Contents

Web App Architecture Analysis & Modularity Assessment

Date: September 4, 2025
Project: Ario Web App (Vue 3 + Nostr + LNbits)
Objective: Evaluate current architecture for modular plugin-based development

---

Executive Summary

The current web application demonstrates solid foundational architecture with excellent Nostr infrastructure (RelayHub/NostrclientHub) but lacks true modularity for plugin-based development.

Overall Modularity Rating: 6.5/10

The app successfully implements core Nostr functionality and component organization but requires significant architectural refactoring to achieve the stated objective of independent, pluggable feature modules.

Current Architecture Assessment

Strengths (What's Working Well)

1. Strong Base Infrastructure (8/10)

- **RelayHub & NostrclientHub**: Excellent centralized Nostr infrastructure with robust connection management, event handling, and browser compatibility
- **Pinia Stores**: Clean state management separation (`nostr.ts`, `market.ts`) with reactive patterns
- **Composables Pattern**: Good reactive logic encapsulation (`useRelayHub`, `useNostrChat`, `useMarket`)
- **UI Component Library**: Well-structured Shadcn/ui components with consistent design system

2. Component Organization (7/10)

- Components grouped by domain (`/market`, `/nostr`, `/events`, `/auth`)
- Page-level routing with lazy loading implementation
- Composables provide reusable business logic across components
- Clean separation of concerns between UI and business logic

3. Service Layer (6/10)

- Clean service abstractions (`nostrmarketService`, `paymentMonitor`)
- API layer separation (`/lib/api`)
- Type-safe interfaces and proper error handling

Critical Issues (What's Missing)

1. No Plugin Architecture (3/10)

- **Hard-coded routes**: All features baked into main router configuration
- **Tight coupling**: Components directly import across modules without abstraction
- **No feature isolation**: Cannot develop/deploy components independently
- **No plugin registration system**: Missing infrastructure for dynamic feature loading

2. Cross-Module Dependencies (4/10) Examples of problematic tight coupling:

```
// Payment monitor importing market types directly
src/lib/services/paymentMonitor.ts: import type { Order } from '@/stores/market'

// Navbar importing chat composable directly
src/components/layout/Navbar.vue: import { nostrChat } from '@/composables/useNostrChat'

// Services importing market store directly
src/lib/services/nostrmarketService.ts: import type { Stall, Product, Order } from '@/stores/ma
```

3. Missing Module Boundaries (5/10)

- No clear module interfaces/contracts

- Shared types scattered across modules without centralized definition
- No dependency injection system for service management
- Missing module lifecycle management

Required Changes for Modular Architecture

Phase 1: Core Infrastructure Refactor

1. Create Module System

```
// /src/modules/base/types.ts
interface ModuleConfig {
  name: string
  routes: RouteConfig[]
  components: ComponentConfig[]
  services: ServiceConfig[]
  dependencies?: string[]
}

interface ModulePlugin {
  install(app: App, options?: any): void
  uninstall(): void
}
```

2. Plugin Registration System

```
// /src/core/plugin-manager.ts
class PluginManager {
  private plugins = new Map<string, ModulePlugin>()

  register(module: ModulePlugin): void
  unregister(name: string): void
  getModule(name: string): ModulePlugin
  getDependencies(name: string): string[]
}
```

Phase 2: Module Extraction

1. Base Module Structure

```
/src/modules/base/
  nostr/
    relayHub.ts          # Centralized relay management
    nostrclientHub.ts    # Client connection handling
    types.ts             # Nostr-specific types
  auth/
    lnbits.ts           # LNbits authentication
    composables/
  pwa/
```

```
    install-prompt.ts
    notifications.ts
  ui/                    # Shadcn components
```

2. Feature Modules

```
/src/modules/
  nostr-feed/
    components/
      NostrFeed.vue
    composables/
      useFeed.ts
    services/
      feedService.ts
    index.ts             # Module export
  market/
    components/
    composables/
    stores/
    services/
    index.ts
  chat/
  events/
```

3. Module Interface Standards

```typescript
// Each module exports standardized interface
export interface NostrFeedModule extends ModulePlugin {
  name: 'nostr-feed'
  components: {
    NostrFeed: Component
  }
  routes: RouteConfig[]
  dependencies: ['base']
  config?: NostrFeedConfig
}
```

Phase 3: Event-Driven Communication

Replace Direct Imports with Event Bus

```typescript
// /src/core/event-bus.ts
interface ModuleEventBus {
  emit(event: string, data: any): void
  on(event: string, handler: Function): void
  off(event: string, handler: Function): void
}

// Example usage:
```

```
// Market module: eventBus.emit('order:created', order)
// Chat module: eventBus.on('order:created', handleNewOrder)
```

Dependency Injection Container

```
// /src/core/di-container.ts
class DIContainer {
  provide<T>(token: string, instance: T): void
  inject<T>(token: string): T

  // Module-scoped injection
  provideForModule(module: string, token: string, instance: any): void
}
```

Recommended Ground-Up Architecture

If rebuilding from scratch, implement these architectural patterns:

1. Module-First Architecture

```
src/
  core/                    # Base app infrastructure
    plugin-manager.ts     # Plugin registration & lifecycle
    di-container.ts       # Dependency injection
    event-bus.ts          # Inter-module communication
    module-registry.ts    # Module discovery & loading
    base-services/        # Core services
  modules/
    base/                # Core functionality (required)
      nostr/            # RelayHub, NostrclientHub
      auth/             # LNbits authentication
      pwa/              # Progressive Web App features
      ui/               # Shadcn component library
    nostr-feed/         # Announcements & social feed
    market/             # Marketplace functionality
    chat/               # Nostr chat implementation
    events/             # Event ticketing system
  app.ts                # Application composition & startup
```

2. Configuration-Driven Setup

```
// /src/app.config.ts
export const appConfig = {
  modules: {
    base: {
      required: true,
      nostr: {
        relays: process.env.VITE_NOSTR_RELAYS
```

```
      }
    },
    'nostr-feed': {
      enabled: true,
      config: {
        refreshInterval: 30000,
        maxPosts: 100
      }
    },
    market: {
      enabled: true,
      config: {
        defaultCurrency: 'sats',
        paymentTimeout: 300000
      }
    },
    chat: {
      enabled: false  // Can be toggled via config
    },
    events: {
      enabled: true,
      config: {
        ticketValidationEndpoint: '/api/tickets/validate'
      }
    }
  },
  features: {
    pwa: true,
    pushNotifications: true,
    electronApp: true
  }
}
```

3. Dynamic Module Loading

```
// /src/core/module-loader.ts
class ModuleLoader {
  async loadModule(name: string): Promise<ModulePlugin> {
    const module = await import(`@/modules/${name}/index.ts`)
    await this.resolveDependencies(module.dependencies || [])
    return module.default
  }

  async unloadModule(name: string): Promise<void> {
    const module = this.getModule(name)
    if (module) {
      await module.uninstall()
      this.cleanupRoutes(name)
```

```
      this.cleanupServices(name)
    }
  }

  private async resolveDependencies(deps: string[]): Promise<void> {
    for (const dep of deps) {
      if (!this.isModuleLoaded(dep)) {
        await this.loadModule(dep)
      }
    }
  }
}
```

4. Module Development Workflow

```
# Develop single module in isolation
npm run dev:module market

# Test module independently
npm run test:module market

# Build specific modules only
npm run build:modules market,chat

# Hot reload module during development
npm run dev:hot-module nostr-feed
```

Key Architectural Improvements

1. Dependency Inversion

- Modules depend on interfaces, not concrete implementations
- Services injected via DI container rather than direct imports
- Clear separation between module contracts and implementations

2. Event-Driven Communication

- Replace direct imports with event-based messaging
- Loose coupling between modules via standardized events
- Centralized event routing and handling

3. Module Lifecycle Management

- Proper setup and teardown hooks for each module
- Resource cleanup when modules are disabled/unloaded
- Dependency resolution during module loading

4. Configuration Over Code

- Enable/disable features via configuration files
- Runtime module toggling without code changes
- Environment-specific module configurations

5. Hot Module Replacement

- Develop modules independently without full app restart
- Live reloading of individual modules during development
- Isolated testing environments for each module

Implementation Roadmap

High Priority (Immediate)

1. **Extract RelayHub/NostrclientHub to base module** (mostly complete)
2. **Create plugin registration system** - Core infrastructure for module loading
3. **Convert nostr-feed to plugin pattern** - Proof of concept implementation

Medium Priority (Phase 2)

4. **Implement event bus communication** - Replace direct imports between modules
5. **Add module lifecycle management** - Proper setup/teardown hooks
6. **Create development tooling** - Scripts for isolated module development

Low Priority (Phase 3)

7. **Add module versioning support** - Handle different module versions
8. **Implement hot module replacement** - Live development workflow
9. **Add module marketplace** - Plugin discovery and installation

---

Conclusion

The current web application has excellent technical foundations, particularly the Nostr infrastructure (RelayHub/NostrclientHub), but requires significant architectural refactoring to achieve true modularity.

Key Strengths to Preserve: - Robust Nostr client implementation - Clean component organization - Solid TypeScript foundations - PWA capabilities

Critical Areas for Improvement: - Plugin architecture implementation - Module boundary enforcement - Event-driven communication - Development workflow optimization

Estimated Effort: - **Phase 1 (Core Infrastructure):** 2-3 weeks - **Phase 2 (Module Extraction):** 3-4 weeks - **Phase 3 (Advanced Features):** 2-3 weeks

The investment in modular architecture will enable independent development of features, easier testing, better code maintainability, and the ability to dynamically enable/disable functionality based on deployment requirements.

Generated by: Claude Code Architecture Analysis
Last Updated: September 4, 2025