ProjectHub - Phase 1 Development Guide (Updated)

Project Overview

ProjectHub is an internal client-project management tool designed to centralize project tracking, team collaboration, and resource management within an organization. It replaces fragmented communication tools with a unified, project-centric platform.

Core Objectives

- Centralize all client and project data in a single platform
- Enable efficient assignment and tracking of developers across projects
- Streamline daily reporting and project documentation
- Facilitate real-time team communication within project context
- Organize project assets and resources systematically

Success Criteria for Phase 1

- All Phase 1 features implemented and functional
- Dynamic role system correctly restricts access based on configuration
- Real-time chat operates without latency issues
- File uploads work within defined size limits (5-10 MB)
- Developer dashboard accurately reflects pending items and actions
- Application meets all coding standards and quality requirements

Tech Stack

Frontend

- **Framework**: Next.js 16.1.0
- **Language**: TypeScript
- **Styling**: Tailwind CSS with `cn()` utility
- **UI Components**: shadcn/ui (all components must be reusable)
- **Forms & Validation**: React Hook Form + Zod

Backend

- **API Layer**: Next.js API Routes
- **ORM**: Drizzle
- **Database**: PostgreSQL
- **Real-time**: Socket.io

Authentication & Security

- **Auth Library**: Better Auth

External Services

- **Media Storage**: Cloudinary
- **Email Templates**: React Email
- **Email Service**: Nodemailer
- **SMTP Provider**: Gmail

Infrastructure

- **Hosting**: Vercel
- **Package Manager**: pnpm

Development Tools

- ESLint + Prettier (Code Quality & Formatting)
- Husky + lint-staged (Pre-commit Hooks)

Database Schema

Better Auth Core Tables (Required)

These tables are automatically managed by Better Auth. Generate them using `npx @better-auth/cli generate`.

User Table

```
{
  id: string (primary key)
  name: string
  email: string (unique)
  emailVerified: boolean
  image: string | null
  createdAt: timestamp
  updatedAt: timestamp
}
```

Note: Better Auth stores user credentials. The password field is stored in the `account` table, not directly in the user table.

Session Table

```
{
  id: string (primary key)
  expiresAt: timestamp
  token: string (unique)
  createdAt: timestamp
  updatedAt: timestamp
  ipAddress: string | null
  userAgent: string | null
  userId: string (foreign key -> user.id)
}
```

Account Table

```
{
  id: string (primary key)
  accountId: string
  providerId: string (e.g., "credential", "google", "github")
  userId: string (foreign key -> user.id)
  accessToken: string | null
  refreshToken: string | null
  idToken: string | null
  accessTokenExpiresAt: timestamp | null
  refreshTokenExpiresAt: timestamp | null
  scope: string | null
  password: string | null (hashed password for credential provider)
  createdAt: timestamp
  updatedAt: timestamp
}
```

Important: Passwords are stored here (in the account table with `providerId: "credential"`), NOT in the user table. This allows users to have multiple auth methods.

Verification Table

```
{
  id: string (primary key)
  identifier: string (email or phone)
  value: string (verification token)
  expiresAt: timestamp
  createdAt: timestamp | null
  updatedAt: timestamp | null
}
```

Purpose: Used for email verification, password resets, and other one-time verification flows.

ProjectHub Custom Tables

Roles Table

```
{
  id: string (primary key)
  name: string (unique) // e.g., "admin", "developer", "tester",
"designer", "project_manager"
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Define available roles in the system. Role names are simple strings like "admin", "developer", "tester", "designer", "project_manager", etc.

Role Management:

- Roles are stored as simple string names
- Multiple roles can be assigned to users via the `user_roles` junction table
- Frontend handles route protection based on assigned roles
- Common roles: `admin`, `developer`, `tester`, `designer`, `project_manager`, `client_manager`, `qa_engineer`, etc.

Examples:

- Admin: Full access to all routes and features
- Developer: Access to projects, tasks, memos, EODs, chat
- Tester: Access to projects, tasks, bugs
- Designer: Access to projects, assets, design reviews

User Roles Table (Junction Table)

```
{
  id: string (primary key)
  user_id: string (foreign key -> user.id)
  role_id: string (foreign key -> roles.id)
  assigned_at: timestamp
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Many-to-many relationship between users and roles. A user can have multiple roles (e.g., both "developer" and "tester").

Unique Constraint: `(user_id, role_id)` to prevent duplicate role assignments.

Clients Table

```
{
  id: string (primary key)
  name: string
  email: string | null
  description: text | null
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Store client information. Each client can have multiple projects.

Projects Table

```
{
  id: string (primary key)
  name: string
  client_id: string (foreign key -> clients.id)
  total_time: number | null // in hours
  completed_time: number | null // in hours
  status: string | null // e.g., "active", "completed", "on-hold"
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Projects belong to clients and serve as the main organizational unit.

User Project Assignment Table (Junction Table)

```
{
  id: string (primary key)
  user_id: string (foreign key -> user.id)
  project_id: string (foreign key -> projects.id)
  assigned_at: timestamp
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Many-to-many relationship between users and projects. Tracks which developers/team members are assigned to which projects.

Unique Constraint: `(user_id, project_id)` to prevent duplicate assignments.

Tasks Table

```
{
  id: string (primary key)
  name: string
  description: text | null
  status: string // "todo", "in_progress", "done"
  deadline: timestamp | null
  estimated_time: number | null // in hours
  completed_time: number | null // in hours
  project_id: string (foreign key -> projects.id)
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Track tasks within projects (Phase 2 feature, but schema included for future).

User Task Assignment Table (Junction Table)

```
{
  id: string (primary key)
  user_id: string (foreign key -> user.id)
  task_id: string (foreign key -> tasks.id)
  assigned_at: timestamp
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Many-to-many relationship for task assignments. Multiple users can be assigned to a single task.

Unique Constraint: `(user_id, task_id)` to prevent duplicate task assignments.

EOD Reports Table

```
{
  id: string (primary key)
  user_id: string (foreign key -> user.id)
  project_id: string (foreign key -> projects.id)
  report_date: date
  client_update: text // Client-facing summary
  actual_update: text // Internal detailed update
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: End-of-Day reports with both internal and client-facing versions.

Unique Constraint: `(user_id, project_id, report_date)` to ensure one EOD per developer per project per day.

Memos Table

```
{
  id: string (primary key)
  memo_content: string (max 140 chars)
  user_id: string (foreign key -> user.id)
  project_id: string (foreign key -> projects.id)
  report_date: date
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Quick daily 140-character updates from developers.

Unique Constraint: `(user_id, project_id, report_date)` to ensure one memo per developer per project per day.

Business Rule: Can only edit on the same day (enforce in application logic using `created_at` date check).

Links Table

```
{
  id: string (primary key)
  name: string
  url: text
  description: text | null
  project_id: string (foreign key -> projects.id)
  client_id: string (foreign key -> clients.id)
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Store project and client-related URLs.

Assets Table

```
{
  id: string (primary key)
  name: string
  file_url: text // Cloudinary URL
  file_type: string // e.g., "image/png", "application/pdf"
  file_size: number // in bytes
  project_id: string (foreign key -> projects.id)
  client_id: string (foreign key -> clients.id)
  uploaded_by: string (foreign key -> user.id)
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Store project and client assets (files uploaded to Cloudinary).

Chat Groups Table

```
{
  id: string (primary key)
  name: string
  project_id: string (foreign key -> projects.id, unique)
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Each project has one chat group. All assigned team members + admins can participate.

Unique Constraint: `project_id` (one chat group per project).

Messages Table

```
{
  id: string (primary key)
  sender_id: string (foreign key -> user.id)
  group_id: string (foreign key -> chat_groups.id)
  content: text
  created_at: timestamp
  updated_at: timestamp
}
```

Purpose: Store chat messages for project groups.

Database Relationships Diagram

```
user (Better Auth)
├── sessions (1:many)
├── accounts (1:many)
├── user_roles (1:many) ──→ roles
├── user_project_assignments (1:many) ──→ projects
├── user_task_assignments (1:many) ──→ tasks
├── eod_reports (1:many)
├── memos (1:many)
├── messages (1:many)
└── assets.uploaded_by (1:many)

roles
└── user_roles (1:many) ──→ user
```

```
clients
├── projects (1:many)
├── links (1:many)
└── assets (1:many)

projects
├── client_id → clients
├── user_project_assignments (1:many) ──→ user
├── tasks (1:many)
├── eod_reports (1:many)
├── memos (1:many)
├── links (1:many)
├── assets (1:many)
└── chat_groups (1:1)

tasks
└── user_task_assignments (1:many) ──→ user

chat_groups
├── project_id → projects (unique)
└── messages (1:many)
```

Phase 1 - Core Features

1. Dynamic Role & Access Management

Overview: Admins can create custom roles and assign them to users. Multiple roles can be assigned to a single user. Route protection is handled on the frontend based on user roles.

Requirements:

- Create, update, and delete custom roles
- Assign multiple roles to users via junction table
- Store roles as simple string names (e.g., "admin", "developer", "tester")
- Frontend handles route protection based on assigned roles
- Role-based component rendering (show/hide features based on roles)

Common Roles:

- `admin`: Full system access
- `developer`: Development-related features
- `tester`: QA and testing features
- `designer`: Design and assets management
- `project_manager`: Project oversight and reporting
- `client_manager`: Client relationship management

Technical Implementation:

- Use Better Auth for authentication
- Create `roles` table and `user_roles` junction table
- Implement client-side role checking hooks
- Create reusable `ProtectedRoute` component

- Use `useAuth` hook to check user roles

Frontend Role Protection Pattern:

```
// hooks/use-auth.ts
export function useAuth() {
  const { data: session } = useSession();
  const userRoles = session?.user?.roles || []; // Array of role names

  const hasRole = (role: string) => userRoles.includes(role);
  const hasAnyRole = (roles: string[]) => roles.some(role =>
userRoles.includes(role));
  const hasAllRoles = (roles: string[]) => roles.every(role =>
userRoles.includes(role));

  return { user: session?.user, userRoles, hasRole, hasAnyRole, hasAllRoles
};
}

// components/protected-route.tsx
export function ProtectedRoute({
  children,
  allowedRoles
}: {
  children: React.ReactNode;
  allowedRoles: string[]
}) {
  const { hasAnyRole } = useAuth();

  if (!hasAnyRole(allowedRoles)) {
    return <UnauthorizedPage />;
  }

  return <>{children}</>;
}

// Usage in pages
<ProtectedRoute allowedRoles={["admin", "project_manager"]}>
  <AdminDashboard />
</ProtectedRoute>
```

---

2. Client Management

Overview: A centralized registry of all clients with relevant business information.

Requirements:

- CRUD operations for clients (Create, Read, Update, Delete)
- Each client can have multiple projects
- Store client business information (name, email, description)
- List view with search and filter capabilities
- Pagination for large datasets (table format)

Key Fields:

- Client name (required)
- Email (optional)
- Description (optional)
- Associated projects (relationship)
- Created/Updated timestamps

UI Requirements:

- **Use shadcn/ui components with `cn()` utility for all styling**
- Table-based listing with pagination (use `Table` component)
- Search by client name/email (use `Input` component)
- Filter options (use `Select` or `DropdownMenu`)
- Add/Edit client forms with validation (use `Form` components)
- Loading states (use `Skeleton` components)
- Confirmation dialogs for deletion (use `AlertDialog` component)

Reusable Components:

```
// components/tables/clients-table.tsx
import { Table, TableBody, TableCell, TableHead, TableHeader, TableRow }
from "@/components/ui/table"
import { Button } from "@/components/ui/button"
import { cn } from "@/lib/utils"

export function ClientsTable({ clients, onEdit, onDelete }) {
  return (
    <Table>
      <TableHeader>
        <TableRow>
          <TableHead>Name</TableHead>
          <TableHead>Email</TableHead>
          <TableHead>Projects</TableHead>
          <TableHead className="text-right">Actions</TableHead>
        </TableRow>
      </TableHeader>
      <TableBody>
        {clients.map((client) => (
          <TableRow key={client.id}>
            <TableCell className={cn("font-
medium")}>{client.name}</TableCell>
            <TableCell>{client.email}</TableCell>
            <TableCell>{client.projectCount}</TableCell>
            <TableCell className="text-right">
              <Button variant="ghost" size="sm" onClick={() =>
onEdit(client)}>
                Edit
              </Button>
              <Button variant="ghost" size="sm" onClick={() =>
onDelete(client)}>
                Delete
              </Button>
            </TableCell>
          </TableRow>
        ))}
      </TableBody>
    </Table>
  )
}
```

---

3. Project Management

Overview: Projects are created under clients and serve as the primary organizational unit.

Requirements:

- CRUD operations for projects
- Each project belongs to one client
- Multiple team members can be assigned to a project via junction table
- Track total time and completed time
- Track project status (active, completed, on-hold)
- All project features (memos, EODs, tasks, chats, assets, links) are scoped to project level

Key Fields:

- Project name (required)
- Client (foreign key, required)
- Total time allocation (optional, in hours)
- Completed time (optional, in hours)
- Status (optional: active, completed, on-hold)
- Assigned team members (via `user_project_assignments`)
- Created/Updated timestamps

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**
- Table-based listing with pagination
- Filter by client, status, assigned members (use `Select` components)
- Search by project name (use `Input` component)
- Project detail view showing:
  - Basic info with status badge (use `Badge` component)
  - Assigned team members list with avatars (use `Avatar` component)
  - Recent activity
  - Quick access tabs: Memos | EODs | Chat | Assets | Links (use `Tabs` component)
- Add/Edit project forms (use `Form`, `Input`, `Select` components)
- Loading states (use `Skeleton` components)

Reusable Components:

```
// components/tables/projects-table.tsx
import { Badge } from "@/components/ui/badge"
import { cn } from "@/lib/utils"

export function ProjectsTable({ projects, className }) {
  return (
    <div className={cn("space-y-4", className)}>
```

```
      {/* Table implementation */}
    </div>
  )
}

// components/project-status-badge.tsx
export function ProjectStatusBadge({ status }: { status: string }) {
  return (
    <Badge
      className={cn(
        status === "active" && "bg-green-500",
        status === "completed" && "bg-blue-500",
        status === "on-hold" && "bg-yellow-500"
      )}
    >
      {status}
    </Badge>
  )
}
```

---

4. Developer Assignment

Overview: Admins can assign multiple team members to projects via the junction table, granting access to all project features.

Requirements:

- Assign multiple team members to a single project
- Remove/reassign team members as needed
- Team members automatically gain access to project features upon assignment
- Track assignment history with `assigned_at` timestamp
- Prevent duplicate assignments (use unique constraint)
- When a team member is assigned:
  - Create/grant access to project chat group
  - Send email notification about assignment

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**
- Multi-select dropdown for member assignment (use `MultiSelect` or custom component)
- Visual list of currently assigned members with avatars (use `Avatar` component)
- Ability to remove members from project with confirmation (use `AlertDialog`)
- Show assignment date for each member
- Real-time updates when assignments change
- Display number of assigned members in project list

Reusable Components:

```
// components/project-team-members.tsx
import { Avatar, AvatarFallback, AvatarImage } from
"@/components/ui/avatar"
import { cn } from "@/lib/utils"
```

```
export function ProjectTeamMembers({ members, onRemove, className }) {
  return (
    <div className={cn("flex flex-wrap gap-2", className)}>
      {members.map((member) => (
        <div key={member.id} className="flex items-center gap-2 rounded-md
border p-2">
          <Avatar>
            <AvatarImage src={member.image} />
            <AvatarFallback>{member.name[0]}</AvatarFallback>
          </Avatar>
          <span>{member.name}</span>
          <Button
            variant="ghost"
            size="sm"
            onClick={() => onRemove(member.id)}
          >
            Remove
          </Button>
        </div>
      ))}
    </div>
  )
}
```

---

5. Project Memos

Overview: Quick daily updates from team members (140 characters max).

Requirements:

- Character limit: Exactly 140 characters maximum
- Frequency: One memo per user per project per day
- Editability: Can only edit on the same day it was created
- Past memos: Read-only for previous days
- Admin view: See all memos with filters
- User view: See only their own memo history

Database Constraint:

- Unique constraint on `(user_id, project_id, report_date)`

Validation Rules:

- Maximum 140 characters
- One memo per day per user per project
- Cannot edit memos from previous days
- Required field validation

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**

- Character counter showing remaining chars (use `Input` or `Textarea` with custom counter)
- Clear indication when memo is no longer editable (disabled state)
- Date-based filtering for admin view (use `DatePicker` component)
- Table view for admin with pagination (use `Table` component)
- Simple card/list view for users showing their history (use `Card` component)
- Quick "Add Today's Memo" button on dashboard (use `Button` component)
- Empty state when no memo exists (use custom empty state component)
- Visual indicator if memo already submitted for today (use `Badge` component)

Reusable Components:

```
// components/forms/memo-form.tsx
import { Textarea } from "@/components/ui/textarea"
import { cn } from "@/lib/utils"

export function MemoForm({ onSubmit, defaultValue, isEditable, className })
{
  const [content, setContent] = useState(defaultValue || "")
  const remaining = 140 - content.length

  return (
    <form onSubmit={onSubmit} className={cn("space-y-4", className)}>
      <div>
        <Textarea
          value={content}
          onChange={(e) => setContent(e.target.value)}
          maxLength={140}
          disabled={!isEditable}
          className={cn("resize-none")}
        />
        <p className={cn(
          "text-sm text-muted-foreground mt-1",
          remaining < 20 && "text-orange-500",
          remaining === 0 && "text-red-500"
        )}>
          {remaining} characters remaining
        </p>
      </div>
      <Button type="submit" disabled={!isEditable}>
        Submit Memo
      </Button>
    </form>
  )
}
```

---

6. EOD Reports

Overview: Mandatory daily End-of-Day reports with two types: internal and client-facing.

Requirements:

- Two report types:
  - Internal EOD (`actual_update`): Detailed technical updates for internal team

    - Client-facing EOD (`client_update`): Summarized progress for external sharing
- Mandatory daily submission
- One EOD per user per project per day
- Historical view of all submitted EODs

Database Constraint:

- Unique constraint on `(user_id, project_id, report_date)`

Validation Rules:

- Both internal and client-facing sections required
- One EOD per day per user per project
- Date validation (cannot submit for future dates)
- Minimum length requirements (e.g., 50 chars each)

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**
- Split form view with two sections or tabs (use `Tabs` component)
- Date picker for report date (use `DatePicker` component)
- Rich text editor or large textarea (use `Textarea` component)
- Submission confirmation (use `Toast` component)
- Historical EODs in table format (use `Table` component)
- Filter by date range, user, project (use `Select` and `DateRangePicker`)
- Export functionality (use `Button` with export icon)
- Modal to view full EOD details (use `Dialog` component)
- Loading states (use `Skeleton` components)
- "Submit EOD" quick action on dashboard (use `Button` component)
- Badge indicator if EOD not submitted for today (use `Badge` component)

Reusable Components:

```
// components/forms/eod-form.tsx
import { Tabs, TabsContent, TabsList, TabsTrigger } from
"@/components/ui/tabs"
import { Textarea } from "@/components/ui/textarea"
import { cn } from "@/lib/utils"

export function EODForm({ onSubmit, className }) {
  return (
    <form onSubmit={onSubmit} className={cn("space-y-4", className)}>
      <Tabs defaultValue="internal">
        <TabsList className="grid w-full grid-cols-2">
          <TabsTrigger value="internal">Internal Update</TabsTrigger>
          <TabsTrigger value="client">Client Update</TabsTrigger>
        </TabsList>
        <TabsContent value="internal">
          <Textarea
            placeholder="Detailed technical update..."
            className={cn("min-h-[200px]")}
          />
        </TabsContent>
        <TabsContent value="client">
          <Textarea
            placeholder="Client-facing summary..."
            className={cn("min-h-[200px]")}
          />
        </TabsContent>
      </Tabs>
      <Button type="submit">Submit EOD Report</Button>
    </form>
  )
}
```

---

7. Project Chat

Overview: Real-time messaging within project context using Socket.io.

Requirements:

- Real-time messaging with instant delivery
- Participants: All assigned team members + admins
- Chat history stored in database
- Searchable message history
- Project-scoped: Each project has ONE chat group (1:1 relationship)
- No latency in message delivery
- Auto-create chat group when project is created

Key Features:

- Send/receive messages in real-time
- Online/offline status of participants
- Message timestamps
- Load previous messages (pagination or infinite scroll)
- Search messages by content
- Typing indicators (optional)
- Unread message count (requires additional tracking)

Technical Implementation:

- Socket.io for real-time communication
- Store messages in `messages` table
- Group messages by `group_id`
- Socket events:
    - `message:send` - Send a message
    - `message:receive` - Receive a message
    - `user:typing` - Typing indicator
    - `user:online` - User comes online
    - `user:offline` - User goes offline
- Use Socket.io rooms for project-specific messaging

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**
- Chat interface with message list and input box
- Show sender name, avatar, and timestamp (use `Avatar` component)
- Auto-scroll to latest message
- Visual distinction between own messages and others (different alignment/colors)
- Loading state for message history (use `Skeleton` components)
- Connection status indicator (use `Badge` or custom component)
- Error handling for failed sends with retry option
- Search bar to filter messages (use `Input` component)
- Empty state for new chats
- "Scroll to bottom" button (use `Button` component)
- Online status indicators (use `Badge` or custom indicator)

Reusable Components:

```
// components/chat/chat-interface.tsx
import { ScrollArea } from "@/components/ui/scroll-area"
import { Input } from "@/components/ui/input"
import { Button } from "@/components/ui/button"
import { cn } from "@/lib/utils"

export function ChatInterface({ messages, onSendMessage, currentUserId,
className }) {
  return (
    <div className={cn("flex flex-col h-full", className)}>
      <ScrollArea className="flex-1 p-4">
        {messages.map((msg) => (
          <div
            key={msg.id}
            className={cn(
              "mb-4 flex",
              msg.senderId === currentUserId ? "justify-end" : "justify-
start"
            )}
          >
            <div className={cn(
              "rounded-lg p-3 max-w-[70%]",
              msg.senderId === currentUserId
                ? "bg-primary text-primary-foreground"
                : "bg-muted"
            )}>
              {msg.content}
            </div>
          </div>
        ))}
      </ScrollArea>
      <div className="border-t p-4">
        <div className="flex gap-2">
          <Input placeholder="Type a message..." />
          <Button onClick={onSendMessage}>Send</Button>
        </div>
      </div>
    </div>
  )
}
```

8. Links Management

Overview: Repository for project-related URLs (documentation, services, references).

Requirements:

- Add, edit, delete links
- Each link has: Name/Label, URL, Description, Project association, Client association (optional)
- Validate URLs before saving
- Categorization/tagging (optional enhancement)

Validation Rules:

- Valid URL format (use Zod URL validator)
- Required: name and URL
- Unique URLs per project (optional constraint)

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**
- Table view with columns: Name | URL | Description | Actions
- Add/Edit link modal or form (use `Dialog` component)
- Quick copy URL button (use `Button` with copy icon)
- Open link in new tab button (use `Button` with external link icon)
- Search and filter functionality (use `Input` and `Select`)
- Pagination for large lists
- Loading states (use `Skeleton` components)
- Confirmation for deletion (use `AlertDialog`)
- Empty state when no links exist

Reusable Components:

```
// components/tables/links-table.tsx
import { Button } from "@/components/ui/button"
import { ExternalLink, Copy } from "lucide-react"
import { cn } from "@/lib/utils"

export function LinksTable({ links, onEdit, onDelete, className }) {
  const handleCopy = (url: string) => {
    navigator.clipboard.writeText(url)
  }

  return (
    <Table className={cn(className)}>
      {/* Table implementation */}
      <Button
        variant="ghost"
        size="sm"
        onClick={() => handleCopy(link.url)}
      >
        <Copy className="h-4 w-4" />
      </Button>
      <Button
        variant="ghost"
        size="sm"
        onClick={() => window.open(link.url, '_blank')}
      >
        <ExternalLink className="h-4 w-4" />
      </Button>
    </Table>
  )
}
```

---

9. Assets Management

Overview: File storage for project assets using Cloudinary.

Requirements:

- Upload files up to 5-10 MB per file
- Supported file types: Images, Documents, Other relevant files
- Store files in Cloudinary
- Associate assets with projects and clients
- Preview for images
- Download functionality

Key Fields:

- Asset name (required)
- File URL (Cloudinary URL, required)
- File type/MIME type (required)
- File size in bytes (required)
- Project ID (required)
- Client ID (optional)
- Uploaded by user_id (required)
- Created/Updated timestamps

Technical Implementation:

- Cloudinary SDK for uploads
- File size validation (max 5-10 MB, enforce on client and server)
- File type validation (whitelist acceptable MIME types)
- Progress indicator during upload
- Store Cloudinary URLs in database
- Use Cloudinary transformations for image optimization

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**
- Drag-and-drop upload interface
- File browser with thumbnails for images
- Upload progress indicator (use `Progress` component)

- File list table with columns: Thumbnail | Name | Type | Size | Uploaded By | Upload Date | Actions
- Preview modal for images (use `Dialog` component)
- Download button (use `Button` component)
- Delete functionality with confirmation (use `AlertDialog`)
- Search and filter (use `Input` and `Select`)
- Pagination
- Loading states (use `Skeleton` components)
- Error handling for failed uploads
- Empty state when no assets exist

Reusable Components:

```
// components/shared/file-uploader.tsx
import { useCallback } from "react"
import { useDropzone } from "react-dropzone"
import { Upload } from "lucide-react"
import { Progress } from "@/components/ui/progress"
import { cn } from "@/lib/utils"

export function FileUploader({
  onUpload,
  maxSize = 10 * 1024 * 1024, // 10MB
  accept = { 'image/*': [], 'application/pdf': [] },
  className
}) {
  const [uploadProgress, setUploadProgress] = useState(0)

  const onDrop = useCallback(async (acceptedFiles: File[]) => {
    for (const file of acceptedFiles) {
      await onUpload(file, setUploadProgress)
    }
  }, [onUpload])

  const { getRootProps, getInputProps, isDragActive } = useDropzone({
    onDrop,
    maxSize,
    accept
  })

  return (
    <div className={cn("space-y-4", className)}>
      <div
        {...getRootProps()}
        className={cn(
          "border-2 border-dashed rounded-lg p-8 text-center cursor-pointer
transition-colors",
          isDragActive && "border-primary bg-primary/5",
          "hover:border-primary hover:bg-primary/5"
        )}
      >
        <input {...getInputProps()} />
        <Upload className="mx-auto h-12 w-12 text-muted-foreground mb-4" />
        <p className="text-sm text-muted-foreground">
          {isDragActive ? "Drop files here..." : "Drag & drop files here,
or click to select"}
        </p>
        <p className="text-xs text-muted-foreground mt-2">
```

```
          Max file size: {maxSize / 1024 / 1024}MB
        </p>
      </div>
      {uploadProgress > 0 && uploadProgress < 100 && (
        <Progress value={uploadProgress} className="w-full" />
      )}
    </div>
  )
}

// components/tables/assets-table.tsx
import { Table, TableBody, TableCell, TableHead, TableHeader, TableRow }
from "@/components/ui/table"
import { Button } from "@/components/ui/button"
import { Download, Trash2, Eye } from "lucide-react"
import { cn } from "@/lib/utils"

export function AssetsTable({ assets, onPreview, onDownload, onDelete,
className }) {
  const formatFileSize = (bytes: number) => {
    if (bytes < 1024) return bytes + ' B'
    if (bytes < 1024 * 1024) return (bytes / 1024).toFixed(2) + ' KB'
    return (bytes / 1024 / 1024).toFixed(2) + ' MB'
  }

  return (
    <Table className={cn(className)}>
      <TableHeader>
        <TableRow>
          <TableHead>Preview</TableHead>
          <TableHead>Name</TableHead>
          <TableHead>Type</TableHead>
          <TableHead>Size</TableHead>
          <TableHead>Uploaded By</TableHead>
          <TableHead>Upload Date</TableHead>
          <TableHead className="text-right">Actions</TableHead>
        </TableRow>
      </TableHeader>
      <TableBody>
        {assets.map((asset) => (
          <TableRow key={asset.id}>
            <TableCell>
              {asset.fileType.startsWith('image/') ? (
                <img
                  src={asset.fileUrl}
                  alt={asset.name}
                  className="h-10 w-10 rounded object-cover"
                />
              ) : (
                <div className="h-10 w-10 rounded bg-muted flex items-
center justify-center">
                  <FileIcon className="h-5 w-5" />
                </div>
              )}
            </TableCell>
            <TableCell className="font-medium">{asset.name}</TableCell>
            <TableCell>{asset.fileType}</TableCell>
            <TableCell>{formatFileSize(asset.fileSize)}</TableCell>
            <TableCell>{asset.uploadedBy.name}</TableCell>
            <TableCell>{new
Date(asset.createdAt).toLocaleDateString()}</TableCell>
```

```
            <TableCell className="text-right">
              <div className="flex justify-end gap-2">
                <Button
                  variant="ghost"
                  size="sm"
                  onClick={() => onPreview(asset)}
                >
                  <Eye className="h-4 w-4" />
                </Button>
                <Button
                  variant="ghost"
                  size="sm"
                  onClick={() => onDownload(asset)}
                >
                  <Download className="h-4 w-4" />
                </Button>
                <Button
                  variant="ghost"
                  size="sm"
                  onClick={() => onDelete(asset)}
                >
                  <Trash2 className="h-4 w-4" />
                </Button>
              </div>
            </TableCell>
          </TableRow>
        ))}
      </TableBody>
    </Table>
  )
}
```

---

10. Developer Dashboard

Overview: Personalized landing page for team members after login.

Requirements:

- Quick action buttons: Add Today's Memo, Submit EOD Report, View My Projects
- Pending/incomplete items list: Projects missing memo/EOD for today
- Show count of pending items
- Highlight overdue items
- Reminders: EOD submission reminder (after 5 PM), Memo submission reminder (at noon)
- Visual badges/notifications
- Recent activity (optional): Recent chat messages, Recent EODs submitted, Team updates

Dashboard Widgets:

1. **Quick Actions Card**
   - Button: "Add Today's Memo" (highlighted if not submitted)
   - Button: "Submit EOD Report" (highlighted if not submitted)
   - Button: "View All My Projects"

2. **Pending Items Card**
   - List of projects missing today's memo
   - List of projects missing today's EOD
   - Visual indicators: badges, warning icons
   - Click to navigate directly to submission form
3. **My Projects Card**
   - List of assigned projects (up to 5, then "View All")
   - Quick navigation to project details
   - Show active/inactive status
   - Show latest activity timestamp
4. **Recent Activity Card** (optional)
   - Recent unread chat messages (last 5)
   - Recent EODs submitted by teammates
   - Recent project assignments

UI Requirements:

- **Use shadcn/ui components with `cn()` utility**
- Grid layout for widgets (use `Card` components)
- Responsive design: 2 columns on desktop, 1 on mobile
- Loading skeletons for each widget (use `Skeleton` components)
- Empty states when no data
- Clear CTAs with contrasting colors (use `Button` components)
- Real-time updates for pending items
- Smooth animations/transitions
- Accessible navigation

Reusable Components:

```
// components/dashboard/quick-actions-card.tsx
import { Card, CardContent, CardDescription, CardHeader, CardTitle } from
"@/components/ui/card"
import { Button } from "@/components/ui/button"
import { Badge } from "@/components/ui/badge"
import { PlusCircle, FileText, FolderOpen } from "lucide-react"
import { cn } from "@/lib/utils"

export function QuickActionsCard({
  hasMemoToday,
  hasEODToday,
  onAddMemo,
  onSubmitEOD,
  onViewProjects,
  className
}) {
  return (
    <Card className={cn(className)}>
      <CardHeader>
        <CardTitle>Quick Actions</CardTitle>
        <CardDescription>Common tasks and shortcuts</CardDescription>
      </CardHeader>
      <CardContent className="space-y-3">
        <Button
          className={cn("w-full justify-start")}
          variant={hasMemoToday ? "outline" : "default"}
          onClick={onAddMemo}
        >
          <PlusCircle className="mr-2 h-4 w-4" />
          Add Today's Memo
          {!hasMemoToday && (
            <Badge variant="destructive" className="ml-auto">
              Pending
            </Badge>
          )}
        </Button>
        <Button
          className={cn("w-full justify-start")}
          variant={hasEODToday ? "outline" : "default"}
          onClick={onSubmitEOD}
        >
          <FileText className="mr-2 h-4 w-4" />
          Submit EOD Report
          {!hasEODToday && (
            <Badge variant="destructive" className="ml-auto">
              Pending
            </Badge>
          )}
        </Button>
        <Button
          className={cn("w-full justify-start")}
          variant="outline"
          onClick={onViewProjects}
        >
          <FolderOpen className="mr-2 h-4 w-4" />
          View All My Projects
        </Button>
      </CardContent>
    </Card>
  )
}

// components/dashboard/pending-items-card.tsx
import { Card, CardContent, CardDescription, CardHeader, CardTitle } from
"@/components/ui/card"
import { Alert, AlertDescription } from "@/components/ui/alert"
import { AlertCircle } from "lucide-react"
import { cn } from "@/lib/utils"

export function PendingItemsCard({ pendingMemos, pendingEODs, className })
{
  const hasPendingItems = pendingMemos.length > 0 || pendingEODs.length > 0

  return (
    <Card className={cn(className)}>
      <CardHeader>
        <CardTitle>Pending Items</CardTitle>
        <CardDescription>Tasks that need your attention</CardDescription>
      </CardHeader>
      <CardContent>
        {!hasPendingItems ? (
          <Alert>
            <AlertDescription className="text-green-600">
              🎉 All caught up! No pending items.
            </AlertDescription>
          </Alert>
        ) : (
          <div className="space-y-4">
            {pendingMemos.length > 0 && (
              <div>
                <h4 className="text-sm font-medium mb-2 flex items-center">
                  <AlertCircle className="mr-2 h-4 w-4 text-orange-500" />
                  Missing Memos ({pendingMemos.length})
                </h4>
                <ul className="space-y-2">
                  {pendingMemos.map((project) => (
                    <li
                      key={project.id}
                      className="text-sm text-muted-foreground hover:text-
foreground cursor-pointer"
                    >
                      {project.name}
                    </li>
                  ))}
                </ul>
              </div>
            )}
            {pendingEODs.length > 0 && (
              <div>
                <h4 className="text-sm font-medium mb-2 flex items-center">
                  <AlertCircle className="mr-2 h-4 w-4 text-red-500" />
                  Missing EODs ({pendingEODs.length})
                </h4>
                <ul className="space-y-2">
                  {pendingEODs.map((project) => (
                    <li
                      key={project.id}
                      className="text-sm text-muted-foreground hover:text-
foreground cursor-pointer"
                    >
                      {project.name}
                    </li>
                  ))}
                </ul>
              </div>
            )}
          </div>
        )}
      </CardContent>
    </Card>
  )
}

// components/dashboard/my-projects-card.tsx
import { Card, CardContent, CardDescription, CardHeader, CardTitle } from
"@/components/ui/card"
import { Badge } from "@/components/ui/badge"
import { Button } from "@/components/ui/button"
import { cn } from "@/lib/utils"

export function MyProjectsCard({ projects, onViewAll, className }) {
  return (
    <Card className={cn(className)}>
      <CardHeader>
        <CardTitle>My Projects</CardTitle>
        <CardDescription>Your active project assignments</CardDescription>
      </CardHeader>
      <CardContent>
```

```
        {projects.length === 0 ? (
          <p className="text-sm text-muted-foreground">No projects assigned
yet</p>
        ) : (
          <div className="space-y-3">
            {projects.slice(0, 5).map((project) => (
              <div
                key={project.id}
                className="flex items-center justify-between p-3 rounded-lg
border hover:bg-accent cursor-pointer transition-colors"
              >
                <div>
                  <p className="font-medium">{project.name}</p>
                  <p className="text-xs text-muted-
foreground">{project.clientName}</p>
                </div>
                <Badge
                  className={cn(
                    project.status === "active" && "bg-green-500",
                    project.status === "completed" && "bg-blue-500",
                    project.status === "on-hold" && "bg-yellow-500"
                  )}
                >
                  {project.status}
                </Badge>
              </div>
            ))}
            {projects.length > 5 && (
              <Button
                variant="outline"
                className="w-full"
                onClick={onViewAll}
              >
                View All ({projects.length})
              </Button>
            )}
          </div>
        )}
      </CardContent>
    </Card>
  )
}

// app/(dashboard)/page.tsx - Dashboard implementation
export default function DashboardPage() {
  return (
    <div className="container mx-auto p-6">
      <h1 className="text-3xl font-bold mb-6">Dashboard</h1>
      <div className="grid grid-cols-1 md:grid-cols-2 gap-6">
        <QuickActionsCard
          hasMemoToday={hasMemoToday}
          hasEODToday={hasEODToday}
          onAddMemo={() => router.push('/memos/new')}
          onSubmitEOD={() => router.push('/eods/new')}
          onViewProjects={() => router.push('/projects')}
        />
        <PendingItemsCard
          pendingMemos={pendingMemos}
          pendingEODs={pendingEODs}
        />
        <MyProjectsCard
```

```
          projects={myProjects}
          onViewAll={() => router.push('/projects')}
          className="md:col-span-2"
        />
      </div>
    </div>
  )
}
```

---

11. Email Notifications

Overview: Triggered email alerts for key events.

Requirements:

- Email notifications for:
  - **Project assignment**: When a user is assigned to a project
  - **Project removal**: When a user is removed from a project
  - **Critical updates**: Admin announcements or system notifications
  - **Password reset**: Better Auth handles this (built-in)
  - **Email verification**: Better Auth handles this (built-in)
- Use React Email for HTML email templates
- Use Nodemailer with Gmail SMTP for sending

Email Types:

1. **Project Assignment Email**
   - To: Assigned team member
   - Subject: "You've been assigned to [Project Name]"
   - Content:
     - Project name and description
     - Client name
     - List of other team members assigned
     - Link to project details page
     - Call-to-action: "View Project"
2. **Project Removal Email**
   - To: Removed team member
   - Subject: "You've been removed from [Project Name]"
   - Content:
     - Confirmation of removal
     - Project name
     - Reason (optional, if admin provides)
     - Contact info for questions
3. **Critical Update Email**
   - To: All relevant users (admins or specific team members)
   - Subject: Custom subject
   - Content: Custom message from admin
   - Use case: System announcements, policy updates, urgent notifications

Technical Implementation:

```
// lib/email/nodemailer-config.ts
import nodemailer from 'nodemailer'

export const transporter = nodemailer.createTransport({
  host: 'smtp.gmail.com',
  port: 587,
  secure: false,
  auth: {
    user: process.env.GMAIL_USER,
    pass: process.env.GMAIL_APP_PASSWORD // Use App Password, not regular
password
  }
})

// lib/email/send-email.ts
import { render } from '@react-email/render'

export async function sendEmail({
  to,
  subject,
  template
}: {
  to: string
  subject: string
  template: React.ReactElement
}) {
  const html = render(template)

  try {
    await transporter.sendMail({
      from: process.env.GMAIL_USER,
      to,
      subject,
      html
    })
    return { success: true }
  } catch (error) {
    console.error('Email send error:', error)
    return { success: false, error }
  }
}
```

Email Template Structure (React Email):

```
// emails/project-assignment.tsx
import {
  Html,
  Head,
  Body,
  Container,
  Heading,
  Text,
  Button,
  Section,
  Hr
} from '@react-email/components'

interface ProjectAssignmentEmailProps {
  userName: string
  projectName: string
  clientName: string
  projectUrl: string
  teamMembers: string[]
}

export default function ProjectAssignmentEmail({
  userName,
  projectName,
  clientName,
  projectUrl,
  teamMembers
}: ProjectAssignmentEmailProps) {
  return (
    <Html>
      <Head />
      <Body style={main}>
        <Container style={container}>
          <Heading style={h1}>You've been assigned to
{projectName}</Heading>
          <Text style={text}>Hi {userName},</Text>
          <Text style={text}>
            You've been assigned to the project
<strong>{projectName}</strong> for client{' '}
            <strong>{clientName}</strong>.
          </Text>
          <Section style={section}>
            <Text style={text}>
              <strong>Your team members:</strong>
            </Text>
            <ul>
              {teamMembers.map((member) => (
                <li key={member}>{member}</li>
              ))}
            </ul>
          </Section>
          <Hr style={hr} />
          <Button href={projectUrl} style={button}>
            View Project
          </Button>
          <Text style={footer}>
            This is an automated message from ProjectHub. Please do not
reply to this email.
          </Text>
        </Container>
      </Body>
    </Html>
  )
}

const main = {
  backgroundColor: '#f6f9fc',
  fontFamily: '-apple-system,BlinkMacSystemFont,"Segoe
UI",Roboto,"Helvetica Neue",Ubuntu,sans-serif'
}

const container = {
  backgroundColor: '#ffffff',
  margin: '0 auto',
  padding: '20px 0 48px',
  marginBottom: '64px'
}
```

```
const h1 = {
  color: '#333',
  fontSize: '24px',
  fontWeight: 'bold',
  margin: '40px 0',
  padding: '0'
}

const text = {
  color: '#333',
  fontSize: '16px',
  lineHeight: '26px'
}

const section = {
  padding: '24px',
  border: '1px solid #dedede',
  borderRadius: '5px',
  marginTop: '24px'
}

const button = {
  backgroundColor: '#5469d4',
  borderRadius: '5px',
  color: '#fff',
  fontSize: '16px',
  fontWeight: 'bold',
  textDecoration: 'none',
  textAlign: 'center' as const,
  display: 'block',
  width: '100%',
  padding: '12px'
}

const hr = {
  borderColor: '#e6ebf1',
  margin: '20px 0'
}

const footer = {
  color: '#8898aa',
  fontSize: '12px',
  lineHeight: '16px'
}

// emails/project-removal.tsx
export default function ProjectRemovalEmail({
  userName,
  projectName,
  reason
}: {
  userName: string
  projectName: string
  reason?: string
}) {
  return (
    <Html>
      <Head />
      <Body style={main}>
        <Container style={container}>
```

```
          <Heading style={h1}>Project Assignment Update</Heading>
          <Text style={text}>Hi {userName},</Text>
          <Text style={text}>
            You have been removed from the project
<strong>{projectName}</strong>.
          </Text>
          {reason && (
            <Section style={section}>
              <Text style={text}>
                <strong>Reason:</strong> {reason}
              </Text>
            </Section>
          )}
          <Hr style={hr} />
          <Text style={text}>
            If you have any questions about this change, please contact
your project manager or
            administrator.
          </Text>
          <Text style={footer}>
            This is an automated message from ProjectHub. Please do not
reply to this email.
          </Text>
        </Container>
      </Body>
    </Html>
  )
}

// Usage in API routes
// app/api/projects/[id]/assign/route.ts
import { sendEmail } from '@/lib/email/send-email'
import ProjectAssignmentEmail from '@/emails/project-assignment'

export async function POST(req: Request) {
  // ... assign user to project logic

  // Send email notification
  await sendEmail({
    to: user.email,
    subject: `You've been assigned to ${project.name}`,
    template: ProjectAssignmentEmail({
      userName: user.name,
      projectName: project.name,
      clientName: client.name,
      projectUrl:
`${process.env.NEXT_PUBLIC_APP_URL}/projects/${project.id}`,
      teamMembers: assignedUsers.map(u => u.name)
    })
  })

  return Response.json({ success: true })
}
```

Note: Chat notifications are handled via Socket.io real-time events, NOT email.

Quality Standards & Best Practices

1. Security

- Proper authentication using Better Auth
- Role-based authorization checks on EVERY route
- Input validation on both client AND server
- Sanitize user inputs to prevent XSS attacks
- Secure password hashing (handled by Better Auth)
- CSRF protection (Better Auth provides this)
- SQL injection prevention (use Drizzle ORM parameterized queries)
- Validate file uploads (type, size, malicious content)
- Environment variables for sensitive data
- Rate limiting for API endpoints

2. Performance

- Optimized load times (target: under 3 seconds)
- Server-side prefetching (use Next.js server components)
- Efficient real-time communication:
  - Use Socket.io rooms for project-scoped messaging
  - Implement message pagination
- Image optimization:
  - Use Cloudinary transformations
  - Use Next.js Image component
- Lazy loading for heavy components
- Code splitting (Next.js handles automatically)
- Caching strategies:
  - Cache static data with React Query or SWR
  - Cache session data with Better Auth
- Database query optimization:
  - Use indexes on foreign keys
  - Avoid N+1 queries (use Drizzle `with` for relations)

3. Code Quality

TypeScript Usage:

```
// Good
interface User {
  id: string
  name: string
  email: string
  roles: string[]
}

// Bad
const user: any = ...
```

- Proper types everywhere
- NEVER use `any` or `unknown` types
- Create interfaces and types for all data structures
- Use type inference where appropriate

- Strict mode enabled in `tsconfig.json`

Component Structure:

- Make components reusable
- Separate business logic from UI (custom hooks)
- Use composition over inheritance
- Keep components small and focused
- Use React Server Components where possible
- **Always use `cn()` utility for className management**

Next.js 15+ Patterns (MUST FOLLOW):

- Use `loading.tsx` for loading states at route level
- Use `error.tsx` for error boundaries
- Use `not-found.tsx` for 404 pages
- Use `layout.tsx` for shared layouts
- Use Suspense boundaries
- Server-side prefetching for data-heavy pages

```
// app/projects/page.tsx
import { Suspense } from 'react'
import { getProjects } from '@/lib/queries'
import { Skeleton } from '@/components/ui/skeleton'

export default async function ProjectsPage() {
  const projects = await getProjects() // Server-side fetch

  return (
    <Suspense fallback={<Skeleton className="h-96" />}>
      <ProjectsList projects={projects} />
    </Suspense>
  )
}
```

Data Handling:

- Use tables with pagination for large datasets (NOT cards)
- Implement server-side pagination
- Show page numbers and "Next/Previous" buttons
- Display total count
- Efficient query optimization
- Use database indexes

4. UI/UX Requirements

- Loading states for ALL async operations (use `Skeleton` components)
- Error messages clear and actionable
- Success feedback for user actions (use `Toast` components)
- Consistent design patterns using shadcn/ui
- Responsive design (mobile, tablet, desktop)
- Accessibility:

  - Proper semantic HTML
  - ARIA labels where needed
  - Keyboard navigation support
  - Color contrast compliance (WCAG AA)
  - Focus indicators visible

5. Form Handling

```
import { z } from 'zod'
import { useForm } from 'react-hook-form'
import { zodResolver } from '@hookform/resolvers/zod'

const memoSchema = z.object({
  content: z.string().max(140, 'Memo must be 140 characters or less'),
  projectId: z.string().uuid('Invalid project ID'),
  reportDate: z.date()
})

const form = useForm({
  resolver: zodResolver(memoSchema)
})
```

- React Hook Form for all forms
- Zod for validation schemas
- Client-side validation (immediate feedback)
- Server-side validation (ALWAYS validate on server)
- Clear error messages
- Disabled submit during submission
- Success/error feedback

6. Error Handling

```
try {
  const result = await submitEOD(data)
  toast({ title: "Success", description: "EOD submitted successfully" })
} catch (error) {
  toast({
    title: "Error",
    description: error.message || 'Failed to submit EOD',
    variant: "destructive"
  })
  console.error('EOD submission error:', error)
}
```

- Try-catch blocks for async operations
- User-friendly error messages
- Error logging for debugging
- Graceful failure modes
- Error boundaries (use `error.tsx`)
- Retry mechanisms for transient failures

7. State Management

- Use React hooks (`useState`, `useReducer`, `useContext`)

- Server state with proper caching (React Query or SWR)
- Optimistic updates where appropriate
- Prevent unnecessary re-renders
- Keep state as local as possible

8. Code Standards

- ESLint + Prettier configured
- Pre-commit hooks (Husky + lint-staged)
- Consistent naming conventions:
  - `camelCase` for variables, functions, hooks
  - `PascalCase` for components, types, interfaces
  - `UPPER_CASE` for constants
  - `kebab-case` for file names (except components)
- Meaningful variable names
- Comments for complex logic
- Remove console.logs before commit

Implementation Guidelines

Project Structure

```
projecthub/
├── app/
│   ├── (auth)/
│   │   ├── login/page.tsx
│   │   ├── register/page.tsx
│   │   └── layout.tsx
│   ├── (dashboard)/
│   │   ├── layout.tsx
│   │   ├── page.tsx (Developer Dashboard)
│   │   ├── clients/
│   │   ├── projects/
│   │   ├── memos/
│   │   ├── eods/
│   │   ├── roles/
│   │   └── settings/
│   ├── api/
│   │   ├── auth/[...all]/route.ts
│   │   ├── clients/
│   │   ├── projects/
│   │   ├── memos/
│   │   ├── eods/
│   │   ├── chat/
│   │   ├── assets/
│   │   └── links/
│   └── globals.css
├── components/
│   ├── ui/ (shadcn/ui components)
│   ├── forms/
│   ├── tables/
│   ├── chat/
│   ├── dashboard/
│   └── shared/
├── lib/
│   ├── db/
│   ├── auth/
│   ├── utils/
│   ├── validations/
│   ├── queries/
│   └── socket/
├── hooks/
├── types/
├── emails/
└── middleware.ts
```

Development Workflow

1. Set up Better Auth
2. Set up Database with Drizzle
3. Implement Role & Permission System
4. Build Core CRUD Operations
5. Implement Features
6. Build Developer Dashboard
7. Implement Email Notifications
8. Testing and Refinement

API Endpoints Reference

Authentication (Better Auth)

- `POST /api/auth/sign-in` - User login
- `POST /api/auth/sign-out` - User logout
- `POST /api/auth/sign-up` - User registration
- `GET /api/auth/session` - Get current session

Clients

- `GET /api/clients` - List all clients
- `GET /api/clients/:id` - Get single client
- `POST /api/clients` - Create client
- `PUT /api/clients/:id` - Update client
- `DELETE /api/clients/:id` - Delete client

Projects

- `GET /api/projects` - List all projects
- `GET /api/projects/:id` - Get single project
- `POST /api/projects` - Create project
- `PUT /api/projects/:id` - Update project
- `DELETE /api/projects/:id` - Delete project
- `POST /api/projects/:id/assign` - Assign user to project

- `DELETE /api/projects/:id/assign/:userId` - Remove user from project

Memos

- `GET /api/memos` - List memos
- `GET /api/memos/:id` - Get single memo
- `POST /api/memos` - Create memo
- `PUT /api/memos/:id` - Update memo (only same day)
- `DELETE /api/memos/:id` - Delete memo

EOD Reports

- `GET /api/eods` - List EOD reports
- `GET /api/eods/:id` - Get single EOD
- `POST /api/eods` - Create EOD report
- `PUT /api/eods/:id` - Update EOD report
- `DELETE /api/eods/:id` - Delete EOD

Chat

- `GET /api/chat/:projectId/messages` - Get chat history
- `POST /api/chat/:projectId/messages` - Send message
- **Socket.io Events**:
  - `message:send` - Send a message
  - `message:receive` - Receive a message
  - `user:typing` - Typing indicator
  - `user:online` - User comes online
  - `user:offline` - User goes offline

Links

- `GET /api/links` - List links
- `POST /api/links` - Create link
- `PUT /api/links/:id` - Update link
- `DELETE /api/links/:id` - Delete link

Assets

- `GET /api/assets` - List assets
- `POST /api/assets/upload` - Upload asset to Cloudinary
- `DELETE /api/assets/:id` - Delete asset

Roles & Permissions

- `GET /api/roles` - List all roles
- `GET /api/roles/:id` - Get single role
- `POST /api/roles` - Create role
- `PUT /api/roles/:id` - Update role

- `DELETE /api/roles/:id` - Delete role

Users (Admin only)

- `GET /api/users` - List all users
- `GET /api/users/:id` - Get single user
- `PUT /api/users/:id` - Update user
- `DELETE /api/users/:id` - Delete user
- `POST /api/users/:id/roles` - Assign role to user
- `DELETE /api/users/:id/roles/:roleId` - Remove role from user

Environment Variables

Create a `.env.local` file with the following variables:

```
# Database
DATABASE_URL=postgresql://user:password@localhost:5432/projecthub

# Better Auth
BETTER_AUTH_SECRET=your-secret-key-here
BETTER_AUTH_URL=http://localhost:3000

# Cloudinary
CLOUDINARY_CLOUD_NAME=your-cloud-name
CLOUDINARY_API_KEY=your-api-key
CLOUDINARY_API_SECRET=your-api-secret

# Email (Gmail SMTP)
GMAIL_USER=your-email@gmail.com
GMAIL_APP_PASSWORD=your-app-password

# Socket.io (Production)
SOCKET_IO_URL=https://your-app.vercel.app

# Next.js
NEXT_PUBLIC_APP_URL=http://localhost:3000
```

Deployment Checklist

Before Deployment

- ✅ All environment variables configured in Vercel
- ✅ Database migrations run on production database
- ✅ Cloudinary credentials set up and tested
- ✅ Gmail SMTP configured with App Password
- ✅ Better Auth secret generated
- ✅ Socket.io configured for production
- ✅ Build passes without errors: `pnpm build`

- ✅ ESLint passes with no errors: `pnpm lint`
- ✅ TypeScript compiles without errors: `pnpm type-check`
- ✅ All console.logs removed (except error logs)

Vercel Configuration

- Environment variables added to Vercel dashboard
- Database connection string added
- Build command: `pnpm build`
- Output directory: `.next`
- Install command: `pnpm install`
- Node version: 20.x

Database Setup

- PostgreSQL database provisioned
- Run migrations: `npx drizzle-kit migrate`
- Seed initial data (roles, admin user)
- Set up database backups

Post-Deployment

- Test all authentication flows
- Verify role-based access control
- Test file uploads
- Test real-time chat functionality
- Test email notifications
- Check performance and load times
- Monitor error logs
- Test on multiple devices and browsers

Support & Maintenance

Monitoring

- Error tracking: Set up Sentry or similar tool
- API monitoring: Monitor response times and error rates
- Socket.io: Track connection issues and message delivery
- Database: Monitor query performance
- Email: Track delivery rates and bounces

Regular Maintenance Tasks

- Database backups: Daily automated backups
- Clean up old assets: Remove unused files from Cloudinary
- Review slow queries: Optimize with indexes

- Update dependencies: Monthly security and feature updates
- Security patches: Apply immediately when available

Performance Optimization

- **Database indexes**: Add indexes on foreign keys and frequently queried fields

```
CREATE INDEX idx_projects_client_id ON projects(client_id);
CREATE INDEX idx_user_project_assignments_user_id ON
user_project_assignments(user_id);
CREATE INDEX idx_user_project_assignments_project_id ON
user_project_assignments(project_id);
CREATE INDEX idx_messages_group_id ON messages(group_id);
CREATE INDEX idx_eod_reports_user_project_date ON eod_reports(user_id,
project_id, report_date);
CREATE INDEX idx_memos_user_project_date ON memos(user_id, project_id,
report_date);
```

- **Caching**: Implement Redis for session caching (optional)
- **CDN**: Use Cloudinary's CDN for fast asset delivery
- **Image optimization**: Use Cloudinary transformations

Component Usage Examples

Using shadcn/ui with cn() utility

```
// Example 1: Basic Button with conditional styling
import { Button } from "@/components/ui/button"
import { cn } from "@/lib/utils"

export function SubmitButton({ isLoading, className }) {
  return (
    <Button
      className={cn(
        "w-full",
        isLoading && "opacity-50 cursor-not-allowed",
        className
      )}
      disabled={isLoading}
    >
      {isLoading ? "Submitting..." : "Submit"}
    </Button>
  )
}

// Example 2: Form with multiple components
import { Input } from "@/components/ui/input"
import { Label } from "@/components/ui/label"
import { Textarea } from "@/components/ui/textarea"
import { Select, SelectContent, SelectItem, SelectTrigger, SelectValue }
from "@/components/ui/select"
import { cn } from "@/lib/utils"

export function ProjectForm({ className }) {
  return (
    <form className={cn("space-y-4", className)}>
      <div className="space-y-2">
        <Label htmlFor="name">Project Name</Label>
        <Input
          id="name"
          placeholder="Enter project name"
          className={cn("w-full")}
        />
      </div>
      <div className="space-y-2">
        <Label htmlFor="client">Client</Label>
        <Select>
          <SelectTrigger className={cn("w-full")}>
            <SelectValue placeholder="Select a client" />
          </SelectTrigger>
          <SelectContent>
            <SelectItem value="client1">Client 1</SelectItem>
            <SelectItem value="client2">Client 2</SelectItem>
          </SelectContent>
        </Select>
      </div>
      <div className="space-y-2">
        <Label htmlFor="description">Description</Label>
        <Textarea
          id="description"
          placeholder="Enter project description"
          className={cn("min-h-[100px]")}
        />
      </div>
    </form>
  )
}

// Example 3: Card with Badge and Avatar
import { Card, CardContent, CardDescription, CardHeader, CardTitle } from
"@/components/ui/card"
import { Badge } from "@/components/ui/badge"
import { Avatar, AvatarFallback, AvatarImage } from
"@/components/ui/avatar"
import { cn } from "@/lib/utils"

export function ProjectCard({ project, className }) {
  return (
    <Card className={cn("hover:shadow-lg transition-shadow", className)}>
      <CardHeader>
        <div className="flex justify-between items-start">
          <div>
            <CardTitle>{project.name}</CardTitle>
            <CardDescription>{project.clientName}</CardDescription>
          </div>
          <Badge
            className={cn(
              "capitalize",
              project.status === "active" && "bg-green-500",
              project.status === "completed" && "bg-blue-500",
              project.status === "on-hold" && "bg-yellow-500"
            )}
          >
            {project.status}
          </Badge>
        </div>
      </CardHeader>
      <CardContent>
        <div className="flex -space-x-2">
          {project.teamMembers.map((member) => (
            <Avatar key={member.id} className={cn("border-2 border-
background")}>
              <AvatarImage src={member.image} />
              <AvatarFallback>{member.name[0]}</AvatarFallback>
            </Avatar>
          ))}
        </div>
      </CardContent>
    </Card>
  )
}

// Example 4: Dialog with Form
import { Dialog, DialogContent, DialogDescription, DialogHeader,
DialogTitle, DialogTrigger } from "@/components/ui/dialog"
import { cn } from "@/lib/utils"

export function AddClientDialog({ onSubmit, className }) {
  return (
    <Dialog>
      <DialogTrigger asChild>
        <Button className={cn(className)}>Add Client</Button>
      </DialogTrigger>
      <DialogContent className={cn("sm:max-w-[425px]")}>
        <DialogHeader>
          <DialogTitle>Add New Client</DialogTitle>
          <DialogDescription>
            Enter the client details below.
          </DialogDescription>
        </DialogHeader>
        <form onSubmit={onSubmit} className="space-y-4">
          <div className="space-y-2">
            <Label htmlFor="clientName">Client Name</Label>
            <Input id="clientName" placeholder="Acme Corp" />
          </div>
          <div className="space-y-2">
            <Label htmlFor="email">Email</Label>
            <Input id="email" type="email" placeholder="contact@acme.com"
/>
          </div>
          <Button type="submit" className="w-full">Create Client</Button>
        </form>
      </DialogContent>
    </Dialog>
  )
}

// Example 5: Table with Actions
import { Table, TableBody, TableCell, TableHead, TableHeader, TableRow }
from "@/components/ui/table"
import { DropdownMenu, DropdownMenuContent, DropdownMenuItem,
DropdownMenuTrigger } from "@/components/ui/dropdown-menu"
import { MoreHorizontal, Edit, Trash } from "lucide-react"
import { cn } from "@/lib/utils"

export function UsersTable({ users, onEdit, onDelete, className }) {
  return (
    <Table className={cn(className)}>
      <TableHeader>
        <TableRow>
          <TableHead>Name</TableHead>
          <TableHead>Email</TableHead>
          <TableHead>Roles</TableHead>
          <TableHead className="text-right">Actions</TableHead>
        </TableRow>
      </TableHeader>
      <TableBody>
        {users.map((user) => (
          <TableRow key={user.id}>
            <TableCell className={cn("font-
medium")}>{user.name}</TableCell>
            <TableCell>{user.email}</TableCell>
            <TableCell>
              <div className="flex gap-1">
                {user.roles.map((role) => (
                  <Badge key={role} variant="secondary">
                    {role}
                  </Badge>
                ))}
              </div>
            </TableCell>
            <TableCell className="text-right">
              <DropdownMenu>
                <DropdownMenuTrigger asChild>
                  <Button variant="ghost" size="sm">
                    <MoreHorizontal className="h-4 w-4" />
                  </Button>
                </DropdownMenuTrigger>
                <DropdownMenuContent align="end">
                  <DropdownMenuItem onClick={() => onEdit(user)}>
                    <Edit className="mr-2 h-4 w-4" />
                    Edit
                  </DropdownMenuItem>
                  <DropdownMenuItem
                    onClick={() => onDelete(user)}
                    className="text-red-600"
                  >
                    <Trash className="mr-2 h-4 w-4" />
                    Delete
                  </DropdownMenuItem>
                </DropdownMenuContent>
              </DropdownMenu>
            </TableCell>
          </TableRow>
        ))}
      </TableBody>
    </Table>
  )
}
```

Phase 2 Preview (Future Features)

The following features are planned for Phase 2 but NOT included in Phase 1:

- **EOD Compliance Tracking**: Monitor EOD submission patterns, identify missed submissions, generate compliance reports
- **Custom Task Statuses**: Allow per-project task status customization beyond default states
- **Deadline Management**: Automated deadline tracking, rollover logic, deadline notifications
- **Analytics & Reporting**: Project progress dashboards, team productivity reports, resource utilization charts
- **Full Task Management System**: Developers create and manage tasks, task deadlines and status tracking, task assignment and collaboration

---

Conclusion

This comprehensive guide provides everything needed to implement Phase 1 of ProjectHub. Key points to remember:

✅ **Database Schema**: Based on Better Auth core tables + custom ProjectHub tables with junction tables
✅ **Role Management**: Simple string-based roles with many-to-many user-role relationships
✅ **Frontend Route Protection**: All authorization handled on frontend based on user roles
✅ **Junction Tables**: `user_roles`, `user_project_assignments`, `user_task_assignments`
✅ **Better Auth integration**: Use for authentication, session management, and email verification
✅ **shadcn/ui with cn()**: All components use shadcn/ui and the cn() utility for styling
✅ **Reusable Components**: Every component is designed to be reusable and composable
✅ **Quality standards**: Follow TypeScript strict typing, Next.js 15+ patterns, and pagination for tables
✅ **Real-time chat**: Socket.io with project-scoped rooms
✅ **File management**: Cloudinary with 5-10 MB limits
✅ **Daily restrictions**: Memos (140 chars, one per day, same-day edit only) and EODs (one per day)
✅ **Email notifications**: React Email + Nodemailer for project assignments and critical updates

Follow these specifications strictly, and Phase 1 will be complete when all success criteria are met.

For any clarifications or issues, refer back to this document or consult with project stakeholders.