Software Design Specification

Study Buddy

Revision 0.1 Beta

Table of Contents

Revision History

Version	Name	Reason For Changes	Date
0.1	*DNM*	*Initial Revision*	*30/10/2022*

Approved By

Name	Signature	Department	Date
David Botero	*David Botero*	Test Lead, UI Co-lead	Oct 28/22
Joshua Lade	*Joshua Lade*	UI Lead, Test Co-lead	Oct 28/22
Hayden Lister	*Hayden Lister*	Contact Person, Project Management	Oct 28/22
Abhay Parashar	*Abhay Parashar*	Version Control Lead, Project Management Co-lead, Dev-lead	Oct 28/22
Robert Stewart	*Robert Stewart*	Documentation, Dev Co-lead	Oct 28/22

1. Introduction

1.1 Purpose

This app will function as a task management system for post-secondary students. Students are able to create different *tasks* (refer to 1.4 definitions), then relate them to a defined *type* of work process, then place those *tasks* within a course *group* identifier. Each *task* is given a due date, and can be marked as completed once it is done. Students can use this web app to see a visual display of when their assignments, tests, and other school-related activities are due, helping them to prioritize their workflow.

1.2 System Overview

Tasks (refer to 1.4 definitions) will be stored within a database that is able to be sorted and edited by the user. Each user will have access to a common NoSQL database, where all JSON files are stored. User authentication will only allow them to access JSON files they have created. The application to do this is a web app that will be accessed by a client computer. Queries and Input will be done via their web browser. Users are able to enter data for *task* names, due dates, *group* names, and *group* colour. Task *types* are defined within the app as hard-coded values.

1.3 Design Map

The System uses GitHub Pages (refer to 1.4) as a central hub of the web app. GitHub Pages hosts two files: an HTML file and a JavaScript file. These files contain all the UI elements for the website, the structure of it, and functionality of the website. Connected to this is an external database stored in Firebase, which is a Google cloud based backend database. This database handles all additions, edits and queries. Another node that is connected to the central GitHub Pages hub is a read-only library function for CSS elements for the front end to use, these are stored on Bootstrap's server. Lastly, the client connects to the GitHub Pages server via their web browser. The user inputs data, and the system presents the data in an organized form, per entry. Their web browser does most of the work using our JavaScript and HTML code.

1.4 Definitions and Acronyms

Task – A specific action or activity that needs to be accomplished by a specific time.
Type – A particular kind of action that needs to be accomplished by a *task*.

Hard-coded as: Assignment, Test, Projects, Labs

Group – A category that groups different *tasks* together who share a common goal. (e.g. a course)

Firebase - A Google owned cloud database and backend service

Bootstrap - A web-based Framework for CSS design using HTML elements

GitHub Pages - An Azure hosted web hosting service, this replaces conventional Linux items used for website hosting, like Apache/Nginx and a DNS.

***MVP* -** Minimum Viable Product, the minimum features required from a product or application to meet the needs of the client

HTML 5 - The Latest version of HTML as of October 2022

***ES6* -** The latest version of JavaScript as of October 2022

2. Design Considerations

2.1 Assumptions

We assume the user has access to the internet because, as an internet-hosted app, they will not be able to reach and utilize it. In our initial configuration, we assume the user has a GitHub account. This is required for user authentication, so if the user does not have a GitHub account, they may not use the app until Authentication is added for other services. We assume the user is using a modern web browser that supports HTML 5, Bootstrap 5, and ES6, thus web browsers such as Internet Explorer 6 will not be supported.

2.2 Constraints

Users need to have a GitHub account to access the service, as it will allow us to securely provide them data. As it is a public facing service, we will require security for our database, and because the storage is handled by Firebase, we will be reliant on their implementation of their user authentication for security. Users with visual impairments may find the colour scheme difficult to use, and without major accessibility improvements the website will continue to have this setback. Functional constraints include limited colour choices, not displaying with a distinct colour on the *tasks* (refer to 1.4 definitions) that have elapsed past their due date.

In addition to these, the *MVP* will place several constraints on the user's interaction with the application. This version of the application will only offer sorting by a *task's* due date, and not by any other parameters. The user will also not be able to review the completed items, as they are hidden when they are marked as complete. The user will not be able to remove *group*s from the list of established *groups*, or be able to add new *types*. Additionally, the system is currently not supporting the deletion of *tasks*.

2.3 System Environment

The system relies on several separate web services. To run the web server itself, the web app will run on GitHub pages. GitHub pages is an Azure web service that allows simple websites to be hosted without need for any configuration. The Backend services include user authentication, storing, and retrieval of the data from the database. These services are completed through Firebase, which is a Google Cloud framework that provides fast and secure services for our application. Client-side, the service is accessed via a web browser. Additional dependencies are the frontend framework libraries stored on external websites. The client machine must support a modern browser that is capable of supporting HTML 5, Bootstrap version 5, and ES6.

2.4 Design Methodology

When considering our *MVP* (refer to 1.4 definitions), we have collected the features that students need to organize their academic *tasks* effectively. This list of core features contains those features which are crucial for our product to fulfill this goal. The goal of the extended and wish list features is to provide additional functionality to the product. However, the implementation of these additional features may take longer than our projected production date is forecast for. If we are able to complete the *MVP* before the expected date, we will reassess adding these features.

The core features of this app are as follows:

1. Adding a *task* with subfields for descriptions, the *group* it belongs to, the due date, and *type* of activity.
2. Organizing the *tasks* by their due dates.
3. Adding new *groups*.
4. Marking a *task* as completed
5. Legend that shows information about the existing *groups* and the colour associated with the *group*.
6. User Authentication that allows the user to access the system by logging into their GitHub account. This will improve the overall security of the system and protect user data.

Our extended goals are as follows:

1. A help menu.
2. The ability to add visual spaces between items that are due in different weeks.
3. Visual flags for *tasks* not completed when past their due date,
4. Sorting by *groups/types*.
5. The ability to change the colour of a group.
6. Ability to see completed *tasks* in a different section
7. Ability to delete tasks.

Lastly, our wish list is as follows:

1. Editing existing *task*(s).
2. Integrating a third party service like google calendar.

When considering our production deadline, additional features will be added from fastest to slowest to implement. This means some wishlist features may be included before extended features because of the client’s time constraints.

2.5 Risks and Volatile Areas

Scaling will be challenging as we are relying on external web services to support the product. To keep the web app free, we are using the lowest tier of these products. This offers free services which have several limitations. These include bandwidth, data quota, and other specific as detailed within the end user licence agreements of these products. Because of this, the user base must be limited without access to some form of revenue stream. Without a revenue stream, the cost of hosting the product will not be sustainable beyond a small number of users.

In addition to this, while Azure and Google do not often experience downtime, in the event that a service interruption were to occur, then users would not be able to access the product. Lastly, as our system is dependent on Bootstrap, any interruption to that service would leave our application without a front end, also rendering the app unusable. Because of these dependencies, uptime of the application is not guaranteed. For the security of the database, the system is reliant on Google cloud services. Additionally, policies on the external services that the system is using may change at any time, which could conceivably interfere with the operation of the application.

Furthermore, the *MVP* implementation of this application imposes restrictions which may interfere with the user experience. The user might unintentionally mark a *task* as completed, and not have the ability to fix this mistake. Additionally, as the system does not support editing *tasks* or *groups*, the user might experience difficulties in organizing their *tasks* if they make input errors. Because of the time constraints on the delivery date of the program, these limitations may not be able to be removed by the end date.

3. Architecture

3.1 Overview

The high level architecture of the product consists of four fundamental components;

- The client, who uses a web browser to access the web app.
- GitHub, our web hosting and DNS service. Our web app will consist of two files; an HTML file that serves as a framework for the UI, and holds the script, which will be JavaScript, that will provide functionality to the website (which uses Azure Cloud services).
- Firebase, our NoSQL database and backend (which uses the Google Cloud service) and,
- Bootstrap, a framework for CSS. (refer to 1.4 definitions)

Please refer to diagram 3.1.1

3.2 Subsystem and Components

Classes

Classes are specified in the "index.html" file and their implementation is done in the "firebase.js" file, which makes use of various methods provided by the firebase and firestore libraries.

Implementation

User:
The user authentication functionality is implemented with a firebase provided method which authenticates the user's credentials and ID, within the GitHub pop-up window.

Data fields:
Name (string)
Unique ID (int)

Methods:
User Details Method: Taken from a firebase library.

Task:
The task is an object provided by a firebase library, which is used to implement the event where the user enters a task, using the functionality that firebase provides such as drop-down menus.

Data fields:
Name (string)
Group Name (value parsed from group object)
Type Name (value parsed from type object)
Due Date (int UNIX time)
Date Edited (int UNIX time)
Date Completed (int UNIX time)
Completed (Boolean Value)
User (value parsed From User Object)

Methods:
Create Task Name Method: Takes Input value from user, takes comparison of other existing Task Objects for name field to ensure that two identical values don't exist. Basic input rejection is performed

Choose Group name Method:
Calls Get Group Name sub-method: Queries Database for existing group objects. After the sub-method is called, a list of group names is given for the user to choose from, the selected name is then written into the field.

Choose Type name Method:
calls get type name sub-method that reads the contents of the JavaScript dictionary full of type names. The user selects one of the types pulled from the sub-method and the selection is input into the field.

Date Added:
Takes a timestamp of when the data fields were entered into the backend service, and takes that Unix time value and inputs it into the field, this is only performed when an object is first added.

Date Edited:
Takes a timestamp of when any data fields are entered into the backend service, and takes that Unix time value and inputs it into the field, this is updated every time any value within the object is changed.

Input Date Due:
Takes a User input value selected from a Front-End Calendar element, error checking is performed to ensure the time is a future value. This value is converted to a Unit time value in the field.

Completed:
A Boolean value taken from user input. Whenever the value is changed, the boolean value within the object is changed from false to true.

User:
Values pulled from User class that is created by the User Authentication Service, this value exists to provide security to objects and labels which objects can be accessed by which users. In our system, only objects created by a particular user can be viewed by that particular user. E.g. The tasks created by Bob can only be viewed by Bob.

Mark Task:
Quarries database for object named from input (which gives value of existing task object, find object and edits it's marked boolean value to true.

Group JSON Object

Data fields:
Colour (int)
Name (string)
Date Added (int UNIX time)
Date Due (int UNIX time)
Dated Edited (int UNIX time)
User that edited it (string)

Methods:
Create Group Name Method: Takes input value from user, takes comparison of other existing Group Objects for name field to ensure that two identical values don't exist, counts number of group objects and will only create name group object if less than 15 exist. Basic input rejection is performed on the test as well.

Choose Colour Method:
Takes Input value from user as an 8-bit integer taken fro front end calendar object, takes comparison of other existing Group Objects for colour field to ensure that two identical values don't exist, counts number of group objects and will only create name group object if less than 15 exist.

Date Added:
Takes a timestamp of when the data fields were entered into the backend service, and takes that Unix time value and inputs it into the field, this is only performed when an object is first added.

Date Edited:
Takes a timestamp of when any data fields are entered into the backend service, and takes that Unix time value and inputs it into the field, this is updated every time any value within the object is changed.

User:
Values pulled from the user class that is created by the User Authentication Service, this value exists to provide security to objects and labels which objects can be accessed by which users. In our system, only objects created by a particular user can be viewed by that particular user. E.g. The tasks created by Bob can only be viewed by Bob.

Type JavaScript Object (Static)

Data fields:
Name (String)

Methods:
Type Name Method: Pulled from a static dictionary of objects within the JavaScript file, these are: Assignment, Exam, Projects, and Lab.

Add Task Method:
Opens up the add task pop up (Appendix B 7.1.3.4).
Calls Create Task Name Method
Calls Select Group Name Method
Calls Choose Type Named Method
Calls Data Added Method
Calls Data Edited Method
Calls Date Due Method
For error checking, it checks if the inputs are empty, and throws an error pop (Appendix B 7.1.E Input Error). Pressing cancel closes the pop-up. If adding was successful, a successful add pop up will display at the bottom of the page (Appendix B 7.1.S).

Add Group Method:
Opens up the add group pop up (Appendix B 7.1.3.1 Add Group).
Call create group name method
Call choose colour method
Call date added method
Call date edited method
Call user method
The user is prompted to input a group and select a colour from the colour wheel (Appendix B 7.1.3.1, 7.1.3.2).
Pressing add will send the data to the database.
Pressing cancel closes the pop-up.

Completed Method:
Pressing the complete button(s) on a task calls the complete method.

Logout Method:

Pressing logout calls the logout method, the logout method which brings user back to the login screen

Login Method: When pressing the “log in with GitHub” button, it calls the Fire Base login method. Which brings the user to the login screen, if the attempt is successful,   is brought to their home page with all data displayed that they have created.

Sort by (Due Date) Method:
Pressing the data button, queries the database for all non completed tasks labelled with the username. They are sorted and displayed by date due, clicking the button again inverts the list.

3.3 Strategy

The overall system makes use of as many frameworks as possible; these frameworks save a lot of time. Due to the time constraint of the project, a simple implementation was required in order to meet the design and deadline requirements.

The alternatives considered were: an SQL database that was rejected due to set up complexity, and the time required to learn how to use it. A JSON database, that was rejected due to the complexity of tying it into a NodeJS backend. The sorting of JSON was a particular challenge. Simple data structures were not possible due to the amount and level of complexity of the data, also, the need for it to be persistent. By using a NoSQL database, with almost all the backend implemented, as well as other features like user authentication, we felt it best met our needs.

On the frontend, React was instantly rejected due to the difficulty in learning it, and the problem of not knowing the basic syntax of vanilla JavaScript. HTML with simple CSS was also rejected due to the time to learn how to implement a frontend. We decided on using bootstrap, which would allow us to create complicated CSS elements that already existed as templates. Bootstrap provided that functionality.

In terms of the web app, we initially felt it would be easier to run client side, but the need to install NodeJS, and the issues of deploying the web app seemed easily solved by an all-in-one hosting solution like GitHub.

4. Database Schema

4.1 Tables, Fields and Relationships

4.1.1 Databases

Firebase will be the database used for this project. It is a NoSQL database that sorts through many separate JSON files, which will function as a black box that we will interact with through its end points using JavaScript.

4.1.2 New Tables

We have a collection of JSON files, these files are randomly ordered, one of the elements within the JSON file contains user data, user authentication allows sorting results based on who is accessing them. Thus, every JSON file represents a *task* (refer to 1.4 definitions). In addition, a separate JSON file for each user would be required to store Group names.

4.1.3 New Fields(s)

Table Name	**Dataset Name**	**Data Type**	**Dataset Description**
Firebase	Task(0-n)	Dictionary	Username is stored from authentication data as a string, date added is stored as a string and taken from serversystemtime(in firebase), date modified is stored as a string and taken from serversystemtime(in firebase), data due is input from the user and stored as a string in UNIX system time, group type is taken from a query to the group JSON file it is a string, task *type* is taken from hard-coded values and stored as a string, completed is a boolean value
Firebase	Groupname[user name]	Dictionary of keys	Dictionary of key-pair values or dictionaries, one of the values is a string containing a course name, the other value is an RGB colour value. In addition, the username is stored in this JSON file.

4.1.4 Fields Change(s)

Table Name	**Dataset Name**	**What to change?**

Firebase	Task	Due Date, Date edited
Firebase	Task	Task Name, Date edited
Firebase	Task	Completed, Date edited
Firebase	Task	Remove Entire Task
Firebase	Group	Group name, Group Colour (add only)

4.1.5 All Other Changes

With regard to security, using the built authentication features of Firebase, all user data will be tied to the user’s GitHub account.

4.2 Data Migration

When a new user is added, a *group* (refer to 1.4 definitions) JSON object is automatically created. Within this object, the value “default”, and a default RGB colour of black are inserted, as well as their username.

5. High Level Design

5.1 Context Diagram

See diagram 5.1.1

User: The user that interacts with the Study Buddy application

Study Buddy: The application that will organize and display tasks to the user

Database: The Firebase database that will hold all the data that will be managed by Study Buddy

Interactions:

User -> Study Buddy:
- The user submits data to Study Buddy

Study Buddy -> User
- Study Buddy displays data to the user

Study Buddy -> Database
- Study Buddy submits data to the Database
- Study Buddy queries the Database for data

Database -> Study Buddy
- Database holds the data submitted by Study Buddy
- Database sends back data that is queried by Study Buddy

5.2 Data Flow Diagram

Please refer to the diagram in the appendix labeled as 5.2 Logical View DFD.
If we look at a logical representation of the system, the components within this are relationships between the user and the database. We consider different operations to the database as Task Management operations, Tasking Sorting Operations.
The interaction between these entities are described here:

The user's interactions with "Task Management" are:
The user will be able to request the task management object to:
- create new *tasks*
- create new *groups*
- complete a specific *task* within the system and the task management system will store the completion date of the *task* in the database

The Database's interactions with "Task Management" and Database are:
- "Task management" will be interacting with the database to make the changes as requested by the user.

The relationships between "Task Sorting" and "Task Management" are:
- The "user management" will be interacting with the "task sorting" to refer to a specific *task* in the database

The Relationships between "Task Sorting" and the "Database" are:
- "Task Sorting" will query the database to find a specific *task* from the database.

6. User Interface Design

6.1 Application Controls

6.1.1 Login Screen:

Upon visiting the website, a login window will be displayed. The user will have to enter their account info for GitHub, at which point the backend will process this information and will then display the home screen of the application.

See figure 6.1.1.1

6.1.2 Main Screen:

The application will be contained within a common home screen, through which all major functionality will be able to be viewed and interacted with by the user. Within the top right of the application will be the user's name, and a logout button. Interacting with the logout button will bring the user back to the login screen. On the top left of the page is an 'Add Task' Button. On the far right side of the screen is the 'Add Group' button, and below this is a legend of the existing *groups* that the user has made, along with their associated colours. As the user adds *groups*, the legend for colours and *group* names will be updated and displayed. As the user adds *tasks*, they will appear in the main left part of the window in a large field which displays their name, *group* they belong to, *type* and due date. *Tasks* will be sorted by due date. Each *task* will appear in the colour which was assigned to its *group*. As *tasks* are added, the list will be updated.

See figure 6.1.2.1

6.1.3 Add Task Popup

When the user clicks the 'Add Task' button on the main screen, an 'Add Task' popup will appear. This popup will feature several items the user can input. There are several fields for the user to interact with to create a *task*.

The first field is the 'Select Group' field: it is a dropdown that allows the user to select any of the *groups* they have created. By default, there exists only a 'default' *group* type.

See figure 6.1.3.1

The second field is the 'Select Type' field: a dropdown menu appears with four options for the user to select from. These are Assignment, Test, Project, and Lab.

See figure 6.1.3.2

The third field is the 'Task Name' field: a text input box for the name of the *task*. The user can enter any combinations of characters for this value.

The fourth field is the ‘Date Due’ field, this is entered from a calendar view selector.
See figure 6.1.3.3

Below these fields are the ‘Add’, and ‘Cancel’ Buttons.
See figure 6.1.3.4

If the user clicks ‘Add Task’ and there are no errors with the user’s input, or if the ‘Cancel’ button is pressed, the user will be taken back to the home screen. If a *task* was successfully added, a second popup window displaying “Task successfully added!” will appear.
See figure 6.1.S

If there are input issues when pressing the ‘Add’ button, the ‘Add Task’ popup will not close, and a second a popup will appear stating “There was an issue with their input, please try again.”. The user must correct their input or select ‘Cancel’ to return to the home screen.
See figure 6.1.E

6.1.4 Add Group Popup

When the user clicks the ‘Add Group’ button, a popup window is displayed with two fields. The first field called ‘Group’ contains a blank text input box, where the user will enter the name of the *group* to be added.
The second field called ‘Colour’ contains a colour wheel, allowing the user to select the colour which will identify all tasks created in this *group*.
Below these boxes are the ‘Add’ and ‘Cancel’ buttons.
See figures 6.1.4.1 and 6.1.4.2

If the User clicks ‘Add Group’, and there are no errors with the user’s input, or if the ‘Cancel’ button is pressed, the user will be taken back to the home screen. If a *group* was successfully added, a second popup window displaying “Group successfully added!” will appear.
See figure 6.1.S for general functionality

If there are input issues when pressing the ‘Add’ button, the ‘Add Group’ popup will not close, and a second a popup will appear stating “There was an issue with their input, please try again.”. The user must correct their input or select ‘Cancel’ to return to the home screen.
See figure 6.1.E

6.2 Screen

For the user, there will only be a few different display options. The three basic types will be the login screen, the main menu screen and a popup window for adding or editing the database. The breakdown of these tasks can be viewed in the Appendix. The main menu screen will be dynamic; as elements are added to groups or tasks, they will be displayed on that screen.

See figure 6.1.2.1 For a General overview

7. Low Level Design

7.1 Main Page

7.1.1 Sorting By Date Button

Sorting of tasks will be accomplished by selecting the ‘Date’ label column. This will sort the due dates by either newest or oldest. To toggle from sorting newest to oldest, the user can click the “Date” label column.
Clicking on this particular object will call queries to the database, and make the queries sort the contents by date. Objects by default will already only be displayed if they belong to a specific user via standard authentication features built into Firebase. All pertaining unmarked JSON objects will be displayed except for the *group* JSON objects which will not be displayed within this field.
With no objects in the table, by default, the web app will be set to not display any task objects until they are added.

7.1.2 Complete Task Button(s)

For every displayed element on the page, there will be a column that has a visual check mark within it. This particular spot on the web app next to that element has a unique button that allows the user to click it. The initial value of the boolean value ‘completed’ is false. However, once the checkmark is clicked, it edits the boolean value ‘completed’ to TRUE for that particular JSON task element. If there are no elements displayed, there will be no functioning *task* button; it requires a value to edit. This will remove the task object from the task list.
One advantage of this design is that it removes the requirement to search if an object exists, and mitigates the amount of text rejection for user input.

7.1.3 Add Group Button

The ‘Add Group’ Button displays a popup on the user system, and on the backend, it interacts with the database to add a JSON object, this particular JSON object is unique by its parameters related to the *group* class of JSON objects. These objects contain, colour, name, username and several timestamps. By default, there will always be at least one object created which is “default.” The default *group* has a colour of black assigned to it.
When adding a new *group*, error handling must be performed on the text input to ensure it is valid and does not match any existing *group* names. In addition, this is a maximum number of *groups* that a user can have that is 15. If 15 *groups* exist, the user may not create more JSON objects of this kind. The colour is then selected by the user via a colour wheel on the frontend, which is turned into an RGB 8-bit integer. Once the input checking has been completed, the ‘Add’ button within the popup will

send the command to the database. Upon being added, the front end displays all group objects in the legend. This option is on the right-hand panel of the web app.

7.1.4 Add Task Button

The ‘Add Task’ Button will cause the frontend to launch a pop-up on the client machine. Input is required from the client for the name of the *task*, and error checking will be performed on this field. A *group* field will display all current *group* objects that exist by querying the database and returning these as options in a dropdown menu to the user. The name field of the particular *group* JSON object will be used as the *group* name within the *task* JSON object. The *type* field will draw from a hard-coded set of default values stored within the JavaScript file. Selecting one of these values will select a string to be input.
Lastly, the date due object will use a front-end calendar application to convert that selected date to UNIX time at midnight of that selected day. Once all these fields are input and error checking has been run, the ‘Add’ button sends those values to be input as a *task* JSON object within Firebase.

7.1.5 Logout

By clicking on the logout button, the user goes back to the login screen. This process is being handled with GitHub and Firebase, the application leverages functionality built into Firebase to ensure this is done securely.

7.1.6 Main Page Display

The main page of the web app will query and display the relevant data from JSON to the user on the main page. The majority of the main page will be dedicated to displaying the four elements of each *task*. The items displayed are the Name, Due Date, *Group*, and *Type* of the *task*. Every time data is added, removed or modified, the JSON table changes, and the output will be updated to reflect these changes. There will be a legend present on the right panel of the web app, which displays all relevant JSON objects that exist that are of the type *group*. The elements within the legend are updated as group objects are added.

7.2 Login Page

7.2.1 Login

By default, when a user visits the web app, they are greeted by a webpage that displays the name of the application above the sign-in button. The JavaScript file handles this with the help of Firebase, using this allows the site and the data of user’s to remain secure.

Appendix A: Project Timeline

November 4th Frameworks

November 16th Test Planning

November 25th Test Suite/ Deployment (Error checking of input values, dependency service downtime)

December 2nd Demo Dry Runs (MVP in Production)

December 9th Final Deployment of Production

Appendix B: Figures

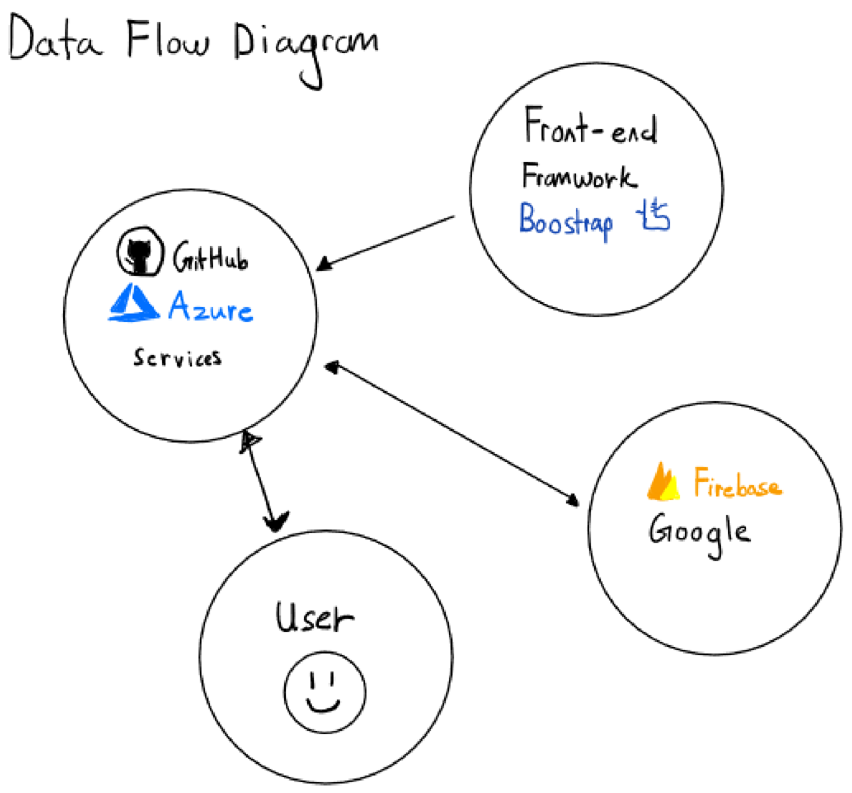

Figure 3.1.1 Physical Services DFD

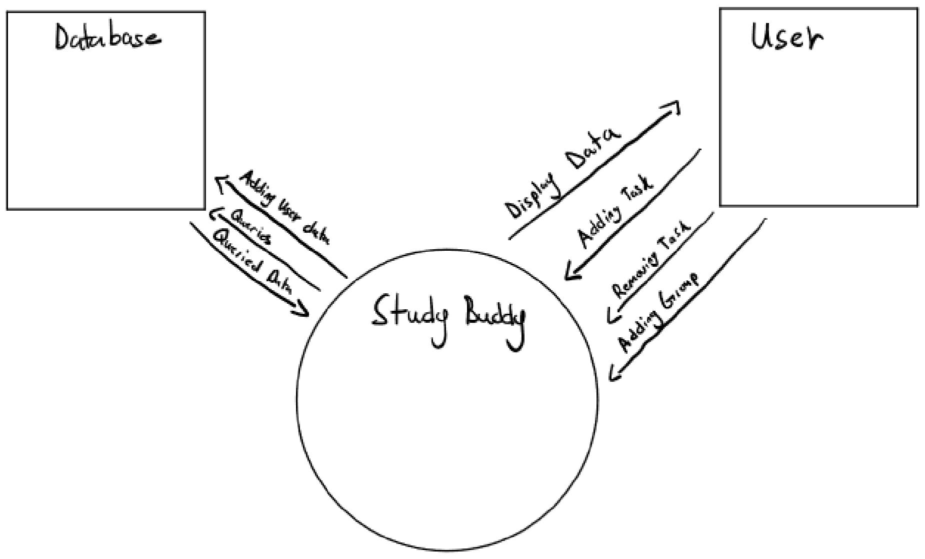

Figure 5.1.1 Context Diagram

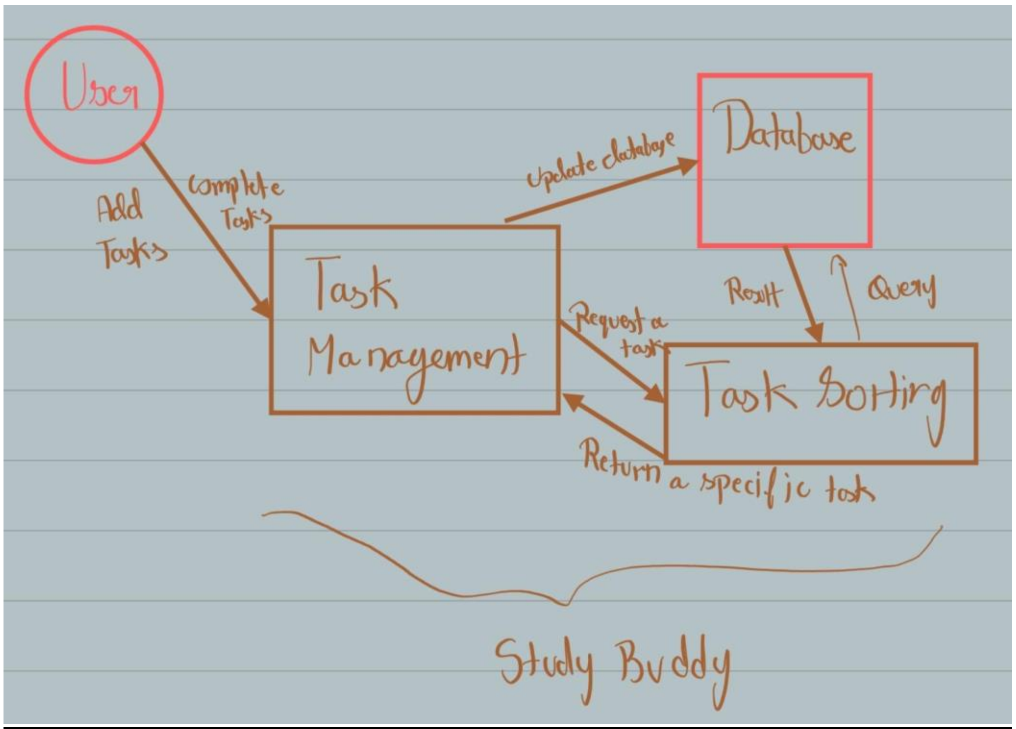

Figure 5.2.1 Logical View DFD

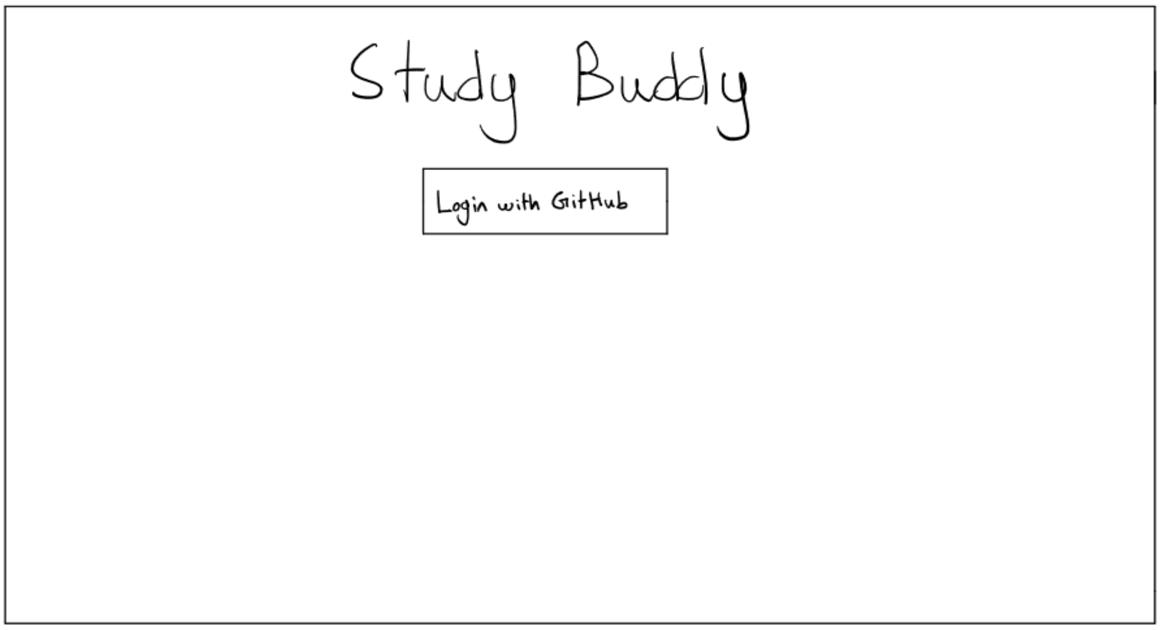

Figure 6.1.1.1 Login Screen

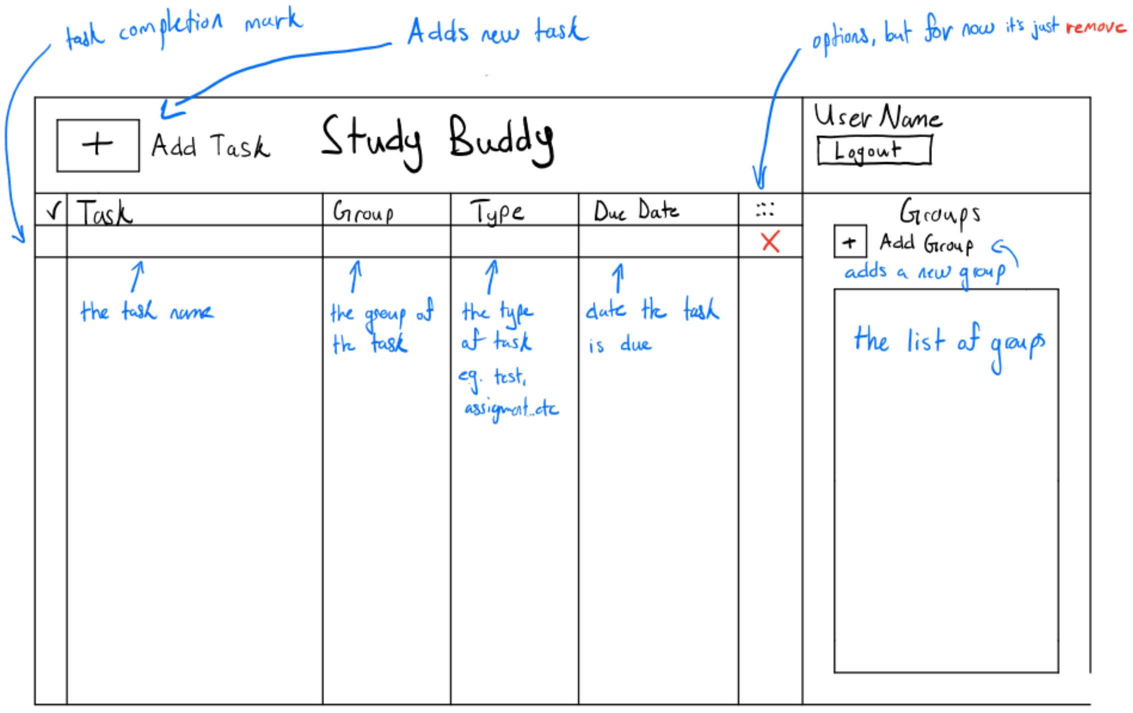

Figure 6.1.2.1 Main Page

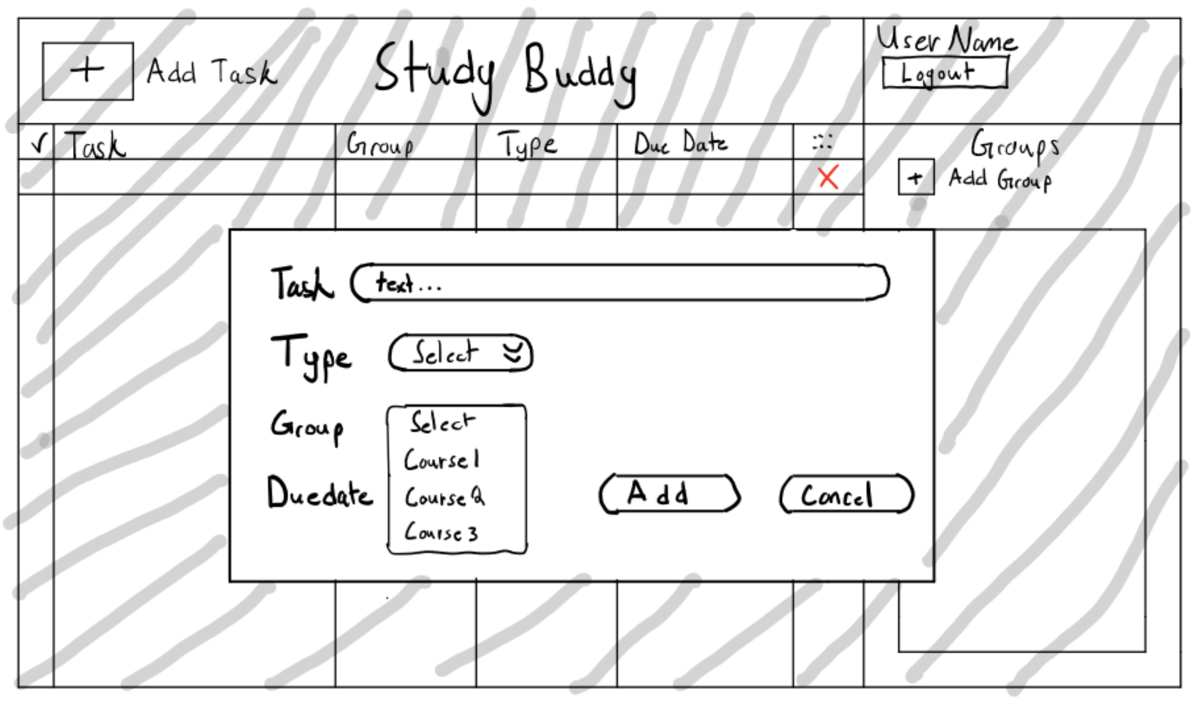

Figure 6.1.3.1 Add Task Group Select Dropdown

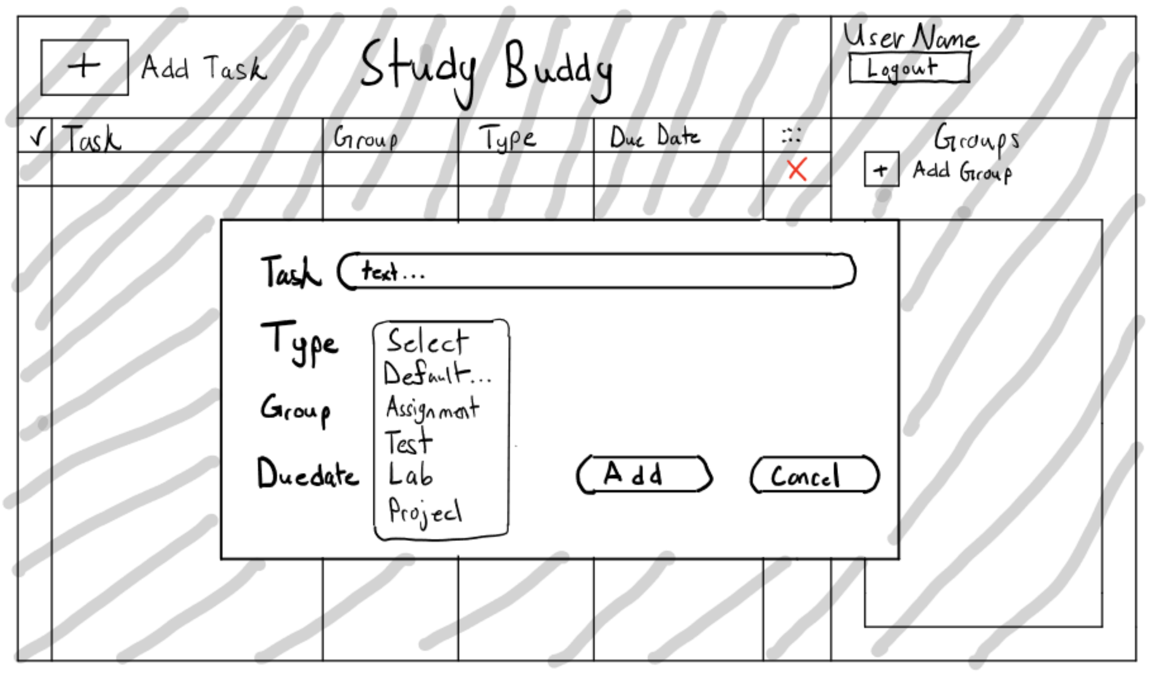

Figure 6.1.3.2 Add Task Type Drop Down

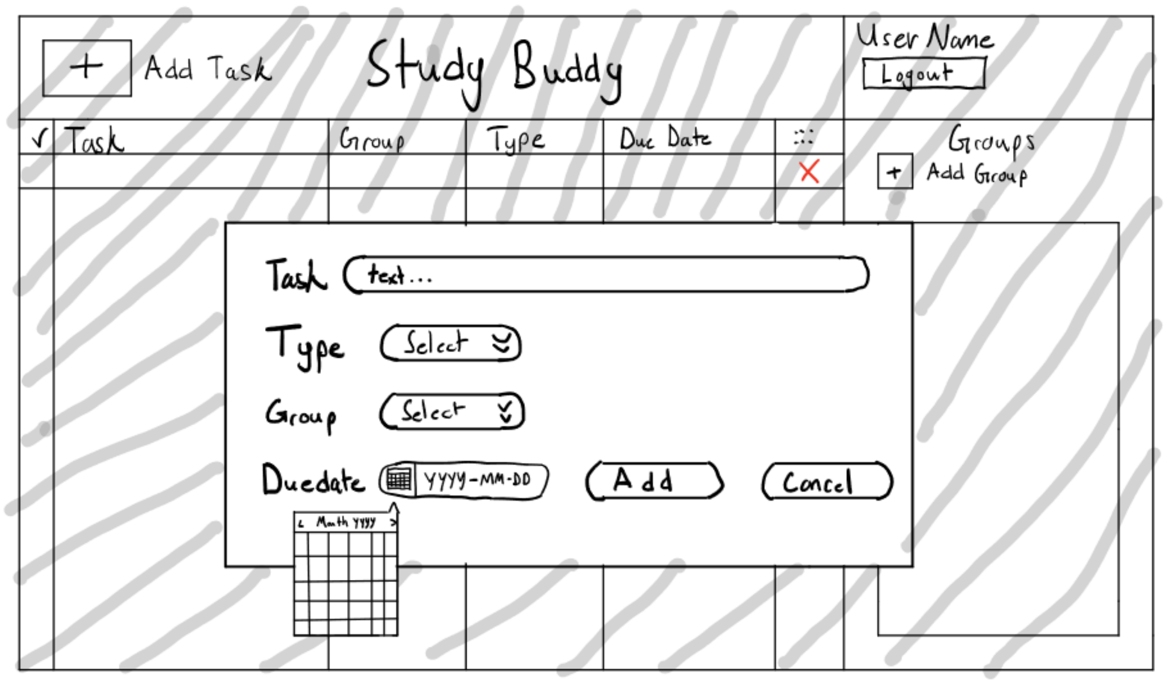

Figure 6.1.3.3 Add Task Date Drop Down

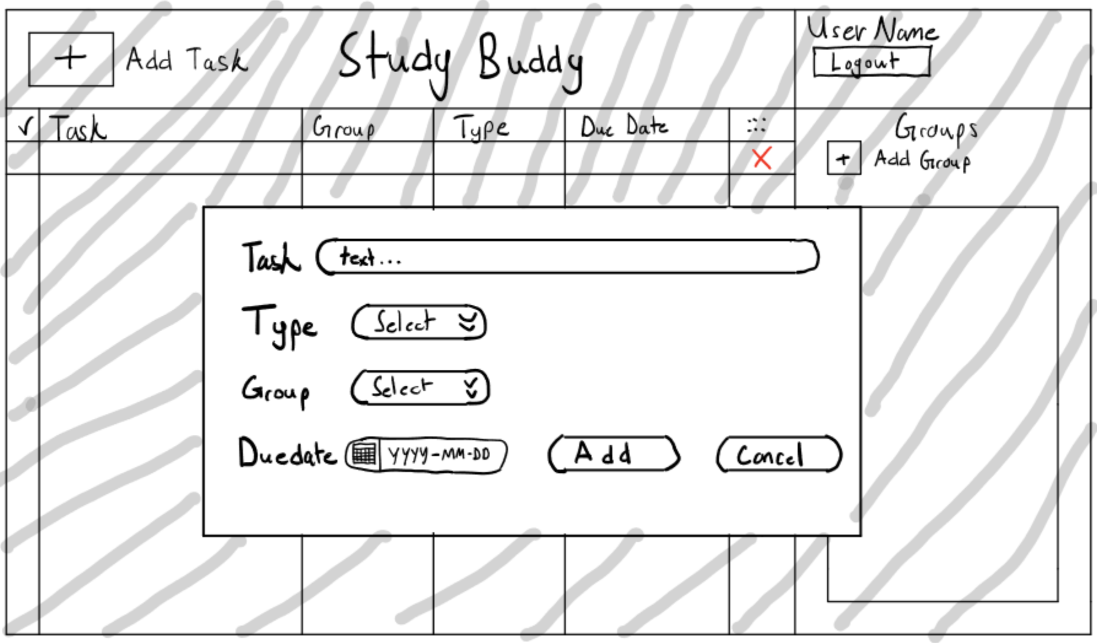

Figure 6.1.3.4 Add Group Popup Menu

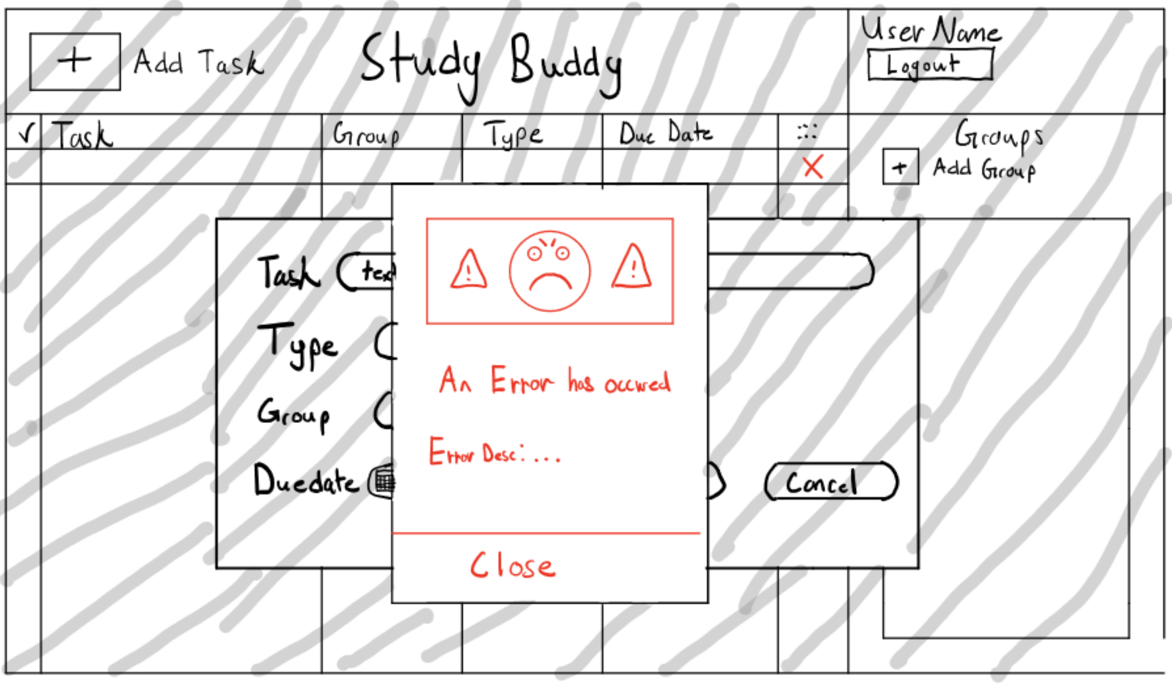

Figure 6.1.E Input Error

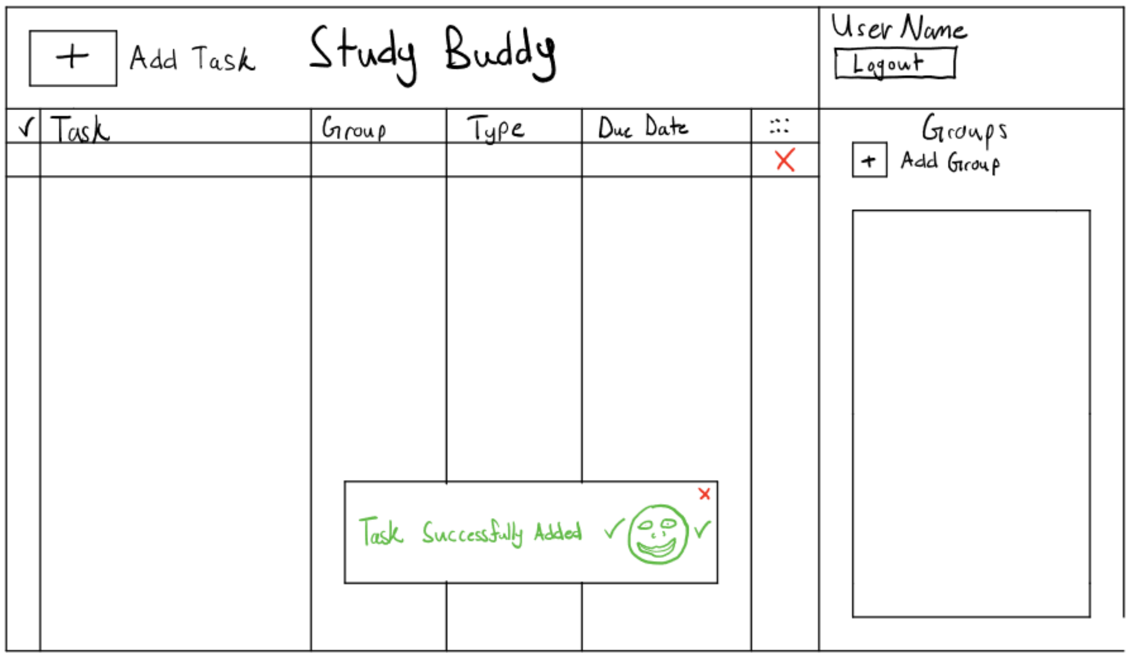

Figure 6.1 S Added Successfully

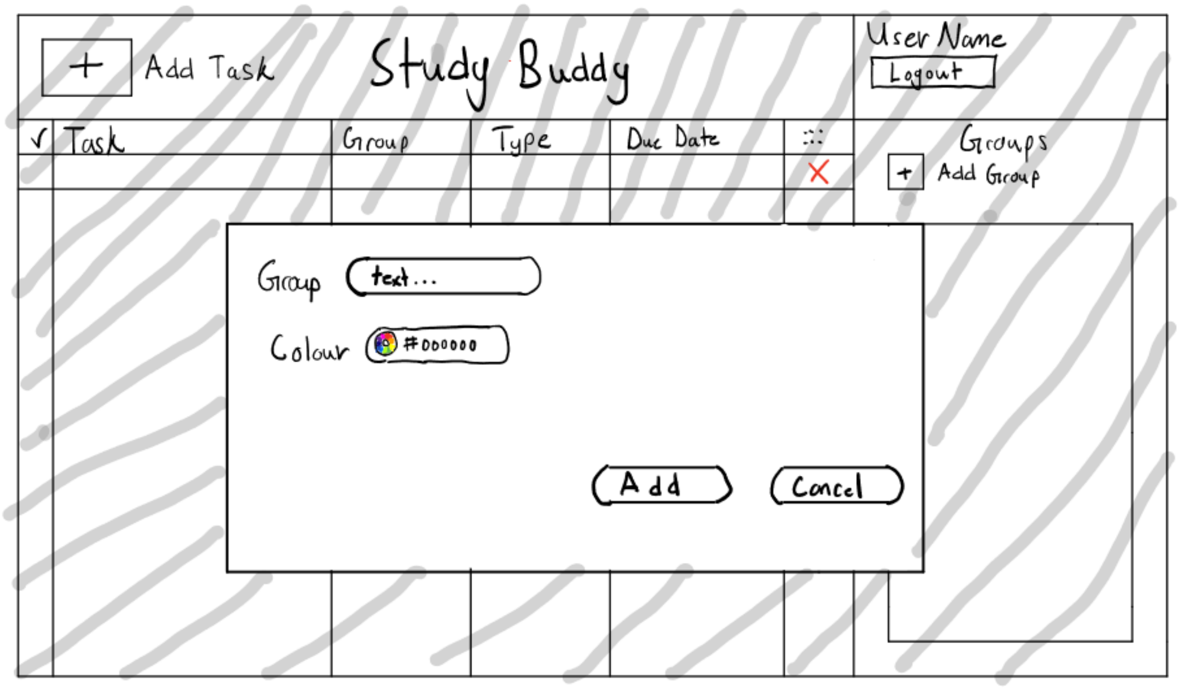

Figure 6.1.3.1 Add Group

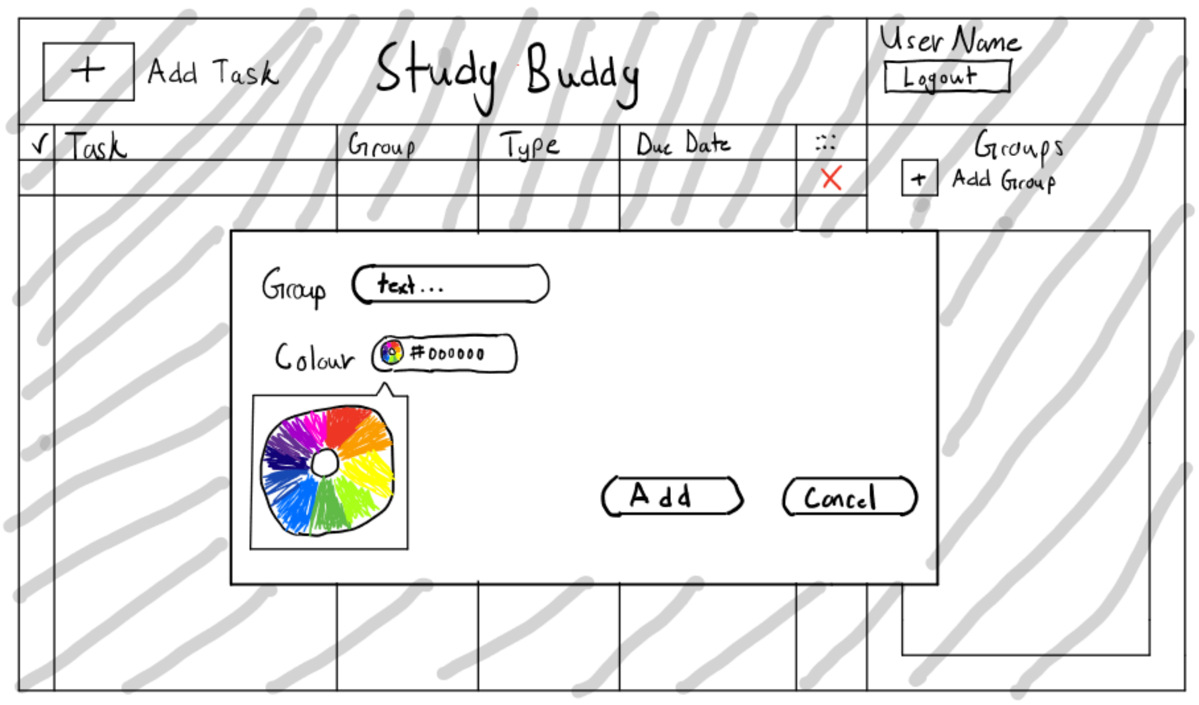

Figure 6.1.3.2 Colour Wheel for group colour selection