Deliverable 1

BYTE

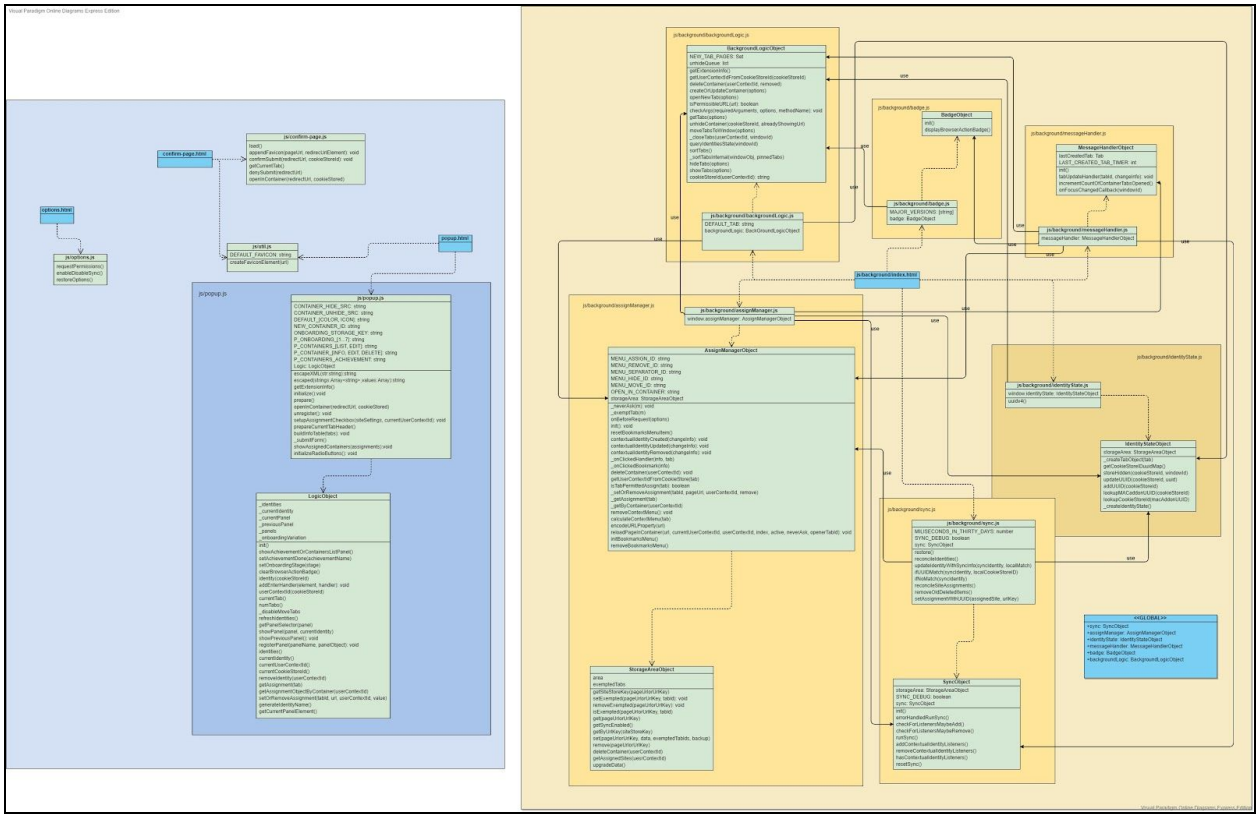
Architecture of Multi-account Containers

Overview

The OSS Multi-account Containers is a JavaScript add-on designed for Mozilla Firefox and allows users to open new tabs in separate containers such as Personal, Work, School, and Shopping. There are three main parts in this add-on we have to work on: popup, options, and the background processes (i.e, the actual functionalities of this project). We can also categorize them into **frontend** (popup and options) and **background processes**.

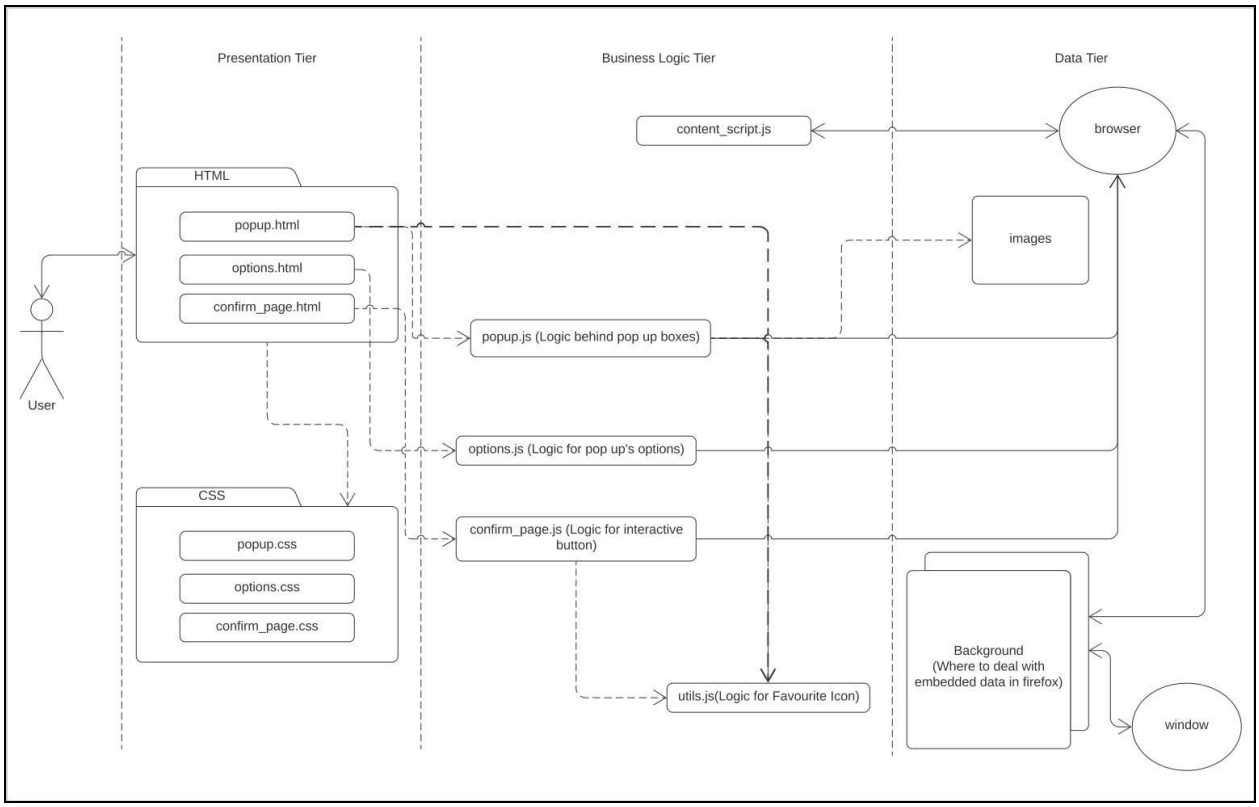
The **frontend popup page** contains the user interface of the popup and its corresponding functionalities: controlling the containers and viewing all added containers. The frontend options page contains the interface and controls for the options.

The **background processes** handle all the processes controlling the containers' activities, including cookies storage and tabs management. According to what tabs are in a container, the background processes will keep track of these tabs without interfering with other containers and their cookies.



Link to the detailed UML available at:
https://github.com/CSCD01/team_05-project/blob/master/docs/uml-details.pdf

Frontend



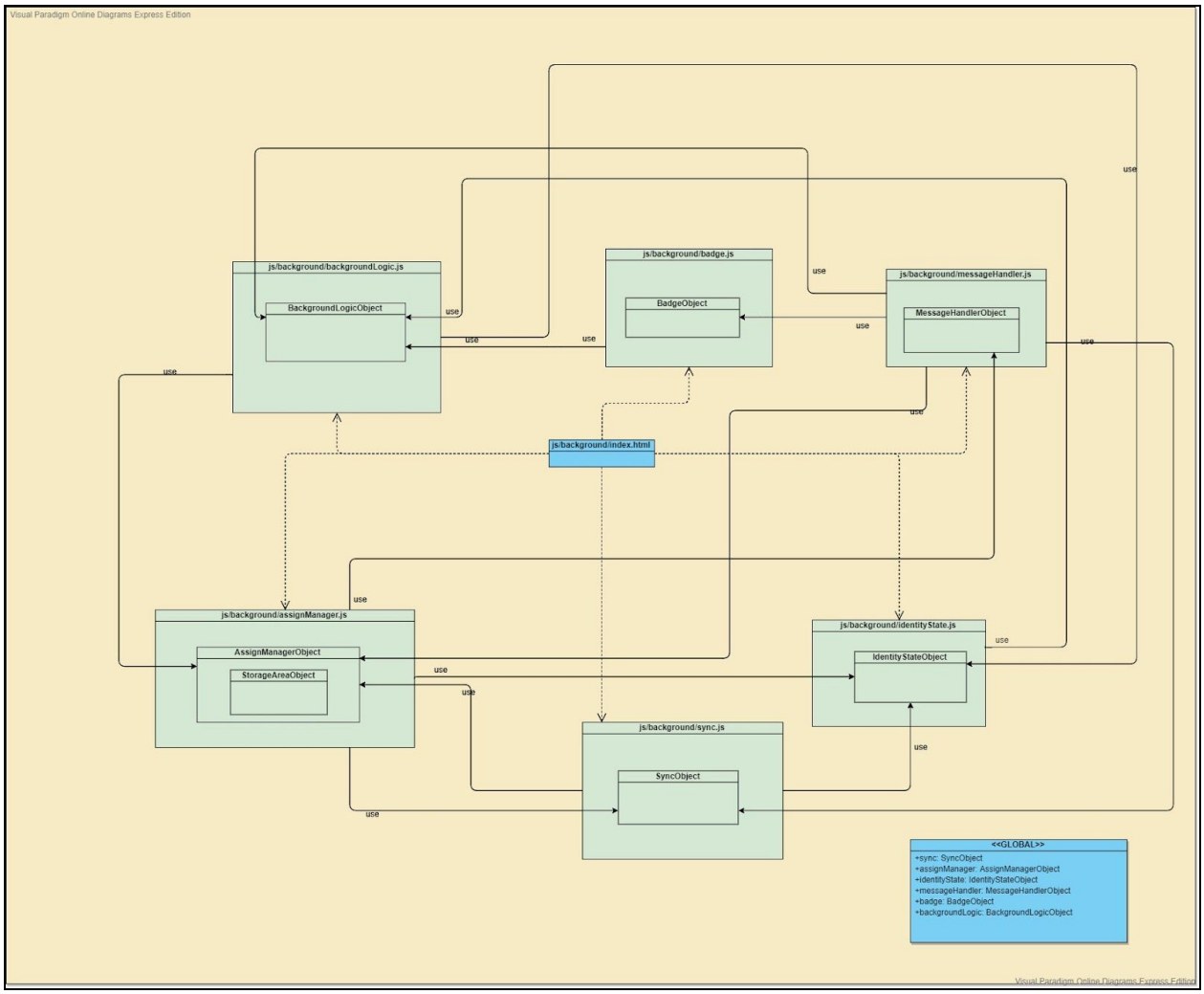
(Link to the 3-tier architecture graph available at: https://github.com/CSCD01/team_05-project/blob/master/docs/tiers-architecture.pdf)

We used **3-Tier Architecture** to show the relationship between the user interface of the popup page (options page and confirm_page), the business (control) logic, and the data used in this project.

The **Presentation Tier** contains the view model of the popup page and options page. Each HTML file in the Presentation Tier demonstrates the result from JavaScript files in the **Business Logic Tier**. They control and handle the user interactions on the pages. Every file in the Business Logic Tier is independent of each other.

The **Data Tier** includes all the resources and embedded data (from Firefox) for this project to run. Images include all the images and icons used by the popup page. Browser and windows are the data objects from the user's Mozilla Browser (built-in module, which is not part of the project). The 'background' code is to deal with embedded data in Mozilla Browser. These data are shared among different files by using 'browser'.

Background

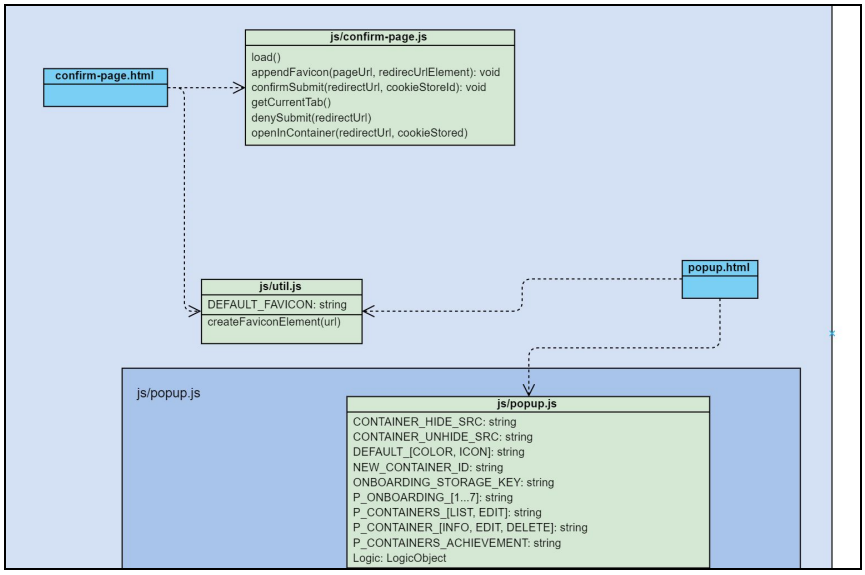


Background components are completely separated from Frontend components, according to the result of the reverse engineering tools we used. Background and Frontend components do not call functions from each other. This Mozilla add-on is run through a command-line tool developed by Mozilla, web-ext, to help build, run, and test web extensions. Therefore, we will just show the structure of the Background component without the connection between the Background and the Frontend components.

Background components include all the processes needed to run behind the scenes, including sync.js to sync user's containers and site assignments across devices, identifyState.js for managing cookies, messageHandler.js for handling actions received from files in the aforementioned Business Logic Tier, and three more files as shown in the graph. Each of these files contains an object as a global variable, so files in Background can access objects from the other files by accessing these global variables.

Interesting Aspects

In the Frontend part, all JavaScript files in the Business Logic Tier are only used by one HTML file (except utils.js which contains logic for creating favourite icons). This shows that every module/class is independent of each other, which means they can be tested and implemented separately.



In the Background part, there is an object in each JavaScript file that corresponds to different functionalities. These objects are declared to be global variables through the .eslintrc.js file. Therefore, the objects can be widely accessed in any file under the Background folder.

```
module.exports = {
  "extends": [
    "../../.eslintrc.js"
  ],
  "globals": {
    "assignManager": true,
    "badge": true,
    "backgroundLogic": true,
    "identityState": true,
    "messageHandler": true,
    "sync": true
  }
};
```

Quality of Architecture

According to our high-level diagram, the architecture is clear and concise in terms of file organization. It separates the Background processes code from the Frontend control and view code. Source, Test, Docs, and Coverage are completely separated.

SOLID Design Principles

- **Single Responsibility Principle** means every module in a program should have only one responsibility, and the responsibility should be encapsulated only by the responsible class. In this project, they separate files according to their responsibility in the Frontend part. For example, popup.js only handles user interactions happening in popup.html, options.js only handles user interactions happening in options.html. Also, each function only does one thing and we can easily understand what it does by reading the function names. The Frontend part follows the principles.

  Meanwhile, the Background part does not follow the Single Responsibility Principle due to the use of global variables for the object in each file (module). It is good that each module only handles one responsibility, but they share variables with each other which violates the Principle (not encapsulated by the responsible class). Therefore, the Background part does not follow the principle.

- **Open Close Principle** means that some entries should be open for extension but closed for modification. In this project, popup.js and messageHandler.js violate this principle because it loops through all the hardcoded elements under the ‘switch’ condition. This causes a potential problem for the future extension because the ‘switch’ condition always needs to be modified whenever there is a new update.

```
switch (onboarded) {
case 7:
  this.showAchievementOrContainersListPanel();
  break;
case 6:
  this.showPanel(P_ONBOARDING_7);
  break;
case 5:
  this.showPanel(P_ONBOARDING_6);
  break;
case 4:
  this.showPanel(P_ONBOARDING_5);
  break;
case 3:
  this.showPanel(P_ONBOARDING_4);
  break;
case 2:
  this.showPanel(P_ONBOARDING_3);
  break;
case 1:
  this.showPanel(P_ONBOARDING_2);
  break;
case 0:
default:
  this.showPanel(P_ONBOARDING_1);
  break;
}
```

Possible Improvements

- **Use the Singleton Design pattern** to share the objects instead of global variables. In order to follow the Single Responsibility Principles, each module should only have one responsibility and be encapsulated by the responsible class. If the program really needs to access objects from other classes, we should use the Singleton Design pattern to provide a global access point instead of that instance. It can still have the same functionality as global variables, but the accessed variables will be protected from being overwritten by other code. This makes sure an instance is encapsulated only by the responsible class.

- **DRY (Don't repeat yourself)** because duplicated code occurs in some js files. We can resolve this by sorting the repeated code into functions.

- **Use the Open Close Principle** instead of using the 'switch' condition in messageHandler.js and popup.js. We can apply the **Observer design pattern** to notify all observers instead of looping through all the elements whenever there is an update. This will improve code quality and readability, and will be easier to handle when there are further changes in the data structures. It allows the number and type of objects being observed to be configured dynamically.

- **Test cases** only cover half of the project. Some issues on GitHub arose from code that is not well tested. Additionally, the test structure needs to be separated into different parts so that each part takes charge of only one js file.

- Some **variables have meaningless names.** It causes difficulties in future maintenance. The naming should satisfy a professional naming style (Google style, Camel Case or Snake Case) instead of using meaningless letters or numbers.

Software Development Process

We have chosen **Kanban** as our software development process for the following reasons.

1. **Visual progress of our work**

With a *Kanban Board*, **we can easily see the current progress of our tasks and what everyone is currently working on**. Being able to view these at a glance is important to identify bottlenecks, determining what tasks team members may be struggling on and offering support where necessary, as well as to save time determining what tasks are left to be done.

2. **Increased productivity and focus**

This can be achieved due to Kanban's *WIP (Work In Progress) limit*. Having a limit to how many tasks we can take up at a time will improve our productivity because **we will spend less time switching between tasks and instead use this time to complete these tasks**. Working on fewer tasks at a time also means an increased focus on these tasks, ensuring a higher quality product at the end.

3. **Encourages feedback and collaboration**

Kanban's *daily stand ups* will be used as a means for sharing information among team members. This information usually entails what tasks we're currently working on and any roadblocks we are having. By being transparent about the problems we're having, **we will be able to rely on each other more as a team and help each other wherever needed**.

Workflow

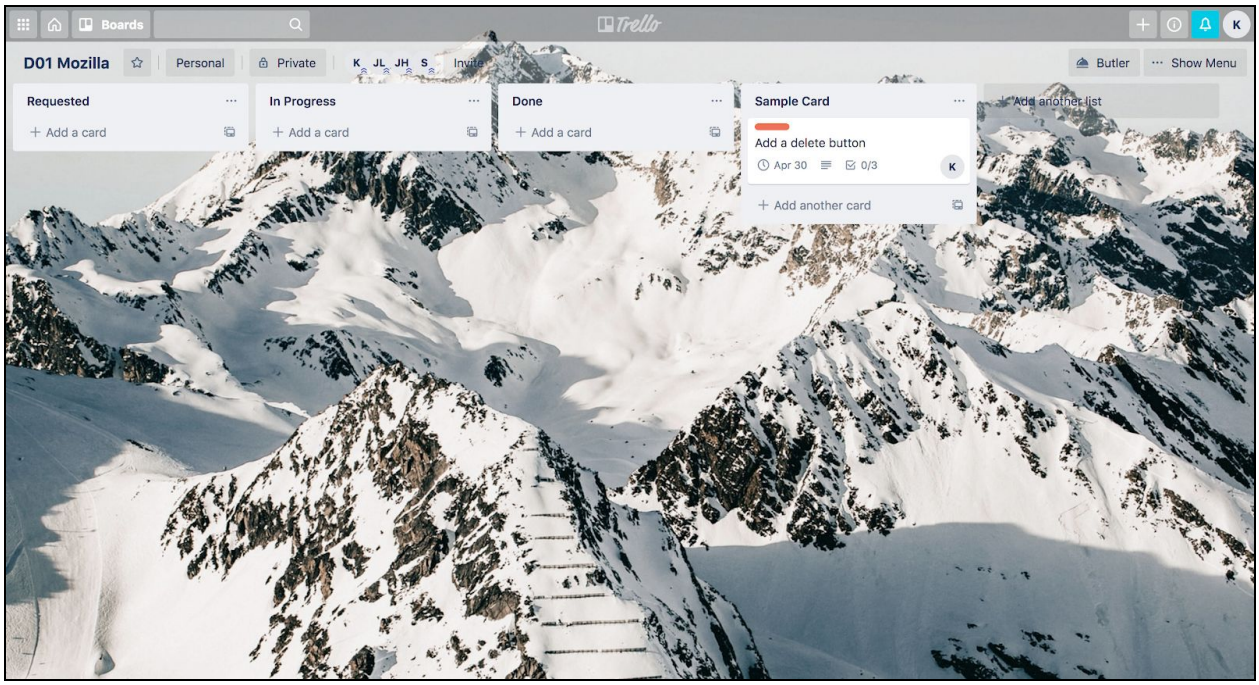
Keeping Kanban's practices in mind, we explain the workflow of our process.

During our weekly meetings, we will **analyse the issues we have to implement and translate them into smaller technical tasks**. These tasks will then be put into our Kanban board's *Requested* list, after which **everyone will choose which tasks they want to work on** (as detailed in our team agreement). Tasks will move from *Requested* to *In Progress* only when we start on these tasks, whether it be research, design, or implementation, and will move from *In Progress* to *Done* only when we deem it as done (also detailed in our team agreement). Additionally, we host **daily stand up meetings that last 15 minutes each to summarise our progress**.

Team members will be **free to pull any tasks from *Requested* to *In Progress* any time they want**, as long as they don't hit the WIP limit. If any discussion is needed as to which tasks to undertake, these can be done during our stand ups.

Kanban Board

Our Kanban board will be hosted on Trello, pictured below.



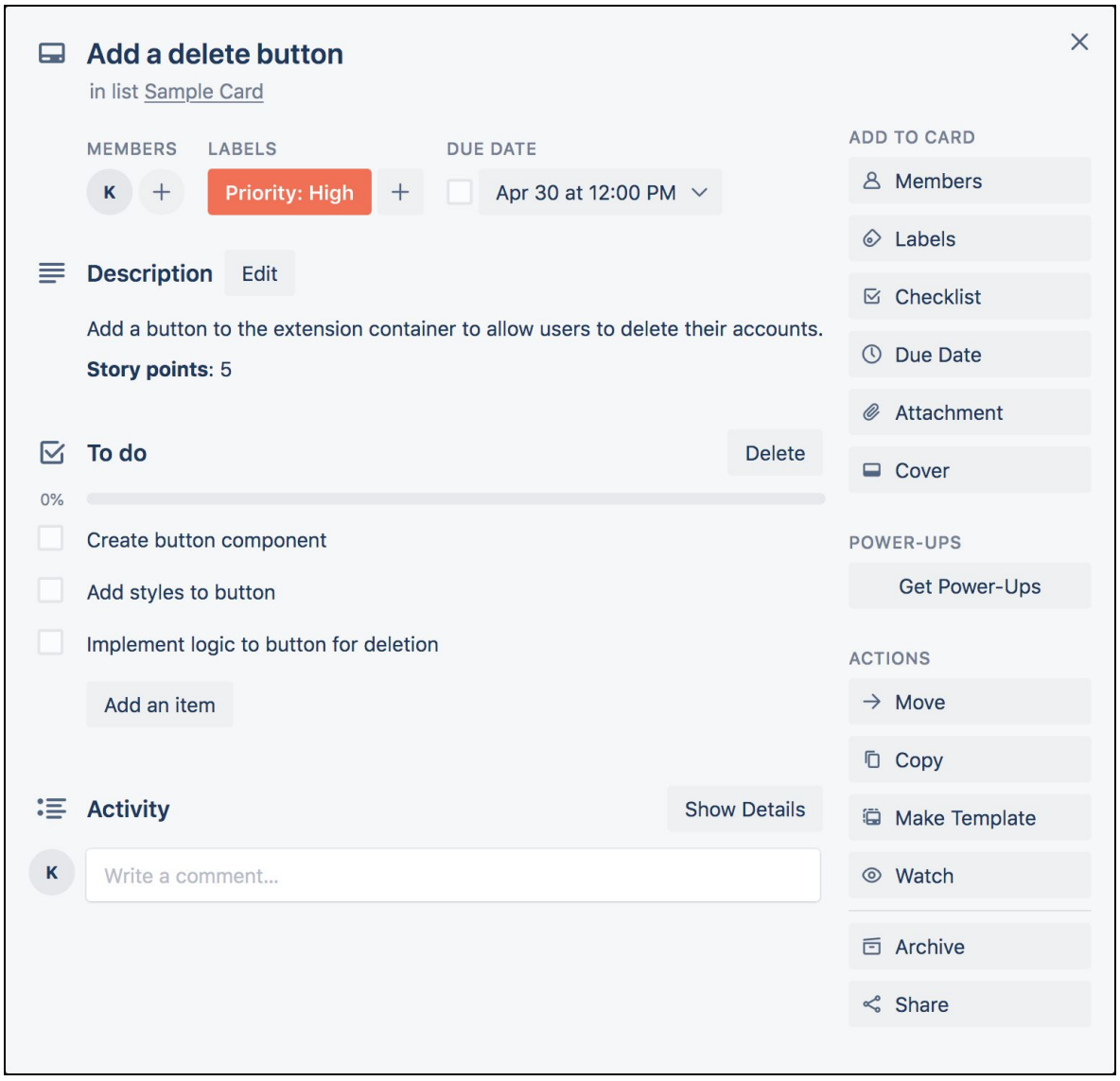
Link: https://trello.com/invite/b/HxNEWM5G/e95972a95b77b4e2c7e2d8f4a33b6617/d01-mozilla

Our board will have **three columns for now**: ***Requested***, for tasks we have yet to start working on; ***In Progress***, for tasks we are currently working on; ***Done***, for tasks we have completed. The *Sample Card* column is merely to display a sample of a *Kanban Card*, which we will discuss in the next section.

Considering the fact that the *In Progress* status can vary between tasks (e.g. a task may be in progress because it is waiting on other tasks to be completed first), **we may consider breaking the *In Progress* column down into a number of other columns**. However, we will proceed with these three columns for now.

Kanban Card

An example Kanban card for our project is pictured below.



Within one card, we will have the following fields:

- **Card title**: A short one-liner describing the task.
- **Assignees**: Listed under *Members*, shows who is to work on the task.
- **Priority**: Shown under *Labels*, determines the priority of the task.
- **Due date**: When we expect the task to be done by.
- **Description**: A longer description of the task that may include more technical details. Will also include story points, which determines the difficulty of the task.
- **To do (optional)**: A further breakdown of the task. Not required, but makes it easier to keep track of current progress.

WIP Limit

In order to determine our *WIP limit*, we must first determine the nature of the tasks we put in each of our Kanban cards.

Where a typical user story could be broken down into smaller individual tasks that would amount to its overall completion, **one Kanban card of ours would consist of one of these smaller individual tasks**. This way, each task will be small enough for one developer to work on. It will also be **easier to visualise the task within the workflow of the board** as opposed to if we were to have an entire user story as a card and these smaller individual tasks within the card.

Now that we know each Kanban card will make up one subtask of any user story we have, we will set a limit of 2 tasks (or Kanban cards) per team member for now. This will add up to a 2×5 = **10 task WIP limit**. Having 2 tasks to work on at a time as opposed to 1 will ensure that **team members are still able to be productive** even if one of their tasks is faced with a roadblock or is waiting on another task to be completed. The WIP limit per team member is also **not too large as to accidentally enable inefficiency with context switching between tasks**.

Daily Stand Ups

Considering the fact that all five team members are full-time students with other courses to worry about (with some of us taking 6 courses this semester), we have decided to **modify Kanban's daily stand ups to stand ups that occur every two days**. This is because we feel **it's unrealistic to get a lot of work done in one day since we have other assignments and group projects**.

Further, we have decided to **have our stand ups in our Discord channel** since it is difficult for all of us to meet up at the same time with our different schedules. Having the flexibility of doing our stand ups online will make it easier for all of us since we won't have to commute to the university to hold them. Also, as mentioned in the above section, our stand ups will consist of each team member explaining what they've done in the time between our previous stand up, what they plan on doing until our next stand up, and what roadblocks they've come across.

As agreed upon in our Team Agreement, we will also have **weekly face-to-face meetings every Wednesday**. These meetings will be longer than our daily stand-ups, and will consist of the following agenda:

- Summarise the work we did in the past week and see everyone's progress on the Kanban board.
- Analyse the issues we have left to implement and translate these issues into more technical terms appropriate for developers.
- Recognise any technical issues anyone is having with their current tasks, and determine if it's still possible to continue with these tasks.

Considerations of Other Processes

Waterfall

Waterfall is known for its linear development. When a project has stable requirements, Waterfall ensures that the final product meets clients' needs and is released on time. This is exactly what we are looking for, but the process does not work exactly in our favour. We are gathering tasks from GitHub where most issues are reported by developers and may lack comprehensive details on what is needed. This means the requirements gathered will not be stable and are likely to change in the future. Waterfall does not allow for changes during the development process, so we will only be able to include them when restarting the entire development process again, which will be very time consuming and inefficient. Hence, we did not choose the Waterfall for this project.

Boehm’s Spiral Model

Boehm’s Spiral Model is known for its extensive risk management since it requires risk analysis at every stage. Our team members do not possess the skills to perform risk analysis nor the financial means to hire a professional risk analyst, so this process is not feasible for our project.

Rational Unified Process (RUP)

Rational Unified Process is known for its iterative and incremental development process. It uses rational tool-sets to design and generate projects but the learning curve for these tools is usually steep. These tools are very expensive and our team does not have the time to learn them either, so this process is not affordable for our project.

Extreme Programming (XP)

Extreme Programming is known for its multiple engineering practices such as TDD, pair programming, simple design, and refactoring. These engineering practices are very hard to follow even for professionals and are very time-consuming. We are on a tight timeline with this project and enforcing these practices will be difficult, so this process is not suitable for our case.

Scrum

Scrum is known for its efficient development process and responsive clients. However, it requires very experienced team members and very active clients as part of the development team. Since our project aim is to contribute to Mozilla, our client will be the owner of the repository. Although we can contact the client directly by email, we are not guaranteed to get a response on time. This means we cannot make adjustments according to the client's feedback and so the whole point of Scrum is lost. Hence, Scrum is not suitable for this project.