Technical Documentation

Architecture Overview

This productivity suite is built using modern web technologies with a focus on performance, maintainability, and user experience.

Project Structure

```
notepad-web/
├── public/                    # Static assets
│   └── vite.svg               # Vite logo
├── src/
│   ├── components/            # React components
│   │   ├── ui/                # Shadcn/ui components
│   │   │   ├── badge.tsx
│   │   │   ├── button.tsx
│   │   │   ├── card.tsx
│   │   │   ├── checkbox.tsx
│   │   │   ├── dialog.tsx
│   │   │   ├── input.tsx
│   │   │   ├── label.tsx
│   │   │   ├── progress.tsx
│   │   │   ├── select.tsx
│   │   │   ├── separator.tsx
│   │   │   ├── tabs.tsx
│   │   │   └── textarea.tsx
│   │   ├── AIChat.tsx         # AI chat interface
│   │   ├── Dashboard.tsx      # Project dashboard
│   │   ├── Editor.tsx         # Rich text editor
│   │   ├── GoalManager.tsx    # Goal management
│   │   ├── KanbanBoard.tsx    # Kanban task board
│   │   ├── Layout.tsx         # Main layout wrapper
│   │   ├── PomodoroTimer.tsx # Pomodoro timer
│   │   ├── ProjectManagement.tsx # Project management hub
│   │   ├── ProjectManager.tsx # Project CRUD operations
│   │   ├── Sidebar.tsx        # Navigation sidebar
│   │   ├── ThemeToggle.tsx    # Dark/light theme toggle
│   │   ├── TimeTracker.tsx    # Time tracking
│   │   └── TodoList.tsx       # Todo list component
│   ├── context/              # React context providers
│   │   └── NotepadContext.tsx # Global state management
│   ├── lib/                  # Utility libraries
│   │   └── utils.ts           # Helper functions
│   ├── services/             # Business logic services
│   │   ├── excelStorage.ts    # Excel export functionality
│   │   └── storage.ts        # IndexedDB storage service
│   ├── types/              # TypeScript definitions
│   │   └── index.ts          # Type definitions
│   ├── App.tsx             # Root application component
│   ├── index.css           # Global styles and Tailwind
│   └── main.tsx            # Application entry point
├── components.json         # Shadcn/ui configuration
├── package.json            # Dependencies and scripts
├── tailwind.config.js      # Tailwind CSS configuration
├── tsconfig.json           # TypeScript configuration
├── vite.config.ts          # Vite build configuration
└── README.md               # Project documentation
```

Component Architecture

Core Components

Layout.tsx

- **Purpose**: Main application layout and navigation
- **Features**:
  - Top navigation bar with module switching
  - Theme toggle integration
  - Responsive design
  - Export functionality

Editor.tsx

- **Purpose**: Rich text editing with Markdown support
- **Features**:
  - Real-time auto-save
  - Markdown rendering
  - Syntax highlighting
  - Document management

ProjectManagement.tsx

- **Purpose**: Central hub for project management features
- **Features**:
  - Tab-based navigation
  - State management for all project data
  - Integration with all project sub-components

UI Components (Shadcn/ui)

The application uses Shadcn/ui components for consistent, accessible UI:

- **Button**: Various button styles and sizes
- **Card**: Container components for content sections
- **Dialog**: Modal dialogs for forms and confirmations
- **Input/Textarea**: Form input components

- **Tabs**: Tab navigation for multi-view interfaces
- **Progress**: Progress bars for visual feedback
- **Badge**: Status and category indicators
- **Checkbox**: Form checkboxes with proper accessibility

State Management

Context API

NotepadContext

```
interface NotepadContextType {
  documents: Document[]
  currentDocument: Document | null
  todos: TodoItem[]
  // ... other state properties
}
```

Responsibilities:

- Document state management
- Todo list state
- Storage service integration
- Auto-save functionality

Local Component State

Each major component manages its own local state using React hooks:

- `useState` for component-specific data
- `useEffect` for side effects and lifecycle management
- `useCallback` for memoized functions
- `useMemo` for computed values

Data Models

Core Interfaces

Document

```
interface Document {
  id: string
  title: string
  content: string
  createdAt: Date
  updatedAt: Date
  tags: string[]
}
```

Task

```
interface Task {
  id: string
  title: string
  description?: string
  status: 'todo' | 'in_progress' | 'review' | 'completed'
  priority: 'low' | 'medium' | 'high' | 'urgent'
  category: string
  tags: string[]
  assignee?: string
  dueDate?: Date
  estimatedHours?: number
  actualHours?: number
  projectId?: string
  parentTaskId?: string
  subtasks: string[]
  createdAt: Date
  updatedAt: Date
  completedAt?: Date
}
```

Project

```
interface Project {
  id: string
  name: string
  description?: string
  status: 'planning' | 'active' | 'on_hold' | 'completed' | 'cancelled'
  priority: 'low' | 'medium' | 'high'
  startDate?: Date
  dueDate?: Date
  completedAt?: Date
  progress: number // 0-100
  color: string
  tags: string[]
  teamMembers: string[]
  createdAt: Date
  updatedAt: Date
}
```

Storage System

IndexedDB Integration

The application uses IndexedDB for client-side persistence:

```
interface StorageProvider {
  // Document operations
  saveDocument(document: Document): Promise<void>
  getDocument(id: string): Promise<Document | null>
  getAllDocuments(): Promise<Document[]>
  deleteDocument(id: string): Promise<void>

  // Task operations
  saveTask(task: Task): Promise<void>
  getAllTasks(): Promise<Task[]>
  updateTask(task: Task): Promise<void>
  deleteTask(id: string): Promise<void>

  // Project operations
  saveProject(project: Project): Promise<void>
  getAllProjects(): Promise<Project[]>
  updateProject(project: Project): Promise<void>
  deleteProject(id: string): Promise<void>

  // Additional operations for goals, categories, time entries
}
```

Excel Export

The `excelStorage.ts` service provides Excel export functionality:

- Document export to spreadsheet format
- Task and project data export
- Formatted worksheets with proper headers
- XLSX file generation using the `xlsx` library

Routing

React Router Configuration

```
<Routes>
  <Route path="/" element={<Layout />} />
  <Route path="/ai-chat" element={<AIChat />} />
  <Route path="/pomodoro" element={<PomodoroTimer />} />
  <Route path="/project-management" element={<ProjectManagement />} />
</Routes>
```

Route Structure:

- `/` - Main notepad interface

- `/ai-chat` - AI chat assistant
- `/pomodoro` - Pomodoro timer
- `/project-management` - Project management suite

Styling System

Tailwind CSS

The application uses Tailwind CSS for styling:

- Utility-first approach
- Responsive design classes
- Dark mode support with `dark:` prefix
- Custom color scheme integration

Theme System

```
:root {
  --background: 0 0% 100%;
  --foreground: 222.2 84% 4.9%;
  --primary: 222.2 47.4% 11.2%;
  /* ... other CSS variables */
}

.dark {
  --background: 222.2 84% 4.9%;
  --foreground: 210 40% 98%;
  --primary: 210 40% 98%;
  /* ... dark mode variables */
}
```

Build Configuration

Vite Configuration

```
export default defineConfig({
  plugins: [react()],
  resolve: {
    alias: {
      "@": path.resolve(__dirname, "./src"),
    },
  },
})
```

TypeScript Configuration

- Strict type checking enabled

- Path mapping for clean imports
- Modern ES target for optimal performance
- JSX support for React components

Performance Optimizations

Code Splitting

- Route-based code splitting with React.lazy
- Component-level splitting for large features
- Dynamic imports for heavy libraries

Memoization

- React.memo for component memoization
- useMemo for expensive calculations
- useCallback for stable function references

Bundle Optimization

- Tree shaking for unused code elimination
- Minification in production builds
- Asset optimization with Vite

Development Workflow

Hot Module Replacement (HMR)

- Instant updates during development
- State preservation across updates
- Fast feedback loop for development

Type Safety

- Full TypeScript coverage
- Strict type checking
- Interface-driven development

Code Quality

- ESLint for code linting
- Consistent code formatting
- Import organization

Testing Strategy

Unit Testing (Recommended)

- Jest for test runner
- React Testing Library for component testing
- Mock service implementations

Integration Testing

- End-to-end workflow testing

- Storage service integration tests
- Component interaction testing

Security Considerations

Client-Side Security

- Input sanitization for user content
- XSS prevention in Markdown rendering
- Secure storage of sensitive data

Data Privacy

- Local-only data storage
- No external data transmission
- User control over data export

Browser Compatibility

Supported Browsers

- Chrome 90+
- Firefox 88+
- Safari 14+
- Edge 90+

Progressive Enhancement

- Core functionality works without JavaScript
- Graceful degradation for older browsers
- Responsive design for all screen sizes

Deployment

Static Site Deployment

The application builds to static files suitable for:

- CDN deployment
- Static hosting services
- Traditional web servers

Environment Configuration

- Development: Hot reload, source maps
- Production: Minified, optimized bundles
- Preview: Production build with local server

Monitoring and Analytics

Performance Monitoring

- Core Web Vitals tracking
- Bundle size monitoring

- Runtime performance metrics

Error Tracking

- Client-side error boundaries
- Graceful error handling
- User-friendly error messages

Future Technical Improvements

Planned Enhancements

- Service Worker for offline functionality
- Web Workers for heavy computations
- Progressive Web App (PWA) features
- Advanced caching strategies
- Real-time collaboration infrastructure

Scalability Considerations

- Component library extraction
- Micro-frontend architecture
- API integration layer
- State management scaling (Redux/Zustand)

This technical documentation is maintained alongside the codebase and should be updated with any architectural changes.