Scrum Space Dynamic UI Mockups

Overview

Note: Updated ER diagram has been included in the reports folder. Bolded data are new additions and we have also included the referencing and embedding colors.

Our updated mockups allow users to effectively navigate and use our web app. The app contains 4 main pages, with some important features nested into pages. Each team member continued their work on the feature they had previously mocked. We've built out most of the basic functionality, but it's not fully functional yet.

First, we'll go through an overview of some important design changes in the application:

Isomorphism

We've transitioned the rendering of our application to the server side. There were many motivations for this change, but the biggest of those motivations was speed. Server side rendering allows for an initial application state to be rendered on the server, which is then sent to the client (instead of a blank page) to finish up any additional rendering. This is practiced by many modern web applications, most notably Facebook.

Redux

Redux is a Flux-like data management framework, built on the concept of uni-directional data flow. All of the application's data lives in one single state tree (takes care of redundancy), and all interaction with this tree is abstracted into the Action -> Reducer flow (standardizes and simplifies data handling). Additionally, we've built in asynchronous server interaction using Redux-thunk, middleware that allows for the return of functions as actions.

To support the previous designs, we've made a couple of modifications to the offered starter application.

First, we're running a node server (in `src/server/server.js`) with some basic express routing. We've pushed all static assets to `/static/`, a virtual path prefix that separates asset access from the application view routes. Additionally, the server provides the server-side rendering functionality (courtesy of Abhay) using react's `renderToStaticMarkup`.

Secondly, because our application is initially rendered on the server (where your browser's Local Storage isn't available), we've pushed the initialization of the mock database to an

initDatabase function. This is of course temporary functionality, until we move server/database functionality to the actual server and database.

The other main changes you will notice is the lack of functions designed to read the mock database. Because the full state tree is provided asynchronously (in client/entry) we trickle down all relevant data to individual components using mapStateToProps from Redux. Because of this, you'll notice that React State is only directly being used for UI state changes, as opposed handling the application data as well.

Next, an in depth description of the following pages of our application:

1. Project View Page & Scrum Board & State/Database interactions
2. Dashboard, Client-side Architecture
3. New Sprint Creation
4. Task Detail View
5. New Project Creation
6. User Settings
7. Git Statistics (Honors feature)

Cut Features

1. Google Hangouts interaction: Our project already has many features that we would like to get working properly. In the off-chance we have time to add this feature, we will.

Project Master List View, Scrum Board, Redux and Database interactions (Dylan Fischler)

Responsible for building the master selection view, constructing and building out functionality of the scrum board, setting up Redux and interfacing it with the mock database, and writing our Gulp build script.

Project Master List View

This is where the user navigates to a project. The projects are organized in a clickable grid, displaying the Project Name, Scrum Time, and a dynamic graph displaying tasks completed over the recent days.

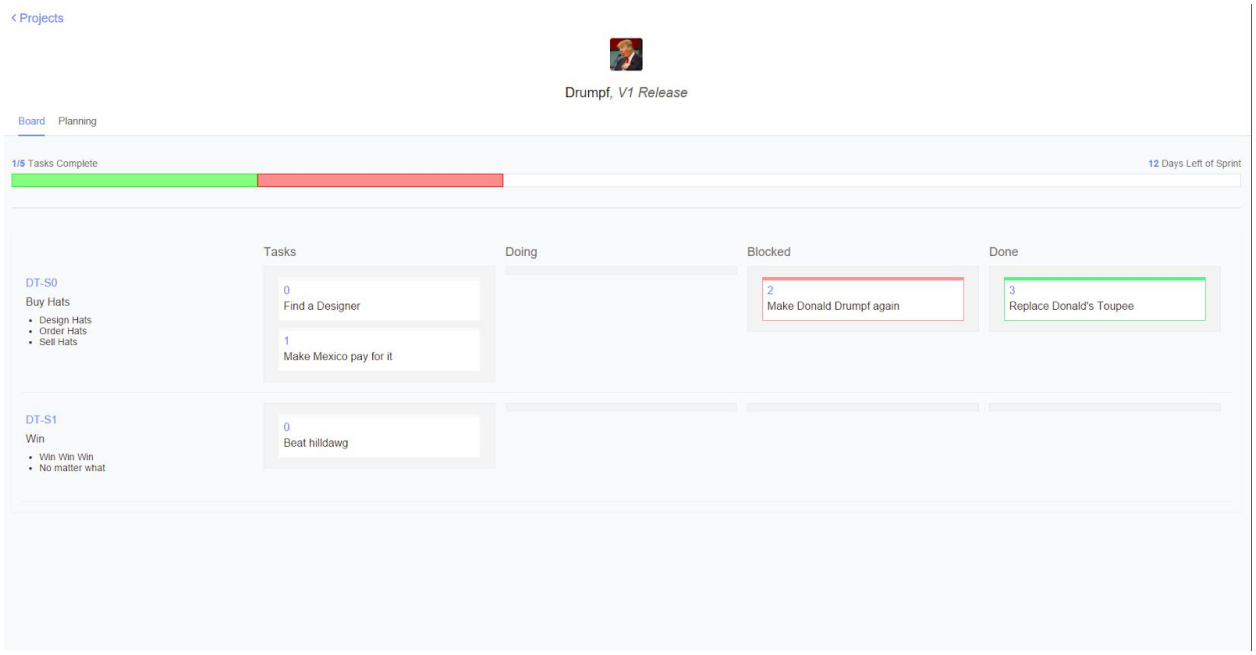
The ProjectList React component provides the mechanism for mapping ProjectItem components into a grid interface.

The ProjectItem component renders the individual grid item. The contents of the ProjectItem, including the dynamic task chart and project information are all wrapped in a react-router Link component.

Currently, the graphs rendered on the ProjectItems are powered by random data. We've already built in functionality to record the history of all task state changes, we'll just need to map this data correctly to the graphs.

Project Detail Page

The Project Detail page is a wrapper for the majority of the main interactions in the application. It dynamically renders the current sprint with a simple find function within the project sprints array, calculating which sprint's time box you're currently in. It's important to note that this functionality ignores the current_sprint field from the ER Diagram and the mock data. We recently saw the flaw in our logic behind this field and have moved to deprecate it from the application. The Sprint Planning section (Ryan's scope) still uses the current_sprint field, so therefore is not directly compatible with the Project Detail Page. This is being worked on by Ryan and will be compatible soon.



The ProjectNav component displays project information and state. Underneath, is a basic tab interface, providing the navigation between the different views in this page.

The main functionality on the page is made up of the sprint progress bar and the Scrum Board itself.

The Progress bar (ProjectProgressBar component) is a dynamic representation of the state of the sprint. The horizontal graph is divided into color coded sections, displaying the distribution of the tasks across the board's states (Unassigned, Doing, Blocked, Done).

The Scrum Board (ScrumBoard component) is the heart of ScrumSpace. It's a digital representation of the "whiteboard and post-it note" normally used in the scrum process. The board is divided into rows by each Story in the sprint (created in the Sprint planning process). The board is then divided into columns for each possible state a task can be in (Unassigned, Doing, Blocked, Done). The tasks are sorted out and rendered into the correct column.

A user interacts with the board by dragging tasks into different states (just like you would move a post it on the scrum board), a functionality built on top of the low level API's provided by React DnD.

While currently, a user can drag a task back and forth between columns, the planned functionality is to control how a task can transition between states. We can build this functionality by hooking our state transition logic into React DnD's canDrop function.

The only outstanding bug in the board is a UI discrepancy that allows for a task to be dragged to another story (this is not allowed in our scrum process). Luckily, this is only a UI issue. Once dropped, the task transitions inside of the correct story.

Outstanding issues:

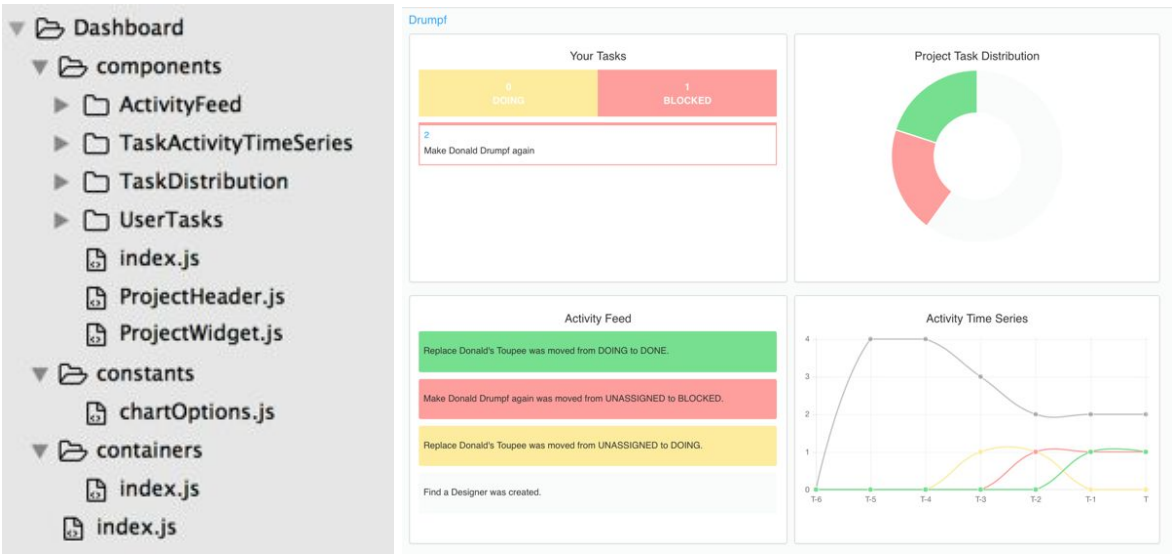
1. There is a problem in displaying stories, as of now all stories are displayed regardless of what sprint that they are in rather than specific to the current sprint.

Redux & Database Interactions

Our application data handling is completely powered by Redux, a state handler that enforces a Flux-like unidirectional data flow and an immutable state tree (the immutability is not enforced, just practiced, *for now*). As detailed in the general description above, Redux is provided all of the applications data in a single state tree. The data is mapped to a user specific state tree object with the stateTree function.

Dashboard, Client-side Architecture (Abhay Vatsa)

Responsible for creating the client side architecture (directory structure/organization, documentation, react router) and Dashboard, which is the loading screen of the application.



Dashboard has four major components that are displayed for every project in SPRINT status. These are organized into folders in it's subdirectory components (*ActivityFeed*, *TaskActivityTimeSeries*, *TaskDistribution, UserTasks*). The purpose of the component is to provide an up-to-date view of the state of user's projects in SPRINT. Since this is read-only component, it has no interaction to update the database, it merely queries the redux state tree.

- *UserTasks* shows tasks that are assigned to you that are not DONE.
- *ActivityFeed* shows a feed of task status changes from the scrum board.
- TaskActivityTimeSeries shows how many tasks were in different statuses at the start of each day in the last week of SPRINT.
- *TaskDistribution* shows how the project's tasks are distributed across statuses at the current time.

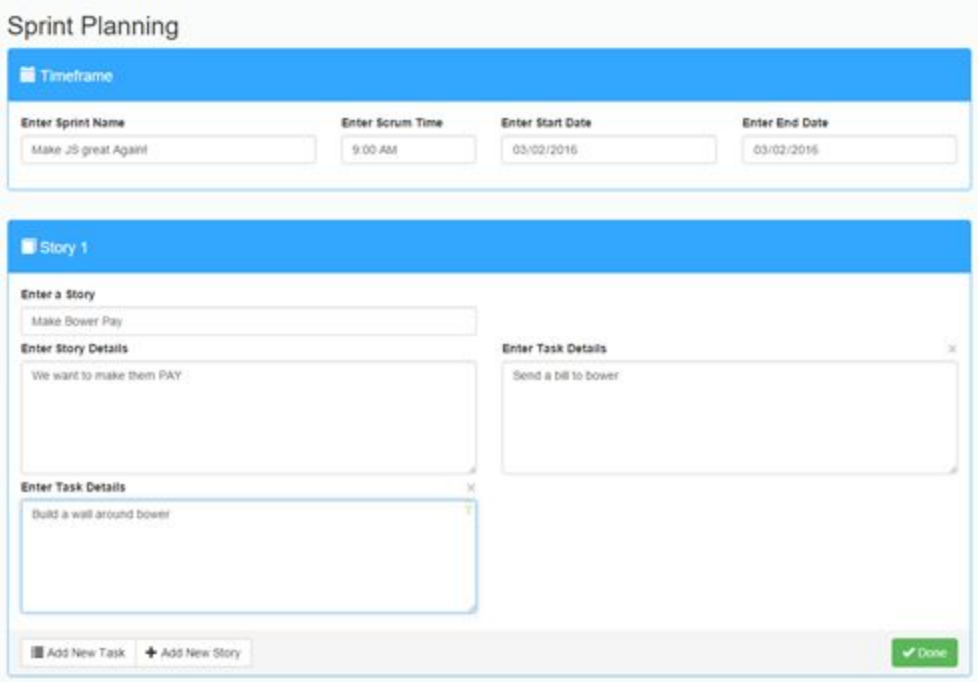
Some outstanding issues:

1. Styling needs to be improved.
2. Stretch goal is to have each component have some User Interactivity, example filtering in the 'Your Task' panel by task status.

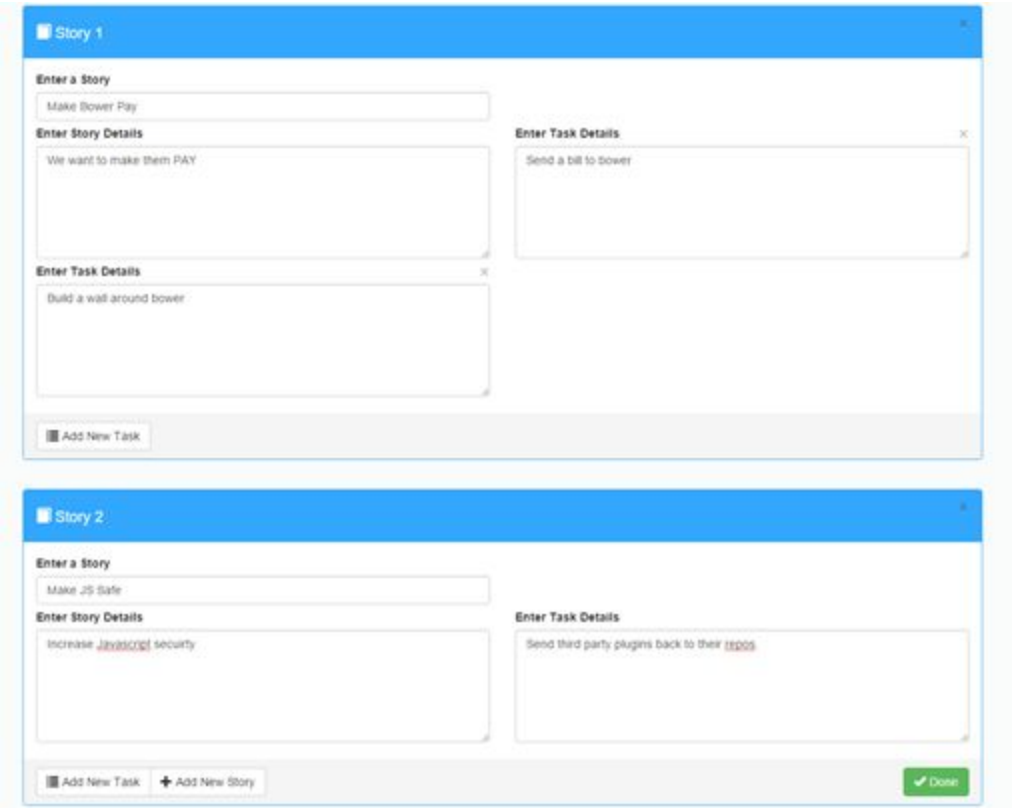
New Sprint Creation (Ryan Jerue)

During the scrum process, periods of works are organized into sprints. One such important feature as consequence is the ability to plan sprints. As each project has sprints, sprint planning is organized for being in individualized for projects. In the sprint planning area. Users will be able to plan their sprint's timeframe, stories, and each story's individual tasks.

Sprints do not need to be planned in one sitting; they may be planned over time. As a result, users may go back and edit things in their sprint. After hitting done, a user may return to sprint planning to find that what they entered before is already prepopulated into their sprint plan.



Users may dynamically add new stories to their sprints, and new stories to their tasks. Adding more stories and tasks effects where the done and add new story buttons are as well as the delete buttons.



The done button is responsible for writing the sprint's changed information to the database. Anything that is removed on the screen is also subsequently removed from the database. The page is very dynamic in its ability to add and remove things allowing the users maximum flexibility.

Writing to the server is done in the serverPostSprint function. The function serves as a one stop shop that is able to edit and create new sprints. First, the function creates a new sprint or updates the sprint with the information that is found in the timeframe panel, and then moves onto stories and tasks. Stories and tasks are stored separate to their sprints for future upgradability. The function will overwrite any stories and their successive tasks that are associated with the sprint being edited. Any empty stories and tasks will also be cleaned out.

Outstanding issues:

1. The creation of sprints requires a refresh in order to properly appear on the next screen. This is because redux is not properly mapping the updated state to the properties.
2. There is an alert button that appears when done is pressed. In the future, this will be a less intrusive modal as the alert button does not conform very well with the design of the application.

3. The developers came into making sprint planning with a very basic knowledge of react and javascript. In the future, the page may be redesigned to include drag and drop functionality.

Task Detail Views (Niharika Venkatathri)

Responsible for making the modals that display the details for a particular task.

The task detail modal would display the name of the task, the task ID, the story it belongs to, the description of the task and the members assigned to the task (if any). The modal currently displays correctly name of the task, the task ID, the story it belongs to, the description of the task and the members assigned to the task by retrieving it from the mock data. The portion of the functionality that does not work and need to be fixed before the next submission is the ability to add and remove members. Currently, there is a delete button next to each member's name in the modal. The functionality needs to be added so that on clicking the delete button, that particular member is removed from the array of members in the mock data. Another functionality that is missing is for member's names to be added to the task upon typing the name in the text box below the list of members. The name of the member needs to be taken from the text box and added to the array of members in the mock data, thereby updating the database. In summary, the part that is incomplete involves the functionality to update the database from the modal.
The plan to implement these features before the next submission is to add methods that would enable the deletion and addition of members. This involves having members be deleted on a button click event, and the members added once a name is typed into the text box.

Outstanding Issues

Currently there are issues related to this component that cause the application crash due to null and undefined references. As such, it is currently not included in the application's flow, but the code is included in the in the repository as it is important to the application's future functionality and purpose. It will be fixed in a later release.

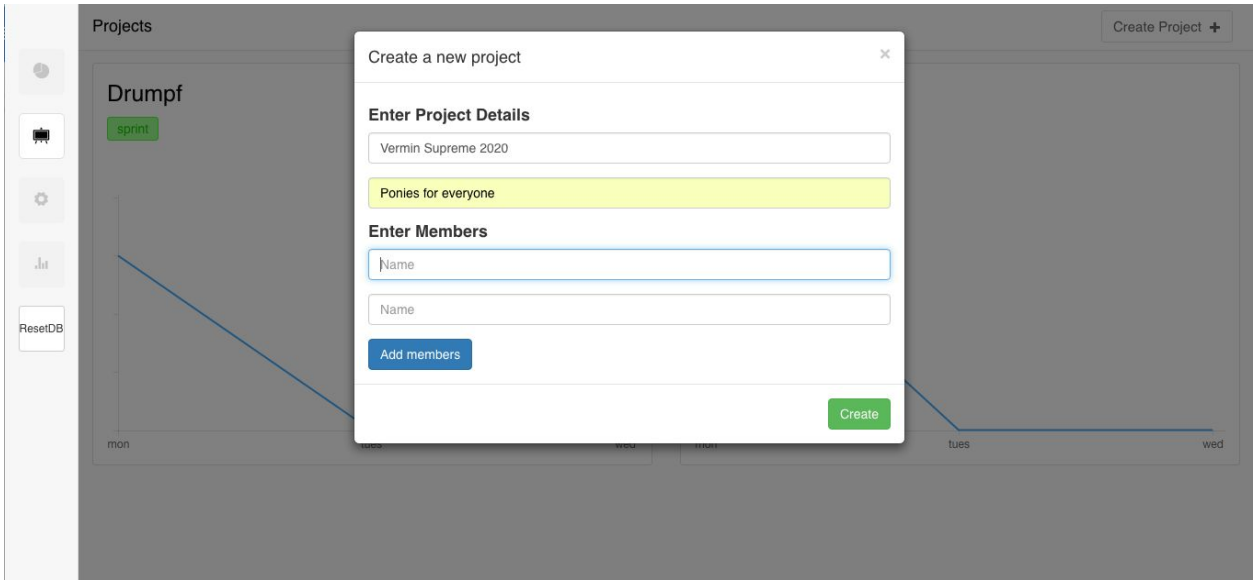
New Project Creation (Rachana Lingutla)

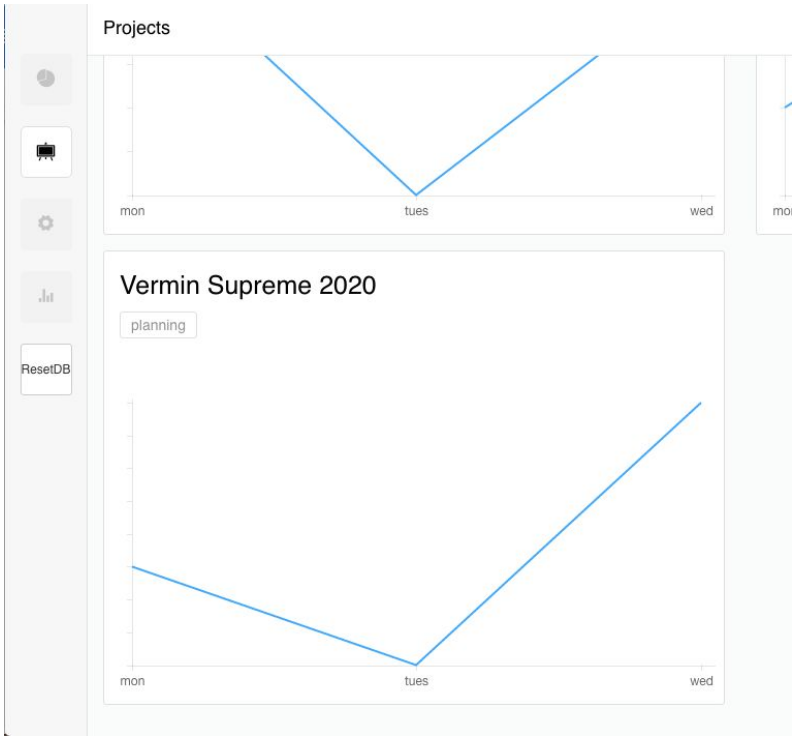
Responsible for making the project creation form dynamic. Users can now enter project names , descriptions and users that will be stored in the database.

Users can create new projects by clicking on a 'Create Project' button in the upper right corner of the 'Projects' page. Instead of putting the project form into a new page, we opted to keep the project creation in a modal. The point of our application is to keep project management simple. This involves keeping our pages to a minimum, and for a feature that won't be used as much is makes sense to *not* create a new page.

The modal pops open over the project page. Users can enter a project name, description and members to add to the project. Since we are writing to the database, the code for this modal includes functions to write to the database. The server.js file includes a function that writes simple data for a new project(the only data can be populated by the user are the title, description and members on the project).

The main server.js function involved in this portion of the app was serverPostNewProject(...project data), which wrote a new project to the database based on the info passed in. I didn't directly use the react-router as my feature doesn't involve linking between pages. Of course, there is a higher-level router that is responsible for rendering and linking the main pages. Since our app also involves Redux, functions were added to the 'actions'(functions that state what action we want to take on the state-tree) and 'reducers'(functions that state what effects we want to have on our tree) files.





Outstanding Issues:

1) Mainly figuring out how we are going to properly attach users to projects. The current modal allows us to enter and store members on a project, but we need a way to check that the user exists in the database (has an account). One method to go about this would be to have a pop-down list that matches user names as you type in a member's name. Ideally, the modal wouldn't let the project be created if the added members weren't in the database.
2) Resetting Modal: Idealistically, we want the entire modal to be reset if the user simply closes out of the modal. Currently, any written text in the title, description and members fields are being reset. But if extra member boxes are added, they won't disappear. I think the way to fix this would be to reset the state of the modal if it is closed out of, but I need to find a good way to do this.
3) Not all of the initial properties of a project are placed into the database. This will be fixed in a later release.

User Settings (Supriya Kankure)

This is the page where users can view and update their personal information. Originally, this page constituted with four panels: users, privacy, projects and external settings. As of now, the projects and external settings page are put on hold for the following reasons.

1) Viewing, editing, and deleting project information are operations that make more sense to be placed on the board where a user can view projects. The layout of this page makes its easier to search for projects as well.
2) The external settings page is too dependent on other third party technologies (email, hangouts, github, etc) that react and mock data alone cannot simulate. It thus is more reasonable to do these features towards the end if time and resources permit.

Currently the panels for updating user information and password work. Mock data is being routed, and displayed initially as a placeholder. When the user types in new data, and clicks save, then the mock data is updated to the new information. The privacy panel works the same way.

Outstanding Issues: Main issue is the data does not sync with database.

1) **Users and Privacy Panel:** The save button does not work and mock data is being pulled in. The privacy method does not check as of now if the old password is really the current password of the user.
2) **External Settings Panel:** As stated above, this is a feature of settings to be put on hold, but not required for the overall application.
    a) While toggling the enable/disable buttons could do something that signals change (i.e change color, confirmation message/alert) it does not do anything with mock data. This is thus meaningless. For example, clicking on enable or disable google hangouts, will only change the UI part of the application, but in-order to actually do this, we will need to authorize the users gmail, and create a link to the hangouts page. Similarly for github notifications, to enable and disable this feature, we will need to authorize the users github account in order to send push or email notifications to the user. This same idea applies to enable or disable email settings. Thus, this panel is put on hold, and is more of an add on/optional feature, that will require a great amount of use of third party APIS. Again, this panel is not dynamic. The code for these panels is present in the repository, it is just not in the application's current flow.

3) **Projects Panel**: this panel is currently removed and will likely be on the project master page.

Git Statistics (Rachana's Honors Feature)

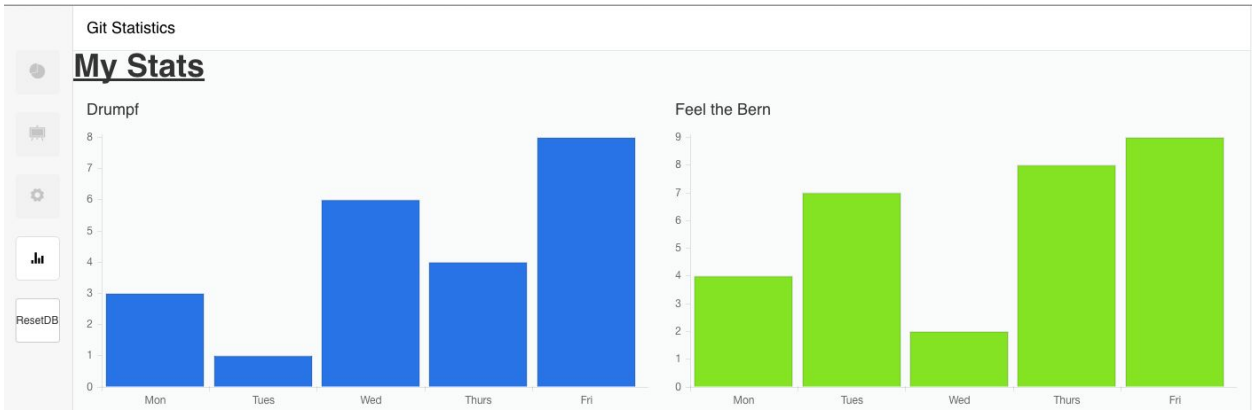
Responsible for displaying user and group statistics.

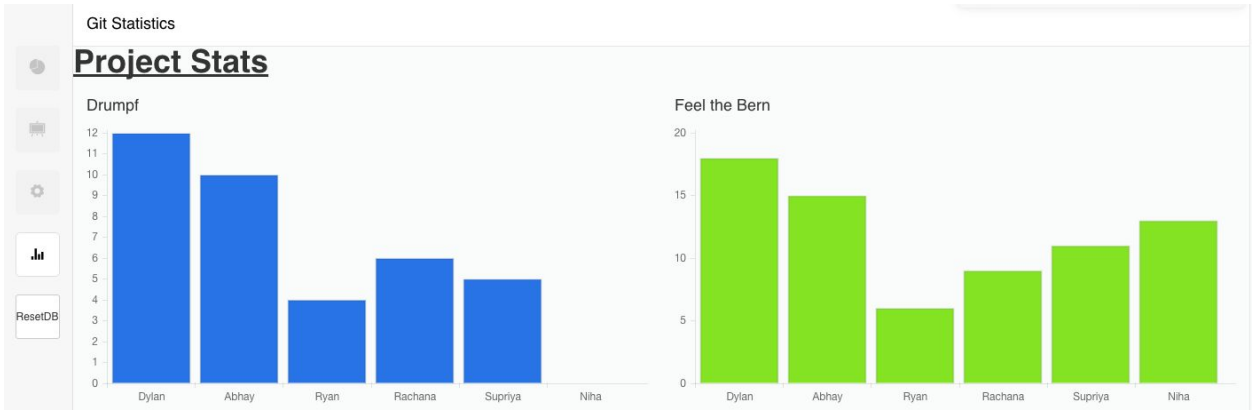
This read-only page allows users to track progress across projects. There are 2 sections(headers are hardcoded), one to display the user's statistics and another to display total(across all group members) contributions to a project. The feature extracts mock data from the database in order to populate the charts. It exists in its own page so that it won't directly interfere with a user's workflow. The charts are rendered with the 'chart.js' library. The charts in both sections are color coded according to projects(i.e. Project 1 has the same color in both graphs). The colors are randomly generated, and are reset every time the database is reset.

Since my feature does not involve interacting with multiple pages, there was no need to directly use any of the 'react-router' functions. It is also important to note that even though the instructions stated that the honor's features needed a server side function, our group's inclusion of Redux avoided this. Redux allows us to simply scan the tree for information we need using 'connect', a redux method that connects the React component to the Redux store of info. The parameters of 'connect' allow us to merge the properties of the data we need into the component we are using. This means that we don't need new server methods for simply reading in data.(Redux and it's data reading functions are also better articulated above)

I have also accounted for the addition of statistics when a new project is created. The 'new project' creation modal creates mock data and colors in order for the user to visually see what a more complicated display looks like.

I have also slightly modified the way I intend to store my data (and by extension modifying the ER diagram). For the individual stats section, the metrics of date and number of commits is still being used. The dates of the project are stored in an array that lies within the project data. There is a corresponding array with the same number of indices to keep track of commits by the day. But for the group commits, I have changed the metrics to the number of commits by person and the user name. This had been my initial intention, but it was not accurately reflected in the ER Diagram. I have updated the diagram to work with the slight changes. For the time being, I have also opted to store each member's individual data related to a project within the project, in order to make rendering the charts according to project color much easier.





Outstanding Issues:

1)New project graphs aren't rendering right away: Basically what's happening is when users create a new project and immediately go to the 'Git Stats' page, an empty graph with the proper title is created in each section. The mock data doesn't load until the page is refreshed. There is an issue with the actions and reducers page but it isn't clear what it is. Closer examination at the functions related to project creation will help resolve this issue.

2)Ideally I would like to move the personal commits data back into the 'user' data. In the GitList.js file, where I am retrieving my data from the tree, I ran out of time to figure out how I would extract data from 2 items that were not nested within each other. Basically if I can figure out how to do this, I will move the data into a more logical spot.

3)Also need to figure out how to properly access the user display names of the people working on a project. Right now, I just inserted our ScrumSpace group members names as members of the projects in the database for display purposes. This will be easier to fix once we have more users and have more data to test on.

Updated ER Diagram