REPORT - PRODUCT FEATURE SITEMAP

Nguyen Lam Anh Duy

June 28, 2025

Contents

1. Introduction 4
2. Data structure 5
    2.1. Explanation 5
        2.1.1. Export/Import function 5
    2.2. Suggest improvement 6
3. Node management 7
    3.1. DFS algorithms 7
    3.2. Explanation for each function 7
        3.2.1. generateId() 7
        3.2.2. findNode(tree, targetId) 7
        3.2.3. findParent(tree, targetId, parent = null) 7
        3.2.4. addNode(tree, parentId, newNode) 7
        3.2.5. updateNode(tree, nodeId, updates) 8
        3.2.6. deleteNode(tree, targetId) 9
        3.2.7. toggleExpanded(tree, nodeId) 9
        3.2.8. deepClone(obj) 10
    3.3. Suggested improvement for algorithm 10
4. UI interaction 11
    4.1. Canvas layout & navigation 11
    4.2. Expand/Collapse functionality 11
    4.3. Node interaction 12
    4.4. Key state variables and refs 12
    4.5. Key functions explained 13
        4.5.1. Viewport Dimension Handling 13
        4.5.2. Layout Calculation and Memoization 13
        4.5.3. Node Rendering 14
        4.5.4. Connection Rendering 14
        4.5.5. Auto Fit-to-View (on tree change) 14
        4.5.6. Zoom Controls (Wheel, Buttons, Reset) 15
        4.5.7. Mouse Event Listeners for Canvas Interaction 15
5. Sitemap search functionality 17
    5.1. Overview 17
    5.2. Search logic 17
    5.3. Debouncing 17
        5.3.1. Synonym dictionary 18
        5.3.2. Exact word matching 18
        5.3.3. Synonym matching 18
        5.3.4. Recursive tree search 18
        5.3.5. Highlighting and output 18
        5.3.6. Suggested improvement 19
6. Use case management 21
    6.1. Overview 21
    6.2. Functional Description 21

6.2.1. getDefaultUseCases() 21
6.2.2. extractUseCasesFromTree(tree) 21
6.3. initializeUseCases(tree) 21
6.3.1. searchUseCases(allUseCases, query) 21
6.4. UI integration for use case selection 22
6.4.1. Use case selection modal 22
6.4.2. Use case input with autocomplete 23
7. Summary 24
Index of Figures 25

1. Introduction

This report presents the design and implementation of a web-based tool that allows users to create, manage, and visualize a hierarchical product feature sitemap. The goal of this application is to help product teams and stakeholders better understand the overall structure and scope of product functionality through an intuitive, mind-map-style interface.

The system supports:

- Full-featured **tree-based data modeling**, with up to 5+ levels of nested nodes.
- **Node management** operations including creation, editing, and deletion.
- An interactive **canvas-based visualization** with zoom, pan, expand/collapse, and detail inspection.
- An enhanced **search mechanism** that supports both exact term matches and synonym-based querying.
- A reusable **use case management** system that enables consistency in feature documentation.

The tool is designed with usability, scalability, and clarity in mind—providing an organized way to link user pain points to relevant solutions by enabling precise, categorized, and searchable feature maps.

The following sections document each module of the system: data structure design, feature CRUD operations, UI/UX interactions, search logic, and use case reuse logic.

2. Data structure

2.1. Explanation

The sitemap is implemented as a hierarchical tree structure stored in a JavaScript object named `initialTree`. Each node in this tree represents either a feature or a feature group and contains the following information:

- `id` (string): Unique identifier of the node.
- `name` (string): The feature name. This field is mandatory.
- `description` (string, optional): A text field explaining the feature.
- `useCases` (array of strings, optional): Lists common use cases related to the feature.
- `valueProposition` (string, optional): Describes the added value the feature brings to the system.
- `children` (array of nodes, optional): Allows recursive nesting to represent sub-features or related components.

The design supports at least 5 levels of nesting and can be extended further if needed. This allows the application to organize complex systems with multiple levels of functional grouping.

In the React application, the tree is stored in a local state variable using the following hook:

```
const [tree, setTree] = useState(initialTree)
```

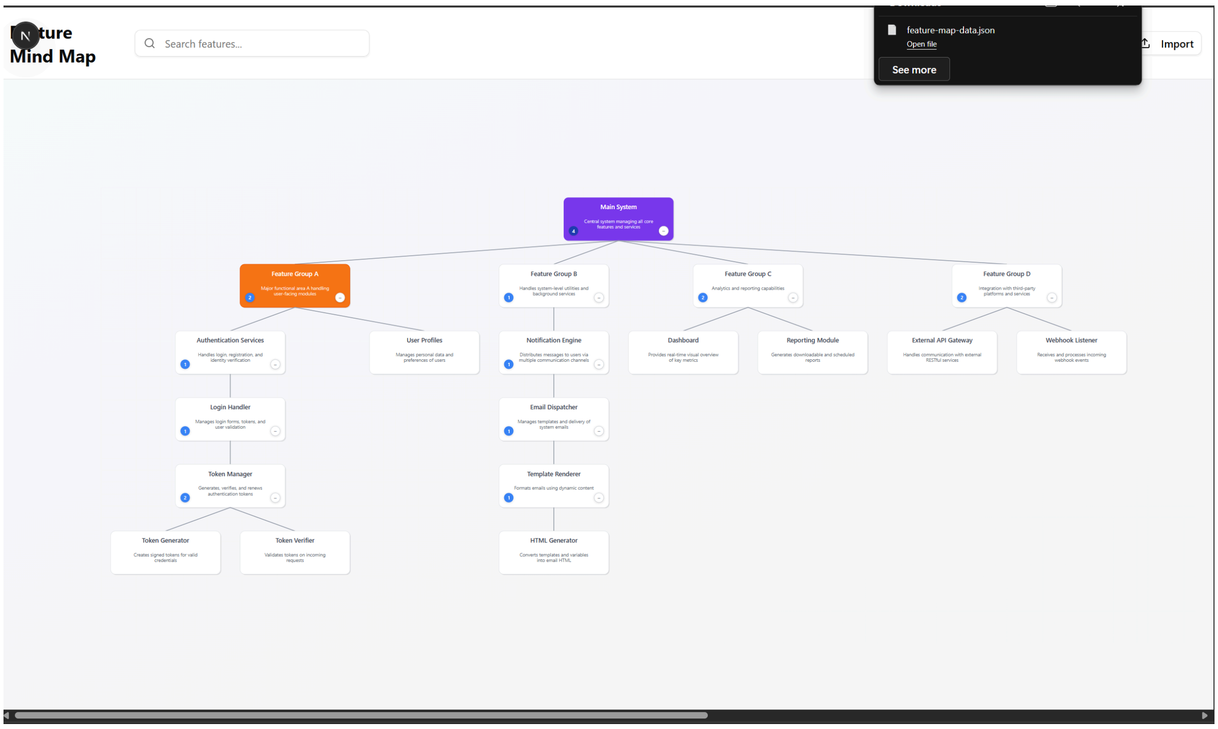
This approach enables real-time updates to the tree in the client-side interface. However, due to limited time and scope, no backend or persistent database has been implemented. As a result, any changes made to the tree are lost upon page refresh.

2.1.1. Export/Import function

To enable manual data portability, the application includes JSON-based export and import functionality:

- Export: When the user clicks the export button, the system generates a .json file that contains both the current tree and the list of allUseCases. The structure is serialized using JSON.stringify() and downloaded using a Blob and temporary URL.

- Import: Users can import a previously exported file. The system parses the file using FileReader and restores both the tree and use cases into local state. It also supports older formats that contain only the tree structure.


Feature Mind Map
Search features...
feature-map-data.json
Open file
See more
Import
Main System
Central system managing all core features and services
4
Feature Group A
Major functional area A handling user-facing modules
2
Feature Group B
Handles system-level utilities and background services
1
Feature Group C
Analytics and reporting capabilities
2
Feature Group D
Integration with third-party platforms and services
2
Authentication Services
Handles login, registration, and identity verification
1
User Profiles
Manages personal data and preferences of users
Notification Engine
Distributes messages to users via multiple communication channels
1
Dashboard
Provides real-time visual overview of key metrics
Reporting Module
Generates downloadable and scheduled reports
External API Gateway
Handles communication with external RESTful services
Webhook Listener
Receives and processes incoming webhook events
Login Handler
Manages login forms, tokens, and user validation
1
Email Dispatcher
Manages templates and delivery of system emails
1
Token Manager
Generates, verifies, and renews authentication tokens
2
Template Renderer
Formats emails using dynamic content
1
Token Generator
Creates signed tokens for valid credentials
Token Verifier
Validates tokens on incoming requests
HTML Generator
Converts templates and variables into email HTML

2.2. Suggest improvement

To enhance the current system, the following improvements are recommended:

- **Persistent Storage**: Store the tree structure in a backend database or a browser storage mechanism (e.g., localStorage or IndexedDB) to prevent data loss on reload.
- **Node Type Field**: Introduce a type field for each node (e.g., “feature-group”, “component”, “module”) to help with rendering and filtering logic in the UI.
- **Modular Definition**: Define reusable constructors or types for nodes to improve code maintainability and clarity.
- **UI Metadata**: Add optional UI-specific metadata such as icon, color to support future visual features like tree styling and custom behaviors.

⟶ These changes would make the data structure more robust, scalable, and user-friendly, especially for large and complex sitemaps.

3. Node management

3.1. DFS algorithms

All node-related operations in the tree (add, update, delete, find) are implemented using the Depth-First Search (DFS) traversal strategy. DFS is chosen because it is efficient for navigating deep, nested tree structures, which is ideal for this application where nodes can be nested up to five levels or more.

The DFS approach involves recursively traversing each child of a node before returning, which ensures that all levels of the tree are visited if necessary. This allows each node to be located by its unique identifier, even if it is deeply nested.

3.2. Explanation for each function

The `treeUtils` module provides utility functions for all core CRUD operations on nodes:

3.2.1. generateId()

Generates a random, unique identifier for a new node. This ID is used to differentiate nodes within the tree structure.

3.2.2. findNode(tree, targetId)

Traverses the tree using DFS to locate a node by its ID. Returns the node object if found; otherwise, returns `null`.

3.2.3. findParent(tree, targetId, parent = null)

A variation of `findNode`, this function returns the parent node of the node with the given `targetId`. This is useful for operations like deletion or contextual rendering.

3.2.4. addNode(tree, parentId, newNode)

Uses DFS to find the parent node with the specified `parentId`, then appends the `newNode` to its `children` array. If the parent is not found, the function returns `false`.

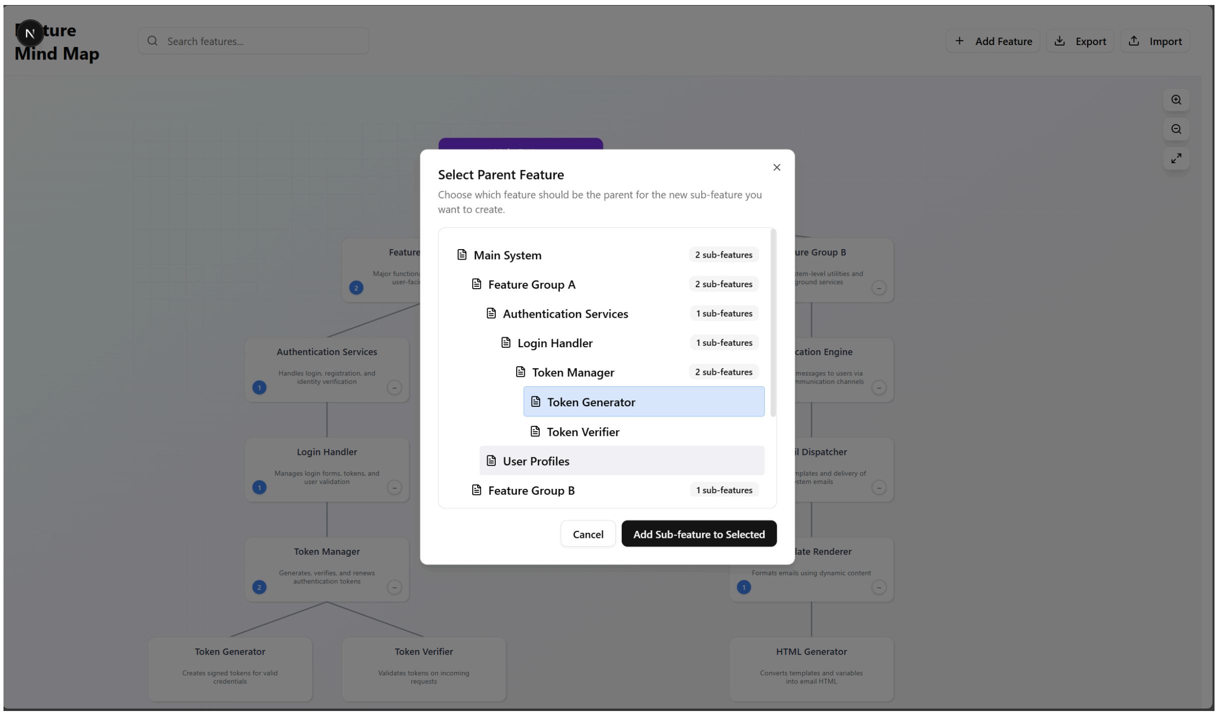

Feature Mind Map
Search features...
Add Feature
Export
Import
Select Parent Feature
Choose which feature should be the parent for the new sub-feature you want to create.
Main System 2 sub-features
Feature Group A 2 sub-features
Authentication Services 1 sub-features
Login Handler 1 sub-features
Token Manager 2 sub-features
Token Generator
Token Verifier
User Profiles
Feature Group B 1 sub-features
Cancel
Add Sub-feature to Selected

Figure 1: UI for creating new node

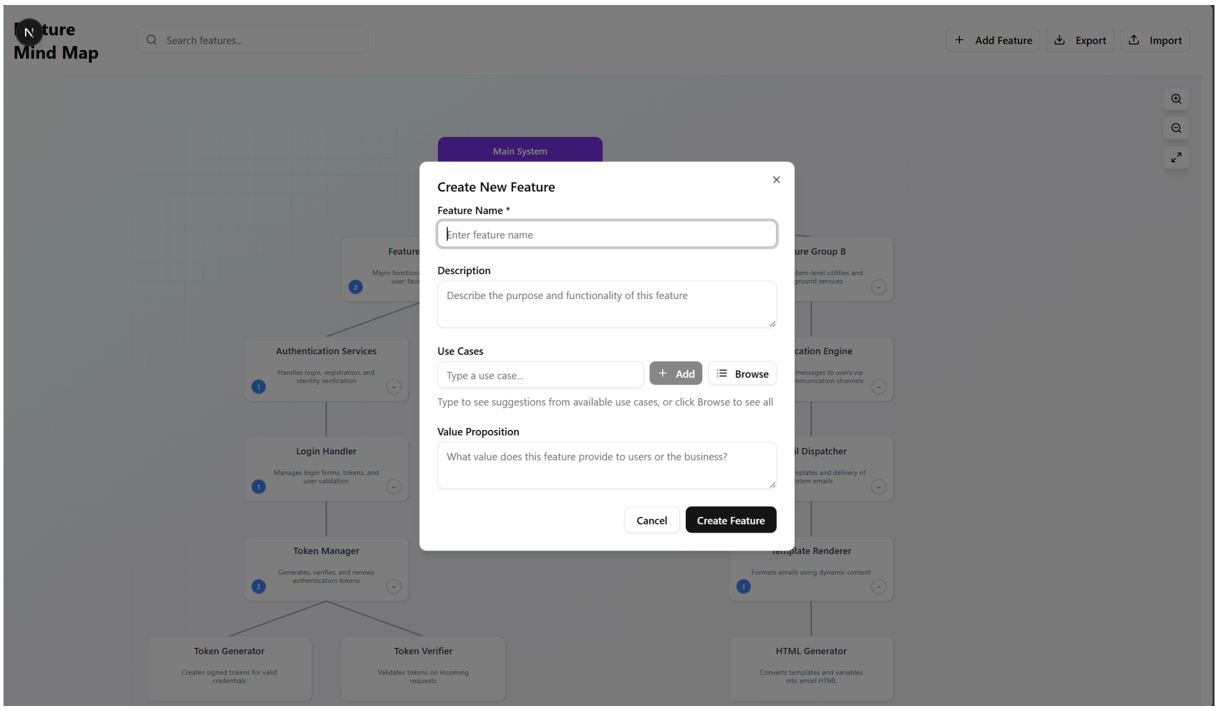
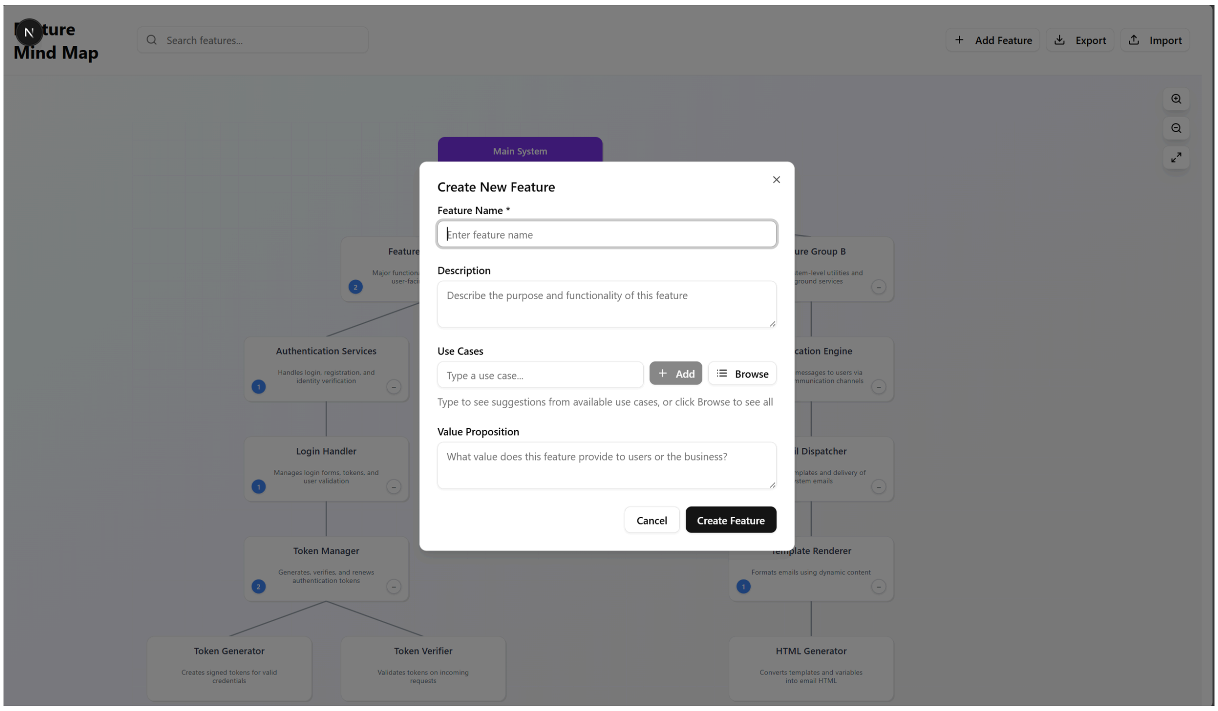


Figure 2: UI for creating new node

3.2.5. updateNode(tree, nodeId, updates)

Locates the node using DFS and applies updates using `Object.assign()`. This allows partial updates to node fields such as `name`, `description`, etc.


Feature Mind Map
Search features...
Add Feature
Export
Import
Main System
Feature Group B
Authentication Services
Handles login, registration, and identity verification
Notification Engine
Login Handler
Manages login forms, tokens, and user validation
Email Dispatcher
Token Manager
Generates, verifies, and renews authentication tokens
Template Renderer
Formats emails using dynamic content
Token Generator
Creates signed tokens for valid credentials
Token Verifier
Validates tokens on incoming requests
HTML Generator
Converts templates and variables into email HTML

Create New Feature

Feature Name *

Enter feature name

Description

Describe the purpose and functionality of this feature

Use Cases

Type a use case... Add Browse

Type to see suggestions from available use cases, or click Browse to see all

Value Proposition

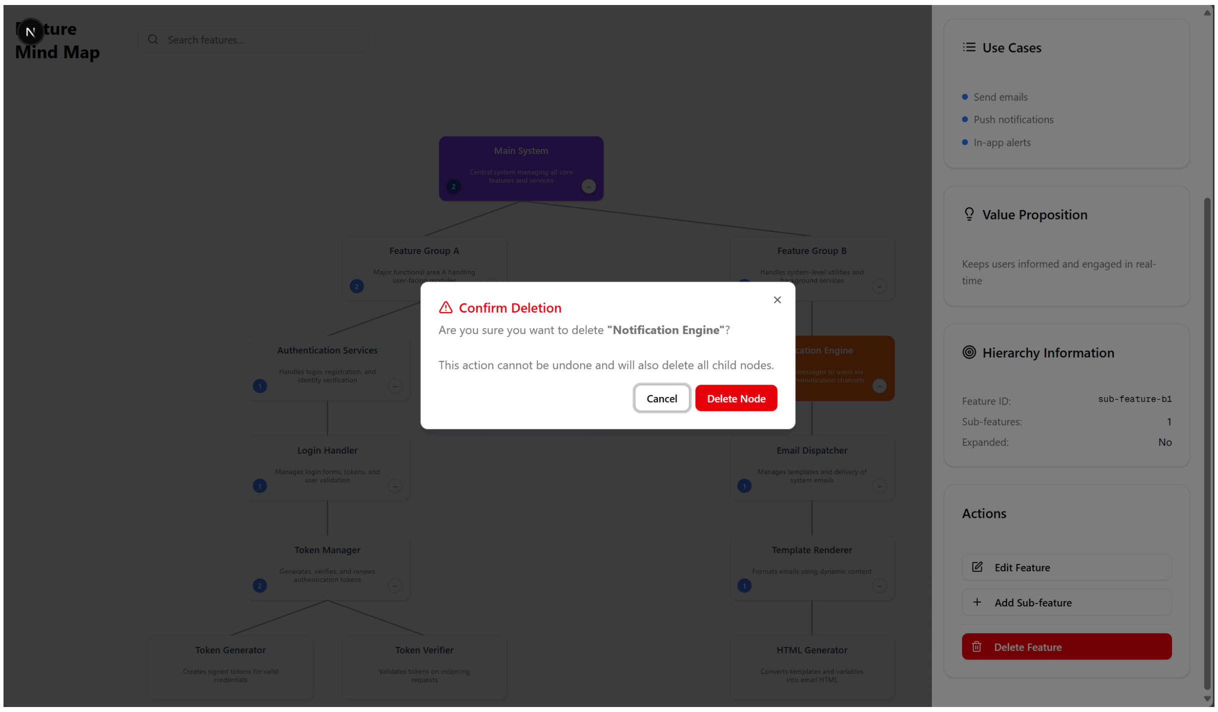
What value does this feature provide to users or the business?

Cancel Create Feature

Figure 3: UI for updating node

3.2.6. deleteNode(tree, targetId)

Recursively searches for the node's parent, then removes the child with `targetId` from the parent's `children` array. It also ensures that all descendant nodes are removed as a result of deleting a subtree.



Confirm Deletion

Are you sure you want to delete **"Notification Engine"**?

This action cannot be undone and will also delete all child nodes.

Cancel Delete Node

Use Cases

- Send emails
- Push notifications
- In-app alerts

Value Proposition

Keeps users informed and engaged in real-time

Hierarchy Information

Feature ID:	sub-feature-b1
Sub-features:	1
Expanded:	No

Actions

Edit Feature

Add Sub-feature

Delete Feature

Figure 4: UI for updating node

3.2.7. toggleExpanded(tree, nodeId)

A UI helper that toggles the `isExpanded` flag on a specific node, allowing sections of the tree to be collapsed or expanded in the user interface.

3.2.8. deepClone(obj)

Creates a deep copy of any object or array structure, ensuring that operations do not mutate the original tree. This is especially important in React for preserving immutability in state management.

3.3. Suggested improvement for algorithm

- **Consider alternative traversal algorithms**: While DFS is optimal for deep tree structures and recursive operations, other algorithms may perform better depending on the use case:
  - **Breadth-First Search (BFS)**: BFS uses a queue to explore nodes level-by-level. It is more suitable when:
    - You want to find the shallowest matching node.
    - You need to process or render nodes level-by-level (e.g., hierarchical visualizations).
  - **Iterative DFS**: Instead of recursion, use an explicit stack to traverse the tree. This avoids potential stack overflow in very deep trees and is easier to debug in environments where recursion is limited.
  - **Pre-order vs. Post-order Traversal**: Depending on whether you want to process a node before or after its children, consider choosing the appropriate traversal variant explicitly (useful in operations like rendering or cascading deletions).
  - **Hybrid Approach**: In certain cases, combining DFS for deep node processing and BFS for layout rendering can improve overall performance and responsiveness in UI-heavy applications.

By selecting the traversal strategy best suited to the task (e.g., BFS for level-wide operations, DFS for targeted updates), the performance and maintainability of tree operations can be significantly improved.

4. UI interaction

4.1. Canvas layout & navigation

The tree layout is rendered using a custom layout algorithm that places each node based on its depth (vertical spacing) and relative subtree width (horizontal spacing). This layout is computed using a two-pass Depth-First Search (DFS):

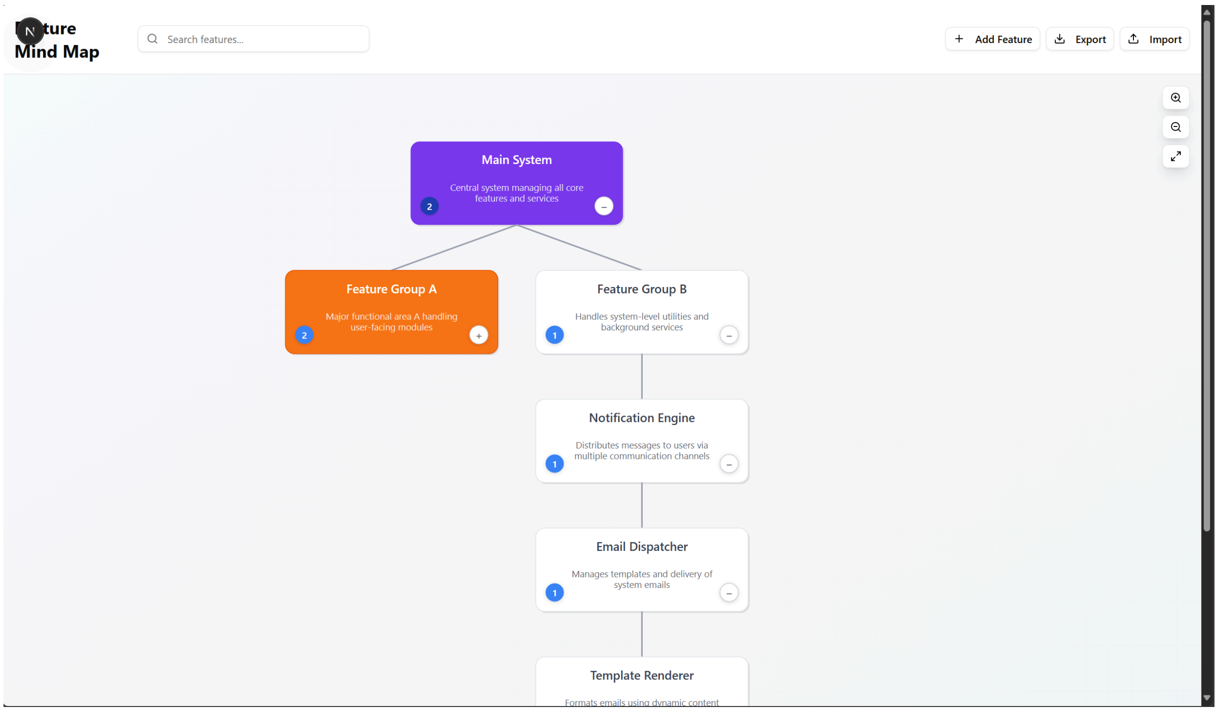
1. `calculateSubtreeWidth` estimates the space required for each node's subtree.
2. `layoutNode` recursively assigns `x` and `y` positions for each node based on that width.

This allows the interface to comfortably render trees with 5+ levels of depth while keeping siblings spaced visually.

The canvas supports:
- **Panning** via mouse dragging.
- **Zooming** via mouse scroll or UI buttons (`ZoomIn`, `ZoomOut`).
- **Auto-center & Fit-to-View** functionality triggered on initial load or via the `reset` button.

4.2. Expand/Collapse functionality

Each node displays an **expand/collapse button** (`+` or `-`) if it has children. Clicking this button updates the `isExpanded` flag for the node, and the tree re-renders accordingly. This logic is handled by the `handleExpandCollapseClick` function and `onNodeToggle` callback.



Main System
Central system managing all core features and services
2

Feature Group A
Major functional area A handling user-facing modules
2

Feature Group B
Handles system-level utilities and background services
1

Notification Engine
Distributes messages to users via multiple communication channels
1

Email Dispatcher
Manages templates and delivery of system emails
1

Template Renderer
Formats emails using dynamic content

Figure 5: Feature group A is not expanded

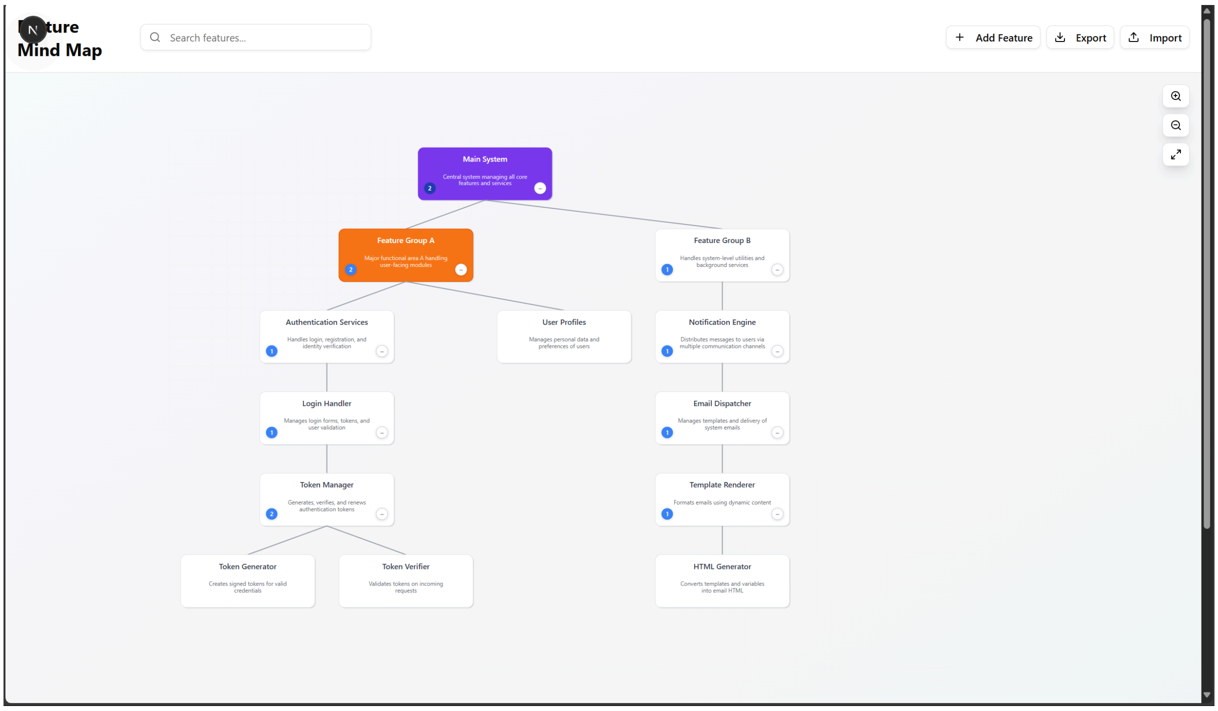

Feature
Mind Map
Search features...
Add Feature
Export
Import
Main System
Central system managing all core features and services
Feature Group A
Major functional area A handling user-facing modules
Feature Group B
Handles system-level utilities and background services
Authentication Services
Handles login, registration, and identity verification
User Profiles
Manages personal data and preferences of users
Notification Engine
Distributes messages to users via multiple communication channels
Login Handler
Manages login forms, tokens, and user validation
Email Dispatcher
Manages templates and delivery of system emails
Token Manager
Generates, verifies, and renews authentication tokens
Template Renderer
Formats emails using dynamic content
Token Generator
Creates signed tokens for valid credentials
Token Verifier
Validates tokens on incoming requests
HTML Generator
Converts templates and variables into email HTML

Figure 6: Feature group A is expanded

4.3. Node interaction

Each node is represented by a styled SVG group:

- **Clicking** a node triggers the `onNodeClick` callback to display its full details.
- Nodes are styled based on:
  - Whether they are selected.
  - Whether they appear in the search results.
  - Their role (root node or regular node).
- **Multi-line text wrapping** is used to display long names and descriptions cleanly.

4.4. Key state variables and refs

At the core of the canvas-based visualization are several state variables and references that track viewport interaction, layout dimensions, and zoom/pan behavior.

```
const svgRef = useRef(null);
const containerRef = useRef(null);
```

- `svgRef` : Ref pointing to the `<svg>` element that renders the entire tree structure.
- `containerRef` : Ref for the HTML container that wraps the SVG. Used for detecting size and mouse interactions.

```
const [transform, setTransform] = useState({ x: 0, y: 0, scale: 1 });
```

- Stores the current transformation state:
  - `x` , `y` : The translation offsets (pan) of the view.
  - `scale` : The zoom level (1 = 100%).

```
const [isDragging, setIsDragging] = useState(false);
const [dragStart, setDragStart] = useState({ x: 0, y: 0 });
```

- `isDragging` : Tracks whether the user is actively dragging (panning) the canvas.

- `dragStart`: Records the initial mouse position when dragging begins. This allows the system to calculate relative movement.

```
const [dimensions, setDimensions] = useState({ width: 800, height: 600 });
```

- Stores the current width and height of the canvas container.
- This is dynamically updated when the window resizes to ensure responsive rendering of the tree.

```
const NODE_WIDTH = 280;
const NODE_HEIGHT = 110;
const LEVEL_HEIGHT = 170;
const MIN_NODE_SPACING = 320;
```

- Constants that define the physical size and spacing between nodes:
  - `NODE_WIDTH` / `NODE_HEIGHT`: Dimensions of each node rectangle.
  - `LEVEL_HEIGHT`: Vertical space between levels in the tree.
  - `MIN_NODE_SPACING`: Minimum horizontal space between sibling nodes.

These values collectively determine the layout, positioning, and interaction behavior of the mind-map canvas.

4.5. Key functions explained

4.5.1. Viewport Dimension Handling

This `useEffect` ensures the canvas automatically updates its dimensions when the browser window is resized.

```
useEffect(() => {
  const updateDimensions = () => {
    if (containerRef.current) {
      const rect = containerRef.current.getBoundingClientRect();
      setDimensions({ width: rect.width, height: rect.height });
    }
  };

  updateDimensions();
  window.addEventListener("resize", updateDimensions);
  return () => window.removeEventListener("resize", updateDimensions);
}, []);
```

4.5.2. Layout Calculation and Memoization

Node positions are computed using a recursive layout algorithm and memoized to avoid unnecessary recalculations.

```
const nodePositions = useMemo(
  () => calculateLayout(tree),
  [tree, dimensions.width, dimensions.height]
);
```

Purpose:
- Uses `useMemo` to avoid re-running expensive layout calculations unless the `tree` or dimensions change.
- Ensures stability and performance in the rendering process.

4.5.3. Node Rendering

Each feature/feature group is rendered as an SVG group (`<g>`) with colored rectangles, text, and control buttons.

```
const renderNode = (node) => {
  const position = nodePositions.get(node.id);
  if (!position) return null;

  return (
    <g key={node.id} className="node-element">
    <rect ... /> // Node shadow
    <rect ... onClick={() => onNodeClick(node)} /> // Main rectangle
    <text ...>{node.name}</text> // Name
    <text ...>{node.description}</text> // Description
    <circle ... /> // Children count
    <circle ... onClick={(e) => handleExpandCollapseClick(e, node.id)} />
    </g>
  );
};
```

Purpose:
- Displays each node with visual styling.
- Clickable to trigger detail viewing.
- Expand/collapse icons for child visibility toggling.

4.5.4. Connection Rendering

Lines are drawn between a node and its children to visually represent hierarchy.

```
const renderConnections = (node) => {
    const position = nodePositions.get(node.id);
    if (!position || node.isExpanded === false || !node.children) return null;

    return node.children.map((child) => {
      const childPosition = nodePositions.get(child.id);
      if (!childPosition) return null;

      return (
        <line
          key={${node.id}-${child.id}}
          x1={position.x}
          y1={position.y + NODE_HEIGHT / 2}
          x2={childPosition.x}
          y2={childPosition.y - NODE_HEIGHT / 2}
          stroke="#9ca3af"
          strokeWidth={2}
          className="transition-all duration-300"
        />
      );
    });
  };
```

4.5.5. Auto Fit-to-View (on tree change)

```
useEffect(() => {
    const bounds = getContentBounds();
    // Calculate scale to fit all content
    const scaleX = dimensions.width / bounds.width;
    const scaleY = dimensions.height / bounds.height;
    const scale = Math.min(scaleX, scaleY, 1); // Don't scale up beyond 100%

    // Calculate position to center content
    const centerX = bounds.minX + bounds.width / 2;
    const centerY = bounds.minY + bounds.height / 2;
    const x = dimensions.width / 2 - centerX * scale;
    const y = dimensions.height / 2 - centerY * scale;

    setTransform({ x, y, scale });
  }, [nodePositions, dimensions, getContentBounds]);
```

Purpose:

- Computes the bounding box that contains all nodes.
- Adds padding to avoid edge clipping.
- Used for auto-centering and zoom adjustment.
- Automatically zooms and centers the tree whenever layout or container size changes.

4.5.6. Zoom Controls (Wheel, Buttons, Reset)

Purpose:

- `handleWheel` : Allows scroll-based zooming, constrained between 10% and 300%.
- `zoomIn` / `zoomOut` : Manual zoom buttons for user control.
- `resetView` : Recalculates bounds and centers the entire tree.

Together, these functions ensure the canvas remains user-friendly and adaptable to varying tree sizes and screen dimensions.

4.5.7. Mouse Event Listeners for Canvas Interaction

This `useEffect` attaches and detaches low-level mouse and wheel event listeners to the canvas container.

```
 useEffect(() => {
    const container = containerRef.current;
    if (!container) return;

    container.addEventListener("mousedown", handleMouseDown);
    container.addEventListener("mousemove", handleMouseMove);
    container.addEventListener("mouseup", handleMouseUp);
    container.addEventListener("mouseleave", handleMouseUp);
    container.addEventListener("wheel", handleWheel, { passive: false });

    return () => {
      container.removeEventListener("mousedown", handleMouseDown);
      container.removeEventListener("mousemove", handleMouseMove);
      container.removeEventListener("mouseup", handleMouseUp);
      container.removeEventListener("mouseleave", handleMouseUp);
      container.removeEventListener("wheel", handleWheel);
    };
  }, [handleMouseDown, handleMouseMove, handleMouseUp, handleWheel]);
```

Purpose:

- Listens for **mouse interactions** on the canvas wrapper:
  - `mousedown` : Begin panning (dragging).
  - `mousemove` : Update pan position in real time.
  - `mouseup` / `mouseleave` : Stop panning on mouse release or leaving canvas.
  - `wheel` : Trigger zoom based on scroll.
- Ensures event listeners are properly **attached on mount** and **cleaned up on unmount** to prevent memory leaks or unintended behavior.

5. Sitemap search functionality

5.1. Overview

The application supports an advanced search feature to help users locate specific nodes in the sitemap. The search operates on both the “Feature Name” and “Description” fields and supports:

- Exact word matches.
- Synonym-based matches using a predefined dictionary.

The results are clearly differentiated in the UI using distinct visual styles (e.g., colored highlights).

5.2. Search logic

The search functionality is implemented in a utility module (`searchUtils`). It includes exact word matching, synonym expansion, and recursive traversal of the tree structure.

5.3. Debouncing

To improve performance and prevent unnecessary re-computation during rapid typing, the search feature uses a debouncing technique. Specifically, the `query` string is passed into a memoized debounced function, which triggers the actual search logic only after the user has stopped typing for a short delay (300ms). This reduces the number of times the search is re-executed and improves responsiveness.

```
const debouncedSearch = useMemo(
  () =>
    debounce((value) => {
      if (value.trim()) {
        onSearch(value);
        setShowResults(true);
      } else {
        onClearSearch();
        setShowResults(false);
      }
    }, 300),
  [onSearch, onClearSearch]
);

useEffect(() => {
  debouncedSearch(query);
  return () => {
    debouncedSearch.cancel();
  };
}, [query, debouncedSearch]);
```

Purpose:

- debounce(...): Waits until the user has stopped typing for 300ms before executing the search.
- Prevents excessive computation and unnecessary re-renders during fast input.
- Cancels any pending execution if the component re-renders before the timer completes.

This ensures that the actual searchNodes function is only triggered when truly needed, which is especially beneficial for larger trees or when the search logic includes expensive operations like synonym checking.

5.3.1. Synonym dictionary

A small, predefined dictionary of related terms is used to expand user queries. For example:

```
"payment": ["billing", "invoice", "checkout", "transaction", "purchase"]
"user": ["account", "profile", "member", "client"]
"login": ["auth", "signin", "access", "authentication"]
```

This allows for more flexible and user-friendly search experiences.

5.3.2. Exact word matching

The function `hasExactWordMatch` determines whether a specific word appears in the node text as an exact match.

```
hasExactWordMatch(text, searchWord) => text.split(/\s+/).includes(searchWord)
```

This ensures that only full-word matches are considered valid, avoiding false positives from partial substrings.

5.3.3. Synonym matching

The hasSynonymMatch function checks if any synonym of the search term appears in the node's name or description.

```
hasSynonymMatch(text, searchWord) {
  const synonyms = getSynonyms(searchWord);
  return synonyms.some((synonym) => hasExactWordMatch(text, synonym));
}
```

This allows queries like "billing" to match nodes with "payment" in the name or description.

5.3.4. Recursive tree search

The searchNodes function performs a recursive traversal of the entire tree to identify nodes matching the query.

- For each node:
  - Combine name and description into searchable text.
  - Check each word in the query against the text using both exact and synonym matching.
  - If matched, collect metadata about how the match occurred (matchType, matchedTerms, matchedSynonyms).
- Recursively search child nodes.
- Sort results with exact matches first.

5.3.5. Highlighting and output

The output of the search function is a list of objects, each containing:

- The matched node
- Match type (“exact” or “synonym”)
- Which terms were matched
- Which synonyms triggered the match

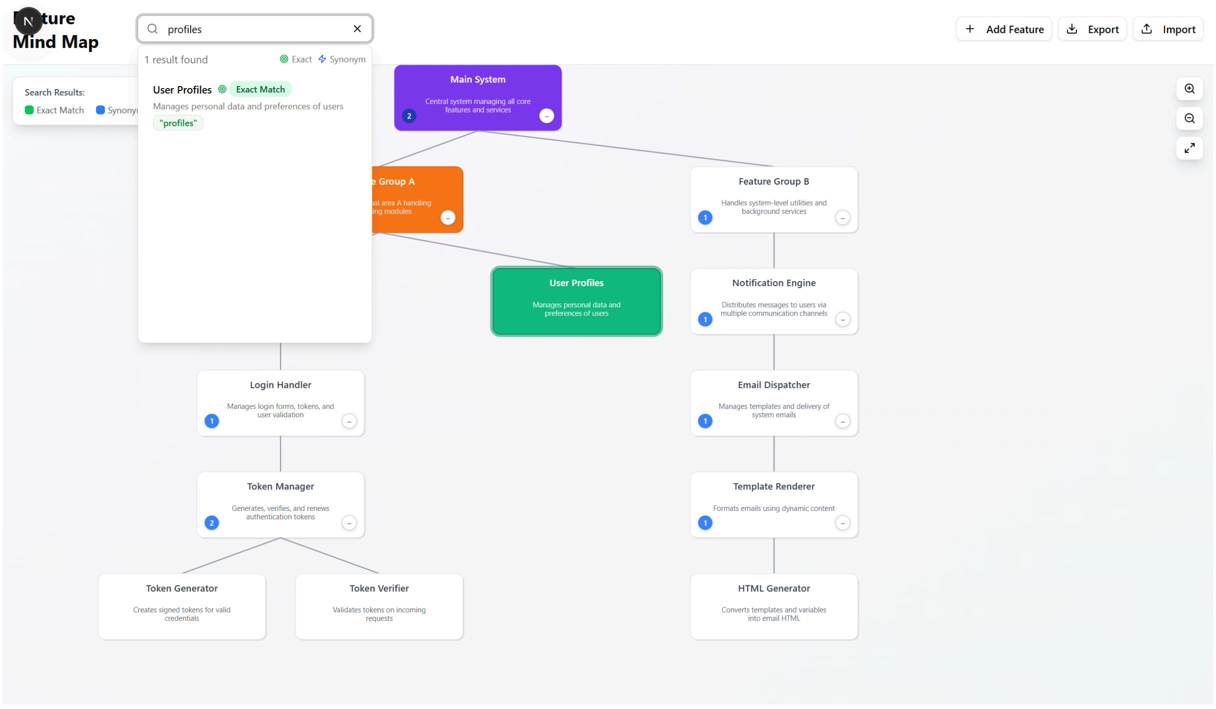

ture Mind Map
profiles
Add Feature
Export
Import
Search Results:
Exact Match
Synony
1 result found
Exact
Synonym
User Profiles
Exact Match
Manages personal data and preferences of users
"profiles"
Main System
Central system managing all core features and services
Group A
al area A handling ing modules
Feature Group B
Handles system-level utilities and background services
User Profiles
Manages personal data and preferences of users
Notification Engine
Distributes messages to users via multiple communication channels
Login Handler
Manages login forms, tokens, and user validation
Email Dispatcher
Manages templates and delivery of system emails
Token Manager
Generates, verifies, and renews authentication tokens
Template Renderer
Formats emails using dynamic content
Token Generator
Creates signed tokens for valid credentials
Token Verifier
Validates tokens on incoming requests
HTML Generator
Converts templates and variables into email HTML

Figure 7: Exact word match

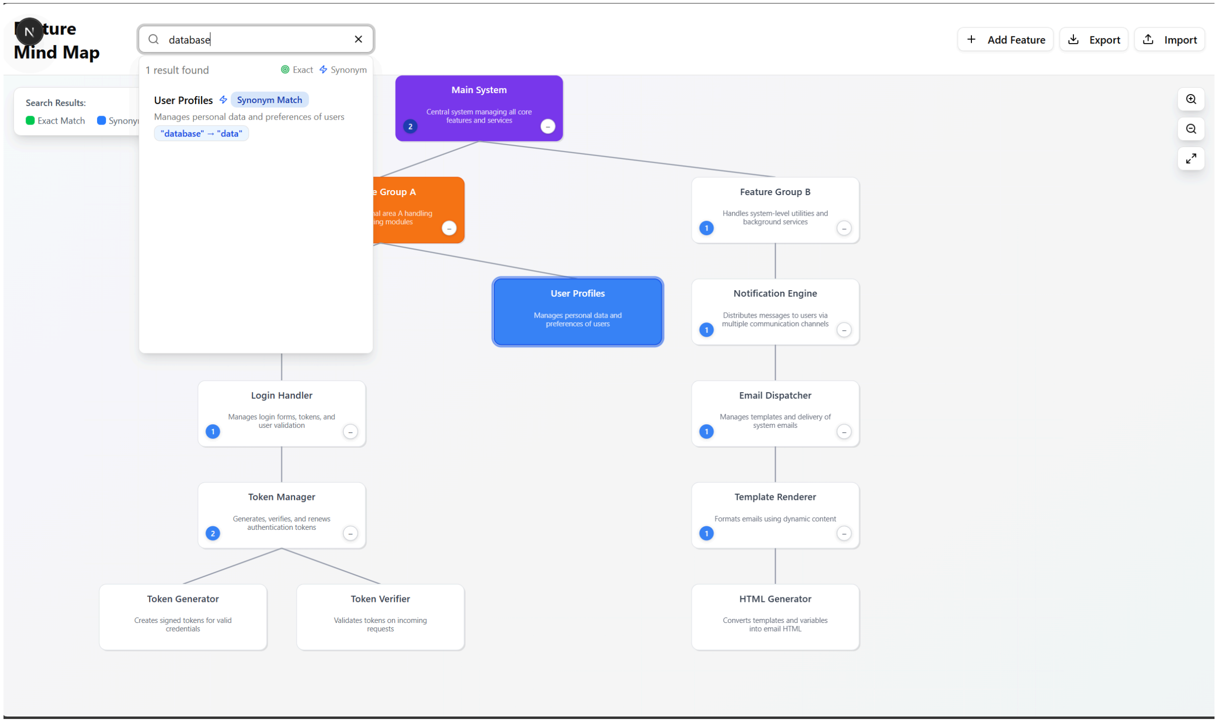

ture Mind Map
database
Add Feature
Export
Import
Search Results:
Exact Match
Synony
1 result found
Exact
Synonym
User Profiles
Synonym Match
Manages personal data and preferences of users
"database" → "data"
Main System
Central system managing all core features and services
Group A
al area A handling ing modules
Feature Group B
Handles system-level utilities and background services
User Profiles
Manages personal data and preferences of users
Notification Engine
Distributes messages to users via multiple communication channels
Login Handler
Manages login forms, tokens, and user validation
Email Dispatcher
Manages templates and delivery of system emails
Token Manager
Generates, verifies, and renews authentication tokens
Template Renderer
Formats emails using dynamic content
Token Generator
Creates signed tokens for valid credentials
Token Verifier
Validates tokens on incoming requests
HTML Generator
Converts templates and variables into email HTML

Figure 8: Synonym word match

5.3.6. Suggested improvement

To enhance the search functionality, the following improvements are suggested:

- Multi-word phrase matching: Support quoted phrases (e.g., “user login”) for more precise results.
- Fuzzy matching: Use libraries like Fuse.js for approximate word matching with typo tolerance.
- Field-specific search: Allow syntax like name:payment or desc:checkout to narrow down scope.
- Search history: Cache past queries for easier repeated use or suggestions.
- Performance Optimization: For very large trees, debounce input and cache last search results.

6. Use case management

6.1. Overview

To improve reusability and consistency across the sitemap, the application includes a utility module for managing use cases. Each node in the sitemap can define its own list of use cases (e.g., `"User login"`, `"Send email"`, etc.), and these use cases are centrally extracted, initialized, and searchable.

This module allows users to:

- Reuse existing use cases instead of creating new ones manually.
- Search for common use cases to ensure naming consistency.
- Distinguish between system-provided (default) and custom use cases.

6.2. Functional Description

The logic is encapsulated in a `useCaseStorage` utility, which provides the following key functions:

6.2.1. getDefaultUseCases()

Returns a shallow copy of the predefined default use cases from `sample-use-cases`.

```
getDefaultUseCases: () => [...defaultUseCases]
```

6.2.2. extractUseCasesFromTree(tree)

Traverses the tree recursively and collects all use cases defined at each node.

```
extractUseCases(node):
  if node has useCases:
    add them to result
  if node has children:
    recursively call extractUseCases(child)
```

This ensures that any manually added or imported use cases from the sitemap structure are also captured.

6.3. initializeUseCases(tree)

Combines:

- Predefined use cases (defaultUseCases)

- Extracted use cases from the current sitemap tree

The result is a deduplicated, alphabetically sorted list of all available use cases.

```
initializeUseCases(tree):
  treeUseCases = extractUseCasesFromTree(tree)
  allUseCases = removeDuplicates(defaultUseCases + treeUseCases)
  return sorted(allUseCases)
```

6.3.1. searchUseCases(allUseCases, query)

Performs a case-insensitive substring search on all available use cases.

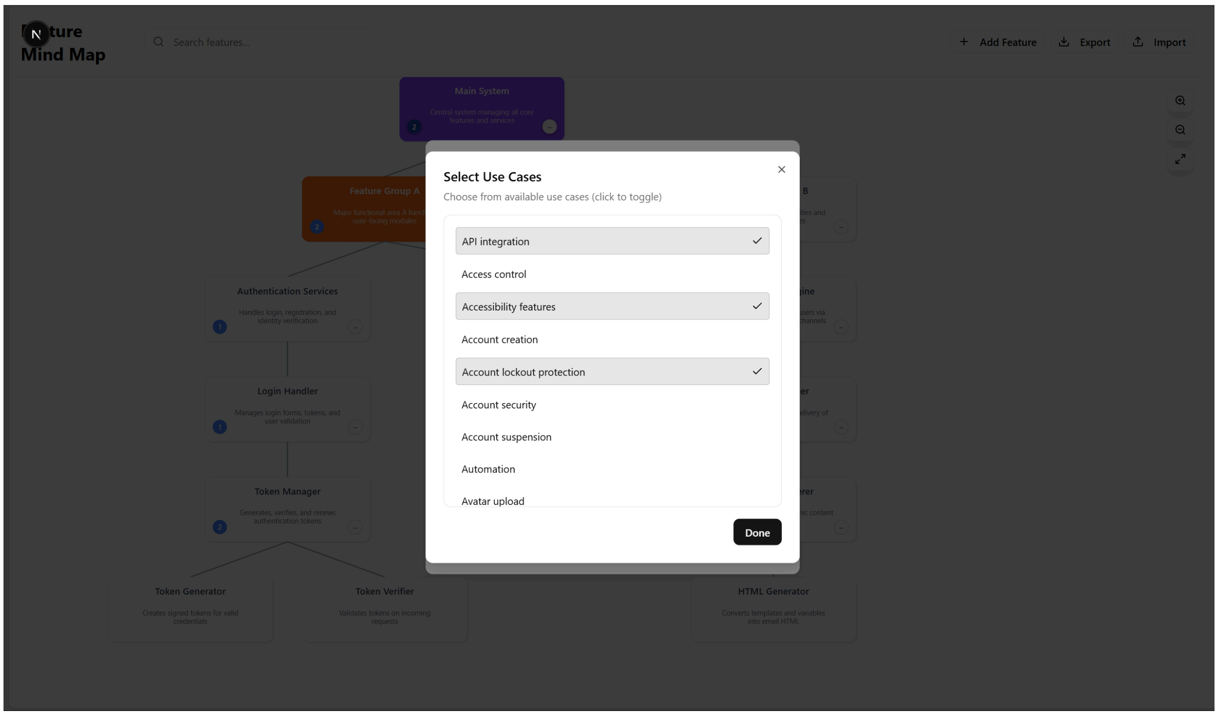
```
searchUseCases(allUseCases, query):
  if query.length < 3:
    return []
  return useCases containing query
```

This enables real-time filtering as users type, improving discoverability and reducing duplication.

6.4. UI integration for use case selection

To improve user experience and maintain consistency across the sitemap, the system includes two user-friendly interfaces for managing use cases when creating or editing a feature node.

6.4.1. Use case selection modal


Select Use Cases

Choose from available use cases (click to toggle)

- API integration ✓
- Access control
- Accessibility features ✓
- Account creation
- Account lockout protection ✓
- Account security
- Account suspension
- Automation
- Avatar upload

Feature Mind Map
Search features...
Add Feature
Export
Import
Main System
Central system managing all core features and services
Feature Group A
Authentication Services
Handles login, registration, and identity verification
Login Handler
Manages login forms, tokens, and user validation
Token Manager
Generates, verifies, and renews authentication tokens
Token Generator
Creates signed tokens for valid credentials
Token Verifier
Validates tokens on incoming requests
HTML Generator
Converts templates and variables into email HTML
Done

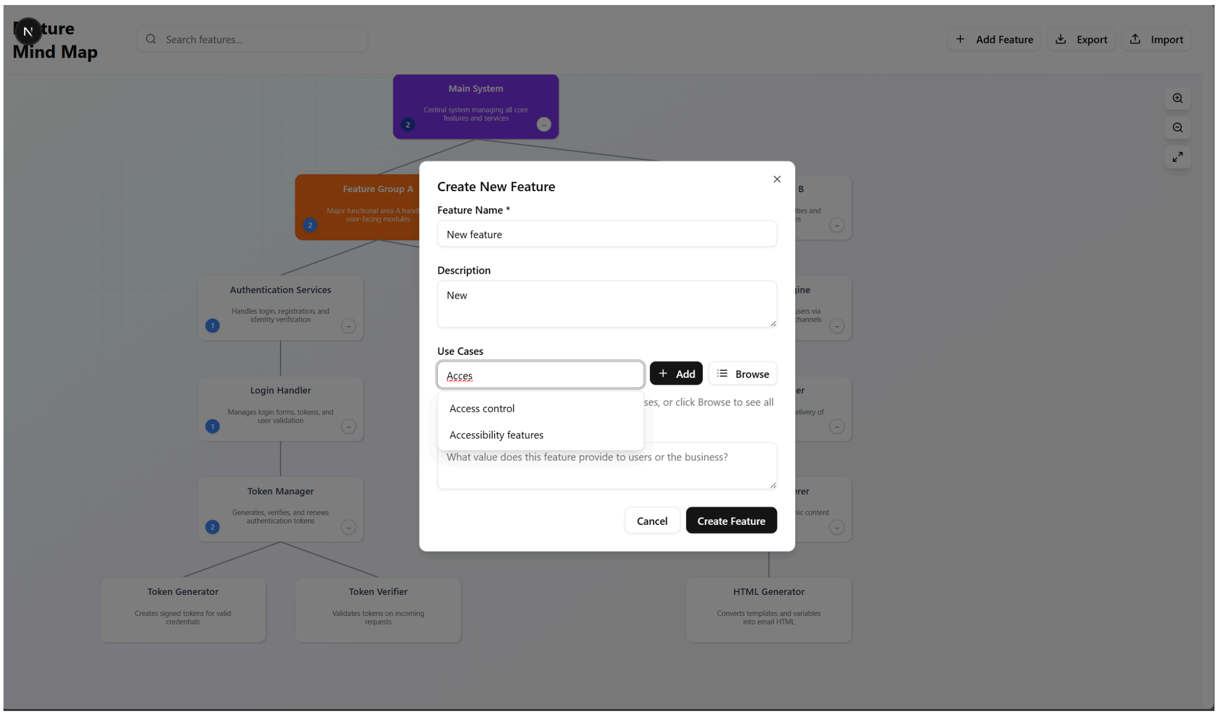
Figure 9: Use case selection modal – users can browse and toggle existing use cases

This modal appears when the user clicks the "Browse" button while creating or editing a feature.

- All available use cases are listed in a scrollable panel.
- Clicking a use case toggles its selection state (✓).
- Preselected use cases appear checked.
- Pressing "Done" finalizes the selection and closes the modal.
- The list is populated by `initializeUseCases(tree)` which merges defaults and current node data.

This design helps reduce duplication, reinforces standard terminology, and encourages reuse of meaningful use case labels.

6.4.2. Use case input with autocomplete


Create New Feature

Feature Name *

New feature

Description

New

Use Cases

Acces

Access control

Accessibility features

ses, or click Browse to see all

What value does this feature provide to users or the business?

Feature Mind Map
Search features...
Add Feature
Export
Import
Main System
Central system managing all core features and services
Feature Group A
Authentication Services
Handles login, registration, and identity verification
Login Handler
Manages login forms, tokens, and user validation
Token Manager
Generates, verifies, and renews authentication tokens
Token Generator
Creates signed tokens for valid credentials
Token Verifier
Validates tokens on incoming requests
HTML Generator
Converts templates and variables into email HTML
Add
Browse
Cancel
Create Feature

Figure 10: Inline use case input with autocomplete and quick add functionality

While typing in the “Use Cases” field in the feature creation form:

- The system suggests matching use cases via a dropdown (case-insensitive search).
- Results are provided by `searchUseCases()` with a minimum query length.
- If a user types a new value that doesn’t exist, they can click `+ Add` to insert it.
- Alternatively, they can click `Browse` to open the full modal selection panel.

This hybrid approach (autocomplete + modal) balances **quick input** for experienced users with **guided selection** for beginners.

Together, these two UI components significantly enhance the reusability, discoverability, and organization of use cases across the entire feature map.

7. Summary

This project demonstrates a functional and user-friendly system that meets all the specified requirements for building and managing a product feature sitemap. It includes:

- A scalable hierarchical data structure that captures feature details clearly.
- A smooth and intuitive user interface for interacting with the sitemap via mind-map-style visualization.
- An advanced search system that enhances discoverability through synonym mapping.
- A use case storage system to promote reuse and consistency.

All functionality was developed within the given timeframe using a modular and extensible codebase. The report also highlights important implementation decisions, including tree traversal logic, canvas rendering algorithms, and interaction design choices.

This tool serves as a strong foundation for future expansion, such as mapping user pain points to features, analytics on feature usage, or integrations with product management platforms.

Thank you for the opportunity to participate in this assessment. I look forward to discussing the technical approach and learning opportunities in more detail.

Index of Figures

Figure 1 UI for creating new node 8
Figure 2 UI for creating new node 8
Figure 3 UI for updating node 9
Figure 4 UI for updating node 9
Figure 5 Feature group A is not expanded 11
Figure 6 Feature group A is expanded 12
Figure 7 Exact word match 19
Figure 8 Synonym word match 19
Figure 9 Use case selection modal – users can browse and toggle existing use cases 22
Figure 10 Inline use case input with autocomplete and quick add functionality 23