

MindMapper Architecture

This document provides a comprehensive overview of MindMapper’s technical architecture, design decisions, and implementation details.

Table of Contents

- 1. [Overview](#)
 - 2. [Technology Stack](#)
 - 3. [Project Structure](#)
 - 4. [Architecture Layers](#)
 - 5. [Data Flow](#)
 - 6. [Security Model](#)
 - 7. [State Management](#)
 - 8. [Layout Engine](#)
 - 9. [IPC Communication](#)
 - 10. [Build System](#)
-

Overview

MindMapper is built as an Electron desktop application with a React-based user interface. The architecture follows modern best practices for security, performance, and maintainability.

Key Principles

- **Separation of Concerns:** Clear boundaries between main process, renderer, and preload
 - **Type Safety:** Full TypeScript coverage for compile-time error detection
 - **Security First:** Context isolation, sandboxing, and minimal privileges
 - **Performance:** Efficient rendering and state updates
 - **Extensibility:** Modular design for future enhancements
-

Technology Stack

Core Technologies

Technology	Version	Purpose
Electron	27.x	Desktop application frame-work
React	18.x	UI library
TypeScript	5.x	Type-safe JavaScript
Vite	5.x	Build tool and dev server
Zustand	4.x	State management

Key Libraries

Library	Purpose
dagre	Graph layout algorithm
lucide-react	Icon set
electron-builder	Application packaging

Project Structure

```

mindmapper/
├── src/
│   ├── main/
│   │   ├── main.ts
│   │   └── preload/
│   │       ├── preload.ts
│   │       └── renderer/
│   │           ├── main.tsx
│   │           ├── App.tsx
│   │           ├── components/
│   │           │   ├── Canvas.tsx
│   │           │   ├── Toolbar.tsx
│   │           │   ├── NodeEditor.tsx
│   │           │   └── Node.tsx
│   │           ├── store/
│   │           │   └── mindMapStore.ts
│   │           ├── types/
│   │           │   ├── mindmap.ts
│   │           │   └── electron.d.ts
│   │           ├── utils/
│   │           │   ├── layout.ts
│   │           │   ├── theme.ts
│   │           │   ├── exporters.ts
│   │           │   └── importers.ts
│   │           ├── templates/
│   │           │   └── brainstorming.ts
│   │           └── styles/
│   │               ├── index.css
│   │               ├── App.css
│   │               ├── Canvas.css
│   │               ├── Toolbar.css
│   │               └── NodeEditor.css
│   ├── dist/
│   ├── release/
│   ├── package.json
│   ├── tsconfig.json
│   ├── vite.config.ts
│   └── electron-builder.yml

```

Main process (Node.js/Electron)
Entry point, window management, IPC handlers

Preload scripts (bridge between main and renderer)
IPC API exposure

Renderer process (React/TypeScript)
React entry point
Root component

React components
SVG canvas for mind map
Top toolbar
Right sidebar editor
Individual node component

State management
Zustand store

TypeScript type definitions
Core data types
Electron API types

Utility functions
Graph layout logic
Theme management
Export functionality
Import functionality

Mind map templates

CSS stylesheets
Global styles
App layout
Canvas styles
Toolbar styles
Editor styles

Compiled output
Packaged applications
Dependencies and scripts
TypeScript configuration
Vite configuration
Packaging configuration

Architecture Layers

1. Main Process Layer

Location: `src/main/main.ts`

Responsibilities:

- Window lifecycle management
- Application menu creation
- File system operations
- IPC handlers for secure file access
- Native OS integration

Key Components:

- `createWindow()` : Creates and configures the main window
- `createApplicationMenu()` : Builds the native menu
- IPC Handlers: `file:saveDialog` , `file:openDialog` , `file:exportPDF` , etc.

Security Measures:

- Runs with full Node.js privileges
- Validates all IPC inputs
- Restricts file system access to user-selected paths

2. Preload Layer

Location: `src/preload/preload.ts`

Responsibilities:

- Bridge between main and renderer processes
- Exposes safe IPC APIs to renderer
- Type-safe API definitions

Key APIs:

```

window.electronAPI = {
  file: { save, saveDialog, load, openDialog, exportPDF, exportJSON },
  dialog: { showMessage },
  app: { getPath },
  menu: { onNew, onOpen, onSave, ... },
  window: { onBeforeClose, allowClose }
}

```

Security Measures:

- Context isolation enabled
- Only whitelisted APIs exposed
- No direct Node.js access from renderer

3. Renderer Layer

Location: `src/renderer/`

Responsibilities:

- User interface rendering
- User interaction handling
- State management
- Visual layout computation

Key Components:

- **App.tsx**: Root component, keyboard shortcuts, menu handlers
- **Canvas.tsx**: SVG rendering, zoom/pan, node visualization
- **Toolbar.tsx**: Action buttons, file operations, theme toggle

- **NodeEditor.tsx**: Node property editing sidebar
- **Node.tsx**: Individual node rendering and interaction

Security Measures:

- Runs in sandboxed environment
- No direct access to Node.js APIs
- All privileged operations go through IPC

Data Flow

User Action Flow

```
graph TD;
    A[User Action] --> B[UI Component (React)];
    B --> C[Event Handler];
    C --> D[Store Action (Zustand)];
    D --> E[State Update];
    E --> F[Component Re-render];
```

File Operation Flow

```
graph TD;
    A[User Action (e.g., Save)] --> B[Store Action (saveMap)];
    B --> C[Serialize Data];
    C --> D[IPC Call (electronAPI.file.saveDialog)];
    D --> E[Main Process Handler];
    E --> F[Show Native Dialog];
    F --> G[Write to File System];
    G --> H[Return Result];
    H --> I[Update Store State (isDirty = false)];
    I --> J[Show Success Message];
```

Layout Computation Flow

```

Mind Map Data (nodes, edges)
  ↓
buildGraphLayout() [utils/layout.ts]
  ↓
Create Dagre Graph
  ↓
Add Nodes (with dimensions)
  ↓
Add Edges
  ↓
Run Layout Algorithm
  ↓
Extract Positions
  ↓
Return Positioned Nodes
  ↓
Render in Canvas

```

Security Model

Context Isolation

Enabled: Yes (via `contextIsolation: true`)

This ensures that the renderer process cannot directly access Node.js or Electron APIs, preventing malicious code injection.

Sandbox

Enabled: Yes (via `sandbox: true`)

Runs the renderer in a restricted environment with minimal privileges.

Node Integration

Disabled: Yes (via `nodeIntegration: false`)

Prevents direct access to Node.js APIs from the renderer.

Content Security Policy

While not explicitly set, the architecture naturally follows CSP principles by isolating privileged operations in the main process.

IPC Security

- All IPC handlers validate inputs
- File paths are sanitized
- User confirmation required for destructive actions
- No `eval()` or dynamic code execution

State Management

Zustand Store

Location: `src/renderer/store/mindMapStore.ts`

State Structure:

```
{
  currentMap: MindMap | null,
  selectedNodeId: string | null,
  editingNodeId: string | null,
  viewport: ViewportState,
  history: MindMap[],
  historyIndex: number,
  currentFilePath: string | null,
  isDirty: boolean
}
```

Key Actions:

- `createNewMap` : Initialize a new mind map
- `loadMap` : Load an existing mind map
- `createNode` : Add a new node
- `deleteNode` : Remove a node and its children
- `updateNodeText` : Edit node text
- `updateNodeStyle` : Change node appearance
- `undo/redo` : History navigation
- `saveMap` : Persist to disk
- `openMap` : Load from disk
- `exportPDF/JSON` : Export operations

History Management:

- Immutable state updates
- Deep copy on each modification
- Linear history (no branching)
- Undo/redo with index pointer

Layout Engine

Algorithm: Dagre

MindMapper uses the Dagre graph layout algorithm for automatic node positioning.

Location: `src/renderer/utils/layout.ts`

Process:

1. Create a directed graph
2. Add nodes with dimensions (width, height)
3. Add edges (parent-child relationships)
4. Configure layout options (direction, spacing, etc.)
5. Run the layout algorithm
6. Extract computed positions

Configuration:

```
{
  rankdir: 'LR',           // Left-to-right layout
  nodesep: 50,             // Space between nodes in same rank
  edgesep: 10,             // Space between edges
  ranksep: 80,             // Space between ranks
}
```

Optimizations:

- Layout computed only when structure changes
- Cached dimensions to avoid recalculation
- Efficient update mechanism

IPC Communication

Pattern: Invoke/Handle

Main process handlers return promises that resolve in the renderer:

```
// Main Process
ipcMain.handle('file:save', async (event, filePath, content) => {
  await fs.writeFile(filePath, content);
  return { success: true };
});

// Renderer Process
const result = await window.electronAPI.file.save(path, data);
```

Pattern: Send/On

Main process broadcasts events to renderer:

```
// Main Process
mainWindow.webContents.send('menu:save');

// Renderer Process
window.electronAPI.menu.onSave(() => {
  // Handle save action
});
```

Error Handling

All IPC handlers follow a consistent error pattern:

```
{
  success: boolean,
  error?: string,
  canceled?: boolean,
  // ... additional fields
}
```

Build System

Development

Command: `npm run dev`

Process:

1. Vite starts dev server on port 5173
2. Electron launches with dev URL
3. Hot module replacement enabled
4. DevTools opened automatically

Technologies:

- Vite for fast HMR
- ESBuild for TypeScript compilation
- Electron in development mode

Production Build

Command: `npm run build`

Process:

1. TypeScript compilation (main + preload)
2. Vite builds renderer bundle
3. Assets optimized and minified
4. Output to `dist/` directory

Packaging

Command: `npm run package`

Process:

1. Run production build
2. electron-builder creates installers
3. Platform-specific packages generated
4. Output to `release/` directory

Platforms:

- Windows: NSIS installer
- macOS: DMG and ZIP
- Linux: AppImage, deb, rpm

Performance Considerations

Rendering Optimization

1. **React Memoization:** Use `React.memo()` for expensive components
2. **Selective Re-renders:** Only update changed nodes
3. **Virtual DOM:** React's efficient diffing algorithm
4. **CSS Transitions:** Smooth animations with GPU acceleration

State Updates

1. **Immutable Updates:** Prevent unnecessary re-renders

2. **Batched Updates:** React automatically batches state changes
3. **Shallow Equality:** Zustand uses shallow comparison

Layout Computation

1. **On-Demand:** Only compute when structure changes
2. **Cached Dimensions:** Store node dimensions
3. **Incremental Updates:** Future optimization opportunity

Error Handling

Error Boundaries

React error boundaries catch rendering errors:

```
<ErrorBoundary fallback={<ErrorUI />}>
  <App />
</ErrorBoundary>
```

IPC Error Handling

All IPC operations return error states:

```
const result = await electronAPI.file.save(path, data);
if (!result.success) {
  showError(result.error);
}
```

User Feedback

- Success messages for completed operations
- Error dialogs for failures
- Confirmation dialogs for destructive actions
- Loading states for async operations

Testing Strategy

Current State

Phase 1 focuses on implementation. Testing will be added in Phase 2.

Planned Testing

1. **Unit Tests:** Jest + Testing Library for components and utilities
 2. **Integration Tests:** Test IPC communication flow
 3. **E2E Tests:** Playwright for full application testing
 4. **Type Tests:** TypeScript for compile-time verification
-

Future Architecture Improvements

Phase 2

1. **Modular Plugin System:** Allow extensions
2. **Service Workers:** Background tasks and caching
3. **Web Workers:** Offload heavy computations
4. **IndexedDB:** Local storage for large data

Phase 3

1. **Multi-Window Support:** Multiple mind maps open simultaneously
 2. **Real-Time Sync:** Collaborative editing with WebSockets
 3. **Cloud Storage:** Direct integration with cloud providers
 4. **Mobile Apps:** React Native for iOS/Android
-

Development Guidelines

Code Style

- Use TypeScript strict mode
- Follow functional programming patterns
- Prefer immutability
- Use descriptive variable names
- Add JSDoc comments for complex functions

Component Design

- Keep components small and focused
- Use composition over inheritance
- Separate logic from presentation
- Extract reusable utilities

State Management

- Keep state minimal and normalized
- Avoid derived state (compute on the fly)
- Use selectors for complex queries
- Document state shape with TypeScript

File Organization

- Group by feature, not by type
 - Keep related files together
 - Use index files for clean imports
 - Maintain consistent naming conventions
-

Deployment

Release Process

1. Update version in `package.json`
2. Update CHANGELOG.md
3. Run tests (when implemented)
4. Build and package: `npm run package`
5. Test packaged application
6. Create GitHub release
7. Upload installers
8. Publish release notes

Auto-Update (Future)

Plans to integrate electron-updater for automatic updates.

Troubleshooting

Build Issues

- Clear `node_modules` and reinstall
- Clear `dist/` directory
- Check Node.js version compatibility

Development Issues

- Restart Vite dev server
- Clear browser cache
- Check for TypeScript errors

IPC Issues

- Verify preload script is loaded
 - Check main process logs
 - Ensure handler is registered
-

Contributing

See the main README for contribution guidelines. Key points:

- Follow the existing architecture patterns
 - Maintain type safety
 - Add documentation for new features
 - Test thoroughly before submitting PRs
-

Resources

- [Electron Documentation](https://www.electronjs.org/docs) (https://www.electronjs.org/docs)
 - [React Documentation](https://react.dev/) (https://react.dev/)
 - [TypeScript Handbook](https://www.typescriptlang.org/docs/) (https://www.typescriptlang.org/docs/)
 - [Zustand Documentation](https://docs.pmnd.rs/zustand) (https://docs.pmnd.rs/zustand)
 - [Dagre Documentation](https://github.com/dagrejs/dagre) (https://github.com/dagrejs/dagre)
-

For questions or clarifications about the architecture, please open an issue on GitHub.