Joyce Yeo Shuhui - Project Portfolio

PROJECT: Concierge

Overview

Concierge™ is a desktop hotel management application for receptionists to handle potential bookings and current guests. The user interacts with it using a CLI, and it has a GUI created with JavaFX. It is written in Java, and has about 10 kLoC.

Summary of contributions

Access my contributed code here.

Major enhancement: added **the ability to login/logout** of the system.

- What it does: Allows users to login to Concierge and access restricted commands which mutate the data.
- Significance: With this feature, hotel managers can implement some level of access control. Some features more commonly used (`find`, `list`) can still be accessed without signing in. By combining this with the ability to export the command history, the auditing process for rogue commands is expedited.
- Highlights:
  - The login/logout feature is dynamic - the users are not prompted to sign in upon starting Concierge, and users can logout and log back in within the same session. This means that there is a very close integration with the existing commands to verify the sign-in status and whether the commands require signing-in.
  - This feature is complete - I worked with the different architectural components of Concierge, from command parsing (`Logic`) to login verification (`Model`) and even password storage (`Storage`).
  - The feature attempts to achieve some level of security with SHA-256 hashing.
- Credits: The password hash algorithm was taken from Baeldung.
- These features were mainly achieved in #168 and #226.

Minor enhancements/ code contributed:

- Renamed the existing classes and methods from `Person` to `Guest`

- Removed `Address` as a field in `Person`, removed the edit and delete commands
- Modified the `add` command to take in room and date details. Room and Booking package by others.
- Modified the `clear` command to maintain empty hotel rooms
- Added function for GUI verification of rooms
- Added most of the Appendices in the Developer Guide

Project Management:

- Managed the team's issue tracker
- Encouraged team to use clean PRs, TODO and Codacy
- Set up RepoSense for the team
- Community
  - PRs reviewed (with non-trivial review comments): #157, #71, #156, #111, #100, #222, #219
  - Reported bugs and suggestions for other teams in the class
    - Discovered impersistence in data, discovered hidden bug by rearranging command, made suggestion on bounds checking

Contributions to the User Guide

> *My contributions to the User Guide below showcase my ability to write documentation targeting end-users.*

Adding a booking: `add` 

Adds a booking associated with a guest, room and booking period.
Format: `add n/NAME p/PHONE_NUMBER e/EMAIL [t/TAG] r/ROOM_NUMBER from/START_DATE to/END_DATE`

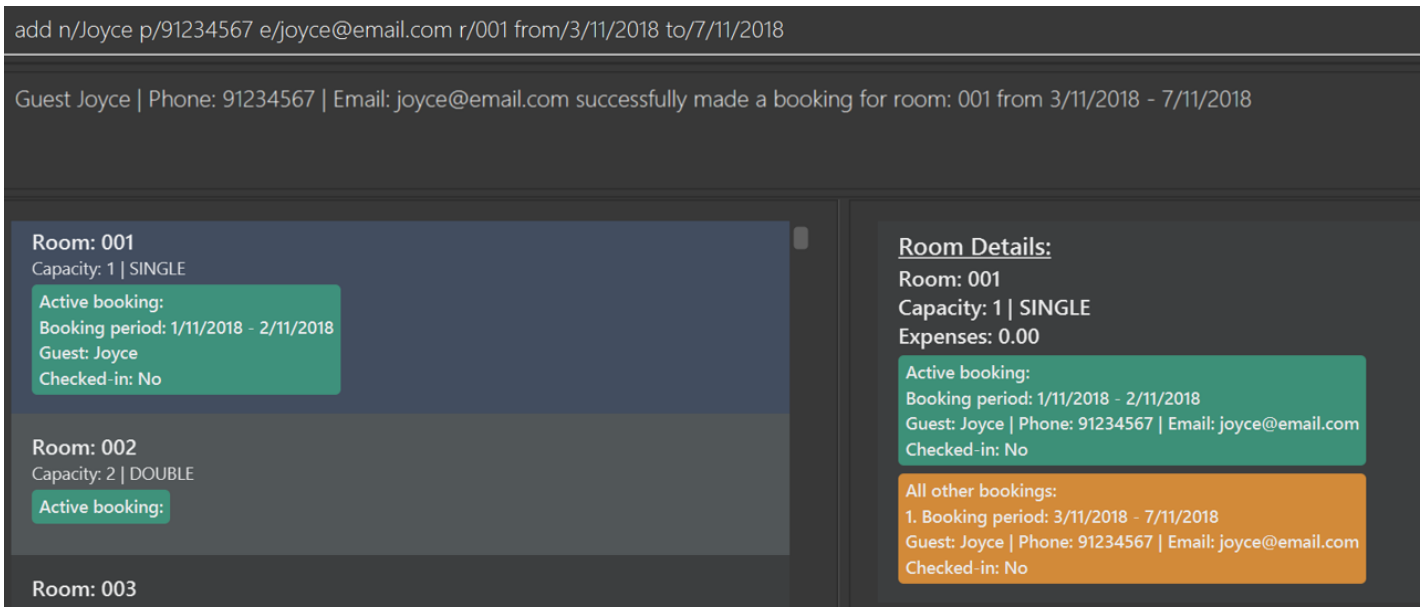
> A valid booking cannot clash with an existing booking. It must also have a start date from today onwards (i.e. not outdated).

- A guest can have any number of tags (including 0)
- A guest can make an unlimited number of bookings with the hotel.
- When adding a booking, the guest will **not** be added to the archived guest list or checked-in guest list. Their personal information will be stored under their booking in the room.

Example: `add n/John Smith p/98765432 e/johnsmith@gmail.com t/VIP r/085 from/17/12/18 to/19/12/18`

Add a guest "John Smith" to room 085 for the period of stay from 17/12/18 to 19/12/18.

Adding an inactive booking: you can view the booking by selecting the relevant room, under "All other bookings". Only active bookings (i.e. start date is today) can be seen on the left room pane.



Login : `login`

Logs in to the Concierge™ application.
Format: `login user/USERNAME pw/PASSWORD`

Note: The username and password are both case-sensitive.

> The default account can be accessed with `login user/admin pw/passw0rd`

A login allows the user to access the commands which can affect the bookings.
Commands which require login: `add`, `checkin`, `checkout`, `reassign`, `service` and `clear`.

Example: `login user/admin pw/passw0rd`

`clear`
`"This command requires you to sign in."`
Attempting to clear Concierge™ without a login is not allowed.

`login user/admin pw/passw0rd`
`"Successfully signed in as: admin"`
Login with a valid account, such as the default one provided.

After signing in, the `clear` command can now be executed.

Adding a new account

```
Currently, Concierge(TM) does not have a feature for users to add an account via the app.
Nevertheless, for the adventurous users who want to do so, this sub-section will be useful.
```

Step 1: Concierge™ uses SHA-256 password hashing. This hash for your **password** can be generated using this tool.

> Concierge™ is designed to work for alphanumeric usernames and passwords in mind. Do not enter symbols (!, @, %…). Do not begin or end your passwords with whitespaces.

Step 2: Add the entry to the `passwords.json` file. This should be in the same location as `concierge.jar`.
Note that entries are separated with a comma.
Format: `"username" : "hashedPassword"`

In the image below, a new account with username "newUser" and password "mySecurePassw0rd" has been added.

Step 3: Close and reload the Concierge™ application, and your new account will be recognised.

> The parsing of the `passwords.json` file is delicate. Currently, if you enter a valid json file format but an incorrect password reference list format, you will end up with no default account. To resolve this, delete the `passwords.json` file and re-run Concierge™.

Logout : `logout`

Logs out of the Concierge™ application.
Format: `logout`

- The special classes of commands (as documented in login) can no longer be executed.

- The `logout` command will erase the command history, so users cannot undo/ redo commands executed before the logout.
    - Even if you login again, you cannot undo your previous actions.
    - This achieves the same effect as closing and re-opening Concierge™ after a logout.

Example: `logout`

Contributions to the Developer Guide

> *My contributions to the Developer Guide below showcase my ability to write technical documentation and the technical depth of my contributions to the project.*

Adding a Booking

The `add` command is used by the receptionist to add the guest to the hotel, and assign him a room.

Current Implementation

We currently accept a `Guest`, `RoomNumber` and `BookingPeriod` as parameters for the `AddCommand` constructor. An example of its usage: `add n/Madith p/83141592 e/madith@themyth.com r/041 from/29/11/2018 to/ 03/12/2018`

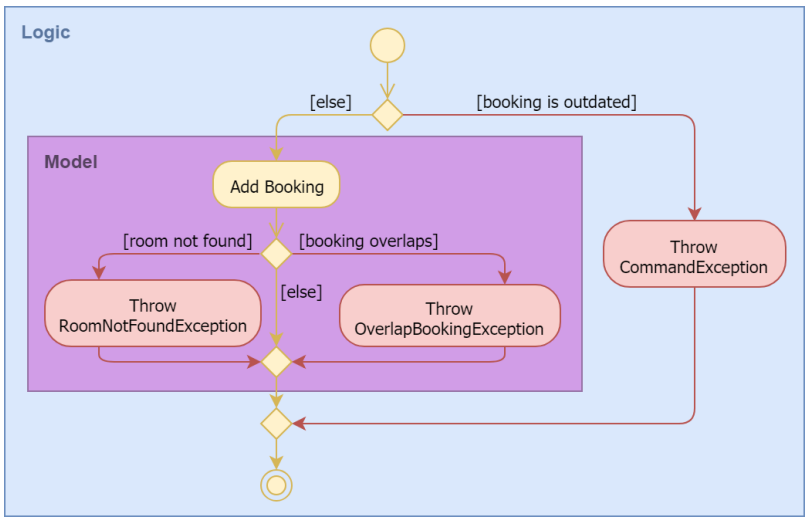
The parsing of the `AddCommand` is very similar to what was already implemented in AddressBook4. More parameters were added, namely the `RoomNumber` and `BookingPeriod`. These are parsed to create the respective objects - `Guest`, `RoomNumber` and `BookingPeriod`.

- In v2.0, users can enter a start date and duration to specify their booking period.

As in AddressBook4, the `Logic` component parses the `AddCommand`, and the `Model` handles its execution.

- In the `Model`, the `Guest` is *no longer* added to Concierge. It was previously the case in AddressBook4.
- A new `Booking` object is created with the `Guest` and `BookingPeriod` as its parameters.
- This `Booking` is then added to the `Room` with the `RoomNumber` specified. Every `Room` maintains a `SortedSet<Booking>` which is encapsulated in the `Bookings` (plural) class.

An Activity Diagram for the execution of `AddCommand#execute` is shown below.



The `AddCommandParser` already checks that `ROOM_NUMBER` is a valid string from `001` to `100`, and the initialisation of Concierge checks that there are 100 rooms. The `RoomNotFoundException` is not expected to occur for any user input, but is left there as a defensive measure.

Design Considerations

Aspect: Check for outdated bookings

Outdated bookings are those which have a start date before today. Concierge disallows users to `add` outdated bookings.

- **Alternative 1 (current choice):** Do the check in `AddCommand#execute`
  - Pros: Very easy to implement. A parameter check in the `execute` method will suffice. Will only affect the `AddCommandSystemTest`.
  - Cons: The actual `Model#addBooking` does not do any check on the `BookingPeriod` being outdated, opening the possibly of outdated `Booking` s being added from via other commands.
- **Alternative 2:** Do the check in `Room#addBooking`
  - Pros: Centralises exceptions thrown related to bookings in the `Booking` class. Increases the cohesiveness of this class.
  - Cons: All the existing tests and sample data calling the `addBooking` method with outdated bookings have to be changed. It also becomes difficult to do unit tests on checking in bookings which are outdated but not expired, since these bookings can no longer be added to the model.

Aspect: Reduce coupling between `Room` and `Guest`

Semantically, we can observe a strong coupling and dependency between `Room` and `Guest`. A `Room` contains a `Guest`, and a `Guest` also has a `Room`. Maintaining this coupling allows for very quick lookup both ways, either given a `Guest` (which is common at the reception desk) or given a `Room` (which is common for

housekeeping).

- **Alternative 1 (current choice):** Add `Guest` as a field in `Room`
  - Pros: An efficient way for managing bookings. Receptionist can quickly determine if the `Room` is free to book. Lookup time for `Guest` not expected to increase greatly, since `Room` s are not expected to have a large number of advanced bookings made.
  - Cons: Difficult to find the `Room` given the `Guest`. When a `Guest` has made an advanced booking and wishes to cancel it, we have to search through all the `Room` s. Nevertheless, we expect most guests to be aware of their rooms.
- **Alternative 2:** Add `Room` as a field in `Guest`
  - Pros: Very customer-centric design. Centralises all the information about the `Guest`, including `Booking` s made and `Expense` s incurred.
  - Cons: Making a new `Booking` with a `Guest` is highly inefficient. `Booking` information is now scattered across individual `Guest` s.
    - [v1.4] On top of the list of rooms, we maintain a separate list of checked-in guests. This list does not retain any booking information, as it is meant to for a quick lookup of the guests' particulars.

Login and Logout

The `login` feature allows hotel managers to control which receptionists have full access to Concierge. When paired with the CommandArchive feature, they can also create a blame history to trace rogue commands.

Current Implementation

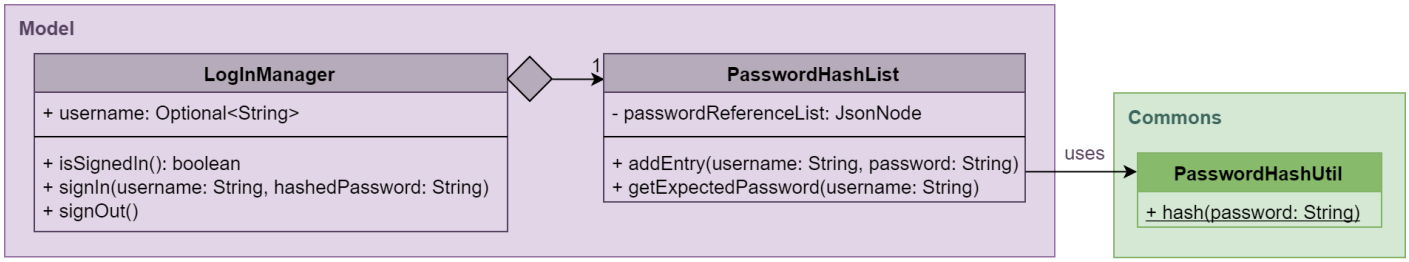
Currently, `login` is implemented as a dynamic feature, so users are not prompted to sign in upon starting Concierge. Instead, they only have to sign in when executing commands which would mutate the data, such as `add`, `checkin`, `checkout`, `reassign`, `service` and `clear`.

Logic

Given the nature of the `login` command being dynamic (can be entered at any point in time, between any commands), it is then natural to implement it like a normal command, extending the abstract `Command` class. The `logout` command is also implemented in this way.

Model

The model handles the signing-in, using its attribute `LogInManager`. The Class Diagram of the login module is shown below.



`LogInManager` uses an optional `username` to keep track of whether the user is currently signed in. The `passwordReferenceList` provides an immutable key-value lookup for usernames and passwords.

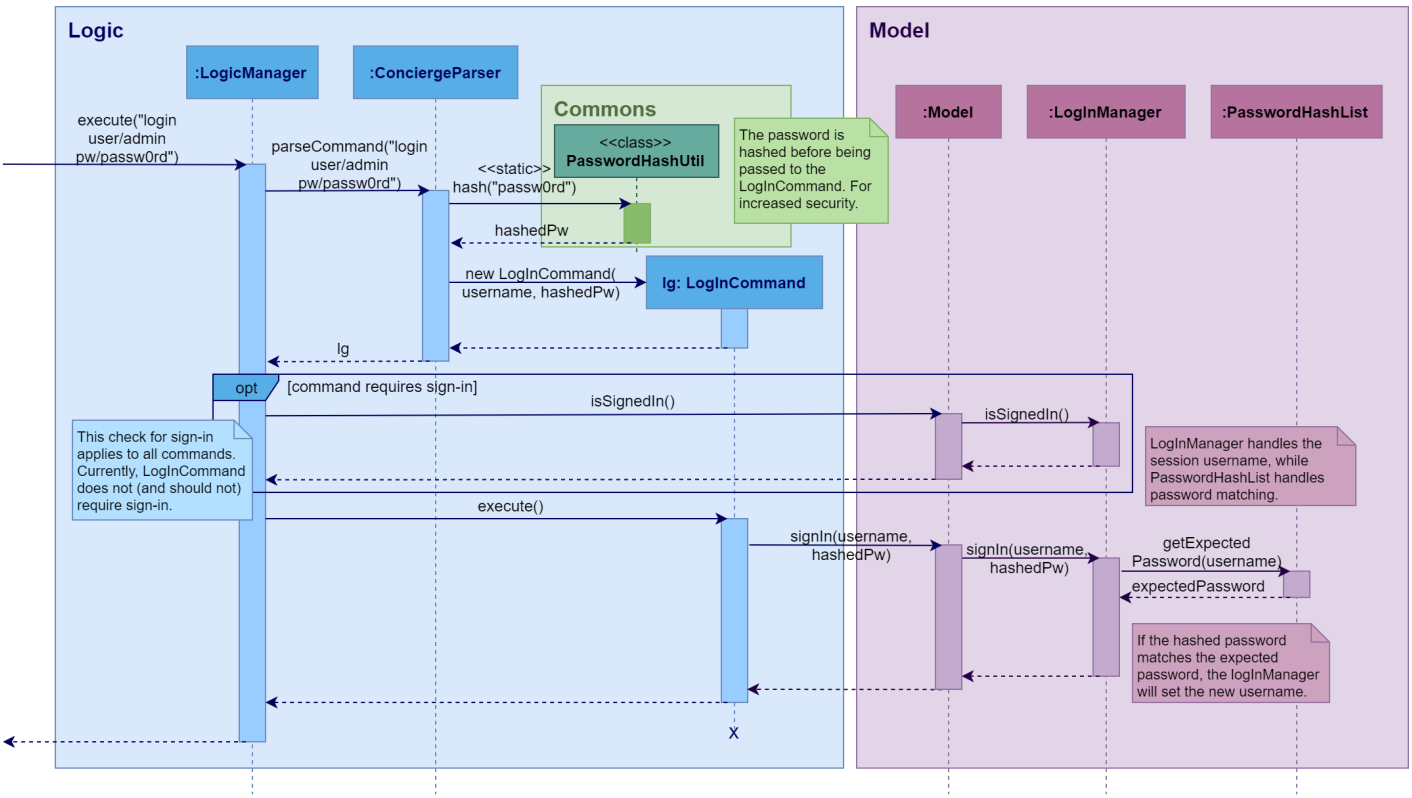
`LogInManager` implements the following operations.

- `LogInManager#isSignedIn()` - checks if the user is currently signed in.
- `LogInManager#signIn(String, String)` - attempts to sign in with the given username and hashed password. This is handled by the `PasswordHashList`. A case-insensitive comparison is used on the hash.
- `LogInManager#signOut()` - signs out of Concierge.

A new method `resetUndoRedoHistory` was added to the `VersionedConcierge` (used for the Undo/ Redo feature). This is used to clear the command history upon a `logout` command, so users cannot undo important commands or redo accidental bad commands after signing out.

Login

Shown below is the Sequence Diagram for executing a valid `login` command. The diagram also illustrates how the `LogicManager` checks for the sign-in requirement of commands.



Storage

The `passwords.json` file is read when Concierge is first opened (i.e. in `MainApp#init`), and is never written to again. The storage function is managed by the `JsonPasswordsStorage` class. Intuitively, passwords are stored as key-value pairs for quick look-up.

A SHA-256 hash was used in building this feature. In future, this hashing algorithm can be changed to a HMAC hash, which adds a username salt. Then, different users will not know if they have chosen the same passwords.

Check for sign-in requirement of commands

`LogicManager` does the checks for whether a command requires a sign in, and whether the model is signed in.

The `Command` class exposes a new `requiresSignIn()` method that returns false by default. To make new command require signing-in, one only has to overwrite this method in that command.

Design Considerations

Aspect: Accessing features of Concierge with/ without login

- **Alternative 1 (current choice):** Login is needed only for some features
  - Pros: Manager can implement some level of access control within Concierge, especially since some of the more commonly used Concierge features (`list`, `find`) are read-only features. This is quicker than mandating a sign-in at the start and creating different user views based on the account privilege (admin vs normal).
  - Cons: Not very intuitive to users. They have to enter commands before being told they need to sign in. The `requiresSignIn()` check takes place after the parsing of the command, so a user can is told they cannot execute the command without a sign-in after their command is parsed correctly.
- **Alternative 2:** Login is needed for all features
  - Pros: Very easy to check login validity. This only occurs when Concierge is first loaded. Subsequent commands can be executed without additional checks on the sign-in requirement.

Aspect: Check for sign-in requirement of commands

Given that sign-in is only required for some commands, the priority in designing this aspect is the ability to easily mandate/ disable compulsory the login requirement for current and future commands.

- **Alternative 1 (current choice):** Do the check in `LogInManager#execute`
  - Pros: Ensures that commands are checked before any execution. Users will not inadvertently change the model before doing the sign-in checks.
  - Cons: Unable to implement commands that can do some actions without sign-in. For example, a future developer may want to make the `add` command such that when the user is not signed-in, the booking is still added but a tag is added to the `Guest`, reminding the manager to verify the booking.

    - Violates the Single Responsibility Principle. The job of `LogicManager` is to parse and execute commands.

- **Alternative 2:** Do the check in `Command#execute`
    - Pros: Increases cohesiveness of `Command` class. The compulsory sign-in is an attribute of a `Command`, so these checks can be done internally. `Command` can implement a method `checkSignIn(Model)`, and commands which require sign-ins can call this method in their respective `execute` methods.
    - Cons: While increasing cohesion, this implementation makes less semantic sense. The logical misstep comes because one is executing the method, then checking if the method can be executed, then "reversing" the execution.

Aspect: Storage of Passwords

The password file is currently read at `MainApp#init`, and saved once. Unlike the Concierge data, this file is no longer referred to when Concierge is in use.

- **Alternative 1 (current choice):** Store passwords in JSON file
    - Pros: JSON is very easy to work with.
        - Able to utilise existing `JsonUtil` methods used by the `UserPrefs` and `Config` classes.
        - Easily parse data into key-value pairs, which semantically matches our needs.
    - Cons: `JsonUtil` file is not completely suitable for a data type that has potentially an unlimited number of entries, since this utility serialises the data to match the class attributes.
- **Alternative 2:** Store passwords in same XML file as all other Concierge data
    - Pros: Centralises data storage in Concierge. There is only one single source of truth for all data.
    - Cons: The XML file is too complicated for the needs of password storage.
        - Concierge does not need to write the the passwords file when in use. `concierge.xml` is constantly being written to, which is an unnecessary and possibly unsafe feature for the passwords component.
        - Creating a new password entry is difficult since once has to add all the layers of XML tags involved. Nevertheless, users are not expected to be adding new accounts on a regular basis.

End Note

I am grateful to have had the opportunity to work on this project with amazing team members who each contributed with their own strengths. Thank you, @adamwth @teowz46 @JiaqingTan @neilish3re.

Last updated 2018-11-12 23:32:44 SGT