extranuts architecture modulaire 2

```
ere is Claude's plan: |
|
╭──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────╮ |
| | Modularization Plan for Parallel Development | |
| | | |
| | Goal | |
| | | |
| | Restructure the codebase to enable multiple developers to work on
different features without conflicts, using a clean | |
| | modular architecture. | |
| | | |
| | Core Principles | |
| | | |
| | 1. Feature-based modules - Each feature is self-contained | |
| | 2. Minimal shared core - Only essential shared types/utilities | |
| | 3. Clear interfaces - Well-defined boundaries between modules | |
| | 4. Single responsibility - Each module does one thing well | |
| | | |
| | Proposed Architecture | |
| | | |
| | Backend (Rust/Tauri) | |
| | | |
| | src-tauri/src/ | |
| | ├── core/ # Minimal shared core | |
| | | ├── mod.rs # Core module exports | |
| | | ├── state.rs # Shared application state | |
| | | └── error.rs # Common error types | |
| | | | |
| | ├── features/ # Feature modules | |
| | | ├── mod.rs # Feature registration | |
| | | | |
| | | ├── notes/ # Notes feature | |
| | | | ├── mod.rs # Public interface | |
| | | | ├── commands.rs # Tauri commands | |
| | | | ├── service.rs # Business logic | |
| | | | ├── models.rs # Note-specific models | |
| | | | └── repository.rs # Database operations | |
| | | | | |
| | | ├── windows/ # Window management feature | |
```

```
| | | | ├── mod.rs | |
| | | | ├── commands.rs | |
| | | | ├── service.rs | |
| | | | └── tray.rs | |
| | | | | |
| | | ├── sync/ # Sync feature | |
| | | | ├── mod.rs | |
| | | | ├── commands.rs | |
| | | | ├── service.rs | |
| | | | └── icloud.rs | |
| | | | | |
| | | ├── preferences/ # Settings feature | |
| | | | ├── mod.rs | |
| | | | ├── commands.rs | |
| | | | ├── service.rs | |
| | | | └── models.rs | |
| | | | | |
| | | └── search/ # Search feature | |
| | | ├── mod.rs | |
| | | ├── commands.rs | |
| | | ├── service.rs | |
| | | └── indexer.rs | |
| | | | |
| | ├── infrastructure/ # Technical infrastructure | |
| | | ├── mod.rs | |
| | | ├── database/ | |
| | | | ├── mod.rs | |
| | | | ├── connection.rs | |
| | | | └── migrations.rs | |
| | | └── storage/ | |
| | | ├── mod.rs | |
| | | └── paths.rs | |
| | | | |
| | ├── lib.rs # Minimal app initialization | |
| | └── main.rs # Entry point | |
| | | |
| | Frontend (SolidJS/TypeScript) | |
| | | |
| | src/ | |
| | ├── core/ # Shared core | |
| | | ├── types/ # Shared types | |
| | | ├── api/ # API client | |
| | | └── utils/ # Common utilities | |
| | | | |
```

```
| | ├── features/ # Feature modules | |
| | | ├── notes/ | |
| | | | ├── components/ # Note-specific components | |
| | | | ├── services/ # API services | |
| | | | ├── stores/ # State management | |
| | | | └── types.ts # Local types | |
| | | | | |
| | | ├── editor/ | |
| | | | ├── components/ | |
| | | | ├── plugins/ # Editor plugins | |
| | | | └── hooks/ | |
| | | | | |
| | | ├── search/ | |
| | | | ├── components/ | |
| | | | ├── services/ | |
| | | | └── utils/ | |
| | | | | |
| | | ├── sync/ | |
| | | | ├── components/ | |
| | | | ├── services/ | |
| | | | └── hooks/ | |
| | | | | |
| | | └── preferences/ | |
| | | ├── components/ | |
| | | ├── services/ | |
| | | └── stores/ | |
| | | | |
| | ├── shared/ # Shared UI components | |
| | | ├── components/ # Generic UI components | |
| | | ├── layouts/ # Layout components | |
| | | └── hooks/ # Shared hooks | |
| | | | |
| | ├── App.tsx # Root component (minimal) | |
| | └── index.tsx # Entry point | |
| | | |
| | Implementation Steps | |
| | | |
| | Phase 1: Backend Modularization | |
| | | |
| | 1. Create core/ module with shared state and error types | |
| | 2. Extract window management into features/windows/ | |
| | 3. Move note operations into features/notes/ | |
| | 4. Separate sync logic into features/sync/ | |
| | 5. Create infrastructure layer for database/storage | |
```

	Phase 2: Frontend Modularization	
	1. Create feature folders matching backend structure	
	2. Move components into feature-specific folders	
	3. Extract shared components into shared/	
	4. Create feature-specific service layers	
	Phase 3: Interface Definition	
	1. Define clear TypeScript interfaces for each feature	
	2. Create feature manifests for registration	
	3. Implement dependency injection for services	
	4. Add feature flags for optional features	
	Phase 4: Testing & Documentation	
	1. Add unit tests per feature module	
	2. Create integration tests for feature interactions	
	3. Document each module's public API	
	4. Add README.md in each feature folder	
	Benefits	
	1. Parallel Development: Developers can work on separate features without conflicts	
	2. Clear Ownership: Each feature can have a designated owner/team	
	3. Easy Testing: Features can be tested in isolation	
	4. Scalability: New features can be added without touching core	
	5. Code Reuse: Shared components are clearly identified	
	6. Onboarding: New developers can focus on one feature at a time	
	Migration Strategy	
	1. Start with backend modularization (less UI impact)	
	2. Migrate one feature at a time	
	3. Keep existing APIs working during migration	
	4. Update imports gradually	
	5. Run tests after each migration step	
	Example: Working on Search Feature	

	A developer working on search would only touch:	
	- src-tauri/src/features/search/	
	- src/features/search/	
	- Their feature's models and commands	
	They wouldn't need to modify:	
	- Core application files	
	- Other features' code	
	- Shared infrastructure (unless adding new capabilities)	
	This structure ensures minimal merge conflicts and clear boundaries between different development efforts.