Johanna - Project Portfolio

PROJECT: Notably

Overview

Notably is a **note-taking desktop app**, optimized for those who **prefer to work with a Command Line Interface (CLI)** while still having the benefits of a Graphical User Interface (GUI). It aims to help NUS tech-savvy students to get their notes taken down **faster than using traditional GUI apps**.

Summary of contributions

- **Major enhancement**: **Suggestion Engine**
  - What it does: `SuggestionEngine` actively gives users an updated list of suggestions of notes as the users key in their input. Furthermore, it displays a response text to enable users to understand the meaning behind their inputted command.
  - Justification: `SuggestionEngine` plays a pivotal role in providing a seamless user experience. This feature is curated for our target users (NUS tech-savvy students) who often have a large number of notes and thus may find difficulty traversing all of their notes to get to a particular note. This is where `SuggestionEngine` comes in by giving a list of suggestions of the notes that the users want to open or delete, just by typing the first few characters of their intended note.
  - Highlights:
    - The `SuggestionEngine` also provides a command input line auto-fill feature. When the user presses the button to select a suggestion, followed by `Enter`, the command input line will be auto-filled with the suggestion.
    - It displays response text and suggestions (if any) for all available valid commands in Notably. The implementation was quite challenging as different commands require different implementation.
- **Major enhancement**: **Search feature**
  - What it does: The Search feature searches the occurrences of a keyword in all of the notes, and not just the currently opened one. It can also search for partial or incomplete word of the note's content and count the number of times the keyword appears in the note. It then displays the list of suggestions of notes to the users with the most relevant search result at the top of the list.
  - Justification: Students often remember a certain keyword from their note but can't precisely remember where it is located. The Search feature thus gives the users convenience to find the relevant note, as suggestions are sorted based on the keyword's highest number of occurrences. If the number of occurrences is the same, the suggestions listing will based on

their respective positions in the hierarchical notes arrangement.

    - Highlights:
        - The feature is complete as it traverses through all of the notes and can even search for incomplete words, hence giving the relevant suggestions to the user even before the user has finished typing. For instance, if the user types `search lect` while actually intending to find the word "lecture", Notably will still display the list of notes which contain the word "lecture".
        - When the user presses the button to select the suggestion followed by `Enter`, the note chosen will immediately be opened. The command line input is also cleared after the user pressed `Enter` in order to not clutter the UI.

- **Code contributed**: [Functional code]
- **Other contributions**:
    - Project management:
        - Wrote most parts of the README: #51
        - Maintained the issue tracker
    - Enhancements to existing features:
        - Wrote additional tests for existing features to increase coverage significantly: #428, #470
    - Documentation:
        - Updated the Search and Auto-Suggestions feature in UG: #461
    - Community:
        - PRs reviewed (with non-trivial review comments): #106, #127, #279, #291, #432
        - Reported bugs and suggestions for other teams in the class: #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16

Contributions to the User Guide

> *Given below are sections I contributed to the User Guide. They showcase my ability to write documentation targeting end-users.*

Find a note based on certain keywords: `search`

If you need to look for a note that contains a specific keyword in its **content**, use the `search` command and Notably will show you the search results sorted by the number of matches in the note. The note with the highest number of match will be at the top of the list, so that you can access it faster.
Format: `search [-s] KEYWORD`

NOTE

- `search` looks through **all** the notes that you have
- Partial matches work as well! It means that when you have a set of notes containing the word "lecture" but no "lect" as a word on its own, you can just type `search lect` and Notably will still show you a list of notes containing the word "lecture", as "lect" is part of the word "lecture".
- Matches are case insensitive, meaning it will find the word no matter if it is in uppercase or lowercase or even mixed-case

Example: Searching for the keyword "Computer science"

Let's look for the keyword "Computer science" if it exists in any of the notes that you have saved in Notably.

```
search -s Computer science
```

If the word "Computer science" exists, a list of suggestions will be generated. This list will be sorted in descending order of the number of matches, i.e. the note with the highest number of matches will be at the top of the list, as seen in the figure below.

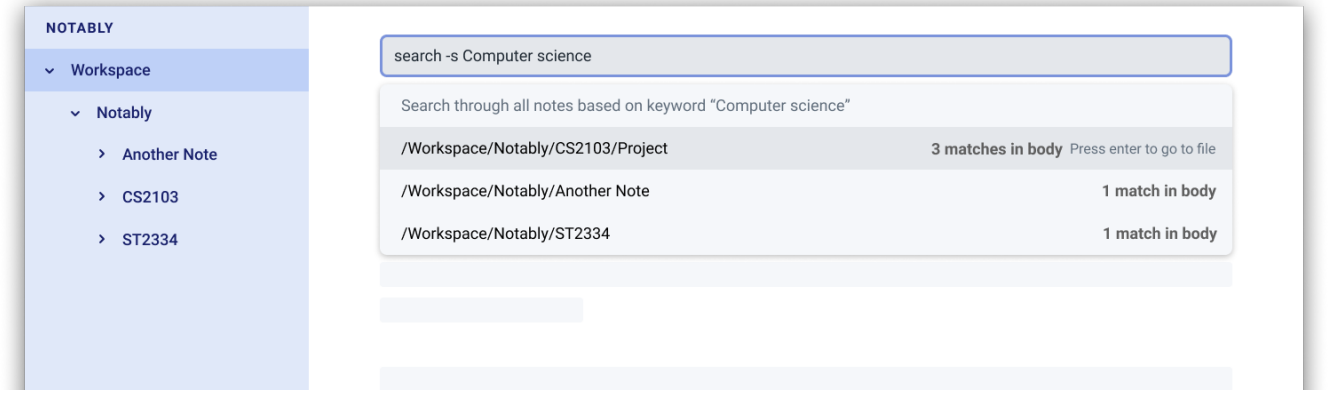

Figure 1. Demo for the `search` command: keyword found

Let's say the first option `/Workspace/Notably/CS2103/Project` is the note you are looking for. You may press kbd:[] and kbd:[Enter] to open the note. The figure below illustrates how the note chosen will be opened and the command line box is cleared.

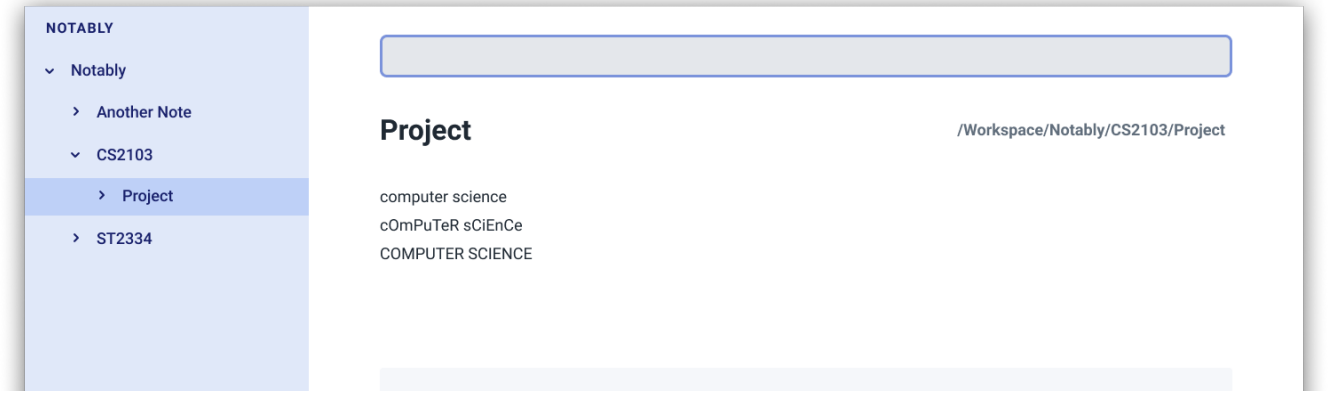

Figure 2. Demo for the `search` command: open a note with the searched keyword

If the keyword you are looking for does not exist in any of your notes, no suggestions will be

generated, as seen in the figure below.

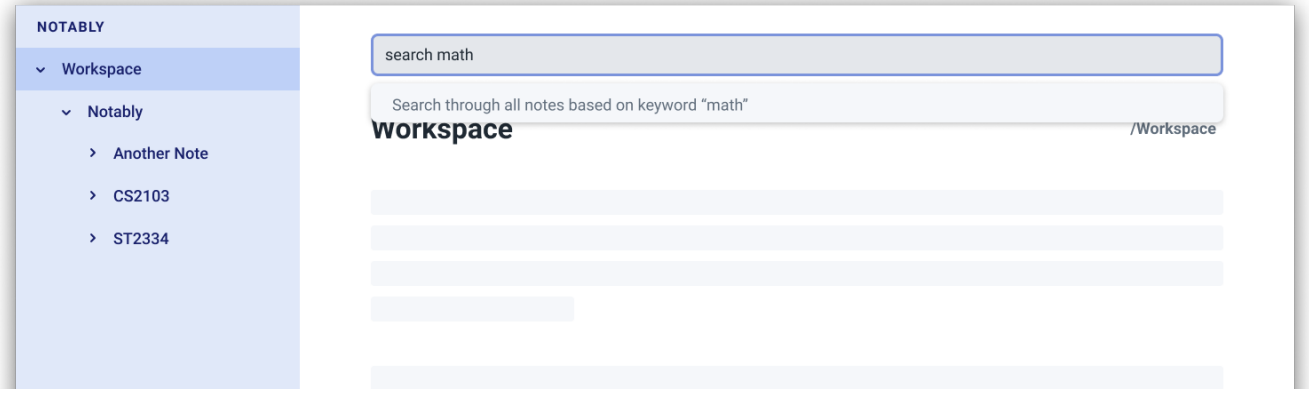

Figure 3. Demo for the `search` command: keyword not found

Auto suggestions

As you type, Notably will provide you with suggestions. You can press the keyboard kbd:[▯] button followed by kbd:[Enter] to select any suggestion in the list.

For example, as you type `open -t Notably`, a list of suggestions will be generated as seen in the figure below.

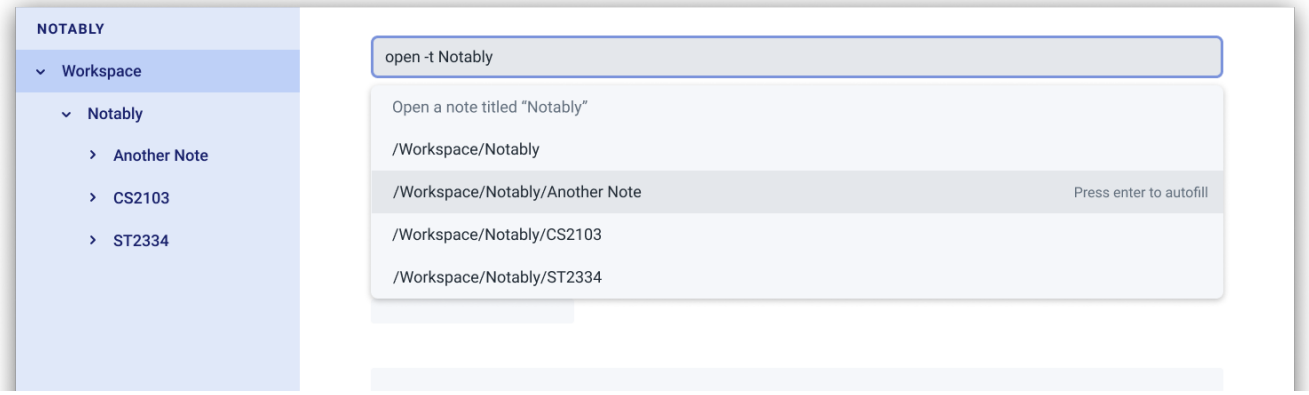

Figure 4. Demo for the suggestions feature: Suggestions are generated as the user keys in his input

Let’s say you would like to choose the option `/Workspace/Notably/Another Note`. After pressing kbd:[▯] to reach that suggestion and pressing kbd:[Enter], the command input line will be auto-filled by the suggestion, as seen in the figure below.

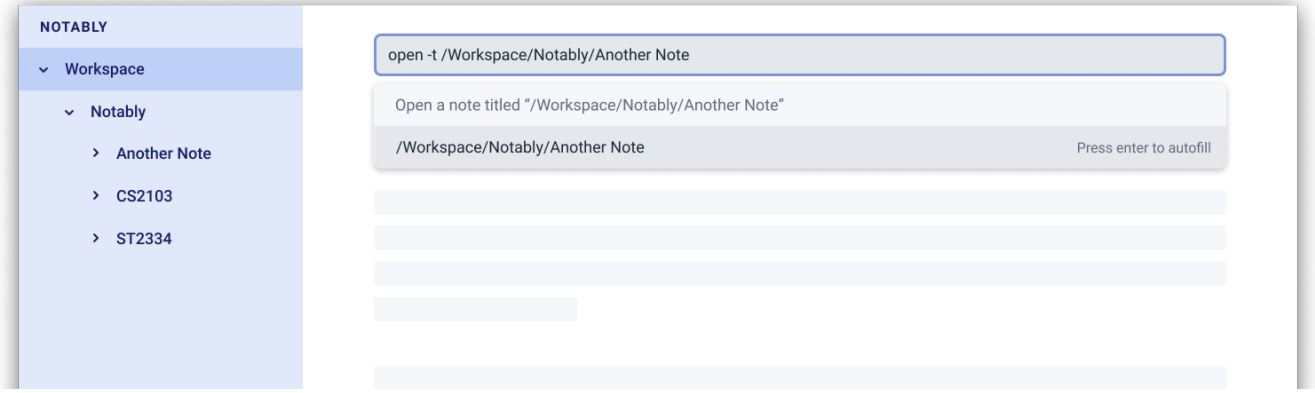

Figure 5. Demo for the suggestions feature: The user input is auto-completed when the user selects a suggestion

If no suggestion list is generated for the command `open`, `delete` or `search`, it means the path, title, or keyword cannot be found anywhere in Notably, as seen in the figures below.

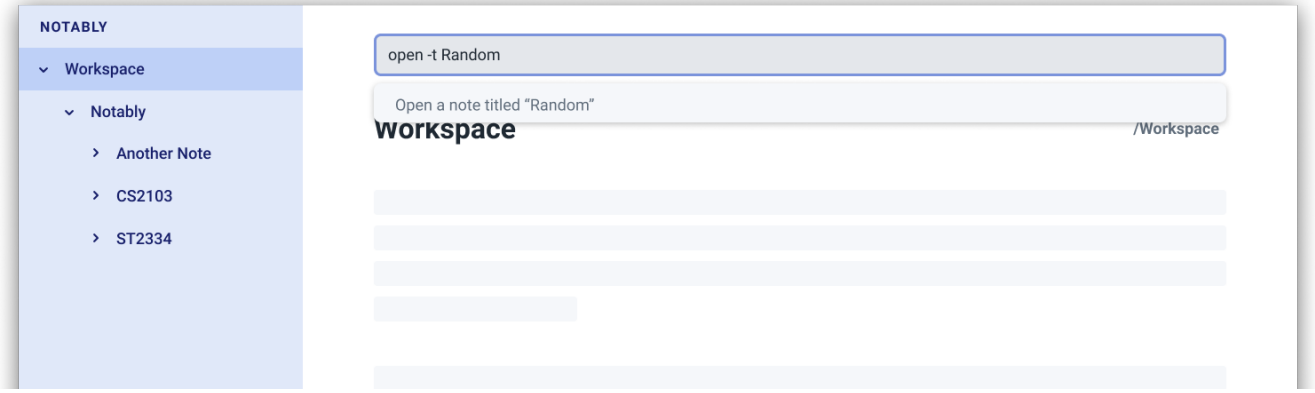

Figure 6. Demo for the suggestions feature: The user inputs a path/ title that does not exist in his Notably app, thus no suggestion list is generated

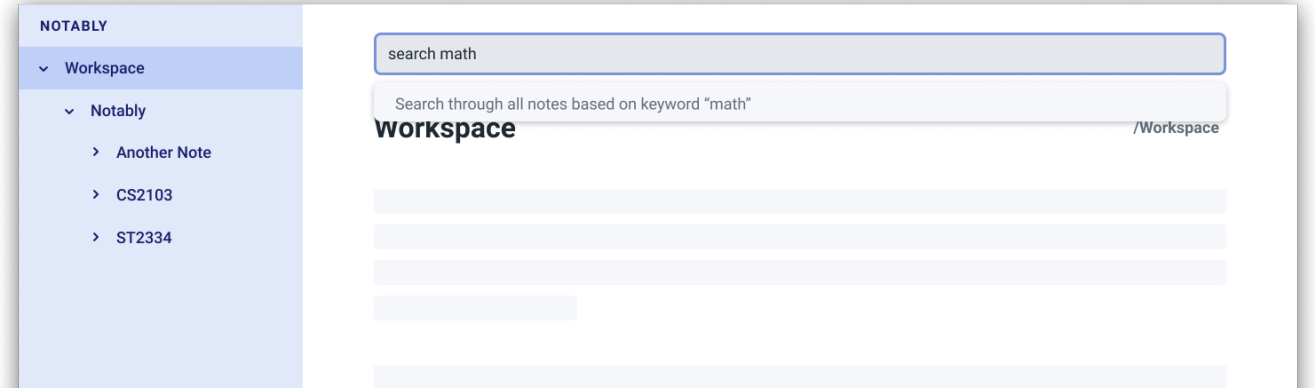

Figure 7. Demo for the suggestions feature: The user inputs a keyword that does not exist in any of his notes, thus no suggestion list is generated

(Coming in v2.0) Suggestion response text when opening or deleting a parent note

Variations of path with `../` (e.g. `open ../`, `open Note/../Note`, etc.) will not generate a comprehensive response text. Currently, typing `open ../` will generate a response text of `Open a note titled "../"` instead of `Open a parent note`.

Response text

Notably also displays a response text which enables you to understand the meaning of the input you type and shows an error message when your input is invalid.

For example, if you type `open /CS`, the response text will indicates that you are trying to `Open a note titled "/CS"`, as seen in the figure below.

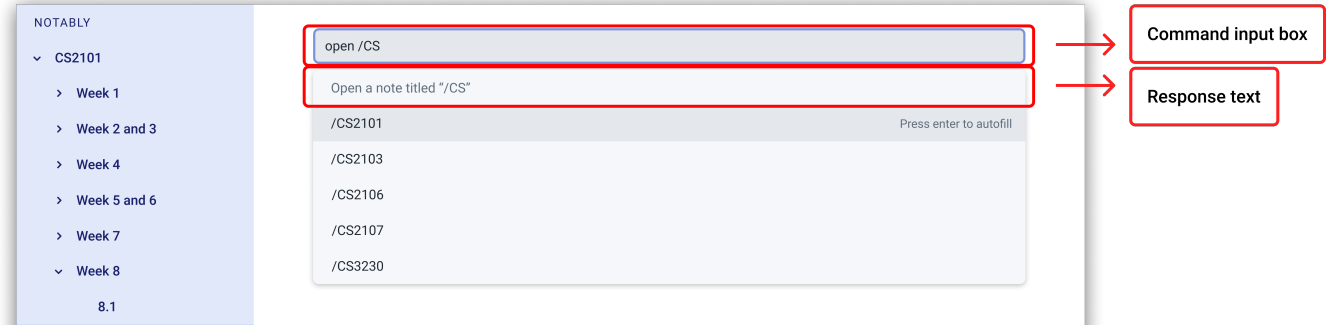

Figure 8. Demo for the response text feature: open command

On the other hand, if you key in an invalid command, the response text will display an error message as seen in the figure below.

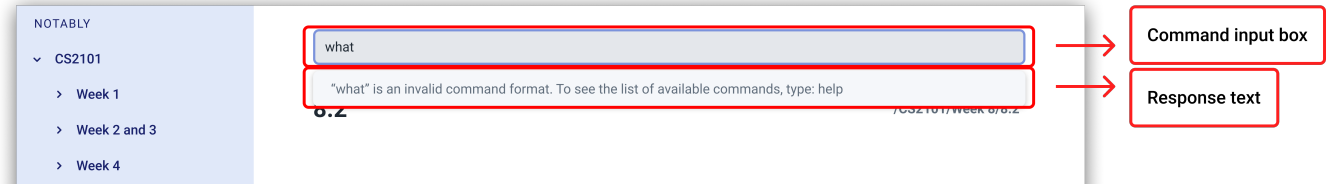

Figure 9. Demo for the response text feature: invalid command

Contributions to the Developer Guide

Given below are sections I contributed to the Developer Guide. They showcase my ability to write technical documentation and the technical depth of my contributions to the project.

SuggestionEngine component

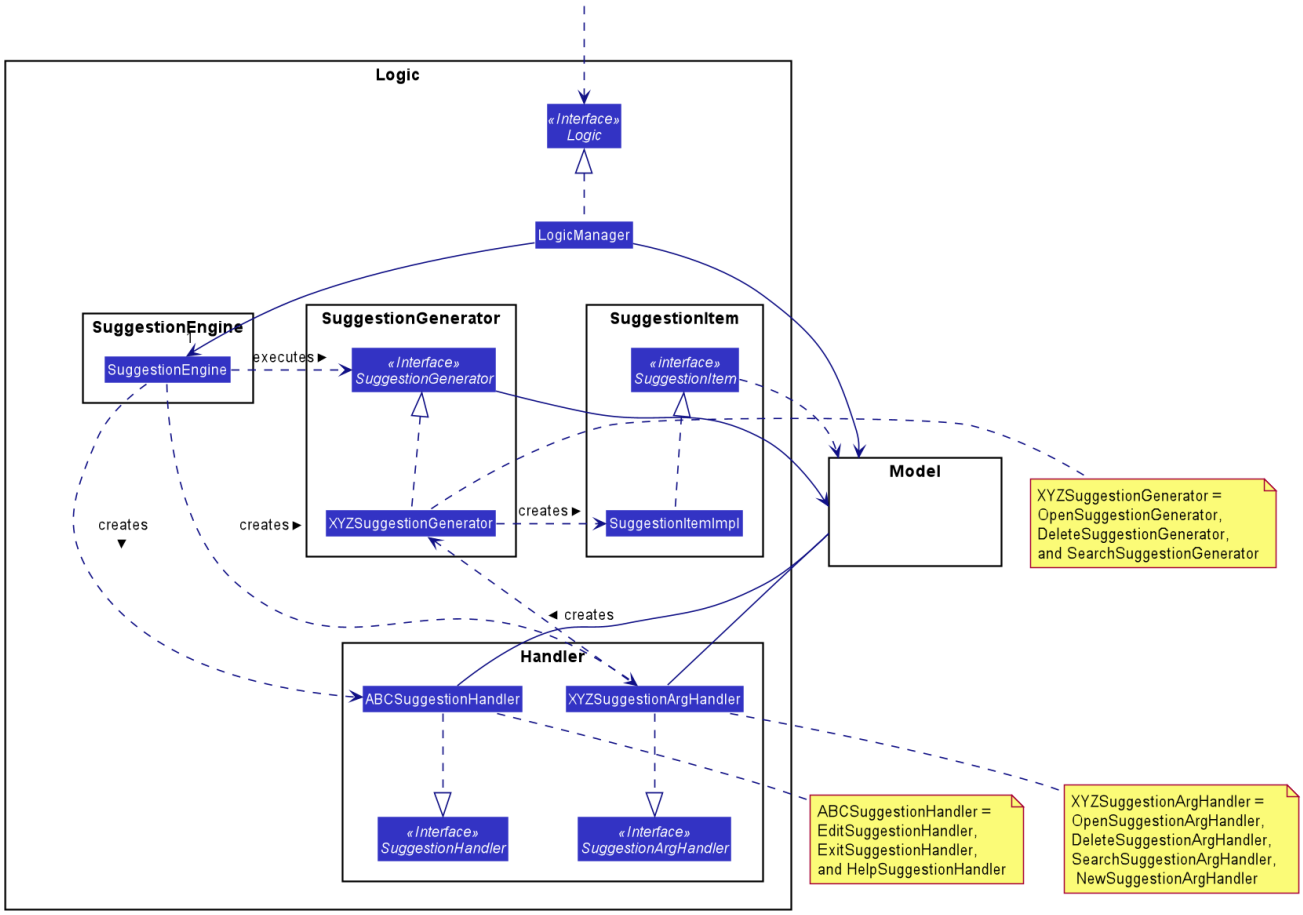

Figure 10. Class Diagram of the Suggestion Engine Component

API : `SuggestionEngine.java`

`SuggestionEngine` gives users the meaning of the command they input and a list of notes suggestions that they want to open, delete, or search.

1. `Logic` uses the `SuggestionEngine` class, to handle the user input.
2. According to the command the user inputs, `SuggestionEngine` will create a `XYZSuggestionArgHandler` or `ABCSuggestionHandler` object which implements `SuggestionArgHandler` and `SuggestionHandler` interface respectively. `XYZSuggestionArgHandler` are for commands that require argument parsing, i.e. `open`, `delete`, `search`, `new`, whereas `ABCSuggestionHandler` are for commands that do not require argument parsing, i.e. `edit`, `exit`, `help`.
3. If `SuggestionArgHandler` object is created: the `responseText` in the `Model` will be updated. This case will also result in the creation of `XYZSuggestionGenerator` object (except for `new` command) which implements `SuggestionGenerator` interface. `XYZSuggestionGenerator` is then executed by the `SuggestionEngine`.
4. If `SuggestionHandler` object is created: the `responseText` in the `Model` will be updated.
5. The `Model` could be affected in 2 ways:
    - Update `responseText` of the `Model` (by the `SuggestionHandler` and `SuggestionArgHandler`): for instance, the input `open /` will set the `responseText` in the `Model` as "Open a note".
    - Store a list of `SuggestionItem` in the `Model` (by the `SuggestionGenerator`).
6. The UI will then be able to retrieve the `responseText` and list of `SuggestionItem` from the `Model` to

be displayed to the user.

Given below is the Sequence Diagram for interactions within the `Logic` and `Suggestion` component for the input `opne /a`.

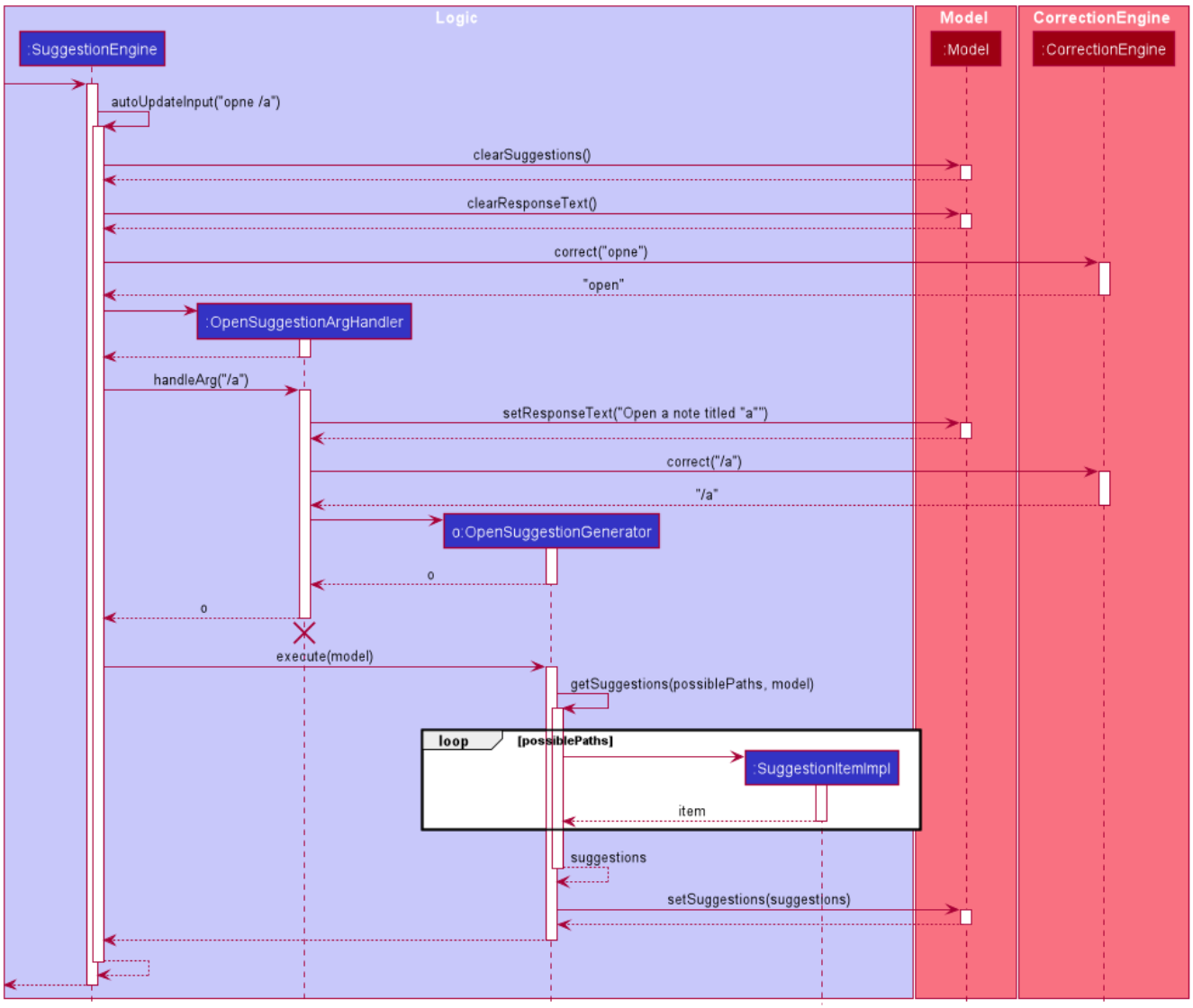

Figure 11. Interactions Inside the Logic and Suggestion Component for the input `opne /a`

NOTE	The lifeline for `OpenSuggestionArgHandler` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

Suggestion Engine

Rationale

`SuggestionEngine` allows the users to traverse their notes conveniently, without having to remember the hierarchical structure of their notes. `SuggestionEngine` gives users the meaning of the command they input and a list of notes suggestions that they want to open, delete, or search.

Current implementation

	SuggestionArgHandler	SuggestionHandler
Purpose	Handles the arguments part of the user input and updates the `responseText` in the `Model` according to the user's command input	Updates the `responseText` in the `Model` according to the user's command input
Commands	`open`, `delete`, `search`, `new`	`edit`, `exit`, `help`
Suggestion Generation	Yes, by `SuggestionGenerator` (except for `new` command, since suggestions are generated based on the existing data in the app)	No

1. `Logic` uses the `SuggestionEngine` class, to handle the user input.
2. According to the command the user inputs, `SuggestionEngine` will create a `XYZSuggestionArgHandler` or `ABCSuggestionHandler` object which implements `SuggestionArgHandler` and `SuggestionHandler` interface respectively. `XYZSuggestionArgHandler` are for commands that require argument parsing, i.e. `open`, `delete`, `search`, `new`, whereas `ABCSuggestionHandler` are for commands that do not require argument parsing, i.e. `edit`, `exit`, `help`.
3. If `SuggestionArgHandler` object is created: the `responseText` in the `Model` will be updated. This case will also result in the creation of `XYZSuggestionGenerator` object (except for `new` command) which implements `SuggestionGenerator` interface. `XYZSuggestionGenerator` is then executed by the `SuggestionEngine`.
4. If `SuggestionHandler` object is created: the `responseText` in the `Model` will be updated.
5. The `Model` could be affected in 2 ways:
    - Update `responseText` of the `Model` (by the `SuggestionHandler` and `SuggestionArgHandler`): for instance, the input `open /` will set the `responseText` in the `Model` as "Open a note".
    - Store a list of `SuggestionItem` in the `Model` (by the `SuggestionGenerator`).
6. The UI will then be able to retrieve the `responseText` and list of `SuggestionItem` from the `Model` to be displayed to the user.

Design considerations

Aspect 1: Design with respect to the whole architecture

1. `SuggestionEngine` is segregated from `Parser` in order to differentiate the logic when the user has finished typing and pressed kbd:[Enter] (which will be handled by `Parser`) in contrast to when the user presses the keyboard kbd:[down] button and kbd:[Enter] to take in the suggestion item.
2. In order to keep the App's data flow unidirectional, `SuggestionEngine` will update the `responseText` (which tells the user the meaning of his command) and the list of `SuggestionItem` into the `Model`. Thus, by not showing the `responseText` and suggestions immediately to the UI, `SuggestionEngine` will not interfere with the `View` functionality.

3. `SuggestionArgHandler`, `SuggestionHandler`, `SuggestionGenerator`, `SuggestionItem`, and `SuggestionModel` are implemented as interfaces, in an attempt to make the design of the `SuggestionEngine` component resilient to change.

Aspect 2: Implementation of suggestions generation

- **Alternative 1:** Have a `SuggestionCommandParser` interface and `SuggestionCommand` interface to parse each of the command, update `responseText` in the `Model`, and give suggestions.
    - Pros: This provides a consistency for all the commands, where each command has a `XYZSuggestionCommandParser` and `XYZSuggestionCommand` class.
    - Cons: The `SuggestionCommandParsers` of the commands that do not require parsing of user input (`edit`, `exit`, `help`) end up passing a `userInput` argument that is not being used anywhere, which makes this design unintuitive. Moreover, since the updating of the `responseText` in the `Model` can be done in each `SuggestionCommandParser`, the `SuggestionCommand`s of `edit`, `exit`, and `help` end up to be redundant.
- **Alternative 2 (current choice):** Create 2 separate interface to handle commands with input parsing and those without, and name it as a `SuggestionArgHandler` and `SuggestionHandler` respectively.
    - Pros: This solves the cons discussed in Alternative 1, as this design gives a separate implementation for the commands with input parsing and those without. It does not force the `Handler` to parse the user input when there is no need to. The naming `Handler` also does not restrict the functionality of the interface and classes to just parse an input, but allows for a flexibility in executing other functionality such as updating the `responseText` in the `Model`.

Use case: Search notes using the Auto-suggestion feature

MSS

1. User types in a keyword of a note's content that he wants to open.
2. Notably lists out the relevant search results, with the most relevant at the top of the list (based on the keyword's number of occurrences in the note).
3. User chooses one of the suggested notes.
4. Notably opens the chosen note.

   Use case ends.

Extensions

2a. No suggestion is being generated.

2a1. Notably displays a response text, indicating that the user is trying to search through all of the notes using that particular keyword.

2a2. Since the empty suggestion conveys that the keyword cannot be found, the user enters a new data.

Steps 2a1-2a2 are repeated until the data entered is correct. Use case resumes from Step 3.

Use case: Open/ Delete notes using the Auto-suggestion feature

MSS

1. User types in an incomplete path or title of a note.
2. Notably lists out suggestions of notes.
3. User chooses one of the suggested notes.
4. Notably opens/ deletes the chosen note.

   Use case ends.

Extensions

1a. Path or title contains invalid character(s) (symbols `-` or `` ` ``)

1a1. Notably displays a response text, indicating that the path or title is invalid.

1a2. User enters a new data.

Steps 1a1-1a2 are repeated until the data entered is correct. Use case resumes from Step 2.

1b. Path or title does not exist

1b1. Notably displays a response text, indicating that the user is trying to open/ delete the note with the particular path or title that the user inputs.

1b2. Notably does not generate any suggestions, which means the note cannot be found.

1b3. User enters a new data.

Steps 1b1-1b3 are repeated until the data entered is correct. Use case resumes from Step 2.