the 'posts' query after a new post is created
- queryClient.invalidateQueries({ queryKey: ['posts'] });
- },
- });
- 
- const handleSubmit = (e) => {
- e.preventDefault();
- // ... get form data
- mutation.mutate({ title: 'New Title', content: 'New Content' });
- };
- 
- return (
- <form onSubmit={handleSubmit}>
- { /\* ... form inputs ... \*/ }
- <button type="submit" disabled={mutation.isLoading}>
- {mutation.isLoading ? 'Saving...' : 'Create Post'}
- </button>
- </form>
- );
- };
- 

## Hands-on Lab: Full-Stack Task List

**Objective:** Connect the React frontend for the Task List app to the Prisma-powered backend from yesterday's lab.

1. **Run Both Servers:** Start the React dev server (`pnpm dev` in `client`) and the Express server (`pnpm dev` in `server`).
2. **Configure CORS:** Ensure your Express server's `cors` middleware allows requests from the React app's origin.
3. **Set Env Var:** Create `.env.local` in the React app and set `VITE_API_BASE_URL`.
4. **Refactor `useQuery`:** Change the `useQuery` hook in your main list component to fetch from `GET /api/tasks` on your backend.

5. **Create a Form:** Build a simple form component to add a new task.
6. **Implement `useMutation`:** Use the `useMutation` hook to handle the form submission, calling `POST /api/tasks`.
7. **Invalidate Queries:** On a successful mutation, use `queryClient.invalidateQueries` to refetch the list of tasks, making your UI update automatically.
8. **Bonus:** Add buttons to delete and toggle the completion status of tasks, each powered by its own `useMutation` hook.

## Team Project Work

**Goal:** Connect the primary pages of your frontend application to your backend API. Achieve full CRUD functionality for at least one major data model.

- **All Projects:**
  1. Start both frontend and backend servers concurrently. (Hint: use a tool like `concurrently` or just two separate terminal windows).
  2. Configure CORS correctly.
  3. Set up the `VITE_API_BASE_URL` environment variable.
  4. Refactor the main list page to fetch data from the backend using `useQuery`.
  5. Create a form page/modal (`/items/new`) for creating a new item.
  6. Implement a `useMutation` hook to handle the creation form, which calls your `POST` endpoint. On success, it should invalidate the list query to show the new item.
- **Project 1 (AssetTrack):** Fetch and display the list of assets on the `AssetListPage`. Implement a form to create a new asset.
- **Project 2 (SupportDesk):** Fetch and display tickets on the `MyTicketsPage`. Implement a form for customers to submit a new ticket.
- **Project 3 (CoverageMapper):** Create an admin page to add new `CoveragePoint` data via a form. The main map page can fetch and display these points (we'll make this visual on Day 7).
- **Project 4 (BillFlow):** Fetch and display the list of invoices. Implement a form to create a new customer or a new billing plan.
- **Project 5 (ProcureNet):** Fetch and display the list of purchase orders. Implement a form to create a new purchase order.

## Daily Demo Checklist



- [ ] Both frontend and backend servers are running.
- [ ] The frontend successfully fetches a list of data from the backend API (verified in the browser's Network tab).
- [ ] The team can demonstrate creating a new item via a form on the frontend.
- [ ] After creation, the UI automatically updates to show the new item without a page refresh.
- [ ] Loading and error states are handled visually (e.g., a "Loading..." message, an error alert).

## Homework

1. **Implement Update/Delete:** Add functionality to edit and/or delete items from your list, each using its own `useMutation` hook.
2. **Error Handling:** Improve error handling. If the API returns an error (e.g., 400 for bad input), display a user-friendly message on the screen.
3. **Refactor API Calls:** Create a dedicated API client file (e.g., `src/api/apiClient.ts`) using `axios` to centralize your base URL and other configurations.

## Short Quiz

1. **Question:** Why do we need to configure CORS on the backend?
  - **Answer:** To relax the browser's Same-Origin Policy, explicitly allowing the frontend (on a different origin) to make requests to the backend server.
2. **Question:** In React Query, what hook do you use to modify server data, and what hook do you use to read it?
  - **Answer:** `useMutation` for modifying (Create, Update, Delete) and `useQuery` for reading.
3. **Question:** What is the purpose of `queryClient.invalidateQueries()`?
  - **Answer:** It marks specific queries as "stale," prompting React Query to refetch their data automatically, which is useful for updating the UI after a mutation.

## Resources

- **CORS:** [MDN - Cross-Origin Resource Sharing \(CORS\)](#)

- **React Query Mutations:** [Official Docs](#)
- **Axios:** [Axios Docs on GitHub](#)

## Instructor Notes

- **Pacing:** This day can be a slog of debugging. Network errors, CORS issues, and off-by-one errors in API logic are common. Encourage pair programming.
- **Troubleshooting:** The browser's **Network Tab** in the developer tools is their best friend today. Teach them how to inspect requests, check headers, and view responses. A **404 Not Found** usually means a typo in the URL. A **CORS error** means the server didn't send the right headers. A **500 Internal Server Error** means the backend code crashed.

## Day 6: Authentication & Authorization with JWT

**Summary:** Today, we secure our application. You'll learn how to implement a complete user authentication system from scratch using JSON Web Tokens (JWT). Users will be able to register, log in, and access routes and data that are protected based on their identity.

### Learning Objectives

- Understand the workflow of token-based authentication.
- Hash and verify user passwords securely using **bcrypt**.
- Create, sign, and verify JSON Web Tokens (JWT) on the server.
- Implement protected routes on the backend using Express middleware.
- Store JWTs securely on the client-side (in **localStorage**).
- Create a global authentication context in React to manage user state.
- Send the JWT with API requests from the frontend to access protected resources.

### Prerequisites

- Completion of Day 5 content.
- A fully integrated client-server application.

## Agenda

- **09:00 - 09:15:** Daily Standup.
- **09:15 - 10:30:** Topic: Authentication Concepts, Password Hashing (`bcrypt`).
- **10:30 - 11:30:** Topic: JSON Web Tokens (JWT) Workflow.
- **11:30 - 12:30:** Topic: Backend Protected Routes with Middleware.
- **12:30 - 13:30:** Lunch Break 🍴
- **13:30 - 14:30:** Topic: Frontend Auth Management (Context, `localStorage`).
- **14:30 - 16:00:** Hands-on Lab: Building a Secure Login System.
- **16:00 - 17:30:** Team Project Work: Integrating Auth into Projects.
- **17:30 - 18:00:** Daily Demos & Wrap-up.

## Topics & Teaching Content

### 1. Password Hashing & JWT Workflow

- **Explanation:** You **NEVER** store passwords in plain text. We use a one-way hashing algorithm like **bcrypt** to turn a password into a long, irreversible string (a hash). When a user logs in, we hash their provided password and compare it to the stored hash.
- **JWTs** are a stateless way to handle authentication.
  1. User logs in with email/password.
  2. Server verifies credentials.
  3. Server creates a JWT (a signed JSON object containing user info like `userId`) and sends it to the client.
  4. Client stores the JWT (e.g., in `localStorage`).
  5. For every subsequent request to a protected resource, the client sends the JWT in the `Authorization` header.

6. Server has middleware that verifies the JWT's signature. If valid, it allows the request to proceed.

- **Installation:**

Bash

```
In 'server' directory
```

- `pnpm add jsonwebtoken bcryptjs`
- `pnpm add -D @types/jsonwebtoken @types/bcryptjs`
- 

## 2. Backend Implementation

- **User Model:** Add a `User` model in `schema.prisma` with `email` and `password` fields.
- **Register Endpoint (/api/auth/register):**  
TypeScript

```
import bcrypt from 'bcryptjs';
```

- `// ...`
- `const hashedPassword = await bcrypt.hash(password, 10); // 10 is the salt rounds`
- `const user = await prisma.user.create({`
- `data: { email, password: hashedPassword },`
- `});`
- `// ... then generate and return a JWT`
-

- **Login Endpoint (/api/auth/login):**  
TypeScript

- ```
import jwt from 'jsonwebtoken';
```
- `// ...`
 - `const user = await prisma.user.findUnique({ where: { email } });`
 - `const isPasswordValid = await bcrypt.compare(password,`
`user.password);`
 - `if (!user || !isPasswordValid) {`
 - `return res.status(401).json({ message: 'Invalid credentials' });`
 - `}`
 - `const token = jwt.sign({ userId: user.id }, process.env.JWT_SECRET,`
`{ expiresIn: '1d' });`
 - `res.json({ token });`
 -

- **Auth Middleware:**
TypeScript

- ```
// src/middleware/auth.ts
```
- `// This middleware checks for a valid JWT`
  - `export const protect = (req, res, next) => {`
  - `const token = req.headers.authorization?.split(' ')[1]; // Bearer TOKEN`
  - `if (!token) return res.status(401).json({ message: 'No token,`  
`authorization denied' });`
  - 
  - `try {`
  - `const decoded = jwt.verify(token, process.env.JWT_SECRET);`

- req.user = decoded; // Add user payload to the request object
- next();
- } catch (e) {
- res.status(401).json({ message: 'Token is not valid' });
- }
- };
- 
- // Applying it to a route
- // import { protect } from '../middleware/auth';
- router.get('/', protect, getAllItems); // This route is now protected
- 

### 3. Frontend Implementation

- **AuthContext:** Create a React Context to provide user state and login/logout functions throughout the app.  
TypeScript

```
// src/contexts/AuthContext.tsx
```

- const AuthContext = createContext(null);
- 
- export const AuthProvider = ({ children }) => {
- const [user, setUser] = useState(null); // Or load from localStorage
- // ... login, logout functions
- // login function would make API call, get token, save to localStorage, update state
- const value = { user, login, logout };
- return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>;
- };
- 
- export const useAuth = () => useContext(AuthContext);

- 

- **Storing the Token:**  
JavaScript

```
// Inside login function after successful API call
```

- `localStorage.setItem('authToken', token);`
- 

- **Sending the Token:** Use an `axios` interceptor to automatically add the `Authorization` header to every request.  
TypeScript

```
// src/api/apiClient.ts
```

- `apiClient.interceptors.request.use((config) => {`
- `const token = localStorage.getItem('authToken');`
- `if (token) {`
- `config.headers.Authorization = `Bearer ${token}`;`
- `}`
- `return config;`
- `});`
-

# Hands-on Lab: Secure Login System

**Objective:** Create a simple two-page React app (Login, Protected Dashboard) with a full backend authentication flow.

## 1. Backend:

- Create a `User` model in Prisma.
- Build `/register` and `/login` endpoints.
- Create a `/api/profile` endpoint that is protected by your `auth` middleware and returns the logged-in user's info.

## 2. Frontend:

- Create `LoginPage`, `RegisterPage`, and a `DashboardPage`.
- Set up an `AuthContext`.
- Implement the login form to call the `/login` endpoint, store the token, and redirect to the dashboard.
- Create a `ProtectedRoute` component that checks for a valid token before rendering its children, otherwise redirects to `/login`.
- On the `DashboardPage`, fetch data from the `/api/profile` endpoint.

## Team Project Work

**Goal:** Integrate a full user authentication system into your project.

### • All Projects:

7. **Backend:** Add a `User` model to your Prisma schema and migrate. Implement `/api/auth/register` and `/api/auth/login` endpoints.
8. **Backend:** Create an `auth` middleware and apply it to all existing CRUD routes (except login/register). Update your CRUD logic to associate data with the logged-in user (e.g., `req.user.userId`).
9. **Frontend:** Create `LoginPage` and `RegisterPage`.
10. **Frontend:** Implement an `AuthContext` to manage user state.
11. **Frontend:** Protect your main application pages so they are only accessible to logged-in users.
12. **Frontend:** Configure your `axios` instance to send the auth token on all requests.

### • Project Specific Roles:



1. **Project 2 (SupportDesk):** Add a `role` field to your `User` model (e.g., `'CUSTOMER'`, `'AGENT'`).
2. **Project 5 (ProcureNet):** Add a `role` field (e.g., `'EMPLOYEE'`, `'MANAGER'`). Create a middleware to check for a specific role.

## Daily Demo Checklist

- ☐ Team can demonstrate the user registration process.
- ☐ Team can demonstrate logging in and receiving a JWT.
- ☐ Attempting to access a protected API endpoint without a token (via Postman) results in a `401 Unauthorized` error.
- ☐ Accessing a protected frontend page while logged out redirects to the login page.
- ☐ After logging in, the user can access protected pages and data relevant to them.

## Homework

1. **Logout Functionality:** Implement a logout button that removes the token from `localStorage` and clears the auth state.
2. **Display User Info:** Show the logged-in user's email in the navbar.
3. **Persist Login:** When the app loads, check `localStorage` for a token. If it exists, validate it with the server (e.g., a `/me` endpoint) and set the user state, so they don't have to log in on every refresh.

## Short Quiz

1. **Question:** Why should you never store passwords in plain text?
  - **Answer:** Because if the database is ever compromised, all user passwords will be exposed. Hashing makes them effectively irreversible.
2. **Question:** What are the three parts of a JWT?
  - **Answer:** Header, Payload, and Signature.
3. **Question:** In Express, what is the primary role of middleware?
  - **Answer:** To execute code during the request-response cycle. It can access and modify `req` and `res`, and call the `next()` function to pass control to the next middleware. This is perfect for tasks like authentication.

## Resources

- **JWT:** [Introduction to JSON Web Tokens](#)
- **Bcrypt.js:** [npm package](#)
- **React Context:** [Official Docs](#)

## Instructor Notes

- **Pacing:** Auth is complex. There are many moving parts. The lab is crucial.
- **Troubleshooting:**
  - **Token issues:** Use [jwt.io](#) to decode tokens and check their contents.
  - **Header issues:** In the browser's Network tab, verify the `Authorization: Bearer <token>` header is being sent correctly.
  - **Environment Variables:** The `JWT_SECRET` must be kept secret and should be in your `.env` file. It must be the same for both signing and verifying.

## Day 7: Advanced Features & Specialization

**Summary:** Today, teams will dive deep into the specialized technologies that make their projects unique. We'll split into focused workshops to learn about interactive maps, real-time communication, and data visualization, and then spend the afternoon implementing these core features.

## Learning Objectives

- **(Maps Track):** Integrate the Leaflet library into a React app, display markers, and handle user interactions with the map.
- **(Real-time Track):** Understand the basics of WebSockets, set up a Socket.io server and client, and build a real-time event-based communication system.
- **(Data Viz Track):** Use the Recharts library to create meaningful charts (bar, line, pie) from API data in a React dashboard.

## Prerequisites

- Completion of Day 6 content.
- A fully authenticated, full-stack application.

## Agenda

- **09:00 - 09:15:** Daily Standup.
- **09:15 - 11:30: Focused Workshops (Concurrent)**
  - **Workshop A (Maps):** For Projects 1 (AssetTrack) & 3 (CoverageMapper).
  - **Workshop B (Real-time):** For Project 2 (SupportDesk).
  - **Workshop C (Data Viz):** For Projects 4 (BillFlow) & 5 (ProcureNet).
- **11:30 - 12:30: Focused Hands-on Labs** (within workshops).
- **12:30 - 13:30:** Lunch Break 🚀
- **13:30 - 17:30:** Team Project Work: Implementing the Advanced Feature.
- **17:30 - 18:00:** Daily Demos (Showcasing the new feature!).

## Workshop A: Interactive Maps with Leaflet

- **Target Audience:** Project 1 (AssetTrack), Project 3 (CoverageMapper).
- **Topics:** Introduction to GIS concepts (latitude, longitude). Setting up Leaflet with React using `react-leaflet`. Displaying a `MapContainer`, `TileLayer` (from `OpenStreetMap`). Adding `Markers` with custom icons and `Popups`. Handling map events like clicks.
- **Installation:** `pnpm add leaflet react-leaflet` and `@types/leaflet`.
- **Lab:** Build a simple "My Favorite Places" map where a user can click on the map to add a marker with a note.
- **Code Snippet (MapComponent.tsx):**  
`TypeScript`

- `import { MapContainer, TileLayer, Marker, Popup } from 'react-leaflet';`
- `import 'leaflet/dist/leaflet.css';`
- `const MapComponent = () => {`
- `const position = [51.505, -0.09]; // [lat, lng]`
- `return (`
- `<MapContainer center={position} zoom={13} style={{ height: '400px',`
- `width: '100%' }}>`
- `<TileLayer`
- `url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"`
- `attribution='&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors'`
- `/>`
- `<Marker position={position}>`
- `<Popup>A pretty CSS3 popup. <br /> Easily customizable.</`
- `Popup>`
- `</Marker>`
- `</MapContainer>`
- `);`
- `};`
- 

## Workshop B: Real-time with Socket.io

- **Target Audience:** Project 2 (SupportDesk).
- **Topics:** WebSockets vs. HTTP polling. Setting up a Socket.io server and integrating it with Express. Emitting events (`socket.emit`) and broadcasting events (`io.emit`). Listening for events on the client. Joining rooms for private communication (e.g., a chat room for a specific ticket).
- **Installation:** `pnpm add socket.io` (server) and `pnpm add socket.io-client` (client).
- **Lab:** Build a very simple group chat application.

- **Code Snippet (Server index.ts):**  
TypeScript

- ```
import { Server } from 'socket.io';
```
- `import http from 'http';`
 - `// ... after app setup`
 - `const server = http.createServer(app);`
 - `const io = new Server(server, { cors: { origin: "http://localhost:5173" } });`
 -
 - `io.on('connection', (socket) => {`
 - `console.log('a user connected:', socket.id);`
 - `socket.on('chat message', (msg) => {`
 - `io.emit('chat message', msg); // Broadcast to everyone`
 - `});`
 - `socket.on('disconnect', () => { console.log('user disconnected'); });`
 - `});`
 -
 - `server.listen(3001, /* ... */);`
 -

Workshop C: Data Visualization with Recharts

- **Target Audience:** Project 4 (BillFlow), Project 5 (ProcureNet).
- **Topics:** Importance of data visualization. Introduction to the Recharts library and its declarative components. Building common charts: `BarChart`, `LineChart`, and `PieChart`. Customizing charts with tooltips, legends, and labels. Fetching and transforming API data into a format Recharts can use.
- **Installation:** `pnpm add recharts`.
- **Lab:** Create a dashboard page that fetches sample data and displays it in a bar chart and a pie chart.
- **Code Snippet (SalesChart.tsx):**
TypeScript

```
import { BarChart, Bar, XAxis, YAxis, Tooltip, Legend, CartesianGrid }  
from 'recharts';
```

- ```
const data = [{ name: 'Jan', revenue: 4000 }, { name: 'Feb', revenue: 3000 }];
```

- 
- ```
const SalesChart = () => (  
•   <BarChart width={600} height={300} data={data}>  
•     <CartesianGrid strokeDasharray="3 3" />  
•     <XAxis dataKey="name" />  
•     <YAxis />  
•     <Tooltip />  
•     <Legend />  
•     <Bar dataKey="revenue" fill="#8884d8" />  
•   </BarChart>  
• );
```
-

Team Project Work

- **Project 1 (AssetTrack):** Integrate the Leaflet map on your dashboard. Fetch asset data (with latitude/longitude from your DB) and display each asset as a marker on the map. Clicking a marker should show a popup with asset details.
- **Project 2 (SupportDesk):** Implement the real-time chat feature. When a customer views a ticket, they should join a [socket.io](#) room for that ticket. Any messages sent by the customer or an agent should appear in real-time for the other party in the same room.
- **Project 3 (CoverageMapper):** Integrate the Leaflet map as the main page. Fetch all [CoveragePoint](#) data from your backend. Visualize the data on the map, perhaps using colored circles (e.g., green for strong signal, red for weak) instead of default markers.

- **Project 4 (BillFlow):** Create a dashboard page with at least two charts using Recharts. For example, a bar chart showing "Monthly Revenue" and a pie chart showing the "Distribution of Customers by Plan." You will need to create new API endpoints to provide this aggregated data.
- **Project 5 (ProcureNet):** Create a dashboard page showing procurement analytics. For example, a bar chart for "Spending per Vendor" and a line chart for "Purchase Orders Over Time." This will require creating new aggregate endpoints on your backend.

Daily Demo Checklist

- **AssetTrack & CoverageMapper:** ☐ A map is rendered on the page. ☐ Markers/ data points from the database are correctly displayed on the map. ☐ Markers are interactive (e.g., show a popup on click).
- **SupportDesk:** ☐ Two browser windows (one as customer, one as agent) can communicate in real-time on a ticket page. ☐ Messages appear instantly without a page refresh.
- **BillFlow & ProcureNet:** ☐ A dashboard page displays at least two different types of charts. ☐ The chart data is fetched from a real backend API endpoint. ☐ Charts have clear labels, tooltips, and are easy to understand.

Homework

1. **Polish:** Refine the advanced feature you built today. Add better styling, more interactivity, or handle edge cases.
2. **Explore:** Research a "stretch goal" for your feature. Map teams: clustered markers. Socket team: typing indicators. Chart teams: dynamic data filtering (e.g., by date range).
3. **Prepare:** Start thinking about the user flow for your final demo. What story will you tell?

Short Quiz

1. **Question (Maps):** In `react-leaflet`, what component is responsible for rendering the map background image (the tiles)?
 - **Answer:** The `<TileLayer>` component.

2. **Question (Real-time):** What is the difference between `socket.emit` and `io.emit` on a Socket.io server?
- o **Answer:** `socket.emit` sends an event only to the specific client that the `socket` object represents. `io.emit` broadcasts the event to *all* connected clients.
3. **Question (Data Viz):** In Recharts, what is the purpose of the `dataKey` prop on a component like `<Bar>` or `<XAxis>`?
- o **Answer:** It tells Recharts which property from the objects in your `data` array to use for that element (e.g., `dataKey="revenue"` on a `<Bar>` will use the `revenue` value for the bar's height).

Resources

- **React Leaflet:** [Official Docs](#)
- **Socket.io:** [Official Docs](#)
- **Recharts:** [Official Website](#)

Instructor Notes

- **Pacing:** The afternoon is pure project time. This is often the most exciting day for students as their apps start to feel "real." Circulate constantly, as each team will have very different problems.
- **Troubleshooting:**
 - o **Leaflet:** Map not showing up is often a CSS issue. Make sure `leaflet.css` is imported and the map container has a defined height.
 - o **Socket.io:** CORS errors are common. Ensure the `cors` options are configured correctly on the Socket.io server instance.
 - o **Recharts:** The most common issue is data formatting. The `data` prop expects an array of objects. Help students `console.log` their API response and structure it correctly.

Day 8: Testing, Deployment & Final Demos

Summary: The final day! We'll learn the essentials of testing, then focus on the ultimate goal: deploying your application to the web for anyone to see. The afternoon is dedicated to polishing your projects and preparing for the final presentations.

Learning Objectives

- Understand the basic principles of automated testing.
- Write a simple unit test for a React component using Vitest.
- Prepare a full-stack application for production deployment.
- Deploy a Node.js/Express backend to a platform like Render or Railway.
- Deploy a React frontend to a platform like Vercel or Netlify.
- Configure environment variables in production and troubleshoot common deployment issues.
- Prepare and deliver a compelling project demo.

Prerequisites

- A complete, working full-stack application from Day 7.

Agenda

- **09:00 - 09:15:** Daily Standup (Final strategy session).
- **09:15 - 10:00:** Topic: Introduction to Testing (Vitest).
- **10:00 - 11:30:** Topic & Walkthrough: Full-Stack Deployment.
- **11:30 - 12:30:** Team Work: Begin Deployment Process.
- **12:30 - 13:30:** Lunch & Demo Prep 🎤
- **13:30 - 16:00:** Final Polish & Demo Practice.
- **16:00 - 17:30: FINAL PROJECT DEMOS!** 🎉
- **17:30 - 18:00:** Wrap-up, Feedback, and Celebration.

Topics & Teaching Content

1. Introduction to Testing with Vitest

- **Explanation:** Writing tests helps ensure your code works as expected and prevents regressions (breaking existing features when you add new ones). **Vitest** is a modern test runner that works great with Vite. We'll write a simple **unit test** for a UI component.
- **Key Concepts:** Test runner, assertions (`expect`), component rendering, mocking.
- **Setup:**
Bash

```
# In 'client' directory
```

- `pnpm add -D vitest jsdom @testing-library/react`
-

Configure `vite.config.ts` and create a `setupTests.ts` file as per Vitest docs.

- **Code Example (Testing a simple component):**
TypeScript

```
// src/components/Button.test.tsx
```

- `import { render, screen } from '@testing-library/react';`
- `import { describe, it, expect } from 'vitest';`
- `import Button from './Button';`
-
- `describe('Button component', () => {`
- `it('renders the button with the correct text', () => {`
- `render(<Button>Click me</Button>);`

- `const buttonElement = screen.getByText(/Click me/i);`
- `expect(buttonElement).toBeInTheDocument();`
- `});`
- `});`
-

2. Full-Stack Deployment

- **Explanation:** Deployment is the process of taking your application from your local machine to a live server on the internet. We'll use separate services for our backend and frontend, which is a standard modern practice.
 - **Backend (e.g., Render):** Connects to your GitHub repo. Needs a "Build Command" (`pnpm install && pnpm build`) and a "Start Command" (`pnpm start`). It also needs your production `DATABASE_URL` and `JWT_SECRET` environment variables.
 - **Frontend (e.g., Vercel):** Also connects to GitHub. It will automatically detect a Vite project. The key is to set the `VITE_API_BASE_URL` environment variable to your *live backend URL*.
- **Backend start script (package.json):**
JSON

- ```
"scripts": {
 "build": "tsc",
 "start": "node dist/index.js",
 "dev": "nodemon src/index.ts"
}
```
- - 
  - 
  - 
  -

- **Key Steps:**
  - Push final code to GitHub.

- Sign up for Render and Vercel.
- Create a "New Web Service" on Render, connect your server repo.
- Configure Render: set build/start commands, add production environment variables.
- Wait for the Render deploy to finish. Copy the live URL.
- Create a "New Project" on Vercel, connect your client repo.
- Configure Vercel: add the `VITE_API_BASE_URL` environment variable, pasting the live Render URL.
- Deploy! Test the live Vercel site.

## Team Project Work (Final Push)

**Goal:** Deploy the application and prepare a flawless demo.

1. **Testing (Optional but encouraged):** Write one simple unit test for a key component.
2. **Deployment:** Follow the deployment checklist (Artifact F) carefully.
  - Deploy the backend first.
  - Test the live backend API with Postman.
  - Deploy the frontend.
  - Configure the frontend environment variable with the live backend URL.
  - Test the full, live application.
3. **Seeding:** Ensure your live database is seeded with good sample data for the demo. Render allows you to run `psql` commands.
4. **Demo Prep:**
  - Finalize your user story. Who is the user? What problem does your app solve?
  - Create a script. Walk through the main features clearly.
  - Assign speaking roles for each team member.
  - Practice the demo at least twice.

## Daily Demo Checklist (Final Presentation)

This is it! The final presentation will be graded using the rubric in **Artifact E**. The key areas are:

- ☐ **Clarity:** Was the problem statement and solution clear?
- ☐ **Functionality:** Did the application work as demonstrated? Were all core features implemented?
- ☐ **Technical Achievement:** Was the use of technology impressive? (e.g., real-time updates, map interactions, clean UI).
- ☐ **User Experience:** Was the app easy to use and visually appealing?
- ☐ **Deployment:** Is the application live and accessible via a public URL?

## Homework

- **Celebrate!** You've built and deployed a full-stack application in 8 days.
- **Update Resume/Portfolio:** Add this project to your portfolio. Write a good description of the architecture and your role.
- **Reflect:** Write down three things you learned, two things you struggled with, and one thing you want to learn next.

## Short Quiz

1. **Question:** What is the main difference between a unit test and an integration test?
  - **Answer:** A unit test checks a small, isolated piece of code (like a single function or component). An integration test checks how multiple pieces work together (e.g., does the frontend form correctly submit data to the backend API?).
2. **Question:** When deploying a frontend and backend separately, which one should you deploy first and why?
  - **Answer:** Deploy the backend first. You need the backend's live URL to correctly configure the frontend's API environment variable before you deploy it.
3. **Question:** What is the most critical information to move from your local `.env` file to the production environment variables on your hosting provider?
  - **Answer:** Secrets and configuration, such as the `DATABASE_URL` and `JWT_SECRET`.

## Resources

- **Vitest:** [Official Docs](#)
- **Render Deployment:** [Node.js Deploy Docs](#)
- **Vercel Deployment:** [Vite Deploy Docs](#)

## Instructor Notes

- **Pacing:** Deployment will take longer than expected. Start it early. Database connection issues and environment variable misconfigurations are the top problems.
- **Demo Mindset:** Encourage teams to focus on telling a story, not just listing features. The "why" is more important than the "what."
- **Contingency:** Have a plan for demos that fail due to deployment issues. The backup should be to demo on localhost. The effort and code quality are more important than a perfect live deploy under pressure.

## A. 5 Project Briefs

### 1. Project: AssetTrack - Telecom Asset Management Dashboard

- **Problem:** A telecom company needs a centralized dashboard to track the status and location of its physical assets (cell towers, fiber nodes, routers) and manage their maintenance schedules.
- **MVP Scope:**
  - Users (technicians, managers) can register and log in.
  - A dashboard view showing key stats (e.g., total assets, assets needing maintenance).
  - An interactive map view (Leaflet) displaying all assets as markers. Clicking a marker shows asset details.
  - A table view of all assets, sortable and searchable.
  - Users can add new assets, including their coordinates.
  - Users can log maintenance activity for an asset.

- **Data Models (schema.prisma):**

Code snippet

```
model User {
```

- id Int @id @default(autoincrement())
- email String @unique
- name String?
- role String @default("TECHNICIAN") // or "MANAGER"
- password String
- }
- model Asset {
- id Int @id @default(autoincrement())
- name String
- type String // "TOWER", "NODE", "ROUTER"
- status String // "ONLINE", "OFFLINE", "MAINTENANCE"
- latitude Float
- longitude Float
- lastMaintenance DateTime?
- nextMaintenance DateTime?
- logs MaintenanceLog[]
- }
- model MaintenanceLog {
- id Int @id @default(autoincrement())
- notes String
- createdAt DateTime @default(now())
- asset Asset @relation(fields: [assetId], references: [id])
- assetId Int
- technician User @relation(fields: [technicianId], references: [id])
- technicianId Int
- }
- 

- **Key API Endpoints:**

- POST /api/auth/register, POST /api/auth/login
- GET, POST /api/assets

- GET, PUT /api/assets/:id
  - GET, POST /api/assets/:id/logs
- **Success Metrics:** A manager can see all assets on a map and schedule a new maintenance task for a technician.

## 2. Project: SupportDesk - Customer Service Portal & Ticketing System

- **Problem:** Customers need an easy way to report issues and get real-time support. Support agents need an efficient way to manage and respond to these issues.
- **MVP Scope:**
  - Two user roles: Customer and Agent. Both can register and log in.
  - Customers can create a new support ticket (with a subject and description).
  - Customers can view a list of their own tickets and their status.
  - Agents can see a dashboard of all unassigned/open tickets.
  - When a customer or agent views a ticket, a real-time chat (Socket.io) interface appears, allowing them to communicate instantly.
- **Data Models (schema.prisma):**  
Code snippet

```
model User {
```

- id Int @id @default(autoincrement())
- email String @unique
- name String?
- password String
- role String @default("CUSTOMER") // or "AGENT"
- createdTickets Ticket[] @relation("CustomerTickets")
- assignedTickets Ticket[] @relation("AgentTickets")
- }
- model Ticket {
- id Int @id @default(autoincrement())
- subject String
- description String
- status String @default("OPEN") // "IN\_PROGRESS", "CLOSED"



- createdAt DateTime @default(now())
- customer User @relation("CustomerTickets", fields: [customerId], references: [id])
- customerId Int
- agent User? @relation("AgentTickets", fields: [agentId], references: [id])
- agentId Int?
- messages Message[]
- }
- model Message {
- id Int @id @default(autoincrement())
- content String
- createdAt DateTime @default(now())
- ticket Ticket @relation(fields: [ticketId], references: [id])
- ticketId Int
- sender User @relation(fields: [senderId], references: [id])
- senderId Int
- }
- 

- **Key API Endpoints:**
  - GET, POST /api/tickets (customers see their own, agents see all)
  - GET, PUT /api/tickets/:id (e.g., to assign an agent or close ticket)
  - GET, POST /api/tickets/:id/messages
- **Success Metrics:** A customer and agent can have a full conversation in real-time within the context of a single support ticket.

### 3. Project: CoverageMapper - Network Coverage Mapping Tool

- **Problem:** Potential customers want to know the quality of network coverage (5G, 4G, etc.) at their specific address before signing up for a service.
- **MVP Scope:**
  - A public-facing page with an interactive map (Leaflet).
  - A search bar where a user can enter an address (can be mocked with lat/lng for simplicity).
  - When searched, the map zooms to the location and displays an overlay or colored markers indicating network strength.

- o An admin-only area (login required) where an administrator can upload new coverage data points (e.g., via a form or CSV upload).
  - o A dashboard for admins showing basic stats (e.g., number of 5G vs 4G data points) using Recharts.
- **Data Models (schema.prisma):**  
Code snippet

```
model User { // For admins
```

- id Int @id @default(autoincrement())
- email String @unique
- password String
- }
- model CoveragePoint {
- id Int @id @default(autoincrement())
- latitude Float
- longitude Float
- networkType String // "5G", "4G", "LTE"
- signalStrength Int // e.g., 1 to 5, or -dBm
- }
- 

- **Key API Endpoints:**
  - o POST /api/auth/login (for admins)
  - o GET /api/coverage-points?lat=...&lng=... (public endpoint to get points in a specific area)
  - o POST /api/coverage-points (protected admin route)
  - o GET /api/coverage-stats (protected admin route for dashboard chart)
- **Success Metrics:** A non-logged-in user can visit the site, enter a location, and visually understand the network coverage in that area.

## 4. Project: BillFlow - Telecom Billing & Plan Management System

- **Problem:** A telecom provider needs an internal tool to manage customer accounts, assign them to billing plans, and track monthly invoices.
- **MVP Scope:**
  - Admin users can log in to a secure dashboard.
  - Admins can perform CRUD operations on Customers.
  - Admins can perform CRUD operations on Plans (e.g., name: "Pro 5G", price: 999, data: "100GB").
  - Admins can assign a Plan to a Customer, creating a Subscription.
  - A dashboard page showing key metrics with Recharts (e.g., Total Revenue, Subscriptions per Plan).
  - A page to view all Invoices, with the ability to filter by status (Paid, Due).
- **Data Models (schema.prisma):**  
Code snippet

- ```
model User { // For admins
  id      Int      @id @default(autoincrement())
  email   String   @unique
  password String
}

model Customer {
  id      Int      @id @default(autoincrement())
  name    String
  email   String   @unique
  subscriptions Subscription[]
  invoices Invoice[]
}

model Plan {
  id      Int      @id @default(autoincrement())
  name    String   @unique
  price   Float
  dataAllowance String
  subscriptions Subscription[]
}

model Subscription {
  id      Int      @id @default(autoincrement())
  customer Customer @relation(fields: [customerId], references: [id])
}
```

- customerId Int
- plan Plan @relation(fields: [planId], references: [id])
- planId Int
- isActive Boolean @default(true)
- }
- model Invoice {
- id Int @id @default(autoincrement())
- amount Float
- dueDate DateTime
- status String // "PAID", "DUE"
- customer Customer @relation(fields: [customerId], references: [id])
- customerId Int
- }
-

- **Key API Endpoints:**

- GET, POST, PUT /api/customers
- GET, POST, PUT /api/plans
- GET, POST /api/subscriptions
- GET /api/invoices
- GET /api/billing-stats

- **Success Metrics:** An admin can create a new customer, subscribe them to a plan, and see the corresponding invoice appear in the system.

5. Project: ProcureNet - Vendor & Equipment Procurement Portal

- **Problem:** Managing equipment procurement from multiple vendors is complex. A company needs a system to track purchase orders (POs) and manage a multi-step approval workflow.
- **MVP Scope:**
 - User roles: Employee and Manager. Both can log in.
 - Employees can create a new PurchaseOrder for a specific Vendor, adding LineItems (e.g., 10x "Router Model X").
 - When a PO is created, its status is "PENDING_APPROVAL".
 - Managers can see a list of all pending POs.
 - Managers can approve or reject a PO, changing its status.

- o A dashboard for managers shows analytics (Recharts) like "Total Spend by Vendor".
 - o Basic CRUD for Vendors.
- **Data Models (schema.prisma):**
Code snippet

```

model User {
  id          Int    @id @default(autoincrement())
  email       String @unique
  password    String
  role        String // "EMPLOYEE", "MANAGER"
  createdPOs  PurchaseOrder[] @relation("Requestor")
  approvedPOs PurchaseOrder[] @relation("Approver")
}

model Vendor {
  id          Int    @id @default(autoincrement())
  name        String @unique
  contactEmail String
  purchaseOrders PurchaseOrder[]
}

model PurchaseOrder {
  id          Int    @id @default(autoincrement())
  status      String // "PENDING_APPROVAL", "APPROVED",
  "REJECTED"
  createdAt   DateTime @default(now())
  requestor   User     @relation("Requestor", fields: [requestorId],
references: [id])
  requestorId Int
  approver    User?    @relation("Approver", fields: [approverId],
references: [id])
  approverId  Int?
  vendor      Vendor   @relation(fields: [vendorId], references: [id])
  vendorId    Int
  lineItems  LineItem[]
}

model LineItem {
  id          Int    @id @default(autoincrement())

```

- `itemName String`
- `quantity Int`
- `pricePerUnit Float`
- `purchaseOrder PurchaseOrder @relation(fields: [purchaseOrderId],`
`references: [id])`
- `purchaseOrderId Int`
- `}`
-
-
- **Key API Endpoints:**
 - `GET, POST /api/purchase-orders` (Employees see their own, Managers see all pending)
 - `PUT /api/purchase-orders/:id/approve` (Manager only)
 - `PUT /api/purchase-orders/:id/reject` (Manager only)
 - `GET, POST /api/vendors`
 - `GET /api/procurement-stats` (Manager only)
- **Success Metrics:** An employee can submit a PO, and a manager can log in, see it, and approve it, triggering a status change visible to the employee.

B. Team Assignment Template & Rotation Plan

- **Team Size:** 3-4 members. This is optimal for communication and ensures everyone has significant work to do.
- **Formation:**
 - Allow students to self-select if possible, but have a backup plan to assign teams based on a mix of perceived strengths (e.g., someone with prior design interest, someone with strong JS skills).
 - Announce teams on Day 1.
- **Initial Roles (to be rotated):**
 - **Frontend Lead:** Has the final say on component structure and styling for the day. Responsible for merging frontend PRs.
 - **Backend Lead:** Has the final say on API design and database schema for the day. Responsible for merging backend PRs.
 - **DevOps/Coordinator:** Responsible for managing the GitHub repo, ensuring the project board (if any) is updated, and leading the daily demo.

- **Rotation Plan:**
 - **Daily Rotation:** At the start of each day (Day 2 onwards), roles rotate. (e.g., Person A: FE -> BE, Person B: BE -> DevOps, Person C: DevOps -> FE).
 - **Rationale:** This forces everyone to get hands-on experience with every part of the stack and project management. It prevents knowledge silos.

C. PR Template and Code Review Checklist

Pull Request (PR) Template ([.github/pull_request_template.md](#))

Markdown

PR Description

A clear and concise description of what this PR does.

Related Task/Ticket

Link to the Trello card, Jira ticket, or GitHub issue.

Changes Made

- *_Change 1: e.g., Created the `POST /api/users` endpoint._*
- *_Change 2: e.g., Added `bcrypt` for password hashing._*
- *_Change 3: e.g., Built the registration form component._*

How to Test

1. *_Check out this branch._*
2. *_Run `pnpm install`._*
3. *_Start the server and client._*
4. *_Navigate to the `/register` page and create a new user._*
5. *_Verify the new user is created in the database with a hashed password._*

Screenshots/GIFs (if applicable)

Add a screenshot of the UI changes or a GIF of the new functionality.

Code Review Checklist

- ☐ My code follows the project's style guidelines.
- ☐ I have performed a self-review of my own code.
- ☐ I have commented on my code, particularly in hard-to-understand areas.
- ☐ I have made corresponding changes to the documentation.

Code Review Checklist (For the Reviewer)

- **Clarity:** Is the code easy to understand? Are variable names clear?
- **Correctness:** Does the code solve the intended problem? Are there any obvious bugs or edge cases missed?
- **Efficiency:** Is the code performant? Are there unnecessary database queries or loops?
- **Security:** Are there any security vulnerabilities (e.g., not hashing passwords, not validating input)?
- **Consistency:** Does the code match the existing style and patterns of the project?
- **Testing:** Does the PR include tests for the new functionality? (For this bootcamp, this is a "nice to have").

D. Daily Standup / Progress Report Template

Time: First 15 minutes of each day.

Format: Each team member answers three questions.

1. What did I complete yesterday?

- *Be specific. "I finished the `POST /api/assets` endpoint and connected it to the 'Add Asset' form" is better than "I worked on the backend."*

2. What will I work on today?

- *"Today, I will implement the JWT authentication middleware and protect the asset routes."*

3. What blockers are in my way?

- *"I'm blocked by a CORS error that I can't figure out." or "I'm not sure how to structure the Prisma schema for our many-to-many relationship." or "No blockers."*

E. Grading/Rubric for Final Demo

Category (20 points)	Needs Improvement (0-8)	Meets Expectations (9-15)	Exceeds Expectations (16-20)	Score
Core Functionality	Key features are missing or buggy. The app is not usable.	The MVP features are complete and work as expected. Minor bugs may exist.	MVP is flawless, and the team implemented one or more stretch goals.	/ 20
Technical Integration	The frontend and backend are poorly connected. Major data	Frontend and backend are connected. Auth works. Data is saved to the DB and	The team used advanced features (real-time, maps, etc.) effectively and correctly. The code is	/ 20
User Experience (UX/UI)	The UI is confusing, hard to use, or visually unappealing.	The app is intuitive and easy to navigate. Styling is clean and consistent with Tailwind.	The UI is polished, responsive, and includes thoughtful touches like loading spinners, error	/ 20
Deployment & Presentation	The app is not deployed, or the demo is disorganized.	The app is deployed live on the web. The demo clearly explains the problem and solution, and all	The deployment is smooth, and the demo is compelling, well-rehearsed, and tells a great user story.	/ 20
Telecom Impact & Theme	The project has a weak connection to the telecom theme.	The project clearly addresses a specific problem within the telecom industry as per the brief.	The project shows a deep understanding of the telecom problem space and presents a creative or particularly	/ 20
Total				/ 100

F. Deployment Checklist

Phase 1: Backend Deployment (Render/Railway)

- ☐ **1. package.json Scripts:**
 - Ensure you have a **build** script: `"build": "tsc"`.
 - Ensure you have a **start** script: `"start": "node dist/index.js"`.
- ☐ **2. Push to GitHub:** Make sure your latest working code is on the **main** branch.
- ☐ **3. Create New Web Service:** In Render, create a new Web Service and connect it to your **server** repository.
- ☐ **4. Configure Settings:**
 - **Runtime:** Node
 - **Build Command:** `pnpm install && pnpm run build` (or `npm install && npm run build`)
 - **Start Command:** `pnpm run start` (or `npm run start`)
- ☐ **5. Add Environment Variables (CRITICAL):**
 - Go to the "Environment" tab.
 - Add **DATABASE_URL**: Copy the value from your `.env` file (ensure it's your cloud Postgres instance, not localhost).
 - Add **JWT_SECRET**: Create a new, long, random string for production.
 - Add any other required API keys.
- ☐ **6. Deploy & Debug:**
 - Trigger the first deploy. Watch the logs for errors.
 - Common errors: build failures (`tsc` errors), runtime crashes (often due to missing env vars).
- ☐ **7. Test Live API:**
 - Once deployed, copy your new backend URL (e.g., `https://my-telecom-app.onrender.com`).
 - Use Postman to test a few endpoints (e.g., `GET /api/health`, `POST /api/auth/login`). **DO NOT PROCEED UNTIL THIS WORKS.**

Phase 2: Frontend Deployment (Vercel/Netlify)

- ☐ **1. Push to GitHub:** Make sure your latest working code for your client is on the **main** branch.
- ☐ **2. Create New Project:** In Vercel, create a new Project and connect it to your **client** repository.

- ☐ **3. Configure Settings:**
 - Vercel should automatically detect Vite and set the correct Build Command (`npm run build`) and Output Directory (`dist`).
- ☐ **4. Add Environment Variable (CRITICAL):**
 - Go to the project's "Settings" -> "Environment Variables".
 - Add `VITE_API_BASE_URL`.
 - The value should be your **live Render URL** from Phase 1, including the `/api` path.
 - Example: `https://my-telecom-app.onrender.com/api`
- ☐ **5. Deploy & Test:**
 - Trigger the deploy. Vercel will build and deploy your React app.
 - Open the live Vercel URL.
 - Test the entire user flow: registration, login, fetching data, creating new data.
 - Use the browser's Network tab to confirm that the frontend is making requests to the correct live backend URL.

Final Smoke Test

- ☐ Can a new user register and log in on the live site?
- ☐ Does data created by one user persist and appear correctly on a page refresh?
- ☐ Are all core features working as they did on localhost?