Complete Technical Documentation for Junior Engineers

Art Inventory Management Application

Table of Contents

Project Overview

This is a **multi-tenant art inventory management application** that allows users to organize their art collections through a project-based system. Think of it like a digital cataloging system where:

- **Users** can belong to multiple **Projects** (like different art collections)
- Each **Project** contains **Inventory Items** (artworks, artifacts, etc.)
- **Projects** have **Members** with different roles (owner, manager, member, viewer)
- **Items** can have photos, detailed information, and categorization

Key Business Concepts

- **Multi-tenancy**: One application serves multiple isolated "tenants" (projects)
- **Role-based Access**: Different users have different permissions
- **Collaborative Management**: Multiple people can manage the same collection
- **Photo Management**: Upload and organize photos of inventory items

---

Technology Stack Explained

Frontend Technologies

Next.js 15 (App Router)

- **What it is**: A React framework that provides file-based routing, server-side rendering, and full-stack capabilities
- **Why we use it**: Modern, performant, handles both frontend and backend in one codebase
- **File-based routing**: Files in `/app` directory become routes automatically

  - `/app/page.tsx` = Homepage (/)
  - `/app/dashboard/page.tsx` = Dashboard page (/dashboard)

React 18

- **What it is**: JavaScript library for building user interfaces
- **Key concepts for juniors**:
  - **Components**: Reusable UI pieces (like LEGO blocks)
  - **State**: Data that can change (like form inputs)
  - **Props**: Data passed between components (like function parameters)
  - **Hooks**: Functions that let you use React features (`useState`, `useEffect`)

TypeScript

- **What it is**: JavaScript with type safety
- **Why it matters**: Catches errors before runtime, better code documentation
- **Example**:

```
// JavaScript (can cause runtime errors)
const name = "John"
const age = name + 5 // "John5" - probably not intended

// TypeScript (catches error during development)
const name: string = "John"
const age: number = name + 5 // ERROR: Cannot add string to number
```

Tailwind CSS

- **What it is**: Utility-first CSS framework
- **How it works**: Instead of writing custom CSS, you use predefined classes
- **Example**:

```
<!-- Traditional CSS -->
<button class="my-button">Click me</button>
<style>.my-button { background: blue; padding: 10px; }</style>

<!-- Tailwind CSS -->
<button class="bg-blue-500 px-4 py-2">Click me</button>
```

shadcn/ui

- **What it is**: Pre-built, accessible UI components built on Radix UI
- **Why we use it**: Professional-looking components that work perfectly together
- **Examples**: Buttons, forms, modals, tooltips

Backend Technologies

Supabase

- **What it is**: Backend-as-a-Service (like Firebase, but with PostgreSQL)
- **Provides**:
  - **Database**: PostgreSQL with Row Level Security
  - **Authentication**: User login/signup

    - **Real-time**: Live data updates
    - **Storage**: File uploads
- **Why we chose it**: Full-featured, PostgreSQL (more powerful than Firebase's NoSQL)

Vercel Blob

- **What it is**: File storage service by Vercel
- **What we use it for**: Storing photos of inventory items
- **Why not Supabase Storage**: Better integration with Vercel deployment

Mailgun

- **What it is**: Email service for sending transactional emails
- **What we use it for**: Invitation emails, notifications
- **Alternative**: Previously used Resend (both are email services)

Architecture Deep Dive

Application Structure

```
inventory-app/
├── app/                        # Next.js App Router pages
|   ├── api/                   # Backend API routes
|   ├── auth/                  # Authentication pages
|   ├── dashboard/             # Main dashboard
|   ├── inventory/             # Inventory management
|   ├── projects/              # Project management
|   └── globals.css            # Global styles
├── components/                # Reusable React components
|   ├── ui/                   # Basic UI components (shadcn/ui)
|   └── *.tsx                 # Application-specific components
├── contexts/                 # React Context for global state
├── hooks/                    # Custom React hooks
├── lib/                      # Utility functions and configurations
|   ├── services/             # Business logic (email, etc.)
|   ├── supabase/             # Database client configurations
|   ├── types/                # TypeScript type definitions
|   └── utils/                # Helper functions
├── scripts/                  # Database migration scripts
└── middleware.ts             # Next.js middleware for auth
```

Multi-Tenant Architecture

What is Multi-tenancy? Imagine an apartment building:

- The building = Our application
- Each apartment = A project
- Residents = Project members
- Only residents of an apartment can access their apartment

Implementation:

1. Every database table includes a `project_id` field
2. Row Level Security (RLS) ensures users only see data from their projects
3. Application logic enforces project-based access control

State Management Strategy

Global State (React Context)

```
// contexts/ProjectContext.tsx
const ProjectContext = createContext({
  activeProject: null,      // Currently selected project
  isLoading: false,         // Loading state
  switchToProject: () => {} // Function to change projects
})
```

Local State (React useState)

```
// For component-specific data
const [formData, setFormData] = useState({
  name: '',
  description: ''
})
```

Server State (Supabase)

```
// Data from database
const { data: projects } = await supabase
  .from('projects')
  .select('*')
```

Database Design & SQL Tables

Core Tables Explained

profiles

- **Purpose**: Extends Supabase auth.users with additional user information
- **Key Fields**:
  - `id` : UUID (matches auth.users.id)
  - `email` : User's email address
  - `full_name` : Display name
  - `created_at` : When profile was created

```
CREATE TABLE profiles (
  id UUID PRIMARY KEY REFERENCES auth.users(id),
  email TEXT NOT NULL,
  full_name TEXT NOT NULL,
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

projects

- **Purpose**: Containers for inventory collections
- **Key Fields**:
  - `id` : Unique identifier
  - `name` : Project name (e.g., "Mom's Art Collection")

- `description`: Optional description
- `created_by`: User who created the project
- `created_at`, `updated_at`: Timestamps

```
CREATE TABLE projects (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name TEXT NOT NULL,
  description TEXT,
  created_by UUID REFERENCES profiles(id),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

project_members

- **Purpose**: Defines who can access which projects and their roles
- **Key Fields**:
  - `project_id`: Which project
  - `user_id`: Which user
  - `role`: owner, manager, member, viewer
  - `joined_at`: When they joined

```
CREATE TABLE project_members (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  project_id UUID REFERENCES projects(id) ON DELETE CASCADE,
  user_id UUID REFERENCES profiles(id) ON DELETE CASCADE,
  role TEXT CHECK (role IN ('owner', 'manager', 'member', 'viewer')),
  joined_at TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE(project_id, user_id)
);
```

inventory_items

- **Purpose**: The actual art/items being tracked
- **Key Fields**:
  - `project_id`: Which project owns this item
  - `name`: Item name
  - `description`: Details about the item
  - `photos`: JSON array of photo URLs
  - `estimated_value`: Monetary value
  - `category`, `area_id`: Organization fields
  - `created_by`: Who added it

```
CREATE TABLE inventory_items (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  project_id UUID REFERENCES projects(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  description TEXT,
  photos JSONB DEFAULT '[]'::jsonb,
  estimated_value DECIMAL(10,2),
  category TEXT,
  area_id UUID REFERENCES project_areas(id),
  created_by UUID REFERENCES profiles(id),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);
```

Row Level Security (RLS) Explained

What is RLS? Row Level Security is like having a bouncer at every table in your database. The bouncer checks every request and only shows rows the user is allowed to see.

Example RLS Policy:

```
-- Users can only see projects they are members of
CREATE POLICY "Users can view their projects" ON projects
  FOR SELECT USING (
    id IN (
      SELECT project_id
      FROM project_members
      WHERE user_id = auth.uid()
    )
  );
```

Why it's important:

- **Security**: Even if application code has bugs, database won't leak data
- **Simplicity**: No need to add WHERE clauses to every query
- **Trust**: Database enforces security, not application code

Database Migration System

What are migrations? Scripts that modify database structure in a controlled way.

Our Migration Files (run in sequence):

1. `001_create_inventory_schema.sql` - Basic inventory tables
2. `002_create_profiles.sql` - User profiles
3. `003_seed_sample_data.sql` - Test data
4. `004_create_projects_schema.sql` - Multi-tenant system
5. `005_migrate_existing_data.sql` - Data migration
6. `006_create_project_areas.sql` - Organization zones
7. `007_create_invitations_schema.sql` - User invitations
8. `008_create_project_categories.sql` - Categorization
9. `009_fix_missing_rls_policies.sql` - Security fixes
10. `010_remove_invitation_system.sql` - Simplified access
11. `011_fix_rls_circular_dependency.sql` - Security improvements
12. `012_create_pending_access_table.sql` - Pending user access
13. `013_fix_missing_profiles.sql` - Profile fixes
14. `014_database_security_audit.sql` - Security verification

Why sequential? Database changes must be applied in order, like building a house (foundation first, then walls, then roof).

Authentication & Security

Authentication Flow

1. User Registration/Login

```
// User signs up
const { data, error } = await supabase.auth.signUp({
  email: 'user@example.com',
  password: 'password'
})

// User logs in
const { data, error } = await supabase.auth.signInWithPassword({
  email: 'user@example.com',
  password: 'password'
})
```

2. Session Management

```
// Check current session
const { data: { session } } = await supabase.auth.getSession()
if (session?.user) {
  // User is logged in
}

// Listen for auth changes
supabase.auth.onAuthStateChange((event, session) => {
  if (event === 'SIGNED_IN') {
    // User logged in
  }
  if (event === 'SIGNED_OUT') {
    // User logged out
  }
})
```

Middleware Protection

Purpose: Protects routes before pages load

```
// middleware.ts
export async function middleware(request: NextRequest) {
  // Get current user
  const { data: { user } } = await supabase.auth.getUser()

  // Define public routes (no login required)
  const publicRoutes = ["/", "/auth/login", "/auth/sign-up"]

  // If user not logged in and trying to access protected route
  if (!user && !isPublicRoute) {
    // Redirect to login
    return NextResponse.redirect('/auth/login')
  }

  // Continue to requested page
  return NextResponse.next()
}
```

Security Layers

1. Database Level (RLS Policies)

- Every database query is automatically filtered
- Users cannot access data they don't own
- Even if application has bugs, data is safe

2. API Level (Server-side validation)

```
// app/api/projects/route.ts
export async function GET(request: Request) {
  // Verify user is authenticated
  const { data: { user }, error } = await supabase.auth.getUser(token)
  if (!user) {
    return NextResponse.json({ error: 'Unauthorized' }, { status: 401 })
  }

  // RLS automatically filters results to user's projects
  const { data: projects } = await supabase
    .from('projects')
    .select('*')

  return NextResponse.json(projects)
}
```

3. Route Level (Middleware)

- Checks authentication before page loads
- Redirects unauthenticated users
- Ensures complete profile setup

4. Component Level (UI Guards)

```
// components/auth-guard.tsx
export function AuthGuard({ children }: { children: React.ReactNode }) {
  const { user } = useAuth()

  if (!user) {
    return <LoginPrompt />
  }

  return <>{children}</>
}
```

Role-Based Access Control

Roles and Permissions:

- **Owner**: Can do everything (delete project, manage all members)
- **Manager**: Can manage inventory, invite/remove members (except owners)
- **Member**: Can add/edit inventory items
- **Viewer**: Can only view items (read-only access)

Implementation:

```
// Check user role in a project
const { data: membership } = await supabase
  .from('project_members')
  .select('role')
  .eq('project_id', projectId)
  .eq('user_id', user.id)
  .single()

if (membership?.role === 'owner') {
  // Show delete project button
}
```

API Design Patterns

RESTful API Structure

Convention:

- `GET` = Read data
- `POST` = Create new data
- `PUT` = Update existing data
- `DELETE` = Remove data

Our API Routes:

Projects API

```
GET    /api/projects           # List user's projects
POST   /api/projects           # Create new project
GET    /api/projects/[id]      # Get project details
PUT    /api/projects/[id]      # Update project
DELETE /api/projects/[id]      # Delete project
```

Project Members API

```
GET  /api/projects/[id]/members     # List project members
POST /api/projects/[id]/members     # Add member to project
PUT  /api/projects/[id]/members/[memberid]  # Update member role
DELETE /api/projects/[id]/members/[memberid] # Remove member
```

API Response Format

Success Response:

```
{
  "data": [
    {
      "id": "uuid",
      "name": "My Art Collection",
      "description": "Family art pieces",
      "created_at": "2024-01-01T00:00:00Z"
    }
  ],
  "count": 1
}
```

Error Response:

```
{
  "error": "Project not found",
  "code": "PROJECT_NOT_FOUND",
  "details": {
    "project_id": "uuid-here"
  }
}
```

Authentication Pattern

Every API route follows this pattern:

```
export async function GET(request: Request) {
  // 1. Extract auth token from header
  const token = request.headers.get('authorization')?.replace('Bearer ', '')

  // 2. Verify user is authenticated
  const { data: { user }, error } = await supabase.auth.getUser(token)
  if (error || !user) {
    return NextResponse.json({ error: 'Unauthorized' }, { status: 401 })
  }

  // 3. Use service role client for database operations
  // (RLS policies still apply based on auth.uid())
  const supabaseService = createClient(
    process.env.NEXT_PUBLIC_INVAPPSUPABASE_URL!,
    process.env.SUPABASE_SERVICE_ROLE_KEY!, // Has elevated permissions
    { auth: { persistSession: false } }
  )

  // 4. Execute database operations
  // RLS ensures user only sees their data
  const { data, error: dbError } = await supabaseService
    .from('projects')
    .select('*')

  // 5. Return response
  if (dbError) {
    return NextResponse.json({ error: dbError.message }, { status: 500 })
  }

  return NextResponse.json({ data })
}
```

File Upload Pattern

Photo Upload API (`/api/upload`):

```
export async function POST(request: Request) {
  try {
    // 1. Authenticate user
    const user = await authenticateUser(request)

    // 2. Parse multipart form data
    const formData = await request.formData()
    const file = formData.get('file') as File

    // 3. Upload to Vercel Blob
    const { url } = await put(
      `inventory/${user.id}/${file.name}`,
      file,
      { access: 'public' }
    )

    // 4. Return uploaded file URL
    return NextResponse.json({ url })
  } catch (error) {
    return NextResponse.json({ error: error.message }, { status: 500 })
  }
}
```

Frontend Architecture

Component Architecture

Three Types of Components:

1. UI Components (`/components/ui/`)

- Basic building blocks (buttons, inputs, modals)
- Come from shadcn/ui library
- Reusable across entire application
- Examples: `Button` , `Input` , `Dialog`

2. Feature Components (`/components/`)

- Application-specific functionality
- Combine multiple UI components
- Handle business logic
- Examples: `InventoryForm` , `ProjectSwitcher`

3. Page Components (`/app/`)

- Top-level pages that users visit
- Combine multiple feature components
- Handle route-specific logic
- Examples: `/app/dashboard/page.tsx` , `/app/inventory/page.tsx`

State Management Patterns

Global State (ProjectContext)

```
// contexts/ProjectContext.tsx
export function ProjectProvider({ children }: { children: ReactNode }) {
  const [activeProject, setActiveProject] = useState<Project | null>(null)
  const [isLoading, setIsLoading] = useState(true)

  const switchToProject = async (projectId: string) => {
    setIsLoading(true)
    try {
      // Fetch project data
      const project = await fetchProject(projectId)
      setActiveProject(project)
    } catch (error) {
      console.error('Failed to switch project:', error)
    } finally {
      setIsLoading(false)
    }
  }

  return (
    <ProjectContext.Provider value={{
      activeProject,
      isLoading,
      switchToProject
    }}>
      {children}
    </ProjectContext.Provider>
  )
}

// Usage in components
function MyComponent() {
  const { activeProject, switchToProject } = useProject()

  return (
    <div>
      <h1>{activeProject?.name}</h1>
      <button onClick={() => switchToProject('new-id')}>
        Switch Project
      </button>
    </div>
  )
}
```

Local Component State

```
function InventoryForm() {
  // Form data state
  const [formData, setFormData] = useState({
    name: '',
    description: '',
    estimatedValue: ''
  })

  // Loading state
  const [isSubmitting, setIsSubmitting] = useState(false)

  // Handle form submission
  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault()
    setIsSubmitting(true)

    try {
      await createInventoryItem(formData)
      // Reset form
      setFormData({ name: '', description: '', estimatedValue: '' })
    } catch (error) {
      console.error('Failed to create item:', error)
    } finally {
      setIsSubmitting(false)
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
```

```
        value={formData.name}
        onChange={e => setFormData(prev => ({
          ...prev,
          name: e.target.value
        }))}
        placeholder="Item name"
      />
      <button type="submit" disabled={isSubmitting}>
        {isSubmitting ? 'Saving...' : 'Save Item'}
      </button>
    </form>
  )
}
```

Data Fetching Patterns

Server Components (for initial data)

```
// app/dashboard/page.tsx
export default async function DashboardPage() {
  // This runs on the server
  const supabase = createServerClient(/* ... */)
  const { data: projects } = await supabase
    .from('projects')
    .select('*')

  return (
    <div>
      <h1>Dashboard</h1>
      <ProjectsList projects={projects} />
    </div>
  )
}
```

Client Components (for interactive data)

```
// components/inventory-list.tsx
'use client'
function InventoryList() {
  const [items, setItems] = useState<InventoryItem[]>([])
  const [loading, setLoading] = useState(true)

  useEffect(() => {
    async function loadItems() {
      try {
        const response = await fetch('/api/inventory')
        const data = await response.json()
        setItems(data)
      } catch (error) {
        console.error('Failed to load items:', error)
      } finally {
        setLoading(false)
      }
    }

    loadItems()
  }, [])

  if (loading) {
    return <div>Loading...</div>
  }
  return (
    <div>
      {items.map(item => (
        <div key={item.id}>{item.name}</div>
      ))}
    </div>
  )
}
```

Form Handling with React Hook Form

Why React Hook Form?

- Better performance (fewer re-renders)
- Built-in validation
- Easy to use with TypeScript

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

// Define validation schema
const formSchema = z.object({
  name: z.string().min(1, 'Name is required'),
  description: z.string().optional(),
  estimatedValue: z.number().min(0, 'Value must be positive')
})

function InventoryForm() {
  const form = useForm<z.infer<typeof formSchema>>({
    resolver: zodResolver(formSchema),
    defaultValues: {
      name: '',
      description: '',
      estimatedValue: 0
    }
  })

  const onSubmit = async (values: z.infer<typeof formSchema>) => {
    try {
      await createInventoryItem(values)
      form.reset() // Clear form
    } catch (error) {
      // Handle error
    }
  }

  return (
    <form onSubmit={form.handleSubmit(onSubmit)}>
      <input
        {...form.register('name')}
        placeholder="Item name"
      />
      {form.formState.errors.name && (
        <span>{form.formState.errors.name.message}</span>
      )}

      <button type="submit" disabled={form.formState.isSubmitting}>
        Submit
      </button>
    </form>
  )
}
```

Development Workflow

Getting Started

1. Environment Setup

```
# Clone repository
git clone [repository-url]
cd inventory-app

# Install dependencies
npm install

# Set up environment variables
cp .env.example .env.local
# Edit .env.local with your Supabase credentials
```

2. Required Environment Variables

```
# .env.local
NEXT_PUBLIC_INVAPPSUPABASE_URL=your_supabase_url
NEXT_PUBLIC_INVAPPSUPABASE_ANON_KEY=your_supabase_anon_key
NEXT_PUBLIC_APP_URL=http://localhost:3000
RESEND_API_KEY=your_resend_key
BLOB_READ_WRITE_TOKEN=your_vercel_blob_token
```

3. Database Setup

```
# Run migration scripts in Supabase SQL editor
# Execute them in order: 001, 002, 003, etc.

# Or use the helper script
node execute_migration.js
```

4. Start Development

```
npm run dev    # Start development server
npm run build  # Build for production
npm run lint   # Run code linting
```

Development Commands Explained

```
# Development server with hot reload
npm run dev

# Production build (checks for errors)
npm run build

# Code quality checks
npm run lint
```

```
# Type checking (without building)
npx tsc --noEmit
```

Project Structure Best Practices

File Naming Conventions:

- Components: `PascalCase` (`InventoryForm.tsx`)
- Pages: `lowercase` (`page.tsx` , `layout.tsx`)
- Utilities: `camelCase` (`formatDate.ts`)
- Constants: `UPPER_SNAKE_CASE` (`API_ENDPOINTS.ts`)

Import Organization:

```
// 1. External libraries
import React from 'react'
import { NextResponse } from 'next/server'

// 2. Internal modules (absolute imports)
import { createClient } from '@/lib/supabase/client'
import { ProjectContext } from '@/contexts/ProjectContext'

// 3. Relative imports
import './styles.css'
```

Git Workflow

Branch Naming:

- `feature/inventory-form-validation`
- `fix/auth-redirect-loop`
- `refactor/database-queries`

Commit Messages:

```
# Good commit messages
git commit -m "Add inventory item validation"
git commit -m "Fix auth redirect loop on logout"
git commit -m "Refactor project context for better performance"

# Bad commit messages
git commit -m "fix stuff"
git commit -m "update"
git commit -m "changes"
```

Deployment & Environment Setup

Vercel Deployment

Why Vercel?

- Built by the Next.js team
- Automatic deployments from Git
- Built-in CDN and edge functions

- Easy environment variable management

Deployment Process:

1. Push code to GitHub
2. Vercel automatically builds and deploys
3. Environment variables set in Vercel dashboard
4. Automatic HTTPS and global CDN

Environment Configuration

Development vs Production:

Development (`.env.local`):

```
NEXT_PUBLIC_APP_URL=http://localhost:3000
# Use development Supabase project
NEXT_PUBLIC_INVAPPSUPABASE_URL=https://dev-project.supabase.co
```

Production (Vercel Environment Variables):

```
NEXT_PUBLIC_APP_URL=https://your-domain.vercel.app
# Use production Supabase project
NEXT_PUBLIC_INVAPPSUPABASE_URL=https://prod-project.supabase.co
```

Build Process

What happens during build:

1. TypeScript compilation (checks for type errors)
2. ESLint runs (checks code quality)
3. Next.js optimization (bundles, minifies, optimizes)
4. Static pages pre-rendered (for better performance)

Build Output:

```
.next/
├── static/          # Static assets (images, CSS, JS)
├── server/          # Server-side code
└── cache/          # Build cache for faster rebuilds
```

Performance Optimizations

Automatic Optimizations:

- **Code Splitting**: Each page loads only required JavaScript
- **Image Optimization**: Next.js optimizes images automatically
- **Static Generation**: Pages that don't change are pre-built
- **Tree Shaking**: Unused code is removed from bundles

Manual Optimizations:

```
// Lazy loading components
const HeavyComponent = lazy(() => import('./HeavyComponent'))

// Dynamic imports for large libraries
const loadLibrary = async () => {
  const { heavyFunction } = await import('heavy-library')
  return heavyFunction()
}

// Memoization for expensive calculations
const expensiveValue = useMemo(() => {
  return computeExpensiveValue(data)
}, [data])
```

Common Patterns & Best Practices

Error Handling Patterns

API Error Handling:

```
// Good error handling
async function createProject(data: ProjectData) {
  try {
    const response = await fetch('/api/projects', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    })

    if (!response.ok) {
      // Handle HTTP errors
      const error = await response.json()
      throw new Error(error.message || 'Failed to create project')
    }

    const result = await response.json()
    return result
  } catch (error) {
    // Log error for debugging
    console.error('Project creation failed:', error)

    // Re-throw with user-friendly message
    throw new Error('Unable to create project. Please try again.')
  }
}

// Usage in component
function CreateProjectForm() {
  const [error, setError] = useState<string | null>(null)

  const handleSubmit = async (data: ProjectData) => {
    setError(null)

    try {
      await createProject(data)
      // Success - redirect or show success message
    } catch (error) {
      setError(error.message)
    }
  }
  return (
    <form onSubmit={handleSubmit}>
      {error && (
        <div className="error-message">
          {error}
        </div>
      )}
      {/* form fields */}
    </form>
  )
}
```

Database Error Handling:

```
// API route error handling
export async function POST(request: Request) {
  try {
    const user = await authenticateUser(request)
    const data = await request.json()

    const { data: project, error } = await supabase
      .from('projects')
      .insert([{ ...data, created_by: user.id }])
      .select()
      .single()

    if (error) {
      // Log detailed error for debugging
      console.error('Database error:', error)

      // Return user-friendly error
      if (error.code === '23505') { // Duplicate key
        return NextResponse.json(
          { error: 'A project with this name already exists' },
          { status: 400 }
        )
      }

      return NextResponse.json(
        { error: 'Failed to create project' },
        { status: 500 }
      )
    }

    return NextResponse.json({ data: project })
  } catch (error) {
    console.error('Unexpected error:', error)
    return NextResponse.json(
      { error: 'Internal server error' },
      { status: 500 }
    )
  }
}
```

Loading States Pattern

Component Loading States:

```
function InventoryList() {
  const [items, setItems] = useState<InventoryItem[]>([])
  const [loading, setLoading] = useState(true)
  const [error, setError] = useState<string | null>(null)

  useEffect(() => {
    async function loadItems() {
      try {
        setLoading(true)
        setError(null)

        const response = await fetch('/api/inventory')
        if (!response.ok) throw new Error('Failed to load items')

        const data = await response.json()
        setItems(data)
      } catch (err) {
        setError(err.message)
      } finally {
        setLoading(false)
      }
    }

    loadItems()
  }, [])

  // Loading state
  if (loading) {
    return (
      <div className="flex items-center justify-center p-8">
        <div className="animate-spin rounded-full h-8 w-8 border-b-2 border-blue-500"></div>
        <span className="ml-2">Loading inventory...</span>
      </div>
    )
  }

  // Error state
  if (error) {
    return (
      <div className="bg-red-50 border border-red-200 rounded-md p-4">
        <p className="text-red-800">Error: {error}</p>
        <button
          onClick={() => window.location.reload()}
          className="mt-2 text-red-600 underline"
        >
          Try again
        </button>
      </div>
    )
  }

  // Empty state
  if (items.length === 0) {
    return (
      <div className="text-center p-8">
        <p className="text-gray-500">No inventory items found.</p>
        <button className="mt-4 bg-blue-500 text-white px-4 py-2 rounded">
          Add First Item
        </button>
      </div>
    )
  }

  // Success state
  return (
    <div>
```

```
      {items.map(item => (
        <div key={item.id}>{item.name}</div>
      ))}
    </div>
  )
}
```

Form Validation Patterns

Client-Side Validation (immediate feedback):

```
import * as z from 'zod'

const inventoryItemSchema = z.object({
  name: z.string()
    .min(1, 'Name is required')
    .max(100, 'Name too long'),
  description: z.string()
    .max(500, 'Description too long')
    .optional(),
  estimatedValue: z.number()
    .min(0, 'Value cannot be negative')
    .optional(),
  category: z.string()
    .min(1, 'Please select a category')
})

function InventoryForm() {
  const form = useForm({
    resolver: zodResolver(inventoryItemSchema)
  })

  return (
    <form onSubmit={form.handleSubmit(onSubmit)}>
      <input {...form.register('name')} />
      {form.formState.errors.name && (
        <span className="text-red-500">
          {form.formState.errors.name.message}
        </span>
      )}
    </form>
  )
}
```

Server-Side Validation (security):

```
// API route
export async function POST(request: Request) {
  try {
    const data = await request.json()

    // Validate data on server (never trust client)
    const validatedData = inventoryItemSchema.parse(data)

    // Proceed with validated data
    const { data: item, error } = await supabase
      .from('inventory_items')
      .insert([validatedData])
      .select()
      .single()
    return NextResponse.json({ data: item })
  } catch (error) {
    if (error instanceof z.ZodError) {
      // Validation error
      return NextResponse.json(
        { error: 'Invalid data', details: error.errors },
        { status: 400 }
      )
    }

    // Other errors
    return NextResponse.json(
      { error: 'Internal server error' },
      { status: 500 }
    )
  }
}
```

Data Fetching Optimization

Prevent Unnecessary Requests:

```
function ProjectSwitcher() {
  const [projects, setProjects] = useState<Project[]>([])
  const [loading, setLoading] = useState(false)
  const [lastFetch, setLastFetch] = useState<number>(0)

  const fetchProjects = useCallback(async () => {
    // Don't fetch if recently fetched (cache for 5 minutes)
    const fiveMinutesAgo = Date.now() - 5 * 60 * 1000
    if (lastFetch > fiveMinutesAgo && projects.length > 0) {
      return
    }

    setLoading(true)
    try {
      const response = await fetch('/api/projects')
      const data = await response.json()
      setProjects(data)
      setLastFetch(Date.now())
    } catch (error) {
      console.error('Failed to fetch projects:', error)
    } finally {
      setLoading(false)
    }
  }, [lastFetch, projects.length])

  // Fetch on mount
  useEffect(() => {
    fetchProjects()
  }, [fetchProjects])

  return (
    <div>
      {loading ? (
        <div>Loading...</div>
      ) : (
        projects.map(project => (
          <div key={project.id}>{project.name}</div>
        ))
      )}
    </div>
  )
}
```

Security Best Practices

Input Sanitization:

```
// Sanitize user input
function sanitizeInput(input: string): string {
  return input
    .trim() // Remove whitespace
    .replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi, '') // Remove scripts
    .substring(0, 1000) // Limit length
}

// Use in API routes
export async function POST(request: Request) {
  const data = await request.json()

  const sanitizedData = {
    name: sanitizeInput(data.name),
    description: sanitizeInput(data.description)
  }

  // Proceed with sanitized data
}
```

Authentication Checks:

```
// Reusable auth check function
async function authenticateUser(request: Request) {
  const token = request.headers.get('authorization')?.replace('Bearer ', '')

  if (!token) {
    throw new Error('No authentication token provided')
  }

  const { data: { user }, error } = await supabase.auth.getUser(token)

  if (error || !user) {
    throw new Error('Invalid authentication token')
  }

  return user
}

// Use in every API route
export async function GET(request: Request) {
  try {
    const user = await authenticateUser(request)
    // Proceed with authenticated user
  } catch (error) {
    return NextResponse.json(
      { error: error.message },
      { status: 401 }
    )
  }
}
```

Troubleshooting Guide

Common Errors and Solutions

"Unauthorized" errors

- **Symptom**: API calls return 401 Unauthorized
- **Causes**:
  1. User not logged in
  2. Session expired
  3. Wrong API endpoint
  4. Missing Authorization header
- **Solutions**:

```
// Check if user is logged in
const { data: { session } } = await supabase.auth.getSession()
console.log('Session:', session)

// Refresh session if expired
const { data, error } = await supabase.auth.refreshSession()

// Ensure API calls include auth header
fetch('/api/endpoint', {
  headers: {
    'Authorization': `Bearer ${session.access_token}`
  }
})
```

Database RLS Policy Violations

- **Symptom**: Database queries return empty results or "insufficient privileges" error
- **Causes**:
  1. User doesn't have permission to access data
  2. RLS policy is too restrictive
  3. Missing project membership
- **Solutions**:

```
-- Check user's project memberships
SELECT * FROM project_members WHERE user_id = auth.uid();

-- Check if RLS is enabled
SELECT schemaname, tablename, rowsecurity
FROM pg_tables
WHERE tablename = 'your_table_name';

-- View existing policies
SELECT * FROM pg_policies WHERE tablename = 'your_table_name';
```

"Project not found" errors

- **Symptom**: Cannot access or switch to projects
- **Causes**:
  1. User is not a member of the project

   2. Project was deleted
   3. Database RLS blocking access
- **Solutions**:

```
// Check project membership
const { data: membership } = await supabase
  .from('project_members')
  .select('*')
  .eq('project_id', projectId)
  .eq('user_id', user.id)

console.log('Membership:', membership)
```

Photo upload failures

- **Symptom**: Images don't upload or display
- **Causes**:
   1. Missing BLOB_READ_WRITE_TOKEN
   2. File size too large
   3. Invalid file format
   4. Network connectivity issues
- **Solutions**:

```
// Check environment variable
console.log('Blob token:', process.env.BLOB_READ_WRITE_TOKEN ? 'Set' : 'Missing')

// Validate file before upload
if (file.size > 10 * 1024 * 1024) { // 10MB limit
  throw new Error('File too large')
}

if (!file.type.startsWith('image/')) {
  throw new Error('File must be an image')
}
```

Build failures

- **Symptom**: `npm run build` fails
- **Common causes and fixes**:

```
# TypeScript errors
npx tsc --noEmit  # Check types without building

# Missing environment variables
# Ensure all required env vars are set in .env.local

# ESLint errors
npm run lint      # Check and fix linting issues
npm run lint -- --fix  # Auto-fix some issues
```

Email sending failures

- **Symptom**: Invitation emails not sent
- **Causes**:

1. Missing or invalid RESEND_API_KEY
2. Email service rate limits
3. Invalid email addresses

- **Solutions**:

```
// Check API key
console.log('Resend key:', process.env.RESEND_API_KEY ? 'Set' : 'Missing')

// Test email sending
const testEmail = await resend.emails.send({
  from: 'test@yourdomain.com',
  to: 'test@example.com',
  subject: 'Test',
  html: '<p>Test email</p>'
})

console.log('Email result:', testEmail)
```

Debugging Tips

1. Check Browser Console

- Open Developer Tools (F12)
- Look for JavaScript errors in Console tab
- Check Network tab for failed API requests

2. Check Server Logs

- In development: logs appear in terminal
- In production: check Vercel logs

```
# Install Vercel CLI
npm i -g vercel

# View production logs
vercel logs [deployment-url]
```

3. Database Debugging

- Use Supabase dashboard SQL editor
- Check table contents and relationships
- Test RLS policies with different users

4. Common Debug Patterns

```
// Log user authentication state
const { data: { session } } = await supabase.auth.getSession()
console.log('Auth state:', {
  isLoggedIn: !!session?.user,
  userId: session?.user?.id,
  email: session?.user?.email
})

// Log API responses
const response = await fetch('/api/endpoint')
const data = await response.json()
```

```
console.log('API Response:', {
  status: response.status,
  ok: response.ok,
  data
})

// Log database queries
const { data, error, count } = await supabase
  .from('table_name')
  .select('*', { count: 'exact' })

console.log('Database result:', {
  rowCount: count,
  hasError: !!error,
  error,
  sampleData: data?.[0]
})
```

Performance Issues

Slow page loads

- **Check bundle size**: `npm run build` shows bundle analysis
- **Optimize images**: Use Next.js Image component
- **Reduce database queries**: Combine related data in single query

Memory leaks

- **Clean up event listeners**:

  ```
  useEffect(() => {
    const subscription = supabase.auth.onAuthStateChange(handler)
    return () => subscription.unsubscribe() // Cleanup
  }, [])
  ```

- **Cancel fetch requests**:

  ```
  useEffect(() => {
    const controller = new AbortController()

    fetch('/api/data', { signal: controller.signal })
      .then(handleData)
      .catch(error => {
        if (error.name !== 'AbortError') {
          console.error(error)
        }
      })

    return () => controller.abort() // Cancel request
  }, [])
  ```

Conclusion

This documentation covers the essential concepts and patterns needed to understand and work with the Art Inventory Management Application. As a junior engineer, focus on:

1. **Understanding the architecture** - How components fit together

2. **Following patterns** - Use established patterns for consistency
3. **Security first** - Always authenticate and validate
4. **Error handling** - Expect things to go wrong and handle gracefully
5. **Testing thoroughly** - Test both happy and error paths

Remember: It's better to ask questions and understand the code than to make assumptions. This codebase follows modern best practices, and understanding it well will serve you in future projects.

Additional Resources

- Next.js Documentation
- React Documentation
- Supabase Documentation
- TypeScript Handbook
- Tailwind CSS Documentation
- shadcn/ui Components