QuickHire Developer Guide

Acknowledgements

{ list here sources of all reused/adapted ideas, code, documentation, and third-party libraries -- include links to the original source as well }

Setting up, getting started

Refer to the guide *Setting up and getting started*.

Design

Architecture

The ***Architecture Diagram*** given above explains the high-level design of the App.

Given below is a quick overview of main components and how they interact with each other.

Main components of the architecture

`Main` (consisting of classes `Main` and `MainApp`) is in charge of the app launch and shut down.

- At app launch, it initializes the other components in the correct sequence, and connects them up with each other.
- At shut down, it shuts down the other components and invokes cleanup methods where necessary.

The bulk of the app's work is done by the following four components:

- `UI` : The UI of the App.
- `Logic` : The command executor.
- `Model` : Holds the data of the App in memory.
- `Storage` : Reads data from, and writes data to, the hard disk.

`Commons` represents a collection of classes used by multiple other components.

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1` .

Each of the four main components (also shown in the diagram above),

- defines its *API* in an `interface` with the same name as the Component.
- implements its functionality using a concrete `{Component Name}Manager` class (which follows the corresponding API `interface` mentioned in the previous point.

For example, the `Logic` component defines its API in the `Logic.java` interface and implements its functionality using the `LogicManager.java` class which follows the `Logic` interface. Other components interact with a given component through its interface rather than the concrete class (reason: to prevent outside component's being coupled to the implementation of a component), as illustrated in the (partial) class diagram below.

The sections below give more details of each component.

UI component

The **API** of this component is specified in `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g. `CommandBox` , `ResultDisplay` , `PersonListPanel` , `StatusBarFooter` etc. All these, including the `MainWindow` , inherit from the abstract `UiPart` class which captures the commonalities between classes that represent parts of the visible GUI.

The `UI` component uses the JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

- executes user commands using the `Logic` component.
- listens for changes to `Model` data so that the UI can be updated with the modified data.
- keeps a reference to the `Logic` component, because the `UI` relies on the `Logic` to execute commands.
- depends on some classes in the `Model` component, as it displays `Person` object residing in the `Model` .

Logic component

API : `Logic.java`

Here's a (partial) class diagram of the `Logic` component:

The sequence diagram below illustrates the interactions within the `Logic` component, taking `execute("delete 1")` API call as an example.

Note: The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline continues till the end of diagram.

How the `Logic` component works:

1. When `Logic` is called upon to execute a command, it is passed to an `AddressBookParser` object which in turn creates a parser that matches the command (e.g., `DeleteCommandParser`) and uses it to parse the command.
2. This results in a `Command` object (more precisely, an object of one of its subclasses e.g., `DeleteCommand`) which is executed by the `LogicManager` .
3. The command can communicate with the `Model` when it is executed (e.g. to delete a person).
   Note that although this is shown as a single step in the diagram above (for simplicity), in the code it can take several interactions (between the command object and the `Model`) to achieve.
4. The result of the command execution is encapsulated as a `CommandResult` object which is returned back from `Logic` .

Here are the other classes in `Logic` (omitted from the class diagram above) that are used for parsing a user command:

How the parsing works:

- When called upon to parse a user command, the `AddressBookParser` class creates an `XYZCommandParser` (`XYZ` is a placeholder for the specific command name e.g., `AddCommandParser`) which uses the other classes shown above to parse the user command and create a `XYZCommand` object (e.g., `AddCommand`) which the `AddressBookParser` returns back as a `Command` object.
- All `XYZCommandParser` classes (e.g., `AddCommandParser` , `DeleteCommandParser` , ...) inherit from the `Parser` interface so that they can be treated similarly where possible e.g, during testing.

Model component

API : `Model.java`

The `Model` component,

- stores the address book data i.e., all `Person` objects (which are contained in a `UniquePersonList` object).
- stores the currently 'selected' `Person` objects (e.g., results of a search query) as a separate *filtered* list which is exposed to outsiders as an unmodifiable `ObservableList<Person>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- stores a `UserPref` object that represents the user’s preferences. This is exposed to the outside as a `ReadOnlyUserPref` objects.
- does not depend on any of the other three components (as the `Model` represents data entities of the domain, they should make sense on their own without depending on other components)

Note: An alternative (arguably, a more OOP) model is given below. It has a `Tag` list in the `AddressBook` , which `Person` references. This allows `AddressBook` to only require one `Tag` object per unique tag, instead of each `Person` needing their own `Tag` objects.

Storage component

API : `Storage.java`

The `Storage` component,

- can save both address book data and user preference data in JSON format, and read them back into corresponding objects.
- inherits from both `AddressBookStorage` and `UserPrefStorage` , which means it can be treated as either one (if only the functionality of only one is needed).
- depends on some classes in the `Model` component (because the `Storage` component's job is to save/retrieve objects that belong to the `Model`)

Common classes

Classes used by multiple components are in the `seedu.address.commons` package.

Implementation

This section describes some noteworthy details on how certain features are implemented.

Storing data manually

The `Storage` component is used to save data automatically whenever a change to the data occurs. While the same component *can* be used to store data manually, there are some limitations:

- Since `Storage` inherits from `AddressBookStorage` , `ScheduleBoardStorage` and `UserPrefsStorage` , any class implementing `Storage` will have to adhere to the contracts of all three interfaces. While it makes sense for the general scope of the application, it doesn't make sense when the user only wishes to save data to a particular location at a particular point in time (and they do not necessarily wish to permanently save the data to this chosen location).
- A workaround will be to implement the methods of `UserPrefsStorage` , and return dummy values like `Optional#empty()` for `readUserPrefs()` or an empty String for `getUserPrefsFilePath()` But this sounds more like we are forced to adhere to the contract of `UserPrefsStorage`
- The alternative will be to create a new storage component for manual storage: `ManualStorage`
  - This interface will inherit from only `AddressBookStorage` and `ScheduleBoardStorage` .
  - A separate manager class, `ManualStorageManager` , will implement `ManualStorage` and implement methods for reading and writing data pertaining to candidates and interview schedules.
  - While this looks very similar to `Storage` and `StorageManager` , the benefit is that, to store data manually, classes and objects no longer have to worry about dealing with user preferences.

[Proposed] Data archiving

{Explain here how the data archiving feature will be implemented}

Documentation, logging, testing, configuration, dev-ops

- Documentation guide
- Testing guide
- Logging guide
- Configuration guide
- DevOps guide

Appendix: Requirements

Product scope

Target user profile:

- a hiring recruiter for a tech company, frequently managing a high volume of candidate profiles
- needs to search, update, and organize candidate contacts quickly
- prefers desktop apps over other types
- prefers typing over using a mouse
- works alone and only share contact data with superiors
- can type fast
- is reasonably comfortable using CLI apps
- finds searching and filtering candidates' details in spreadsheets time-consuming

Value proposition: to be able to manage candidates' details and provide a more efficient way to organize potential candidates to their company compared to traditional methods. It is optimized for hiring recruiters who prefer a CLI.

User stories

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

Priority	As a …	I want to …	So that I can…
`* * *`	user	add a new applicant's contact details	start adding new applicant's details into the application quickly
`* * *`	user	list all applicants' contact	verify the stored data
`* * *`	user	delete applicant's contact	remove applicants that are no longer applying for a job
`* * *`	user	exit the application	
`* *`	user	have all my applicant's contact saved automatically	use the application without losing any changes made
`* *`	user	find an applicant's contact	locate details of persons without having to go through the entire list
`* *`	new user	view usage instructions	refer to instructions when I forget how to use the application
`* *`	user	edit an applicant's contact	rectify any discrepancies in the applicant's contact details
`* *`	new user	import my list of applicant's contact	seamlessly migrate data from using one device to this another

Priority	As a …	I want to …	So that I can…
* *	user	add remarks to an applicant's contact details	note down interesting details about a candidate
* *	user	backup the data of past applicants	recover the data in case of any issues
* *	user	view statistics of applications to a specific role	make informed decisions on recruiting priorities
* *	user	add an interview schedule for a candidate	keep track of upcoming interviews and stay organized
* *	user	delete an interview schedule for a candidate	remove outdated or cancelled interviews
* *	user	edit an interview schedule for a candidate	update interview details when changes occur
* *	user	clear all interview schedules	reset the schedule for re-planning or when starting a new recruitment cycle
* *	user	manually save data pertaining to applicants and their interview schedules	backup the data for archival and recovery purposes
*	new user	play around with sample data	gain more familiarity with using the application
*	user	change the theme of the UI	use whichever I prefer based on my vison and environment
* *	user	filter through job titles easily	shortlist candidates to fill the vacant job position

Use Cases

Use case: UC01 - Listing applicants

MSS

1. User requests the list of applicants
2. QuickHire shows the list of applicants
   Use case ends.

Extensions

- 2a. The list is empty.
    - 2a1. Notify user about the empty list.

  Use case ends.

Use case: UC02 - Adding an applicant

MSS

1. User requests to add an applicant
2. QuickHire adds a new applicant

   Use case ends.

Extensions

- 2a. Duplicate applicant.
    - 2a1. QuickHire shows an error message.

  Use case ends.

Use case: UC03 - Delete an applicant

MSS

1. User lists applicants (UC01)
2. User requests to delete a specific applicant in the list
3. QuickHire deletes the person

Use case ends.

Extensions

- 2a. The given index is out of range.
    - 2a1. QuickHire shows and OutOfRange error.

      Use case resumes at step 2.
- 2b. The given parameters are invalid.
    - 2b1. QuickHire shows an error message.

Use Case: UC04 - Exiting the Application

MSS

1. User requests the exit application
2. QuickHire exits the user

   Use case ends.

Use Case: UC05 - Edit an applicant

MSS

1. User lists applicants (UC01)
2. User requests to edit details of a specific applicant in the list
3. QuickHire edits the specified details

Use case ends.

Extension

- 2a. The list is empty.
  Use case ends.
- 2b. The given index is invalid.
  - 2b1. QuickHire shows an error message.
    Use case resumes at step 2.
- 2c. The given parameters are invalid.
  - 2c1. QuickHire shows an error message.

Use Case: UC06 - Adding remarks to an applicant

MSS

1. User requests to add remarks to an applicant
2. QuickHire adds the given remark to the applicant's details
   Use case ends.

Use Case: UC07 - Finding applicants

MSS

1. User request to find applicants using some keywords
2. QuickHire shows the list of applicants matching the provided keywords
   Use case ends.

Extensions

- 2a. The list is empty (i.e., keywords did not match any applicants).
  - 2a1. Notify user about the empty list.
  Use case ends.

Use Case: UC08 - Viewing statistics of applications to a specific job

MSS

1. User requests to view statistics of applications to a specific job
2. QuickHire shows the list of jobs and their corresponding number of applications

Extensions

- 2a. The stats list is empty (i.e., no one applied for a job) *2a1. Notify user about empty list

Use Case: UC09 - Saving details of applicants into a file

MSS

1. User requests to save data of applicants and interview schedules into two separate files
2. QuickHire saves the displayed list of applicants and interview schedules into the specified files
   Use case ends.

Extensions

- 2a. User requests to save all applicants into the file
  - 2a1. QuickHire saves *all* applicants into the file

  Use case ends.
- 2b. File(s) specified by user already exists in the system
  - 2b1. QuickHire displays error message saying that the file(s) already exists

  Use case ends.
- 2c. File(s) specified by user already exists in the system, and user requests to *overwrite* any existing file
  - 2c1. QuickHire saves details of applicants to the file(s) (without any errors)

  Use case ends.
- 2d. User does not provide either file
  - 2d1. QuickHire display error message indicating the command format

  Use case ends.

Use Case: UC10 - Saving details of filtered data (of applicants)

MSS

1. User finds applicants (UC07)
2. User requests to save applicants into a file
3. QuickHire saves the displayed list of applicants into a file
   Use case ends.

Extensions

- 2a. File exists and user did not request to overwrite file
  - 2a1. Notify user that file already exists

  Use case ends.
- 2b. User requests to save all data

    - 2b1. Save all data (instead of just filtered ones) into file

    Use case ends.

Use case: UC11 - Listing interview schedules

Similar to use case 1 except for using to list schedules

Use case: UC12 - Adding an interview schedule

MSS

1. User requests to add an interview schedule
2. QuickHire adds a new schedule

   Use case ends.

Extensions

- 2a. Timing clashed with existing interview schedules.
    - 2a1. QuickHire shows an error message.

  Use case ends.
- 2b. The given parameters are invalid.
    - 2b1. QuickHire shows an error message.

  Use case ends.

Use case: UC13 - Delete an interview schedule

Similar to use case 03 except for using to delete an interview schedule.

Use Case: UC14 - Edit an interview schedule

MSS

1. User lists interview schedules (UC11)
2. User requests to edit details of a specific schedule in the list
3. QuickHire edits the specified details

Use case ends.

Extension

- 2a. The list is empty.

  Use case ends.
- 2b. The given index is invalid.
    - 2b1. QuickHire shows an error message.

      Use case resumes at step 2.

- 2c. The given parameters are invalid.
    - 2c1. QuickHire shows an error message.

Use case: UC15 - Clear all interview schedules

MSS

1. User requests to clear the list of interview schedules
2. QuickHire shows the empty list of interview schedules
   Use case ends.

Use Case: UC16 - Changing theme of the UI

MSS

1. User requests to change theme to a specific theme
2. QuickHire changed to requested theme
   Use case ends.

Extensions

- 1a. User specified incorrect theme
    - 1b1. Notify user of incorrect value

  Use case ends.

Non-Functional Requirements

1. Should work on any *mainstream OS* as long as it has Java `17` or above installed.
2. Should be able to hold up to 1000 candidates for hire without a noticeable sluggishness in performance for typical usage.
3. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse.
4. All user commands can work without internet without any noticable difference.
5. All user commands have a response time of under 5 seconds, (assuming <= 1000 candidates).
6. The system should be able to handle a growing number of candidates for hire without a noticable dip in performance.
7. The system should be usable by anyone, including novice users.
8. The system should store all data locally, hence no requirement for a server.

Glossary

- **Mainstream OS**: Windows, Linux, Unix, MacOS
- **Locally**: A location on the users Hardrive/SSD.

- **Server**: An external offline location accessed through the internet for the storage of large data.
- **Novice users**: Users with limited to no prior command-line operation knowledge.

Appendix: Instructions for manual testing

Given below are instructions to test the app manually.

Note: These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

Launch and shutdown

1. Initial launch
   1. Download the jar file and copy into an empty folder
   2. Double-click the jar file Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.
2. Saving window preferences
   1. Resize the window to an optimum size. Move the window to a different location. Close the window.
   2. Re-launch the app by double-clicking the jar file.
      Expected: The most recent window size and location is retained.
3. *{ more test cases … }*

Deleting a person

1. Deleting a person while all persons are being shown
   1. Prerequisites: List all persons using the `list` command. Multiple persons in the list.
   2. Test case: `delete 1`
      Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.
   3. Test case: `delete 0`
      Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.
   4. Other incorrect delete commands to try: `delete`, `delete x`, `...` (where x is larger than the list size)
      Expected: Similar to previous.
2. *{ more test cases … }*

Saving data

1. Dealing with missing/corrupted data files
   1. *{explain how to simulate a missing/corrupted file, and the expected behavior}*
2. *{ more test cases … }*