Tauri With Custom Rust Backend Structure

Structured Rust backend with custom commands, state management, and organized modules.

#tauri #rust #desktop #commands #backend

PNG PDF Copy Prompt

Project Directory

```
my-tauri-app/
    package.json
    vite.config.js
    index.html
  > src/                    Frontend
      main.js
      App.vue
  > lib/                    Tauri bindings
      commands.js           Invoke wrappers
      events.js             Event listeners
  > components/
      FileList.vue
      Settings.vue
  > stores/                 Pinia stores
      app.js
> src-tauri/
    Cargo.toml
    tauri.conf.json
    build.rs
  > src/
      main.rs
      lib.rs                Plugin and stat…
    > commands/             Tauri commands
        mod.rs
        files.rs            File operations
        settings.rs
        system.rs           System info
    > state/                Managed state
        mod.rs
        app_state.rs
        config.rs
    > services/             Business logic
        mod.rs
        file_service.rs
        db.rs               SQLite via sqlx
    > utils/
        mod.rs
        paths.rs
        error.rs            Custom error ty…
    icons/
  > capabilities/
        default.json
  public/
```

Why This Structure?

When your Tauri app needs real Rust power—file processing, database access, system integration—you need organized backend code. This structure separates concerns: `commands/` for IPC handlers, `services/` for business logic, `state/` for managed state.

Key Directories

src-tauri/src/commands/ - One file per command group, registered in `lib.rs`

src-tauri/src/services/ - Pure Rust logic, no Tauri dependencies

src-tauri/src/state/ - App state managed via `tauri::State`

src/lib/ - Frontend wrappers for `invoke()` calls

Command Pattern

```rust
// src-tauri/src/commands/files.rs
#[tauri::command]
pub async fn read_file(path: String) -> Result {
    std::fs::read_to_string(&path)
        .map_err(|e| e.to_string())
}

// src/lib/commands.js
import { invoke } from '@tauri-apps/api/core';
export const readFile = (path) => invoke('read_file', { pa
```

Getting Started

1. `npm create tauri-app@latest`
2. Create `src-tauri/src/commands/` directory
3. Add command modules and register in `lib.rs`
4. Create frontend wrappers in `src/lib/commands.js`

When To Use This

- Apps with significant Rust business logic
- File system heavy operations
- Database-backed desktop apps
- System utilities needing native access

Best Practices

- Keep commands thin—delegate to services
- Use `Result` for command return types
- Wrap `invoke()` calls in typed JS functions
- Handle errors in services, format for UI in commands
- Use async commands for I/O operations

Testing Strategy

Services - Unit test pure Rust logic with `#[cfg(test)]`

Commands - Integration test via mock app handle

Frontend - Mock `invoke()` for component tests