Heaven UI Architecture

Overview

Heaven UI is a Tauri-based desktop application for Solo Git, built with React, TypeScript, and Tailwind CSS. The architecture emphasizes clean separation between web components, desktop-specific functionality, and shared utilities.

Design Principles

1. Separation of Concerns

- **Web Components**: Platform-agnostic React components that work in any environment
- **Desktop Components**: Tauri-specific components that interact with the native system
- **Shared Utilities**: Common types, hooks, and utilities used across the application

2. Component-First Architecture

- Each component is self-contained with its own logic and styling
- Components communicate through well-defined props interfaces
- State management is kept local when possible, lifted when necessary

3. Type Safety

- Comprehensive TypeScript interfaces for all data structures
- Strict type checking enabled
- No `any` types except when interfacing with external libraries

4. Accessibility First

- WCAG AA compliance
- Keyboard navigation for all interactions
- Screen reader support with proper ARIA labels
- Focus management in modals and overlays

5. Performance Optimization

- Lazy loading for heavy components
- Virtualization for long lists
- Debouncing for search inputs
- Efficient re-render strategies with React.memo

Directory Structure

```
src/
├── components/
│   ├── web/                    # Platform-agnostic React components
│   │   ├── CommandPalette.tsx  # Command palette with AI suggestions
│   │   ├── FileExplorer.tsx    # File tree view
│   │   ├── StatusBar.tsx       # Status bar with test results
│   │   ├── VoiceInput.tsx      # Voice-enabled input field
│   │   ├── EmptyState.tsx      # Reusable empty states
│   │   └── index.ts            # Barrel export
│   │
│   ├── desktop/                # Tauri-specific components
│   │   ├── NativeMenuBar.tsx   # Native OS menu integration
│   │   ├── FileSystemBridge.tsx # File system operations via Tauri
│   │   └── index.ts
│   │
│   ├── shared/                 # Shared utilities and infrastructure
│   │   ├── types/              # TypeScript type definitions
│   │   │   ├── common.ts       # Common types (ViewMode, Notification, etc.)
│   │   │   ├── command.ts      # Command palette types
│   │   │   ├── file.ts         # File system types
│   │   │   ├── commit.ts       # Git commit types
│   │   │   ├── test.ts         # Testing types
│   │   │   ├── git.ts          # Git operation types
│   │   │   └── index.ts        # Barrel export
│   │   │
│   │   ├── hooks/              # Custom React hooks
│   │   │   ├── useClickOutside.ts
│   │   │   ├── useFocusTrap.ts
│   │   │   ├── useDebounce.ts
│   │   │   ├── useLocalStorage.ts
│   │   │   └── index.ts
│   │   │
│   │   └── utils/              # Utility functions
│   │       ├── fileIcons.ts    # File icon mapping
│   │       ├── classNames.ts   # CSS class utilities
│   │       ├── keyboard.ts     # Keyboard event helpers
│   │       ├── time.ts         # Time formatting
│   │       ├── format.ts       # General formatting
│   │       └── index.ts
│   │
│   ├── ui/                     # Base UI components (Button, Input, etc.)
│   │   └── Button.tsx
│   │
│   └── layout/                 # Layout components
│       ├── MainLayout.tsx
│       ├── Header.tsx
│       └── Sidebar.tsx
│
├── hooks/                      # App-level hooks
│   └── useKeyboardShortcuts.ts
│
├── styles/                     # Global styles
│   ├── App.css
│   └── globals.css
│
├── App.tsx                     # Main application component
└── main.tsx                    # Application entry point
```

Component Architecture

Web Components (`/components/web`)

These are pure React components that don’t depend on Tauri APIs. They can be:
- Tested in isolation
- Used in a web browser
- Reused in other React applications
- Easily documented with Storybook

Examples:
- `CommandPalette` : Search interface with AI suggestions
- `FileExplorer` : Tree view for file navigation
- `StatusBar` : Status display with test results
- `VoiceInput` : Input field with voice capability
- `EmptyState` : Reusable empty state component

Desktop Components (`/components/desktop`)

These components use Tauri APIs to interact with the native system:
- File system operations
- Native menus and dialogs
- System notifications
- Shell commands

Examples:
- `NativeMenuBar` : Uses Tauri’s menu API
- `FileSystemBridge` : Wraps Tauri’s file system APIs
- `SystemNotification` : Native OS notifications

Shared Infrastructure (`/components/shared`)

Types (`/types`)

Centralized TypeScript type definitions for the entire application:
- `common.ts` : Common types (ViewMode, GlobalState, Notification)
- `command.ts` : Command system types
- `file.ts` : File system types
- `commit.ts` : Git commit and timeline types
- `test.ts` : Test execution and results types
- `git.ts` : Git operations types

Hooks (`/hooks`)

Custom React hooks for common functionality:
- `useClickOutside` : Detect clicks outside an element
- `useFocusTrap` : Trap focus within a modal
- `useDebounce` : Debounce a value
- `useLocalStorage` : Persist state in localStorage

Utils (`/utils`)

Utility functions:
- `fileIcons.ts` : File icon and color mapping
- `classNames.ts` : CSS class utilities (cn, clsx)
- `keyboard.ts` : Keyboard shortcuts and modifiers

- `time.ts`: Time and duration formatting
- `format.ts`: General formatting utilities

State Management

Local State

- Component-specific state using `useState`
- Derived state using `useMemo`
- Side effects using `useEffect`

Lifted State

- State shared between siblings is lifted to common parent
- Props are passed down for data and callbacks up for events

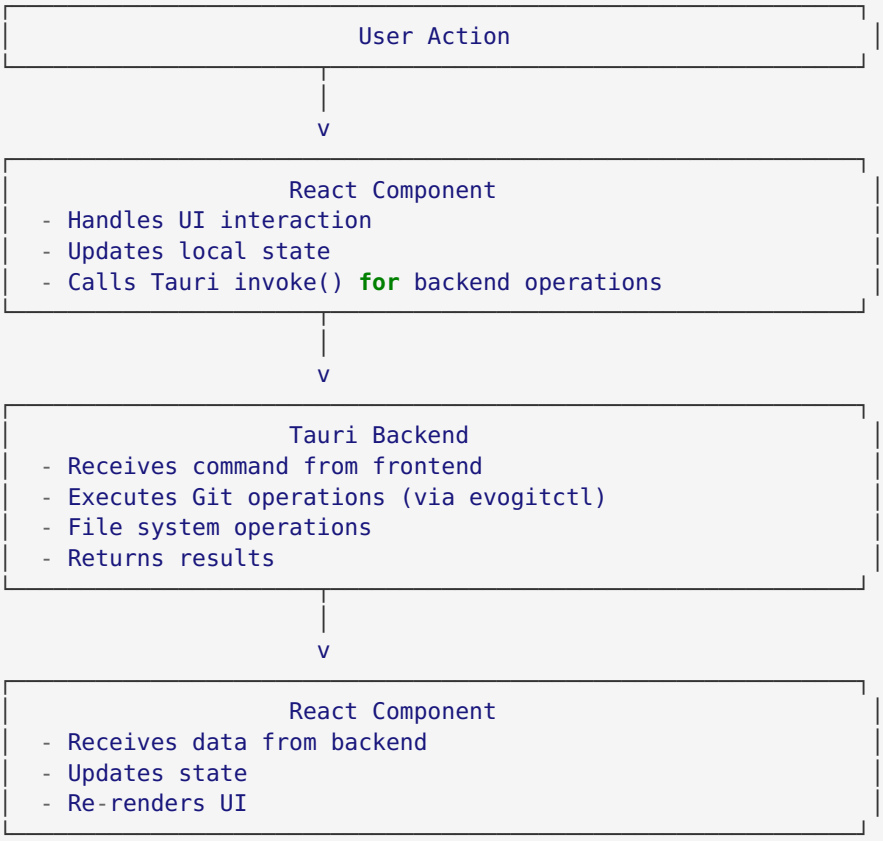
Global State

- Currently using React Context (could be extended with Zustand or Redux)
- Global state stored in `GlobalState` interface
- Accessed via Tauri's `invoke` API

Persistent State

- `useLocalStorage` hook for client-side persistence
- Settings and preferences stored locally
- Synced across app restarts

Data Flow



Key Patterns

1. Component Composition

```
// Good: Compose small components
<CommandPalette>
  <AISuggestions suggestions={suggestions} />
  <CommandList commands={commands} />
</CommandPalette>

// Instead of one monolithic component
```

2. Props Interface Pattern

```tsx
export interface ComponentProps {
  // Required props first
  id: string
  label: string

  // Optional props with ?
  description?: string
  disabled?: boolean

  // Event handlers
  onClick?: () => void
  onChange?: (value: string) => void

  // Styling
  className?: string

  // Accessibility
  'aria-label'?: string
}
```

3. Barrel Exports

```tsx
// index.ts
export * from './ComponentA'
export * from './ComponentB'

// Usage
import { ComponentA, ComponentB } from './components'
```

4. Custom Hooks Pattern

```tsx
export function useCustomHook(param: string) {
  const [state, setState] = useState(initialState)

  useEffect(() => {
    // Side effect
    return () => {
      // Cleanup
    }
  }, [param])

  return { state, setState }
}
```

5. Utility Functions

```tsx
// Pure functions, easy to test
export function formatDate(date: Date): string {
  return date.toLocaleDateString()
}
```

Styling Strategy

Tailwind CSS

- Utility-first CSS framework
- Custom theme in `tailwind.config.js`
- Design tokens mapped to Tailwind classes

Custom Theme

```
colors: {
  heaven: {
    bg: { primary, secondary, tertiary, hover, active },
    blue: { primary, hover },
    accent: { orange, green, cyan, purple, pink, red },
    text: { primary, secondary, tertiary, disabled },
    syntax: { keyword, function, string, ... }
  }
}
```

Component Styling

- Use Tailwind classes directly in JSX
- Use `cn()` utility for conditional classes
- Avoid inline styles except for dynamic values

Accessibility Strategy

Semantic HTML

- Use proper HTML elements (`<button>` , `<nav>` , `<main>`)
- Avoid `<div>` for interactive elements
- Use landmark regions

ARIA Labels

- Add `aria-label` for icon buttons
- Use `aria-describedby` for form validation
- Add `aria-live` for dynamic content
- Use `aria-expanded` for expandable sections

Keyboard Navigation

- All interactive elements focusable
- Visible focus indicators
- Escape closes modals
- Arrow keys for navigation
- Tab/Shift+Tab for focus movement

Focus Management

- `useFocusTrap` hook for modals
- Focus restoration on modal close
- Skip links for main content

Color Contrast

- All text meets WCAG AA (4.5:1)
- Interactive elements have 3:1 contrast
- Don’t rely solely on color

Testing Strategy

Unit Tests

- Test individual functions and utilities
- Test custom hooks
- Test component logic

Integration Tests

- Test component interactions
- Test data flow between components

E2E Tests

- Test complete user workflows
- Test keyboard navigation
- Test accessibility

Manual Testing

- Screen reader testing (NVDA, JAWS, VoiceOver)
- Keyboard-only navigation
- Different screen sizes
- Color contrast validation

Performance Considerations

Code Splitting

- Lazy load heavy components
- Use React.lazy() and Suspense

Memoization

- Use React.memo() for expensive components
- Use useMemo() for expensive calculations
- Use useCallback() for event handlers

Virtualization

- Virtualize long lists (react-window)
- Load data on demand
- Infinite scrolling

Debouncing

- Debounce search inputs
- Debounce resize handlers
- Use useDebounce hook

Bundle Optimization

- Tree shaking
- Minimize dependencies
- Code splitting
- Lazy loading

Development Workflow

1. Start Development Server

```
npm run tauri:dev
```

2. Type Checking

```
npm run type-check
```

3. Linting

```
npm run lint
```

4. Build for Production

```
npm run tauri:build
```

Future Enhancements

Planned Features

- [ ] Code editor with Monaco integration
- [ ] Commit graph visualization with D3
- [ ] Real-time collaboration features
- [ ] AI-powered code suggestions
- [ ] Advanced test dashboard
- [ ] Plugin system for extensibility

Technical Debt

- [ ] Add comprehensive unit tests
- [ ] Add Storybook for component documentation
- [ ] Implement proper error boundaries
- [ ] Add loading states for all async operations
- [ ] Optimize re-renders with React DevTools Profiler

Resources

Documentation

- React Docs (https://react.dev/)

- TypeScript Handbook (https://www.typescriptlang.org/docs/)
- Tailwind CSS (https://tailwindcss.com/)
- Tauri Docs (https://tauri.app/)