**Search Implementation Documentation**

**Architecture Overview**

The search functionality is implemented using a layered approach that combines real-time menu data synchronization with flexible search capabilities:

**1. Core Components**

* **SearchService**: A standalone service that handles the search logic
* **SearchContext**: React context that bridges the menu data with the search service
* **MenuContext**: Manages menu data and provides updates
* **SearchModal**: UI component for search interaction

**2. Data Flow**

MenuContext (menu data) → SearchContext → SearchService → UI

↑ ↓

Auto-sync Search Results

**3. Key Features**

**Dynamic Menu Data Handling**

* Automatically syncs with the latest menu data
* No manual refresh needed when menu.json changes
* Uses React's useEffect for real-time updates

**Search Algorithm**

The search implements a multi-faceted scoring system:

1. **Text Matching**
   * Word-by-word scoring
   * Exact matches (score: 1.0)
   * Partial matches (score: 0.5)
   * Name matches get double weight (score: 2.0)
2. **Semantic Matching**
   * Ingredient pattern recognition
   * Dietary preference detection (vegetarian, vegan, gluten-free)
   * Category-based matching
3. **Negative Search Support**
   * Handles queries like "pasta without mushrooms"
   * Pattern matching for exclusions:
     + "without [ingredient]"
     + "no [ingredient]"
     + "with no [ingredient]"
   * Applies penalties for unwanted ingredients (-30 points)
4. **Special Pattern Recognition**
   * Healthy food patterns
   * Dietary restrictions
   * Preparation methods
   * Ingredient categories (meats, dairy, eggs, gluten, nuts)

**4. Implementation Details**

**Search Process**

1. Query Normalization
2. const normalizedQuery = query.toLowerCase().trim();
3. const queryWords = normalizedQuery.split(/\s+/);
4. Score Calculation
5. score = textMatchScore + (semanticScore \* 3) + categoryBonus - penalties;
6. Result Filtering
7. threshold = -25; // Allows negative scores for better filtering
8. results.filter(result => result.score > threshold)

**Ingredient Pattern Detection**

* Predefined patterns for common ingredients
* Category-based grouping
* Plural/singular form handling

**Result Grouping**

* Results grouped by category
* Maintains original menu structure
* Preserves item metadata (price, description, dietary info)

**5. Provider Structure**

<MenuProvider>

<SearchProvider> {/\* Has access to menu data \*/}

<App /> {/\* All components can use search \*/}

</SearchProvider>

</MenuProvider>

**6. Performance Considerations**

* Debounced search execution
* Early returns for empty queries
* Optimized scoring calculations
* Caching of menu data

**7. Usage Example**

// In any component:

const { search } = useSearch();

const results = await search("pasta without mushrooms", {

topK: 5,

threshold: -25

});

This architecture ensures that:

1. Search is always using the latest menu data
2. Changes to menu.json are immediately reflected
3. Complex search patterns are handled efficiently
4. The UI remains responsive and user-friendly

**Key Files in Search Implementation**

1. **src/services/searchService.js**
   * Core search logic implementation
   * Contains pattern matching and scoring algorithms
   * Defines ingredient patterns and search rules
   * Handles negative search queries
2. **src/context/SearchContext.jsx**
   * React context for search functionality
   * Bridges MenuContext with searchService
   * Provides search capabilities to components
   * Auto-syncs with menu data changes
3. **src/context/MenuContext.jsx**
   * Manages menu data state
   * Handles menu version control
   * Provides menu data to SearchContext
   * Detects and loads menu.json changes
4. **src/components/common/SearchModal.jsx**
   * Search UI implementation
   * Handles user input and keyboard shortcuts
   * Displays search results with highlighting
   * Shows quick links and search suggestions
5. **src/components/Navbar.jsx**
   * Contains search trigger button
   * Handles search modal open/close
   * Keyboard shortcut (⌘K) integration
6. **index.html**
   * Initializes window.searchService
   * Sets up global search configuration
   * Defines search result formatting
7. **src/App.jsx**
   * Sets up provider hierarchy
   * Ensures SearchProvider is wrapped by MenuProvider
   * Configures global error boundaries
8. **src/main.jsx**
   * Root application setup
   * Provider order configuration
   * Router and content provider setup

This modular structure allows for:

* Clear separation of concerns
* Easy maintenance and updates
* Efficient data flow
* Scalable search functionality