TreeHub Architecture Documentation

This document provides a comprehensive overview of TreeHub's system architecture, design patterns, and technical decisions.

🏗️ System Overview

TreeHub is built as a modern, scalable web application using a full-stack TypeScript approach with Next.js at its core. The architecture follows industry best practices for SaaS applications with a focus on mobile-first design and offline capabilities.

High-Level Architecture

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Client Apps   │    │   Web Browser   │    │   Mobile PWA    │
│                 │    │                 │    │                 │
└─────────────────┘    └─────────────────┘    └─────────────────┘
         │                      │                      │
         └──────────────────────┼──────────────────────┘
                                │
                      ┌─────────────────┐
                      │   Load Balancer │
                      │    (Vercel)     │
                      └─────────────────┘
                                │
                      ┌─────────────────┐
                      │   Next.js App   │
                      │   (App Router)  │
                      └─────────────────┘
                                │
         ┌──────────────────────┼──────────────────────┐
         │                      │                      │
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Database      │    │   Storage       │    │   External      │
│  PostgreSQL     │    │   AWS S3        │    │    APIs         │
└─────────────────┘    └─────────────────┘    └─────────────────┘
```

🎯 Design Principles

1. Mobile-First Architecture

- **Progressive Web App (PWA)**: Installable, offline-capable
- **Responsive Design**: Optimized for tablets and smartphones
- **Touch-Friendly UI**: Large touch targets, gesture support
- **Offline-First**: Core features work without internet

2. Performance-Oriented

- **Server-Side Rendering (SSR)**: Fast initial page loads
- **Static Site Generation (SSG)**: Pre-rendered marketing pages
- **Code Splitting**: Reduced bundle sizes
- **Image Optimization**: Next.js Image component with WebP support

3. Scalability

- **Serverless Architecture**: Auto-scaling with Vercel Functions
- **Database Optimization**: Connection pooling, query optimization
- **CDN Distribution**: Global edge network for static assets
- **Caching Strategy**: Multi-layer caching (browser, CDN, database)

4. Security-First

- **Authentication**: NextAuth.js with multiple providers
- **Authorization**: Role-based access control (RBAC)
- **Data Protection**: Encryption at rest and in transit
- **Input Validation**: Zod schemas for type-safe validation

🏛️ Frontend Architecture

Technology Stack

- **Framework**: Next.js 14 with App Router
- **Language**: TypeScript for type safety
- **Styling**: Tailwind CSS with custom design system
- **Components**: Radix UI primitives with custom styling
- **State Management**: Zustand for global state, React Query for server state
- **Forms**: React Hook Form with Zod validation
- **Animation**: Framer Motion for smooth transitions

Component Architecture

```
src/
├── app/                    # Next.js App Router
│   ├── (auth)/            # Authentication routes
│   ├── (dashboard)/       # Protected dashboard routes
│   ├── api/               # API routes
│   ├── globals.css        # Global styles
│   ├── layout.tsx         # Root layout
│   └── page.tsx           # Home page
├── components/            # Reusable components
│   ├── ui/                # Base UI components
│   │   ├── button.tsx
│   │   ├── input.tsx
│   │   └── ...
│   ├── forms/             # Form components
│   ├── charts/            # Data visualization
│   └── layout/            # Layout components
├── hooks/                 # Custom React hooks
├── lib/                   # Utility functions
├── types/                 # TypeScript definitions
└── stores/                # Zustand stores
```

State Management Strategy

1. Server State (React Query)

- API data fetching and caching
- Background updates and synchronization

- Optimistic updates for better UX

```
// Example: Client data fetching
const { data: clients, isLoading } = useQuery({
  queryKey: ['clients'],
  queryFn: () => api.clients.list(),
  staleTime: 5 * 60 * 1000, // 5 minutes
})
```

2. Global Client State (Zustand)

- User preferences and settings
- UI state (sidebar, modals, notifications)
- Offline data synchronization

```
// Example: User store
interface UserStore {
  user: User | null
  preferences: UserPreferences
  setUser: (user: User) => void
  updatePreferences: (prefs: Partial<UserPreferences>) => void
}

const useUserStore = create<UserStore>((set) => ({
  user: null,
  preferences: defaultPreferences,
  setUser: (user) => set({ user }),
  updatePreferences: (prefs) =>
    set((state) => ({
      preferences: { ...state.preferences, ...prefs }
    })),
}))
```

3. Local Component State (useState/useReducer)

- Form state and validation
- Component-specific UI state
- Temporary data before submission

🔧 Backend Architecture

API Design

TreeHub follows RESTful API conventions with Next.js API routes:

```
/api/
├── auth/                   # Authentication endpoints
│   ├── [...nextauth].ts    # NextAuth.js handler
│   └── signup.ts           # User registration
├── clients/                # Client management
│   ├── index.ts            # GET /api/clients, POST /api/clients
│   └── [id]/               # GET/PUT/DELETE /api/clients/[id]
├── properties/             # Property management
├── jobs/                   # Job scheduling and management
├── invoices/               # Invoice generation
└── upload/                 # File upload handling
```

Database Architecture

Schema Design

TreeHub uses PostgreSQL with Prisma ORM for type-safe database access:

```
// Core entities
model User {
  id        String   @id @default(cuid())
  email     String   @unique
  name      String?
  role      Role     @default(USER)
  company   String?
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  // Relations
  jobs      Job[]
  clients   Client[]
}

model Client {
  id         String     @id @default(cuid())
  name       String
  email      String?
  phone      String?
  address    String?
  createdAt  DateTime   @default(now())
  updatedAt  DateTime   @updatedAt

  // Relations
  properties Property[]
  jobs       Job[]
  invoices   Invoice[]
  userId     String
  user       User       @relation(fields: [userId], references: [id])
}

model Property {
  id          String    @id @default(cuid())
  name        String
  address     String
  coordinates Json?     // {lat: number, lng: number}
  notes       String?
  createdAt   DateTime  @default(now())
  updatedAt   DateTime  @updatedAt

  // Relations
  trees       Tree[]
  jobs        Job[]
  clientId    String
  client      Client    @relation(fields: [clientId], references: [id])
}

model Tree {
  id          String      @id @default(cuid())
  species     String
  diameter    Float?      // DBH in inches
  height      Float?      // Height in feet
  condition   Condition   @default(GOOD)
  coordinates Json?       // GPS coordinates
  notes       String?
  photos      String[]    // Array of photo URLs
  createdAt   DateTime    @default(now())
  updatedAt   DateTime    @updatedAt

  // Relations
  propertyId  String
```

```
  property    Property    @relation(fields: [propertyId], references: [id])
  jobTrees    JobTree[]   // Many-to-many with jobs
}

model Job {
  id                String      @id @default(cuid())
  title             String
  description       String?
  status            JobStatus   @default(SCHEDULED)
  priority          Priority    @default(MEDIUM)
  scheduledDate     DateTime?
  completedDate     DateTime?
  estimatedDuration Int?        // Minutes
  actualDuration    Int?        // Minutes
  notes             String?
  photos            String[]    // Before/after photos
  createdAt         DateTime    @default(now())
  updatedAt         DateTime    @updatedAt

  // Relations
  clientId          String
  client            Client      @relation(fields: [clientId], references: [id])
  propertyId        String?
  property          Property?   @relation(fields: [propertyId], references: [id])
  assignedUserId    String?
  assignedUser      User?       @relation(fields: [assignedUserId], references: [id])
  trees             JobTree[]   // Many-to-many with trees
  invoice           Invoice?
}
```

Database Optimization

- **Indexes**: Strategic indexing on frequently queried fields
- **Connection Pooling**: Efficient database connection management
- **Query Optimization**: Prisma query optimization and N+1 prevention
- **Migrations**: Version-controlled schema changes

Authentication & Authorization

NextAuth.js Configuration

```
export const authOptions: NextAuthOptions = {
  adapter: PrismaAdapter(prisma),
  providers: [
    CredentialsProvider({
      name: "credentials",
      credentials: {
        email: { label: "Email", type: "email" },
        password: { label: "Password", type: "password" }
      },
      async authorize(credentials) {
        // Custom authentication logic
        const user = await authenticateUser(credentials)
        return user
      }
    }),
    GoogleProvider({
      clientId: process.env.GOOGLE_CLIENT_ID!,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET!,
    }),
  ],
  session: {
    strategy: "jwt",
    maxAge: 30 * 24 * 60 * 60, // 30 days
  },
  callbacks: {
    async jwt({ token, user }) {
      if (user) {
        token.role = user.role
        token.company = user.company
      }
      return token
    },
    async session({ session, token }) {
      session.user.role = token.role
      session.user.company = token.company
      return session
    },
  },
}
```

Role-Based Access Control (RBAC)

```
enum Role {
  ADMIN     // Full system access
  MANAGER   // Team and client management
  USER      // Basic field operations
  CLIENT    // Read-only client portal access
}

// Middleware for API route protection
export function withAuth(handler: NextApiHandler, requiredRole?: Role) {
  return async (req: NextApiRequest, res: NextApiResponse) => {
    const session = await getServerSession(req, res, authOptions)

    if (!session) {
      return res.status(401).json({ error: 'Unauthorized' })
    }

    if (requiredRole && !hasRole(session.user.role, requiredRole)) {
      return res.status(403).json({ error: 'Forbidden' })
    }

    return handler(req, res)
  }
}
```

📱 Mobile Architecture

Progressive Web App (PWA)

TreeHub is designed as a PWA to provide native app-like experience:

```
// next.config.js PWA configuration
const withPWA = require('next-pwa')({
  dest: 'public',
  register: true,
  skipWaiting: true,
  runtimeCaching: [
    {
      urlPattern: /^https:\/\/api\.treehubusa\.com\/.*$/,
      handler: 'NetworkFirst',
      options: {
        cacheName: 'api-cache',
        expiration: {
          maxEntries: 100,
          maxAgeSeconds: 24 * 60 * 60, // 24 hours
        },
      },
    },
  ],
})
```

Offline Capabilities

- **Service Worker**: Caches critical resources and API responses
- **Local Storage**: Stores user preferences and temporary data
- **IndexedDB**: Offline data synchronization queue
- **Background Sync**: Syncs data when connection is restored

```
// Offline data synchronization
class OfflineSync {
  private queue: SyncItem[] = []

  async addToQueue(action: string, data: any) {
    const item: SyncItem = {
      id: generateId(),
      action,
      data,
      timestamp: Date.now(),
      retries: 0,
    }

    this.queue.push(item)
    await this.saveQueue()

    if (navigator.onLine) {
      this.processQueue()
    }
  }

  async processQueue() {
    while (this.queue.length > 0) {
      const item = this.queue[0]

      try {
        await this.syncItem(item)
        this.queue.shift()
      } catch (error) {
        item.retries++
        if (item.retries >= MAX_RETRIES) {
          this.queue.shift() // Remove failed item
        }
        break // Stop processing on error
      }
    }

    await this.saveQueue()
  }
}
```

🔄 Data Flow Architecture

Request/Response Flow

```
1. User Action (UI Component)
   ↓
2. Event Handler (React)
   ↓
3. API Call (React Query)
   ↓
4. Next.js API Route
   ↓
5. Business Logic Layer
   ↓
6. Database Query (Prisma)
   ↓
7. Response Processing
   ↓
8. Cache Update (React Query)
   ↓
9. UI Re-render (React)
```

Real-time Updates

For real-time features like job status updates:

```
// WebSocket connection for real-time updates
const useRealTimeUpdates = () => {
  const queryClient = useQueryClient()

  useEffect(() => {
    const ws = new WebSocket(process.env.NEXT_PUBLIC_WS_URL!)

    ws.onmessage = (event) => {
      const { type, data } = JSON.parse(event.data)

      switch (type) {
        case 'JOB_STATUS_UPDATE':
          queryClient.invalidateQueries(['jobs', data.jobId])
          break
        case 'NEW_MESSAGE':
          queryClient.invalidateQueries(['messages'])
          break
      }
    }

    return () => ws.close()
  }, [queryClient])
}
```

🚀 Deployment Architecture

Vercel Platform

- **Edge Network**: Global CDN for fast content delivery
- **Serverless Functions**: Auto-scaling API endpoints
- **Preview Deployments**: Branch-based preview environments

- **Analytics**: Built-in performance monitoring

Environment Configuration

```
// Environment-specific configurations
const config = {
  development: {
    apiUrl: 'http://localhost:3000/api',
    dbUrl: process.env.DATABASE_URL,
    logLevel: 'debug',
  },
  staging: {
    apiUrl: 'https://staging.treehubusa.com/api',
    dbUrl: process.env.DATABASE_URL,
    logLevel: 'info',
  },
  production: {
    apiUrl: 'https://treehubusa.com/api',
    dbUrl: process.env.DATABASE_URL,
    logLevel: 'error',
  },
}
```

📊 Monitoring & Observability

Error Tracking

```
// Sentry configuration for error monitoring
import * as Sentry from '@sentry/nextjs'

Sentry.init({
  dsn: process.env.SENTRY_DSN,
  environment: process.env.NODE_ENV,
  tracesSampleRate: 1.0,
  beforeSend(event) {
    // Filter sensitive data
    if (event.user) {
      delete event.user.email
    }
    return event
  },
})
```

Performance Monitoring

- **Core Web Vitals**: LCP, FID, CLS tracking
- **API Response Times**: Database query performance
- **Bundle Analysis**: JavaScript bundle size optimization
- **User Analytics**: Feature usage and user behavior

🔒 Security Architecture

Data Protection

- **Encryption**: AES-256 for sensitive data at rest
- **HTTPS**: TLS 1.3 for data in transit

- **Input Validation**: Zod schemas for all inputs
- **SQL Injection Prevention**: Prisma ORM parameterized queries

Authentication Security

- **Password Hashing**: bcrypt with salt rounds
- **Session Management**: Secure HTTP-only cookies
- **CSRF Protection**: Built-in Next.js CSRF tokens
- **Rate Limiting**: API endpoint protection

```
// Rate limiting middleware
import { Ratelimit } from '@upstash/ratelimit'
import { Redis } from '@upstash/redis'

const ratelimit = new Ratelimit({
  redis: Redis.fromEnv(),
  limiter: Ratelimit.slidingWindow(10, '10 s'),
})

export async function rateLimitMiddleware(req: NextApiRequest) {
  const identifier = getClientIP(req)
  const { success } = await ratelimit.limit(identifier)

  if (!success) {
    throw new Error('Rate limit exceeded')
  }
}
```

🔧 Development Architecture

Code Quality

- **TypeScript**: Strict type checking
- **ESLint**: Code linting and style enforcement
- **Prettier**: Code formatting
- **Husky**: Git hooks for pre-commit checks

Testing Strategy

```
// Testing pyramid
├── Unit Tests (Jest + React Testing Library)
│   ├── Components
│   ├── Hooks
│   ├── Utilities
│   └── API Functions
├── Integration Tests
│   ├── API Routes
│   ├── Database Operations
│   └── Authentication Flows
└── E2E Tests (Playwright)
    ├── Critical User Journeys
    ├── Mobile Workflows
    └── Cross-browser Testing
```

CI/CD Pipeline

```yaml
# GitHub Actions workflow
name: CI/CD
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
      - run: npm ci
      - run: npm run type-check
      - run: npm run lint
      - run: npm test
      - run: npm run build

  deploy:
    needs: test
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: amondnet/vercel-action@v25
        with:
          vercel-token: ${{ secrets.VERCEL_TOKEN }}
          vercel-args: '--prod'
```

📈 Scalability Considerations

Performance Optimization

- **Code Splitting**: Route-based and component-based splitting
- **Image Optimization**: WebP format, responsive images
- **Caching Strategy**: Multi-layer caching (browser, CDN, API)
- **Database Optimization**: Query optimization, connection pooling

Horizontal Scaling

- **Stateless Design**: No server-side session storage
- **Database Scaling**: Read replicas, connection pooling
- **CDN Distribution**: Global edge network
- **Microservices Ready**: Modular API design for future splitting

This architecture provides a solid foundation for TreeHub's current needs while maintaining flexibility for future growth and feature additions.