Developer Onboarding Guide

Generated from Markdown source on 11/5/2025, 2:02:31 PM

---

pTreks Developer Onboarding Guide

Version

Version: 1.1.0
Created: November 5, 2025
Modified: December 18, 2025
Status: Active Development Guide

Executive Summary

This comprehensive developer onboarding guide provides everything a new developer needs to understand, work with, and build upon the pTreks platform. pTreks is a sophisticated outdoor activity and trekking management system built on Node.js and PostgreSQL, featuring real-time GPS tracking, group coordination, messaging, file management, subscription-based pricing, and comprehensive admin capabilities. This document covers system architecture, client applications (user-facing and admin), API structure, database design, development workflows, and future roadmap.

Who This Guide Is For:

- New backend developers joining the team
- Frontend developers building client applications
- Full-stack developers working across the system
- System architects evaluating the platform
- DevOps engineers deploying and maintaining the system

Table of Contents

System Overview

What is pTreks?

pTreks is a comprehensive outdoor activity management platform designed for groups to coordinate treks, track locations in real-time, communicate during adventures, and manage outdoor activities. The system serves two primary client types:

1. **End-User Mobile/Web Application**: For outdoor enthusiasts to create and join groups, organize treks, share GPS locations, communicate, and upload photos/videos
2. **Admin Dashboard Application**: For administrators to manage users, monitor system health, review analytics, handle moderation, and configure system settings

Key Value Propositions

- **Group Coordination**: Organize and manage outdoor activities with groups of any size
- **Real-Time Safety**: GPS location tracking during treks for safety and coordination
- **Communication**: Built-in messaging and commenting for group and trek discussions
- **File Management**: Secure photo/video uploads with automatic thumbnail generation
- **Subscription Management**: Tiered pricing (Free, Pro, Team, Enterprise) with Stripe integration
- **Comprehensive Analytics**: Engagement scoring, activity tracking, and system-wide analytics
- **Moderation Tools**: Content flagging, user management, and automated moderation workflows

System Scale

- **Database**: 54 tables, 32 functions, 45 triggers, 270 indexes
- **API Endpoints**: 493+ endpoints across 56 categories
- **Active Users**: Growing user base with comprehensive activity tracking
- **Data Volume**: ~3.5M rows (99.7% in application logs for audit trail)
- **Database Size**: 1.9 GB

Technology Stack

Backend Core

- **Runtime**: Node.js (latest LTS)
- **Framework**: Express.js for RESTful API
- **Database**: PostgreSQL 14+ with PostGIS extension for geospatial data
- **Authentication**: JWT (JSON Web Tokens) with bcrypt password hashing
- **Process Management**: PM2 for production deployment and background processes

Key Dependencies

- **Database**: `pg` (node-postgres) for PostgreSQL connectivity
- **Validation**: Joi for request validation and schema definition
- **Security**: Helmet.js for HTTP security headers, CORS middleware
- **File Processing**: Sharp for image processing and thumbnail generation
- **Email**: Nodemailer with HTML template engine
- **Real-Time**: WebSocket support for live features
- **Payment**: Stripe SDK for subscription management
- **Logging**: Winston for structured logging

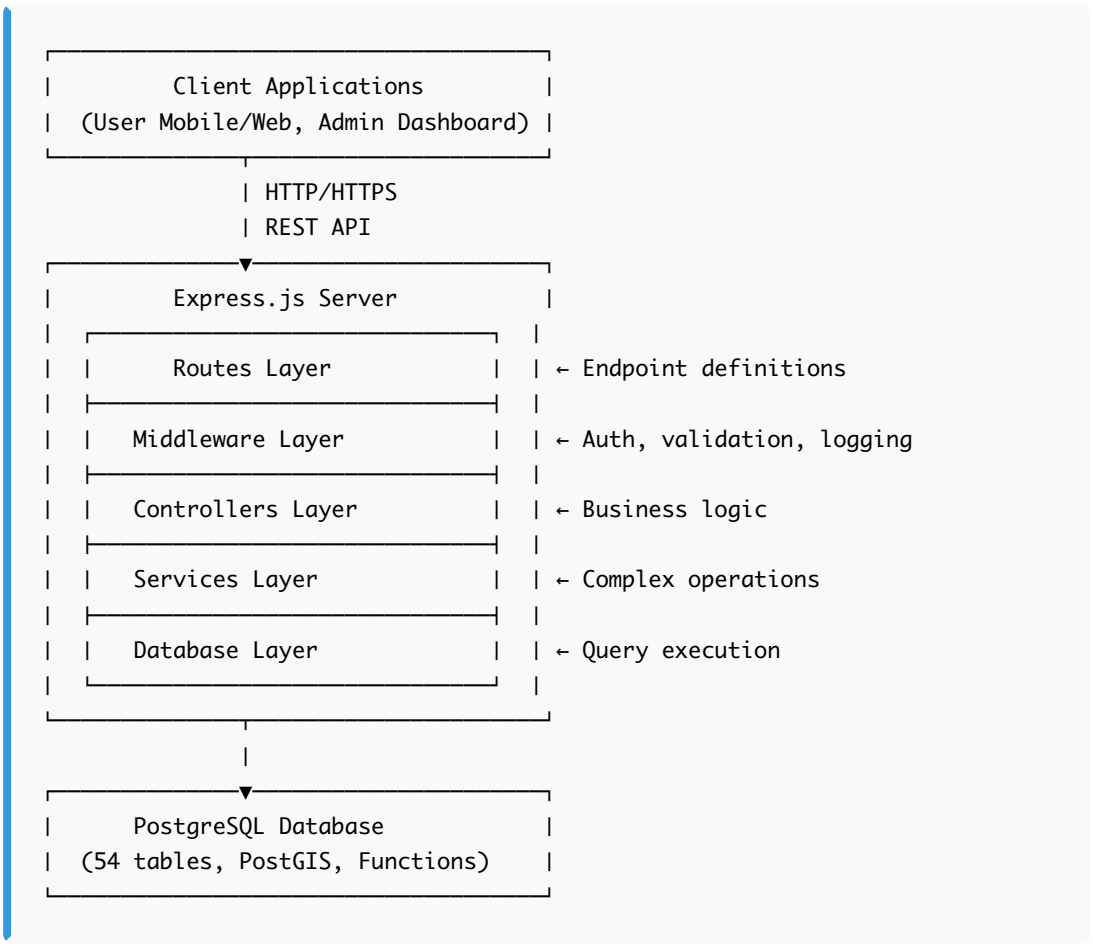
Development Tools

- **Package Manager**: npm
- **Code Quality**: ESLint (if configured)
- **Version Control**: Git
- **Process Management**: PM2 for development and production
- **Database Tools**: psql for direct database access

System Architecture

Layered Architecture

pTreks follows a strict layered architecture pattern:



Request Flow

1. **Client Request** → HTTP request to Express server
2. **Route Matching** → Express routes match URL pattern
3. **Middleware Pipeline** → Authentication, validation, rate limiting, logging
4. **Controller** → Business logic execution
5. **Service Layer** (if needed) → Complex operations (subscriptions, email, etc.)
6. **Database Query** → SQL execution via node-postgres
7. **Response** → JSON response back to client

Background Processes

The system runs 13+ background processes managed by PM2:

- **Main API Server**: `ptreks-api` (port 3000)
- **Email Queue Processor**: Processes queued emails
- **Subscription Maintenance Processor**: Handles subscription lifecycle (expirations, renewals, grace periods)
- **Account Cleanup Processor**: Removes terminated accounts after grace period
- **Activity Logs Cleanup Processor**: Prunes old logs, creates summaries
- **Trek Auto-Finish Processor**: Automatically finishes old treks
- **Scheduled Deletion Processor**: Permanently deletes flagged content
- **File Integrity Processor**: Validates file system integrity
- **Session Cleanup Processor**: Removes expired sessions
- **Analytics Snapshot Processor**: Creates analytics snapshots
- **Engagement Score Processor**: Calculates daily engagement scores
- **Abandoned Upgrade Cleanup Processor**: Cleans up abandoned payment flows

All processes are configured via `ecosystem.config.js` with PM2 cron scheduling.

Client Applications

End-User Client Application

Purpose: Mobile/web application for outdoor enthusiasts to use pTreks

Key Capabilities (based on public API endpoints):

Authentication & User Management

- User registration with email verification
- Sign-in/sign-out with JWT tokens
- Password reset and change
- User profile management (name, bio, location, preferences)
- Profile photo upload and management
- Account deactivation/termination

Groups Management

- Create and join groups
- Browse available groups (public and private)
- Search groups by name, type, location
- View group details, members, and treks
- Manage group membership (join, leave, favorite)

- Group messaging and comments
- Group settings and preferences

Treks Management

- Create treks with GPS routes
- Browse upcoming and past treks
- Join/leave treks
- Real-time GPS location sharing during treks
- View trek participants and their locations
- Trek messaging and comments
- Upload photos/videos during treks (moments)
- Mark treks as started/finished
- View trek statistics and history

Communication

- Group messaging (real-time or async)
- Trek messaging (during active treks)
- Comments on groups, treks, and moments
- Notifications for mentions, messages, and updates
- Real-time WebSocket connections for live updates

File Management

- Upload photos/videos to groups, treks, and moments
- Automatic thumbnail generation
- View and manage uploaded assets
- Delete own uploads

Subscription Management

- View available subscription plans (Free, Pro, Team, Enterprise)
- View current subscription status
- Upgrade/downgrade subscription plans
- Change billing cycle (monthly/yearly)
- View billing history and invoices
- Manage payment methods (via Stripe)

Personal Features

- View personal statistics (groups, treks, activities)
- View engagement score and trends
- View activity history
- Manage notification preferences
- View testimonials and ratings

API Endpoint Pattern: Most user-facing endpoints are under `/api/v1/` without `/admin/` prefix

Authentication: Bearer token (JWT) required for most endpoints

---

Admin Dashboard Application

Purpose: Comprehensive web-based administrative interface for system management, monitoring, and configuration

The pTreks Admin Dashboard is a powerful, enterprise-grade React application designed for administrators, system administrators, and developers to manage every aspect of the pTreks platform. Built with modern React patterns and a focus on data visualization, real-time monitoring, and efficient workflow management, the admin dashboard provides complete control over the system while maintaining an intuitive and productive user experience.

Key Capabilities (based on `/admin/` API endpoints):

User Management

- View all users with pagination, search, and filtering
- View comprehensive user details (profile, roles, subscriptions, activity, engagement scores)
- Edit user profiles and roles
- Activate/deactivate user accounts
- Terminate user accounts with secure confirmation workflow
- View user engagement scores and analytics
- Manage user sessions (view active sessions, terminate sessions)
- View detailed user activity logs
- Filter users by account status (active, terminated, deactivated)
- View user's groups and treks from user detail page

Content Moderation

- View flagged content (groups, treks, comments, messages)
- Review flag history and reasons
- Approve/reject flags
- Hide/restore content
- Delete content permanently (with scheduled deletion support)
- View moderation queue and statistics
- Content flag workflow management

Analytics & Reporting

- System-wide engagement statistics with interactive charts
- User engagement leaderboards
- Engagement score distribution analysis
- Engagement trends over time (historical analytics)
- User segmentation (high/medium/low engagement)
- Dashboard with comprehensive analytics widgets
- Alert users below engagement threshold
- Activity logs and audit trails
- Real-time system monitoring with live metrics
- Advanced analytics with customizable date ranges
- Entity statistics (groups, treks, users, content)
- Content statistics (messages, comments, assets, moments)
- Process monitoring with execution history

Subscription Management

- View all subscriptions with pagination and filtering
- View subscription details and history
- Monitor subscription lifecycle (active, grace period, suspended, etc.)
- View billing transactions with comprehensive filtering
- Manage subscription plans (create, edit, delete)
- Configure plan features (title, description, display order, basic/advanced)
- Live preview of subscription plans as end-users see them
- Plan comparison table for side-by-side feature comparison
- View billing configuration (enabled status, currency, payment providers, etc.)
- Feature flag management (yearly billing, pricing enabled)
- Invoice management with PDF viewing and download
- Subscription lifecycle tracking and management

System Configuration

- Manage system parameters (rate limits, email limits, cooldown periods, etc.)
- Configure email rate limiting per tier (free, paid, admin)
- View and update email queue status
- Process email queue manually
- View email archives with search
- Manage background processes (restart, monitor, view logs)
- View system health and status
- System parameter editing with validation
- Email system configuration and monitoring

Email Management

- View email queue status and statistics
- Preview queued emails with full content
- Retry failed emails
- Process email queue manually
- View email archives with pagination
- Search email history
- View recipient email history
- Email rate limiting configuration per user tier
- Admin invitation management
- Email system monitoring and health checks

Background Process Management

- View status of all 13+ background processes
- Real-time process health monitoring
- Restart background processes manually
- Monitor process execution logs and activity
- View process execution history and timeline
- Process configuration display (dynamic, server-driven)
- Manual trigger capabilities for certain processes
- Process log viewing with filtering

Security & Audit

- View security audit logs with filtering

- Monitor password reset tokens
- View session management logs
- Review access patterns
- Account lifecycle management (activate, deactivate, terminate)
- Role management and assignment
- Security compliance reporting
- Audit trail navigation

API Documentation & Reference

- Interactive API reference browser
- Categorized endpoint documentation (56+ categories)
- Search and filter endpoints by method, category, keywords
- View API documentation as Markdown
- Download API documentation as PDF or JSON (OpenAPI)
- Integration with Swagger documentation
- Dynamic endpoint keyword generation from path segments

Documentation Management

- Server documentation browser with categorized organization
- 15+ document categories (Implementation Plans, API Documentation, Database, etc.)
- Collapsible category sections with document summaries
- View documents as Markdown with live rendering
- Download documents as Markdown, HTML, or PDF
- Document metadata display (category, summary, creation date)

Statistics & Monitoring

- Real-time server monitoring with live metrics
- Server log statistics and analysis
- User activity monitoring with tier-based filtering
- Rate limiting rule management (create, edit, delete, view violations)
- System monitoring dashboard
- Advanced analytics with chart visualizations (Recharts)
- Historical analytics with date range selection
- Process monitoring with execution history

Content Management

- Bug reports management with reporter linking
- Testimonials management with approval workflow
- FAQ management with categories
- View and manage all content types (groups, treks, comments, messages, assets, moments)
- Content detail pages with comprehensive information
- Status management for groups and treks

API Endpoint Pattern: All admin endpoints are under `/api/v1/admin/`

Authentication: Bearer token (JWT) required with Admin+ role (admin, sysadmin, root)

Role Requirements: Most admin endpoints require `admin` role minimum, some require `sysadmin` or `root`

Admin Dashboard React Application

Overview

The pTreks Admin Dashboard is a sophisticated, production-ready React.js web application built specifically for platform administrators, system operators, and developers. Unlike the end-user client which focuses on outdoor activity management, the admin dashboard provides comprehensive system administration, monitoring, analytics, and configuration capabilities. The application is designed with a focus on data density, real-time monitoring, efficient workflows, and powerful administrative tools.

Technology Stack

Core Framework:

- **React 18.2.0**: Modern React with hooks, concurrent features, and improved performance
- **React Router DOM 6.3.0**: Client-side routing with nested routes and navigation
- **React Scripts 5.0.1**: Create React App configuration for development and build

State Management:

- **React Context API**: Minimal global state management (authentication)
- **Local Storage**: Token persistence and user preference storage
- **Component State**: Extensive use of `useState` and `useEffect` for component-level state management
- **Service Layer**: Centralized API communication via service modules

UI Libraries & Tools:

- **Recharts 3.1.2**: Comprehensive charting library for data visualization (line charts, bar charts, pie charts, etc.)
- **Axios 1.4.0**: HTTP client for API communication with interceptors
- **JWT Decode 3.1.2**: JWT token parsing for role-based access control
- **HTML-to-Text 9.0.5**: Email content rendering and formatting

Development Tools:

- **Port 4100**: Development server port (configurable via `PORT` environment variable)
- **ESLint**: Code quality and style enforcement
- **Web Vitals**: Performance monitoring

Application Architecture

Component Structure:

```
src/
├── components/
|   ├── signedin/          # All authenticated admin pages
|   |   ├── api/           # API documentation and reference
|   |   ├── assets/        # Asset management
|   |   ├── billing/       # Invoice management
|   |   ├── bugreports/    # Bug report management
|   |   ├── comments/      # Comment moderation
|   |   ├── dashboard/     # Main dashboard
|   |   ├── documentation/ # Server documentation viewer
|   |   ├── email/         # Email queue and archives
```

```
|   |   ├── engagement/    # Engagement score management
|   |   ├── faqs/          # FAQ management
|   |   ├── groups/        # Group management
|   |   ├── messages/      # Message moderation
|   |   ├── moments/       # Moment management
|   |   ├── security/      # Security, audit, lifecycle management
|   |   ├── server/        # Server configuration and monitoring
|   |   ├── statistics/    # Analytics and statistics
|   |   ├── subscription/  # Subscription and billing management
|   |   ├── testimonials/  # Testimonial management
|   |   ├── treks/         # Trek management
|   |   └── users/         # User management
|   ├── shared/            # Reusable components
|   |   ├── AdminHeader.js
|   |   ├── AdminFooter.js
|   |   ├── AdminPageLayout.js
|   |   ├── Breadcrumbs.js
|   |   ├── CircularProgress.js
|   |   ├── PageLayout.js
|   |   ├── PageNavigation.js
|   |   └── QuickSearch.js
|   └── SignIn.js          # Authentication page
├── services/              # API service layer
|   ├── authService.js
|   ├── invoiceService.js
|   ├── notificationsService.js
|   ├── planFeaturesService.js
|   ├── subscriptionLifecycleService.js
|   └── ...
├── constants/             # Application constants
|   ├── constants.js       # API URLs, environment config
|   ├── notificationConfig.js
|   └── pages.js           # Page metadata and keywords
└── App.js                 # Main application component
```

Routing Architecture:

- **Authentication**: Sign-in page (`/`) with protected route redirect
- **Dashboard** (`/dashboard`): Main admin dashboard with system overview and quick navigation
- **User Management** (`/users` , `/users/:userId`): User list with pagination, search, filtering, and detailed user views
- **Content Management**: Groups (`/groups`), Treks (`/treks`), Comments (`/comments`), Messages (`/messages`), Assets (`/assets`), Moments (`/moments`)
- **Subscription Management**: Plans (`/subscription-management`), Features (`/subscription-plan-features`), Lifecycle (`/subscription-lifecycle-management`), Invoices (`/invoices`)
- **Analytics & Statistics**: Statistics (`/statistics`), Engagement Scores (`/engagement-scores`), Leaderboards, Advanced Analytics
- **Security & Audit**: Security (`/security`), Account Status (`/account-status`), Lifecycle Management (`/lifecycle-management`), Session Management (`/session-management`), Audit Compliance (`/audit-compliance`), Role Management (`/role-management`)

- **Server Management**: Server (`/server`), Background Processes (`/background-processes`), System Parameters (`/system-parameters`), File Management (`/file-management`)
- **Email Management**: Email System (`/email-system`), Email Queue (`/email-queue`), Email Archives (`/email-archives`), Email Rate Limiting (`/email-rate-limiting`)
- **Documentation**: API Reference (`/api-reference`), Server Documentation (`/documentation`)
- **Content Moderation**: Bug Reports (`/bug-reports`), Testimonials (`/testimonials`), FAQs (`/faqs`)
- **Monitoring**: System Monitoring (`/system-monitoring`), Real-Time Monitor (`/server-real-time-monitor`), Process Monitor (`/process-monitor`)

Protected Routes: All routes except `/` (sign-in) require authentication and appropriate admin role. Protected routes automatically redirect to sign-in if not authenticated.

State Management

State Patterns:

- **Local State**: Component-specific state using `useState` hooks
- **Service Layer State**: API responses cached in component state
- **Context State**: Minimal global state via `authService` (authentication token and user data)
- **Derived State**: Computed values from props or API responses
- **URL State**: Search parameters for filtering, pagination, and tab navigation

Authentication Flow:

1. User signs in → JWT token stored in `localStorage`
2. Token included in `Authorization: Bearer <token>` header for all API requests
3. `authService` manages authentication state and user profile
4. JWT token decoded to extract user roles and rank for access control
5. Protected routes check `authService.isAuthenticated()` before rendering
6. Role-based UI rendering (buttons, links, sections) based on user rank
7. Token expiration handled via axios interceptor (401 redirects to sign-in)

Component Organization

Shared Components (`src/components/shared/`):

- **Layout Components**: `AdminPageLayout` (Header + Footer wrapper), `PageLayout` (alternative layout)
- **Navigation**: `AdminHeader` (navigation, user menu, role-based navigation), `AdminFooter` (footer links)
- **Navigation Helpers**: `Breadcrumbs` (breadcrumb navigation), `PageNavigation` (back/home navigation)
- **UI Elements**: `CircularProgress` (loading spinner), `QuickSearch` (dashboard quick navigation search)
- **Error Handling**: Error boundaries and error state displays

Feature-Specific Components:

- **Dashboard**: `Dashboard.js` - Main admin dashboard with system overview, quick stats, and navigation
- **Users**: `UsersList.js` , `UserDetail.js` , `UserGroupsList.js` , `UserTreksList.js`
- **Content**: `GroupsList.js` , `GroupDetail.js` , `TreksList.js` , `TrekDetail.js` , `CommentsList.js` , `MessagesList.js`

- **Subscriptions**: `SubscriptionManagement.js`, `PlanFeaturesManagement.js`, `SubscriptionLifecycleManagement.js`, `InvoiceManagement.js`
- **Analytics**: `Statistics.js`, `EngagementScores.js`, `Leaderboards.js`, `AdvancedAnalytics.js`
- **Security**: `Security.js`, `AccountStatus.js`, `LifecycleManagement.js`, `SessionManagement.js`, `AuditCompliance.js`, `RoleManagement.js`
- **Server**: `Server.js`, `BackgroundProcesses.js`, `SystemParameters.js`, `FileManagement.js`, `ServerLogs.js`
- **Email**: `EmailSystem.js`, `EmailQueue.js`, `EmailArchives.js`, `EmailRateLimiting.js`
- **Documentation**: `ApiReference.js`, `Documentation.js`, `MdDocument.js`

Naming Conventions:

- **Components**: PascalCase (e.g., `Dashboard.js`, `UserDetail.js`)
- **Files**: Match component name exactly
- **Folders**: lowercase-only (e.g., `users`, `subscription`, `security`)
- **Services**: camelCase with `Service` suffix (e.g., `authService.js`, `invoiceService.js`)

Styling Approach

CSS Architecture:

- **Component-Scoped CSS**: Inline styles using `<style jsx="true">` blocks for component-specific styling
- **Global Styles**: App-level styles in `App.css`
- **Responsive Design**: Mobile-first approach with media queries
- **Dark Theme**: Consistent black/near-black theme (`#000000`, `#0a0a0a`, `#1a1a1a`) matching pTreks ecosystem
- **Accent Colors**: Green (`#4CAF50`) for primary actions, blue for information

Style Patterns:

- **Color Scheme**: Pure black backgrounds (`#000000` or `#0a0a0a`), `#1a1a1a` content boxes, `#333` borders, `#2a2a2a` secondary backgrounds
- **Typography**: Nunito font family (consistent with end-user client), consistent font sizes
- **Spacing**: Consistent padding and margins using `rem` units
- **Responsive Breakpoints**: Mobile (480px), Tablet (768px), Desktop (1024px+)
- **Table Styling**: Comprehensive table styles with hover effects, striped rows, and responsive design
- **Card Layouts**: Responsive grid layouts for metrics, statistics, and content cards

Responsive Design:

- **Mobile**: Single column layouts, stacked elements, compact navigation, collapsible sections
- **Tablet**: 2-column grids where appropriate, optimized spacing, touch-friendly buttons
- **Desktop**: Multi-column layouts, full feature sets, expanded navigation, data-dense displays

API Integration

API Client (`src/services/`):

- **Service Modules**: Separate service files for each domain (auth, invoices, subscriptions, etc.)

- **Axios Instance**: Configured with base URL (`BACKEND_BASE_URL` from constants)
- **Request Interceptor**: Automatically adds JWT token to all requests
- **Response Interceptor**: Handles 401 errors (token expiration) with automatic redirect
- **Error Handling**: Centralized error logging and user-friendly error messages
- **Base URL**: `http://localhost:3000/api/v1` (development) or configured production URL

API Patterns:

```
// Standard API call pattern
import { authService } from '../services/authService';
import { BACKEND_BASE_URL } from '../constants';

const fetchData = async () => {
  try {
    const token = authService.getToken();
    const response = await fetch(`${BACKEND_BASE_URL}/admin/endpoint`, {
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json'
      }
    });
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('API Error:', error);
    // Handle error (toast notification, etc.)
  }
};
```

Service Layer (`src/services/`):

- `authService.js` : Authentication, token management, user data retrieval, role checking
- `invoiceService.js` : Invoice management (list, details, PDF download/regeneration, summary, export)
- `subscriptionLifecycleService.js` : Subscription lifecycle management (admin endpoints)
- `planFeaturesService.js` : Plan features CRUD operations
- `notificationsService.js` : Notification management and display

Key Features & User Experience

Dashboard & Navigation:

- **Main Dashboard**: System overview with health status, quick stats, notification center, and quick navigation
- **Quick Search**: Global search functionality for quick navigation to any page
- **Breadcrumb Navigation**: Context-aware breadcrumb navigation for deep pages
- **Role-Based Navigation**: Menu items and features displayed based on user role (admin, sysadmin, root)
- **Notification Center**: System-wide notifications with dismissal and priority levels

User Management:

- **User List**: Paginated table with search, filtering by account status (active/terminated/deactivated)
- **User Detail**: Comprehensive user profile view with tabs for profile, groups, treks, subscriptions, activity
- **Account Actions**: Activate, deactivate, terminate accounts with secure confirmation dialogs
- **Role Management**: View and edit user roles
- **Activity Tracking**: View user activity logs and engagement scores

Content Management:

- **Content Lists**: Paginated lists for all content types (groups, treks, comments, messages, assets, moments)
- **Content Detail Pages**: Comprehensive detail views with metadata, status, and related content
- **Content Moderation**: Flag management, hide/restore, delete with scheduled deletion
- **Status Management**: View and update content status (groups, treks)

Subscription & Billing:

- **Plan Management**: Create, edit, delete subscription plans
- **Feature Management**: Manage plan features with title, description, display order, basic/advanced classification
- **Live Preview**: Preview subscription plans exactly as end-users see them
- **Plan Comparison**: Side-by-side feature comparison table
- **Subscription Lifecycle**: Monitor and manage subscription lifecycle (active, grace period, suspended, etc.)
- **Billing Transactions**: View and filter billing transactions with comprehensive filtering
- **Invoice Management**: View invoices, download PDFs, regenerate PDFs, view invoice details with embedded PDF viewer
- **Feature Flags**: Manage yearly billing and pricing enabled flags

Analytics & Statistics:

- **Dashboard Analytics**: System-wide statistics with interactive charts (Recharts)
- **Engagement Scores**: User engagement scoring, leaderboards, distribution analysis
- **Historical Analytics**: Trends over time with customizable date ranges
- **Entity Statistics**: Detailed statistics for groups, treks, users, content
- **Content Statistics**: Statistics for messages, comments, assets, moments
- **Real-Time Monitoring**: Live system metrics and server monitoring
- **Process Monitoring**: Background process execution history and status

Security & Audit:

- **Security Dashboard**: Security overview with audit logs
- **Account Lifecycle**: Manage account status (activate, deactivate, terminate)
- **Session Management**: View and manage user sessions
- **Audit Compliance**: Comprehensive audit trail navigation
- **Role Management**: User role assignment and management

Server Management:

- **System Configuration**: Edit system parameters (rate limits, email limits, etc.)
- **Background Processes**: Monitor and restart 13+ background processes
- **Process Logs**: View process execution logs and activity
- **System Parameters**: View and edit system-wide configuration

- **File Management**: System file management and integrity checks
- **Server Logs**: View and search application logs

Email Management:

- **Email Queue**: View and manage queued emails
- **Email Archives**: Search and view email history
- **Email Rate Limiting**: Configure email limits per user tier
- **Email System**: Email system configuration and monitoring

Documentation:

- **API Reference**: Interactive API documentation browser with 56+ categories, search, filtering
- **Server Documentation**: Categorized documentation viewer with 15+ categories
- **Document Viewing**: Markdown rendering, PDF download, HTML export

Development Patterns

Component Patterns:

- **Functional Components**: All components use function syntax with hooks (no class components)
- **Custom Hooks**: Reusable logic extracted to custom hooks where appropriate
- **Error Boundaries**: Error handling at component level
- **Loading States**: Consistent loading indicators (`CircularProgress` component)
- **Empty States**: User-friendly empty state messages
- **Conditional Rendering**: Extensive use of conditional rendering based on authentication, roles, and data availability
- **Protected Routes**: Authentication and role checks before rendering protected content

Data Fetching:

- **useEffect**: Fetch data on component mount
- **Dependency Arrays**: Proper dependency management to prevent infinite loops
- **Loading States**: Show spinners during data fetching
- **Error Handling**: Display user-friendly error messages
- **Pagination**: Server-side pagination with URL state management
- **Filtering**: URL-based filtering with query parameters

Form Handling:

- **Controlled Components**: All form inputs are controlled
- **Validation**: Client-side validation before API calls
- **Error Display**: Inline error messages for form fields
- **Submit Handling**: Prevent double-submission, show loading states
- **Modal Forms**: Extensive use of modals for create/edit operations

Navigation:

- **Programmatic Navigation**: Use `useNavigate` hook from React Router
- **Route Parameters**: Access via `useParams` hook
- **Query Parameters**: Access via `useSearchParams` hook for filtering and pagination
- **State Passing**: Pass state between routes via `navigate(path, { state: {...} })`

Modal Patterns:

- **Full Modal**: Overlay with backdrop, prevents outside clicks

- **Confirmation Dialogs**: For destructive actions (with double confirmation for critical operations)
- **Form Modals**: For editing and creating entities
- **Toast Notifications**: For success/error feedback (non-blocking)

Data Visualization:

- **Recharts Integration**: Interactive charts for analytics and statistics
- **Responsive Charts**: Charts adapt to container size
- **Chart Types**: Line charts, bar charts, pie charts, area charts
- **Real-Time Updates**: Charts update with new data

File Organization Best Practices

Component Files:

- Each component in its own file
- Component and styles in same file (inline styles)
- Related components grouped in feature folders
- Detail components often nested in same folder as list components

Constants:

- Application-wide constants in `constants.js`
- API endpoints, URLs, configuration values
- Environment-specific values (development vs. production)
- Page metadata in `constants/pages.js`

Services:

- One service file per domain (auth, invoices, subscriptions, etc.)
- Service functions handle API communication
- Error handling centralized in services
- Token management handled by `authService`

Utilities:

- Utility functions for common operations
- Date formatting, number formatting
- Data transformation helpers

Development Workflow

Getting Started:

1. **Install Dependencies**: `npm install`
2. **Start Development Server**: `npm start` (runs on port 4100)
3. **Configure Backend**: Ensure backend server is running on port 3000
4. **Environment Setup**: Configure `constants.js` for development/production

Development Commands:

- `npm start` : Start development server (port 4100)
- `npm build` : Build production bundle
- `npm test` : Run test suite (if configured)

Code Quality:

- Follow React best practices (hooks, functional components)

- Use ESLint for code quality
- Maintain consistent naming conventions
- Add comments for complex logic
- Handle errors gracefully
- Remove debug code before committing

Testing:

- Test user flows end-to-end
- Test responsive design on multiple screen sizes
- Test authentication and role-based access
- Test API integration
- Test error scenarios
- Test pagination and filtering

Integration with Backend

API Endpoints:

- All admin endpoints prefixed with `/api/v1/admin/`
- Authentication required for all endpoints (JWT token)
- Role-based access control (admin, sysadmin, root)
- Standard request/response format (JSON)
- Error responses follow consistent structure

Pagination:

- Server-side pagination with `page` and `limit` parameters
- URL-based pagination state
- Pagination controls with page numbers and page size selection

Filtering:

- URL-based filtering with query parameters
- Filter state synchronized with URL
- Filter clearing and reset functionality

Real-Time Features:

- Polling for real-time data when needed (e.g., process status, server metrics)
- Optimistic updates for better UX (e.g., immediate UI updates before API confirmation)
- Auto-refresh for monitoring pages

File Operations:

- PDF download and viewing via blob URLs
- File upload with progress tracking
- Secure file access with authentication

Error Handling:

- Network errors: Display user-friendly messages
- 401 errors: Automatic redirect to sign-in
- 403 errors: Show permission denied message
- 404 errors: Show not found message
- 500 errors: Show server error message

Performance Considerations

Optimization Strategies:

- **Code Splitting**: Lazy load routes and components where appropriate
- **Memoization**: Use `React.memo` for expensive components
- **Pagination**: Server-side pagination to limit data transfer
- **Debouncing**: Debounce search inputs to reduce API calls
- **Caching**: Cache API responses in component state when appropriate

Bundle Size:

- Tree shaking for unused code
- Dynamic imports for large libraries (if needed)
- Optimize images and assets

Unique Features

Dynamic Configuration:

- Background process configuration loaded dynamically from server
- System parameters editable through UI
- Feature flags for enabling/disabling features (yearly billing, pricing)

Comprehensive Filtering:

- Multi-field filtering with URL synchronization
- Filter persistence across navigation
- Clear filters functionality

Data Visualization:

- Interactive charts for analytics
- Real-time metrics display
- Historical trend analysis

PDF Management:

- Embedded PDF viewer for invoices
- PDF download and regeneration
- Secure PDF access with authentication

Role-Based UI:

- UI elements shown/hidden based on user role
- Action buttons disabled for unauthorized users
- Navigation items filtered by role

Future Enhancements

Planned Features:

- Enhanced real-time monitoring with WebSocket support
- Advanced search across all content types
- Bulk operations for user management
- Export functionality for reports and analytics
- Enhanced mobile responsiveness
- Keyboard shortcuts for power users

Technical Improvements:

- State management optimization (consider Redux/Zustand if needed)
- Performance monitoring integration
- Enhanced error tracking
- Automated testing suite
- TypeScript migration (if desired)
- Enhanced accessibility (ARIA labels, keyboard navigation)

Resources for Developers

Key Files to Understand:

- `src/App.js` : Main application component with routing and authentication
- `src/components/signedin/dashboard/Dashboard.js` : Main dashboard component
- `src/services/authService.js` : Authentication and token management
- `src/constants.js` : Application configuration (environment variables, URLs, API endpoints)
- `src/components/shared/AdminPageLayout.js` : Main layout wrapper (Header + Footer)
- `src/components/shared/AdminHeader.js` : Navigation header with user menu, role-based navigation

Documentation:

- React documentation: https://react.dev
- React Router documentation: https://reactrouter.com
- Recharts documentation: https://recharts.org
- Axios documentation: https://axios-http.com

Getting Help:

- Review existing components for patterns
- Check shared components for reusable solutions
- Review API integration examples in feature components
- Consult backend API documentation for endpoint details
- Check `constants/pages.js` for page metadata and keywords

End-User React Client Application

Overview

The pTreks end-user client is a modern, responsive React.js web application that provides a comprehensive interface for outdoor enthusiasts to manage groups, organize treks, track GPS locations, communicate with fellow adventurers, and manage their subscriptions. The application is built with a focus on user experience, performance, and maintainability.

Technology Stack

Core Framework:

- **React 18.2.0**: Modern React with hooks, concurrent features, and improved performance
- **React Router DOM 6.30.1**: Client-side routing with nested routes and navigation
- **React Scripts 5.0.1**: Create React App configuration for development and build

State Management:

- **React Context API**: Global state management for authentication, user data, groups, treks, stats, and notifications
- **Local Storage**: Token persistence and user preference storage
- **Custom Hooks**: Reusable stateful logic (`useAuth` , `useToast` , `useStats` , etc.)

UI Libraries & Tools:

- **React Icons 5.5.0**: Comprehensive icon library (Font Awesome, Material Design, etc.)
- **GSAP 3.13.0**: Advanced animations and transitions
- **Leaflet 1.9.4 / React-Leaflet 4.2.1**: Interactive maps for GPS tracking and route visualization
- **Marked 12.0.0**: Markdown parsing for documentation and content rendering
- **Axios 1.11.0**: HTTP client for API communication with interceptors

Development Tools:

- **Port 4000**: Development server port (configurable via `PORT` environment variable)
- **ESLint**: Code quality and style enforcement
- **Web Vitals**: Performance monitoring

Application Architecture

Component Structure:

```
src/
├── components/
|   ├── authentication/      # Sign-in, sign-up, password reset
|   ├── developer/           # Developer documentation and tools
|   ├── faq/                 # Frequently asked questions
|   ├── help/                # Help system with topics
|   ├── home/
|   |   ├── signedIn/        # Authenticated user features
|   |   |   ├── account/     # Account management
|   |   |   ├── groups/      # Group management
|   |   |   ├── treks/       # Trek management
|   |   |   └── invoices/    # Billing and invoices
|   |   └── signedOut/       # Public pages (pricing, features, etc.)
|   ├── onboarding/          # New user onboarding flow
|   ├── search/              # Global search functionality
|   └── shared/              # Reusable components
├── contexts/                # React Context providers
├── services/                # Business logic services
├── utils/                   # Utility functions
└── Constants.js             # Application constants
```

Routing Architecture:

- **Homepage** (`/` or `/main`): Shows different content based on authentication status
  - **Signed-out**: Marketing content (Hero, About, Testimonials, Image Gallery, Carousel)
  - **Signed-in**: Dashboard with groups overview (`SignedInGroups` component)
- **Public Routes**: Pricing (`/pricing`), plan comparison (`/pricing/compare`), features (`/features/*`), about (`/about`), contact (`/contact`), FAQ (`/faq`), help (`/help`), developer recruitment (`/join`)

- **Authentication Routes**: Sign-in (`/signin`), sign-up (`/signup`), email verification (`/signup/confirm`), password reset (`/forgot-password` , `/reset-password`), invite verification (`/invite/verify`), sign-out (`/signout`)
- **Onboarding Routes**: Desktop (`/onboarding/1` through `/onboarding/7`), Mobile (`/mobile-onboarding/1` through `/mobile-onboarding/5`)
- **Protected Routes**: Dashboard (`/account`), groups (`/groups`), treks (`/treks`), account management, billing (`/account/billing` , `/invoices`)
- **Dynamic Routes**: Group details (`/group/:groupId/*`), trek details (`/trek/:trekId/*`), invoice details (`/invoice/:invoiceNumber`)
- **Nested Routes**: Group management (comments, messages, files, users, QR codes, invitations), trek management (comments, messages, moments, files, users, participants, QR codes, invitations)
- **Legal Routes**: Terms of Use (`/tou`), Privacy Policy (`/pp`)
- **Developer Routes**: Documentation (`/developer/documents`), document viewers (`/developer/document/:filename` , `/developer/client-document/:filename`)

State Management

Context Providers (wrapped in `App.js`):

1. **AuthContext**: User authentication, profile data, account status
2. **ToastContext**: Global toast notifications for user feedback
3. **GroupsContext**: Groups data and operations
4. **StatsContext**: User statistics and engagement scores
5. **UpcomingTreksContext**: Upcoming treks data
6. **ModeContext**: Application mode (beginner/advanced)

State Patterns:

- **Local State**: Component-specific state using `useState`
- **Context State**: Global shared state via Context API
- **Derived State**: Computed values from props or other state
- **Server State**: Data fetched from API, cached in context or local storage

Authentication Flow:

1. User signs in → JWT token stored in `localStorage`
2. Token included in `Authorization: Bearer <token>` header for all API requests
3. `AuthContext` manages authentication state and user profile
4. Protected routes check `isAuthenticated` from context
5. Token expiration handled via axios interceptor (401 redirects to sign-in)

Component Organization

Shared Components (`src/components/shared/`):

- **Layout Components**: `Layout` (Header + Footer wrapper), `PageWrapper` (wraps Layout with App div)
- **Core Layout**: `Header` (navigation, user menu, mobile menu), `Footer` (links, copyright, developer section)
- **Modals**: `DeleteConfirmationModal` , `EditCommentModal` , `EditMessageModal` , `NotificationConfirmationDialog` , `AccountStatusModal`
- **Forms**: `NewCommentModal` , `NewMessageModal`
- **UI Elements**: `CircularProgress` , `DashboardCard` , `EngagementScoreCard` , `Toast` , `NotificationBanner`
- **Specialized**: `QRCodeScanner` (for joining groups/treks), `GeocodingIndicator` , `InvitationLimitsDisplay` , `CaptchaModal`

- **Error Handling**: `ServerErrorHandler` (displays server errors to users)

Feature-Specific Components:

- **Groups**: `Groups`, `GroupNew`, `GroupEdit`, `GroupManageFiles`, `GroupManageUsers`
- **Treks**: `Treks`, `TrekNew`, `TrekEdit`, `TrekDetails`, `TrekMessages`, `TrekComments`
- **Account**: `Dashboard`, `Profile`, `BillingDashboard`, `Invoices`
- **Billing**: `Pricing`, `PricingCompare`, `PaymentConfirmationModal`, `PlanChangeSuccessModal`

Naming Conventions:

- **Components**: PascalCase (e.g., `Dashboard.js`, `GroupEdit.js`)
- **Files**: Match component name exactly
- **CSS Files**: Same name as component (e.g., `Dashboard.css`)
- **Folders**: lowercase-only (e.g., `groups`, `treks`, `account`)

Styling Approach

CSS Architecture:

- **Component-Scoped CSS**: Each component has its own `.css` file
- **Global Styles**: App-level styles in `App.css`
- **Responsive Design**: Mobile-first approach with media queries
- **Dark Theme**: Consistent black/near-black theme (`#000000`, `#0a0a0a`, `#1a1a1a`)
- **Accent Colors**: Green (`#4CAF50`) for primary actions, blue for information

Style Patterns:

- **Color Scheme**: Pure black backgrounds, `#1a1a1a` content boxes, `#333` borders
- **Typography**: Nunito font family, consistent font sizes (12px base, scaled headings)
- **Spacing**: Consistent padding and margins using `rem` units
- **Responsive Breakpoints**: Mobile (480px), Tablet (768px), Desktop (1024px+)
- **CSS Overrides**: Use `!important` when necessary for component isolation

Responsive Design:

- **Mobile**: Single column layouts, stacked elements, compact navigation
- **Tablet**: 2-column grids where appropriate, optimized spacing
- **Desktop**: Multi-column layouts, full feature sets, expanded navigation

API Integration

API Client (`src/utils/api.js`):

- **Axios Instance**: Configured with base URL and default headers
- **Request Interceptor**: Automatically adds JWT token to all requests
- **Response Interceptor**: Handles 401 errors (token expiration) with automatic redirect
- **Error Handling**: Centralized error logging and user-friendly error messages
- **Base URL**: `http://localhost:3000/api/v1` (development) or configured production URL

API Patterns:

```
// Standard API call pattern
import api from '../utils/api';

const fetchData = async () => {
```

```
  try {
    const response = await api.get('/endpoint');
    return response.data;
  } catch (error) {
    console.error('API Error:', error);
    // Handle error (toast notification, etc.)
  }
};
```

Service Layer (`src/services/`):

- `StatsService.js` : User statistics and engagement score calculations, formatting utilities (date, time, distance, storage size, duration)
- `NotificationsService.js` : Notification management and sending (group/trek notifications, mentions, etc.)
- `EmailUsageService.js` : Email invitation limit tracking and validation
- `SystemParametersService.js` : System configuration fetching (rate limits, email limits, etc.)

Key Features & User Experience

Authentication & Onboarding:

- Email-based registration with verification code
- **Optional onboarding flow**: 7-step desktop onboarding or 5-step mobile onboarding (accessible via header button, not automatic)
- Onboarding covers: Welcome, features overview, getting started, creating groups, creating treks, GPS tracking, and completion
- Password reset with secure token flow (forgot password → email verification → reset)
- Account status monitoring (active, suspended, etc.) via `AccountStatusModal`
- Session management with automatic token refresh
- Email verification required before full account access
- Invite code verification for group/trek invitations

Group Management:

- Create, edit, and delete groups (public/private)
- Group member management with role-based permissions
- QR code generation for easy group joining
- Group messaging and comments
- File upload and management
- Group search and discovery

Trek Management:

- Create treks with GPS route planning
- Real-time GPS location tracking during treks
- Trek messaging and comments
- Photo/video uploads (moments) during treks
- Trek statistics and history
- Participant management

Communication:

- Real-time messaging (groups and treks)
- Comment threads on groups, treks, and moments
- Edit/delete own messages and comments

- Notification system for mentions and updates
- Private and public communication options

Billing & Subscriptions:

- View available plans (Free, Pro, Team, Enterprise)
- Upgrade/downgrade subscriptions
- Billing cycle management (monthly/yearly)
- Invoice viewing and PDF download
- Stripe payment integration
- Plan comparison table
- Pending plan change management

Dashboard & Analytics:

- Personal statistics overview (groups, treks, messages, comments, assets, sessions)
- Engagement score display with level indicators
- Activity history and trends
- Upcoming treks (from `UpcomingTreksContext`)
- Recent groups (from `GroupsContext`)
- Profile photo management
- Quick actions and navigation

Search & Discovery:

- Global search functionality (`/search`)
- Search groups, treks, and users
- Filter and sort results
- Help system with searchable topics (`/help`)
- Context-aware help (pre-filled search terms)

Legal & Documentation:

- Terms of Use and Privacy Policy pages
- Developer documentation viewer (markdown rendering)
- Client and server document viewing
- Document categorization and navigation

Developer Features (for developers+):

- Access to technical documentation
- Server and client document viewers
- Developer onboarding guide
- API documentation access

Development Patterns

Component Patterns:

- **Functional Components**: All components use function syntax with hooks (no class components)
- **Custom Hooks**: Reusable logic extracted to custom hooks (e.g., `useAuth`, `useToast`, `useStats`, `useGroups`)
- **Error Boundaries**: Error handling at component level (ErrorTracker utility)
- **Loading States**: Consistent loading indicators (`CircularProgress` component)
- **Empty States**: User-friendly empty state messages (e.g., "No groups yet", "No treks found")

- **Conditional Rendering**: Extensive use of conditional rendering based on authentication, roles, and data availability
- **Protected Routes**: Authentication checks before rendering protected content

Data Fetching:

- **useEffect**: Fetch data on component mount
- **Dependency Arrays**: Proper dependency management to prevent infinite loops
- **Loading States**: Show spinners during data fetching
- **Error Handling**: Display user-friendly error messages
- **Caching**: Cache data in context or local storage when appropriate

Form Handling:

- **Controlled Components**: All form inputs are controlled
- **Validation**: Client-side validation before API calls
- **Error Display**: Inline error messages for form fields
- **Submit Handling**: Prevent double-submission, show loading states

Navigation:

- **Programmatic Navigation**: Use `useNavigate` hook from React Router
- **Route Parameters**: Access via `useParams` hook
- **Query Parameters**: Access via `useSearchParams` hook
- **State Passing**: Pass state between routes via `navigate(path, { state: {...} })`

Modal Patterns:

- **Full Modal**: Overlay with backdrop, prevents outside clicks
- **Confirmation Dialogs**: For destructive actions
- **Form Modals**: For editing and creating entities
- **Toast Notifications**: For success/error feedback (non-blocking)

File Organization Best Practices

Component Files:

- Each component in its own file
- Component and CSS file in same directory
- Related components grouped in feature folders

Constants:

- Application-wide constants in `Constants.js`
- API endpoints, URLs, configuration values
- Environment-specific values (development vs. production)

Utilities (`src/utils/`):

- `api.js` : Axios instance with interceptors for authentication and error handling
- `ErrorTracker.js` : Error tracking and logging utility (initialized in App.js)
- `geocoding.js` : Geocoding utilities for location services
- `passwordUtils.js` : Password validation and security utilities
- `clientDocumentParser.js` : Markdown document parsing for client-side docs
- `dialogPreferences.js` : User preferences for dialog behavior (localStorage)
- `emailInvitationValidation.js` : Email invitation validation logic
- `onboardingNavigation.js` : Onboarding route navigation helpers (detects mobile vs desktop)

Context Providers:

- One context per domain (auth, groups, stats, etc.)
- Provider components in `src/contexts/`
- Custom hooks for consuming context (e.g., `useAuth()`)

Development Workflow

Getting Started:

1. **Install Dependencies**: `npm install`
2. **Start Development Server**: `npm start` (runs on port 4000)
3. **Configure Backend**: Ensure backend server is running on port 3000
4. **Environment Setup**: Configure `Constants.js` for development/production

Development Commands:

- `npm start`: Start development server (port 4000)
- `npm build`: Build production bundle
- `npm test`: Run test suite (if configured)
- `npm run docs:all`: Generate documentation (HTML and PDF)

Code Quality:

- Follow React best practices (hooks, functional components)
- Use ESLint for code quality
- Maintain consistent naming conventions
- Add comments for complex logic
- Handle errors gracefully

Testing:

- Test user flows end-to-end
- Test responsive design on multiple screen sizes
- Test authentication flows
- Test API integration
- Test error scenarios

Integration with Backend

API Endpoints:

- All endpoints prefixed with `/api/v1/`
- Authentication required for most endpoints (JWT token)
- Standard request/response format (JSON)
- Error responses follow consistent structure

Real-Time Features:

- WebSocket support for live updates (planned, not yet fully implemented)
- Polling for real-time data when needed (e.g., GPS locations during active treks)
- Optimistic updates for better UX (e.g., immediate UI updates before API confirmation)
- Context providers automatically refresh data on navigation

File Uploads:

- FormData for multipart/form-data uploads
- Progress tracking for large files
- Thumbnail generation handled by backend

- File validation on client and server

Error Handling:

- Network errors: Display user-friendly messages
- 401 errors: Automatic redirect to sign-in
- 403 errors: Show permission denied message
- 404 errors: Show not found message
- 500 errors: Show server error message

Performance Considerations

Optimization Strategies:

- **Code Splitting**: Lazy load routes and components
- **Memoization**: Use `React.memo` for expensive components
- **Virtualization**: For long lists (if needed)
- **Image Optimization**: Use appropriate image sizes and formats
- **Caching**: Cache API responses when appropriate

Bundle Size:

- Tree shaking for unused code
- Dynamic imports for large libraries
- Optimize images and assets

Future Enhancements

Planned Features:

- Progressive Web App (PWA) support
- Offline functionality
- Push notifications
- Enhanced mobile responsiveness
- Advanced search with filters
- Real-time collaboration features
- Enhanced analytics dashboard

Technical Improvements:

- State management migration (Redux/Zustand) if needed
- Performance monitoring integration
- Enhanced error tracking
- Automated testing suite
- TypeScript migration (if desired)
- Enhanced accessibility (ARIA labels, keyboard navigation)

Resources for Developers

Key Files to Understand:

- `src/App.js` : Main application component with routing and context provider hierarchy
- `src/index.js` : Application entry point (React 18 root API)
- `src/contexts/AuthContext.js` : Authentication state management, user profile, account status
- `src/utils/api.js` : API client configuration with interceptors
- `src/Constants.js` : Application configuration (environment variables, URLs, API endpoints)

- `src/components/shared/Layout.js` : Main layout wrapper (Header + Footer)
- `src/components/shared/PageWrapper.js` : Page wrapper with Layout integration
- `src/components/shared/Header.js` : Navigation header with user menu, mobile menu, role-based navigation
- `src/components/shared/Footer.js` : Footer with links, developer section (conditional on role)
- `src/components/shared/` : Reusable component library (modals, forms, UI elements)

Documentation:

- React documentation: https://react.dev
- React Router documentation: https://reactrouter.com
- Axios documentation: https://axios-http.com
- Leaflet documentation: https://leafletjs.com

Getting Help:

- Review existing components for patterns
- Check shared components for reusable solutions
- Review API integration examples in feature components
- Consult backend API documentation for endpoint details

API Architecture

API Structure

Base URL: `/api/v1/`

Versioning: URL-based versioning (`/api/v1/`) allows for future `v2` without breaking existing clients

Endpoint Categories

Public Endpoints (No Authentication)

- `GET /api/v1/subscription-plans` - View available plans
- `GET /api/v1/subscription-plans/configuration` - View billing configuration
- `POST /api/v1/auth/signup` - User registration
- `POST /api/v1/auth/verify-email` - Email verification
- `POST /api/v1/auth/signin` - User sign-in
- `POST /api/v1/auth/forgot-password` - Password reset request

User Endpoints (Authentication Required)

- **Users**: `/api/v1/users/*` - User profile management
- **Groups**: `/api/v1/groups/*` - Group operations
- **Treks**: `/api/v1/treks/*` - Trek operations
- **GPS Locations**: `/api/v1/gps-locations/*` - Location tracking
- **Messages**: `/api/v1/group-messages/*` , `/api/v1/trek-messages/*` - Communication
- **Comments**: `/api/v1/group-comments/*` , `/api/v1/trek-comments/*` - Discussions
- **Assets**: `/api/v1/assets/*` - File uploads
- **Moments**: `/api/v1/moments/*` - Trek-specific photos/videos
- **Subscriptions**: `/api/v1/subscriptions/*` - Subscription management
- **Analytics**: `/api/v1/analytics/*` - Personal analytics

Admin Endpoints (Admin+ Role Required)

- **Users**: `/api/v1/admin/users/*` - User management
- **Moderation**: `/api/v1/admin/content-flags/*` - Content moderation
- **Analytics**: `/api/v1/admin/engagement-scores/*` - System analytics
- **Email**: `/api/v1/admin/email-queue/*`, `/api/v1/admin/email-archives/*` - Email management
- **System**: `/api/v1/admin/system-parameters/*` - System configuration
- **Background Processes**: `/api/v1/admin/background-processes/*` - Process management
- **Subscriptions**: `/api/v1/admin/subscriptions/*` - Subscription management

Request/Response Format

Request Headers:

```
Authorization: Bearer <JWT_TOKEN>
Content-Type: application/json
```

Response Format:

```
{
  "success": true,
  "data": { ... },
  "message": "Optional message"
}
```

Error Response:

```
{
  "success": false,
  "error": "Error message",
  "details": { ... }
}
```

Pagination

Most list endpoints support pagination:

- `page` (default: 1)
- `limit` (default: 10, max: 100)

Response Format:

```
{
  "success": true,
  "data": [ ... ],
  "pagination": {
    "page": 1,
    "limit": 10,
    "total": 150,
    "totalPages": 15
```

```
  }
}
```

Rate Limiting

Rate limiting is enforced based on user cost tier (free, paid, admin):

- Tier-based limits per endpoint category
- Configurable via `rate_limit_rules` table
- Emergency override via `EMERGENCY_DISABLE_RATE_LIMITING` environment variable

API Documentation

Complete API documentation available at:

- `Documents/Public/Source/02-api-documentation/API.md` - Comprehensive endpoint reference
- Swagger documentation (if configured) - Interactive API explorer

---

Database Architecture

Database Overview

PostgreSQL with PostGIS extension for geospatial capabilities

Statistics:

- 54 tables
- 32 functions (stored procedures)
- 45 triggers (automated actions)
- 270 indexes (performance optimization)
- ~3.5M rows (99.7% in application_logs)
- 1.9 GB database size

Core Tables

User Management

- `users` - User accounts and authentication
- `user_profiles` - Extended user information
- `user_roles` - Role assignments (RBAC)
- `user_sessions` - Active user sessions
- `user_cost_tiers` - Subscription-based rate limiting tiers
- `user_engagement_scores` - Daily engagement calculations

Groups & Treks

- `groups` - User groups/organizations
- `group_users` - Group membership
- `treks` - Outdoor activities/journeys
- `trek_users` - Trek participation
- `gps_locations` - Real-time location tracking (PostGIS)

Communication

- `group_messages` - Group messaging
- `trek_messages` - Trek messaging
- `group_comments` - Group discussions
- `trek_comments` - Trek discussions

Content Management

- `assets` - File metadata
- `moments` - Trek-specific photos/videos
- `content_flags` - Content moderation flags
- `flag_history` - Flag action history

Subscriptions & Billing

- `subscription_plans` - Plan definitions (Free, Pro, Team, Enterprise)
- `plan_features` - Feature descriptions per plan (informational)
- `user_subscriptions` - Active user subscriptions
- `subscription_plan_changes` - Plan change history
- `billing_transactions` - Payment records
- `usage_tracking` - Feature usage monitoring

System Management

- `system_parameters` - System-wide configuration
- `rate_limit_rules` - Rate limiting configuration
- `email_queue` - Queued email messages
- `email_archives` - Email history
- `application_logs` - Comprehensive audit trail
- `user_activity_logs` - User activity tracking

Database Design Principles

1. **Normalization**: Data normalized to 3NF to prevent redundancy
2. **Referential Integrity**: Foreign key constraints ensure data consistency
3. **Audit Trails**: Comprehensive logging via `application_logs` and `flag_history`
4. **Soft Deletes**: Content flagged for deletion rather than immediately removed
5. **Geospatial Support**: PostGIS for GPS location storage and queries
6. **JSON Support**: JSONB columns for flexible data (engagement scores, raw data)

Key Relationships

- **Users** → **Groups**: Many-to-many via `group_users`
- **Users** → **Treks**: Many-to-many via `trek_users`
- **Groups** → **Treks**: One-to-many (treks belong to groups)
- **Users** → **Subscriptions**: One-to-many (users can have subscription history)
- **Users** → **Roles**: Many-to-many via `user_roles`
- **Content** → **Flags**: One-to-many (content can have multiple flags)

Database Triggers

Automated triggers handle:

- User profile creation on user registration

- User cost tier updates on subscription changes
- Timestamp management (`created_at` , `updated_at`)
- Geometry calculation for GPS coordinates
- Cascade deletions for data integrity

Database Functions

PostgreSQL functions provide:

- Complex calculations (engagement scores, prorations)
- Data transformations
- Reusable query logic
- Performance optimization

Core Features & Capabilities

Authentication & Authorization

JWT-Based Authentication:

- Token-based stateless authentication
- Tokens expire after 24 hours (configurable)
- Refresh token support for extended sessions

Role-Based Access Control (RBAC):

- 7-tier role system: `root` , `sysadmin` , `admin` , `developer` , `moderator` , `member` , `guest`
- Roles assigned via `user_roles` table
- Role-based endpoint access control
- Role-based rate limiting (cost tiers)

Email Verification:

- Required for account activation
- Secure token-based verification
- Expiring verification links

Subscription & Billing

Four-Tier Pricing Model:

- **Free**: $0/month - Basic features (5 groups, 5 members, 5 treks)
- **Pro**: $9.99/month - Enhanced features (25 groups, 25 members, 25 treks)
- **Team**: $24.99/month - Team features (50 groups, 50 members, 50 treks)
- **Enterprise**: $49.99/month - Unlimited features

Stripe Integration:

- Payment processing via Stripe
- Webhook handling for payment events
- Automatic subscription lifecycle management
- Proration calculations for plan changes

Subscription Lifecycle:

- Active, Grace Period, Suspended, Canceled, Expired, Paused

- Automated renewals
- Grace period for payment failures (7 days)
- Automatic downgrades on expiration

Real-Time Features

GPS Location Tracking:

- Real-time location updates during treks
- PostGIS geospatial storage and queries
- Location history tracking
- Proximity-based queries

WebSocket Support:

- Real-time messaging
- Live location updates
- Instant notifications
- Connection management

File Management

Hybrid Storage Strategy:

- **File System**: Large files (photos, videos) stored on disk
- **Database**: Small files (profile photos) stored as BYTEA
- **Thumbnails**: Automatic generation with 'TN' suffix

Security:

- File type validation
- Size limits per tier
- Virus scanning (if configured)
- Secure file paths (hashed directory structure)

Communication System

Messaging:

- Group messaging (persistent)
- Trek messaging (during active treks)
- Real-time delivery via WebSocket
- Message history and pagination

Comments:

- Comments on groups, treks, and moments
- Threaded discussions
- Edit/delete own comments
- Moderation support

Analytics & Engagement

Engagement Scoring:

- Daily engagement score calculation
- Multi-factor scoring (groups, treks, messages, comments)
- Historical tracking and trends

- User segmentation

Activity Tracking:

- Comprehensive user activity logging
- Feature usage tracking
- Analytics snapshots
- System-wide statistics

Moderation System

Content Flagging:

- Flag inappropriate content
- Multiple flag types (spam, inappropriate, etc.)
- Flag history tracking
- Automated moderation workflows

Admin Moderation:

- Review flagged content
- Approve/reject flags
- Hide/restore content
- Permanent deletion

Security & Authentication

Security Layers

1. **HTTP Security Headers** (Helmet.js):
    - Content Security Policy (CSP)
    - HTTP Strict Transport Security (HSTS)
    - X-Frame-Options
    - X-Content-Type-Options
2. **CORS Configuration**:
    - Restricted origins
    - Credential support
    - Preflight handling
3. **Rate Limiting**:
    - Tier-based limits (free, paid, admin)
    - Category-based limits (entity_create, email_send, etc.)
    - Emergency override capability
4. **Input Validation**:
    - Joi schema validation
    - SQL injection prevention (parameterized queries)
    - XSS prevention (input sanitization)
5. **File Upload Security**:
    - File type validation
    - Size limits

- Secure file storage
- Virus scanning (optional)

6. **Password Security**:
    - bcrypt hashing (10 rounds)
    - Password complexity requirements
    - Secure password reset flow

Authentication Flow

1. **Signup**:
    - User provides email, password, name
    - Account created with `is_verified=false`
    - Verification email sent
    - Free subscription automatically assigned
2. **Email Verification**:
    - User clicks verification link
    - Account activated (`is_verified=true`)
    - User cost tier created
    - Sign-in enabled
3. **Sign-In**:
    - User provides email and password
    - Password verified via bcrypt
    - JWT token generated
    - Token returned to client
    - Active subscription required (enforced)
4. **Token Usage**:
    - Client includes token in `Authorization: Bearer <token>` header
    - Middleware validates token
    - User context attached to request
    - Role-based access control applied

Role Hierarchy

```
root (highest privilege)
  ↓
sysadmin
  ↓
admin
  ↓
developer
  ↓
moderator
  ↓
member
  ↓
guest (lowest privilege)
```

Roles are assigned via `user_roles` table. Users can have multiple roles, with highest privilege role used for access control.

Background Processes

Process Management

All background processes managed via **PM2** with ecosystem configuration:

Configuration File: `ecosystem.config.js`

Process Types:

- **Fork Mode**: Independent Node.js processes
- **Cron Mode**: Scheduled tasks (PM2 cron syntax)

Key Background Processes

1. **Subscription Maintenance Processor**
    - **Schedule**: Every 15 minutes
    - **Purpose**: Handles subscription lifecycle (expirations, renewals, grace periods)
    - **Tasks**: Expire subscriptions, process renewals, handle grace periods
2. **Email Queue Processor**
    - **Schedule**: Every 1 minute
    - **Purpose**: Processes queued email messages
    - **Tasks**: Send emails, handle failures, retry logic
3. **Account Cleanup Processor**
    - **Schedule**: Daily at 2:00 AM
    - **Purpose**: Removes terminated accounts after grace period
    - **Tasks**: Permanently delete accounts, cleanup related data
4. **Activity Logs Cleanup Processor**
    - **Schedule**: Daily at 2:30 AM
    - **Purpose**: Prunes old activity logs, creates summaries
    - **Tasks**: Delete old logs, create daily summaries
5. **Trek Auto-Finish Processor**
    - **Schedule**: Daily at 3:00 AM
    - **Purpose**: Automatically finishes old treks
    - **Tasks**: Set trek status to "finished" for old treks
6. **Engagement Score Processor**
    - **Schedule**: Daily at 2:00 AM
    - **Purpose**: Calculates daily engagement scores
    - **Tasks**: Compute scores, store in database

Manual Control: All processes can be restarted manually via admin API endpoints

Monitoring: Process status available via `/api/v1/admin/background-processes/status`

Development Workflow

Getting Started

1. **Clone Repository**:

```
git clone <repository-url>
cd backend_node/source
```

2. **Install Dependencies**:

```
npm install
```

3. **Database Setup**:
   - Install PostgreSQL 14+ with PostGIS extension
   - Create database: `createdb ptreks_db_dev`
   - Run database initialization (tables created automatically on server start)

4. **Environment Configuration**:
   - Copy `.env.example` to `.env`
   - Configure database connection
   - Set JWT secret
   - Configure email settings (if needed)

5. **Start Development Server**:

```
npm start
# or
pm2 start ecosystem.config.js
```

Development Commands

- `npm start` - Start server (development mode)
- `npm test` - Run tests (if configured)
- `pm2 start ecosystem.config.js` - Start all processes with PM2
- `pm2 logs` - View logs
- `pm2 restart <process-name>` - Restart specific process

Code Organization

Directory Structure:

```
src/
├── config/          # Configuration files
├── controllers/     # Request handlers (business logic)
├── database/        # Database queries
├── middleware/      # Express middleware
├── models/          # Data models and validation
├── routes/          # Route definitions
├── services/        # Complex business logic
```

```
├── utils/           # Utility functions
└── server.js        # Main server file
```

File Naming Conventions

- **Controllers**: `camelCase.js` (e.g., `userController.js`)
- **Routes**: `camelCaseRoutes.js` (e.g., `userRoutes.js`)
- **Services**: `camelCaseService.js` (e.g., `subscriptionService.js`)
- **Database**: `camelCaseQueries.js` (e.g., `userQueries.js`)
- **Models**: `camelCase.js` (e.g., `validation.js`)

Git Workflow

1. **Create Feature Branch**: `git checkout -b feature/your-feature-name`
2. **Make Changes**: Write code, tests, documentation
3. **Commit**: `git commit -m "Descriptive commit message"`
4. **Push**: `git push origin feature/your-feature-name`
5. **Create Pull Request**: Request code review

Code Style

- **Indentation**: 2 spaces
- **Quotes**: Single quotes for strings
- **Semicolons**: Required
- **Async/Await**: Preferred over promises
- **Error Handling**: Try-catch blocks, proper error responses

Development Patterns

Copy-and-Substitute Pattern: When creating new functionality that mirrors existing working code, always use the copy-and-substitute pattern instead of creating code from scratch:

1. **Copy existing working files** (controllers, queries, routes, validation schemas)
2. **Apply systematic string substitutions** to rename entities (e.g., `assets` → `moments`, `asset` → `moment`)
3. **Make minimal targeted modifications** for differences

Benefits:

- Starts with proven, working code - reduces probability of introducing new bugs
- Faster development - no need to recreate patterns from scratch
- Consistency - maintains existing patterns and conventions
- Easier maintenance - code follows established patterns
- Lower risk - builds on tested functionality

Example: When creating the "moments" system, the team copied the existing "assets" system files and applied systematic substitutions, then made minimal modifications for differences. This pattern is documented in `Documents/Public/Source/12-development/Development_Patterns.md`.

When to Use:

- Similar entities (assets/moments, users/moderators, etc.)
- CRUD operations that follow the same patterns
- File management systems
- API endpoints with similar structures

- Database operations with similar schemas

Configuration & Environment Variables

Configuration Management

Centralized Configuration (`src/config/constants.js`): The constants file serves as the single source of truth for application-wide configuration. It reads from environment variables and provides sensible defaults.

Key Configuration Objects:

1. **PRICING_CONFIG**:
    - `DEFAULT_PLANS` : Array of plan definitions (name, display_name, description, price_monthly, price_yearly)
    - `TIERS` : Tier name constants (FREE, PRO, TEAM, ENTERPRISE)
    - `ENABLE_PRICING` : Master switch (default: false, requires explicit activation)
    - `ENABLE_YEARLY_BILLING` : Feature flag for yearly billing cycles
    - `PLAN_CHANGE_STATUS` : Status constants for subscription plan changes
    - `INVOICE_CONFIG` : Invoice management settings (PDF cache, cleanup, email settings)
2. **FILE_CONFIG**:
    - File paths (treks, groups, moments, temp directories)
    - Maximum file sizes (250MB general, 15MB profile images)
    - Allowed file types (images, videos, documents)
    - Thumbnail configuration (width, quality)
3. **HTTP_STATUS**:
    - Standard HTTP status codes (200, 400, 401, 403, 404, 500, etc.)
4. **ERROR_MESSAGES / SUCCESS_MESSAGES**:
    - Centralized message constants for consistent error and success responses
5. **BACKGROUND_PROCESS_CONFIG**:
    - Process scheduling (intervals, start times)
    - Process enable/disable flags
    - Admin notification settings

Environment Variables

Required Environment Variables (`.env` file):

Database Configuration:

```
DB_HOST=localhost
DB_PORT=5432
DB_NAME=ptreks_db_dev
DB_USER=postgres
DB_PASSWORD=your_password_here
```

Server Configuration:

```
PORT=3000
NODE_ENV=development  # or "production"
```

Security:

```
JWT_SECRET=your_jwt_secret_here
SESSION_SECRET=your_session_secret_here
```

Pricing/Billing System (Disabled by default):

```
# Master controls - MUST be explicitly enabled
ENABLE_PRICING=false  # Set to true to enable billing system
ENABLE_YEARLY_BILLING=false  # Set to true to enable yearly billing cycles

# Stripe Configuration (required if ENABLE_PRICING=true)
STRIPE_SECRET_KEY=sk_test_...  # Use sk_test_ for development, sk_live_ for production
STRIPE_WEBHOOK_SECRET=whsec_...  # Webhook signing secret
UPGRADE_PAYMENT_TIMEOUT_MINUTES=30  # Timeout for payment completion

# Encryption (required if ENABLE_PRICING=true)
ENCRYPTION_KEY_PRICING=secure_random_key_min_32_chars
```

Email Configuration:

```
EMAIL_PROVIDER=resend  # or "gmail" for testing
RESEND_API_KEY=your_resend_api_key_here

# Or for Gmail testing:
# SMTP_HOST=smtp.gmail.com
# SMTP_PORT=587
# SMTP_USER=your_gmail@gmail.com
# SMTP_PASS=your_gmail_app_password
```

Rate Limiting:

```
# Emergency override - completely disables ALL rate limiting system-wide
EMERGENCY_DISABLE_RATE_LIMITING=false  # Set to true only in emergencies
```

Logging:

```
LOG_LEVEL=info  # debug, info, warn, error
CONSOLE_LOG_LEVEL=warn  # Console-specific log level
```

Configuration Best Practices

1. **Never Hardcode Sensitive Values**: Always use environment variables for secrets, API keys, and database credentials

2. **Use Constants.js for Non-Sensitive Config**: Centralize application settings in `constants.js`
3. **Provide Sensible Defaults**: Constants should have fallback values for development
4. **Document Required Variables**: Update `.env.example` when adding new required variables
5. **Validate Critical Config**: Check that required environment variables are set on server startup
6. **Feature Flags**: Use environment variables for feature toggles (e.g., `ENABLE_PRICING`, `ENABLE_YEARLY_BILLING`)

Dynamic Configuration

PRICING_CONFIG.DEFAULT_PLANS:

- Used by `server.js` to seed `subscription_plans` table on startup
- Used by `existingFeatures()` to dynamically resolve plan IDs for `plan_features` seeding
- Used by validation schemas to validate plan names
- Single source of truth for plan definitions - prevents hardcoded plan data

Example Usage:

```
// In server.js - Dynamic plan seeding
const planValues = PRICING_CONFIG.DEFAULT_PLANS.map(
  (plan) => `('${plan.name}', '${plan.display_name}', ...)`
).join(",\n");

// In validation.js - Dynamic plan name validation
name: Joi.string()
  .valid(...PRICING_CONFIG.DEFAULT_PLANS.map(p => p.name))
  .required()

// In planChangeService.js - Using tier constants
const validPlanNames = PRICING_CONFIG.DEFAULT_PLANS.map(p => p.name);
```

Environment-Specific Configuration

Development:

- `NODE_ENV=development`
- `ENABLE_PRICING=false` (default)
- Test Stripe keys (`sk_test_...`)
- Local database connection
- Verbose logging enabled

Production:

- `NODE_ENV=production`
- `ENABLE_PRICING=true` (explicitly set)
- Live Stripe keys (`sk_live_...`)
- Production database connection
- Optimized logging levels

Configuration Loading:

- `dotenv` package loads `.env` file automatically

- Environment variables accessed via `process.env.VARIABLE_NAME`
- Constants.js reads from environment and provides defaults
- Server startup validates critical configuration

Testing & Quality Assurance

Testing Strategy

Manual Testing:

- Test endpoints via Postman or curl
- Verify database state after operations
- Test error scenarios

Automated Testing (if configured):

- Unit tests for utility functions
- Integration tests for API endpoints
- Database transaction tests

Testing Endpoints

Local Development:

- Server: `http://localhost:3000`
- API Base: `http://localhost:3000/api/v1`

Test Users:

- Admin: `admin@example.com` (password in documentation)
- Test users created via signup endpoint

Common Test Scenarios

1. **Authentication**: Signup, signin, token validation
2. **Authorization**: Role-based access control
3. **CRUD Operations**: Create, read, update, delete
4. **Pagination**: List endpoints with pagination
5. **Error Handling**: Invalid inputs, missing data, unauthorized access
6. **Rate Limiting**: Verify tier-based limits
7. **Subscriptions**: Plan changes, prorations, renewals

Deployment & Operations

Production Deployment

PM2 Process Management:

- All processes managed via PM2
- Ecosystem configuration for process definitions
- Automatic restart on failure
- Log rotation and management

Environment Variables:

- Database connection strings
- JWT secrets
- Stripe API keys
- Email configuration
- Feature flags

Monitoring

Logs:

- Application logs: `logs/application-*.log`
- Error logs: `logs/error-*.log`
- PM2 logs: `pm2 logs`

Health Checks:

- API health endpoint (if configured)
- Database connection status
- Background process status

Backup Strategy

Database Backups:

- Regular PostgreSQL backups
- Point-in-time recovery capability
- Backup retention policy

File System Backups:

- Asset files backed up separately
- Thumbnail preservation
- Backup verification

Future Roadmap

Planned Features

1. **Mobile App Development**: Native iOS/Android applications
2. **Advanced Analytics**: Enhanced reporting and insights
3. **Social Features**: User connections, following, activity feeds
4. **Event Management**: Event scheduling and RSVP system
5. **Payment Enhancements**: Apple Pay, Google Pay integration
6. **API v2**: Enhanced API with GraphQL support
7. **Internationalization**: Multi-language support
8. **Advanced Search**: Elasticsearch integration
9. **Real-Time Notifications**: Push notifications
10. **White-Label Options**: Custom branding for Enterprise

Technical Improvements

1. **Performance Optimization**: Query optimization, caching
2. **Scalability**: Horizontal scaling, load balancing
3. **Monitoring**: APM integration, performance monitoring

4. **Testing**: Comprehensive test suite
5. **Documentation**: Enhanced API documentation, tutorials

Quick Reference

Important Files

- **Main Server**: `src/server.js`
- **API Documentation**: `Documents/Public/Source/02-api-documentation/API.md`
- **Database Schema**: `Documents/Public/Source/05-database/Database_Table_Definitions.md`
- **Architecture**: `Documents/Public/Source/12-development/Architectural_Approach.md`
- **PM2 Config**: `ecosystem.config.js`

Common Tasks

Add New Endpoint:

1. Add route in `src/routes/`
2. Add controller in `src/controllers/`
3. Add validation in `src/models/validation.js`
4. Add database query in `src/database/`
5. Update API documentation in `Documents/Public/Source/02-api-documentation/API.md`
6. Update `apiController.js` changelog if needed

Add New Background Process:

1. Create process file
2. Add to `ecosystem.config.js`
3. Configure schedule
4. Add restart endpoint (optional)

Database Migration:

1. Add table creation in `src/server.js`
2. Test on development database
3. Update `Database_Table_Definitions.md`
4. Document changes

Key Constants & Configuration

Quick Reference:

- **Constants File**: `src/config/constants.js` - See Configuration & Environment Variables section for details
- **Environment Variables**: `.env` file - See Configuration & Environment Variables section for complete list
- **Key Config Objects**: `PRICING_CONFIG`, `FILE_CONFIG`, `HTTP_STATUS`, `ERROR_MESSAGES`, `SUCCESS_MESSAGES`, `BACKGROUND_PROCESS_CONFIG`
- **Critical Flags**: `ENABLE_PRICING` (default: false), `ENABLE_YEARLY_BILLING` (default: false), `EMERGENCY_DISABLE_RATE_LIMITING` (emergency override)
- **Database Config**: `src/config/database.js` - PostgreSQL connection pool

Support Resources

- **Documentation**: `Documents/Public/Source/` - Comprehensive documentation organized by category
- **API Reference**: `Documents/Public/Source/02-api-documentation/API.md` - Complete endpoint documentation
- **Architecture Docs**: `Documents/Public/Source/12-development/` - Architectural patterns and development guides
- **Database Docs**: `Documents/Public/Source/05-database/` - Schema definitions and database management
- **Development Patterns**: `Documents/Public/Source/12-development/Development_Patterns.md` - Copy-and-substitute pattern and best practices
- **File Organization**: `Documents/Public/Source/11-system-management/File_Organization_on_Backend.md` - Backend file structure guide

Common Development Patterns

Copy-and-Substitute Pattern (Recommended for Similar Features): When creating functionality similar to existing code:

1. Copy existing working files (controller, queries, routes, validation)
2. Apply systematic string substitutions (e.g., `sed -i 's/assets/moments/g'`)
3. Make minimal targeted modifications for differences
4. Test thoroughly

Benefits: Reduces bugs, faster development, maintains consistency, easier maintenance.

Example: The "moments" system was created by copying the "assets" system and applying substitutions. This pattern is documented in `Development_Patterns.md`.

Dynamic Plan ID Resolution: When working with subscription plans, always query the database for plan IDs instead of hardcoding:

```
// ✅ Good - Dynamic resolution
const planIdsResult = await pool.query(
  'SELECT id, name FROM subscription_plans WHERE name = ANY($1::text[])',
  [planNames]
);
const planIdMap = {};
planIdsResult.rows.forEach(row => { planIdMap[row.name] = row.id; });

// ❌ Bad - Hardcoded IDs
const freePlanId = 1;  // Breaks if IDs change
```

Constants Usage:

- Use `PRICING_CONFIG.DEFAULT_PLANS` for plan definitions
- Use `PRICING_CONFIG.TIERS` for tier name constants
- Import from `constants.js` rather than hardcoding values
- Database triggers may contain hardcoded plan names (acceptable - they execute at PostgreSQL level)

Conclusion

pTreks is a sophisticated, production-ready platform for outdoor activity management with comprehensive administrative capabilities. This guide provides the foundation for understanding the system architecture, both client applications (end-user and admin dashboard), API structure, database design, and development workflows. The guide includes detailed documentation for both the end-user React client application and the admin dashboard React application, covering their respective architectures, features, and development patterns.

Whether you're working on the backend API, the end-user client for outdoor enthusiasts, or the powerful admin dashboard for system management, this guide provides the essential information to get started and work effectively on the pTreks platform.

For detailed information on specific features, refer to the comprehensive documentation in the `Documents/Public/Source/` directory.

Welcome to the pTreks development team!

Last Updated: December 18, 2025