File Explorer/Folder-Tree Navigation UI

Table of Contents

        * Performance vs Features
        * Security vs Usability
        * Scalability Considerations

Table of Contents

Clarify the Problem and Requirements

☐ Back to Top

Problem Understanding

☐ Back to Top

Design a comprehensive file explorer and folder-tree navigation interface that enables users to browse, manage, and organize files and directories efficiently, similar to Windows Explorer, macOS Finder, or cloud storage interfaces like Google Drive or Dropbox. The system must handle large directory structures, provide intuitive navigation, and support various file operations while maintaining excellent performance.

Functional Requirements

☐ Back to Top

- **Tree Navigation**: Expandable folder tree with lazy loading and infinite scroll

- **File Operations**: Create, rename, delete, copy, move, upload, download files/folders
- **Multi-view Support**: List view, grid view, thumbnail view with sorting options
- **Search & Filtering**: Real-time search, advanced filters, faceted navigation
- **File Preview**: Thumbnail generation, quick preview for images, documents, videos
- **Selection Management**: Multi-select, bulk operations, keyboard shortcuts
- **Drag & Drop**: Intuitive file/folder movement and organization
- **Context Menus**: Right-click menus with relevant actions and shortcuts

Non-Functional Requirements

☐ Back to Top

- **Performance**: <200ms directory loading, <100ms search results, smooth 60fps scrolling
- **Scalability**: Handle directories with 100K+ files, deep folder hierarchies (20+ levels)
- **Responsiveness**: Adaptive UI for desktop, tablet, and mobile devices
- **Accessibility**: Full keyboard navigation, screen reader support, WCAG 2.1 AA compliance
- **Offline Support**: Cache frequently accessed directories, offline file operations
- **Memory Efficiency**: Efficient rendering of large file lists without memory leaks
- **Cross-browser**: Consistent experience across modern browsers

Key Assumptions

☐ Back to Top

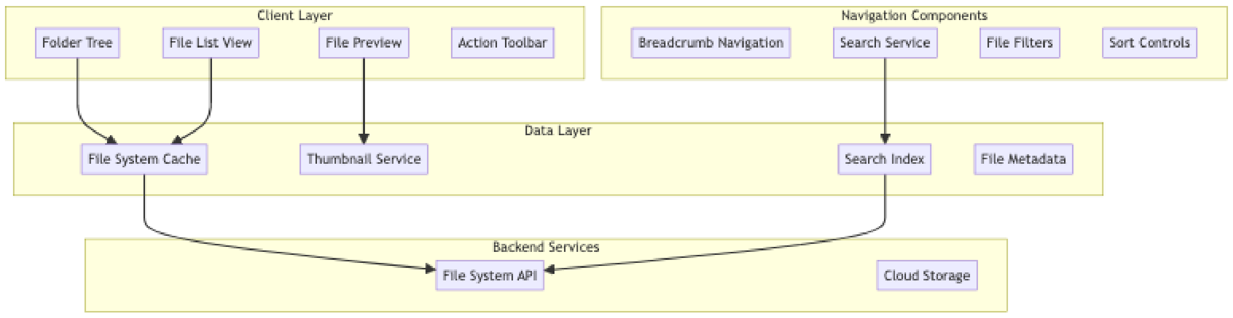
- Typical directory size: 100-1000 files, maximum 100K files per directory
- Folder depth: Average 5-10 levels, maximum 20 levels
- File types: Mixed content including documents, images, videos, archives
- User interactions: Browse-heavy with occasional file operations
- Network latency: 50-500ms depending on connection quality
- Storage backends: Local filesystem, cloud storage APIs, or network drives
- Device types: Desktop (primary), tablets and mobile (secondary)

High-Level Design (HLD)

☐ Back to Top

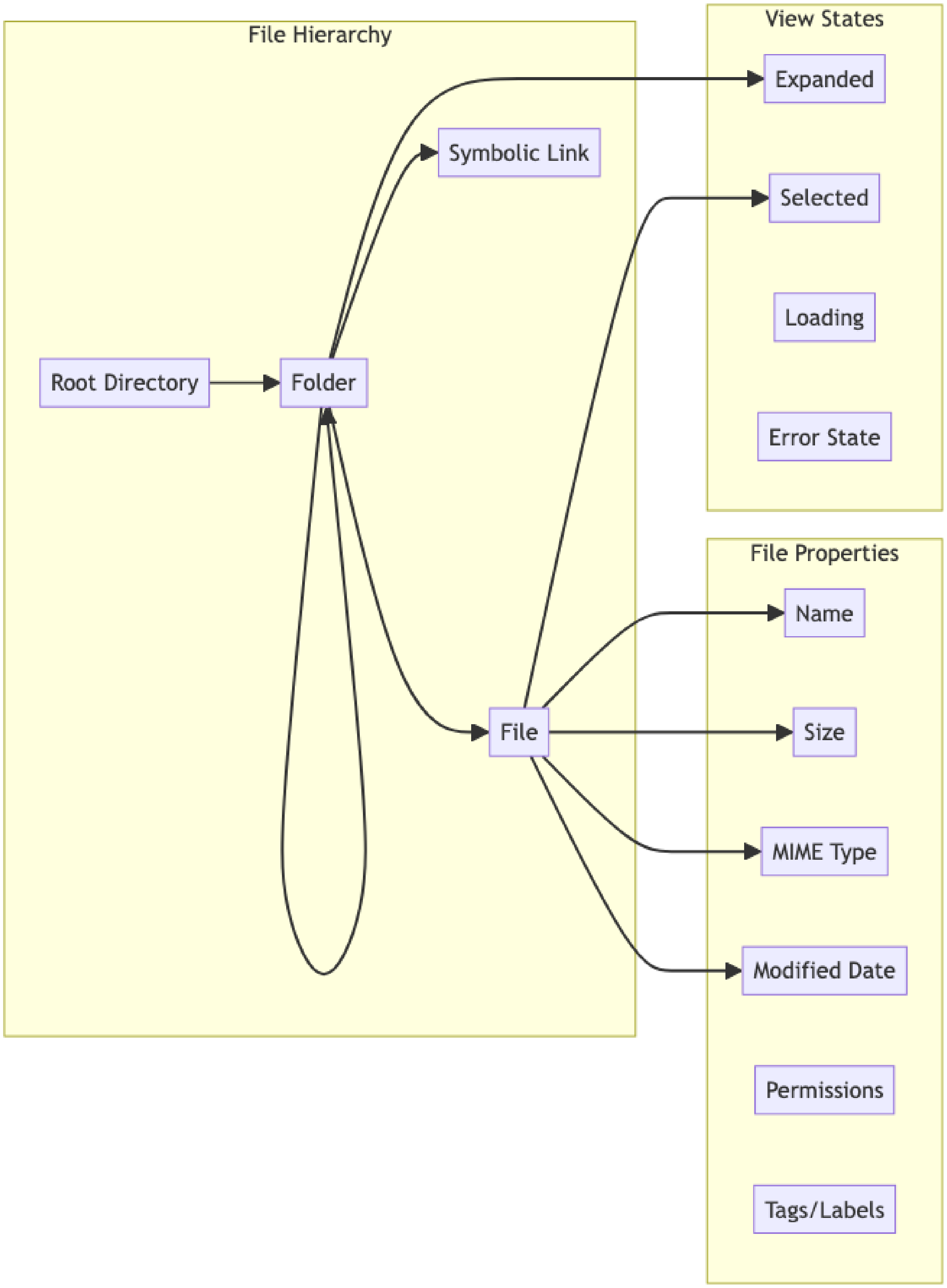
System Architecture Overview

☐ Back to Top



File System Data Model

☐ Back to Top

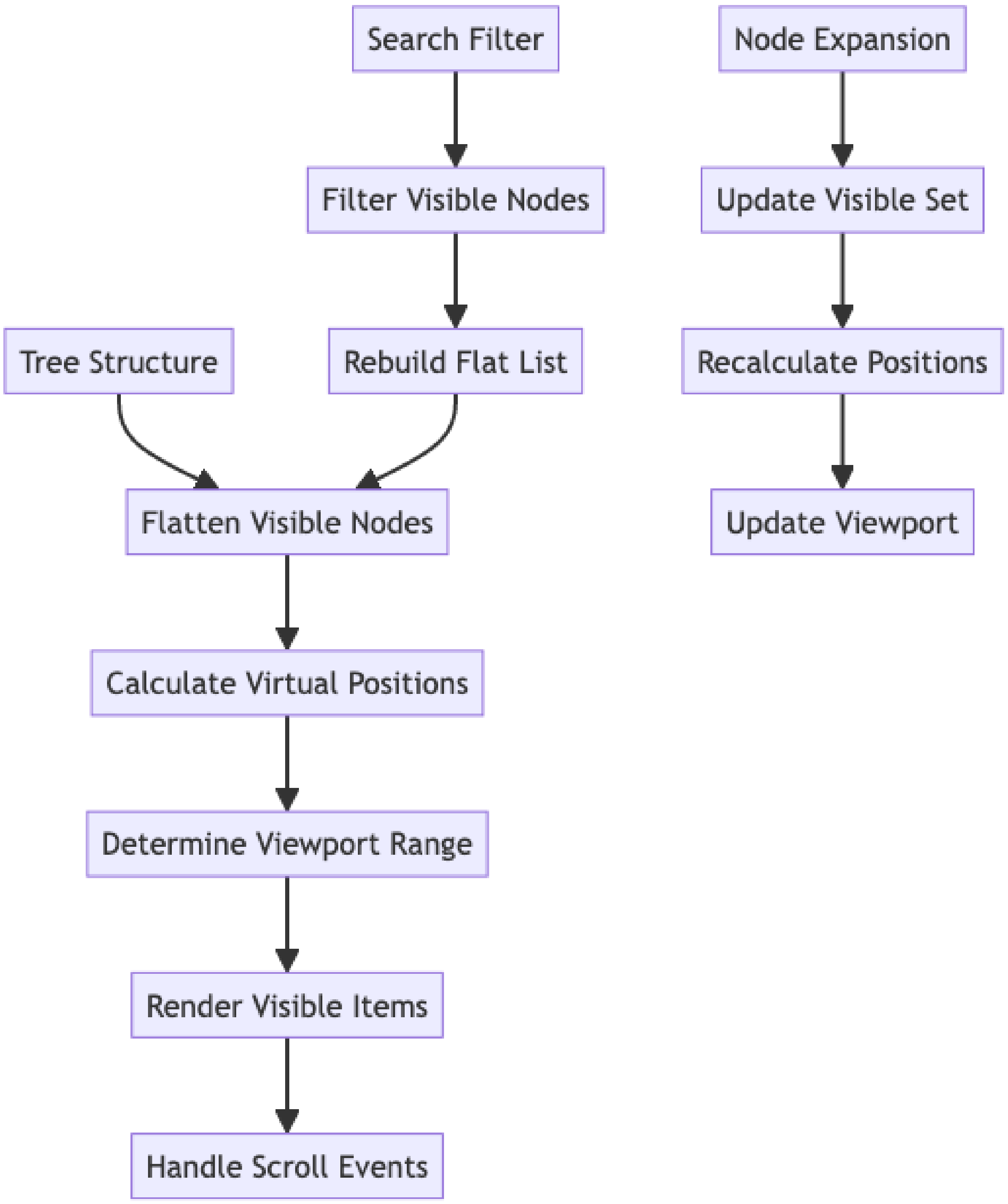

File Hierarchy
Root Directory
Folder
Symbolic Link
File
View States
Expanded
Selected
Loading
Error State
File Properties
Name
Size
MIME Type
Modified Date
Permissions
Tags/Labels

Low-Level Design (LLD)

☐ Back to Top

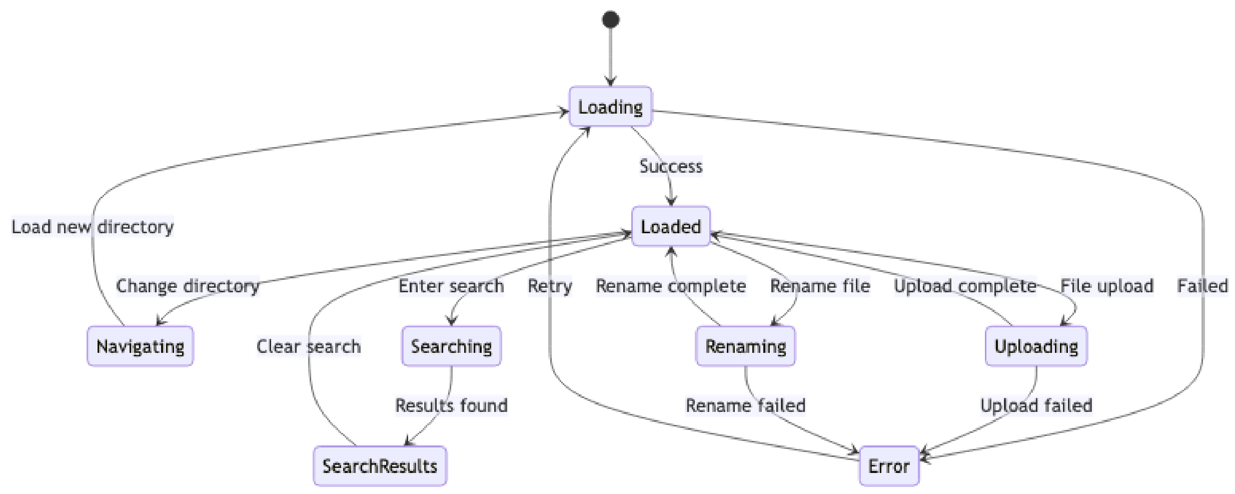
Tree Virtualization Algorithm

☐ Back to Top



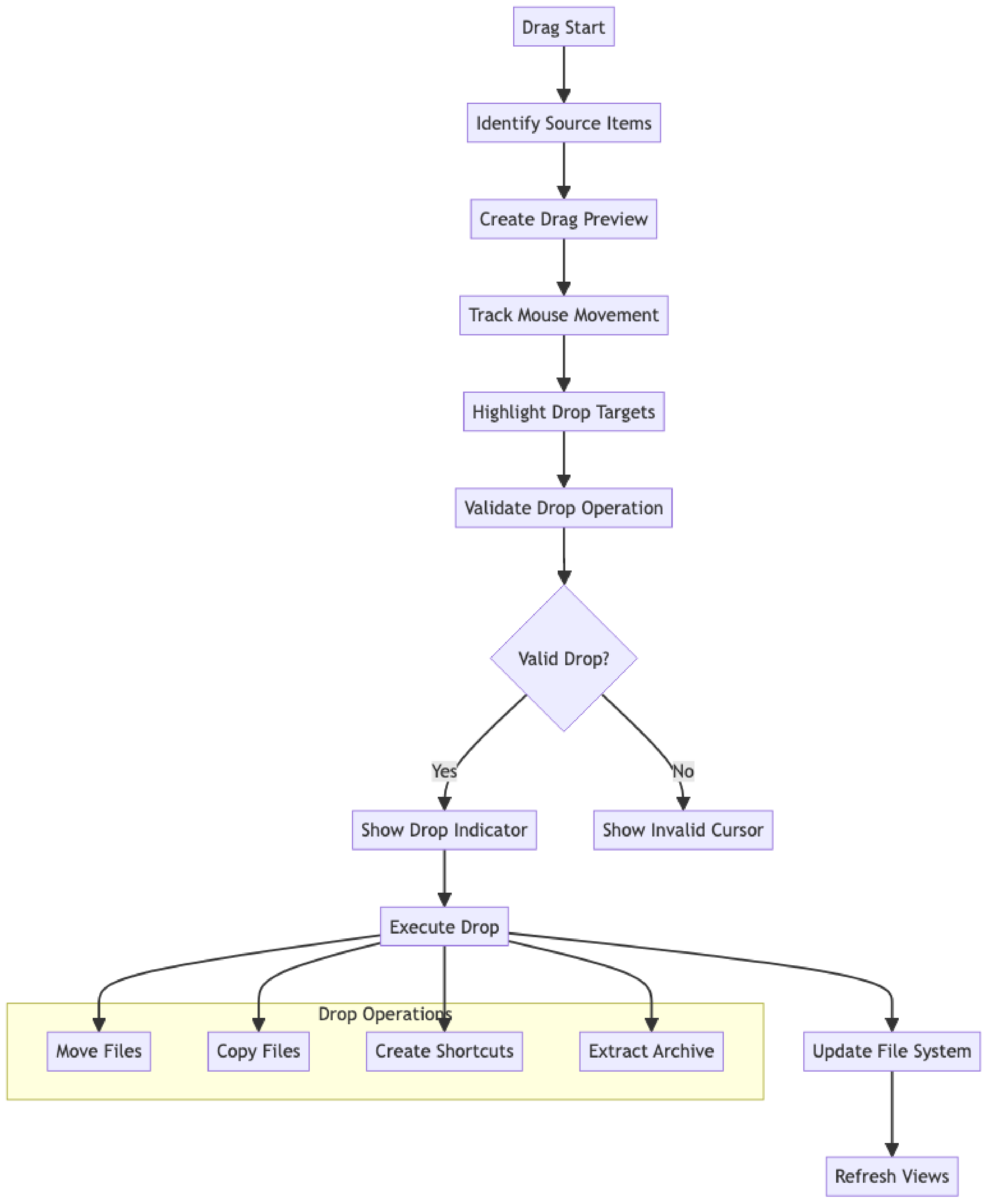
File System Navigation State Machine

☐ Back to Top



Drag and Drop File Operations

☐ Back to Top



Core Algorithms

☐ Back to Top

---

1. Tree Virtualization Algorithm

☐ Back to Top

Purpose: Efficiently render large directory trees without performance degradation.

Virtual Tree Structure:

```
VirtualTreeNode = {
  id: string,
  path: string,
  depth: number,
  isExpanded: boolean,
  isVisible: boolean,
  children?: VirtualTreeNode[],
  parent?: VirtualTreeNode
}
```

Flattening Algorithm:

```
function flattenVisibleNodes(root, filter?):
  result = []
  stack = [root]

  while stack.length > 0:
    node = stack.pop()

    if not filter or filter(node):
      result.push(node)

    if node.isExpanded and node.children:
      // Add children in reverse order for correct stack processing
      for i = node.children.length - 1; i >= 0; i--:
        stack.push(node.children[i])

  return result
```

Viewport Calculation: - Calculate scroll position offset - Determine visible item range based on item height - Add overscan buffer for smooth scrolling - Handle dynamic item heights for different file types

2. Lazy Loading Strategy

☐ Back to Top

Directory Loading Algorithm:

```
function loadDirectory(path, options):
  if cache.has(path) and not options.forceRefresh:
    return cache.get(path)

  loadingPromise = fetchDirectory(path)
    .then(files => {
      processedFiles = files.map(file => ({
        ...file,
        icon: getFileIcon(file.type),
        thumbnail: shouldGenerateThumbnail(file) ? null : undefined
      }))

      cache.set(path, processedFiles)
      return processedFiles
    })
    .catch(error => {
      cache.delete(path)
      throw error
    })

  cache.set(path, loadingPromise)
  return loadingPromise
```

Progressive Loading Strategy: - Load directory metadata first - Fetch file thumbnails asynchronously - Implement exponential backoff for failed requests - Prioritize visible items for thumbnail generation

3. File Search Algorithm

☐ Back to Top

Multi-criteria Search:

```
SearchCriteria = {
  query: string,
  fileTypes: string[],
  sizeRange: { min: number, max: number },
  dateRange: { start: Date, end: Date },
  path: string,
  recursive: boolean
}
```

Search Implementation:

```
function searchFiles(criteria):
  results = []

  if criteria.query:
    // Full-text search in file names
    nameMatches = searchIndex.queryNames(criteria.query)
    results = results.concat(nameMatches)

    // Content search for supported file types
    if canSearchContent(criteria.fileTypes):
      contentMatches = searchIndex.queryContent(criteria.query)
      results = results.concat(contentMatches)

  // Apply filters
  results = results.filter(file => {
    return matchesFileType(file, criteria.fileTypes) &&
           matchesSizeRange(file, criteria.sizeRange) &&
           matchesDateRange(file, criteria.dateRange) &&
           matchesPath(file, criteria.path)
  })

  // Sort by relevance
  return sortByRelevance(results, criteria.query)
```

Fuzzy Search Implementation: - Use Levenshtein distance for typo tolerance - Implement prefix matching for fast autocomplete - Weight matches by file type relevance - Consider file access frequency in ranking

4. Thumbnail Generation Pipeline

□ Back to Top

Thumbnail Processing:

```
ThumbnailRequest = {
  filePath: string,
  size: { width: number, height: number },
  priority: 'high' | 'normal' | 'low',
  format: 'webp' | 'jpeg' | 'png'
}
```

Generation Strategy:

```
function generateThumbnail(request):
  cacheKey = createCacheKey(request.filePath, request.size)

  if thumbnailCache.has(cacheKey):
    return thumbnailCache.get(cacheKey)

  // Check if file type supports thumbnails
  if not supportsThumbnails(request.filePath):
    return getDefaultIcon(getFileType(request.filePath))

  // Generate thumbnail based on file type
  thumbnail = await generateForFileType(request)

  // Cache with expiration
  thumbnailCache.set(cacheKey, thumbnail, TTL)

  return thumbnail
```

Supported File Types: - **Images**: Native browser support + WebGL processing - **Videos**: Canvas-based frame extraction - **Documents**: PDF.js for PDF files, Office file previews - **Code**: Syntax-highlighted previews - **Archives**: Show content summary

5. File System Operations Queue

☐ Back to Top

Operation Batching:

```
FileOperation = {
  type: 'copy' | 'move' | 'delete' | 'rename' | 'create',
  source: string | string[],
  destination?: string,
  options: OperationOptions
}
```

Queue Management:

```
function executeOperations(operations):
  queue = new OperationQueue()

  for operation in operations:
    // Group related operations
    batchKey = getBatchKey(operation)
    queue.addToBatch(batchKey, operation)

  // Execute batches in dependency order
  while queue.hasMore():
    batch = queue.getNextBatch()

    try:
      results = await executeBatch(batch)
      queue.markComplete(batch.id, results)
    catch error:
      queue.markFailed(batch.id, error)
      handleOperationError(error, batch)
```

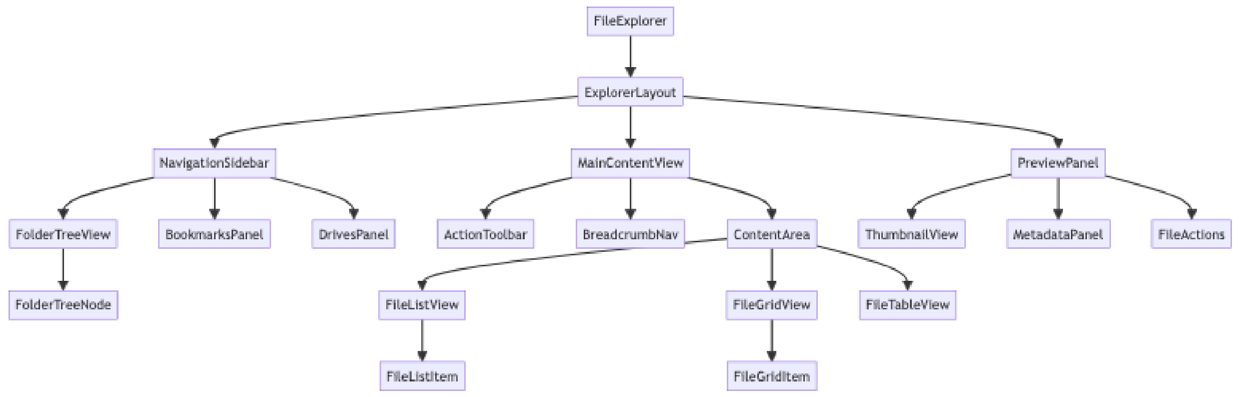
Conflict Resolution: - Detect naming conflicts before execution - Provide user choices (replace, skip, rename) - Support undo for completed operations - Handle partial failures gracefully

Component Architecture

□ Back to Top

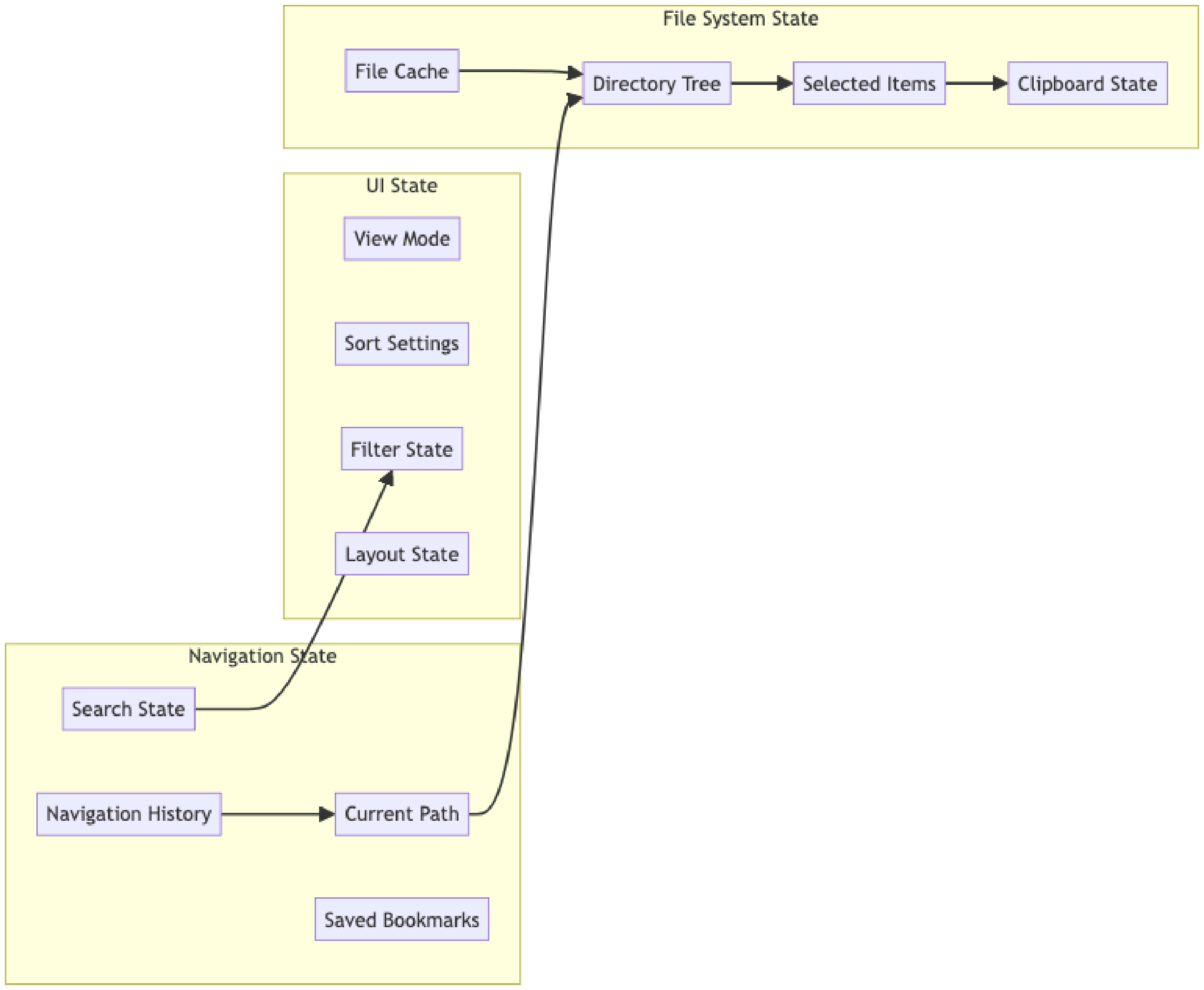
File Explorer Component Hierarchy

□ Back to Top



State Management Architecture

□ Back to Top



React Component Implementation ☐ Back to Top

FileExplorer.jsx

```
import React, { useState, useEffect, useCallback, useRef } from 'react';
import { FileSystemProvider } from './FileSystemContext';
import ExplorerLayout from './ExplorerLayout';
import { useFileSystem } from './hooks/useFileSystem';

const FileExplorer = ({ initialPath = '/' }) => {
  const [currentPath, setCurrentPath] = useState(initialPath);
  const [selectedItems, setSelectedItems] = useState([]);
  const [viewMode, setViewMode] = useState('list'); // 'list', 'grid', 'table'
  const [sortBy, setSortBy] = useState('name');
  const [sortOrder, setSortOrder] = useState('asc');
  const [searchQuery, setSearchQuery] = useState('');
  const [directoryTree, setDirectoryTree] = useState({});
  const [fileCache, setFileCache] = useState(new Map());
  const [clipboard, setClipboard] = useState({ items: [], operation: null }); // 'cut'
  const [history, setHistory] = useState({ back: [], forward: [] });

  const {
    listDirectory,
    createFolder,
    deleteItems,
    moveItems,
    copyItems,
    renameItem,
    getFileInfo
  } = useFileSystem();

  useEffect(() => {
    loadDirectory(currentPath);
  }, [currentPath]);

  const loadDirectory = async (path) => {
    try {
      const items = await listDirectory(path);
      setFileCache(prev => new Map(prev).set(path, items));

      // Update directory tree
      updateDirectoryTree(path, items.filter(item => item.type === 'directory'));
    } catch (error) {
      console.error('Failed to load directory:', error);
    }
  };

  const updateDirectoryTree = (path, directories) => {
    setDirectoryTree(prev => ({
      ...prev,
      [path]: directories.map(dir => ({
        ...dir,
        path: `${path}/${dir.name}`.replace(/\/+/g, '/')
      }))
    }));
  };

  const navigateTo = useCallback((newPath) => {
    if (newPath !== currentPath) {
      setHistory(prev => ({
        back: [...prev.back, currentPath],
        forward: []
      }));
      setCurrentPath(newPath);
      setSelectedItems([]);
    }
  }, [currentPath]);

  const navigateBack = useCallback(() => {
    if (history.back.length > 0) {
      const previousPath = history.back[history.back.length - 1];
      setHistory(prev => ({
        back: prev.back.slice(0, -1),
        forward: [currentPath, ...prev.forward]
      }));
      setCurrentPath(previousPath);
    }
  }, [history.back, currentPath]);

  const navigateForward = useCallback(() => {
    if (history.forward.length > 0) {
      const nextPath = history.forward[0];
      setHistory(prev => ({
        back: [...prev.back, currentPath],
        forward: prev.forward.slice(1)
      }));
      setCurrentPath(nextPath);
    }
  }, [history.forward, currentPath]);

  const handleItemSelect = useCallback((item, isMultiSelect = false) => {
    if (isMultiSelect) {
      setSelectedItems(prev => {
        const isSelected = prev.some(selected => selected.path === item.path);
        if (isSelected) {
          return prev.filter(selected => selected.path !== item.path);
        } else {
          return [...prev, item];
        }
      });
    } else {
      setSelectedItems([item]);
    }
  }, []);

  const handleItemDoubleClick = useCallback((item) => {
    if (item.type === 'directory') {
      navigateTo(item.path);
    } else {
      // Open file (could trigger preview or download)
      window.open(`/api/files/download?path=${encodeURIComponent(item.path)}`);
    }
  }, [navigateTo]);

  const handleCreateFolder = useCallback(async (name) => {
    try {
      const newFolderPath = `${currentPath}/${name}`.replace(/\/+/g, '/');
      await createFolder(newFolderPath);
      loadDirectory(currentPath);
    } catch (error) {
      console.error('Failed to create folder:', error);
    }
  }, [currentPath, createFolder]);

  const handleDelete = useCallback(async (items = selectedItems) => {
    try {
      await deleteItems(items.map(item => item.path));
      loadDirectory(currentPath);
      setSelectedItems([]);
    } catch (error) {
      console.error('Failed to delete items:', error);
    }
  }, [selectedItems, currentPath, deleteItems]);

  const handleCopy = useCallback(() => {
    setClipboard({
      items: selectedItems,
      operation: 'copy'
    });
  }, [selectedItems]);

  const handleCut = useCallback(() => {
    setClipboard({
      items: selectedItems,
      operation: 'cut'
    });
  }, [selectedItems]);

  const handlePaste = useCallback(async () => {
    if (clipboard.items.length === 0) return;
    try {
      const destinationPath = currentPath;
      const sourcePaths = clipboard.items.map(item => item.path);

      if (clipboard.operation === 'copy') {
        await copyItems(sourcePaths, destinationPath);
      } else if (clipboard.operation === 'cut') {
        await moveItems(sourcePaths, destinationPath);
        setClipboard({ items: [], operation: null });
      }

      loadDirectory(currentPath);
    } catch (error) {
      console.error('Failed to paste items:', error);
    }
  }, [clipboard, currentPath, copyItems, moveItems]);

  const handleRename = useCallback(async (item, newName) => {
    try {
      const newPath = `${item.path.split('/').slice(0, -1).join('/')}/${newName}`;
      await renameItem(item.path, newPath);
      loadDirectory(currentPath);
    } catch (error) {
      console.error('Failed to rename item:', error);
    }
  }, [currentPath, renameItem]);

  const getCurrentDirectoryItems = () => {
    const items = fileCache.get(currentPath) || [];

    let filteredItems = items;

    // Apply search filter
    if (searchQuery) {
      filteredItems = items.filter(item =>
        item.name.toLowerCase().includes(searchQuery.toLowerCase())
      );
    }

    // Apply sorting
    filteredItems.sort((a, b) => {
      let comparison = 0;

      // Directories first
      if (a.type === 'directory' && b.type !== 'directory') return -1;
      if (a.type !== 'directory' && b.type === 'directory') return 1;

      switch (sortBy) {
        case 'name':
          comparison = a.name.localeCompare(b.name);
          break;
        case 'size':
          comparison = (a.size || 0) - (b.size || 0);
          break;
        case 'modified':
          comparison = new Date(a.modified) - new Date(b.modified);
          break;
        default:
          comparison = a.name.localeCompare(b.name);
      }

      return sortOrder === 'asc' ? comparison : -comparison;
    });

    return filteredItems;
  };

  const value = {
    currentPath,
    selectedItems,
    viewMode,
    sortBy,
    sortOrder,
    searchQuery,
    directoryTree,
    clipboard,
    history,
    currentItems: getCurrentDirectoryItems(),
    navigateTo,
    navigateBack,
    navigateForward,
    onItemSelect: handleItemSelect,
    onItemDoubleClick: handleItemDoubleClick,
    onCreateFolder: handleCreateFolder,
    onDelete: handleDelete,
    onCopy: handleCopy,
    onCut: handleCut,
    onPaste: handlePaste,
    onRename: handleRename,
    setViewMode,
    setSortBy,
    setSortOrder,
    setSearchQuery
  };

  return (
    <FileSystemProvider value={value}>
      <div className="file-explorer">
        <ExplorerLayout />
      </div>
    </FileSystemProvider>
  );
};

export default FileExplorer;
```

ExplorerLayout.jsx

```
import React, { useContext } from 'react';
import { FileSystemContext } from './FileSystemContext';
import NavigationSidebar from './NavigationSidebar';
import MainContentView from './MainContentView';
import PreviewPanel from './PreviewPanel';

const ExplorerLayout = () => {
  const { selectedItems } = useContext(FileSystemContext);

  return (
    <div className="explorer-layout">
      <div className="sidebar-section">
        <NavigationSidebar />
      </div>

      <div className="main-section">
        <MainContentView />
      </div>

      {selectedItems.length === 1 && (
        <div className="preview-section">
          <PreviewPanel item={selectedItems[0]} />
        </div>
      )}
    </div>
  );
};
```

```
export default ExplorerLayout;
```

MainContentView.jsx

```
import React, { useContext } from 'react';
import { FileSystemContext } from './FileSystemContext';
import ActionToolbar from './ActionToolbar';
import BreadcrumbNav from './BreadcrumbNav';
import FileListView from './FileListView';
import FileGridView from './FileGridView';
import FileTableView from './FileTableView';

const MainContentView = () => {
  const { viewMode, currentItems } = useContext(FileSystemContext);

  const renderContentView = () => {
    switch (viewMode) {
      case 'grid':
        return <FileGridView items={currentItems} />;
      case 'table':
        return <FileTableView items={currentItems} />;
      case 'list':
      default:
        return <FileListView items={currentItems} />;
    }
  };

  return (
    <div className="main-content-view">
      <ActionToolbar />
      <BreadcrumbNav />

      <div className="content-area">
        {renderContentView()}
      </div>
    </div>
  );
};

export default MainContentView;
```

FileListView.jsx

```
import React, { useContext } from 'react';
import { FixedSizeList as VirtualList } from 'react-window';
import { FileSystemContext } from './FileSystemContext';
import FileListItem from './FileListItem';
```

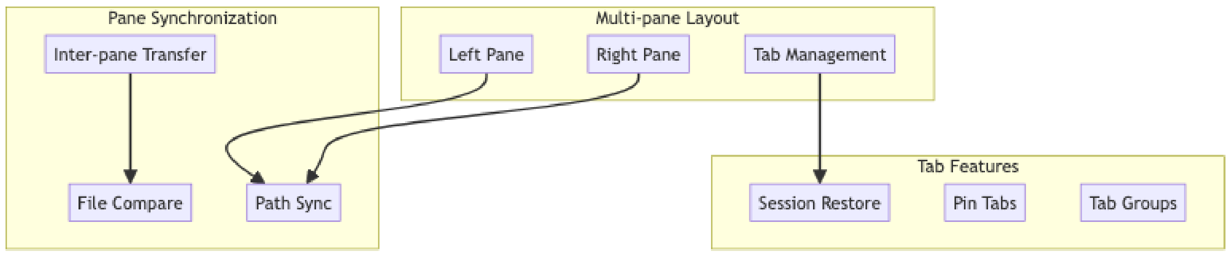
```
const FileListView = ({ items }) => {
  const { onItemSelect, onItemDoubleClick } = useContext(FileSystemContext);

  const ItemRenderer = ({ index, style }) => (
    <div style={style}>
      <FileListItem
        item={items[index]}
        onSelect={onItemSelect}
        onDoubleClick={onItemDoubleClick}
      />
    </div>
  );

  return (
    <div className="file-list-view">
      {items.length === 0 ? (
        <div className="empty-directory">
          <div className="empty-state">
            <h3>This folder is empty</h3>
            <p>Drag files here or create a new folder</p>
          </div>
        </div>
      ) : (
        <VirtualList
          height={600}
          itemCount={items.length}
          itemSize={40}
          overscanCount={10}
        >
          {ItemRenderer}
        </VirtualList>
      )}
    </div>
  );
};

export default FileListView;
```

Advanced Features

☐ Back to Top

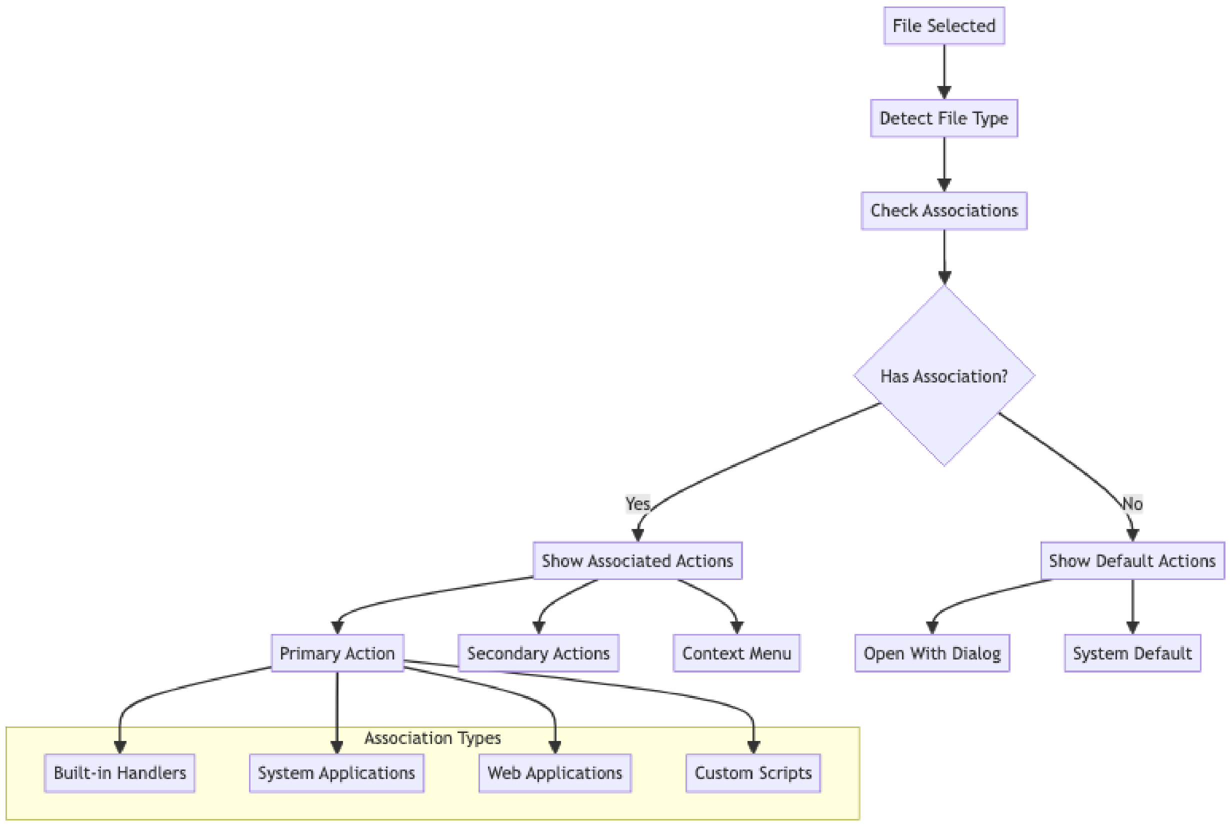
Multi-pane Navigation

☐ Back to Top



File Type Associations

☐ Back to Top



TypeScript Interfaces & Component Props

☐ Back to Top

Core Data Interfaces

```
interface FileSystemItem {
  id: string;
  name: string;
  type: 'file' | 'folder';
  path: string;
  parentId?: string;
  size: number;
  mimeType?: string;
  createdAt: Date;
  modifiedAt: Date;
  permissions: FilePermissions;
  metadata: FileMetadata;
  isHidden: boolean;
}

interface FilePermissions {
  read: boolean;
  write: boolean;
  execute: boolean;
  owner: string;
  group?: string;
  public?: boolean;
}

interface FileMetadata {
  checksum?: string;
  encoding?: string;
  thumbnailUrl?: string;
  previewUrl?: string;
  tags?: string[];
  description?: string;
  version?: number;
}

interface FolderNode {
  item: FileSystemItem;
  children?: FolderNode[];
  isExpanded: boolean;
  isLoading: boolean;
  hasMoreChildren: boolean;
  depth: number;
}
```

```
interface FileSystemState {
  currentPath: string;
  selectedItems: string[];
  clipboard: ClipboardItem[];
  viewMode: 'list' | 'grid' | 'tree';
  sortBy: SortOption;
  showHidden: boolean;
  searchQuery: string;
}
```

Component Props Interfaces

```
interface FileExplorerProps {
  rootPath?: string;
  onFileSelect: (file: FileSystemItem) => void;
  onFolderChange: (path: string) => void;
  onFileAction: (action: string, items: FileSystemItem[]) => void;
  allowMultiSelect?: boolean;
  allowUpload?: boolean;
  allowDelete?: boolean;
  showPreview?: boolean;
  customActions?: FileAction[];
}

interface FileTreeProps {
  nodes: FolderNode[];
  selectedPath?: string;
  onNodeSelect: (node: FolderNode) => void;
  onNodeExpand: (nodeId: string) => void;
  onNodeCollapse: (nodeId: string) => void;
  virtualScrolling?: boolean;
  maxDepth?: number;
  showIcons?: boolean;
}

interface FileListProps {
  items: FileSystemItem[];
  selectedIds: string[];
  onItemSelect: (item: FileSystemItem) => void;
  onItemDoubleClick: (item: FileSystemItem) => void;
  onSelectionChange: (selectedIds: string[]) => void;
  viewMode: 'list' | 'grid';
  sortBy: SortOption;
  showThumbnails?: boolean;
}
```

```
interface FileUploadProps {
  onUpload: (files: File[], targetPath: string) => void;
  onProgress: (progress: UploadProgress) => void;
  allowedTypes?: string[];
  maxFileSize?: number;
  maxFiles?: number;
  showProgress?: boolean;
  dragAndDrop?: boolean;
}
```

API Reference

☐ Back to Top

File System Navigation

- `GET /api/files/browse` - Browse directory contents with pagination and filtering
- `GET /api/files/tree` - Get folder tree structure with lazy loading support
- `GET /api/files/search` - Search files and folders with advanced criteria
- `GET /api/files/:id/info` - Get detailed file information and metadata
- `POST /api/files/recent` - Get recently accessed files and folders

File Operations

- `POST /api/files/upload` - Upload files with progress tracking and validation
- `GET /api/files/:id/download` - Download file with resume capability
- `PUT /api/files/:id/rename` - Rename file or folder with conflict detection
- `DELETE /api/files/:id` - Delete file or folder (move to trash or permanent)
- `POST /api/files/copy` - Copy files and folders to target location

Folder Management

- `POST /api/folders` - Create new folder with permissions and metadata
- `PUT /api/folders/:id/move` - Move folder to different location in hierarchy
- `GET /api/folders/:id/size` - Calculate total size of folder and contents
- `POST /api/folders/:id/compress` - Create archive of folder contents
- `POST /api/folders/:id/extract` - Extract archive to target folder

File Preview & Thumbnails

- `GET /api/files/:id/preview` - Get file preview for supported formats
- `GET /api/files/:id/thumbnail` - Get generated thumbnail image
- `POST /api/files/:id/generate-preview` - Generate preview for file type

- `GET /api/files/:id/metadata` - Extract and return file metadata
- `GET /api/files/:id/content` - Get file content for text-based files

Permissions & Sharing

- `GET /api/files/:id/permissions` - Get file or folder permission settings
- `PUT /api/files/:id/permissions` - Update access permissions for item
- `POST /api/files/:id/share` - Create shareable link with expiration
- `GET /api/files/shared` - Get files shared with current user
- `DELETE /api/files/:id/share` - Revoke sharing access for file

File System Operations

- `POST /api/files/batch` - Execute multiple file operations in single request
- `GET /api/files/stats` - Get file system usage statistics
- `POST /api/files/backup` - Create backup of selected files and folders
- `GET /api/files/versions` - Get version history for file
- `POST /api/files/:id/restore` - Restore file from version history

Search & Indexing

- `POST /api/search/index` - Trigger file content indexing for search
- `GET /api/search/suggestions` - Get search suggestions based on content
- `POST /api/search/advanced` - Advanced search with multiple criteria
- `GET /api/search/saved` - Get saved search queries and filters
- `POST /api/search/tag` - Add or remove tags from files for organization

Performance Optimizations

☐ Back to Top

Memory Management

☐ Back to Top

File Cache Strategy:

```
FileCache = {
  metadata: LRU<string, FileMetadata>,
  thumbnails: LRU<string, Blob>,
  directoryContents: Map<string, FileEntry[]>,
  searchResults: TTLCache<string, SearchResult[]>
}
```

Optimization Techniques: - Implement lazy loading for off-screen items - Use weak references for large file previews - Compress cached directory listings - Implement garbage collection for unused cache entries - Store frequently accessed paths in persistent storage

Rendering Optimizations

☐ Back to Top

Virtual Scrolling Implementation: - Calculate visible range based on scroll position - Implement predictive loading for smooth scrolling - Use CSS transforms for position updates - Batch DOM updates for better performance - Implement intersection observer for visibility detection

Image Optimization: - Generate multiple thumbnail sizes - Use progressive JPEG for large images - Implement WebP support with fallbacks - Lazy load thumbnails based on visibility - Use image sprites for file type icons

Network Optimization

☐ Back to Top

Request Batching:

```
RequestBatcher = {
  pendingRequests: Map<string, Promise>,
  batchSize: number,
  batchTimeout: number,
  queue: RequestQueue
}
```

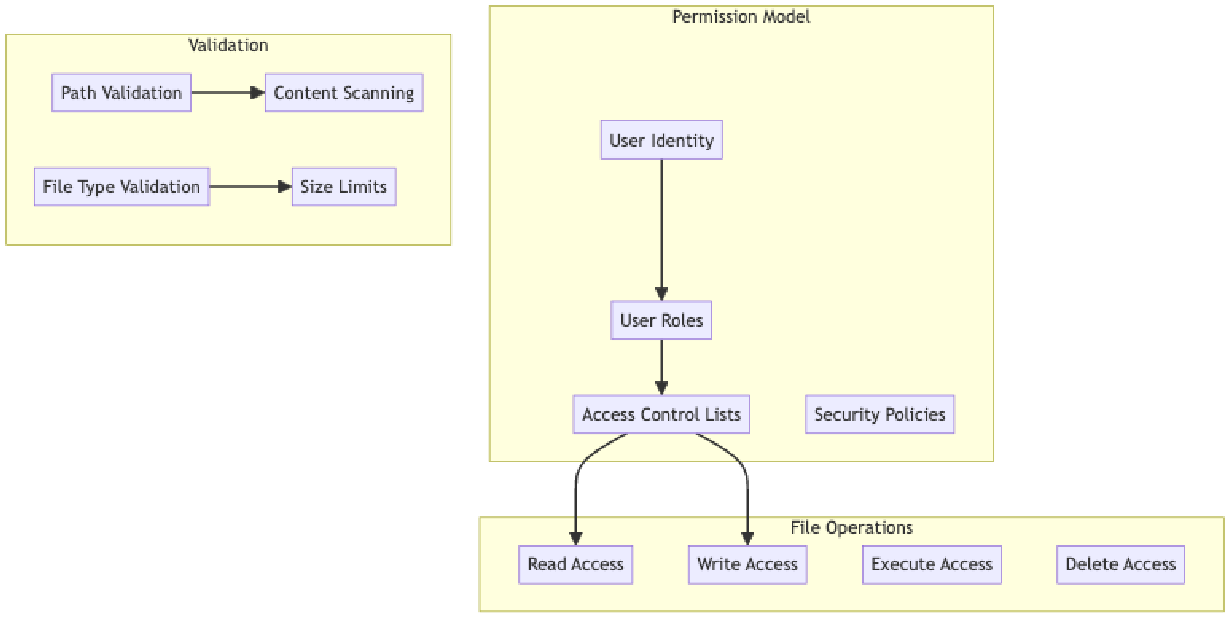
Optimization Strategies: - Batch file metadata requests - Use HTTP/2 multiplexing for concurrent requests - Implement request deduplication - Cache directory listings with ETags - Use compression for large file lists

Security Considerations

☐ Back to Top

Access Control

☐ Back to Top



Input Validation

☐ Back to Top

Path Sanitization: - Validate file paths for directory traversal attacks - Normalize path separators across platforms - Restrict access to system directories - Implement path length limits - Validate Unicode characters in filenames

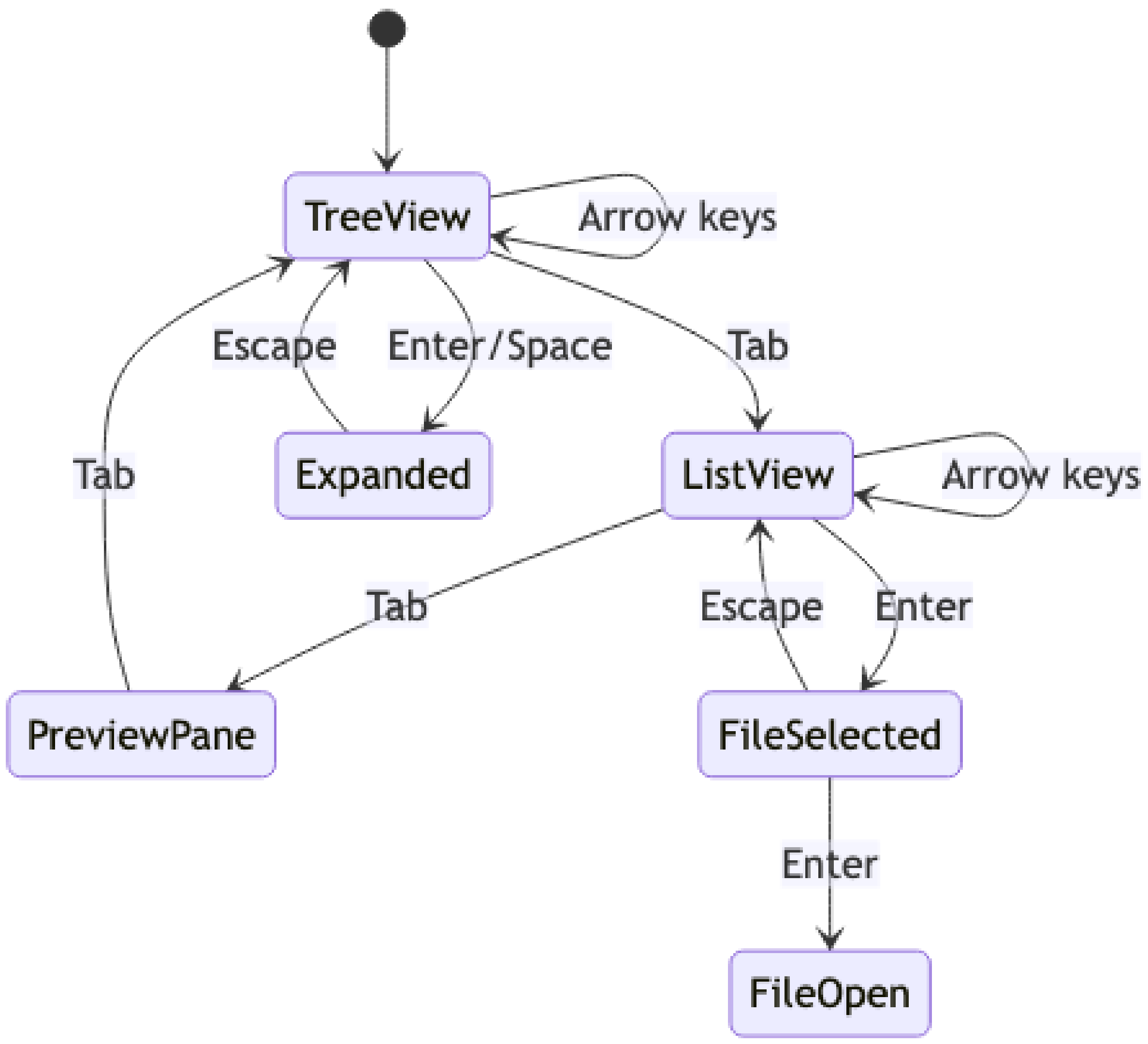
File Type Security: - Validate file types by content, not extension - Block execution of dangerous file types - Scan uploaded files for malware - Implement content security policies - Use sandboxed previews for unknown file types

Accessibility Implementation

☐ Back to Top

Keyboard Navigation

☐ Back to Top



Accessibility Features: - Full keyboard navigation support - Screen reader announcements for file operations - High contrast mode for file type icons - Focus indicators for all interactive elements - ARIA labels for complex tree structures

Screen Reader Support

 Back to Top

File Announcement Pattern:

```
"File 1 of 50, document.pdf, PDF file, 2.3 MB,
modified yesterday, press Enter to open,
Space to select, F2 to rename"
```

Navigation Landmarks: - Navigation region for folder tree - Main region for file list - Complementary region for file preview - Search region for file search functionality

Testing Strategy

☐ Back to Top

Unit Testing Focus Areas

☐ Back to Top

Core Algorithm Testing: - Tree virtualization accuracy - File operation queue management - Search algorithm correctness - Cache invalidation logic

Component Testing: - File list rendering performance - Drag and drop interactions - Tree expansion and collapse - File selection mechanisms

Integration Testing

☐ Back to Top

File System Integration: - Cross-platform file operations - Network file system support - Permission handling - Error recovery mechanisms

Performance Testing: - Large directory handling - Concurrent file operations - Memory usage patterns - Network efficiency

End-to-End Testing

☐ Back to Top

User Workflow Testing: - Complete file management scenarios - Multi-platform compatibility - Touch and mouse interaction - Keyboard-only navigation

Trade-offs and Considerations

☐ Back to Top

Performance vs Features

☐ Back to Top

- **Real-time updates**: File system watching vs battery usage
- **Thumbnail quality**: Image quality vs loading time
- **Search capability**: Index size vs search speed
- **Cache strategy**: Memory usage vs response time

Security vs Usability

☐ Back to Top

- **File access**: Security restrictions vs user convenience
- **Preview generation**: Safety vs functionality
- **Path validation**: Security vs flexibility
- **File operations**: Safety checks vs operation speed

Scalability Considerations

☐ Back to Top

- **Directory size**: Performance with large directories
- **File count**: Memory usage vs file system size
- **Network latency**: Remote file systems vs responsiveness
- **Concurrent users**: Multi-user access vs consistency

This file explorer system provides a comprehensive foundation for modern file management with advanced features like virtualized rendering, intelligent search, and robust file operations while maintaining high performance and accessibility standards.