Email Manager - Developer Guide

By: `Team AY1920S1-CS2113T-F11-3` Since: `Sept 2019` Licence: `MIT`

1. Introduction

Welcome to the **Email Manager** Developer Guide!

Email Manager is an email and task manager app, specifically designed for NUS School of Computing Students to manage their emails and busy schedules. As a text-based application, it is optimized for those who prefer typing and working with Command Line Interface (CLI). Email Manager also has a developed Graphical User Interface (GUI) that allows users to view email and task details in an appealing, well-organized format.

2. About the Developer Guide

This developer guide provides detailed documentation on the implementation of all the various features Email Manager offers. It also suggests methods for you to modify and build upon it.
Throughout this developer guide, there will be various icons used, as shown below:

	This is a tip. Follow these tips to aid your development of Email Manager.

□□	**This is a note. Read these for additional information.**

□□	**This is a warning. Heed these warnings to avoid making mistakes that will hamper your development efforts.**

3. Setting up

This section shows how to set up Email Manager on your desktop and begin your development journey.

4. Design

4.1. Architecture

4.1.1. How the architecture components interact with each other

The *Architecture Diagram* given above explains the high-level design of the App. Given below is a quick overview of each component.

4.1.2. UI component

4.1.3. Logic component

4.1.4. Model component

4.1.5. Common classes

5. Implementation

This section describes some noteworthy details on how certain features in **Email Manager** are implemented.

5.1. Task Management

There are three main types of tasks that Email Manager works with: `todo`, `deadline` and `event`.

5.1.1. Natural Dates Support

The email manager aims to help computing students handle their tasks efficiently. Therefore, one of its main goals is to speed up the process at which students enter their task details so that their task can be added into the task list quickly. The benefits of having this Natural Dates support are:

- Reduce the time and effort needed to key in the date and time for deadline and event tasks.

The Natural Dates support is facilitated by two main classes, namely `TaskCommandParseHelper` and `TaskParseNaturalDateHelper`.

`TaskParseNaturalDateHelper` is an element of the will retrieve the parsed time string from `TaskCommandParseHelper` and convert the extracted string to LocalDateTime format. It implements the following operations:

- `TaskParseNaturalDateHelper#isCorrectNaturalDate(input)` - Checks if input contains natural date format.
- `TaskParseNaturalDateHelper#convertNaturalDate(day, time)` - Converts string day and time to local date in LocalTimeDate format.
- `TaskParseNaturalDateHelper#getDate(input)` - Returns a date and time(if applicable) after checking if natural date input contains a time element.

`TaskCommandParseHelper` is an element of the Command component. It handles all parsing of inputs when the input type is set to `task`.

Given below is an example usage scenario and how Natural Dates Support behaves at each step.

Step 1: The user launches the application. The input type is currently in `email` mode. The user wishes to add a task and key in `flip` to switch input type to `task` mode.

Step 2: The user executes `deadline homework -time Mon 1200` to add a new deadline task.

- `TaskCommandParseHelper` takes in the command, parses and extracts the date and time information of the task and saves it inside a list of type `ArrayList<Command.Option>`.
- The extracted date and time will go through `TaskParseNaturalDateHelper#getDate()`, which calls the relevant methods in the class to process the date and time.

Step 3: The user wishes to update the date and time for the task above, let the task above be task 1 in the task list. The user executes `update 1 -time Tue` to change the task date from Mon to Tue.

- `TaskParseNaturalDateHelper` will be called to process the time information and update the task accordingly. When no time is entered, the time is set to 0000 (HHmm).

5.1.2. Updating of Tags

Tasks have a variety of attributes that a user may want to add or change. The current attributes that can be modified are: `time`, `doafter`, `priority` and `tags`. The updating of task details is facilitated by `TaskUpdateCommand`. It extends `Command`, overriding the `execute` method.

The details of the update mechanism are as follows:

Step 1: `TaskCommandParseHelper` takes in the command from the user input text field, and parses it. If the command starts with `update` than it will separate all the parameters into an ArrayList, passing it into the `TaskUpdateCommand`.

Step 2: `TaskUpdateCommand.execute()` will then go through the ArrayList, calling the appropriate method in `TaskList`.

Step 3: The method in `TaskList` will retrieve the specified task and call the related setter to change the value.

5.2. Email Management

5.2.1. Email Auto Parsing

The emails fetched or stored locally will be automatically parsed to extract important information for tagging, task creation and reminder purposes. The parsing consists of two stages, the **format parsing** and **content parsing**. Email format parsing is to parse the email components like subject, sender and body from the raw string fetched from the server or stored in local file. The content parsing is to parse the keyword included all components of email.

Email Format Parsing

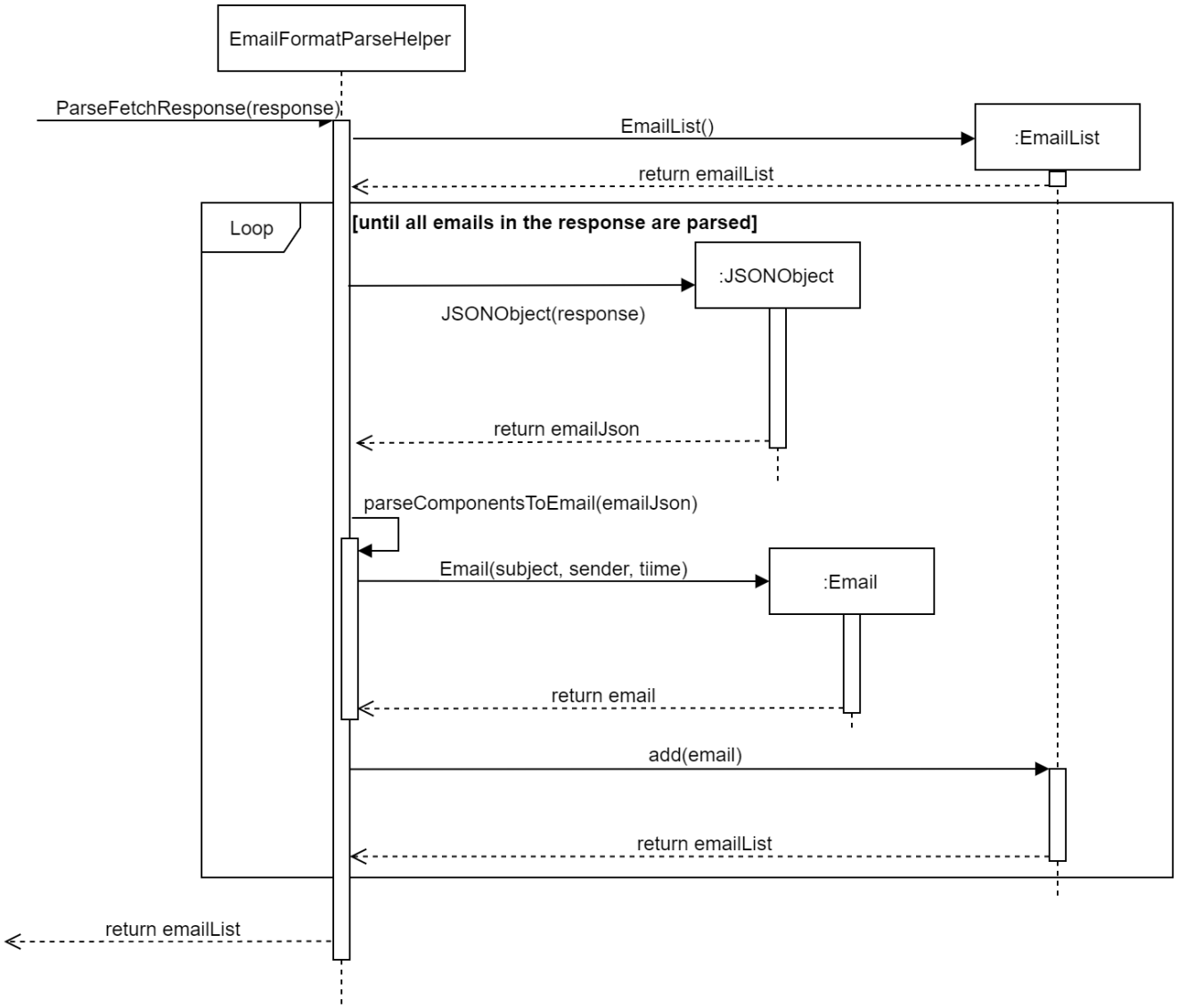

Figure N: Email Format Parsing

The email format parsing starts at the call of `ParseFetchResponse(response)`. The variable `response` here is a string of the http response from the Outlook server to the fetch API call.

It first creates an `EmailList` to store all the emails parsed from the response. This function only

returns the list instead of directly adding the email parsed to the model because the storage or network component will decide whether and how the emails are to added to the model.

Then each email contained in the response of is parsed to a `JSONObject` called `emailJson` for easier manipulation.

With this `emailJson`, `parseComponentsToEmail(emailJson)` is called to extract different components of the json and instantiate an `Email` object to be added to the `emailList` created earlier.

This process repeats until all the email information in the response is processed.

Email Content Parsing

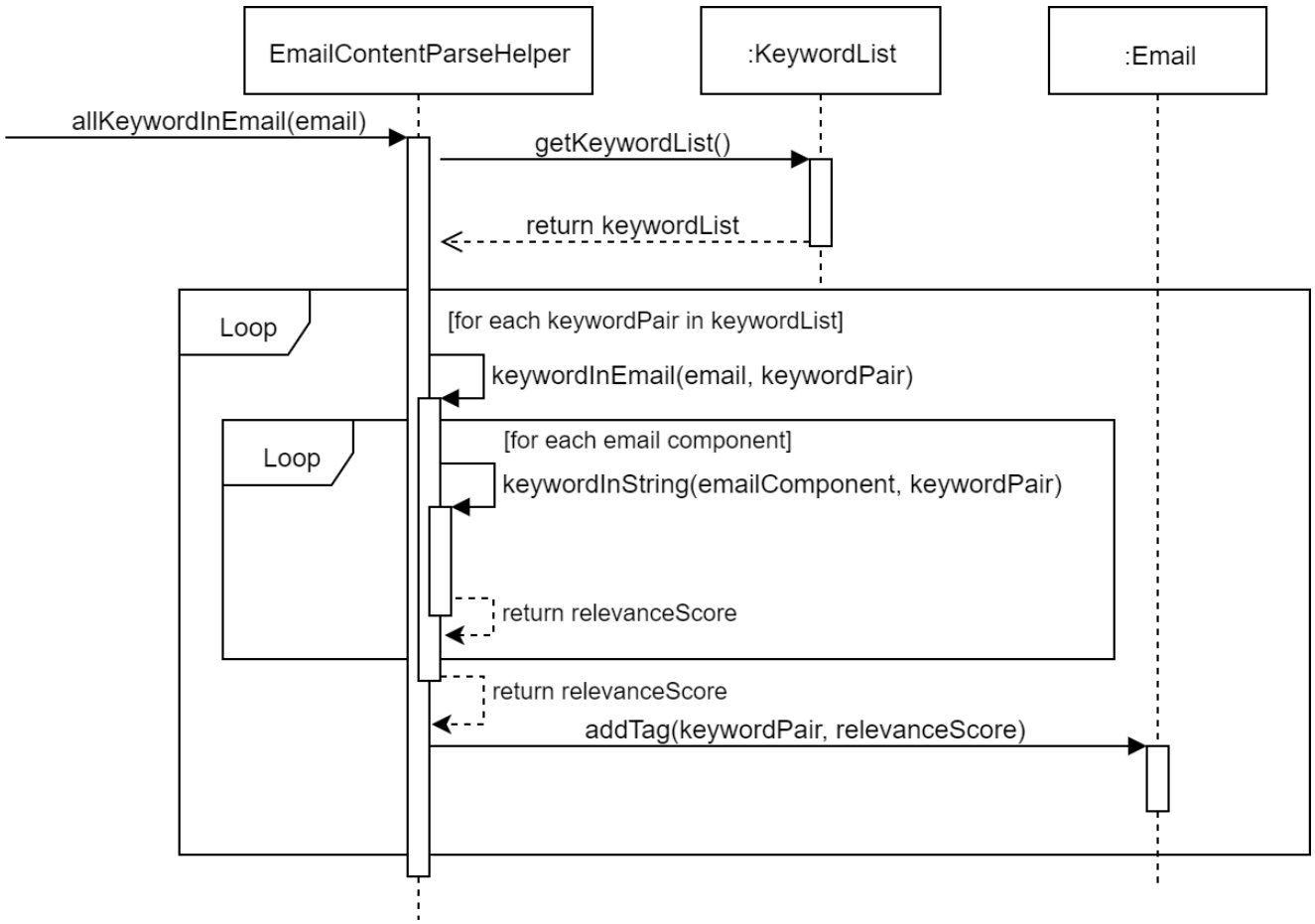

Figure N+1: Email Content Parsing

5.2.2. Email Tags (Tagging and Searching Emails)

This section describes how Email Manager managing emails by tags, including tagging of email and searching of emails with the relevant tags.

Tagging Email: `update`

- Format: `update ITEM_NUMBER [-tag TAG1] [-tag TAG2]`
- Tags the specified item with the tag(s) minimum number of tags is 1.

Filtering email: 'list`

Format: `list [-tag TAG1] [-tag TAG2]`
Gives a list of emails with the tags. Minimum number of tags is 1, and the maximum is 2.

The tags associated with emails is stored in HashMap<String, HashMap<String, EmailList>>, where the String here refers to the tag name, and EmailList is an ArrayList<Email>.

6. Documentation

7. Testing

8. Dev Ops

Email content parsing is to parse the keyword from different components of an email.

Email content parsing starts with the calling of `allKeywordInEmail(email)` and parse in the email in `Email` type.

It then gets all the `KeywordPair` from the `KeywordList`. A `KeywordPair` contains a human readable `keyword` signifying the "meaning" of the keyword. It also contains multiple `expressions` which will be looked through the email for matches.

For each email component (subject, sender, body etc), the `keywordInString(emailComponent, keywordPair)` is called to search for matches across these components. It will then return a relevance score. Higher relevance score means a more occurrence. Keyword with all its expressions absent in the email will get a 0 relevance score. Also, subject and sender will have a higher weights compared to the email body.

Each keyword searched in the email with a relevance score higher than 0 (at least 1 occurrence), will be added to the email as a tag by calling `addTag(keywordPair, relevanceScore)`.

8.1. Design considerations

Aspect: Handling updating of tags * Alternative 1 (current choice): if at least one tag is entered as part of the command, it will overwrite all current tags of the task being modified. **Pros: makes it consistent with other parameters of the update command and gives users an option to replace/remove tags** Cons: if there are many tags, and the user only wants to add on an extra tag, the user will need to retype all existing tags into the command. * Alternative 2: if a tag is entered as part of the command, it will amend on top of existing tags. **Pros: this will save users time if they only want to add on tags** Cons: no option to remove tags * Alternative 3 (proposed): there will be an option to add tags and an option to remove tags. **Pros: gives users highest amount of flexibility and control over the tags they want to keep.** Cons: added complexity in commands

8.2. Logging

Appendix A: Suggested Programming Tasks to Get Started

Appendix B: Product Scope

Target user profile:

1. National University of Singapore (NUS) School of Computing Students.
2. Busy computing student who is tired of receiving too many emails.
3. Busy computing student who has a lot of todos, deadlines and events.
4. Students who prefer desktop apps over other types.
5. Students who prefer typing over other means of input.

Value proposition:

1. Helps busy computing student to manage their emails.
2. Helps busy computing student to manage their tasks and schedules.
3. Reminds busy computing students of their important emails and tasks.

Appendix C: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

As a/an	I can	So that…	Priority
active student	get emails filtered out on student-life activities	I can get the interesting event info immediately	* * *
advanced user	tag the emails	I can search for them efficiently	* * *
busy student	prioritize my tasks by setting priority levels	I can work on more pressing task first	* * *
busy student	assign emails with color codes according to priority	I can have a clear view of priorities	* * *
computing student	filter out different types of emails by specifying the type we want to filter	I can access the type of emails we want easily	* * *
computing student	sort email according to module code	I can easily access the information related to my project	* * *

As a/an	I can	So that…	Priority
computing student	set auto delete function to delete emails from a specific address	I can keep my mailbox clean	* * *
computing student	filter out emails by specifying a keyword	I can access the email I am interested easily	* * *
computing student	auto-categorize the emails	my mailbox is not messy	* * *
email sender	send emails with tags	recipients can filter emails easily	* * *
email user	get the list of unread emails	I can attend the unread easily	* * *
job hunting student	add alarm to job/internship application deadlines	I will not miss any important application deadlines	* * *
advanced user	advance search based on Regular expression	I can search with complex filters	* *
advanced user	use shorter versions of command and auto-completion of command	I can type faster command	* *
busy student	check my calendar to see if there are new added	I can keep track of my task efficiently	* *
busy student	highlight tasks that are due soon (<24 hours)	I can work on things that are more pressing	* *
busy student	set alarm to review some important emails	I can remember to attend to some important emails that I don’t have time to handle now	* *
busy student	sync with NUSMODS to automatically set deadlines for homework	I can my deadlines or homework assigned to a specific time	* *
busy student	undo my previous command	recover to the previous state	* *
computing student	set important emails to reply by a specific date	I won’t miss any important deadlines	* *
computing student	update my calendar if the email contains a date	I won’t miss out important deadline	* *

As a/an	I can	So that…	Priority
computing student	download all uploaded files sent through emails	I can get the latest version of the file	* *
computing student	get connected with list on contacts on email	I can easily send to or find the email user	* *
computing student	filter out competition/hackathon emails	I won't miss any interesting competition	* *
computing student	find teams for competition/hackathon	I can quickly find teams after the competition email is sent out	* *
email user	send, forward or reply to email	I do not need to switch to email app after reading from this app	* *
email user	highlights links, action items	I can take action and quickly get to a website	* *
email user	automatically restore emails that were thrown to the junk mail by the system	I won't miss any important emails just because they are in the junk mail without me knowing it	* *
job hunting student	put away all outdated emails on internships/jobs	I can focus on the newest and valid ones	* *
job hunting student	get the jobs and internship emails sorted	I can find a job	* *
team member	tag emails with project stages	I can access emails from different stages of our project	* *
team member	sort all emails from my team members together	I can easily access the information related to my project	* *
team member	send progress tracking emails to other team members periodically	The team can be always updating each other's progress	* *

Appendix D: Use Cases

System: `Email Manager`

Actor: `User` (SoC student)

Use Case: Fetch email from account

MSS

1. User starts system or enters the fetch command.
2. System retrieves account key from file, connects to Microsoft and logs in. New emails are retrieved, combined with those from local storage and displayed.

   Use case ends.

Extensions

- 2a. System is unable to retrieve a valid account key, receives error from Microsoft.
  - 2a1. Opens Microsoft portal in browser.
  - 2a2. The user types in the username and password into Microsoft portal.
  - 2a3. System saves the account key from Microsoft for future logins, downloads new emails, and displays email.

    Use case ends.
- 2b. System does not receive a response from Microsoft server.
  - 2b1. System notifies user of failure to log in and loads email from local file.

    Use case ends.

Use Case: Set priority to different keyword

MSS

1. The user selects the “Keyword Priority” from the menu
2. System presents the user with all current priority settings
3. The user selects “New” from the menu
4. System displays a text box for input of keyword/regular expression.
5. The user types in the keyword/regular expression, selects the priority level of this keyword and selects “Save” and confirm

   Use case ends.

Use Case: View Email List with Priority

MSS

1. The user selects “All Email” from the menu

2. The user selects “by Priority” from the menu
3. System displays all the priorities and keywords under each priority
4. The user selects the priority range that he/she wants to view
5. System leads the user back to the email list page

   Use case ends.

Extensions * 3a. The user can include or exclude a particular keyword from that priority

+ Use case ends.

Use Case: Auto categorisation of emails

MSS

1. User creates a new categorize name.
2. User specifies the keyword for this category.
3. The app will look through the emails and put the related-emails under the category.

   Use case ends.

Appendix E: Non-Functional Requirements

Email Manager meets the following non-functional requirements:

- Security of user login credentials (user enters details directly into Microsoft portal)
- Data Accessibility (efficient storage)
- Time-out
- Clean layout
- Works with common operating systems

Appendix F: Glossary

Mainstream OS

- Windows
- macOS
- Linux