

TaskBuddy

User Guide v0.5

Welcome to TaskBuddy!
All tasks displayed.

< > today

November 2015

month week day

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	10a - 11a ID : 3 Desc : buy lunch	10a - 12p ID : 10 Desc : sth	9p - 7:05p ID : 2 Desc : EE3204 lab
8	9	10	11	12	11:59p ID : 5 Desc : cs3241 lab5	14
15	11:59p ID : 4 Desc : cs2104 lab5	16	17	18	19	20
22	23	24	25	12p - 2p ID : 6 Desc : cs2103 exam	26	27
29	30	1	2	3	4	5
6	7	8	9	10	9p - 9p ID : 7 Desc : startup weekend singapore	12

cs2103 demo ID: 1

Venue: Empty

Period: Empty

Deadline: Empty




study guide ID: 12

Venue: Empty

Period: Empty

Deadline: Empty

Supervisor: Tong Chun Kit Extra feature: GoodGui

		
<p>Jerry Tan Si Kai Team lead Backend</p>	<p>Audrey Tiah Documentation Design Code Quality</p>	<p>Jean Castillo Deliverables Integration Scheduling/tracking</p>

User guide

Introduction

TaskBuddy is a simple task management software that allows for quick scheduling of your day to day activities. By using simple natural language commands, you will be able to access these commands shown below. Note that required arguments are written in **red**, while optional arguments are in **bold**:

Keywords are reserved by the program to aid in parsing input. They are highlighted in **green**. Important parameters are supposed to follow the keywords as shown below. Refrain from using the keywords for other purposes. However, they can be escaped by enclosing them in double quotes - “ ”.

As you are reading the use cases below, note that certain arguments are to be enclosed in double quotes. This is to allow keywords to be interpreted literally within the double quotes.

Order of the keywords do not matter. For example, when adding a new task, you could specify the venue first, followed by the start**Time** and end**Time** followed by description with this:

add at “NUS” on 12/10/2015 from 0800 to 1000 do “attend CS2103 lecture”

Date and Time Format

For fields that require date :

For a full listing of formats supported, please visit <http://natty.joestelmach.com/doc.jsp>.

Please note that for number date formats, TaskBuddy follows the MM/DD/YYYY or MM/DD/YY format. The year field can be omitted and the date will be defaulted to the current year.

Examples:

21/05/2016

21/05/16

21/05

For word date formats, TaskBuddy does not follow a specific format. Below shows some of the words recognised by TaskBuddy.

Examples:

Tomorrow

next thursday

next month

For time format, the supported formats are as follows:

HHmm : 2345

hhmmaa : 1130pm

Shorter forms: 11pm, 10:30pm

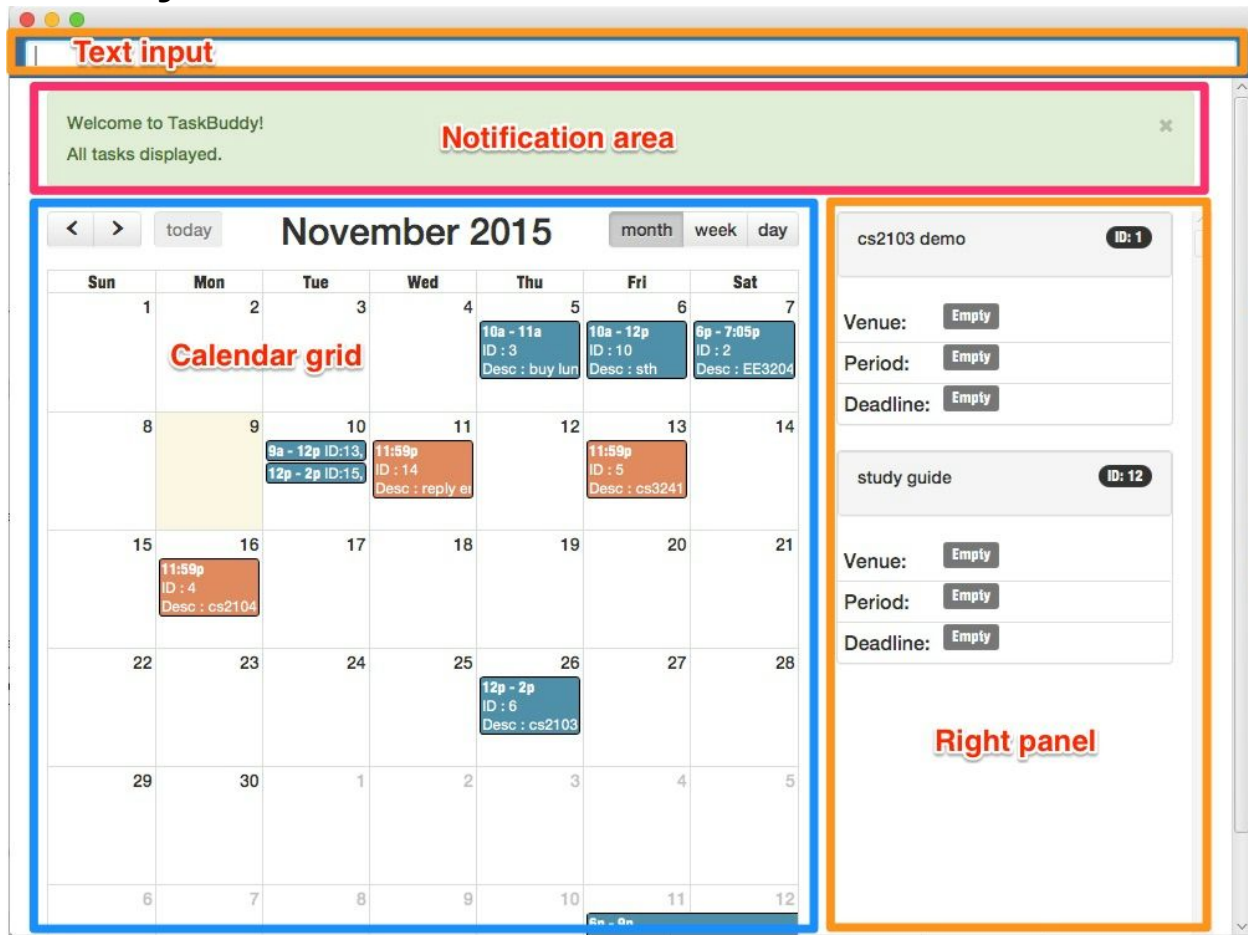
Words : noon, afternoon

Types of tasks supported

- 1) Floating tasks - a task with no specified start and end times, and no deadline
- 2) Event - A task with a specified start and end time and no deadline
- 3) Deadlines - A task with no start and end times, but has a specified deadline.

Note that a task can have both start, end times as well as deadline, although we will leave that up to the user's interpretation on how to use such a feature.

GUI Layout



Initial view

The above screen shows the initial view when user starts the application. The calendar grid shows the month's activities. Events and deadlines are shown on the calendar. Events in blue, deadlines in orange. The header of the calendar shows the current month, while the left and right buttons on the left allows user to traverse forward and backward in time. The 'month', 'week', 'day' buttons on the top right is for user to switch between monthly, weekly and daily view. At the top is the textbox for user input. On the right is a panel that displays the floating tasks by default.

Text input

Type in your commands here.

Notification area

This area contains all the feedback messages from your commands. They fall into 4 categories - errors, warnings, success, help and parameter-related.

Errors occur when there is something seriously wrong with the command or with the program. There will be prompts in the message to help you correct the error.

Warnings arise from actions that the user should not be doing under normal circumstances but the program still allows and execute the instructions anyway. User can type undo if they realize it is a mistake.

Parameter-related messages are more specific messages that pinpoint where errors in the user's command have occurred and offer suggestions to help you get the right command. Help messages contain syntax information for a specific command that the user is trying to execute.

Success messages contain information relating to the successful execution of a command, including things like filepath, taskId and what command was performed.

Calendar grid

The calendar grid shows events and deadlines. It has 3 views - month, week and day. The default is month. The optimal view will be determined by the user input once the user starts using the software.

Right panel

The right panel shows floating tasks by default. When the user runs a search query for tasks, the search results show up in the right panel as well, which may be a combination of events, deadlines and floating tasks. Search queries do not filter out which tasks will be displayed on the calendar, but they affect the default calendar view.

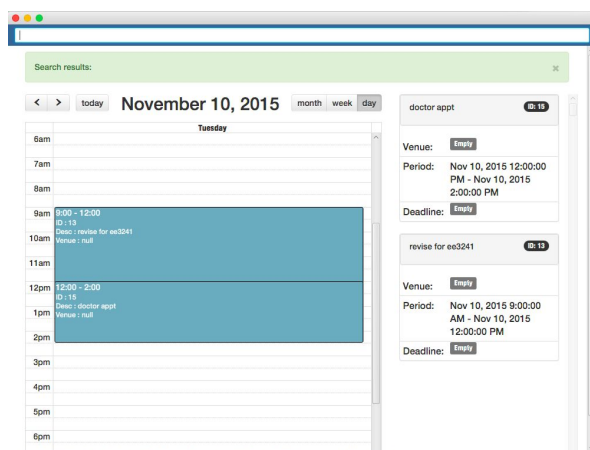
Calendar view optimization

Based on the user input, the program figures out what is the optimal calendar view to show next. For example:

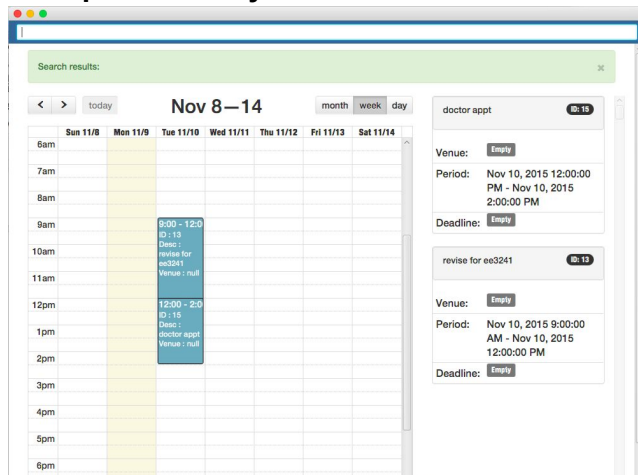
Example input	Calendar view
Display today	daily
Display tomorrow	daily
Display on next week	weekly
Display on next wednesday	weekly

Display on next month	monthly
Display on 3 weeks later	weekly
Display on 3 weeks ago	weekly
Display on 3 months later	monthly
Display on 3 months ago	monthly
Display on next july	monthly

Example of daily view



Example of weekly view



Use Cases

Function : PATH

Description : Change the file path to read and write from. Only change the path, does not read or write from that location.

Command : **path [absolute filepath]**

Returns : Path has been changed to “absolute filepath”, if successful, else an error message

Function : FILEOPEN

Description : Open from the specified filepath.

Command : **fileopen**

Returns : A new display of all the tasks from the new file. Error if file does not exist

Function : FILESAVE

Description : Write to the specified filepath

Command : **filesave**

Returns : Your calendar has been saved to “absolute filepath” if successful, error message otherwise

Function : Add

Description : Add a task to the calendar. Can be a floating task with no specified time, a task with a deadline, or a task with start and end times. Note that if the start time is specified, the default end time will be 1 hour after the start time. If only the end time is specified, the start time will be 1 hour before the end time. These details can always be added later on using the edit command. Minimum information required is a description of the task. There are 2 ways to do this currently, which is detailed below.

1. For adding a task that starts and ends on the same day:

Command : **add** **do** “[description]” **on** [date] **from** [startTime] **to** [endTime] **at** “[venue]”

2. For adding a task that starts and ends on different days:

Command : **add** **do** “[description]” **from** [startDate] [startTime] **to** [endDate] [endTime] **at** “[venue]”

3. For adding a task that has a deadline:

Command : **add** **do** “[description]” **by** [deadline] **at** “[venue]”

Returns : A formatted description of the added task if successful, an example of the correct way to add a task otherwise.

Example for floating task : add do “buy lunch”

buy lunch		ID: 17
Venue:	Empty	
Period:	Empty	
Deadline:	Empty	

Example for event : add do “buy lunch” on 12/10/2015 from 1200 to 1300 at “NUS”

buy lunch		ID: 18
Venue:	NUS	
Period:	Oct 12, 2015 12:00:00 PM - Oct 12, 2015 1:00:00 PM	
Deadline:	Empty	

Example for Deadline : add do “buy lunch” by 12/10/2015 1200 at “NUS”

Function : EDIT

Description : Edit a task in the calendar.

Command : **edit** [task_id] **do** “[description]” **from** [startDate] [startTime] **to** [endDate] [endTime] **by** [deadline] **at** “[venue]”

Returns : A formatted description of the edited task if successful. “Task not found” message if the task was not found. Help message for the command if command is not in the correct format.

Example : edit 19 do “buy dinner”

buy dinner	Past due	ID: 19
Venue:	NUS	
Period:	Empty	
Deadline:	Oct 12, 2015 12:00:00 PM	

Description : Setting one or many field/s to null, using the keywords at, from, to, by. Note that descriptoin cannot be set to null

Command : **edit** [task_id] **no** **at** **from** **to** **by**

Returns : An updated view of the task which was edited.

Function : Changing between floating, event and deadline

Command : edit [task_id] [keyword]

where keyword = floating, event or deadline
Description : Using this command forces a task to be a specific type by setting the period or deadline or both to be null. If keyword is floating, the start, end time and deadline will be set to null. If keyword is event, the deadline will be set to null. If keyword is deadline, the start and end time will be set to null. Note that if the user does not specify a value for the other field, it will be left untouched, which can potentially still be null.

Example : **Edit 2 floating**

Returns : Task 2 with null start and end time and deadline.

Function : DELETE

Description : Delete a task from the calendar using the task id.

Command : **delete [task_id]**

Returns : A formatted description of the deleted task such as "Task id has been deleted" if successful. However if unsuccessful, "The task was not found!" message will be shown.

Example for a valid task id: delete 17

Example for an

Task 17 has been deleted

buy lunch ID: 17

Venue: Empty

Period: Empty

The task was not found!

invalid task id:
delete 25

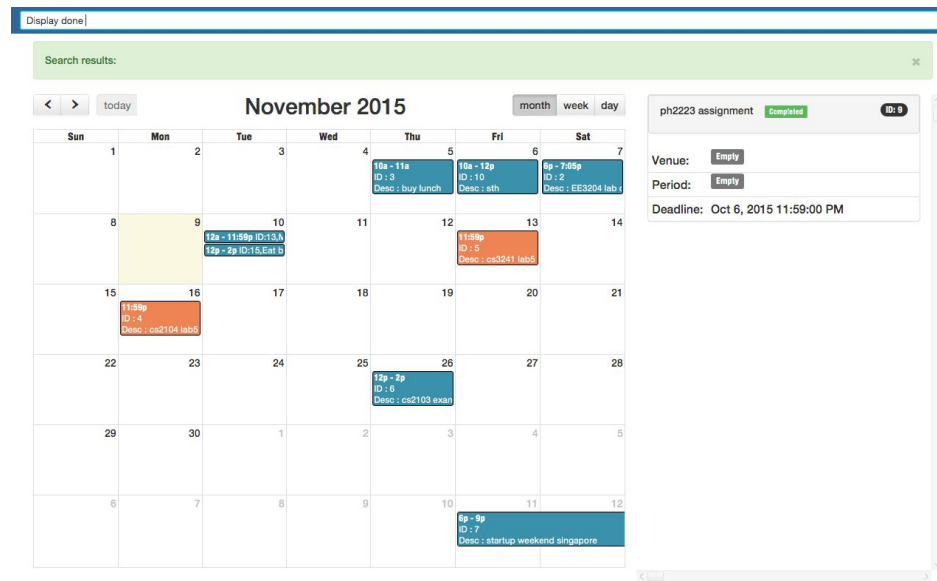
Function : DISPLAY

Description : Display tasks filtered by keywords.

Command : **Display** [task_id] **do** “[description]” **by** [date] [endTime] **at** “[venue]”

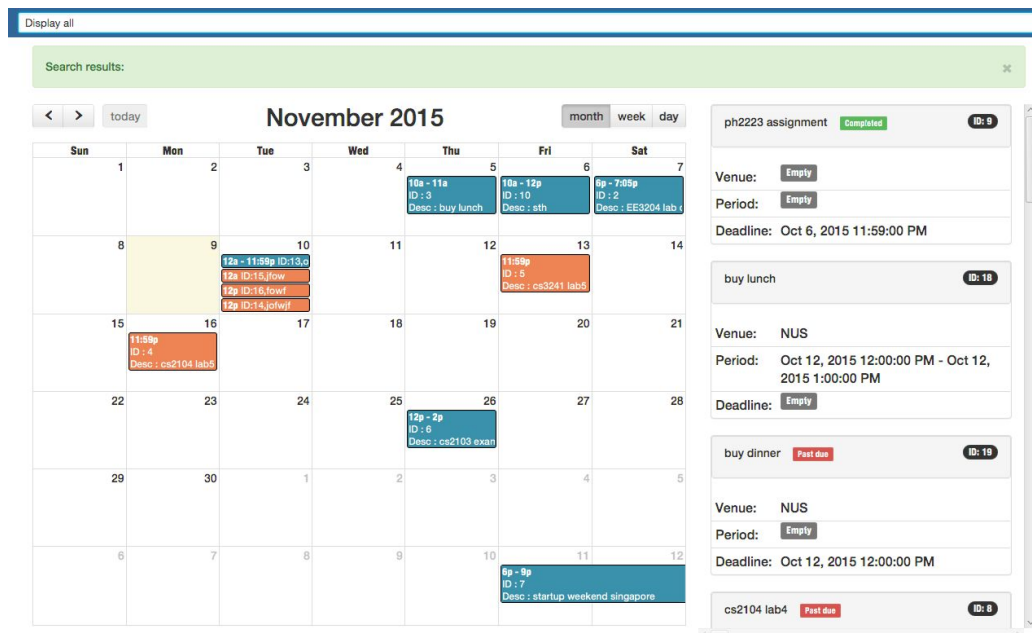
Returns : A list of tasks fitting user criteria. An empty list if none is found. The Display command alone will display all tasks.

Example for showing only completed tasks : display done



Example for showing all tasks: display all

The tasks will be arranged starting with events and floating arranged in descending chronological order going down the page and floating tasks right at the bottom.



Example for showing a specific task: display [task_id]

Display 13

Search results:

< > today

November 2015

month week day

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
				10a - 11a ID : 3 Desc : buy lunch	10a - 12p ID : 10 Desc : sth	9p - 7:55p ID : 2 Desc : EE3204 lab
8	9	10	11	12	13	14
		12a - 11:55p ID:13,14		11:55p ID : 5 Desc : cs3241 lab5		
		12a ID:15_low 12p ID:16_low 12p ID:14_johrf				
15	16	17	18	19	20	21
	11:55p ID : 4 Desc : cs2104 lab5					
22	23	24	25	26	27	28
			12p - 2p ID : 8 Desc : cs2103 exam			
29	30	1	2	3	4	5
6	7	8	9	10	11	12
				9p - 9p ID : 7 Desc : startup weekend singapore		

Meeting with John ID: 13

Venue: Empty

Period: Nov 10, 2015 12:00:00 AM - Nov 10, 2015 11:59:00 PM

Deadline: Empty

Example for showing single day : Display [keyword]
Display on [keyword2]

Possible keyword : today, tomorrow, monday to sunday

Possible keyword2 : All keywords, specific dates(word format or number format)

The calendar will be shown in the day mode if a single day keyword is specified, a week mode if a week is specified, and month mode if a range of longer days are specified.

Display tomorrow

Search results:

< > today

November 10, 2015

month week day

Tuesday

12am	12:00 - 11:59 ID : 13 Desc : Meeting with John Venue : null
1am	
2am	
3am	
4am	
5am	
6am	
7am	
8am	
9am	
10am	
11am	
12pm	12:00 - 2:00 ID : 15 Desc : Eat brunch with Sarah Venue : town
1pm	

Eat brunch with Sarah ID: 15

Venue: town

Period: Nov 10, 2015 12:00:00 PM - Nov 10, 2015 2:00:00 PM

Deadline: Empty

Meeting with John ID: 13

Venue: Empty

Period: Nov 10, 2015 12:00:00 AM - Nov 10, 2015 11:59:00 PM

Deadline: Empty

Function : UNDO

Description : Goes back to the state just before the last **write** action. Examples of write actions are **edit** and **delete**.

Command : **undo**

Returns : A formatted description of the undone task such as “Successfully undone change(s) to Task id” if successful. However if there is no more changes, “No more changes to undo” message will be shown.

Example when there are changes to be undone: undo

Successfully undone change(s) to Task 15.

Example when there is no changes to be undone: undo

No more changes to undo.

Function : REDO

Description : Goes forward a state after an UNDO action

Command : **redo**

Returns : A formatted description of the redone task such as “Successfully redone change(s) to Task id” if successful. However if there is no more changes, “No more changes to redo” message will be shown.

Example when there are changes to be redone : redo

Successfully redone change(s) to Task 15.

Example when there is no change to be redone : redo

No more changes to redo.

Function : EXIT

Description : Exits from the program

Command : **exit**

Example : exit

Parameters

[description]

Description of what the task is supposed to be about. You can type your task in any combinations of alphabets, numbers and symbols. This is the minimal that you have to fill up, otherwise an error message will appear.

[date]

The date for an event that is supposed to start and end on the same day.

[startDate]

Date when an event is supposed to start

[endDate]

Date when an event is supposed to end

[deadline]

Date when a Deadline is supposed to be due. If you want to specify in numbers, please use this format:

- 09/19/2015 1800 (MM/dd/yyyy)

[startTime]

Time when the task is supposed to start. As a time field, it will follow a 24 hour format.

Example, 2359, 1645

[endTime]

Time when the task is supposed to end. As a time field, will follow a 24 hour format only.

Example: 2359, 1645

[venue]

Likewise, you can type your venue in any combinations of alphabets, numbers and symbols.

Appendix A: User stories. As a user, ...

[Likely]

ID	I can ... (i.e. Functionality)	so that I ... (i.e. Value)
readMultipleFormats	read multiple formats of inputs	do not have to worry about how I should type my task
updateTask	add to function to edit task	can edit my task if there's a need to
multipleDisplayFormats	have different types of display options	can view it the way I want my tasks to be. For example: By priority, timing, deadlines
taskDone	mark my task to be done	will know that particular task has been cleared in my scheduler
searchTask	search for a specific task	can find that particular task easily among all the tasks
interpretSentence	make a task with a sentence template	can get functionality from task addition
shortcutKey	open the application with a shortcut	can easily switch between daily tasks and their agenda
fileLocation	select file storage location	can store and move the tasks easily
GUIfancy	visualize tasks	can mentally organize individual tasks in their mind
undoTask	undo my task	can undo the previous action if it was a mistake
deleteTask	delete a task	can remove a task that is completed or unwanted task are taken of
helpUserManual	make use of the user manual	can find out how to use the software when I run into problems

checkFreeSlot	see if there's any available slots for the day	can schedule new tasks or leisure activities
postponeTask	postpone a task	can shift it to another timing/date easily
remindTask	be reminded of the tasks that are undone	can complete tasks within the deadline
prioritizeTask	prioritize tasks	can see which are the most urgent tasks that need to be done
autoSuggestion	get automated suggestion of free time slots	can easily know when to schedule a task among the free time slots
themeCategorization	categorize my tasks according to theme/s	quickly see all tasks related to a theme
notification	be prompted by a notification when I've added/deleted a task	am aware of what has been added/deleted. This is to ensure that the adding or deleting of a task is not by any mistake

[Unlikely]

ID	I can ... (i.e. Functionality)	so that I ... (i.e. Value)
colorTask	add a color to a task	can be aware of some of the tasks which falls under the same category e.g. school
importGoogleCalendar	import a google calendar file	can use my other calendars alongside my task
geoLocation	add a location to my task	know where exactly is the venue for a particular task
syncDevices	sync the tasks to my devices	can be aware of the tasks that need to be completed from my handheld devices instead
popoutWindow	start using my scheduler right after windows startup without having to click on it	will be reminded what are the tasks that needs to be done. I can also add new tasks quickly into my scheduler
sendSchedule	send my weekly/monthly schedule to my friends	am able to connect with my friends easily by finding a common free time slot
overlapTask	be alerted if to-be added task overlaps with another	will not add in more than one of the common task

Appendix B: Non Functional Requirements

Constraint-Desktop:

- The software should work on a desktop without network/Internet connection. It should not be a mobile app or a cloud-based application.

Reason: Desktop apps are more general than other platforms such as mobile/embedded/web and therefore, provides a good starting point before you move to those platforms.

Constraint-CLI:

- Command Line Interface is the primary mode of input. Design the app in a way that you can do stuff faster by typing compared to mouse or key combinations.

Reason: GUIs are quite tightly coupled with the platform. e.g., a Web GUI is quite different from an Android GUI which may be quite different from an iOS GUI and so on.

Constraint-Standalone:

- The software should work stand-alone. It should not be a plug-in to another software.

Constraint-No-Database:

- The software should not use relational databases. Data storage must be done using text files you create yourself.

Reason: Using relational databases reduces the scope for applying Object-oriented techniques.

Constraint-Human-Editable-File:

- The data should be stored locally and should be in a human editable text file. The intention of this constraint is to allow advanced users to manipulate the task list by editing the data file.

Constraint-OO:

- A significant part of the software should follow the Object-oriented paradigm. However, some parts of the application can be non-OO if it can be justified.

Reason: For you to practice OOP that you learned in the course.

Constraint-Windows:

- The software should work on the Windows 7 or Windows Vista OS. It should work on both 32bit and 64bit PCs.

Constraint-No-Installer:

- The software should work without requiring an installer. Having an optional installer is OK as long as the portable (non-installed) version has all the critical functionality

Appendix C: Product survey

Product: Todoist **Documented by:** Audrey Tiah

Strengths:

- Clean GUI, easy to use as it is very user-friendly
- Have shortcut hotkeys to every function, hence navigating using keyboard only is possible
- Allows both categorisation, labelling and prioritising of tasks for different levels of control
- Allows different date format and natural language

Weaknesses:

- Only very basic functions are available
- Scheduling has to be done manually
- Unable to undo a certain action

Product: Google Calendar **Documented by:** Jean Castillo

Strengths:

-Segregation of calendars:

- The ability to use many different calendars at once and display/hide them really brings the organizational part of things into scope.
- Color coded for easy recollection
- Easy import .icx files for use in other websites

-Time period selection

- Makes looking at weekly tasks, day tasks, and monthly tasks a breeze. Helps to set priorities for long term and short term

-Customizable eminders

- The use of multiple reminders (that sync with other devices) helps to keep track of meetings/classes

-Print feature

- For on the go travel without a phone

-Auto sync(google accounts)

- Vital for automatic alerts and keeping an organized schedule

-Visibility

- Advertise to others in the network about your event

-Attachment

- Best for attaching vital components to tasks/meetings

-Find a time

- Looks for available time slots in the current schedule for easy insertion of meetings

-Quick add/Detailed add

- The detailed (or default create) can open a window that has distinct fields for user input. The user can use a quick add feature in order to create events in a faster manner.
- Quick add uses common language and does not give a detailed account for descriptions etc

Weaknesses:

- Too many clicks sometimes
- no easy import functionality (must go to setting etc)
- Prioritizing could use work

Product: Fantastical 2 **Documented by:** Tan Si Kai

Strengths

- Clean looking GUI with sleek transitions
- Integration of calendars across different emails and accounts
- Quick adding of tasks with strong Natural Language Processing (NLP) capabilities allows user to type and add a task in a single sentence

Weaknesses:

- Lack of ability to tag tasks with a certain priority
- Lack of ability to show free time slots in a day
- Lack of ability to categorize tasks
- Lack of ability to search for tasks by category
- Lack of ability to add floating tasks with no start time and end time

TaskBuddy

Developer Guide v0.5

This developer guide aims to help you to gain knowledge about our overall direction and structure of our project. The guide will be presented in a top-down fashion, covering from the overall architecture to the functionality of individual classes. Through this guide, we hope to give you a clearer picture of how to modify the program and make contributions to the project. The project encompasses the task of managing user tasks into an organized and presentable fashion which does not obstruct daily activities.

Architecture

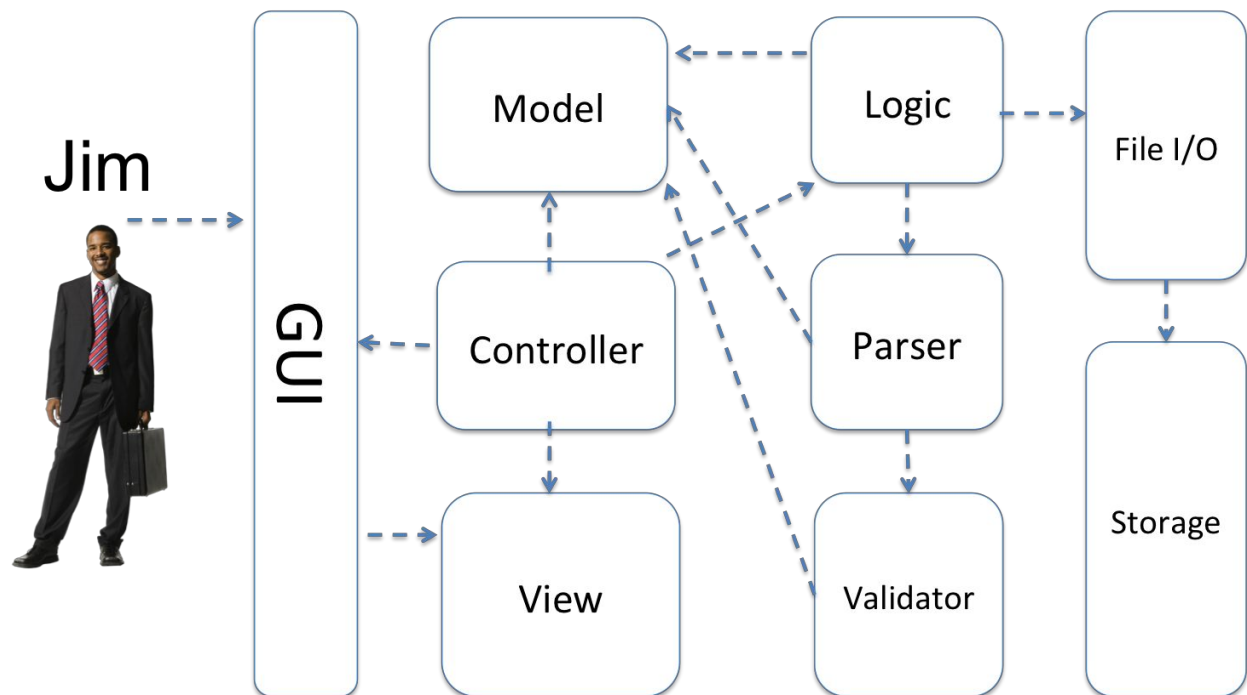


Figure 1: TaskBuddy Architecture. ---> denotes dependency

Design Philosophy

TaskBuddy is currently made up of 9 main components. It applies the Model-View-Controller framework using FreeMarker as a java-to-HTML template rendering engine. The Controller acts as a traffic controller - it acts on user inputs, passes instructions to logic, constructs the data model, render it together with the view before pushing out to the GUI. The advantage of having such a framework is that the data is abstracted from the presentation, and the computations are also separated from both data model and view.

One may notice that the Model component is a dependency for many other components, which may be bad from a design standpoint. However, the Context class in the Model is meant to be a transparent container for data to be placed into and subsequently displayed to the user, thus its public method - particularly `context.displayMessage(fieldName)` is likely to be used over and over again as code execution make its way through all the different components in the system. As user input is parsed, it goes through Logic, Parser, Validator etc. At any point in the sequence of code execution when it is clear that the user has input an invalid command, invalid date format, or on the other hand, successfully executed a command that affects the internal state of data, the programmer may call `context.displayMessage(fieldName)` to set that message for display to the user on the next screen update. This is a convenient way for the programmer to display messages without bothering with Stringbuffers and passing strings around from function to function. Of course, the programmer must first create the message they want to display in the Context object.

The next section goes into each component and their subcomponents in detail.

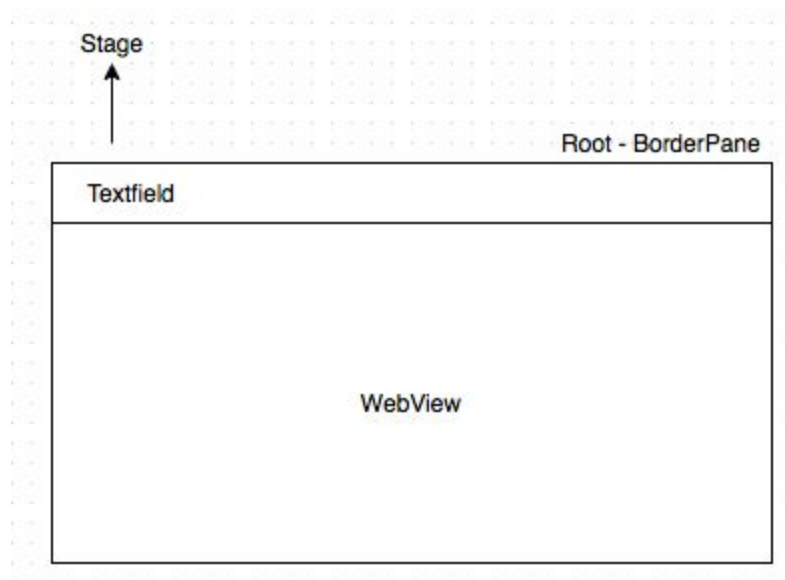
Components

GUI

Classes

JavaFXGUI

The GUI is a JavaFX application that the user interacts with directly. Program main method is in Controller class. On startup, the Controller initializes the rest of the components before launching JavaFX application. The figure below shows the JavaFX component hierarchy. BorderPane is the root node, with Textfield at the top and a WebView at the bottom. The WebView loads a “output.html” file using the WebEngine and reloads the webpage from local filesystem whenever JavaFXGUI.update() method is called.



EventListeners

The Textfield is the primary way for user to interact with the application. An EventHandler<ActionEvent> is registered for the Textfield using the textfield.setOnAction() method.

To ensure that the execution of the event does not block the GUI, we have to invoke the

```
Platform.runLater(new Runnable(){
    @Override
    public void run() {
        // Code here
    }
});
```

method which places the task to be executed on a separate queue which runs in a separate thread from the JavaFX application thread.

Controller

Classes

Controller

Purpose

The controller component is the traffic controller of the entire application. It receives user input from the EventHandler triggered by the “enter” key of the Textfield object in GUI and passes the instruction to Logic which will perform the computations. Once control returns back to the controller, it prepares the actual HTML file to be displayed to user by calling renderView().

RenderView() performs 2 key operations. It gets dataModel from Context class using context.getDataModel(). A dataModel is a HashMap<String, Object> which contains the data that we want to display to the user. Next, get the HTML template from “./templates/html/index.ftl” and render it with the dataModel using

template.process(dataModel, fileWriter) which will then write a new HTML file at “./templates/html/output.html”. JavaFXGUI.update() is subsequently called, which invokes the WebEngine.load() on the output.html file which was just generated. For a pictorial representation of the entire process, please see the sequence diagram.

Model

Classes

Context

Pair

Task

Singleton

The context object is a singleton class with a private constructor. The only way to get an instance of this class is to call Context.getInstance() from outside. This is to make sure there can only be one instance of context in use at any point in time to ensure consistency in user input and the expected display output.

Using the Context to generate dataModel.

The Context class represents the dataModel of the entire application. It contains all the possible error, success, help and warning messages that the user may encounter while using the application. It also contains an ArrayList of Tasks that is to be displayed to the user based on their command. To help keep track of which messages are to be displayed, and if they are to be displayed, can only be displayed once, the Pair object was created. A Pair has a immutable key which contains the message and a boolean value. To set a particular message to display to the user, call context.displayMessage(fieldName), which will set the value of that Pair to true. After all computations are done, a call to getDataModel() will be made. It will run through all fields of the context object and add those messages with value set to true to the dataModel object. This way, the programmer does not have to deal with StringBuffers and worry about multiple messages appearing because of multiple calls to

context.displayMessage(). A call to context.clearMessage(fieldName) can be used to unset a single message, although this is rarely used.

Displaying Tasks

Tasks are added to dataModel object from the displayTaskSet field in Context. To add tasks to the displayTaskSet from elsewhere in the application, call addTask(task).

Clearing context for next display

One can think of the context as a container for data. As the application runs, data is produced and stuffed into the context. Once a call to renderView() has been made, the data is consumed and placed into the template. Thus, there is a need to reset the context by calling clearAllMessages() after that which will empty the displayTaskSet and set all Pair values to false for the next command.

View

Files *external library files are in blue

HTML/FTL

templates/html/output.html

templates/html/index.ftl

CSS

[templates/css/bootstrap.css](#)

templates/css/custom.css

[templates/css/fullcalendar.min.css](#)

Javascript

[templates/js/bootstrap.js](#)

[templates/js/jQuery_v1.11.2.js](#)

[templates/js/fullcalendar.min.js](#)

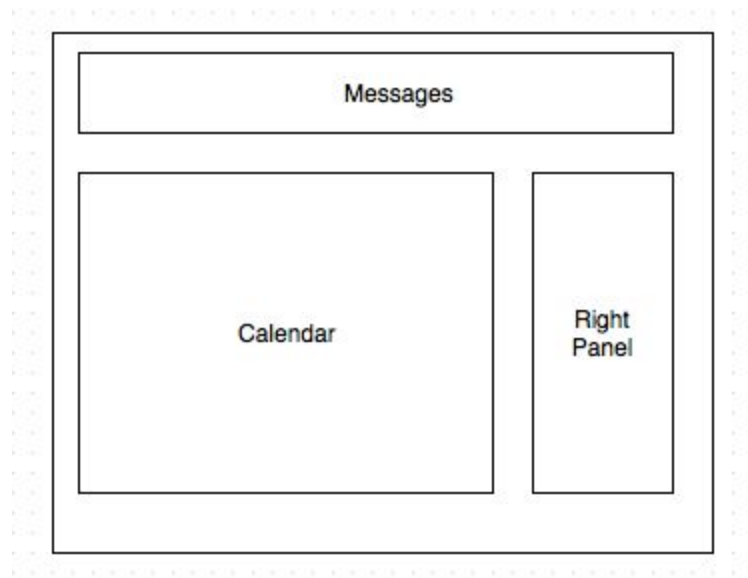
[templates/js/moments.js](#)

templates/js/calendar.js

The main view file is output.html. Index.ftl is a HTML file with FTL markups that places the dataModel into the right places. Styling is achieved through CSS stylesheets. Bootstrap v3 (<http://getbootstrap.com/>) provides the basic framework for styling while customized styles can be layered on with css classes in the custom.css file. Javascript libraries - moments, jQuery and fullCalendar are used to draw the calendar grid.

Web page layout

The figure below shows the basic layout of the template - index.ftl. The calendar grid occupies 70% of the space. Messages are displayed at the top, which can be dismissed by the user using the cross button on the right. The right panel, when the user is not running any search queries, shows only floating tasks. When the user makes a search query through the display command, the right panel contains the results of the query, which can be events, deadlines and/or floating tasks depending on the query. Note that the query does not affect whether tasks are rendered on the calendar grid, it only affects the time window and the calendar view which affects which tasks are displayed on the calendar after a command.



Switching fullCalendar views

FullCalendar has 3 views - monthly (default), agendaDay and agendaWeek. The view is monthly by default and can be changed dynamically according to user input. Validator infers which is the optimal view for the user based on his input and sets the corresponding VIEW_DAY, VIEW_MONTH or VIEW_WEEK field in the context class.

When `renderView()` is called, the view option is inserted into the HTML document (with `display:none`) and on load, `calendar.js` looks for the variable in the document and passes that to `fullCalendar.js` to switch view. Alternatively, the user can simply click on the header buttons to change view at any time.

Setting `defaultDay` on `fullCalendar`

The `defaultDay` determines the range of dates to display on `fullCalendar`. It is today by default. Note that when the user types in “display on 3 weeks later” the `defaultDay` has to be changed to a day within the week 3 weeks from now. This is done by calling `context.setDefaultDate(dateString)` in `TaskHandler`. Note that `dateString` has to be in ISO8601 format which is the format that `moment.js` is able to recognize and parse.

Passing event data to `fullCalendar`

The JSON file is read and converted into a string which is then inserted into a hidden HTML element during `renderView()`. On load, `calendar.js` retrieves that data and parses into javascript event objects which are passed to `fullCalendar.js` to render the calendar grid.

Logic

Classes

`TaskHandler`

`UndoableSignificantEdit`

`TaskUndoManager`

`TaskEdit`

`TaskListEdit`

`TaskPeriodEdit`

`TaskDescEdit`

`TaskVenueEdit`

`TaskDoneEdit`

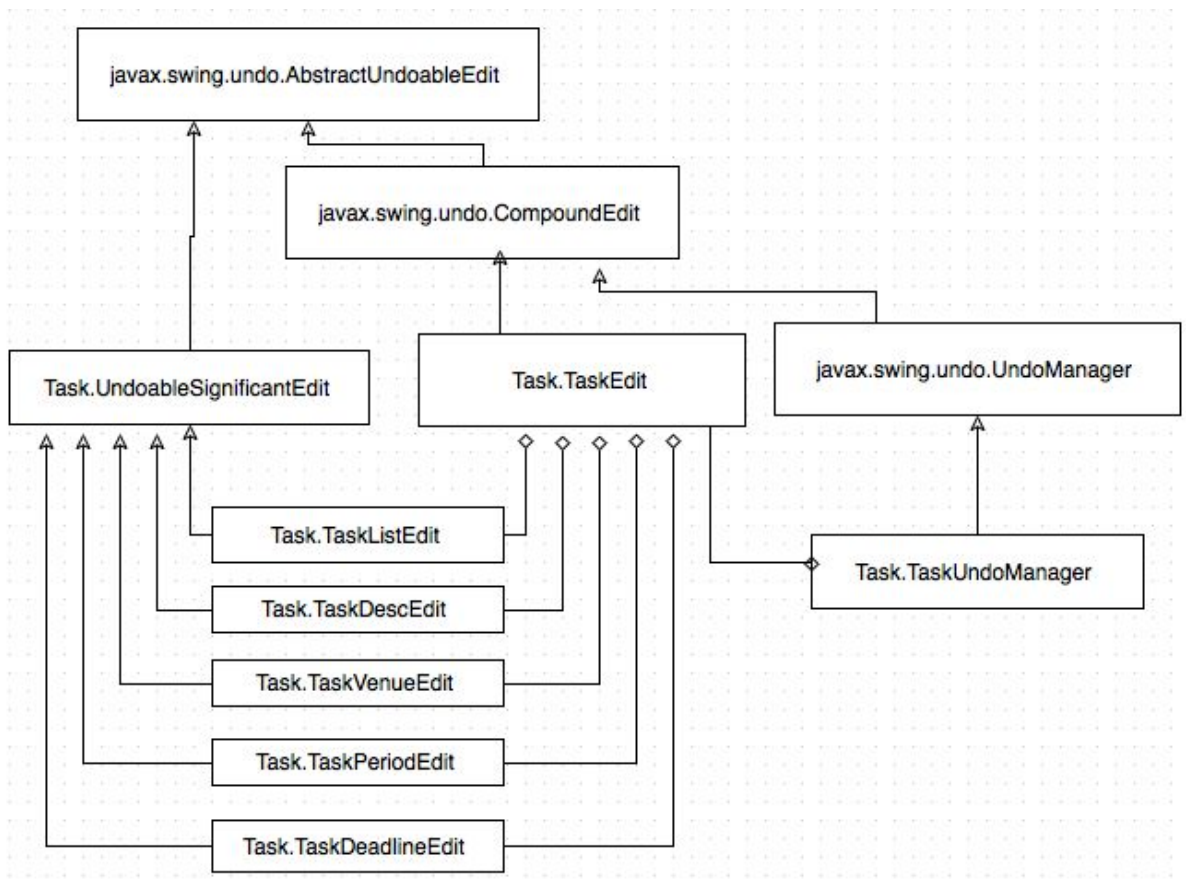
`TaskDeadlineEdit`

Logic is the brains behind the application. Once the user intent is correctly identified through the command and HashMap of parameters returned from Parser, executeCommand(userInput) takes the user input and performs the necessary operations on the tasks.

TaskUndoManager

The TaskUndoManager extends java's UndoManager, which together with TaskEdit, and UndoableSignificantEdit, helps to keep track of changes to Task objects and the taskList so that undo and redo operations are supported. The figure below shows the class hierarchy between the edit classes.

Class hierarchy diagram for TaskUndoManager



EditToBeUndone() and **EditToBeRedone()** overrides the protected superclass method and is invoked just before the undo() and redo() method is called in order to preserve information about the task that was edited for display to user.

TaskEdit is a compoundEdit which contains one or many UndoableSignificantEdit. A single user command may result in one or more UndoableSignificantEdits. These edits are added to a single TaskEdit until the very end, where the last edit is set to be significant, before calling end() on a taskEdit. Finally, the taskEdit is added to the TaskUndoManager. For detailed information on how UndoManager works, especially on UndoableEdits and CompoundEdits, please refer to the java documents (<https://docs.oracle.com/javase/7/docs/api/javax/swing/undo/UndoManager.html>).

Parser

Classes

StringParser

Parser component receives a string of user input and binds specific phrases to pre-determined parameters based on the agreed input format (see user guide). It then validates these arguments by calling the Validator, which will try to parse these phrases into Date objects bound to the same parameters. Returns a HashMap<PARAMETER, Object> to Logic which will continue execution.

List of parameters

DESC - description of task

VENUE - location of task

DATE - date following “on” keyword in user input

START_DATE - start date of event

END_DATE - end date of event

START_TIME - start time of event

END_TIME - end time of event

DEADLINE_DATE - deadline date of a task with deadline

DEADLINE_TIME - deadline time of a task with deadline

TASKID - an internally assigned unique integer to represent this task

IS_DONE - boolean field to indicate whether the task has been completed

IS_PAST - boolean field to indicate whether a deadline is overdue
HAS_ENDED - boolean field to indicate whether an event has ended
DELETE_PARAMS - array of parameters to set to null
SPECIAL - miscellaneous information we want to capture from user input like display all, done, undone

Validator

Classes

Validator

Validator component contains various validation methods to verify that the right input and format was received from the user. It receives phrases placed in parameters from the StringParser and go through each of them. For date related parameters, the Validator converts the phrases into Date objects with the help of Natty - a Natural Language Processing date parsing library (<http://natty.joestelmach.com/>) and bound them to the correct parameters. For phrases that specify a range of time, the Validator infers the start and end date and times. For example, for a phrase like 'on this week', the start date and end date will be inferred as the first and last day of the current week respectively. Default start times and end times of 1200am and 1159pm are also given to inputs that do not specify time.

FileIO

Classes

FileIO

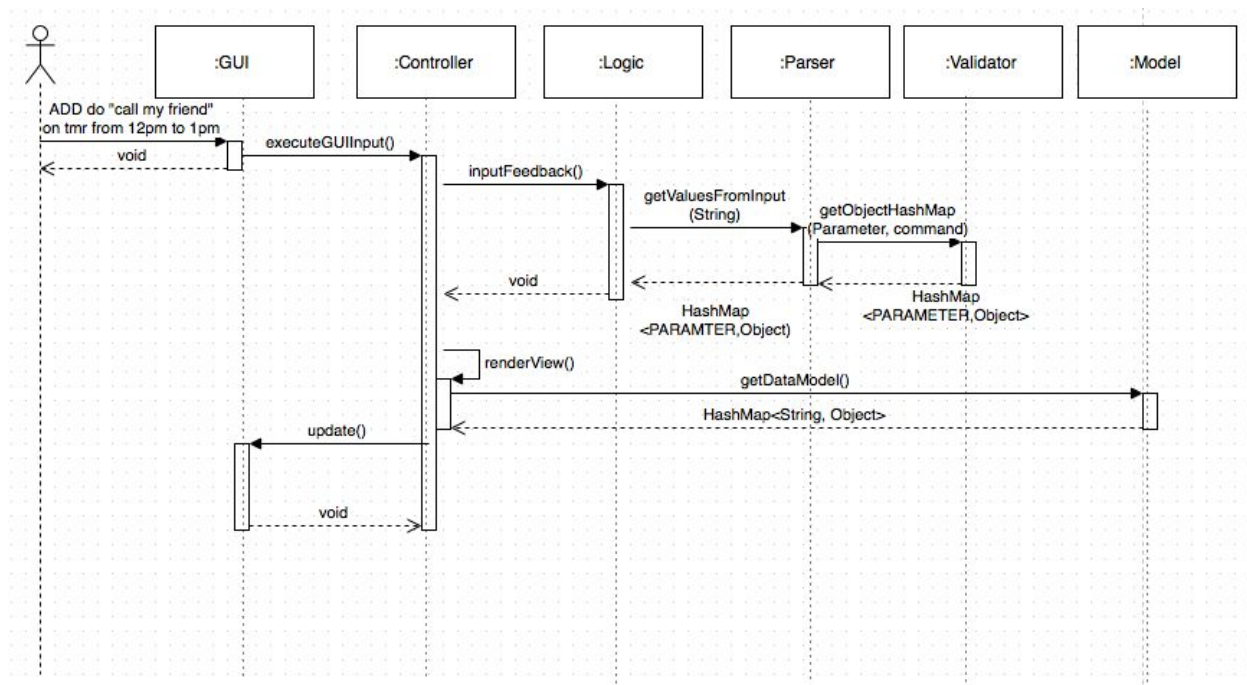
FileIO component is responsible for reading from and writing to a JSON file. By default, the location is ./data/calendar.json (relative to project root). The file location can be changed by passing in an absolute or relative path when starting the program from the command line, or by using PATH command in the GUI.

Sequence Diagram

The following sequence diagram illustrates the interaction between each component when the user uses the program to add a task.

Example: Add do "call my friend" on tmr from 12pm to 1pm

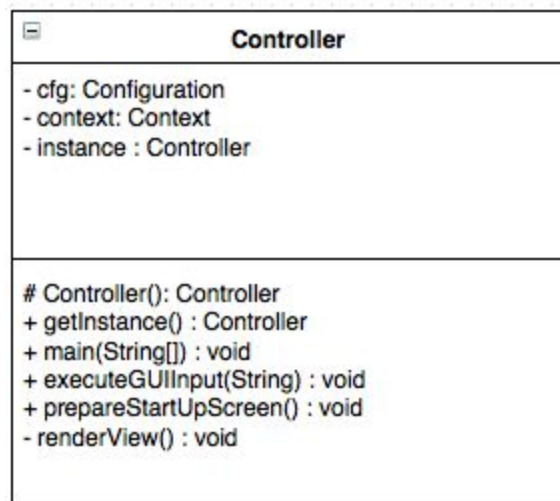
Note that the first call is the handle() function of the EventHandler<ActionEvent> of the Textfield. It is asynchronous, meaning that it spins off a new thread which does the computations before returning immediately after that.



Class Diagrams

Note that only certain important classes are shown here, and for the classes that are shown, only the non-trivial fields and methods are documented.

Controller Class

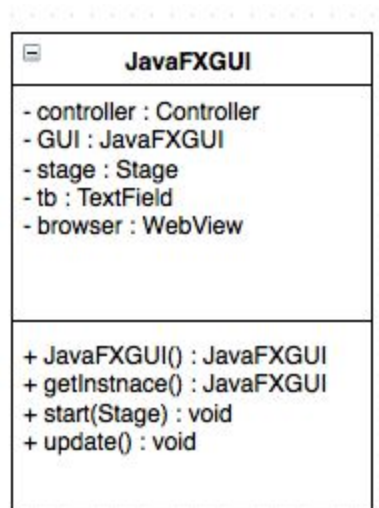


Notable APIs

Return type	Method and Description
Controller	<code>getInstance() : Controller</code> is designed to be a singleton. <code>getInstance()</code> returns the only instance of Controller class.
void	<code>main(String[] args) : void</code> : Main method of the entire application. User can launch from command line with an absolute filepath specifying location of their calendar JSON save file.
void	<code>executeGUIInput(String input) : void</code> : takes user input from Textfield in GUI and passes it to Logic.

void	prepareStartupScreen() : After JavaFXGUI launches, this method is called to initialize the first screen the user will see.
void	renderView() : Takes a dataModel from Context, and FTL template file and renders a HTML file which JavaFXGUI's WebView will load and display to the user.

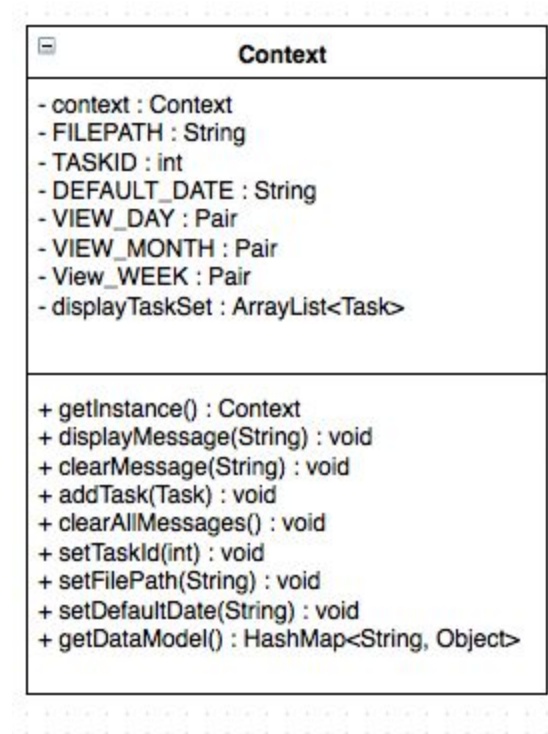
JavaFXGUI Class



Notable APIs

Return type	Method and Description
JavaFXGUI	JavaFXGUI() - the constructor for this class must be public in order for Application.launch() to be called successfully from Controller.
void	start(Stage stage) - As JavaFXGUI extends Application, it must implement and override parent start() method which takes in a javafx.stage.Stage and initializes all the components of the GUI. Finally, render the window by calling stage.show().
void	update() - this function is called by the Controller once all computations are done to update the result of the user command. In this function, the WebView's WebEngine reloads the HTML file.

Context Class

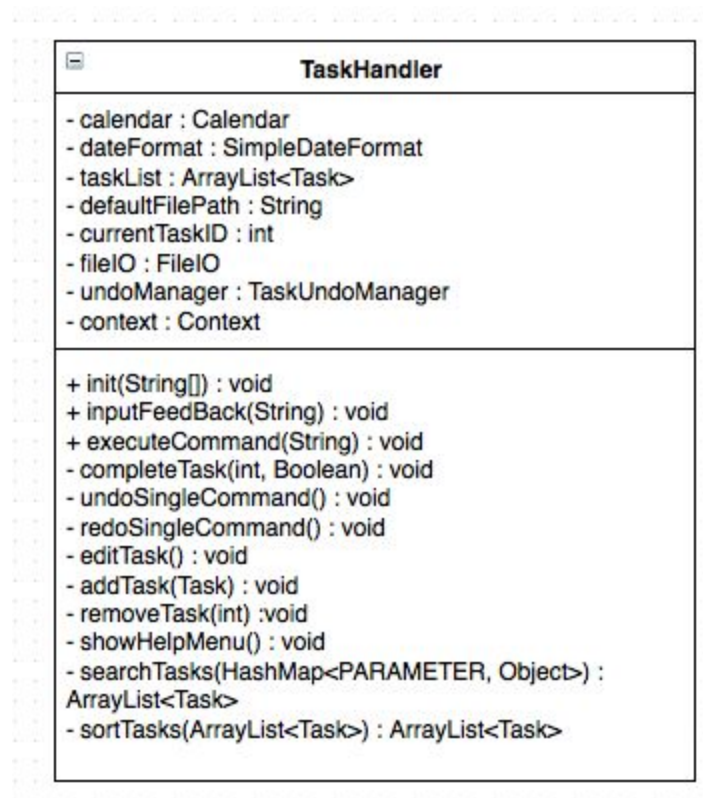


Notable APIs

Return type	Method and Description
Context	getInstance() - Context is designed to be a Singleton class. Call this function and assign it to a private field in classes where you will be needing to set data for display in Context.
void	displayMessage(String fieldName) - call this anywhere in the code where the execution of user input has reached a definitive conclusion about the state of the data or their query. For example, it may be called in the catch block of a ParseException while parsing a string into Date object meant for the startDate parameter, so we can call context.displayMessage("ERROR_INVALID_DATEFORMAT") to display that message in the next screen update. Note that the input to this function reflects fieldNames in the Context class which must match exactly.

void	addTask(Task task) : call this function outside Context when there are tasks you would like to display to user in the right panel. Rendering order goes from 0th element of the displayTaskSet ArrayList to the length-1 element.
void	getDataModel() : at the end of computation, call this to run through all the fields in context and place all data required to be displayed into a HashMap<String, Object>, known as the dataModel. FreeMarker will subsequently place the data from dataModel into the specified places in index.ftl.
void	clearAllMessages() : Resets all messages in the Context object and empties displayTaskSet. Call this after a call to renderView() in Controller.

TaskHandler Class



On initialization, TaskHandler takes a user-defined storage location or “./data/calendar.json” by default. If the file does not exist, FileIO will create an empty file automatically. If the file exists, it reads the file and load it into memory.

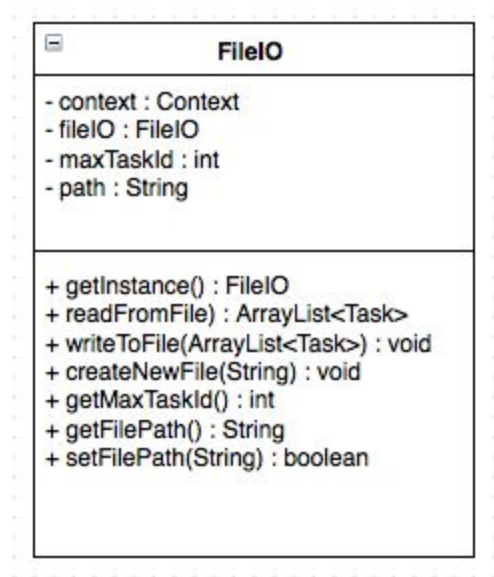
The TaskHandler class receives user input from GUI and passes it to PARSER to break down into PARAMETERS which can be validated by VALIDATOR and then subsequently executed by executeCommand(). All commands that change data and have been successfully executed, will be written to disk immediately.

Notable APIs

Return type	Method and Description
String	executeCommand(String userInput) : Takes a string and executes based on user intention
Task	searchTasks(int taskId) : Takes the task ID, searches through taskArray and returns the task with the specified ID, null otherwise.
void	completeTask(int taskId, Boolean isDone) : given a taskId, mark its isDone field as true or false
void	undoSingleCommand() : undoes the last user command that was executed successfully.
void	redoSingleCommand() : redoes the last user command that was undone successfully.
void	editTask() : Takes a user input and edits the user specified fields of a single task.
void	addTask(Task task) : adds a single task to the taskList
void	removeTask(int taskId) : removes task with taskId from the taskList.
void	showHelpMenu() : displays the help menu to user

ArrayList<Task> >	searchTasks(HashMap<PARAMETER,Object> param) : takes a parsed parameter table of search query parameters and looks for tasks in taskList satisfying those criteria.
ArrayList<Task> >	sortTasks(ArrayList<Tasks>) :

FileIO Class



FileIO reads from the specified file location and returns an ArrayList of tasks. It also writes a given ArrayList of tasks into the same file location.

When reading and instantiating tasks into memory, it also keeps track of the largest TaskID encountered so far. This information is exposed to Logic through the `getMaxTaskId()` method.

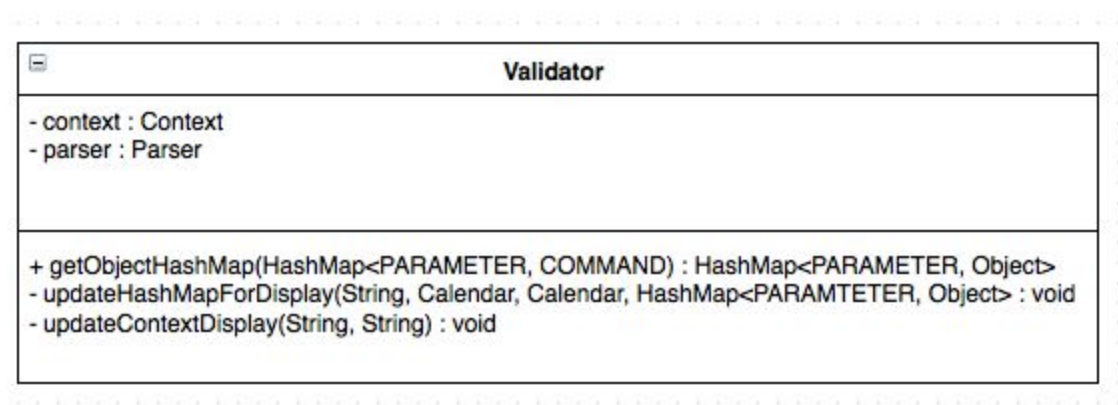
The underlying implementation uses GSON library. The jar file is contained in “/lib/gson-2.3.1.jar”

Notable APIs

Return type	Method and Description
-------------	------------------------

ArrayList<Task>	readFromFile() : reads a JSON file and instantiates tasks in memory
void	writeToFile(ArrayList<Task> : writes the list of tasks to file in JSON format
void	createNewFile() : creates a new JSON file at filepath PATH with “Tasks” key that equates to an empty array.
int	getMaxTaskId() : returns the current maximum taskId encountered while reading the JSON file. This is called after a call to readFromFile() to obtain the next taskId to assign to new tasks added by the user.
boolean	setFilePath(String path) : Takes an absolute filepath and validates it first, returning false if it is an invalid location, changing the class’s path variable and returning true otherwise.
String	getFilePath() : Returns the current filepath in use by fileIO.

Validator Class



The Validator class takes in a hashmap containing the inputs in their respective parameters from the Parser class. It then evaluates each of the inputs and returns another hashmap containing its respective objects.

This class plays the important role of validating all user inputs and at the same time convert the inputs into their respective type. For example, inputs that are valid in the

date and time parameters are converted into Date objects and placed in the hashmap. The Validator also determines the starting and ending dates and times when needed.

Determining Dates and Times:

For event Dates, starting time is default to be 1200am and end time 1159pm if times are not specified.

Examples:

'on monday to friday' : Monday(date) 12am to Friday(date) 1159pm

'on monday' : Monday(date) 12am to Monday(date) 1159pm

For deadline Dates, end time is default to be 1159pm if times are not specified.

Examples:

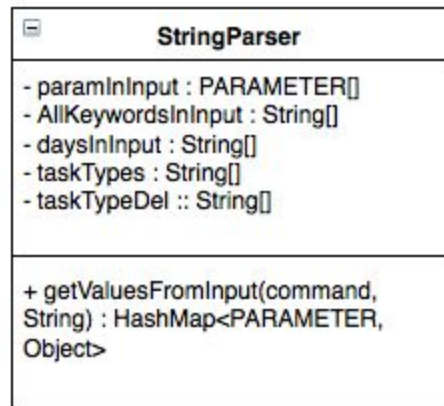
'by friday' : Friday(date) 1159pm

'by 11/21/2015' : 11/21/2015 1159pm

Notable APIs

Return type	Method and Description
hashmap	getObjectHashMap(HashMap<PARAMETER, String> hashmap , COMMAND_TYPE command) : Takes in a hashmap that contains all the parsed inputs in their respective parameters. It then validates the inputs and converts them to their respective objects, before storing them into a hashmap and return it.
void	updateContextDisplay(String deadline , String keyDate) : Determines the appropriate calendar view for the user, given the query strings. There are 3 different views: the daily view, weekly view and monthly view.

Parser Class



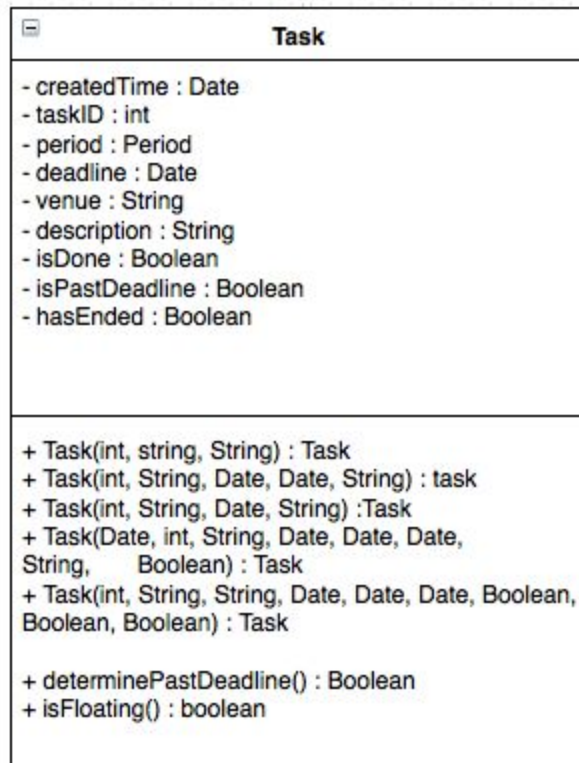
The Parser class takes in a string containing the user input from the Logic class. A new `HashMap<PARAMETER, String>` is created which will be returned after appropriate handling. The parser then evaluates the String starting with any parameters that require parenthesis; parenthesis are to be removed as they may contain keywords that may otherwise confuse the parser. Thereafter it moves to keyword matching with the correct number of parameters associated with each word. Shortcut keywords, that is keywords such as “on” that eliminate the repetitive use of a parameter for another keyword, must change the parameterInput before evaluation. This parameterInput determines the amount of parameters each keyword will look for. After the string has been parsed it is validated through the validator in order to ensure the integrity of data. The resulting hashmap is then returned.

This class plays the important role of analyzing the order of all user inputs. As per the testing many different kinds of inputs are considered and therefore a traversal method of analyzing the input string is used.

Notable APIs

Return type	Method and Description
hashmap	<code>getValuesFromInput(COMMAND_TYPE command, String userInput)</code> : Takes in a String and the command it is related to so that it may be traversed properly.

Task Class



Notable APIs

Return type	Method and Description
Task	Task (int <code>taskId</code> , String <code>desc</code> , String <code>venue</code>) : Constructor for tasks without <code>startTime</code> , <code>endTime</code> and <code>deadline</code>
Task	Task (int <code>taskId</code> , String <code>desc</code> , Date <code>startTime</code> , Date <code>endTime</code> , String <code>venue</code>) : Constructor for tasks with <code>startTime</code> and <code>endTime</code>
Task	Task (int <code>taskId</code> , String <code>desc</code> , Date <code>deadline</code> , String <code>venue</code>) : Constructor for tasks with <code>deadline</code>

Task	Task (Date <code>createdTime</code> , int <code>taskId</code> , String <code>desc</code> , Date <code>startTime</code> , Date <code>endTime</code> , Date <code>deadline</code> , String <code>venue</code> , Boolean <code>isDone</code>) : Constructor for tasks with <code>startTime</code> , <code>endTime</code> and/or <code>deadline</code> . Used when reading from JSON file.
Task	Task(int <code>taskId</code> , String <code>desc</code> , String <code>venue</code> , Date <code>startTime</code> , Date <code>endTime</code> , Date <code>deadline</code> , Boolean <code>isDone</code> , Boolean <code>isPastDeadline</code> , Boolean <code>hasEnded</code>) : Constructor used for temporary tasks.
Boolean	<code>determinePastDeadline()</code> : Returns true if the deadline of the task is before current time, false otherwise. Returns null if the task is not a deadline.
boolean	<code>isFloating()</code> : Returns true if a task has no <code>startTime</code> , <code>endTime</code> and <code>deadline</code>

Testing

TaskBuddy uses Junit 4 testing framework. Test classes are placed in test folder. Test files for FileIO class are placed in test/data/test*.json.

Only public methods are tested. Every class undergoes rigorous unit testing to ensure quality and minimize bugs.

Classes involved in Unit Testing

TaskEdit, StringParser, Validator

Classes involved in Integrated Testing

TaskHandler