📦 Project Tech Stack Recap:

Layer	Tech Used	Purpose
Frontend	Next.js (App Router) + React	UI rendering, routing, client-server components
	Tailwind CSS	Styling and layout
	Lucide-react icons	Modern icons for UI
	Shadcn/ui	Clean, accessible, headless UI components (Button, Card)
Authentication	Firebase Authentication	User authentication, role management
Backend	Next.js API Routes	API endpoints (CRUD for projects, tasks, comments)
Database	MongoDB (via Mongoose)	Storing users, projects, tasks, comments, stats
Types	TypeScript	Strong typing, catching errors during dev time

---

📖 What You Just Shared (ProjectCard Component)

- Fetches and shows project name, description, status
- Shows project stats: total tasks, members, end date, and progress bar
- Admin-only dropdown menu for editing or deleting projects
- View details button linking to project-specific dashboard page
- Uses:
  - `useAuth` context for role-based features
  - fetch API call to delete project

   - DropdownMenu, Card, Progress components from shadcn/ui
   - Tailwind CSS utility classes for layout and design

📌 Frontend (Next.js, React, TypeScript)

Q: What is the difference between server components and client components in Next.js?
A:
Server components run only on the server, no client-side JS bundled, ideal for static content or server-side data fetching.
Client components (marked with `"use client"`) run in the browser, supporting interactivity (state, event listeners, hooks).
E.g., I used client components like ProjectCard because it uses `useAuth()` and event handlers.

Q: How does the `use client` directive work in Next.js App Router?
A:
Adding `"use client"` at the top marks the file to be bundled for client-side JS. It is necessary when using React hooks like `useState`, `useEffect`, or context consumers.

Q: Why did you use `useAuth` context instead of prop drilling?
A:
To avoid deeply passing props through many components, I created a global `useAuth` context to manage auth state and roles, making user info accessible globally.

Q: How did you manage type safety in your project?
A:
Using TypeScript interfaces and types for models like Project and ProjectStats, ensuring props and API data conform to expected structures at compile-time.

Q: What are controlled vs uncontrolled components in React?
A:
Controlled components have their state managed by React (`useState`).
Uncontrolled components rely on DOM internal state via refs.
I primarily use controlled components for inputs and stateful components for predictability.

Q: How does React's `useState` and `useEffect` work under the hood?
A:
`useState` manages state via React's fiber architecture internally.
`useEffect` queues side effects after render, like data fetching or DOM manipulation.

Q: How did you handle conditional rendering for admin features?
A:
Used `user?.role === "admin"` to render admin-only UI elements such as the Edit/Delete dropdown.

Q: What is the difference between SSR, SSG, ISR in Next.js?
A:

- **SSR:** Renders on every request (e.g., `getServerSideProps`)
- **SSG:** Pre-renders at build time (static)
- **ISR:** Static pages regenerated in the background on demand
  My app uses API routes and client fetch for dynamic, user-specific dashboards.

📌 Tailwind CSS & UI Libraries

Q: How is Tailwind CSS different from traditional CSS or CSS-in-JS?
A:
Tailwind is utility-first CSS embedded in markup, enabling rapid UI development without separate CSS files. This leads to faster, more maintainable, scalable UIs.

Q: How did you customize or extend the default Tailwind config?
A:
I extended `tailwind.config.ts` with custom colors and spacing utilities matching the app's design.

Q: What benefits does Shadcn/ui provide over other component libraries like Material UI?
A:

- Unstyled by default, built on Radix UI for accessibility
- Highly customizable, not opinionated like MUI
- Integrates smoothly with Tailwind CSS

📌 Backend (Next.js API Routes + MongoDB)

Q: How did you structure Next.js API routes for CRUD?
A:
Each entity (project, task) has its own route `/api/project` with REST methods (`GET`, `POST`, `PUT`, `DELETE`) handled via `req.method` switches.

Q: How do you connect to MongoDB from Next.js API routes?
A:
Created a reusable `connectDB.ts` utility using Mongoose, imported and called before any DB operations.

Q: What are Mongoose schemas and how did you define them?
A:
Schemas define data shape, types, and validation for MongoDB documents.
Example: Project schema has fields like name, description, status, assignedUsers, endDate.

Q: How did you handle API error handling and status codes?
A:
Used try-catch blocks, responded with HTTP status codes like 200 (success), 404 (not found), 500 (server error) along with meaningful JSON messages.

📌 Authentication (Firebase Authentication)

Q: Why did you choose Firebase Authentication over Clerk or NextAuth.js?
A:
Firebase Auth provides:

- Easy integration with email/password, social logins (Google, Facebook, etc.)
- Built-in user management and secure token handling
- Real-time user state management
- Rich SDKs for frontend and backend use
- Wide community support and scalability

Q: How did you set up role-based access control with Firebase Auth?
A:
I used Firebase Custom Claims to assign roles (e.g., `admin`, `user`) on user accounts.
On signup or via Firebase Admin SDK, I set these claims.
In the frontend, after login, I fetch the token with claims and store roles in auth context for role-based UI rendering and API route protection.

Q: How does Firebase protect your API routes and restrict admin actions?
A:
On API routes, I verify Firebase ID tokens using the Firebase Admin SDK, validating the user identity and extracting custom claims for roles.
Only users with the `admin` role in their claims can perform sensitive actions like project deletion.

📌 TypeScript

Q: What are TypeScript interfaces and how did you use them?
A:
Interfaces define object shapes.
Example:

ts
CopyEdit
```
interface Project {
  _id: string;
  name: string;
  description: string;
  status: string;
  assignedUsers: string[];
}
```

Used these interfaces for component props and API response types to enforce structure.

Q: How does TypeScript improve maintainability?
A:

- Prevents type bugs at compile-time
- Makes code self-documenting
- Improves autocompletion and developer tooling

Q: How did you handle optional props?
A:
Used `?` operator:

ts
CopyEdit
```
interface ProjectCardProps {
  stats?: ProjectStats;
}
```

Allows `stats` to be optional without type errors.

📌 Project Management Concept Questions

Q: How would you calculate project completion percentage?
A:

ts
CopyEdit
```
(completedTasks / totalTasks) * 100
```

My `ProjectStats` returns this completion rate, which is displayed with a progress bar.

Q: How would you structure a task dependency system?
A:
Add a `dependencies` field (array of task IDs) in task documents.
Before marking a task complete, ensure all dependencies are completed.

Q: How would you add real-time task updates?
A:
Implement WebSockets (Socket.IO) or Next.js Server Actions with React Server Components.
Alternatively, poll the API periodically for updates.

📌 Bonus System Design Questions

Q: How would you scale this app for thousands of concurrent users?
A:

- Horizontally scale Next.js instances behind a load balancer
- Use MongoDB Atlas with sharding and replica sets
- Add Redis caching for hot data
- Use CDN for static assets

Q: How would you structure your MongoDB collections?
A:
Collections:

- users
- projects (with assignedUser IDs array)
- tasks (linked to projects)
- comments (linked to tasks)

Add indexes on `projectId`, `assignedUsers`, `endDate` for performance.

Q: How would you implement optimistic UI updates for deleting a project?
A:
Remove the project card from UI immediately before API confirmation.
If the delete fails, rollback by restoring the card and showing an error toast.