AutoCRM Development Guide

Overview

This guide will help you build a CRM application focused on ticket management. It's designed for developers using an AI-first approach with Cursor and Claude.

Table of Contents

Understanding Core Features

Required Ticket Data Model Features

1. Standard Identifiers & Timestamps

What: Every ticket needs unique IDs and time tracking **Why**: Critical for:

- Unique reference for each ticket
- Tracking issue lifetime
- Measuring response times
- Auditing history

Current Status: Implemented in Supabase types:

typescript tickets: { Row: { id: string created_at: string updated_at: string // ...other fields } }

2. Dynamic Status Tracking

What: Current state of a ticket **States**:

- Open: New ticket
- In Progress: Being worked on
- Resolved: Solution provided
- Closed: Confirmed fixed

Why:

- Shows which tickets need attention
- Tracks progress
- Measures team performance
- Helps workload management

Current Status: Implemented with color coding:

typescript const getStatusBadgeColor = (status: string | null) => { switch (status) { case "open": return "bg-blue-500"; case "in_progress": return "bg-yellow-500"; case "resolved": return "bg-green-500"; case "closed": return "bg-gray-500"; default: return "bg-gray-500"; } };

3. Priority Levels

What: Urgency/importance indicator **Levels**:

- Low: Non-urgent issues
- Medium: Standard issues
- High: Urgent issues
- Critical: Emergency issues

Why:

- Helps teams prioritize work
- Sets customer expectations
- Enables SLA tracking
- Identifies systemic issues

Current Status: Field exists in types but needs UI implementation

4. Custom Fields

What: Additional fields for business-specific data **Examples**:

- Product category
- Department
- Customer segment
- Issue type

Why:

- Adapts CRM to business needs
- Enables detailed reporting
- Improves ticket routing
- Facilitates knowledge base organization

Current Status: Not implemented

5. Tags

What: Labels for categorization **Examples**:

- Bug
- Feature Request
- Billing
- Technical

Why:

- Easy categorization
- Pattern identification
- Reporting capabilities
- Knowledge base organization

Current Status: Not implemented

6. Internal Notes

What: Staff-only comments **Features**:

- Private discussions
- Work notes
- Internal updates
- Team collaboration

Why:

- Team collaboration
- Knowledge sharing
- Training material
- Audit trail

Current Status: Not implemented

7. Full Conversation History

What: Complete communication record **Includes**:

- Customer messages
- Staff responses
- Status changes
- Internal notes

Why:

- Context preservation
- Duplicate prevention
- Training material
- Quality assurance

Current Status: Basic implementation

Top 5 Additional Priority Features

1. Queue Management - Customizable Views

What: Different ticket organization views **Examples**:

- By status
- By priority

- By age
- By agent

Why:

- Efficiency improvement
- Focus management
- Workload balancing
- Performance tracking

Current Status: Basic dashboard implementation

2. Customer Portal - Ticket Tracking

What: Customer self-service interface **Features**:

- View tickets
- Update tickets
- Check status
- View responses

Why:

- Reduces support load
- Customer empowerment
- 24/7 access
- Transparency

Current Status: Not implemented

3. Team Management - Agent Assignment

What: Ticket ownership system **Features**:

- Assign tickets
- Transfer ownership
- Track workload
- Set availability

Why:

- Clear ownership
- Workload management
- Accountability
- Performance tracking

Current Status: Basic structure exists (assigned_to field)

4. Quick Responses - Templates

What: Pre-written response library **Features**:

- Common responses
- Customizable templates
- Rich text formatting
- Variable insertion

Why:

- Faster responses
- Consistency
- Quality control
- Training aid

Current Status: Not implemented

5. Performance Tools - Metrics Tracking

What: Analytics dashboard **Metrics**:

- Response times
- Resolution rates
- Customer satisfaction
- Agent performance

Why:

- Performance monitoring
- Process improvement
- Resource planning
- Quality control

Current Status: Not implemented

Implementation Guide

Implementing Custom Fields

Step 1: Database Setup

1. Open Cursor
2. Ask Claude:

Help me add custom fields to my ticket type in Supabase. I need: product_category (enum) department (enum) issue_type (enum)

Step 2: UI Components

1. Show CreateTicketDialog.tsx to Claude:

Help me add dropdown selectors for: Product category Department Issue type Include validation and error handling

Step 3: Display Updates

1. Show Dashboard.tsx to Claude:

Help me display the new custom fields in ticket cards with: Appropriate icons Color coding Filtering options

Implementing Tags

Step 1: Database Setup

1. Ask Claude:

Help me create: tags table ticket_tags junction table Include indexes and foreign keys

Step 2: Tag Selection UI

1. Show CreateTicketDialog.tsx to Claude:

Help me add: Multi-select tag input Tag creation Tag suggestions Include validation

Step 3: Tag Display

1. Show Dashboard.tsx to Claude:

Help me: Display tags on ticket cards Add tag-based filtering Implement tag colors

[Continue with remaining features...]

API-First Design

Understanding API-First

Think of building a CRM like building a restaurant:

Traditional Approach:

1. Build dining room (UI)
2. Design kitchen layout (backend)
3. Create menu (API)

API-First Approach:

1. Design menu (API)
2. Build kitchen (backend)
3. Create dining room (UI)

Benefits

1. **Flexibility**: Multiple frontends can use same API
2. **Future-Proofing**: Easier to add features
3. **Integration**: Simple third-party connections
4. **Testing**: Clear contract for testing

5. **Documentation**: API-first forces good documentation

Current Setup Analysis

Your Supabase implementation is partially API-first:

typescript // Good: Using typed API calls const { data, error } = await supabase .from("tickets") .select("") .order("created_at", { ascending: false });

Recommended API Structure

src/ api/ tickets/ types.ts # Type definitions queries.ts # Database queries mutations.ts # Database updates index.ts # Public API users/ types.ts queries.ts mutations.ts index.ts tags/ types.ts queries.ts mutations.ts index.ts

Implementation Steps

1. Create API Structure

bash mkdir -p src/api/{tickets,users,tags} touch src/api/{tickets,users,tags}/{types,queries,mutations,index}.ts

2. Define Types

typescript // src/api/tickets/types.ts export interface Ticket { id: string; title: string; description: string; status: TicketStatus; priority: TicketPriority; // ... other fields }

3. Create Queries

typescript // src/api/tickets/queries.ts export const getTickets = async () => { const { data, error } = await supabase .from("tickets") .select("") .order("created_at", { ascending: false }); if (error) throw error; return data; };

4. Create Mutations

typescript // src/api/tickets/mutations.ts export const createTicket = async (ticket: CreateTicketInput) => { const { data, error } = await supabase .from("tickets") .insert(ticket) .select(); if (error) throw error; return data[0]; };

Next Steps

1. Review existing code
2. Set up API structure
3. Implement custom fields
4. Add tags system
5. Build internal notes
6. Create conversation history
7. Develop queue management
8. Build customer portal
9. Implement team management
10. Add quick responses

11. Create performance tools

Remember:

- Test each feature thoroughly
- Document API changes
- Keep UI consistent
- Consider error handling
- Think about scalability

Need help with any specific section? Ask Claude through Cursor!