

Project Manual Cover Page (v0.5)

[-] [x]



Completed
Incomplete
What's up

ID	Task Name	Start Time	End Time
1	interview with dreamalliance inc.	Tomorrow (8.00 AM)	
2	cs3103 lab change	Wednesday (8.00 AM)	
3	2010 PS&D still WA		Thursday (11.59 PM)
4	2101 earlier time	Friday (12.00 AM)	Friday (12.00 PM)
5	1101R hmwk4		Nov 16 Monday (11.59 PM)
6	complete credit card form for dad		Nov 20 Friday (11.59 PM)
7	rent for nov		Nov 30 Monday (11.59 PM)
8	noc course/event	Dec 7 Monday (8.00 AM)	Dec 14 Monday (11.59 PM)
9	email NS department for exit permit		Dec 20 Sunday (11.59 PM)
10	270031 06237		
11	Prelude No.4 in Em op 28 - wrong E notes		
12	email arthur about the offers		
13	buy green tea		
14	1101r lec 16		

You pressed tab!
 Celebi: Switching view to incomplete tasks

Supervisor:

Han

Rui

Extra feature: GoodGUI

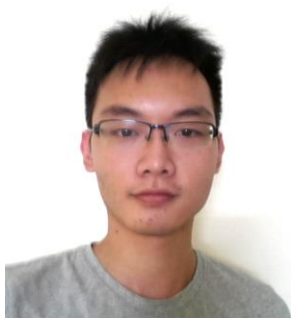

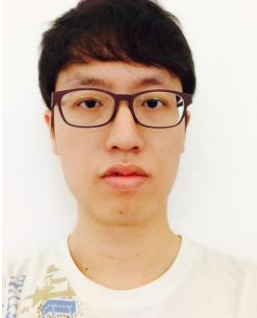

 <p>Ken Lee</p> <p>Team lead</p> <p>Integration</p> <p>Logic component</p>	 <p>Liu Yang</p> <p>Scheduling</p> <p>Storage component</p>	 <p>Leow Yijin</p> <p>Documentation</p> <p>Parser component</p>	 <p>You Jing</p> <p>Visual Design</p> <p>UI component</p>
---	---	---	--

Table of Contents

Project Manual Cover Page (v0.5)	1
Table of Contents	2
1 Credits	5
1.1 Third Party Libraries Used	5
Celebi User's Guide	6
2 About Celebi	6
3 Getting Started	7
3.1 First Time Setup	7
3.2 Launching Celebi	7
3.3 Closing Celebi	7
3.4 Saving changes	7
3.5 The interface	8
4 Using Celebi (Basic)	9
4.1 Adding Tasks	9
4.2 Editing Tasks	11
4.3 Completing Tasks	12
4.4 Switching between Default and Completed Views	13
4.5 Re-opening Tasks	13
4.6 Deleting Tasks	14
4.7 Undo Last Command	15
4.8 Help	15
5 Using Celebi (Intermediate)	16
5.1 Adding Tags	Error! Bookmark not defined.
5.2 Removing Tags	Error! Bookmark not defined.
5.3 Searching and Filtering	16
5.4 Recurring Tasks	Error! Bookmark not defined.
6 Using Celebi (Advanced)	17
6.1 Changing the Storage File	19
6.2 Valid Date Formats	19
Celebi Developer's Guide (v0.1)	21
7 Welcome	21

8 Setup	21
8.1 Source Code Repository	21
8.2 Development Environment	22
9 Architecture	23
10 The Common Package.....	25
10.1 The Task Class	25
10.2 The TasksBag Class.....	26
10.3 The Configuration Class:	26
11 The UI Component	27
11.1 The CelebiViewController Class	28
11.2 CelebiViewController important APIs	29
11.3 The UI Class	29
11.4 UI important APIs.....	29
11.5 FXML files.....	30
11.6 CSS files	30
11.7 TTF files	30
12 The Logic Component.....	31
12.1 The Logic Class	31
12.2 Feedback Class	32
12.3 ActionInvoker class.....	33
12.4 Action classes.....	33
12.5 Logic Testing	33
13 The Parser Component.....	35
13.1 The CommandData Interface	35
13.2 CommandData important APIs	36
13.3 The ParserControllerImpl Class	36
13.4 ParserControllerImpl important APIs	37
13.5 The AliasesImpl Class	37
13.6 Aliases Important APIs.....	38
14 The Storage Component.....	39
14.1 The TaskJson Class	39
14.2 The Storage Class.....	40
14.3 Storage important APIs.....	40
14.4 The Database Class:.....	41

14.5 Database Important APIs	42
Appendix A: User stories.....	44
[High Priority] (likely)	44
[Medium Priority] (likely)	45
[Low Priority] (unlikely)	46
Appendix B: Non Functional Requirements.....	47
Appendix C: Product survey	48

1 Credits

Good software is about collaboration. We give credit to other products and developers who make our work simpler.

1.1 Third Party Libraries Used

Oracle: JavaFX

License: [Oracle Binary Code License Agreement for Java SE Platform Products and JavaFX](#)

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.

We use the JavaFX library to design and implement our graphical user interface. It is used by the UI architectural component.

For more information, visit <http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm>

JSON.simple

License: [Apache 2.0 License](#)

JSON.simple is a simple Java toolkit for JSON. You can use JSON.simple to encode or decode JSON text. It implements the RFC4627 JSON specification.

We use the JSON.simple library to serialise our persistent data across use sessions. It is used by the Storage architectural component.

For more information, visit <https://code.google.com/p/json-simple/>

Celebi User's Guide

2 About Celebi

Celebi is your very own personal assistant. It keeps track of your to-do items, your tasks, your deadlines, and your important events. Spend less time worrying and choosing, and more time doing. Celebi also gives you the power to prioritise and manage your tasks. With Celebi, you can:

- Have your tasks automatically sorted by completion dates and priority.
- Tag your tasks and filter them to easily view similar items together.
- Quickly search for a task even if you forget its name.
- Mark finished tasks as completed, and view them in your task history.
- Set up recurring tasks and events to automate your schedule.
- Access every command using lines of simple and natural text.
- Enter and chain complex commands in a single line of input for optimum efficiency.

All this and more in a lightweight, robust, and independent package. Unlike more popular to-do list software apps, you do not need an active internet connection. Neither do you need to set up or bundle extra software dependencies. Get what you need, and nothing more.

Celebi is naturally easy to learn and use, but its strength lies in its advanced commands and shortcuts. Mastering those will massively accelerate your workflow. Refer to section 5 for the detailed functionality of the advanced command formats.

Multiply your productivity with Celebi today!

3 Getting Started

Orientate yourself.

3.1 First Time Setup

- 1 Check the current version of Java you have installed.
You can do so by visiting <https://www.java.com/en/download/installed.jsp>
- 2 If your installed version is missing or less than Java 8.0,
Download and install the latest version at <https://www.java.com/en/download/>
- 3 Place the Celebi.exe file you downloaded in your desired folder
- 4 (Optional) Pin Celebi.exe to your start bar or create a desktop shortcut for easy access.

3.2 Launching Celebi

You can launch Celebi anytime by clicking its icon.

3.3 Closing Celebi

You can close Celebi at any time by clicking the x button in the top right corner of the window, or by typing this command:

```
>> quit
```

3.4 Saving changes

All changes you make in Celebi are saved immediately.

However, if you force Celebi to terminate from a power outage or the task manager, your last change might not be successfully saved.

3.5 The interface



Figure 1: User Interface

You will interact with Celebi through its graphical user interface (GUI). The GUI is shown in figure 1.

There are only 3 main areas you should focus on. The **task display panel**, **command line input**, and the **command history**.

The **task display panel** is the large rectangular panel right below the Celebi logo. It displays the tasks you are currently viewing. We call the current view the "active view".

As you can see from figure 1, each task is matched with a unique ID number in the active view. You will use that ID number to identify specific tasks for your commands.

The **command line input** is the thin rectangular box at the bottom of the GUI. It is your main way of interacting and giving commands to Celebi. You use it by typing your full command into the box, then hitting enter.

The **command history** is the green box directly above the command line input. It holds a history of the commands you have entered previously since Celebi was launched.

You can resize any panel by dragging its vertical borders.

4 Using Celebi (Basic)

Get started quickly. Learn Celebi's basic functionality.

4.1 Adding Tasks

Whenever a new to-do item comes in, Celebi really wants to help.

Let Celebi track any new tasks by adding them.

Celebi differentiates between 3 basic types of tasks: **Floating tasks, Deadlines, and Events.**

Floating tasks

Floating tasks have no associated dates. They best represent things you want to do without any time pressure. If you want to read a book or learn a musical piece for leisure, this is the type of task to use.

To add a normal floating task to remind yourself to read the Harry Potter book series, type this:

```
>> add read Harry Potter series
```

The new "read Harry Potter series" task will immediately appear in the display panel's active view.

The general format of the command is:

```
>> add <name>
```

<name> is the name of the new task you are creating

Deadlines

Deadlines are tasks with a clear deadline or end date. They represent anything you need to complete by a certain date. If you have an assignment due or a deadline from your boss, this is the type of task to use.

To add an important deadline for your math homework assignment due next Monday 11pm:

```
>> add! Math assignment; due next Monday
```

To add a normal deadline for your final project milestone due 26th December 2016:

```
>> add final project milestone; due 2016 dec 26
```

The general format is:

```
>> add <name>; due <date>
```

Where:

- **<date>** is the due date.

Celebi is smart and can recognise nearly any format of date you provide. However, if Celebi cannot understand your date, you can try following the example format exactly or check *Section 6.2*.

Events

Events are tasks with a start and end date. They represent fixed blocks of time in your calendar taken up by that task. If you have to attend a meeting or go to a seminar, this is the type of task to use.

To add a normal event for a performance review meeting next Wednesday from 4 to 5 pm:

```
>> add Perf review meeting (sigh); next Wednesday from 4 to 5 pm
```

To add an important event to remind you to watch your daughter Laura's ballet recital on 10th December this year, 2 to 4 pm:

```
>> add! watch Laura ballet recital; 10 dec from 2 to 4 pm
```

The general format for events that fit in one day is:

```
>> add(!) <name>; <day> from <start time> to <end time>
```

Where:

- **<day>** is the calendar date of the event
- **<start/end time>** are the event's start and end times.

The general format for potentially multi-day events is:

```
>> add(!) <name>; from <start date> to <end date>
```

Where:

- **<start/end date>** are the event's start and end dates.

For more details on the date formats Celebi recognises, check *Section 6.2*.

4.2 Editing Tasks

Life does not always go according to plan.

Celebi is understanding, and lets you update your existing tasks if circumstances change.

If your math homework assignment (ID 2) deadline was extended to next Wednesday 8am:

```
>> edit 2 due next wed 8am
```

If you change your mind about reading Harry Potter (ID 1), and want to read Lord of the Rings instead:

```
>> edit 1 name LOTR series
```

The general format is:

```
>>  edit <task id> <data field> <new value>
```

Where:

<task id> is the ID number associated with your task in the display panel (see section 3.5)

<data field> is which data field you want to change in the task.

<new value> is the new value for the data field. The format is dependent on the data field.

A quick reference for the allowed <data field>s and their corresponding allowed <new value>s.

Task data field	<data field>	<new value>
Task name	"name"	Any text without semicolons
Start date	"start"	<date> (see <i>Section 4.1</i> or XXX)
End date / deadline	"end"/"due"	<date> (see above)

Figure 2: Edit command <data field> and <new value> pairings

4.3 Completing Tasks

Good job! You have been productive, and completed a task or two.

Let Celebi share in your joy as you gleefully mark your task complete!

Like in section 4.2, use the display panel's task ID number to help Celebi identify the task you want to mark as completed.

To mark your math homework assignment (ID 2) as completed:

```
>>  mark 2
```

The general format is:

```
>>  mark <task id>
```

4.4 Switching between Default and Completed Views

By now you should be familiar with the **default view**. It shows all your uncompleted tasks, separated by task type and sorted by ascending date.

To view your completed tasks, Celebi provides the **completed view**, which shows your completed tasks in descending order of their completion date.

To switch back to the **default view**:

```
>> clear
```

To switch to the **completed view**:

```
>> show completed
```

Those two views are the only two pre-set views Celebi provides you.

If you need more control over your active view and organising your tasks, Celebi provides several features like tagging, filtering, and searching (*Section 5*)

4.5 Re-opening Tasks

Oops. Maybe you haven't done a good enough job.

Or something crops up and you have to go back and redo a task.

It's ok. Celebi won't judge you.
Celebi will silently re-open the task for completion, and even pretend nothing happened.

If you need to redo your math assignment (ID 1):

```
>> show completed (to load completed tasks into view)
>> reopen 1
```

The general format is:

```
>> reopen <task id>
```

4.6 Deleting Tasks

Perhaps you got lucky and your boss reassigns your task to another person.

Maybe you just want some privacy.

Either way, Celebi will willingly develop amnesia, just for you.

To delete your math homework assignment (ID 2):

```
>> delete 2
```

To delete all completed items:

```
>> show completed (switch to completed view)
>> delete all
```

To delete multiple items from the current view (IDs 5, 6, and 7):

```
>> delete 5 6 7
```

To delete all items in current view:

```
>> delete all
```

The general format for multiple deletions is:

```
>> delete <task 1> <task 2> ... <task N>
```

Where:

- **<task1> <task2> ... <task N>** are the IDs of the N tasks you want to delete.

The shortcut to delete all tasks in the current active view is:

```
>> delete all
```

4.7 Undo Last Command

You make a mistake. A typo in your last command to Celebi. If you are unlucky, that was a delete command, and you deleted an important task.

Or you are lucky, and you simply added an unwanted task.

You freeze, then breathe a sigh of relief as you type:

```
>>  undo
```

The undo command reverses the effects of your last task-altering command. Task-altering means that your list of tasks has changed.

This means the undo command only works on the: [add](#), [edit](#), [complete](#), [reopen](#), [delete](#) commands.

Celebi stores every command you have entered since you opened it, so you do not have to worry about making an irreversible mistake/

4.8 Help

Celebi can help you list all commands directly, if you type:

```
>>  help
```

If you need information on a particular command, type:

```
>>  help <command>
```

Where:

- **<command>** is any valid keyword (first word) for the command you want help with.

Example for getting help for the edit command:

```
>>  help edit
```

5 Using Celebi (Intermediate)

Learn how Celebi can organise your tasks.

5.1 Searching and Filtering

Tasks are piling up, and now it is getting hard to search through the tasks looking for the ones you need.

Do not worry. Celebi offers you a fine level of control over the tasks being shown.

You can create your own temporary custom view by **searching** or **filtering**.

If your last accessed pre-set view was the **default view**, the search/filter will only account for the tasks in that view, the **uncompleted tasks**.

If your last accessed pre-set view was the **completed view**, the search/filter will only act on the tasks in that view, the **completed tasks**.

Searching

You can search for tasks by the characters in their names.

To display all **uncompleted** tasks with "math" in their names:

```
>> clear
>> search math
```

This will restrict the active view to show only uncompleted tasks with "math" in their name.

To display all **completed** tasks with "Harry Po" in their names:

```
>> show completed
>> search Harry Po
```


The general command format for searching is:

```
>> search <search key>
```

Where:

- **<search key>** is the substring (any sequence of characters) you want to find within the task names.

Filtering (By Dates)

Most often you want to find all the tasks that need to be finished before a certain date.

Celebi lets you filter tasks by date relative to a specific reference date you provide.

To get all tasks that need action before next week:

```
>> filter before next week
```

To get all tasks that will occur or become due after 20 November this year:

```
>> filter after 20/10
```

The general format is:

```
>> filter before|after <reference date>
```

Where:

- **before|after** tells Celebi to search for tasks before or after the reference date
- **<reference date>** is the date you are comparing with. Format follows examples and *Section 6.2*.

```
>> filter between <first date> and <last date>
```

Celebi also allows you to select a date range and retrieve all tasks that fall inside that range. So, if something urgent cropped and you need to shift your tasks and appointments, this is the command to use.

5.2 Setting Custom Command Aliases

If the commands are too troublesome to type, you can specify your own command keywords as shortcuts.

To map a new alias for the delete command, type the following command:

```
>> alias delete x
```

Now you will realise that you can just type "x" in place of "delete" to speed up your task deletions!

If you made a poor alias mapping, you can always overwrite it by performing the alias command again. However, if you feel like your custom aliases are too cluttered and wish to start anew, just type:

```
>> alias reset
```

And all your custom aliases will be cleared again.

6 Using Celebi (Advanced)

Delve into Celebi's inner workings and optimise your workflow.

6.1 Changing the Storage File

By default, Celebi stores all your tasks in a human-readable format, as a text file "tasks.txt" in the same folder that Celebi is in.

If you wish to change the save file location and name for any reason, use this command:

```
>> saveto <new file path>
```

6.2 Valid Date Formats

Celebi is smart enough to understand most date formats it receives in its commands, and does not treat dates in a case-sensitive manner.

However, Celebi is not perfect. If your date is not being understood and you must know why, you can check this section for simple examples and an in depth look at how Celebi parses text into dates.

For this section, "time" refers to the time within the day, "date" refers to the calendar day, "datetime" refers to the absolute date.

Celebi does not allow dates in the past.

Relative Dates (Convenient Shorthand)

Celebi understands common English temporal statements.

In the absence of time statements, Celebi takes the time to be the current time.

In the absence of date statements, Celebi takes the date to be today.

User text	Resultant Datetime
Date – level statements	
Today	Today
Tomorrow, tmr	1 day later
Next week	1 week from now
Next month	28 days from now
Monday - Sunday	The next occurrence of the weekday.
Next Monday-Sunday	The corresponding weekday in the next calendar week.
X days later	X days/weeks/months from now
X weeks later	
X months later	
Time – level statements	
Now	Current datetime (supersedes date-level statements)
Noon	1200h
Midnight	0000h
X hours later	Time X hours from now
Dawn	0600h
Morning	0900h
Evening	1800h
Night	2100h
Any combination of above Date and Time level statements: combine datetimes	

Figure 3: Relative date parsing table

Absolute Dates

You should specify absolute dates in full. Prioritise the larger fields first (ie Year Month Day) as opposed to (Day Month Year) or (Month Day Year) to avoid any ambiguity.

Celebi Developer's Guide (v0.1)

7 Welcome

Welcome to Celebi's developer guide.

If you are reading this, you should be familiar with common software engineering terminology and have some coding experience under your belt. If you have neither, this guide may not be for you.

The developer guide's purpose is to orientate you, the aspiring new developer, on Celebi's software architecture and component design. Whether you wish to extend Celebi's functionality with your own plugins, or look at how things work here, you will get a working understanding of Celebi's entire design.

This guide does not go too in-depth into the inner workings of the classes and components, but will provide you with a high-level overview of their purposes and execution flow. If you need to see the full implementation, you can find it in the repository provided below in *Section 8*.

Have fun!

8 Setup

8.1 Source Code Repository

The entire project can be found at <https://github.com/cs2103aug2015-w11-3j/main>.

If you are not coming on board in an official capacity, you can still feel free to fork the project and play with it. If you have implemented any improvements, feel free to contact either of us through our github accounts and submit a pull request.

8.2 Development Environment

Celebi was developed exclusively in Java. We used Java version 8, so you should prepare that version at a minimum. We used two external libraries as listed below. Make sure you add them to your build path.

Oracle: JavaFX

License: [Oracle Binary Code License Agreement for Java SE Platform Products and JavaFX](#)

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.

We use the JavaFX library to design and implement our graphical user interface. It is used by the UI architectural component.

For more information, visit <http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm>

JSON.simple

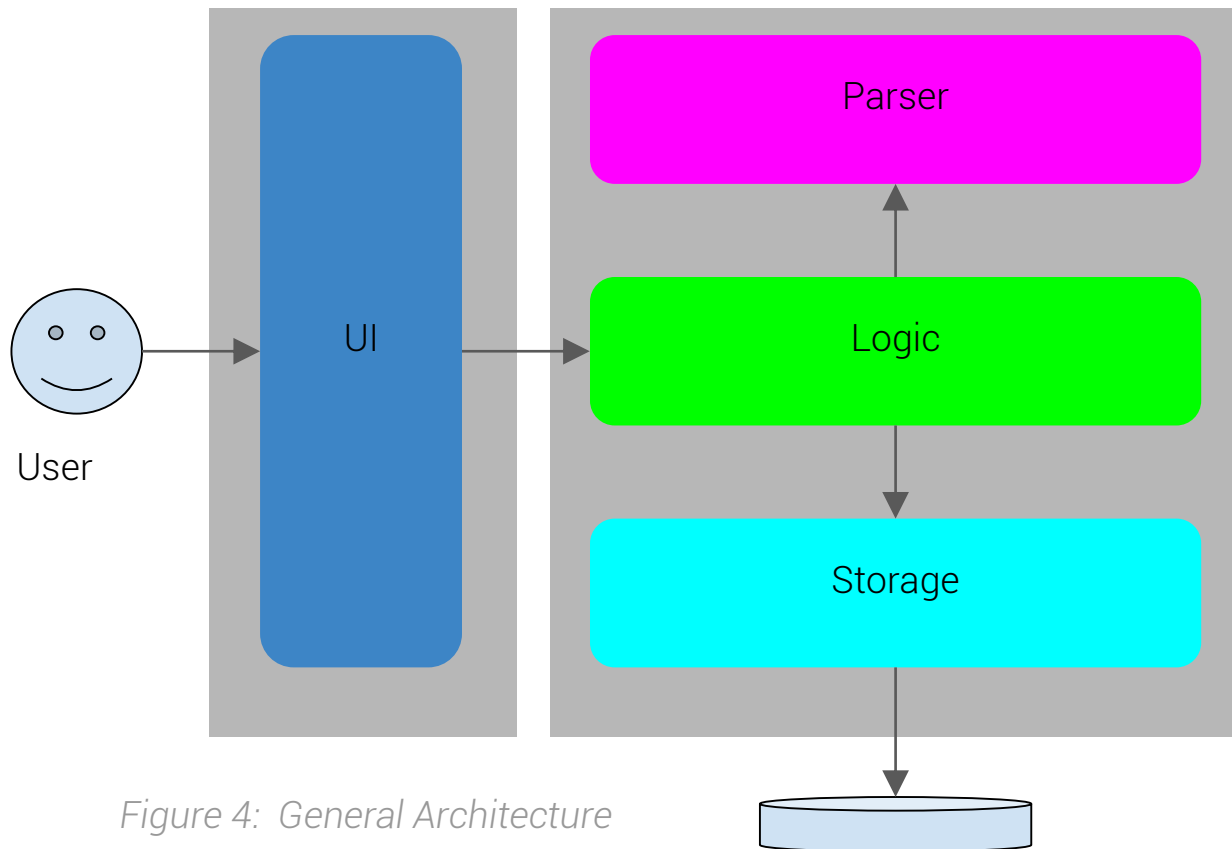
License: [Apache 2.0 License](#)

JSON.simple is a simple Java toolkit for JSON. You can use JSON.simple to encode or decode JSON text. It implements the RFC4627 JSON specification.

We use the JSON.simple library to serialise our persistent data across use sessions. It is used by the Storage architectural component.

For more information, visit <https://code.google.com/p/json-simple/>

9 Architecture



The front-end is denoted by the gray rectangle on the left. It contains only one component, the UI component.

The back-end is denoted by the gray rectangle on the right. It contains the other three components: Parser, Logic, and Storage.

The façade design pattern is enforced here. With the exception of the UI component, all other components have an interface to fix its API for inter-component interaction and to define its behaviour.

Coupling is minimised; you can see that the Parser and Storage components have no dependencies on other components.

All component classes are grouped by component at the package level. Not shown here is the common package, which contains miscellaneous datatype and container classes. Every component will rely on the utility classes in the common package.

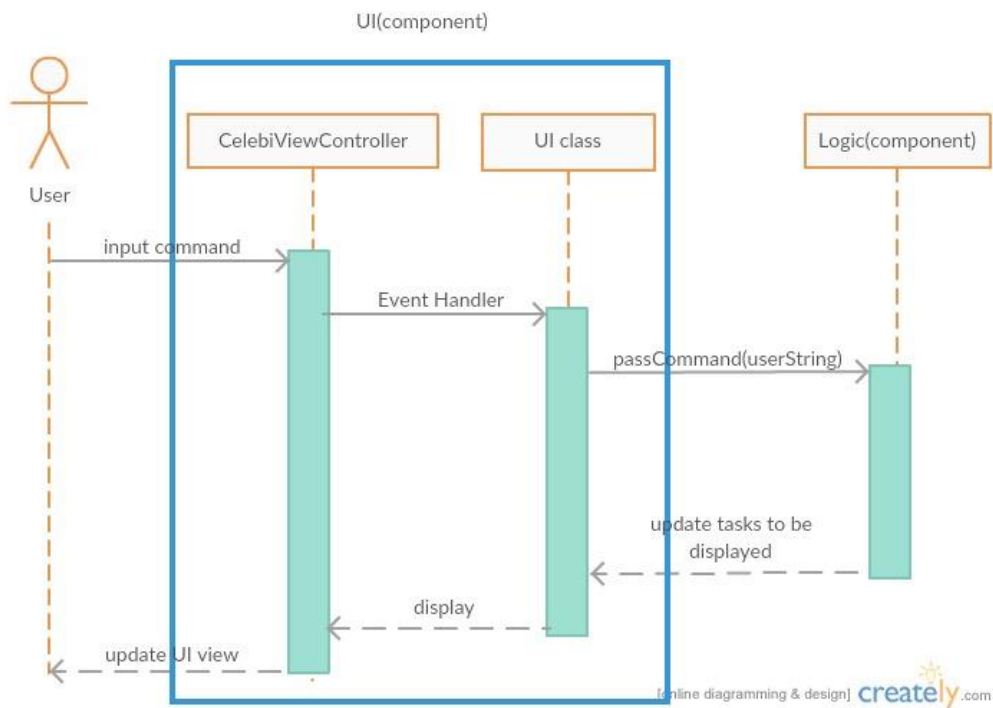


Figure 5: UI front-end sequence diagram

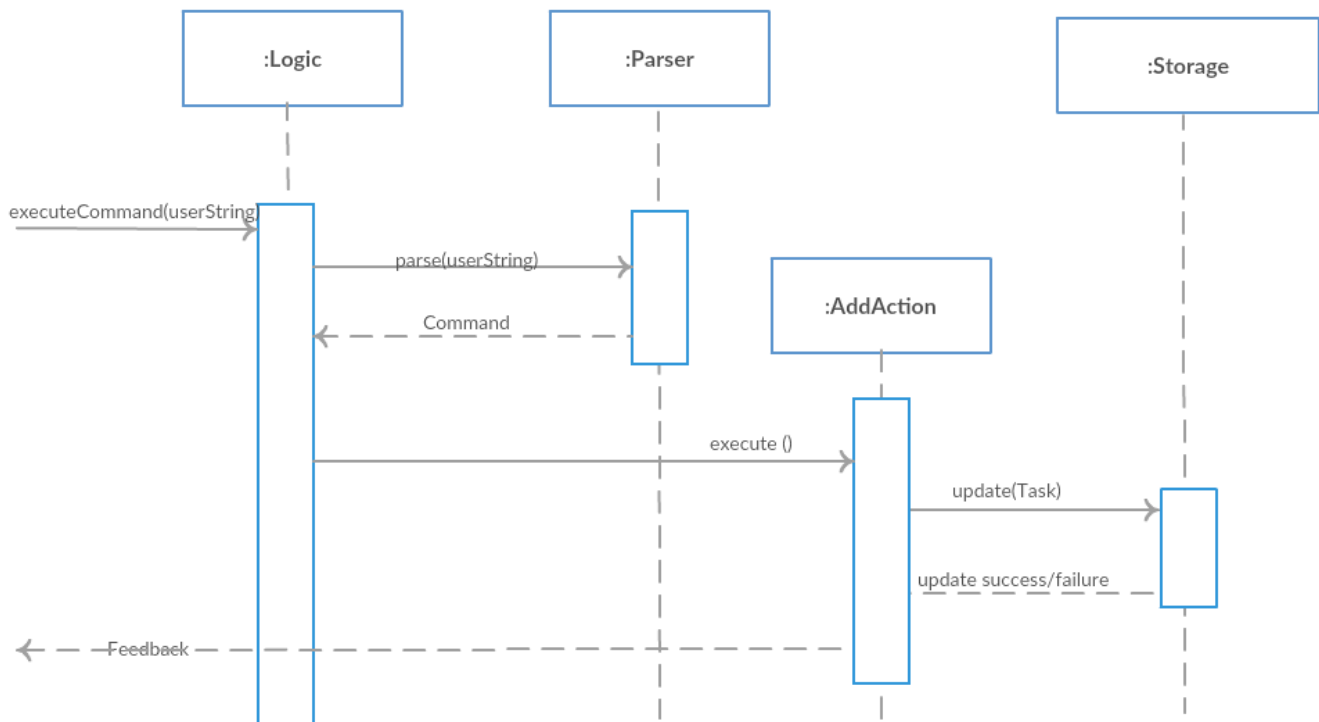


Figure 6: Logic, Parser and Storage back-end sequence diagram

10 The Common Package

The common package currently holds 3 notable classes: Task, TasksBag, and Configuration. The Task class and the TasksBag class represent the central data objects for Celebi. The Configuration class is slightly different. It acts as the data holder object for the user's configuration settings, and also provides enough heavy logic to elevate it above the status of a mere data container. Read more in *Section 10.3*.

10.1 The Task Class

This class represents the datatype for Tasks. Recall that Celebi is a task manager, and the important user data are all in Tasks. The Task class, when instantiated as individual objects, each represent one unique task. There is a one to one mapping between the abstract task objects the user manipulates, and the actual data which is stored in Task objects.

Task object data fields:

These are the data used in Task's setter and getter methods. Within the Task class itself, there is some data wrapping using javaFX's ObjectProperty classes, but this will only matter to the UI component.

ID (int) : Each task object can be uniquely identified by this ID. It is separate from the display ID mentioned in the user guide. The ID is never negative, and never the same as another Task's.

name (String) : Each task has a non-empty name. Every task **MUST** have a name and ID. The name cannot be null or the empty string, and should not contain semicolons.

startDate, endDate (Date) : Each task has a start and end date. Any one of them can be null, if they are not applicable (e.g. start date for deadlines, any date for floating tasks). If you intend to extend this class, we suggest that you take advantage of java 8's new datetime libraries and deprecate the util.Date based implementation here.

isCompleted (Boolean) : Represents the task's completion status. If true, the task is considered "completed". It is always possible to toggle completion status. It cannot be null

10.2 The TasksBag Class

The TasksBag class represents a general purpose container for Task objects. The tasks inside the TasksBag do not have a life cycle dependency on it.

Its main purpose is to wrap and pass multiple Tasks between components, and to provide an easily extensible base for implementing collection level operations on Tasks. As of version 0.5, the TasksBag class supports simple date-based sorting and filtering operations.

10.3 The Configuration Class:

The Configuration class represents the user's settings for Celebi. This class directly maps to a file named config.json. That file provides a way for user defined preferences (like custom aliases, UI theme etc.) to persist across sessions. The Configuration class attempts to synchronise the config file data with its own state every time it detects a change.

If the config file fails to load or is otherwise corrupted, the Configuration class also has predefined defaults for each configuration setting, and continues maintaining any new config changes in memory (the user is warned that changes may not persist past shutdown). This way, users will experience minimum interruption.

Data fields:

StorageLocation (String): The location of the user's saved tasks. Defaults to current working directory.

Skin(String): Symbolises the active display theme. Easily extensible with the XML based GUI (see *Section 11.5+*)

Alias(Map): User-defined command keyword mappings. Power users would find it useful to optimise their workflow.

11 The UI Component

The UI component is solely responsible for creating and maintaining Celebi's graphical user interface. The GUI is also the user's sole point of interaction with Celebi. User input is received mainly via text in the command line input box. Minor clickable components also exist, but they do not provide any extra functions beyond those already available through text commands.



Figure 7: v0.1 UI

The user's input is captured as a String through the input box, and is routed to the Logic component for execution.

The UI component also displays a view of all relevant tasks (this view can be altered via user commands) and keeps it up-to-date at all times.

The result of the command execution is also recorded in the command history panel.

The GUI layout is specified in the FXML files under the src/ui/view directory.

There are 3 classes in the UI component: Main, CelebiViewController, and UI.

Main is used as the software's entry point and the GUI's initialiser, and extends the `javafx.application.Application` class. When Celebi is launched, Main's main method is called. Main then sets up the GUI and input event handlers and exits gracefully.

We are more concerned about the other 2 classes, which demarcate the boundary between the user and the back-end. The general sequence of execution is shown in figure 5 above.

UI class and CelebiViewController inside UI component are restricted to have only one instance since they are all singleton classes. This applies the Singleton pattern.

Celebi is designed using MVC(Model-View-Controller) pattern to reduce the coupling of data, presentation and control logic. The back-end part is the model, while UI component is the combination of view and controller. The view in charge of displaying is CelebiView.fxml, and CelebiViewController together with UI class forms the controller.

11.1 The CelebiViewController Class

When the user inputs a command, it passes through this class first. CelebiViewController's event handler then calls the UI class's `passCommand` method with the user input String as the argument. The next execution step will be covered under *Section 11.2*. **Aside from taking in user command, this controller also processes the user's tab key event to switch between the three views. The event is also passed to UI class, but this time the method `passKeyEvent` handles the event directly instead.**

The UI class then calls this class's `appendFeedback` method to show the user the feedback message from executing their command. UI also calls this class's `display` method to update the task table display panel with any changes in the user's tasks.

11.2 CelebiViewController important APIs

```
public void updateTableItems(ObservableList<Task> taskList);
```

Used by UI class's display method to update the GUI's task table view.

```
public void showFeedback(String newFeedback);
```

Used by UI class's passCommand method to display the resultant message from executing their command.

```
public void showWarning(String newFeedback);
```

used by UI class's doDefault method to display the warning message if there is any.

```
public void updateUI(TasksBag bag);
```

used by UI class's display method to update UI looking.

11.3 The UI Class

Acts as an intermediary between the back-end Logic component and the UI component. The UI class abstracts and separates the application's command execution flow from the GUI event and maintenance code, which is encapsulated in the CelebiViewController class.

11.4 UI important APIs

```
public void passCommand(String userInput);
```

Used by CelebiViewController's key press event handler to pass the user's input string to the Logic component for execution. Also pipes the execution feedback messages back to CelebiViewControllers' appendFeedback method.

```
public void display(TasksBag bag);
```

Used by passCommand to update the resultant tasks data returned from the Logic component to the GUI task view panel. Calls CelebiViewControllers' setTableItems method to do so.

```
public void passKeyEvent(KeyCode key);
```

Used by celebiViewController handle the tab key event for switching views.

11.5 FXML files

In the ui.view package, there are two FXML files: RootView.fxml and CelebiView.fxml. They specify the layout of Celebi's GUI. FXML files can be open and edited using SceneBuilder, which is an official visual layout tool for JavaFX applications from Oracle. More information can be found at: <http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html>

11.6 CSS files

Under the ui.resource package, there are two CSS files, which are skinDay.css and skinNight.css. They specify the looking of Celebi under different modes respectively, which are day mode and night mode.

11.7 TTF files

Under the ui.resource package, there are two TTF files, which are Oxygen regular.ttf and Oxygen 700.ttf. They are the default fonts of Celebi. They are free fonts that can be retrieved from Google Font.

GUI Test:

The testing under UI is done using FXRobot inside TestFX library. It is a system test that includes tests for several use cases. It mainly focuses on testing the correct behavior of UI component such as command area, feedback area, tableView and tab. It also tests the skin mode.

To set up the test environment, make sure you have added the TestFX and Guava library into project path. After that, simply run the UITest.java from inside ui package.

12 The Logic Component

The Logic component acts as the central processing component and communicates with all other components. It is responsible for directing user input to the Parser component for parsing, instructing the Storage component to save and retrieve user data where necessary, and relaying command execution feedback and any changes in state back to the UI component.

Logic also implements the command validation and execution logic. It is, in essence, the central “brain” of Celebi, and is the glue that binds the other components together.

The notable classes would be discussed below.

12.1 The Logic Class

As Logic brings many different subcomponents together, Logic exploits the use of Façade pattern to provide a unified interface for the UI to use.

The Logic class has two distinct entry points to interpret a user's actions.

```
public CommandFeedback executeCommand(String cmd) throws LogicException;
```

String commands which the user enters into the textbox are passed into this function to evaluate the command.

```
public KeyEventFeedback executeKeyEvent(KeyCode whichKey) throws  
LogicException;
```

The other form of input that is, key command which are special keys which the user presses, should be passed into this function to evaluate the command.

The Logic class follows very sequential steps to execute a user's string command.

First, it calls Parser's `parseCommand` with the user's input string and retrieves the corresponding command object.

Parser has its own validation checks on the input's format, but Logic will perform subsequent checks on the Command data's values to make sure the values are acceptable.

If the command is illogical, like an attempt to set a task's start date after its end date or like an attempt to work on a non-existent task, Logic will throw an exception to notify UI and subsequently the user of this illegal command.

If the command validation checks are passed, Logic will execute the command with its internal private methods, returning a Feedback object to UI for user feedback.

12.2 Feedback Class

The use of Feedback Class is to act as a protocol where the Logic and UI communicates. It has a container class, Tasks Bag, containing all the task that the Logic is currently managing and allow the UI to display them accordingly.

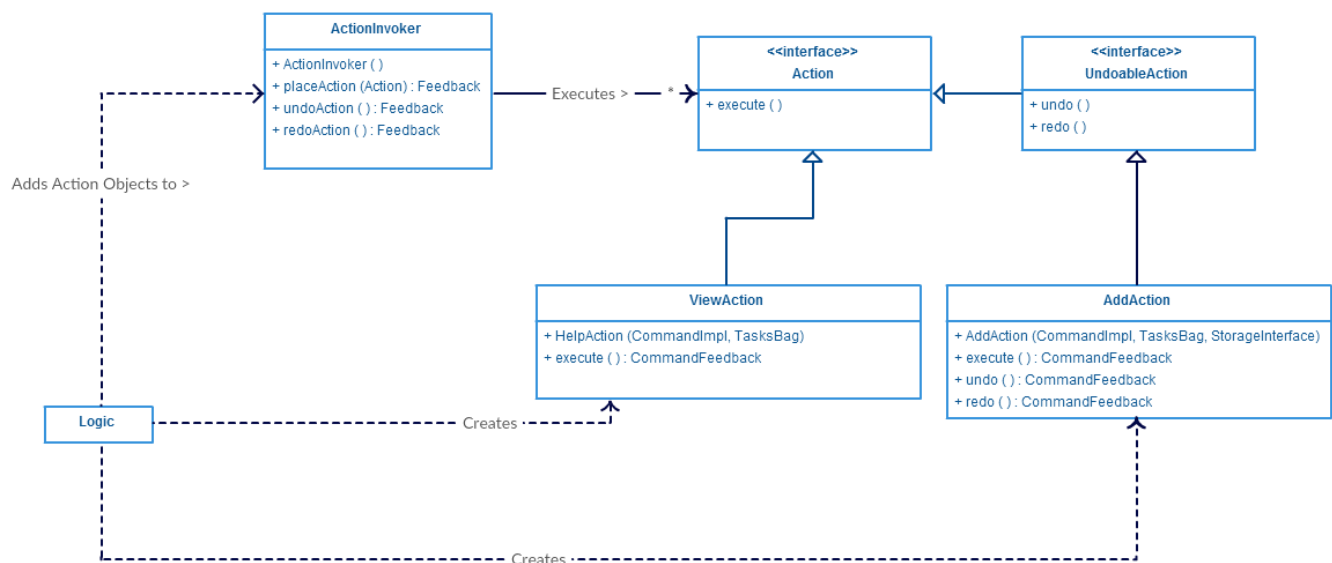


Figure 8: Logic Action Class Diagram

The extended `CommandFeedback` class provides more functionalities such as user message and warnings of the resultant action. This class is the return type of `executeCommand` as seen above.

The other extended class which is `KeyEventFeedback`, provides the user message and the `KeyCode` which was pressed.

12.3 ActionInvoker class

As there are a vast number of different actions that the user can choose to perform, we have chosen to use the Command Design Pattern to decouple the actual actions from the Logic class. The diagram below illustrates the classes involved and their interactions.

Once the Logic class identifies the user's command from the Parser, it creates the respective actions and passes it over to the ActionInvoker class.

The ActionInvoker class provides the execution of the undo and redo stack as well as its management. The redo stack would be emptied upon successful execution of a new action so as to prevent inconsistency when executing a redo.

12.4 Action classes

The various action classes represent each different unique command given by the user in an object form. This provides us with the benefit of undoing and redoing the user's action. In addition, actions are categorized under the two major categories, Undoable Actions and Actions.

Undoable Actions are extended actions which can be undone and by extension, redone. The list of actions can be found in section { *Undoable Actions* }. One important thing to note is that as we Celebi provides multiple views for the user, it is highly recommended that during construction time of the Action, the relevant task should be selected and pointed to at that time instance. As the switch might switch views in between Actions, using the reference number of the task might not be the intended task to undo or redo.

Actions are actions which cannot be undone. Actions which do not mutate tasks, for instance changing themes, are generally categorized here. The list of actions can be found at section {something2} as well.

12.5 Logic Testing

The test cases for Logic component can be found in the LogicTest class. As of v0.5, 70% of the code blocks have been covered in the Logic components. The environment settings have been setup in the test before and after functions. It is recommended to provide new test cases for the new functionalities added and perform regression testing to ensure full functionalities of other functions.

Undoable Actions

Class Name	Execute Behaviour
AddAction	Adds task into TasksBag.
UpdateAction	Updates task's name and dates.
DeleteAction	Removes the specified task from TasksBag.
MarkAction	Marks the tasks as completed.
UnmarkAction	Marks the task as incomplete.

Actions

SearchAction	Sets the current search state to the given String.
FilterDateAction	Sets the current date search state to the given date or dates.
FilterClearAction	Clears both the search and date search state.
MoveFileAction	Moves the persistent storage file to the provided folder.
ThemeChangeAction	Switches the theme of Celebi to the provided theme.
AliasAction	Sets an alias to another action.
HelpAction	Display the generic help.
ViewAction	Switches the current view to the provided view.
ViewToggleAction	Cycles the current view to the next view.

13 The Parser Component

The Parser component is responsible for extracting meaning from the user's input string. It performs checks on the input's **format**, but not its meaning.

The Parser component consists of a single Command datatype class and the main Parser class.

Logic calls the Parser class to parse the user input. After extracting all useful information from it, the Parser class then wraps it up in a Command datatype object and returns it the Logic class for further processing.

The internal parsing logic is implemented using a mixture of regular expressions, string transformations, and switch cases.

13.1 The CommandData Interface

Classes implementing this interface act as data containers to represent user commands. A CommandData object holds all the information needed for Logic to validate and execute the user's input. This class implements the CommandInterface interface, which details the relevant getter methods. The class's constructor is package private to prevent other components from creating their own Command objects.

Since it is a datatype class, its important APIs are its getter methods. The CommandInterface.java file in the parser package details each getter and its usage.

Command also holds a public static enum for all possible command types. External classes like Logic and UI will use this enum's values to distinguish the different command types.

The current implementing class of this CommandData interface is called CommandDataImpl. It implements simple setters and getters. If you wish to implement some sort of tracking or further processing, you can swap your own implementation into the system.

13.2 CommandData important APIs

There are too many useful methods, and all of them are simple getters. You can read them in the `CommandInterface.java` source file. You should take note of the `CommandData.Type` enum which uniquely identifies the specific action this `CommandData` object represents.

These enum values represent all the possible user commands as detailed in the User Guide. Names clearly reference their command.

13.3 The ParserControllerImpl Class

This class takes the user input string from Logic and parses it, returning a `Command` object with the necessary data for command execution.

The Parser uses a mixture of regular expressions, dateformatters, and manual tokenisation to parse the raw user input. If we define a "token" as a piece of the input string that carries context and meaning, then the Parser tries to tokenise the input, searching for those tokens.

This class's execution flow can be split into 3 steps: multiplexing, parsing, and creating the appropriate `CommandData` object.

At the start, it extracts the first token and parses it as the keyword to identify the specific command the user wishes to execute. The input string is then transferred to the appropriate helper function/class for further processing. These helpers do the necessary parsing and front-end validation of the user's arguments, then finally they call the simple factory methods to instantiate the appropriate `CommandData` object. Finally, the `CommandData` object is returned to logic for further processing.

You can easily extend the functionality of the parser component by replacing the helper functions or factory methods and redirecting them to your own classes.

13.4 ParserControllerImpl important APIs

Like the Logic class, the Parser class is decoupled and self-contained with a single entry point: its `parseCommand` method. It is the only way this class receives input from external classes and the only way this class sends output back to the external classes.

```
public Command parseCommandData(String userRawInput);
```

It is used by Logic's `executeCommand` method. It calls no other external classes.

```
public static ParserController getInstance();
```

The singleton accessor for this class. It is especially useful for Parsing classes to implement the singleton pattern, especially in a single threaded environment. Regex engines and date formatters are expensive to create, so using the singleton can ensure that your cost is paid at the start and will not slow down the other parts of the program/

13.5 The AliasesImpl Class

The `AliasesImpl` class extends the `Aliases` abstract class, and provides a façade for all queries related to user input keywords. For example, logic may need to know which keywords are reserved so the user does not accidentally overwrite them with the alias command, and UI may want to know what keywords and aliases are currently active to perform syntax highlighting.

The `Aliases` abstract class also centralises all pre-defined keyword data in one place, so if you wish to add a new keyword alias or change some existing keywords, you as a developer can do it directly from the class's files.

13.6 Aliases Important APIs

```
public boolean isCmdAlias (String alias);  
  
public abstract boolean isCustomAlias (String alias);  
  
public abstract boolean isDefaultAlias (String alias);
```

These are used for any class that needs to know whether a token string matches a subset of command aliases. There are 3 subsets: user-defined custom aliases, preset default aliases, and reserved keywords.

```
public abstract boolean isReservedCmdAlias (String alias);
```

Checking for reserved keywords can be done through this method. Most notably, Logic's AliasAction class uses this to make sure the user does not lock themselves out of their own commands.

```
public abstract CommandData.Type getCmdFromAlias (String alias);
```

Used by ParserController and any other related Parsers for identifying and dispatching input strings to their respective command handlers.

```
public abstract void setCustomAlias (String alias, CommandData.Type target)  
throws IOException;  
  
public abstract void clearCustomAliases () throws IOException;
```

These methods link up with the common package's Configuration class to synchronise the user's aliasing operations with the parser component, logic component, and the actual disk state.

14 The Storage Component

The Storage component is responsible for serialising, saving, and loading persistent user data and state across launch sessions. The save file path can be customised by the user.

The persistent data is written to the file in a human-readable format. The notation it uses is JSON, using the JSON.simple library for parsing and data conversion.

Storage parses the file's JSON into individual Task objects when loading, and serialises and writes all the user's Task objects as JSON into the database file. The only external class that calls the Storage component is the Logic class.

The Storage component consists of three classes. They are Storage, TaskJson, and Database. The Storage class acts as the gatekeeper and entry point to this component. The TaskJson class is a special datatype class that acts as the bridge between a pure Task object and its serialised JSON form. The Database class handles the actual file I/O, parsing, and serialisation for saving and loading.

14.1 The TaskJson Class

A subclass of JSONObject which specifically represents a task in Celebi. It is designed for the translation between JSON object and the Task class. It's fields have a one to one mapping with those in the Task object.

When loading Tasks from the database, the file JSON is first parsed into a TaskJson object using the simple.json library, then finally converted into a normal Task object.

When saving Tasks to the database, it goes the other way round: Task objects are first converted into TaskJson objects, then finally serialised as JSON into the database file.

It has no notable APIs relevant for external use. Important methods include its constructor for Task->TaskJson conversion and simple getters for Storage to perform TaskJson -> Task conversions.

14.2 The Storage Class

Storage is the only class in the component to interact with external classes. In its interface, it defines only three public methods to modify the storage file: *save*, *load*, and *delete*. It is self-contained and its internal workings remain hidden from outsiders. Logic invokes Storage upon any changes in state, and Storage then controls the Database class to perform the actual I/O.

You can observe Storage's interactions with Database through the class diagram and sequence diagram

14.3 Storage important APIs

```
void init();
```

Initialises the component. It will find the storage file, or create one if not exists.

```
boolean save(Task t);
```

Serialises and saves Task *t* to the database file, returns true on success. Called by Logic upon additions and updates to Tasks. Returns true if execution is successful

```
void load(TasksBag bag);
```

Loads all Tasks from the database file into the TasksBag *bag*. Called by Logic for state initialisation and loading of the latest state.

```
boolean delete(Task t);
```

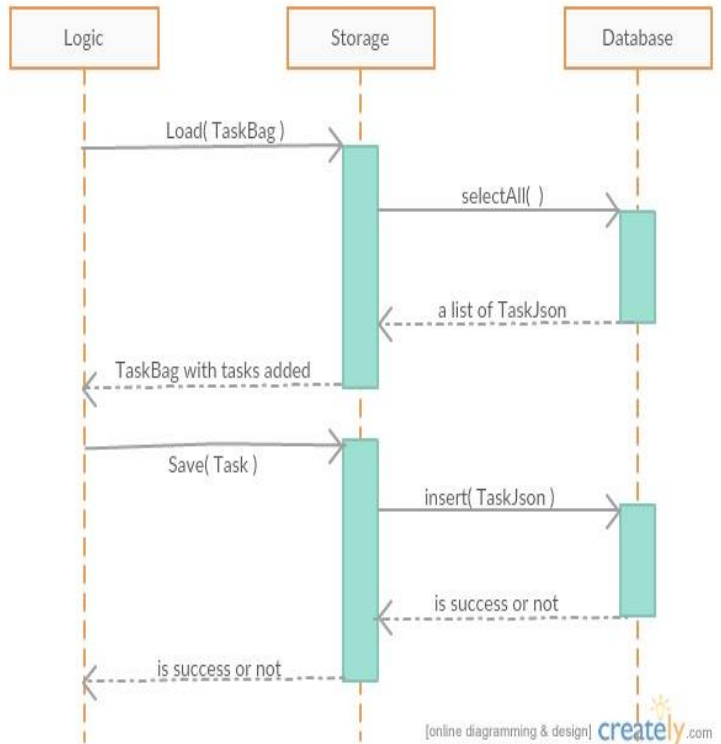
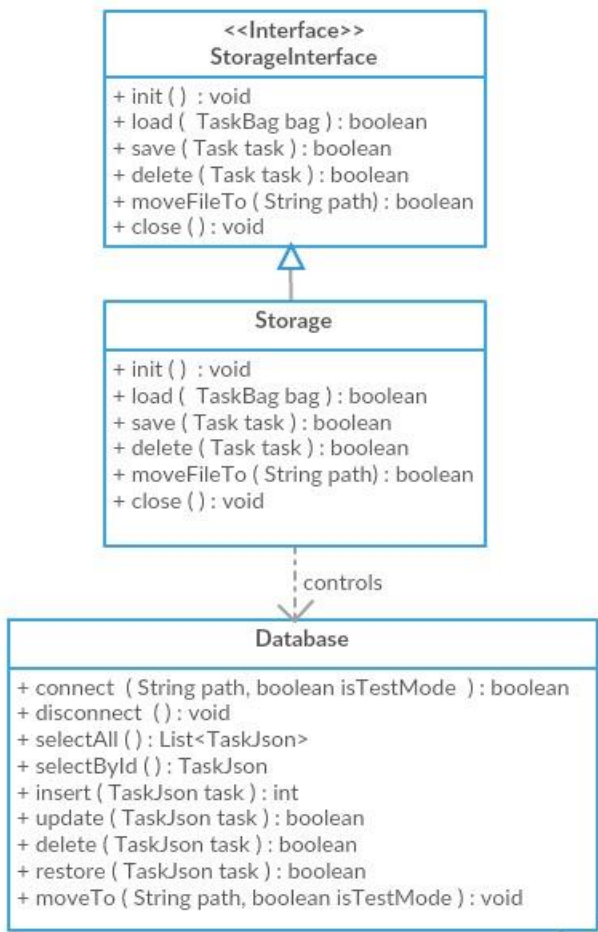
Deletes *t*'s data from the database file, effectively removing it from persistent data. Called by Logic when executing delete commands. Like *save()*, it returns true upon success.

```
void moveFileTo(String destination)
```

Moves the storage file to a new location specified by *destination*. Throws an exception which will be caught by logic, if the move fails due to an I/O issue.

```
void close();
```

Closes the file and releases resources. Also resets all data fields before ceasing.



Figures 9 and 10: Class diagram and Sequence diagram for Storage component. In Figure 10 (sequence diagram), the general use case for Storage is shown.

14.4 The Database Class:

Implements the actual file I/O logic, including appending, insertion, removal, loading of binary data, as well as moving the storage file to a certain location. Also handles serialisation and JSON load parsing.

Database keeps a direct and strict map of the content of tasks.json, which is an array of the tasks (in JSON format). It is a class that not designed to be used by other components. All its methods are either private (used by itself) or only accessible within the Storage component (used by Storage class).

14.5 Database Important APIs

```
boolean connect(String path, boolean isTestMode);
```

Look up the file named tasks.json in the specified directory, if not found, it will create this file. Returns true if found or created, returns false if encountered exception.

```
boolean load();
```

Read from the connected file, to load all the persisted tasks. Returns true if successfully loaded, returns false if Database is not connected to any file or exception occurs.

```
boolean disconnect();
```

Close the file, reset all the data. Returns true if successfully close the file, returns false if Database is not connected to any file.

```
List<TaskJson> selectAll ();
```

Returns all tasks loaded from tasks.json as a list.

```
TaskJson selectById(int id);
```

Returns the task specified by storage id. If the task of that id does not exist, return null.

```
int insert (TaskJson task);
```

Add a new task into storage file, returns the storage id of the newly added task.

```
boolean delete (TaskJson task;
```

Remove a task from the storage file, returns true if successfully removed.

```
boolean update (TaskJson task);
```

Update a task in storage file, returns true if successfully updated.

```
boolean restore (TaskJson task);
```

e-add a deleted task in storage file to its original position with its original id, returns true if successfully updated, returns false if exception encountered.

```
boolean moveTo (String destination, boolean isTestMode);
```

move the storage file to a new location. Throws exception for Logic to catch if movement fails.

Appendix A: User stories

[High Priority] (likely)

ID	I can ... (i.e. Functionality)	so that I ... (i.e. Value)
Create		
addEventTask	add tasks that have start and end dates	to support events with clear start and end date
addDeadlineTask	add tasks with deadlines but no start dates	can track tasks without start dates
addFloatingTask	add free floating tasks that do not have dates	can add tasks without associated dates
taskFieldTitle	specify a title for any new task	can distinguish tasks with my own custom title strings
Read		
displayTasks	display tasks with a simple command	so that i can view all my saved tasks
Update		
updateTask	update any task's data fields	dont need to delete and recreate a task if its properties change in real life
Delete		
deleteTask	remove/delete my tasks	to clear clutter and for privacy
Input parsing		
parseDateNum	be able to enter dates in a numerical format	dont have to enter full dates as text
Others		
closeApp	type command to close the app	can exit the app from within the command line just like with other comands
commandHelp	type command to get information of all legal commands	can check commands I am unsure about
Data extensions		

taskFieldCompleted	mark tasks as completed	can separate and archive tasks by completion
--------------------	-------------------------	--

[Medium Priority] (likely)

Data extensions		
taskFieldDescription	Insert description of my tasks	so that i can include more information to the title
taskFieldScheduleWork	Assign which days I want to make progress for tasks (events excluded)	can plan and schedule my task workload
taskFieldTags	Add tags to my tasks	Can separate properties that define my tasks
taskFieldPriority	set tasks with different priorities	Easily prioritise my tasks
taskFieldRecur	setup constant period recurrence for my task	save time readding recurring tasks
Display		
displayTodaysTasks	View the tasks marked for today's agenda	can know what is marked for completion today and do them
commandFeedbackMsg	Get a confirmation message when entering a command	is know my command is not misunderstood and is successful
displayCompleted	View my tasks history	for archival purposes
Search		
searchStrings	do a text search using name and description	I can find an item that i have forgotten using keywords
filterTags	Filter tasks based on tags	can look for tasks that belong in the same group
searchDates	Filter tasks using a date to compare to	find tasks that need to be done before a certain date

Sort		
sortDates	sort tasks by their start/end dates	so i can see which tasks are urgent
sortPriority	sort tasks by priority	so i can see which tasks are important
Others + input parsing		
undoLast	Undo my last task i performed by accident	can easily recover from a careless mistake
changeStoreLoc	Choose location of my data storage on computer	specify the best location based on my needs
parseDateAlpha	Enter dates in common alphabetical formats (monday,tues, etc)	Can copy dates from outside applications directly for parsing

[Low Priority] (unlikely)

recordStatistics	check my statistics of past tasks and performance	see how well i am doing
setCommandAlias	Set custom alias for commands	can use my own convenient shorthand
taskHierarchy	link tasks together as master task and subtasks	organise and split tasks into seperate trackable subcomponents
parseMessage	paste a message/email's contents directly and have it parsed	save a lot of time from data entry
addCustomFields	add my own custom text fields to the tasks	have multiple organisable sections for different description parts
customRecurrence	specify my own complex rules for recurrence	automate complex task recurrences

Appendix B: Non Functional Requirements

The software should...

- Run on Windows 7 or later.
- Run without Internet connection.
- Not require a mouse to access any function.
- Have some GUI.
- Save regularly enough that user never risks a big loss of progress.
- Ask users for the location to save/refer to their data.
- Allow more than one format for commands to be accepted.
- Make it easy to identify task timing clashes visually.

Store save files in a human-readable format.

Appendix C: Product survey

Product: Wunderlist **Documented by:** Ken Lee Shu Ming

Strengths:

Drag drop and reorder tasks
Create folders to categorize tasks
Can click to reveal details (normally only have title)
Simple GUI
Textbox placeholder to instruct
Can add/delete subtasks
Completed tasks are not shown on default

Weaknesses:

Unable to remove tasks
No idea where my data is stored
Unable to sort or quick search

Product: Things **Documented by:** Liu Yang

Strengths:

Simplified user flow
Can add tags and notes
Clean and neat User Interface
Quite flexible, can choose to be shown how many days in advance
Can add projects
Can add tasks that 'you may finish on some uncertain day'
Can trace the log
Has a global search on the right bottom corner

Weaknesses:

Need to pay
Require download
Only have end date field, but no start date
Hard to find a way to delete task (only drag, but not obvious)
Unable to undo operations HI

Product: any.do **Documented by:** Leow Yijin

Strengths:

Intuitive drag and drop to shift tasks
Easily set reminders
Prompts you to plan your tasks for the day everyday
Syncs across all devices
Phone app has shortcuts for common actions like calling, email etc
Supports subtasks for organisations
Attach any type of file
Share and delegate tasks among multiple users
Search covers all text withing task objects

Weaknesses:

Calendar view depends on another app from the same developer
Discontinued desktop app in favor of webapp, now needs connection
No advanced tagging support for better filtering control
Only 3 layers of abstractions: List > Task > Subtask
No support for undoing operations

Product: Todoist **Documented by:** You Jing

Strengths:

The events are well organized. The classification is very clear.
It's very good that user is able to view today and this week's event list easily. This helps the user to decide what to do next.
Good that can mark an event as done & important.
Can see both completed and incomplete tasks.
It has simple search so that the user can find the event easily.

Weaknesses:

This is an online application, which can only be used with Internet. But the user wants to use it offline.
Event only has title, no detail description.
Only two types of events: with ddl and without ddl.

[w11-3j][V0.1]

Does not have undo operation.

Does not provide a way to show which time slot is block.