

BellsNotice - Code Architecture & File Roles

Table of Contents

1. Architecture Overview
 2. Directory Structure
 3. Core Files & Their Roles
 4. Component Architecture
 5. Data Flow
 6. State Management
 7. Authentication Flow
 8. Key Algorithms & Logic
-

Architecture Overview

BellsNotice follows a **modern full-stack architecture** using Next.js 16 App Router with:

- **Server Components** by default for performance
- **Client Components** marked with “use client” for interactivity
- **Supabase** as backend-as-a-service (BaaS)
- **TypeScript** for type safety
- **Component-based architecture** with React
- **Progressive Web App (PWA)** capabilities

Architecture Principles

1. **Separation of Concerns:** Clear separation between UI, logic, and data layers
 2. **Component Reusability:** DRY principle with reusable components
 3. **Type Safety:** TypeScript interfaces for all data structures
 4. **Performance:** Server-side rendering and code splitting
 5. **Scalability:** Modular design for easy feature additions
-

Directory Structure

bellsnotice/

```
app/                      # NEXT.JS APP ROUTER
    auth/                  # Authentication routes
        sign-up/
            loading.tsx      # Loading state for signup
            page.tsx         # Registration form
        sign-up-success/
            page.tsx         # Signup success page

    dashboard/              # User dashboard
```

```

loading.tsx          # Loading state
page.tsx            # Main dashboard (Profile, Comments, Notices, Requests, Analytic)

filter/
  loading.tsx      # Notice filtering page
  page.tsx         # Loading state
                    # Advanced filtering UI

notice/
  [id]/
    edit/
      page.tsx      # Notice management
      page.tsx      # Dynamic notice routes
    # Edit existing notice
    # Edit form
    page.tsx        # Notice detail view
  create/
    loading.tsx    # Create new notice
    page.tsx       # Loading state
                    # Create notice form

notices/
  loading.tsx      # All notices listing
  page.tsx         # Loading state
                    # Paginated notice list

profile/
  [id]/
    page.tsx       # User profiles
    # Dynamic profile routes
    # Public profile view

request-notice/
  page.tsx         # Notice request submission
                    # Request form

saved/
  loading.tsx     # Saved notices
  page.tsx        # Loading state
                    # User's saved notices

search/
  loading.tsx     # Search functionality
  page.tsx        # Loading state
                    # Search results page

admin/
  loading.tsx     # Admin panel
  page.tsx        # Loading state
                    # Admin dashboard

globals.css          # Global styles and Tailwind directives
layout.tsx           # Root layout with providers
page.tsx             # Home page (Login/Home)

components/          # REACT COMPONENTS

ui/                 # shadcn/ui COMPONENT LIBRARY

```

```

avatar.tsx          # User avatar display
button.tsx          # Button component
card.tsx            # Card container
dialog.tsx          # Modal dialogs
input.tsx           # Form inputs
label.tsx           # Form labels
select.tsx          # Dropdown selects
sheet.tsx           # Slide-over panels
switch.tsx          # Toggle switches
tabs.tsx            # Tabbed interfaces
textarea.tsx         # Multi-line text input
alert-dialog.tsx    # Confirmation dialogs
toast.tsx           # Toast notifications
hover-card.tsx      # Hover tooltips
tooltip.tsx          # Tooltip popovers
toggle-group.tsx    # Toggle button groups
command.tsx          # Command palette
sidebar.tsx          # Sidebar navigation
chart.tsx           # Chart components
field.tsx            # Form field wrapper
accordion.tsx        # Collapsible sections
alert.tsx            # Alert banners
aspect-ratio.tsx     # Aspect ratio container
checkbox.tsx          # Checkbox input
collapsible.tsx       # Collapsible content
context-menu.tsx      # Right-click menus
dropdown-menu.tsx    # Dropdown menus
menubar.tsx           # Menu bar
navigation-menu.tsx  # Navigation menus
popover.tsx           # Popover content
progress.tsx          # Progress indicators
radio-group.tsx        # Radio button groups
scroll-area.tsx       # Custom scrollbars
separator.tsx         // Visual separators
slider.tsx            // Range sliders

pages/               # PAGE-SPECIFIC COMPONENTS
home-page.tsx        # Home page logic (notices, carousels)
login-page.tsx        # Login form and authentication
search-page-client.tsx # Search functionality client

navigation/           # NAVIGATION COMPONENTS
navigation-menu.tsx  # Main navigation menu

notice-card.tsx       # Notice card display component
notice-create-modal.tsx # Quick notice creation modal

```

```

admin-notice-manager.tsx # Admin notice management table
top-navbar.tsx          # Top navigation bar
theme-provider.tsx      # Theme context provider
RegisterSW.tsx          # Service worker registration

lib/                      # UTILITIES & HELPERS

  supabase/              # SUPABASE CLIENTS
    client.ts             # Browser client (client components)
    server.ts             # Server client (server components)

  utils.ts                # Helper functions (cn, etc.)

hooks/                  # CUSTOM REACT HOOKS
  use-toast.ts           # Toast notification hook
  use-mobile.ts          # Mobile device detection hook

public/                 # STATIC ASSETS
  images/                # Images and icons
    bells-20notice-20icon.jpg
  manifest.json          # PWA manifest
  favicon.ico            # Favicon

styles/                 # ADDITIONAL STYLES

  .gitignore              # Git ignore patterns
  components.json          # shadcn/ui configuration
  package.json              # Project dependencies
  package-lock.json         # Dependency lock file
  pnpm-lock.yaml            # PNPM lock file
  tsconfig.json             # TypeScript configuration
  next.config.mjs           # Next.js configuration
  postcss.config.mjs        # PostCSS configuration
  proxy.ts                  # Development proxy
  PROJECT_DOCUMENTATION.md # Comprehensive project docs
  README.md                 # GitHub README
  CODE_ARCHITECTURE.md      # This file

```

Core Files & Their Roles

1. Application Entry Points

app/layout.tsx **Role:** Root layout component that wraps the entire application

Responsibilities: - Defines HTML structure and metadata - Loads fonts (Geist, Geist Mono) - Registers Service Worker for PWA - Provides Analytics tracking - Applies global CSS styles

Key Code:

```
export const metadata: Metadata = {
  title: "Bells Notice - Stay Connected",
  description: "A modern notice board for staying informed",
  // PWA configuration
  manifest: "/manifest.json",
  themeColor: "#0f172a",
  icons: { /* ... */ }
}

export default function RootLayout({ children }: { children: React.ReactNode }) {
  return (
    <html lang="en">
      <body className={`${geist.className} antialiased`}>
        <RegisterSW /> {/* PWA Service Worker */}
        {children}
        <Analytics /> {/* Vercel Analytics */}
      </body>
    </html>
  )
}
```

app/page.tsx Role: Main home page that handles authentication routing

Responsibilities: - Checks user authentication status - Renders either login page or home page based on auth state - Manages top navbar and navigation menu visibility - Listens to auth state changes

Key Logic:

```
// Check user authentication
const { data: { user } } = await supabase.auth.getUser()
setUser(user)

// Listen to auth state changes
supabase.auth.onAuthStateChange((event, session) => {
  setUser(session?.user || null)
})

// Conditional rendering
{user ? <HomePage user={user} /> : <LoginPage />}
```

2. Authentication Pages

`components/pages/login-page.tsx` **Role:** Login form with dual authentication (user and admin)

Responsibilities: - User login via email or matric number - Admin login via password - Matric number lookup for email retrieval - Error handling and loading states - Redirect after successful login

Key Functions:

```
// Handle user login
const handleLogin = async () => {
  // Check if credential is matric number
  if (credential.includes("/")) {
    // Fetch email from profiles table
    const { data: profiles } = await supabase
      .from("profiles")
      .select("email")
      .eq("matric_number", credential)
      .single()
    email = profiles.email
  }
  // Sign in with email and password
  await supabase.auth.signInWithEmailAndPassword({ email, password })
}

// Handle admin login
const handleAdminLogin = () => {
  if (adminPassword === "adminnotice") {
    sessionStorage.setItem("adminAuthenticated", "true")
    router.push("/admin")
  }
}
```

`app/auth/sign-up/page.tsx` **Role:** User registration form

Responsibilities: - Collect user information (email, name, academic details) - Dynamic department selection based on college - Account type selection (user vs rep) - Create Supabase auth user - Create profile record with academic information

Key Logic: - Multi-step form with validation - Dynamic department list based on college selection - Supabase auth signup followed by profile creation

3. Dashboard Pages

`app/dashboard/page.tsx` **Role:** User dashboard with multiple tabs

Responsibilities: - Profile management (view and edit) - Comment history - Notice management (for reps) - Notice request handling (for reps) - Analytics view (for reps) - Read receipt settings

Major Features: - **Profile Tab:** Display and edit user profile, upload avatar, manage academic info - **Comments Tab:** View and delete own comments - **Notices Tab** (Reps only): Create, edit, delete own notices - **Requests Tab** (Reps only): View pending/processed requests, accept/decline - **Analytics Tab** (Reps only): View notice statistics

Key Functions:

```
// Fetch user data
const { data: profile } = await supabase
  .from("profiles")
  .select("*")
  .eq("id", user.id)
  .single()

// Handle notice request acceptance
const handleAcceptRequest = async () => {
  // Fetch media from request
  const { data: requestMediaFiles } = await supabase
    .from("notice_request_media")
    .select("*")
    .eq("request_id", selectedRequest.id)

  // Create notice
  const { data: newNotice } = await supabase
    .from("notices")
    .insert({ title, description, author_id: user.id })
    .select()
    .single()

  // Transfer media to notice
  for (const media of requestMediaFiles) {
    await supabase
      .from("notice_media")
      .insert({
        notice_id: newNotice.id,
        media_type: media.media_type,
        media_url: media.media_url
      })
  }

  // Update request status
  await supabase
```

```

    .from("notice_requests")
    .update({
      status: "approved",
      response_message: responseMessage,
      responded_at: new Date().toISOString(),
      notice_id: newNotice.id
    })
    .eq("id", selectedRequest.id)
}

```

4. Notice Pages

`app/notice/create/page.tsx` **Role:** Notice creation form

Responsibilities: - Create new notices with rich content - Upload images, videos, and files - Add tags for categorization - Set expiry date - Mark as important (admin only)

Key Functions:

```

// Upload file to Supabase Storage
const uploadFile = async (file: File, bucket: string, path: string) => {
  const fileName = `${Date.now()}_${file.name}`
  const { data, error } = await supabase.storage
    .from(bucket)
    .upload(`/${path}/${fileName}`, file)

  const { data: { publicUrl } } = supabase.storage
    .from(bucket)
    .getPublicUrl(`/${path}/${fileName}`)

  return publicUrl
}

// Create notice with media
const handleSubmit = async () => {
  // Insert notice
  const { data: newNotice } = await supabase
    .from("notices")
    .insert({ title, description, author_id: user.id })
    .select()
    .single()

  // Upload and attach media
  for (const image of images) {
    const url = await uploadFile(image, "bellsnotice", "images")
    await supabase

```

```

    .from("notice_media")
    .insert({
      notice_id: newNotice.id,
      media_type: "image",
      media_url: url
    })
  }

// Add tags
if (form.tags.length > 0) {
  // Create or get existing tags
  const { data: existingTags } = await supabase
    .from("tags")
    .select("id, name")
    .in("name", form.tags)

  // Link tags to notice
  await supabase
    .from("notice_tags")
    .insert(tagIds.map(tag_id => ({
      notice_id: newNotice.id,
      tag_id
    })))
}
}
}

```

app/notice/[id]/page.tsx **Role:** Individual notice detail view

Responsibilities: - Display full notice content - Show attached media (images, videos, files) - Display author profile - Show comments section - Increment view count - Allow reactions and saving

Key Logic: - Fetch notice details with author profile - Fetch associated media - Fetch comments with user profiles - Increment view count on first view - Display reaction count

components/notice-card.tsx **Role:** Reusable notice card component

Responsibilities: - Display notice preview (title, description, author) - Show view count and reaction count - Link to full notice - Display author avatar

Key Code:

```

export function NoticeCard({ notice, authorProfile }: NoticeCardProps) {
  // Fetch reaction count
  useEffect(() => {
    const fetchReactionCount = async () => {
      const { count } = await supabase

```

```

        .from("reactions")
        .select("*", { count: "exact", head: true })
        .eq("notice_id", notice.id)
        setReactionCount(count || 0)
    }
    fetchReactionCount()
}, [notice.id])

return (
<Link href={`/notice/${notice.id}`}>
  <div className="card">
    <h2>{notice.title}</h2>
    <p>{notice.description}...</p>
    <div className="author">
      <img src={authorProfile?.profile_image_url} />
      <span>{authorProfile?.display_name}</span>
    </div>
    <div className="stats">
      <Heart /> {reactionCount}
      <Eye /> {notice.view_count} views
    </div>
  </div>
</Link>
)
}

```

5. Home Page Components

`components/pages/home-page.tsx` Role: Main home page with notice carousels

Responsibilities: - Display important notices (red highlighted, carousel) - Display featured notices (blue highlighted, carousel) - Display latest notices (random selection, grid) - Carousel navigation (prev/next) - Auto-refresh every 30 minutes

Key Logic:

```

// Fetch notices by category
const { data: importantData } = await supabase
  .from("notices")
  .select("*")
  .eq("is_important", true)
  .order("created_at", { ascending: false })
  .limit(20)

const { data: featuredData } = await supabase

```

```

    .from("notices")
    .select("*")
    .eq("is_featured", true)
    .order("created_at", { ascending: false })
    .limit(20)

    // Shuffle for random selection
    const randomSelected = allData?.sort(() => Math.random() - 0.5).slice(0, 12)

    // Carousel navigation
    const handleImportantNext = () => {
        setImportantCarouselIndex((i) => (i + 6) % Math.max(importantNotices.length, 6))
    }

    const importantDisplayed = importantNotices.slice(
        importantCarouselIndex,
        importantCarouselIndex + 6
    )

```

6. Navigation Components

`components/top-navbar.tsx` **Role:** Top navigation bar

Responsibilities: - Display user avatar and name - Navigation links (Dashboard, Saved, Search) - Logout functionality - Mobile-responsive menu

`components/navigation/navigation-menu.tsx` **Role:** Main navigation menu

Responsibilities: - Display navigation options based on user role - Home, Notices, Search links for all users - Create Notice link for reps and admins - Admin panel link for admins

7. Supabase Clients

`lib/supabase/client.ts` **Role:** Browser Supabase client for client components

Responsibilities: - Create singleton Supabase browser client - Use environment variables for configuration - Cache client instance for reuse

Key Code:

```

let supabaseClient: ReturnType<typeof createBrowserClient> | null = null

export function createClient() {
    if (!supabaseClient) {
        supabaseClient = createBrowserClient(
            process.env.NEXT_PUBLIC_SUPABASE_URL!

```

```

        process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!
    )
}
return supabaseClient
}

```

lib/supabase/server.ts **Role:** Server Supabase client for server components

Responsibilities: - Create Supabase server client - Handle cookie-based authentication - Provide server-side data fetching

Key Code:

```

import { createServerClient } from "@supabase/ssr"
import { cookies } from "next/headers"

export function createClient() {
  const cookieStore = cookies()
  return createServerClient(
    process.env.NEXT_PUBLIC_SUPABASE_URL!,
    process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!,
    {
      cookies: {
        get(name: string) {
          return cookieStore.get(name)?.value
        },
      },
    }
  )
}

```

8. Configuration Files

package.json **Role:** Project dependencies and scripts

Key Scripts:

```

{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "eslint ."
  }
}

```

Key Dependencies: - **next:** Next.js framework - **react:** React library - **@supabase/supabase-js:** Supabase client - **@supabase/ssr:** Supabase SSR

support - tailwindcss: CSS framework - lucide-react: Icon library - react-hook-form: Form handling

next.config.mjs **Role:** Next.js configuration

Purpose: - Configure Next.js behavior - Set up PWA configuration - Define experimental features

tailwind.config.ts **Role:** Tailwind CSS configuration

Purpose: - Define custom theme - Extend Tailwind utilities - Configure plugins

Component Architecture

Component Hierarchy

```
RootLayout (app/layout.tsx)
  RegisterSW (PWA Service Worker)
  Analytics (Vercel Analytics)
  Page Content
    TopNavbar
    NavigationMenu
    Main Content
      LoginPage
      HomePage
        ImportantNotices (Carousel)
        FeaturedNotices (Carousel)
        LatestNotices (Grid)
        NoticeCard (× Multiple)
    Dashboard
      ProfileTab
      CommentsTab
      NoticesTab
      RequestsTab
      AnalyticsTab
    NoticeDetail
      NoticeContent
      NoticeMedia
      CommentsSection
      Comment (× Multiple)
  NoticeCreate
    NoticeForm
    MediaUpload
    TagInput
```

Component Patterns

1. Server Components (Default)

- Fetch data on the server
- Better performance
- SEO friendly
- No “use client” directive

Example: `app/notices/page.tsx`

2. Client Components

- Marked with “use client”
- Handle user interactions
- Manage local state
- Use browser APIs

Example: `components/notice-card.tsx, app/dashboard/page.tsx`

3. Container/Presenter Pattern

- Container: Handles data fetching and logic
- Presenter: Handles UI rendering

Example: - Container: `app/dashboard/page.tsx` - Presenter: `components/notice-card.tsx`

4. Composition Pattern

- Small, reusable components
- Combined to build complex UIs
- Props-based configuration

Example: shadcn/ui components (Button, Card, Dialog, etc.)

Data Flow

Authentication Flow

1. User visits `/app/page.tsx`
2. Page checks Supabase auth state
3. If authenticated:
 - Show HomePage with TopNavbar
 - User can access protected routes
4. If not authenticated:
 - Show LoginPage
 - User enters credentials
 - Supabase validates credentials
 - Auth state updated

- Redirect to home

Notice Creation Flow

1. User clicks "Create Notice" (Rep/Admin only)
2. Navigate to /app/notice/create/page.tsx
3. User fills form (title, description, tags, media)
4. On submit:
 - a. Validate form data
 - b. Upload media to Supabase Storage
 - c. Create notice record in database
 - d. Create media records linked to notice
 - e. Create or link tags
 - f. Redirect to notice detail page

Notice Request Flow

1. Regular user submits notice request
2. Select target representative
3. Request saved in notice_requests table
4. Rep sees request in dashboard
5. Rep accepts/declines:
 - Accept: Creates notice, transfers media, updates status to "approved"
 - Decline: Updates status to "rejected" with optional message
6. User notified of decision

Search Flow

1. User enters search query
 2. Debounce input (300ms)
 3. Query database:
 - Search in title (ILIKE)
 - Search in description (ILIKE)
 - Apply filters (tags, department, date)
 4. Return matching notices
 5. Display in search results
 6. Real-time updates as user types
-

State Management

Local State (useState)

Used for: - Form inputs (title, description, etc.) - UI toggles (modals, dropdowns) - Loading states - Error messages - Local selections

Example:

```

const [form, setForm] = useState({
  title: "",
  description: "",
  tags: []
})

const [loading, setLoading] = useState(true)
const [error, setError] = useState(null)

```

Server State (Database)

Managed by: - Supabase database - Real-time subscriptions (optional) - Server-side data fetching

Example:

```

const { data: notices } = await supabase
  .from("notices")
  .select("*")
  .order("created_at", { ascending: false })

```

Global State (Context API)

Used for: - Theme (light/dark mode) - Authentication state (via Supabase) - Toast notifications (useToast hook)

Example:

```

// Theme provider
<ThemeProvider>
  <App />
</ThemeProvider>

// Toast notifications
const toast = useToast()
toast({ title: "Success", description: "Notice created!" })

```

Session Storage

Used for: - Admin authentication flag - Temporary state that doesn't need persistence

Example:

```

sessionStorage.setItem("adminAuthenticated", "true")
const isAdmin = sessionStorage.getItem("adminAuthenticated") === "true"

```

Authentication Flow

User Authentication

1. Registration Flow

- User fills registration form
- Supabase auth.signUp() creates auth user
 - Profile record created in profiles table
 - Redirect to sign-up-success page
 - User can now login

2. Login Flow

- User enters email/matrik and password
- If matrik number: Look up email in profiles table
 - Supabase auth.signInWithEmailAndPassword()
 - Session created in Supabase
 - Auth state listener updates UI
 - Redirect to home/dashboard

3. Logout Flow

- User clicks logout
- Supabase auth.signOut()
 - Session destroyed
 - Auth state listener updates UI
 - Redirect to login page

Admin Authentication

1. Admin Login Flow

- User clicks "Sign in as Admin"
- Enters admin password
 - Validate against hardcoded password ("adminnotice")
 - Set sessionStorage flag
 - Redirect to admin panel

2. Admin Session Check

- On protected admin routes:
- Check sessionStorage for "adminAuthenticated"
 - If false, redirect to login
 - If true, allow access

Route Protection

```
// Check authentication in page components
useEffect(() => {
  const checkUser = async () => {
    const { data: { user } } = await supabase.auth.getUser()

    if (!user) {
      router.push("/")
      return
    }

    // Check user role
    const { data: profile } = await supabase
      .from("profiles")
      .select("user_type")
      .eq("id", user.id)
      .single()

    if (profile?.user_type !== "rep") {
      router.push("/")
      return
    }
  }

  checkUser()
}, [])
```

Key Algorithms & Logic

1. Carousel Navigation

Purpose: Navigate through notice carousels with pagination

Algorithm:

```
// Calculate displayed items
const displayed = notices.slice(
  currentIndex,
  currentIndex + itemsPerPage
)

// Next button
const handleNext = () => {
  setCurrentIndex((i) =>
    (i + itemsPerPage) % Math.max(notices.length, itemsPerPage)
```

```

        )
    }

// Previous button
const handlePrev = () => {
    setCurrentIndex((i) =>
        (i - itemsPerPage + Math.max(notices.length, itemsPerPage)) %
        Math.max(notices.length, itemsPerPage)
    )
}

// Auto-refresh
setInterval(() => {
    setCurrentIndex((i) => (i + itemsPerPage) % maxIndex)
}, 30 * 60 * 1000) // 30 minutes

```

2. Random Notice Selection

Purpose: Display random notices for variety

Algorithm:

```

// Fisher-Yates shuffle
const shuffleArray = (array) => {
    const newArray = [...array]
    for (let i = newArray.length - 1; i > 0; i--) {
        const j = Math.floor(Math.random() * (i + 1))
        ;[newArray[i], newArray[j]] = [newArray[j], newArray[i]]
    }
    return newArray
}

// Shuffle and select
const randomNotices = shuffleArray(allNotices).slice(0, 12)

```

3. Tag Management

Purpose: Handle tag creation and linking

Algorithm:

```

// Get existing tags
const { data: existingTags } = await supabase
    .from("tags")
    .select("id, name")
    .in("name", form.tags)

// Extract existing names

```

```

const existingTagNames = existingTags?.map(t => t.name) || []

// Find new tags
const newTags = form.tags.filter(t => !existingTagNames.includes(t))

// Create new tags
if (newTags.length > 0) {
  const { data: createdTags } = await supabase
    .from("tags")
    .insert(newTags.map(name => ({ name })))
    .select()

  // Combine all tag IDs
  const allTagIds = [
    ...existingTags?.map(t => t.id) || [],
    ...createdTags?.map(t => t.id) || []
  ]

  // Link tags to notice
  await supabase
    .from("notice_tags")
    .insert(allTagIds.map(tag_id => ({
      notice_id: newNotice.id,
      tag_id
    })))
}

```

4. Media Upload Handling

Purpose: Upload files and create database records

Algorithm:

```

// Upload each file
for (const file of files) {
  // Generate unique filename
  const fileName = `${Date.now()}_${file.name}`

  // Upload to Supabase Storage
  const { data, error } = await supabase.storage
    .from("bellsnote")
    .upload(`/${path}/${fileName}`, file)

  if (error) throw error

  // Get public URL
  const { data: { publicUrl } } = supabase.storage

```

```

    .from("bellsnotice")
    .getPublicUrl(`${path}/${fileName}`)

    // Store URL for database insertion
    mediaUrls.push({
        type: getFileType(file),
        url: publicUrl,
        isLink: false
    })
}

// Insert media records
await supabase
    .from("notice_media")
    .insert(mediaUrls.map(m => ({
        notice_id: newNotice.id,
        media_type: m.type,
        media_url: m.url,
        is_link: m.isLink
    })))

```

5. Search with Filters

Purpose: Advanced search with multiple criteria

Algorithm:

```

// Build query
let query = supabase
    .from("notices")
    .select("*")

// Add text search
if (searchQuery) {
    query = query.or(`title.ilike.%${searchQuery}%,description.ilike.%${searchQuery}%`)
}

// Add tag filter
if (selectedTags.length > 0) {
    query = query.in("id",
        subquery: supabase
            .from("notice_tags")
            .select("notice_id")
            .in("tag_id", selectedTags)
    )
}

```

```

// Add date filter
if (dateRange.from) {
    query = query.gte("created_at", dateRange.from)
}

// Add department filter (via author)
if (selectedDepartment) {
    query = query.in("author_id",
        subquery: supabase
            .from("profiles")
            .select("id")
            .eq("department", selectedDepartment)
    )
}

// Execute query
const { data: results } = await query.order("created_at", { ascending: false })

```

6. Cascade Deletion

Purpose: Delete notice and all related data

Algorithm:

```

const handleDeleteNotice = async (noticeId: string) => {
    // Delete in order (to respect foreign keys)

    // 1. Delete tags
    await supabase
        .from("notice_tags")
        .delete()
        .eq("notice_id", noticeId)

    // 2. Delete media
    await supabase
        .from("notice_media")
        .delete()
        .eq("notice_id", noticeId)

    // 3. Delete comments
    await supabase
        .from("comments")
        .delete()
        .eq("notice_id", noticeId)

    // 4. Delete reactions
    await supabase

```

```

    .from("reactions")
    .delete()
    .eq("notice_id", noticeId)

    // 5. Delete saved notices
    await supabase
        .from("saved_notices")
        .delete()
        .eq("notice_id", noticeId)

    // 6. Delete notice
    await supabase
        .from("notices")
        .delete()
        .eq("id", noticeId)
}

```

7. Reaction Toggle

Purpose: Toggle like/unlike on notices

Algorithm:

```

const handleToggleReaction = async (noticeId: string) => {
    // Check if already reacted
    const { data: existingReaction } = await supabase
        .from("reactions")
        .select("*")
        .eq("user_id", user.id)
        .eq("notice_id", noticeId)
        .single()

    if (existingReaction) {
        // Remove reaction
        await supabase
            .from("reactions")
            .delete()
            .eq("id", existingReaction.id)
    } else {
        // Add reaction
        await supabase
            .from("reactions")
            .insert({
                user_id: user.id,
                notice_id: noticeId
            })
    }
}

```

```
// Refresh reaction count
const { count } = await supabase
  .from("reactions")
  .select("*", { count: "exact", head: true })
  .eq("notice_id", noticeId)

  setReactionCount(count || 0)
}
```

Performance Optimizations

1. Server Components

- Use server components by default for better performance
- Only use client components when necessary
- Reduce JavaScript bundle size

2. Code Splitting

- Dynamic imports for large components
- Route-based code splitting with Next.js
- Lazy loading for modals and dialogs

3. Image Optimization

- Use Next.js Image component
- Automatic resizing and optimization
- Lazy loading for below-fold images

4. Debouncing

- Debounce search inputs (300ms)
- Reduce API calls during typing
- Improve user experience

5. Pagination

- Limit query results (e.g., 20 items per page)
- Implement pagination for large datasets
- Use cursor-based pagination for better performance

6. Caching

- Leverage Supabase caching
- Use React Query for client-side caching (future enhancement)

- Implement HTTP caching headers
-

Security Considerations

1. Row Level Security (RLS)

- Enable RLS on all Supabase tables
- Create policies based on user roles
- Ensure users can only access their own data

2. Input Validation

- Validate all user inputs on client and server
- Use Zod schemas for form validation
- Sanitize user-generated content

3. File Upload Security

- Validate file types and sizes
- Use virus scanning for uploads
- Store files in secure bucket with proper policies

4. Authentication

- Never expose service role keys on client-side
- Use Supabase Auth for secure authentication
- Implement proper session management

5. SQL Injection Prevention

- Use parameterized queries (Supabase handles this)
 - Avoid raw SQL queries
 - Validate all query parameters
-

Future Enhancements

1. Real-time Updates

- Use Supabase Realtime for live updates
- Notify users of new notices instantly
- Live comment updates

2. Advanced Analytics

- Track user engagement metrics

- Notice performance analytics
- User behavior analysis

3. Email Notifications

- Email alerts for new notices
- Notification preferences
- Digest emails

4. Offline Support

- Enhanced PWA capabilities
- Offline notice viewing
- Sync when online

5. Advanced Search

- Full-text search with better ranking
 - Search suggestions and autocomplete
 - Advanced filtering options
-

This document provides a comprehensive overview of the BellsNotice codebase architecture. For implementation details, refer to the actual source code files.

Last Updated: January 2025