System Design Plan

1. Requirements - (Generic and functional)
2. Architecture
3. Data model
4. Interface / API Design
5. Store Design
6. Optimization
7. Accessibility

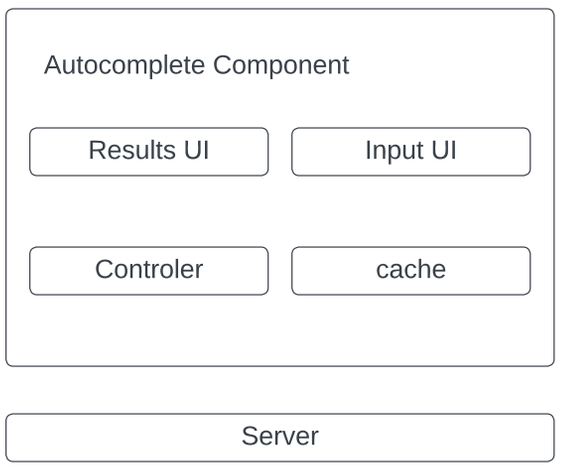
General Requirements

1. The component / widget is generic enough to be used by different devices
2. The input field UI and results UI should be customizable.

Functional Requirements

1. The search query must be debounced
2. The query string and the results must be cached
3. Only a certain length of results must be displayed. More can be scrolled.
4. The results can be text, image or media

Architecture
UI and dependency graph



1. Input handles the user input and passes it to the controller.
2. The results UI receives the results from the controller.
3. Results UI also handles the user selection of the results and informs the controller.
4. stores the results of the previous queries, so that controller can first check in the cache before sending the rewuest to the server.
5. Passes the user input and results between the components.
6. Fetches results from the server if the cache is empty for a particular string.

Data Model

Controller
- Current search string,
- Props or options exposed via component API (Autocomplete component)

Cache
- Initial results
- Cached Results

Interface definition

Basic Component API
- Number of results
- Search API url
- Event Listeners (onchange, onConfirm, onBlur, onClose, onFocus)
- Customized rendering (styling) - Theming options object, - classNames, render function as callback (Inversion of control) --> This method needs to be invoked with some data.

Advanced API
- Min query length (only trigger the api after min characters are typed)
- Debounce duration (Triggering the backend API after every character typed is not feasible. Using debouncing technique we can limit the api calls, by triggering them only after certain duration has passed)
- API timeout duration - How long to wait for the results

Optional Cache configuration options
- Datasource: 'network only', 'network and cache', 'cache only'
- cacheduration (Time to live): timestamp for each entry, and evicting them from the cache after the time stamp

Server API
- A HTTP API that takes in search query, results limit, page number and returns the results appropriately.

Optimizations

Network
- Handling concurrent requests or race conditions
  User caould make changes to the search query while th previous request is not yet processed. But we cannot show the previous results. We also cannot rely on the return order of results from the server.
  a. Attach a time stamp to each request, to identify the latest request and its response.
  b. Save the results in an object/map, keyed by the search query string and only present the results for the right query - Cache(better option).
- Failed requests and retries - Retry the query if the prev request failed, if the server is offline, display appropriate error state.
- Offline usage - Read purely from the cache, - Do not fire any request, so the cpu cycles are not wasted, indicate the no network error on the component
- **Client side Performance**
- Showing cached data quickly
- Debouncing - so that server payload and cpu processing is reduced
- Memory usage in check by clearing the cache periodically
- Virtualized list (Showing only certain number of results - so that no need to create new dom nodes, and recycle the existing ones)
- **User Experience**
- Autofocus, loading, error, no network states, truncate long strings with elipsis, responsiveness, tabindex for keyboard focus, fuzzy search,
- **Accessibility**
- Appropriate aria roles for non semantic html, enter for onConfirm, up-down to navigate the results, escape to dismiss the popup.

Caching

- To improve the performance of the query

Cache structure
- **1. Hash map with search query as key and results as value**
- O(1) fetching, but con is duplicated data, especially if debouncing is not implemented, can hence consume lot of memory.
- **2. List of results**
- Save all the results ever, and do filtering on the frontend side - Not performance optimal, but there won't be duplicates
- **3. Normalized map of results**
- Structure the cache like a database - fast lookup and no duplicates
- Instead of storing in flat array, each result entry is one row in the db object and is identified by the unique id, cache simply refers to each item's id.
- const result = {
- 1: {id:1, type: 'organization', text: 'facebook',
- 2: {id:2, type: 'organization', text: 'face',
- 3: {id:3, type: 'organization', text: 'fastrack',
- 4: {id:4, type: 'organization', text: 'faces of Covid',
- }
- const cache = {
- fa: [1, 2],
- fac: [1, 2, 4]
- }
- Pre processing needs to be done before showing the results to the user. But the cost of it is negligible.
- For short lived websites option 1 is better - since cache is cleared often
- For long lived websites, option 3 is better.
- For the key as an empty string add initial results either based on historical searches (FB) or based on popular topic (like google)