Nihal Rajan (Project Leader)

Dion Peng

Qusai Ghabrah

Shashank Kumar

TAs: Jakob Hain, Rylan Kasitz

CS 30700: Design Document - Team 6

Project Name: *Ollie*

1 Purpose

Without an effective management system, it can be difficult to track all the aspects of project stories such as assignee, priority, status, etc. Additionally, discussions about the stories may occur through email, Microsoft Teams, and other platforms which would make searching for conversations about a specific topic from weeks or even days ago difficult. One way to solve this is by having a designated section for comments regarding each ticket as JIRA does. Our project will include a lot of the core functionality of JIRA, but we plan to add other features such as different user roles, email notifications about tickets not recently updated, and scripting where when somebody pushes a commit, a ticket is automatically created for the code reviewer.

2 Design Outline

1. We are going to use the client-server architecture model. The three main components would be the client, the server, and the database. The client component’s purpose would be to provide the UI of the web app that would enable users to use CRUD functions for their stories. The server component’s purpose would be to host the web app and handle the back-end logic (e.g. communication with the database). The database component will store and verify credentials along with storing tickets/ticket information, and store the data created on the client side in order to view or manipulate them later on.
2. In the client-server model, there are four components. They are the client, the services the client interacts with, the database for storing project and user data, and the server component which responds to the services that the client has invoked.

Between the client and the client services, the interactions are the login and editing of the task board in order to create tickets and modify them based on attributes like priority of the task. The client interacts with the task board by using the services of the website and changes different attributes in the system based on their needs.

Between the client service and the server component, the services that the client invokes on the task board are stored using the server. Attributes of the task are updated by the client invoking the services of the website and these attributes are stored using the server so they can be reused. Along with this, username and password storage is also one key component that is required when talking about database storage. The user is stored along with a hashed password on the server side and checked when trying to log into the page, which is another interaction between client invoked components and the server. Along with this, the database stores ticket information and tickets.

(c)



3 Design Issues

**Functional Issues:**

| Descriptive Title | Solution options for the issue | Justification of choice |
| --- | --- | --- |
| How should we let users create their accounts? | 1: Manual sign-up process (Fill name, email, password, etc…)  2: Allow users to sign-up using their Google account  3: Allow users to sign-up using their Github account | Using Github will make it easier for us when we work on the scripting part of our project. |
| How should we let users create tickets with different priorities? | 1. Time estimate 2. Story points | Story points are a subjective thing and an unnecessary abstraction for time/effort estimate. |
| What kind of modifications to tickets should be easily accessible? | 1. Delete 2. Hours Estimate 3. Assigner 4. Assignee 5. Moving between statuses 6. Moving to/from different sprints and backlog | These are things every ticket should have and we should make it as smooth as possible for users to fill in. |
| How should we let users search for different tickets? | 1. Search within sprint 2. Search for just title 3. Search by ticket hash | If possible, we would like to include all of the options for search functionality. This is because as the ticket number increases, different searches can lead to the right ticket faster. Along with this, different information known about the ticket with different searches can help find the right one. |
| What database would be the most appropriate for our data? | 1. SQL - Relational 2. MongoDB - NoSQL | We chose a SQL database in this case for its uniformity along with its ease of usage - very easy to integrate into our application. |
| How are we hosting the service? | 1. Web application 2. Desktop application | We are choosing the web application because of the cross platform compatibility. Since we plan to use Github pages to host the web app, this allows for a more manageable application while delivering to the most users possible. |

**Non-Functional Issues:**

| Descriptive Title | Solution options for the issue | Justification of choice |
| --- | --- | --- |
| Keep UI clean with minimal clicks required | Keep the UI simple to start with and just focus on designing it to handle key functionality such as creating a ticket and deleting a ticket. After that, it can be determined which areas of the UI have room to include some other functionality without becoming too complicated for users. | It is easier to start with a simple approach and add complexity later than include UI components for all functionality at the beginning and try to get rid of components to make it more intuitive. |
| UI component flow is succinct and displays required information without overcrowding  (user-friendliness) | Make sure that processes when interacting when the UI, like ticket creation, is user-friendly and makes sense in the context of the application. Only Include information that is required on each ticket like the ticket number, priority, and the ticket information. | User-friendly design speeds up the usage of the application and makes the overall workflow much more efficient. |

4 Design Details

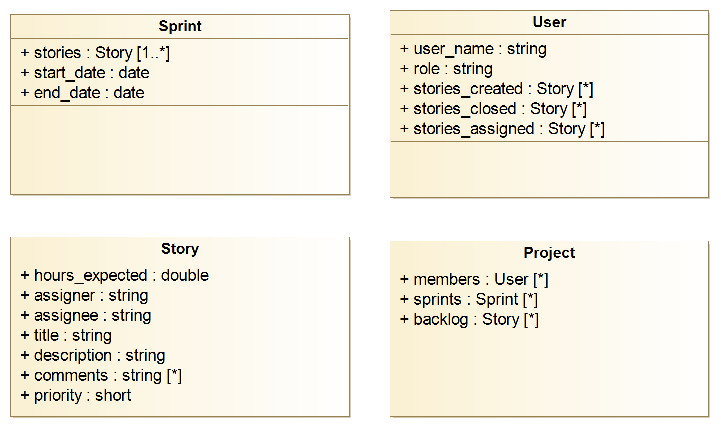
(a) Include class level design of the system (i.e. class diagrams) and be as detailed as you can.

**User:** user\_name: *string*, role: *string*, projects: *Project[]*, stories\_created: *Story[]*, stories\_closed: *Story[],* stories\_assigned: *Story[]*

**Project:** members: *User[]*, sprints: *Sprint[], backlog: Story[]*

**Sprint**: stories: *Story[]*, start\_date: *date*, end\_date: *date*

**Story:** assigner: *User*, assignee: *User*, title: *string*, description: *string*, comments: *string[]*, priority: *int, hours\_expected: double*

**

(b)

User class will include name, role, and assigned stories for the current user. Project class will include an array of sprints and an array of users who are working on the project. Sprint class will include a list of stories and the start/end dates. Stories will include all the details of a needed story (Assignee, description, priority/ETA, etc…).

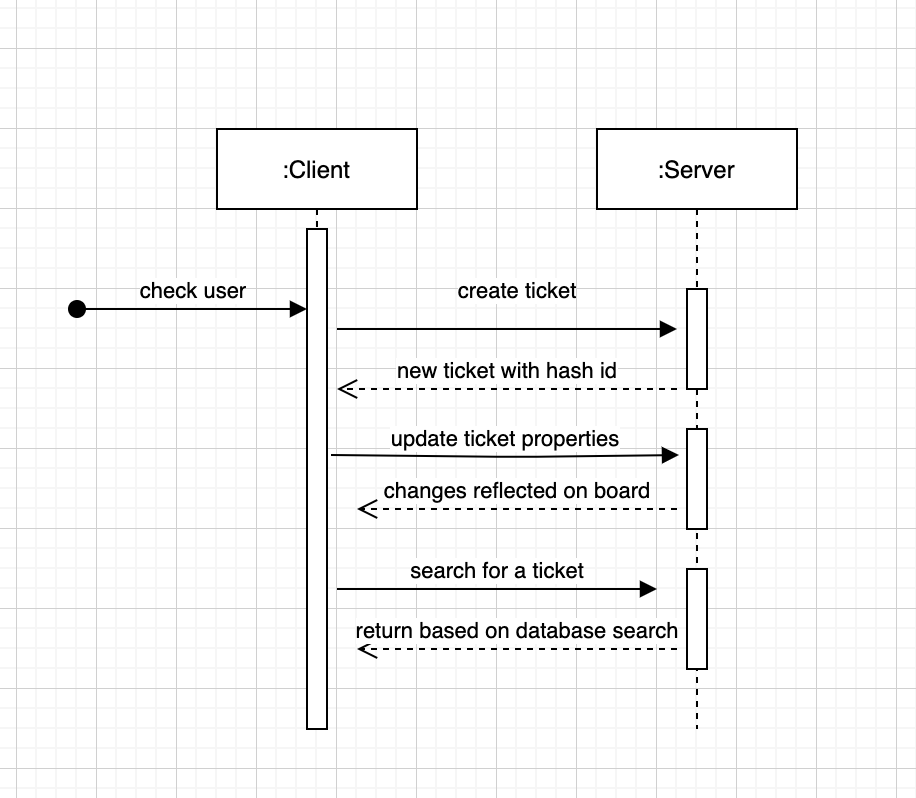
The way these classes will interact will follow a general sequence:

* After the user logs into the service and gets authenticated, we will get a User object from the database
* The user will choose which project to view and the service will get it from the Project array in the User object
* The user will choose which sprint to view which we will get from the Project object in the previous step
* The user will be shown a UI that will enable them to see current stories, their statuses, and all the other details provided in the Story objects.

Other ways that the classes will interact include:

* Adding stories that are created/closed by a user in a Project object to their User object, and also updating the story array in the Project object

(c) Add sequence diagrams for different activities in the system, which will be helpful at the later stages of your project.



(d) If necessary, try to also include activity diagrams (or state diagrams) and UI mockups.

