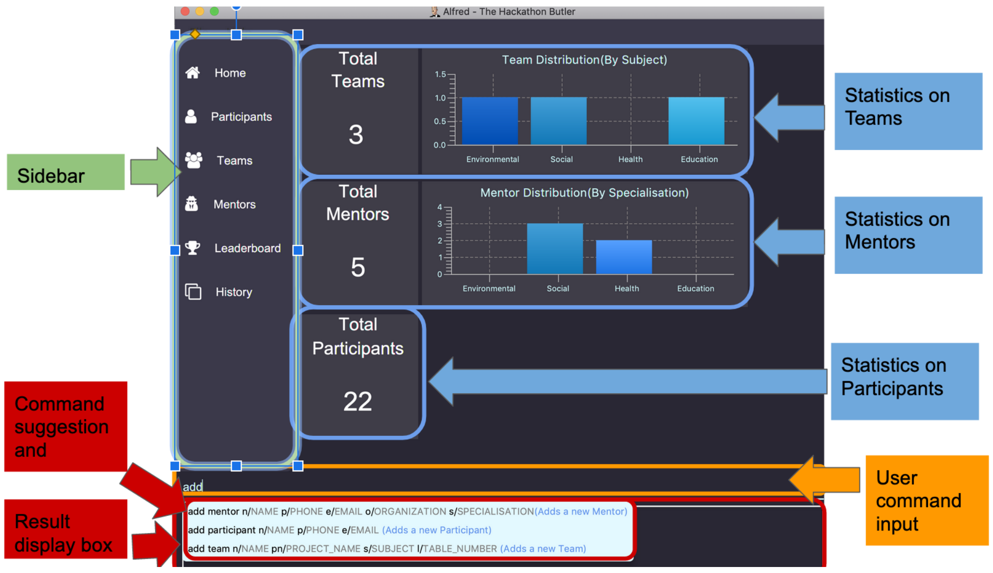
John (Kyungho) Min - Project Portfolio

PROJECT: AddressBook - Level 3

Overview

Alfred is a desktop application used for organizing hackathons. This project was morphed from `SE-EDU Addressbook - Level 3`, which you can find the link here. This application allows the user to easily manage the relationships between teams and participants/mentors taking part in an hackathon event along with providing a simple mechanism for judging and scoring of teams. It is written in Java with more than 25 kLoC.

This is what our project looks like:



```
add
add mentor n/NAME p/PHONE e/EMAIL o/ORGANIZATION s/SPECIALISATION(Adds a new Mentor)
add participant n/NAME p/PHONE e/EMAIL (Adds a new Participant)
add team n/NAME pn/PROJECT_NAME s/SUBJECT l/TABLE_NUMBER (Adds a new Team)
```

Figure 1. The GUI for **Alfred**

I implemented the `import` and `export` features for **Alfred**. The following sections will delve deeper into the enhancements I have made along with the relevant documentations I have added to the user and developer guides.

Note the following format used in this document:

`Words` in red	[small]#Indicates that this word is either associated with written code, parameters for commands, or Java classes.
Words in `UPPER_CASE`	Indicates a parameter that you need to supply for a command.
Words in `{braces}`	Indicates parameters for a command that you can select from. i.e. {A/B/C} indicates that you may either choose A, B, or C.
Words in `[brackets]`	Indicate parameters that are optional for the command. i.e. `add [param]` indicates that both `add` and `add param` commands will work.
`Entity`	Refers to a collective set of Participants, Mentors, and Teams.

Summary of contributions

Major Enhancement 1: added the ability to bulk register entities into **Alfred** through CSV files.

- **What it does:** Allows the user to add multiple entities into **Alfred** at once instead of one command at a time.

- **Justification:** This feature improves the product significantly as it allows the user to keep track of registered entities in CSV files and mass import it after registration period.
- **Highlights:** This enhancement requires the user to keep a correctly formatted CSV file for this command to work properly. Rules for formatting is included in our user guide.

Major Enhancement 2: added the ability to export all the data in **Alfred** to an external CSV file.

- **What it does:** Allows the user to keep a record of past hackathon events via a one-line command.
- **Justification:** For big events such as hackathons, it will be important to keep a record of each hackathon event. This documentation can be used for many purposes, including sharing aggregate data with other organizations and analyzing past events to predict necessary investments for future events.
- **Highlights:** The user can choose to export data regarding a certain entity type by specifying it in the command. This may be useful if the user wants to run statistical analysis regarding a specific entity type.

Minor enhancement: morphed existing `add`, `delete`, and `edit` commands to fit the class structure of **Alfred**

- **Highlights:** Although this is included as a minor enhancement, it required creation of 12 new Java classes along with syncing of new methods exposed by our new manager class.

Code contributed: [Contributions]

Other contributions:

- Project Management:
  - Managed README.adoc, AboutUs.adoc, ContactUs.adoc, and our team profile pictures for our team showcase (Pull requests: #29, #36)
- Enhancements to existing features:
  - Updated existing utility classes to abstract commonly used methods (Pull requests: #14 #116)
  - Morphed existing commands to fit with our project's structure (Pull requests #2, #4, #14, #63)
- Documentation:
  - Added `ImportCommand` and `ExportCommand` sections in our user guide and developer guide: (Pull request: #349)
- Community:
  - PRs reviewed with non-trivial review comments: (#7, #13 … #159 …)
  - Reported bugs and suggestions for our team (Issues: #324, #325, #326, 2, 3)
  - Fixed bugs raised by my teammate and people from other teams (Issues: #270, #272, #322)
- Tools:
  - Integrated a third party library (Netlify) to the project (README.adoc commit)

Contributions to the User Guide

The following block shows an excerpt from our user guide showing my contributions to the documentation of various constraints of parameters used in our application. It is basically like a glossary of parameters to be used in **Alfred** commands.

Entity Types

- **Mentor**
  - A mentor has a name, phone number, email address, an organization, and a specialty (i.e. subject name) which must take on values mentioned below under "Parameters."
  - Two mentors are considered as the same mentor if and only if they have the same names along with one of their phone numbers or emails.
- **Participant**
  - A participant has a name, phone number, and an email address.
  - Two participants are considered as the same participant if and only if they have the same names along with one of their phone numbers or emails.
- **Team**
  - A team has a team name, a subject to focus on, score, its project name, and its location (table number).
  - A team may also contain one mentor and an arbitrary number of participants (setting of restrictions coming in v2.0).
  - Two teams are considered as the same team if and only if they have the same team names or project names.

All entities will also receive a unique ID, which means no two entity will share the same ID. That is, entities with same ID will also be considered as a same entity. This is to be considered for specific commands such as the [CSV File Formatting] under the Bulk Registration command.

Parameter Constraints

`Name`, `Organization`, `ProjectName` - can be any combination of spaces, letters, and these special characters (,.-')

`Phone` - can be any combination of numbers (at least three digits), space, hyphens (-), and periods (.) with or without a country code. Country code of Singapore (`+65`) will be automatically included if it is not added.

`Email` - must include an address and an email domain. It can include special characters (-,.), excluding bracket.

`SubjectName` - the subject a mentor or team will be focusing on in the Hackathon
Must be one of the values below:

- Environmental
- Social
- Health
- Education

- Entertainment
- Social

`Score` - must be an integer ranging from 0 to 100.

`Location` - must be an integer ranging from 1 to 1000, indicating a table number. Only one team is allowed per table

---

From here on, I included an excerpt of my contributions to the documentation of the usage of `ImportCommand` and `ExportCommand` in our user guide.

Bulk Registration: `import fp/PATH_TO_CSV_FILE [fp/PATH_TO_ERROR_FILE]`

You may import multiple entities at once into Alfred through the specification of a CSV file.
If the PATH_TO_ERROR_FILE is specified, Alfred will create a new CSV file with all of the lines that were not able to be loaded.

Example:

- `import fp/C:/User/Hackathon2019/participant.csv` will import data from the participant.csv file into Alfred.
- `import fp/Hackathon2019/participant.csv` will look for the CSV file in your current directory (or the folder where alfred.jar is downloaded).

TIP

First, locate the desired file in your respective file manager.
On Windows, hold down `shift`, click the file, then click `Copy as path` to copy its file path.
On Mac, right-click the file, hold down `OPTION` key, then click `Copy (item name) as Pathname` to copy its file path.

Export Data: `export [{mentor/participant/team}] [fp/DESIRED_CSV_FILE_PATH]`

You may export Alfred data to an external CSV file. If the entity type is specified, Alfred will export all the data corresponding to that entity type only. If the desired CSV file path is left empty, Alfred will create a CSV file at the default location (`./AlfredData/Alfred_Data.csv`).

Example:

- `export` will export all entities' data in Alfred to the default file path: `/AlfredData/Alfred_Data.csv`.
- `export mentor fp/data/Alfred.csv` will export all mentor data in Alfred to `/data/Alfred.csv`. If the any folders do not happen to exist, Alfred will create them for you.

Contributions to the Developer Guide

The following block shows an excerpt from our developer guide showing my contributions to the documentation of how the `ImportCommand` was implemented.

Bulk Registration

The Bulk Registration feature, referred as the import command, allows you to add multiple entities into Alfred at once through a CSV file. The file must be stored locally as Alfred will attempt to retrieve it through the file path provided by the user. In order for the import command to successfully execute, it is required that the CSV file is formatted according to Alfred's requirements, which you can read more about in our user guide.

This feature will be explained further in the following subsections.

Implementation Overview

Since this feature manages data from a CSV file, import command relies on the `CsvUtil` class. The `CsvUtil` class handles reading from and writing data to a CSV file. Below shows the relationships between different classes in Alfred.

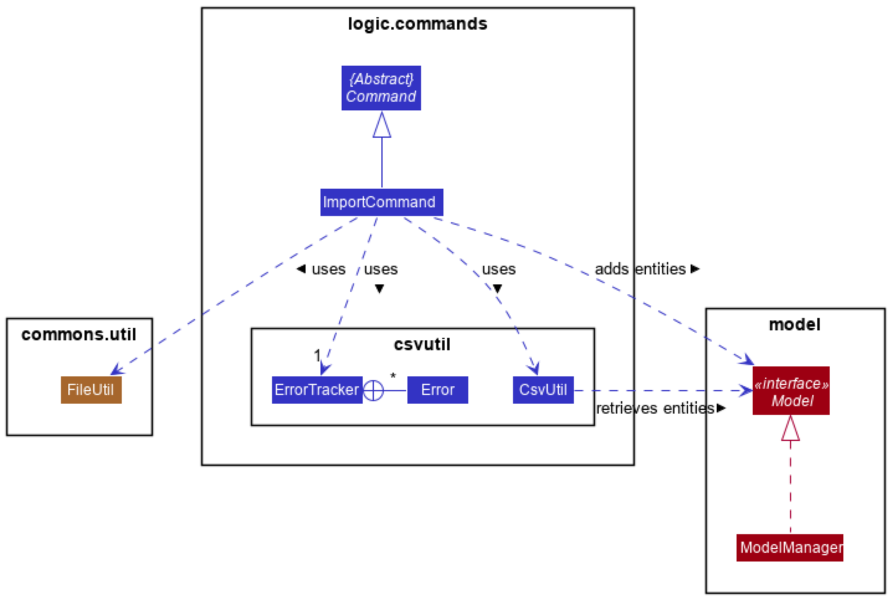


Figure 2. Import Command Class Diagram

In the above class diagram, you can see that

1. The `ImportCommand` uses the `FileUtil` class, and this is so for a number of reasons. First is to validate whether user inputted file path is, in fact, a valid file path. Once it is verified, another check is done to see if the file exists at the given file path. If the file is not able to be located, the `ImportCommand` will not complete its execution.

2. In addition to the `CsvUtil` class, the `ImportCommand` also utilizes an `ErrorTracker` class. This class will store any lines in the CSV file that is invalid along with the reason why it is so. Each `Error` object referenced by the `ErrorTracker` will correspond to one line in the CSV file and the cause of the error.

The activity diagram below will explain the overall flow of `ImportCommand`.

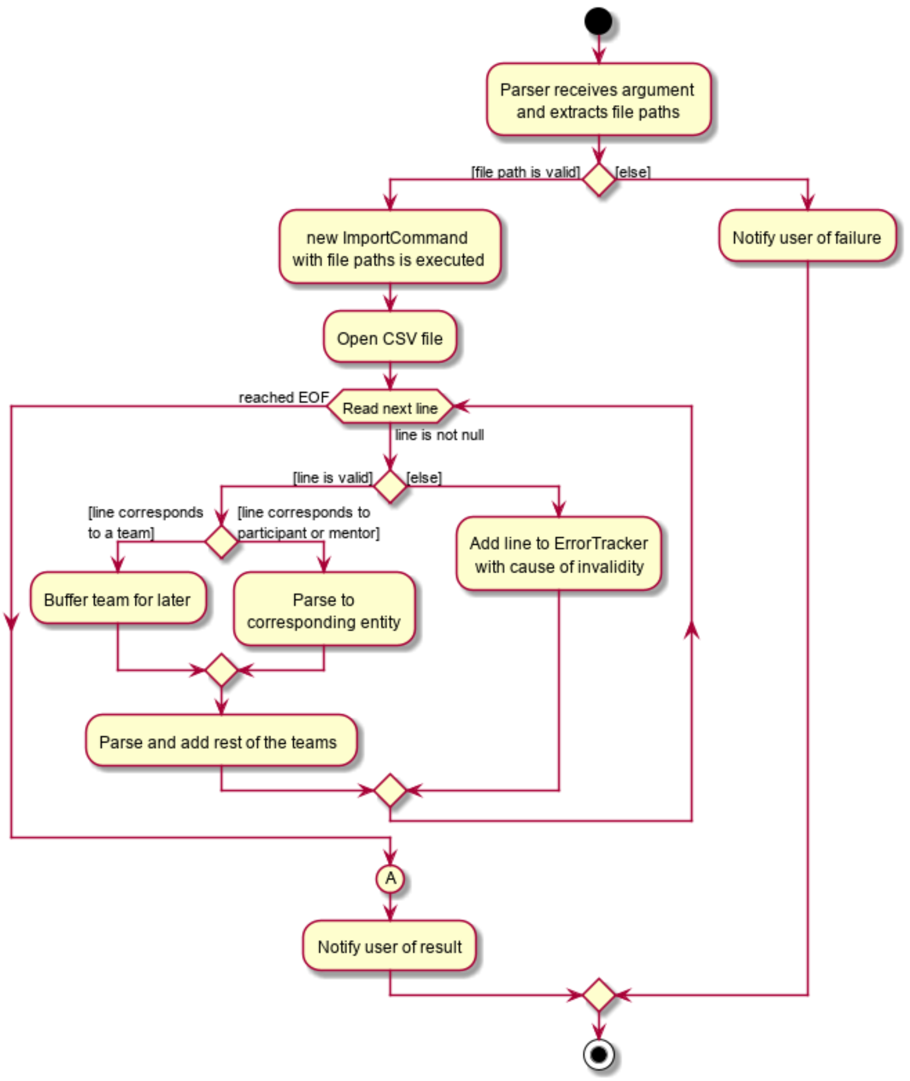

Parser receives argument and extracts file paths
[file path is valid]
[else]
new ImportCommand with file paths is executed
Notify user of failure
Open CSV file
reached EOF
Read next line
line is not null
[line is valid]
[else]
[line corresponds to a team]
[line corresponds to participant or mentor]
Add line to ErrorTracker with cause of invalidity
Buffer team for later
Parse to corresponding entity
Parse and add rest of the teams
A
Notify user of result

Figure 3. Import Command Activity Diagram

In the above diagram, you can see that teams are buffered for later use, which the reason is explained below. Also, **node A**, located right before the end, will be discussed in this section. Now, the sections below will give a detailed explanation of different portions of this feature.

Implementation: `ImportCommand`

Once a valid user input is parsed and passed into the `ImportCommand`, the command will open the file and read its content line by line. Each line is then parsed into the corresponding entity by the `CsvUtil` class. This will be explained further below. The following sequence diagram shows the steps involved in mass importing data into Alfred.

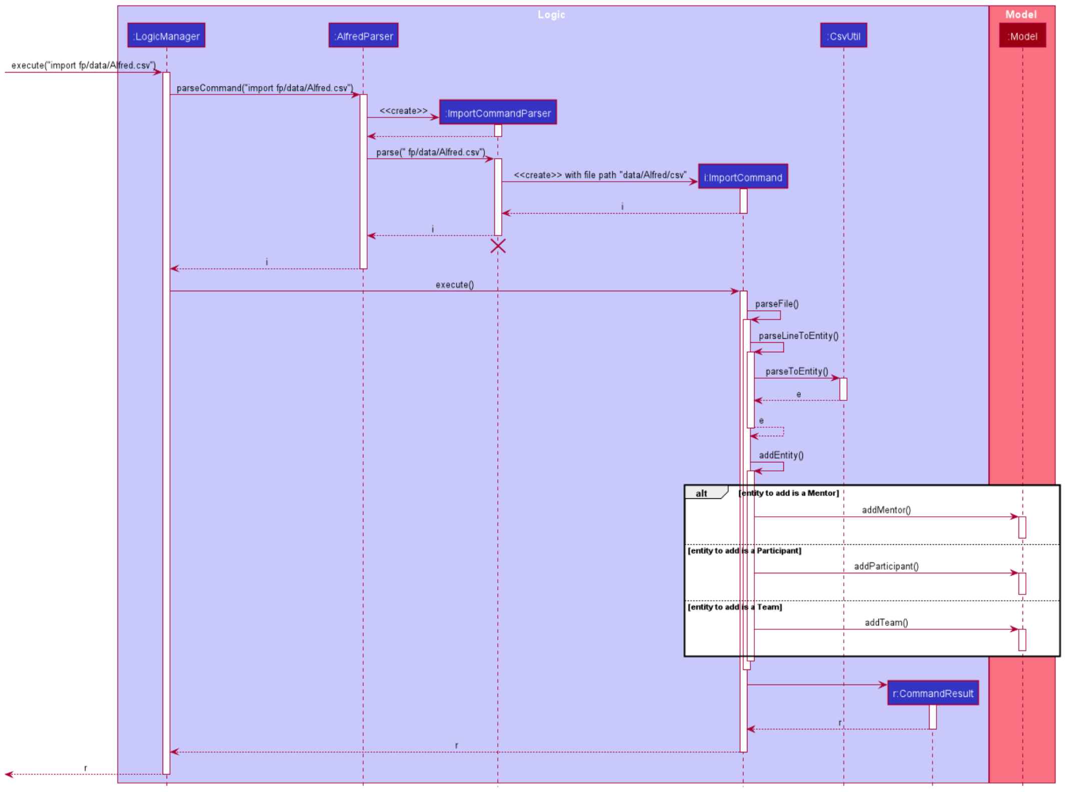

Logic
Model
:LogicManager
:AlfredParser
:CsvUtil
:Model
execute("import fp/data/Alfred.csv")
parseCommand("import fp/data/Alfred.csv")
<<create>>
:ImportCommandParser
parse(" fp/data/Alfred.csv")
<<create>> with file path "data/Alfred/csv"
i:ImportCommand
i
i
i
execute()
parseFile()
parseLineToEntity()
parseToEntity()
e
e
addEntity()
alt
[entity to add is a Mentor]
addMentor()
[entity to add is a Participant]
addParticipant()
[entity to add is a Team]
addTeam()
r:CommandResult
r
r
r

Figure 4. Import Command Sequence Diagram

`FileUtil` shown in the class diagram was omitted from the above sequence diagram for simplicity as it adds little to the overall flow of execution.

As the above figure shows, the file path from the user input is extracted and passed as a field for `ImportCommand`. Then, Alfred proceeds to convert file content into relevant entities.

When `ImportCommand` parses and adds entities to `Model`, it is crucial that teams are the last entities to be added. In the above sequence diagram, this process of buffering teams was also omitted for simplicity. Basically, in the `parseLineToEntity()` method, if a line in CSV file corresponds to a team, the line will be buffered to be parsed after all the other lines have been parsed. The reason for this is because teams may have dependencies on other participants and mentors. It is required that all of the participants and mentors associated with a team, say Team A, exist inside the `Model` before Team A can be added.

So as `ImportCommand` accesses the CSV file line by line, the line representing a team will be stored in a `Queue<String>` for later use. When the end of file is reached and all other participants and mentors are parsed and added to `Model`, the `ImportCommand` will poll from the `Queue`, parse into a relevant team, and add it to the `Model` until the `Queue` is empty.

Design Considerations

When designing this feature, different aspects - list below - were considered.

Aspect 1: Storing of File Path

- **Alternative 1:** Store as a `String`
  - Cons: May have cross-platform issues.

- **Alternative 2 (Current Choice):** Store as a `Path`
    - Pros: Fixes cross-platform issues (by handling all possible separator characters used by different OS's)

Alternative 2 was chosen because of the additional benefits `Path` class provides. Additional overhead of `Path` class (`File` class could also have been used) proved more effective than storing as `Strings`.

Aspect 2: Representation of an Invalid Line in CSV File

- **Alternative 1:** Represent as a `String` and print to user
    - Pros: Gets the message across. The user will know the content of the line that is causing the problem.
    - Pros: Simple to manage.
    - Cons: The user will have to locate where the line is in the CSV file.
    - Cons: The user will not know why the line causes a problem.
- **Alternative 2 (Current Choice):** Create an `Error` wrapper class
    - Pros: Able to store line number, content of line, and cause of error in one object.
    - Pros: Makes sorting of multiple `Error` objects easier through a `compareTo()` method. This proves useful when a buffered line (representing a team) contains error.
    - Pros: Able to display multiple information to user in a neat fashion.

We chose Alternative 2 because `Error` class will be able to provide a more detailed explanation more simply than using a `String`. By displaying to the user the line number and the reason why the line was not able to be imported into Alfred would save user tons of time trying to locate where the line is in the CSV file and why it caused a problem. However, we figured that knowing the line number would not help much in locating the line in the CSV file if the file is huge, hence the next aspect.

Aspect 3: Display of Errors

- **Alternative 1:** Display to user through `CommandResult` box of the GUI
    - Pros: Gets its job done.
    - Cons: May overcrowd the `CommandResult` box for a big CSV file with lots of errors.
- **Alternative 2:** Creates a new CSV file containing all the errors.
    - Pros: Provides a 'clean slate' for the user to correct their errors.
    - Pros: The user does not have to locate the lines in their original CSV file.
    - Cons: The user will not know the reason why certain line caused an error.
    - Cons: May be a bit overkill, especially if only one or two lines were invalid.

This aspect has no **(Current Choice)** attached to any alternative because Alfred utilizes both. As mentioned before, Alternative 2 will be carried out if the user specifies an error file path. Then, whether or not the user has provided the error file path, Alfred will still include an error message in the `CommandResult` box if there are any.

Aspect 4: Assigning of Participants and/or Mentor to Teams through `ImportCommand`

- **Alternative 1:** Do not allow assigning to Teams through `ImportCommand`
  - Pros: Simple to implement.
  - Pros: Placement of lines in CSV file will not cause problems while adding to Alfred.
  - Cons: Defeats the purpose of "bulk registration" if the user has to go through a 2-step process just to add one team.
- **Alternative 2 (Current Choice):** Allow assigning
  - Pros: The user does not have to go through an n-step process to add multiple teams.
  - Cons: Relatively difficult to implement. Have to take care of dependency issues between participants/mentors and teams.

Initially, our plan was to disallow users from assigning other entities to Teams through `ImportCommand`. However, we soon realized that a huge purpose of Alfred and this feature was to facilitate managing of relationships between Teams and other entities. So, we decided to allow assignment by buffering parsing of Teams to a later stage - after all the other participants and mentors have been parsed and added into Alfred.

Now that the structure and flow is explained, I will show the use cases of the feature in the following section.

Use case: Import external data through a CSV file

MSS

1. User writes a CSV file with Entity data.
2. User requests to import the CSV file located at user specified path into the HackathonManager.
3. HackathonManager finds and retrieves the CSV file.
4. HackathonManager adds each Entity in the CSV field.
5. HackathonManager displays original updated list of Entities.

   Use case ends.

Extensions

2a. User does not specify the path to the CSV file.

2a1. HackathonManager asks the user to specify the file path.

2a2. User specifies the file path.

Use case resumes from step 3.

2b. User also requests that the HackathonManager create an error file.

Use case resumes from step 3.

3a. HackathonManager cannot find the file or the path contains illegal characters (note that illegal path characters may vary from OS to OS).

3a1. HackathonManager informs user of failure of execution.

Use case ends.

4a. User specified CSV file contains invalid formatting

4a1. HackathonManager imports the valid lines only.

4a2. HackathonManager informs the user which lines were invalid and why.

Use case resumes from step 5 if user did not request for an error file.

4a3. HackathonManager creates a CSV file with the invalid lines at user specified path.

Use case resumes from step 5.

Further Action

If you want to read more about our project and **Alfred**, here are the relevant links to our websites:

- **Alfred** Website: click
- User Guide: click
- Developer Guide: click