Gitrack

Documentation

Submitted in Partial Fulfillment of the Requirements
in CMSC 129 Software Engineering 2

Miana, Jose Roy C.
Po, Justin Andre E.
Sullano, Yul Vincent E.
Tumulak, Patricia Lexa U.
Valles, Oscar Vian L.

Contents

1 Introduction

1.1 Problem or Opportunity

Project manager and developer communication is very important in executing a project. Good communication between these two enables better estimation of time and amount of work done. Various communication and project management tools exist to keep track of the amount of work done — e.g. Github Projects, Trello, Scrummate, and Jira. However, these tools do not have an integrated way of viewing exactly what the current progress of a task is — in the context of development.

Checking the current commits in a feature branch lets the developers know the current state of the code base. However, this information is not easily digestible by the project managers that do not have technical backgrounds.

1.2 Solution

Gitrack makes it easier for both. A web application serves as the task board for the project managers and developers. This contains cards for every feature that needs to be built. For every commit on a specific feature branch, a resulting card on a board gets updated. If that feature is now complete, this card will then be updated accordingly and moved to the correct column.

2 Technical Documentation

2.1 Architectural Diagram

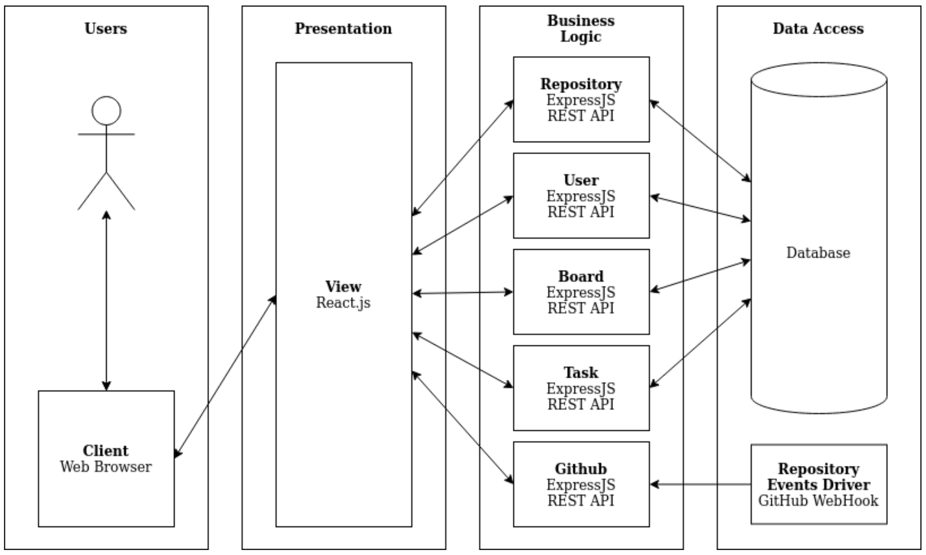


Figure 1: Architectural Diagram of GiTrack

The architectural style chosen is the layered architecture style. This was chosen because the application being developed is a web application, and the nature of most web applications is layered. Gitrack is no different.

This architectural style presents the following advantages: it is modular, it is maintainable, scalable, and flexible. This allows the team to develop the application in groups, and allows the application to start small, and expand later on without much difficulty. Issues and failures can also be easily addressed and recovered from if only a single layer fails.

The deepest layer is the Data Access layer. This is a database that is powered by SQLite. This layer stores all of the data for the application. This includes all of the tasks, repositories, boards, and others. The Repository Events Driver is a GitHub webhook that sends data to the business logic layer whenever an event happens on a connected repository

The business logic layer handles sending and receiving data to and from the Presentation layer using a REST API. This layer also serves and manipulates data that is stored in the database. This layer will be implemented using ExpressJS

This layer contains the majority of the components of the applications: repository, users, board, card, and columns. The Repository component handles connection with the GitHub API, and handles pertinent information like authentication tokens and branches available. The Users component handles authentication, and access. This component manages the information about the user. The Board component manages pertinent data about the board, including the connected repositories and the columns that the board has. The Column component manages each column, handling their order and the tasks contained inside. The Tasks component manages the tasks, this component presents the latest commits, the assigned developers and project managers, and all other pertinent info. This component also manages the connected branch and repo to the task.

The presentation layer exposes parts of the program to the users to interact with. This allows the users to manipulate the cards, add columns, and connect to the repository. The view portion of the presentation layer will be created using ReactJS.

2.2 Frontend

GiTrack's frontend was built with *React*, an open-source frontend JavaScript library for building user interfaces. This library was chosen because of its flexibility, reusability, reliability, great community support, speed, and the vast amount of libraries that can be integrated with it. *Redux* was utilized as the global state container. This is used for storing information that is to be used throughout the entire application (e.g. user information, access token, refresh token), minimizing the number of API calls. *Axios*, a promise-based HTTP client, was used to communicate with the backend by handling the many different HTTP requests that the frontend needs. In order to properly organize API calls we used a concept known as *services*, wherein all related endpoints are grouped together and are assigned names to make them easier to remember.

```
const BASE_URL = "/auth";
const AuthService = {
  login: ({ body }) => axios.post(`${BASE_URL}/login`, body),
  register: ({ body }) => axios.post(`${BASE_URL}/register`, body),
  logout: ({ body }) => axios.post(`${BASE_URL}/logout`, body),
};
```

Figure 2: Code snippet of AuthService.js

2.3 Backend

GiTrack's backend was built using *Express*. Express is a web application framework that allows for creation of REST APIs with robust routing and high performance. Important additional dependencies include *bcrypt* and *jsonwebtoken* for authentication, *@octokit/request* for GitHub API requests, and *sqlite* and *sqlite3* for communicating with the database. All of these run on top of *Node.js v15*.

The backend follows the standard REST API standards where different methods, such as GET, POST, PATCH, PUT, DELETE are used to specify the specific action that is requested. Information about the specific request can be found in a couple of different places based on the type of request. GET methods usually have information pertinent to the request in the URI or as Query parameters. As an example, the id of the board is placed in the URI when asking for information about that board: `GET /board/id`. POST methods have information about the request in the Request Body.

Schema and documentation for the backend can be found here: https://api.gitrack.codes/docs/

2.4 Database

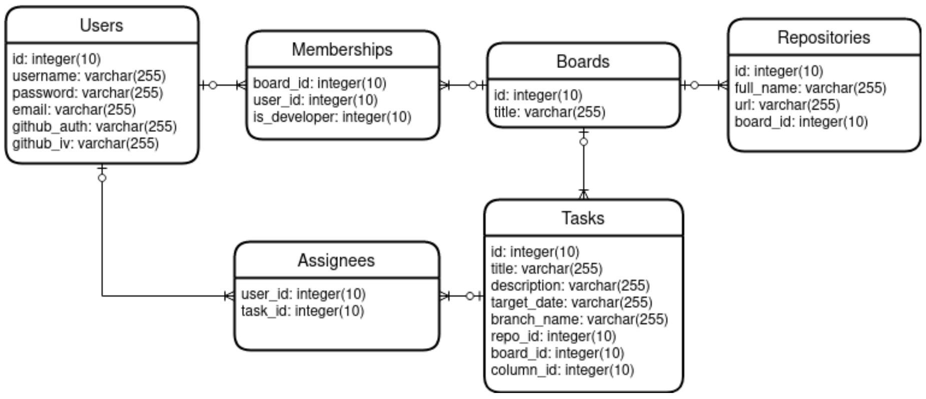


Figure 3: Entity Relationship Diagram

The GiTrack database uses SQLite and consists of six (6) tables, namely: *Users*, *Memberships*, *Boards*, *Repositories*, *Tasks*, and *Assignees*. The Entity Relationship Diagram (ERD) in Figure 3 shows the columns in the tables and their corresponding data types.

The *Users* table lists the users' data and keys them with an automatically incremented and unique id. Users' usernames, passwords, and email addresses are stored under the `username`, `password`, and `email` columns respectively. There can only be one account registered to an email address, thus the values in the `email` column must be unique. The same applies for usernames. The `github_auth` and `github_iv` columns default to `null` on user registration and are only given values when the user authenticates GiTrack to access their GitHub information. The auth token stored in `github_auth` is encrypted and the value stored in its respective `github_iv` column is used to decrypt it.

The *Boards* table lists the project boards in the GiTrack system. The only columns in this table are the id, which is the primary key, and the board title. The id of each board is used in other tables to identify the board of which they are a part.

The *Memberships* table has three columns and no primary key. The columns are `board_id`, `user_id`, and `is_developer`. The `board_id` column specifies the board of which the user is a member and the `user_id` identifies the user based on their unique id. The `is_developer` column contains flags in the form of 1s and 0s to indicate whether that user is a developer or a project manager for that board, where a value of 1 would indicate that the user is a developer and a value of 0 would indicate that the user is the project manager.

The *Repositories* table lists the repositories used in the GiTrack system. They are differentiated using the unique `id`, which is the primary key. A repository's full name, acquired from the GitHub API, is stored under the `full_name` column. The URL to the repository is stored under the `url` column. Finally, the board to which the repository is assigned is indicated with the board's id under the `board_id` column.

The *Tasks* table is the table with the most columns. Columns under this table are `id`, `title`, `description`, `target_date`, `branch_name`, `repo_id`, `board_id`, and `column_id`. The `id` column serves as the primary key which differentiates task objects. The `title` column contains the title of the tasks and the `description` column contains a more in-depth description of the task to be done. Task deadlines are stored as strings under the `target_date` column. The branch where development of the task is specified in `branch_name,` which is acquired from the GitHub API. The branch has to be a part of the repository specified by the `repo_id` in the *Repositories* table. In turn, the repository specified by `repo_id` has to be assigned to the board specified by the `board_id`, which is the board wherein the task is assigned. The values in the `column_id` column indicate the status of the task and is one of three values: 0, 1, or 2. A value of 0 indicates that work for that task has not been started. A value of 1 indicates that work for that task is currently ongoing. Finally, a value of 2 indicates that work for that task has been completed and the task's branch has been merged to the master branch of the repository.

Finally, the *Assignees* table has only two columns and no primary key. The `user_id` column contains the id of the user assigned to the task specified by the `task_id`. Users can only be assigned to tasks if they are within the same board.

2.5 Deployment

Deployment was done using a single *DigitalOcean* droplet that runs Ubuntu 20.04 (LTS) x64. It has 1 vCPU and a 25GB Disk. *NGINX* was used as a web server that serves both the frontend and backend. The frontend was served normally, the backend was served using NGINX as a reverse proxy pointing to `localhost:3000`.

The domain, `gitrack.codes` was bought on name.com. The nameservers of the domain were pointed to the nameservers of DigitalOcean. This was done so that management of the server and domain all reside on DigitalOcean.

SSL encryption was made possible through Let's Encrypt and their program *Certbot*. Three domains were included in the certificate: `gitrack.codes`, `www.gitrack.codes`, and `api.gitrack.codes`.

2.6 Signup

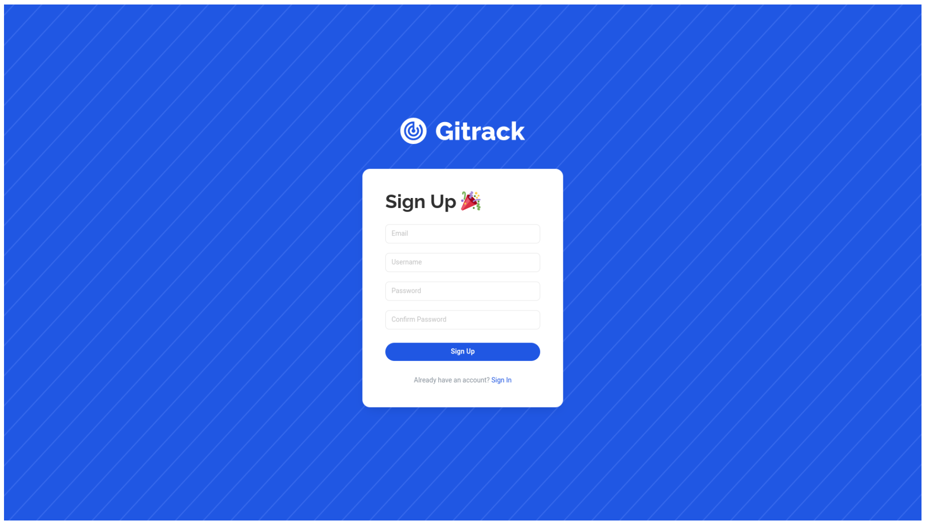


Figure 4: Signup Page

2.6.1 Description

The Signup page, accessed through a link from the Login page, is where one can create an account for Gitrack. The creation of an account only requires the user's email address, their username, and their password. Before submitting their signup credentials, the user would need to confirm their password.

2.6.2 Frontend

The Signup page was built upon a template page called `LoginSignup` wherein the background color and pattern was set. This is also where the routes redirect between the Login page and the Signup page. This was implemented using the `BrowserRouter` from the *React Router DOM* library.

For the Signup page itself, it's main components are, as mentioned above, the template page LoginSignup, the Gitrack logo in white, and a widget called LoginSignupCard which is used in both the Signup and Login pages. This widget consists mainly of the title in large text at the top of the card, the input fields, and the submission button (Sign Up for this page's case).

Each field requires the corresponding input (Email, Username, Password) in order to Sign Up. Once the Sign Up button is clicked, it triggers the `onSubmit` function which first checks the inputted passwords. Passwords are hidden during typing and passwords inputted within the Password and Confirm Password fields must match. If not, a "Passwords must match" error message appears under the Confirm Password field. The `register()` function from `AuthService` is then called. `AuthService` is where the different HTTP requests for each endpoint needed to login, signup, and logout are written. The `register()` function is a POST request to `/auth/register` that sends the email, username, and password fields as the body. If the server response indicates any errors such as missing fields, it is caught in a `catch` method and an error message is called indicating the corresponding error. If `register()` was successful, the `login()` function is then called with the username and password as the request body. The global state is then updated with the information of the user.

2.6.3 Backend

This was implemented through a function `registerUser()` which accepts the parameters `username`, `password`, and `email`. The function encrypts the password using *bcrypt* to keep it secure. If hashing the password fails, the function throws an error message `HASH`, which indicates there was a problem with hashing. If hashing the password succeeds, the function will then check if the username already exists within the GiTrack database. If the username already exists, an error message `DUPLICATE_USER` will be thrown. If the provided username passes the existence check, the function will then check if the email exists within the GiTrack database. If the email is already in use by another user, an error message `DUPLICATE_EMAIL` will be thrown. When all the checks have succeeded, the function will try to insert the user information into the Users table. If, for any reason, inserting the values fails, an error message `INSERT_FAILED` will be thrown and the error details will be logged in the console.

The endpoint for this function, `/auth/register`, receives a POST request which contains the user's `username`, `password`, and `email` in its request body. The endpoint returns status code 400 if the request body does not contain values for all of these data, together with error messages `MISSING_USERNAME`, `MISSING_PASSWORD`, or `MISSING_EMAIL`. If the request body is complete, it proceeds to try and register the user with the given credentials. If either the username or email is already in use, the endpoint returns status code 409 with the error message `DUPLICATE_EMAIL` or `DUPLICATE_USER`, depending on which gets thrown first in the `registerUser()` function. If `registerUser()` fails for any other reason, it returns status code 500 with the error message thrown. If `registerUser()` succeeds, the endpoint returns status code 201 and an empty error message to tell the frontend that registration was successful.

2.7 Login

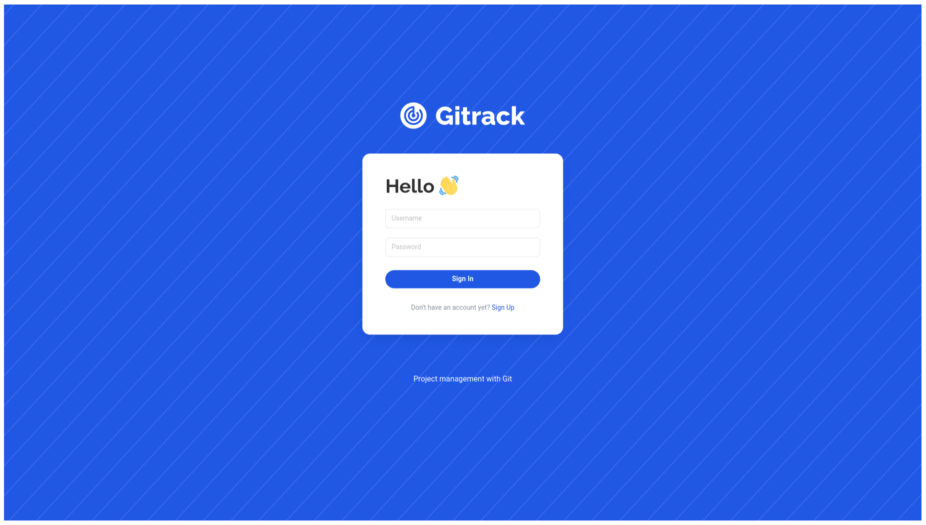

Gitrack
Hello 👋
Username
Password
Sign In
Don't have an account yet? Sign Up
Project management with Git

Figure 5: Login Page

2.7.1 Description

The Login page is the first page that greets the user once they enter the website. It is how they are able to access the rest of the site by logging in if they have an existing account or to access the Signup page if they don't. It also features a little description of what Gitrack is for — *"Project management with Git"*.

2.7.2 Frontend

The Login page was built upon the same `LoginSignup` page. The only difference between the Signup and Login pages in terms of the interface are the title, and number of input fields. The only input required is the Username and the Password. The submission button is now a "Sign In" button on this page.

If any input fields are missing, an error message is displayed indicating the missing field required. Once the Sign In button is clicked, an `onSubmit()` function is triggered that first calls the `login()` function from `AuthService`. `login()` is a POST request to `/auth/login` that sends the username and password as the body. On successful login, the user is granted access and refresh tokens and the global state is updated. If there are errors in either the username or password or both, an error message is displayed indicating "Incorrect username or password."

A `watch()` method is used to listen to any changes made to the username or password. Errors are cleared if there were any changes made.

2.7.3 Backend

This is implemented using the `loginUser()` function which accepts the parameters `username` and `password`. The function first searches for the given username in the database. This search will return `undefined` if there is no such `username` in the database, and the function will throw an error message `USER_NOT_FOUND`. If the search does not return `undefined`, it will return an object with properties `username` and `password` with the user's unique `id`, `username` and encrypted `password`. The function will then compare the hash of the `password` argument and compare it to the user's `password` in the database. For security reasons, the function will throw an error message `USER_NOT_FOUND` if the password entered by the user does not match the password in the database after encryption. The function will throw an error message `HASH` if the password check fails for other reasons. Finally, the function will return the user's unique `id` if all the credentials match.

The endpoint for this, `/auth/login`, receives a POST request which contains the user's `username`, `password`, and `email` in its request body and returns a JSON with the properties `id`, `username`, `access_token`, `refresh_token`, and `error_message`. If either of these values are missing, the endpoint returns status code 400 with error message `MISSING_PASSWORD` or `MISSING_USERNAME`. If both of these are present, the system will try to log the user into the system using the `id` returned from the `loginUser()` function. This id will then be used to sign a JSON Web Token (JWT) and grant the user an access token and a refresh token. The refresh token is added to the list of refresh tokens for every user who has logged in, which will be checked if the user would request for a new access token, and deleted upon logging out. If `loginUser()` throws the `USER_NOT_FOUND` error message, meaning the entered password does not match the password in the database, the endpoint returns a status code 403 with the JSON containing null values for all properties except the `error_message` property, which will be `USER_NOT_FOUND`. If either the access token or refresh token signing fails, the endpoint returns a status code 500 with a JSON similar to the one returned with code 403, with an `error_message` corresponding to the error thrown by JWT. If no operation fails, the endpoint returns a status code 200 with a JSON containing the `id` of the user, the user's `username`, their `access_token`, their `refresh_token`, and an `error_message` of null.

2.8 Main Dashboard

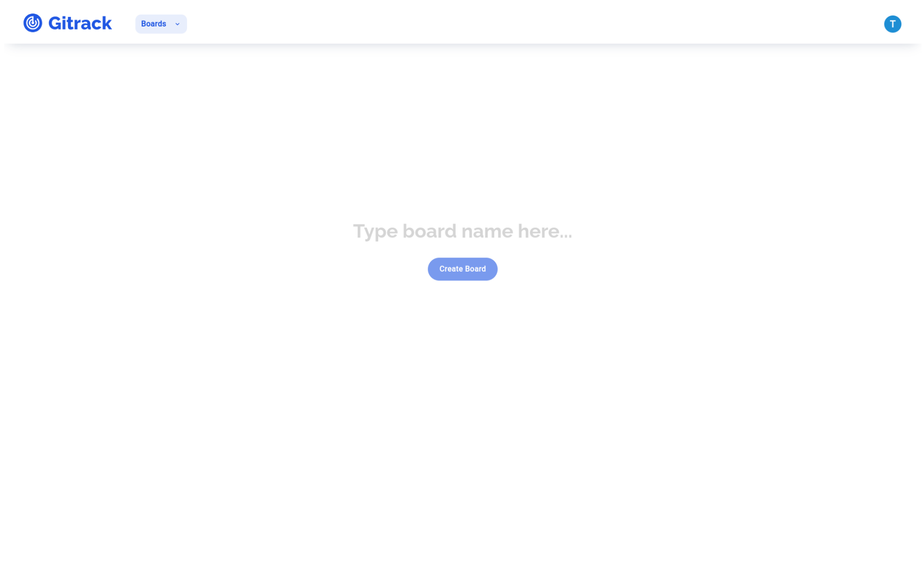


Figure 6: Create Board

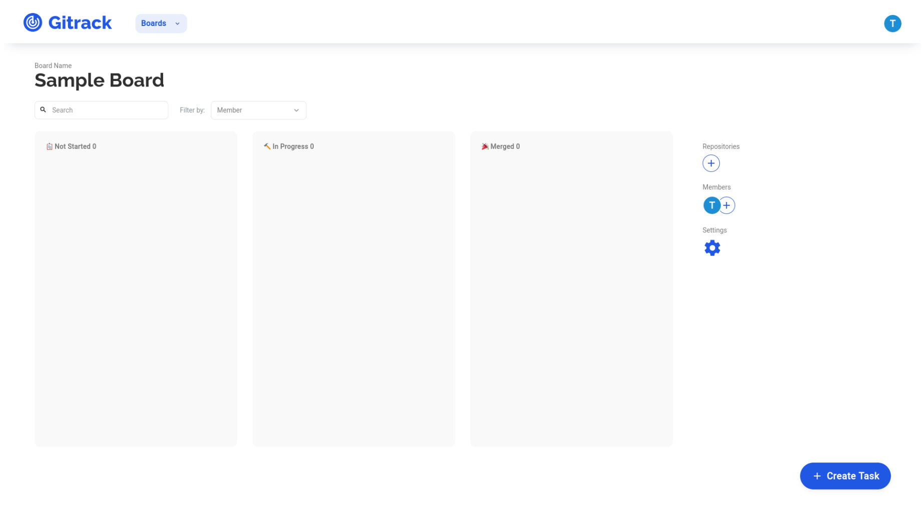


Figure 7: Main Dashboard

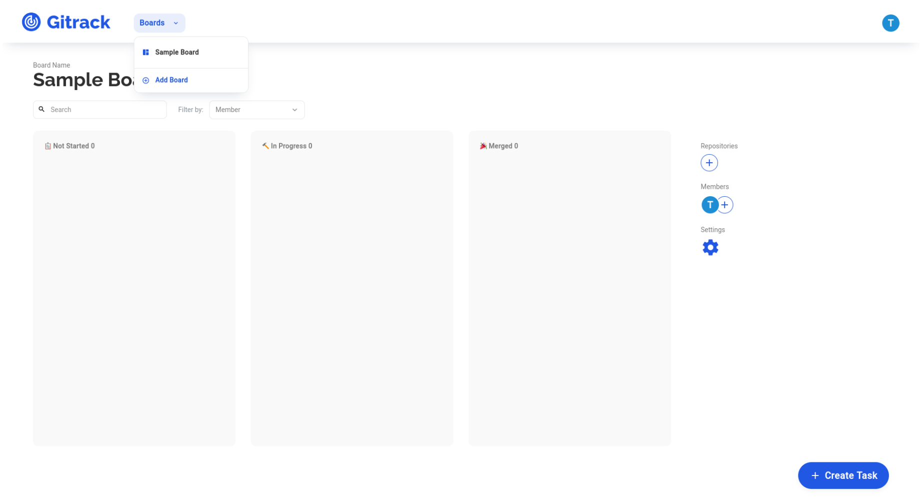


Figure 8: Boards Dropdown

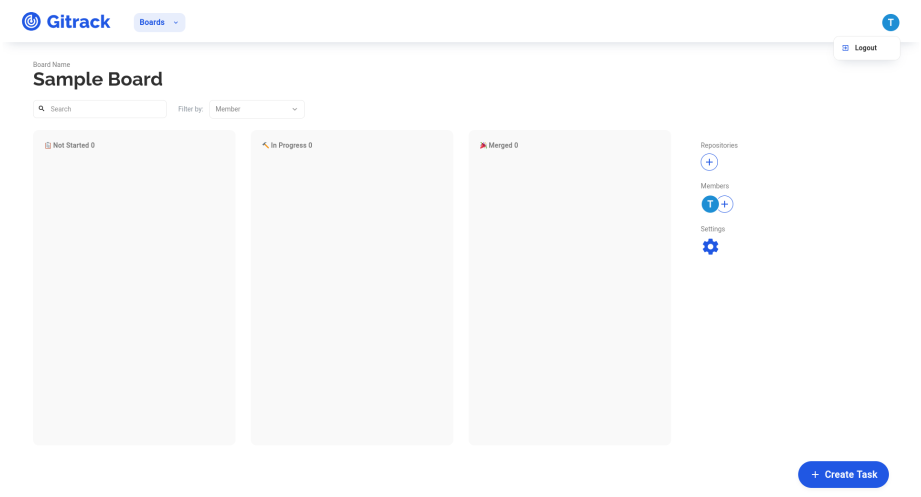


Figure 9: User Dropdown

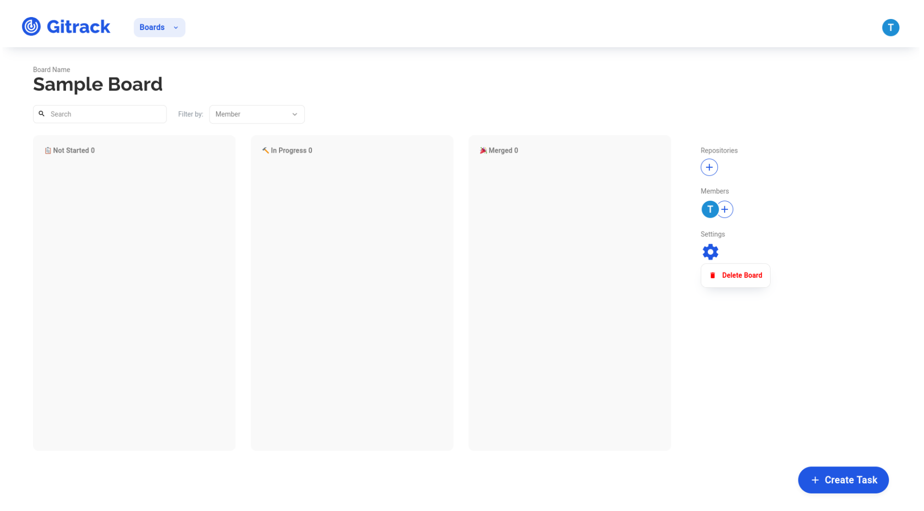


Figure 10: Delete Board

2.8.1 Description

The Main Dashboard is the core component of the application. This gives the user access to five main features: creating a new board, viewing information related to the board (e.g. tasks, repositories, and members), navigating to other boards, logging out of their account, and deleting the board they are currently on.

2.8.2 Frontend

The Main Dashboard is wrapped in a parent route known as board, which handles the navigation of routes that share the same UI elements as the dashboard. This makes sure that the navigation bar is rendered in all of the screens that require it.

Create Board Once the user has finished signing up, they will immediately be redirected to the Create Board screen. They are simply presented with an input for the board name and a button to create it. When the "Create" button is clicked it will trigger an `onClick()` function that would call the `create()` function from the `BoardService`. `create()` is a POST request to `/board` that receives the board's name as its body. If successful, it would then call the `refreshBoards()` function which would retrieve the new list of boards for the Boards Dropdown and redirect them to the url of the new board.

Dashboard When the user is redirected to the new board, the `retrieveTasks()`, `retrieveRepos()`, and `retrieveMembers()` functions from the `BoardService` are simultaneously called once the component is mounted. All three functions are GET requests that retrieve the information needed for the board. If a repository has already been linked to the board then the branches endpoint from `GithubService` is then called to retrieve the different branches of the linked repositories. Once all of the necessary information has been retrieved, the frontend would then display the user interface that contains the board's name, its members, its repositories, and its tasks, which are wrapped in three different columns representing the progress of each task. The user could also search for specific tasks and even filter tasks by assignee using the input and dropdown above the columns respectively.

Navigation Bar The navigation bar at the top of the screen contains two components that allow the user to navigate their way through the application. First is the Boards Dropdown, which contains the list of boards that the user is currently on, as well as the ability to add a new board. Clicking on any of the boards would redirect the user to the selected board. The second one, the User Dropdown, allows the user to simply log out of their account when necessary by clicking the "Logout" button. Clicking the button would trigger a call to the `logout()` function from the `AuthService`.

Delete Board Finally, the Main Dashboard component also allows the user to delete the board they are currently on. Once the user clicks the gear icon, it would display a dropdown that includes the option to delete the board. If clicked, a modal would be displayed asking for confirmation, and once confirmed, would call the `remove()` function of `BoardService` and refresh the list of boards in the Boards Dropdown.

2.8.3 Backend

Create Board This component was implemented through `createBoard()` function which accepts the parameters `title`, and `userId`. To create the board, the title is inserted into the *Boards* table. The user is also inserted into the *Memberships* table along with the board it corresponds to, and it's `is_developer` is set 0 which means that the user is not a developer. If successful, it then returns the last ID of the *Boards* table. Otherwise, throws an error message of `INSERT_FAILED`.

For its endpoint, `boards/`, receives a POST request where it accepts title from the request body, and `id` from the user. It also returns a JSON with keys `id`, `title`, and `error_message`. It first checks if the title is `undefined.` If it is, then it responds with 400 status code, returns `null` value for the JSON keys, and an error of `MISSING_TITLE` for the `error_message` key. If the title is defined, it then tries and calls `createBoard()` function. If the board was created or inserted into the database successfully, it returns a JSON with `boardId` for the id key, `title` key, and a null `error_message` key. If unsuccessful, it responds with 500 status code, returns a JSON with null value on its keys, and the error thrown by the `createBoard()` for the `error_message` key.

Delete Board This was implemented using the `deleteBoard()` function which accepts the URI parameter `id`, which specifies the id of the board to be deleted from the database. This function deletes all data related to the board specified by `id` across all tables. `deleteBoard()` does this by making five sequential delete queries to the database in order to delete all rows where `board_id` equals `id`. It first deletes all repositories linked to the board from the *Repositories* table. It then deletes all assignments in the *Assignees* table where the assigned tasks in the *Tasks* table are part of the board. Following this, all tasks in the *Tasks* table which belong to the board are deleted. It then removes all the users in the *Memberships* table if they are part of the pard. Finally, it deletes the board from the *Boards* table. If any of these queries fail, `deleteBoard()` throws an error message `DELETE_FAILED`.

This endpoint, `boards/:id`(\\d+), receives a DELETE request where the request parameter is the `id` of the board to be deleted. This endpoint returns an object that contains the `id` of the board, the board's `title,` and an `error_message`. It also makes use of `userId`, the unique id of the user. If the `id` request parameter is missing, the endpoint returns a status code 400 and an object with null values for the `id` and `title` and an `error_message` of `MISSING_ID`. It then checks if the current user has permission to perform this delete operation using the `getPermissions()` function, which checks if the current user is a developer or a project manager. If the user does not have permissions, the endpoint returns a status code 403 with an object with null values for the id and title and the `error_message` thrown by `getPermissions()`. The endpoint then checks if the board exists using `getBoardById()` and stores its returned board object in variable `board`. the board variable receives an object with properties `id` and `title`. The endpoint then tries to perform `deleteBoard()`. If `deleteBoard()` fails, it returns status code 500 and an object with `null` values for the `id` and `title` and the `error_message` thrown by `deleteBoard()`. If `deleteBoard()` succeeds, it returns status code 200 and an object with the value of the `id` property of the object stored in board, the value of the `title` property of the object stored in board, and an `error_message` of `null`.

Get All Tasks In Board This functionality was implemented by the function `getTaskinBoard()`. This function has one argument, `boardId`. This argument is used to get all tasks in the *Task* table that has its `board_id` set to `boardId`. Once all tasks has been retrieved, all assignees of the tasks are then queried from the *Assignees* table. Once all assignees have been found, these information is then returned. If for some reason these queries have failed, an error, *TASK_NOT_FOUND* is thrown.

The endpoint of this is located at `/boards/:id(\\d+)/tasks` and is requested with a GET method. This endpoint has `id` as a request parameter and uses `userId` from the request user. If `id` is missing, the endpoint immediately returns a status code of 400, nulls other keys and sets `error_message` to be `MISSING_ID`. If `id` is present, permissions of the user are checked using `getPermissions()`. If permissions are not enough the endpoint returns a status code 403 with an object with null values for `tasks` and the `error_message` thrown by `getPermissions()`. If permissions are enough, `getTasksInBoard()` is called with `id` as an argument. The endpoint then returns `tasks` and `null` for the `error_message`. If there were errors during querying the tasks, a status code of 500 is returned and the pertinent error code is sent in `error_message`.

Get All Members In Board `getMembersInBoard()` is used to implement this functionality. This function has `boardId` as its parameter. This function gets the `id` and `username` of all users found in the *Memberships* table that has their `board_id` set to `boardId`. If there were issues querying this information, `GET_FAILED` is thrown.

The endpoint to access this data is `/boards/:id(\\d+)/members`. This endpoint checks if `id` is defined in the request parameters. If it is not, the endpoint immediately returns a status code of 400, nulls all keys in the return body, and sets `error_message` to be `MISSING_ID`. If it passes this check, the `id` is then validated to determine if `id` points to an actual board. If not, a status code of 400 is returned, all of the keys in the return body are set to null, and `error_message` is set to `NOT_FOUND`. After all those checks were passed by the request, `getMembersInBoard()` is called. The return body has `board_id` set to `id`, `members` set to the return value of `getMembersInBoard()`, and `error_message` is set to `null`.

Get Repositories Connected to the Board `getRepositoriesInBoard()` returns all repositories that are connected to a board. This function receives `boardId` as an argument. A query is then done on the *Repositories* table to get all rows where `board_id` is set to `boardId`. These rows are then returned.

`/boards/:id(\\d+)/repos` is the endpoint that calls this function. This endpoint has `id` as a request parameter, and uses `userId` stored in the request user. First, `id` is checked to make sure that it is set. If it is not, then a status code is set to 400, all keys in the response body are set to null and `error_message` is set to `MISSING_ID`. Afterwards permissions are checked using `getPermissions()` as usual. If permissions are not enough the endpoint returns a status code 403 with an object with null values for `members` and the `error_message` thrown by `getPermissions()`. If it passes both checks, `getRepositoriesInBoard()` is called with id as the argument. The return value of this is then set as repos in the return body. `error_message` is also set as `null`. If any other errors occur, the endpoint returns a status code of 500 and `error_message` is set to the error message.

Get Branches of all Repositories Two distinct functions make this functionality possible: `getGithubToken()` and `getRepository()`. `getGithubToken()` has one parameter, `id`. `id` is used to get `github_auth` and `github_iv` of the user from the database. If no authentication was found, an error, `NOT_GITHUB_AUTHENTICATED`, is thrown. Otherwise, the GitHub token is decrypted using the iv and the token. `getRepository()` has one parameter, `id`. This `id` is used to get the `full_name` of the repository from the *Repositories* database.

To get the branches of a repository, a GET request is sent to `/github/:id(\\d+)/branches`. This endpoint requires `id` as a request parameter and `userId` from the request user. If `id` is not set, the endpoint returns a status code of 400 and a return body with all keys set to null except `error_message` which is set to `MISSING_REPO_ID`. After checking if all needed data is present, the GitHub oauth token is requested using `getGithubToken()`. Afterwards, the full name of the repository is obtained using `getRepository()`. After these two information are available, a GET request is then sent to `/repos/owner/repo/branches` using `@octokit/request` to get the branches of a repository. These branches are then returned and `error_message` is set to `null`. If an error occurs from the request, a 403 status code is returned, all keys are set to null in the request body, and `error_message` is set to `NOT_GITHUB_AUTHENTICATED`. If any other error occurs, a 500 status codes is sent and the `error_message` is set to the error message of the error.

2.9 Github Authentication

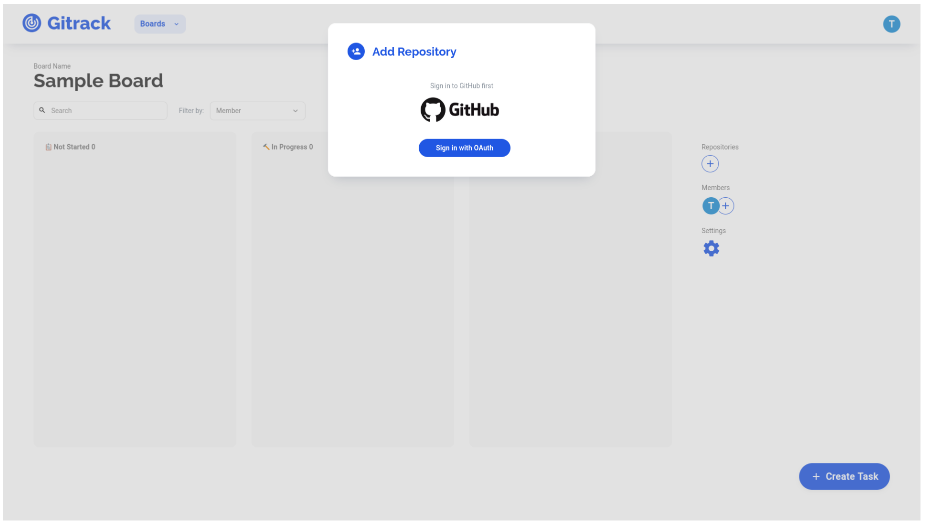


Figure 11: Github Authentication Modal

2.9.1 Description

Authenticating with GitHub is a core part of the application. This allows for connecting a board to a repository, and a task to a specific branch. Other core functionality of the application builds upon these two features.

2.9.2 Frontend

Clicking on the button to add a repository while a user has not connected their GitHub account will bring up the Github Authentication Modal which is defined by the `SigninGithubModal` component. This component calls the `link()` function under `GitHubService`. This function is a GET request to `/github/link` which returns a link for the user to access in order to allow authorization on GitHub. Once authorized, GitHub redirects the user to `https://gitrack.codes/github/` with `code` and `state` as query parameters. These parameters are used for the `callback()` function in `GithubService`. This function is a GET request to `/github/link/callback`. If the request was successful, the user is redirected to the boards page. Otherwise, the user is asked to retry the authentication process once again. All of the interactions in the URI are implemented by the `Github` component.

2.9.3 Backend

This functionality is handled through two endpoints: `/github/link`, `/github/link/callback`. `/github/link` generates a custom URI for users to click on. This link is a link for `https://github.com/login/oauth/authorize`. This link has additional parameters added: `scope`, `client_id`, and `state`. `scope` is the permissions that the application requests from the user. For GiTrack, this is the permission to modify repositories and to write webhooks. `client_id` is the Client ID for the OAuth application that was generated in GitHub. `state` is a randomly generated string that is used to validate the request for authorization. This is checked by both GitHub and by the application in succeeding requests.

After the user has allowed GiTrack these permissions, The frontend sends a request to `/link/callback` with two parameters: `code`, and `state`. These two queries come from GitHub and are used to continue the authentication flow. State is the exact same state that was created in `/github/link`. If the `state` received does not match a `state` stored in memory, this callback is rejected. If the `state` was present, these two parameters are used to send a POST request to `https://github.com/login/oauth/access_token` which receives the following data in its request body: `code`, `state`, `client_id`, and `client_secret`. `code` and `state` are the two parameters, `client_id` is the ID for the OAuth application and `client_secret` is a secret generated by GitHub to verify if the client is really the client and not a third party attacker. This endpoint returns an access token that is to be used for all privileged requests to GitHub in the future. This includes private repositories, writing webhooks, and others.

This access token is stored in the database using the function `addGithubToken()`. This function has two arguments: `id`, and `githubAuthToken`. `id` is the ID of the user and `githubAuthToken` is the auth token. This function encrypts the token using *AES-192-CTR*. The encryption returns two values, a token and the initialization vector. These two values are stored in the database under the columns `github_auth` and `github_iv` respectively. The auth token is encrypted because this allows a lot of permissions for anyone that has access to this token. If it falls into the wrong hands, this may destroy the entire GitHub account of the user

If the user already is authenticated, `addGithubToken()` throws an error message: `ALREADY_GITHUB_AUTHENTICATED`. The endpoint returns a 500 status code with the error message. Otherwise, it simply returns a 200 status code.

2.10 Add Repository

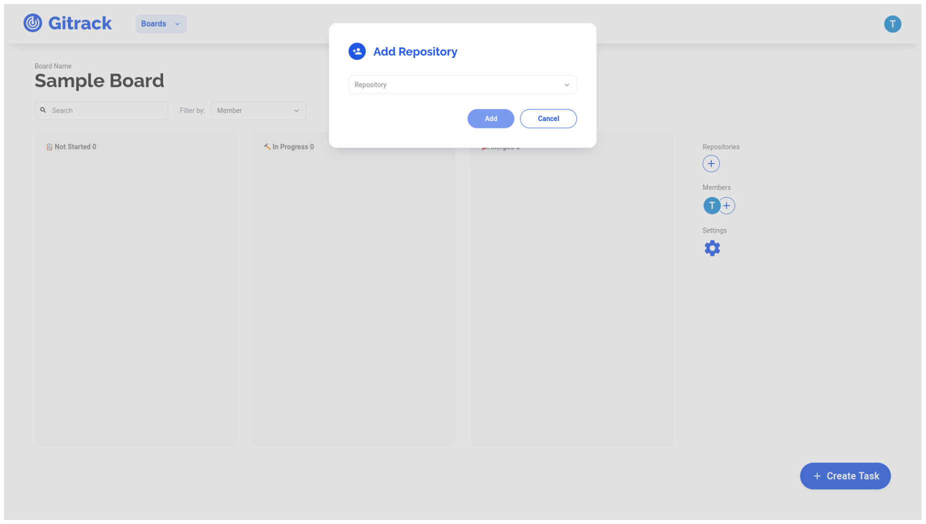


Figure 12: Add Repository Modal

2.10.1 Description

The *Add Repository* screen shows a modal with a dropdown menu that lists the repositories of the GitHub-authenticated user. This allows the user to add a repository to the current board and enables linking branches in the said repository to tasks.

2.10.2 Frontend

Add Repository is a modal that only gets displayed when the "Add" button under "Repositories" on the board is clicked and if the user has already connected their Github account. It was implemented using a modal component with the same UI elements such as the buttons, icon, and title as other modals of the same size. The only difference is the title, the "action" of the primary button which is "Add", and, importantly, the content of the modal itself which is a dropdown menu which lists all the repositories in the user's Github account.

`repos()`, which is one of the functions in `GithubService`, is immediately called once Add Repository is opened. `repos()` is a GET request to `/github/repos`.

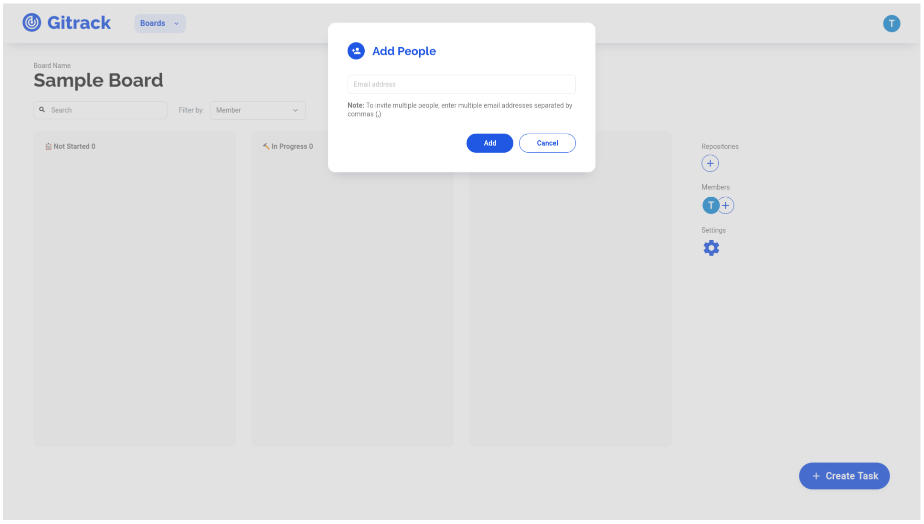
The server returns all the repositories connected to the user's account. The repositories' full names are displayed within the dropdown menu. Once a repository is chosen and the "Add" button is clicked, the `connectRepo()` function is called which is from `BoardService`. `connectRepo()` is a POST request to `/board/$boardId/connect` that sends the `boardID,` the selected repo's `id`, `full_name`, and `url`. This effectively assigns the chosen repository to the board. `refreshBoardRepos()` is then called and the modal is closed.

2.10.3 Backend

This is implemented using the `connectRepository()`, `getBoardById()`, and `getGithubToken()` functions. `connectRepository()` accepts four parameters: 1. `id`: the repository ID from GitHub, 2. `name`: the repository's name, 3. `url`: the URL to the repository, and 4. `boardId`: the id of the board to which the repository will be connected. `connectRepository()` adds these values to the Repositories table. `getBoardById()` returns an object containing a board's information in the *Boards* table based on its unique id, which is accepted as a function parameter id. `getGithubToken()` retrieves the user's GitHub authentication token from the database using their unique id, which is accepted as the function parameter userId. If the user has not authenticated their GitHub account, they will not have an authentication token in the database, and `getGithubToken()` will throw and error message of `NOT_GITHUB_AUTHENTICATED.`

The endpoint, `boards/connect`, receives a POST request with the board `id` as a parameter. It also makes use of the user's unique `id` to check for the required permissions. The request body contains: `repo_id`, the repository ID from GitHub; `full_name`, the full name of the repository from GitHub; and `url`, the URL to the repository. The endpoint returns a JSON with keys `id`, `full_name`, `board_id`, and an `error_message`. First, the endpoint checks if any of the `ids`, `fullName`, or `url` are missing. If it is missing, the endpoint returns a status code 400 with null values for the JSON keys and an `error_message` of `MISSING_ID`, `MISSING_FULL_NAME`, and `MISSING_URL` respectively. The endpoint checks if the user is a developer or a project manager with `getPermissions()`. The endpoint then retrieves a `board` object and stores it in a variable `board`, whose properties are the board information in the *Boards* table, using the `getBoardById()` function using the `id` request parameter as an argument. If `getBoardById()` does not find a board with the specific `id`, it returns `undefined` and `board` gets this value. In this case, the endpoint returns status code 403 and null values for the JSON keys and for security reasons, an error_message of `NOT_ENOUGH_PERMISSIONS`. If board is not `undefined`, the endpoint proceeds to retrieve the user's GitHub authentication token using `getGithubToken()` and stores it in a variable `authToken`. If this fails, the endpoint returns a status code 403 with null values for the JSON keys and the `error_message` thrown by `getGithubToken()`. If it succeeds, the endpoint proceeds to connect the repository to the board using `connectRepository()`, and creates a webhook to the repository (see Section M). If `connectRepository()` fails, the endpoint returns a status code 500 with null values for the JSON keys and the `error_message` thrown by `connectRepository()`. If it succeeds, it returns a status code 200 with the JSON keys having the following values: repoId for the `id` key, fullName for the `full_name` key, the id request parameter for the `board_id` key, and a null value for the `error_message` key.

2.11 Add Members



Note: To invite multiple people, enter multiple email addresses separated by commas (,)

Figure 13: Add Members Modal

2.11.1 Description

The Add Members shows a modal with an input form that lets the user type-in an email of other existing users for them to be added to the board.

2.11.2 Frontend

Add Members is a modal that only gets displayed when the "Add" button under "Members" on the board is clicked. It uses the same modal component and UI as the previous component, Add Repository. The only difference with

Add Repository is the title, the subtext under the input field that contains instructions on how to add members, and the input field for the email addresses.

In order to add a member to the board, the user must enter the specific member's email address. As stated in the note underneath the input field, it is possible to add multiple members by separating the email addresses with commas. Once the "Add" button is clicked, the `onSubmit` function is called. This takes the email/s entered and passes them as arguments for the `exists()` function found in `UserService`. This is a GET request to `/user/exists`. If the emails are connected to existing users, their ids are returned. The `addDevelopers()` function from `BoardService` is then called. `addDevelopers()` is a POST request to `/board/$boardId/add-developer` that sends the `board_id` and the `developer_ids` to be added as members to the corresponding board back to the server. `refreshBoardMembers()` is then called and the modal is closed.

If one or more emails aren't connected to any users, the `exists()` function returns null. An error message is then displayed indicating "One or more users do not exist."

2.11.3 Backend

This was implemented through a function `addDevtoBoard()` which accepts the parameters `boardId` and `devId`. It then checks if the User with `id` specified in `devId` exists in the database. If the user does exist, it is then added into the *Memberships* table. Otherwise throws an error `USER_NOT_FOUND`. If inserting into the *Memberships* table fails, `INSERT_FAILED` error is then thrown.

For its endpoint, `boards/:id(\\d+)/add-developer`, receives a POST request which accepts an `id` from the request parameter, accepts `userId` from the user, and accepts `developer_ids` from the request body. It also returns a JSON with the keys `board_id`, `dev_id`, `duplicate_devs`, and `error_message`. An empty array of developer IDs is then declared. It checks if the `id` is undefined. If it is, then it responds with status code 400, returns `null` value for the JSON keys, and an `error_message` of `MISSING_ID`. After, it checks if the user has permissions to do this function by trying the `getPermissions()` function. If it fails, then it responds with a 403 status code, also returns `null` for the JSON keys, and the thrown error from the `getPermission()` function for the `error_message`. And if the user has permissions, it then proceeds to try and call `addDevtoBoard()` function. If the developers have successfully been added, it then responds and returns `id` for `board_id`, string converted `devsToAdd` for `dev_id`, filtered `devIds` for the `duplicate_devs`, and `null` for the `error_message`. If it fails to add the developers to the board, it responds with a 500 status code, returns `null` for `board_id`, `dev_id`, and `duplicate_devs` keys, and an `error_message` from the error thrown of `addDevToBoard()` function.

2.12 Create Task

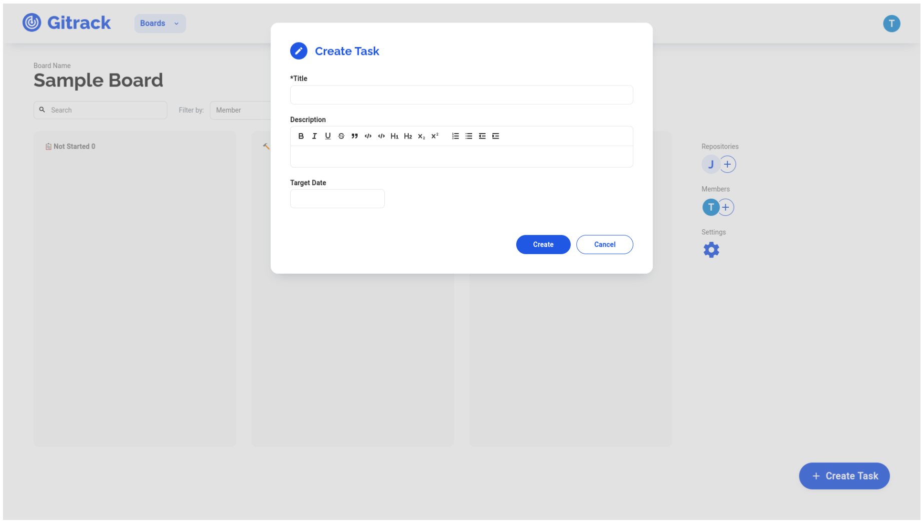


Figure 14: Create Task Modal

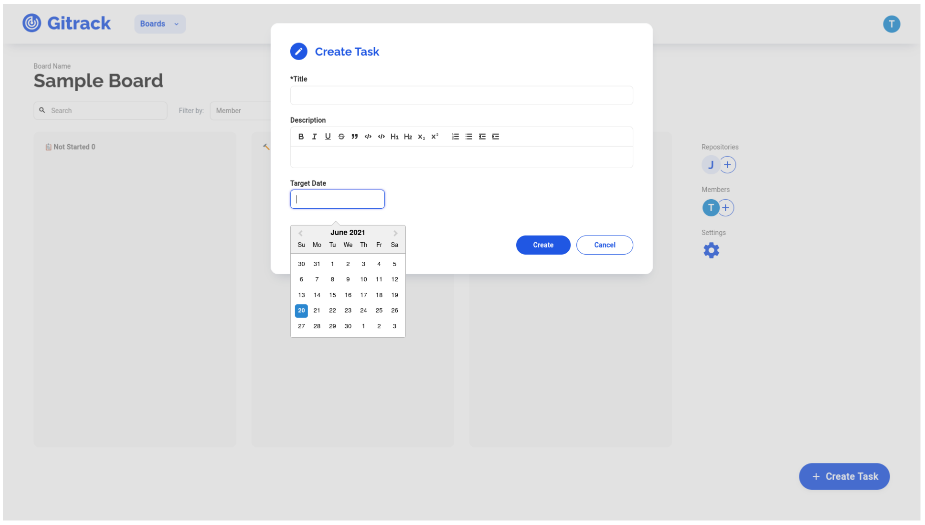


Figure 15: Create Task Modal with Target Date input clicked

2.12.1 Description

The Create Task component shows a modal that lets the user enter the specifications of their task. The specification includes the title of the task, description, and target date. Out of the three fields, only the title is required to be able to successfully create a task.

2.12.2 Frontend

The three inputs in the Create Task component function very differently from one another. The Title input is a very simple input that is only capable of plain text. The Description input is a rich text editor that is capable of formatting text. This allows the user to emphasize key words or phrases, add numbering or even add bullet points. This was possible through the integration of an open source WYSIWYG editor known as *Quill*. Lastly, the Target Date input, when clicked, displays a calendar that would set the date by selecting the desired date on the calendar. This was implemented using an open source datepicker component known as *React Datepicker*. Once the user clicks the 'Create' button, an *onSubmit()* function is triggered. It first checks whether the Title input is filled, since it is the only one that is required to create a task. If it isn't, it would display an error message telling the user to fill in the input. After that, the `create()` function from the `TaskService` would then be called. The create() function is a POST request to /task that sends the board_id, title, description, and target_date as its body. Once successful, it retrieves the updated list of tasks in the board by calling the refreshBoardTasks() function and would then proceed to close the modal.

2.12.3 Backend

This component is implemented through a function addTask() which accepts the parameters title, description, targetDate, and boardId. It then tries to insert into the *Tasks* table in the database. If insert is successful, it returns the last ID that was inserted. Otherwise, throws an error of INSERT_FAILED.

For this component's endpoint, it receives a POST request which accepts board_Id, title, description, and target_date from the request body, and userId from the user. It also returns a JSON with id, column_id, board_id, title, description, target_date, and error_message keys. It then checks if boardId, title, description, and targetDate are undefined. If one those are undefined, it responds with 400 status code, returns an error_message of MISSING_BOARD_ID, MISSING_TITLE, MISSING_DESCRIPTION, and MISSING_TARGET_DATE respectively, and null values for JSON keys. After, it checks if the user has permissions to do this function by trying the getPermissions() function. If it fails, then it responds with a 403 status code, also returns null for the keys, and an error thrown from the getPermission() function for the error_message. And if the user has permissions, it then proceeds to try and call addTask() function. If it is successful, it then responds and returns taskId for id, 0 for column_id, boardId for board_id, title, description, targetDate for target_date key, and null for the error_message. If it fails to add the developers to the board, it responds with a 500 status code, returns null for the JSON keys, and an error_message from the error thrown by the addTask() function.

2.13 Task Card

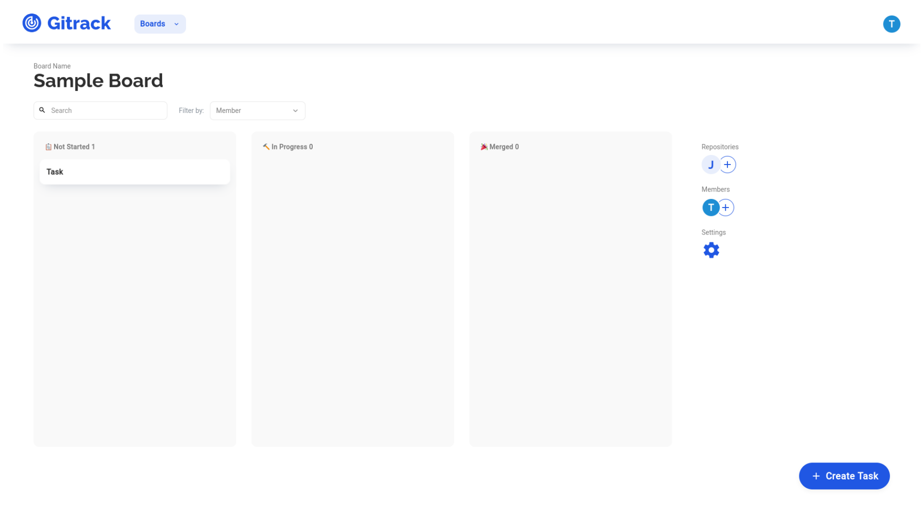
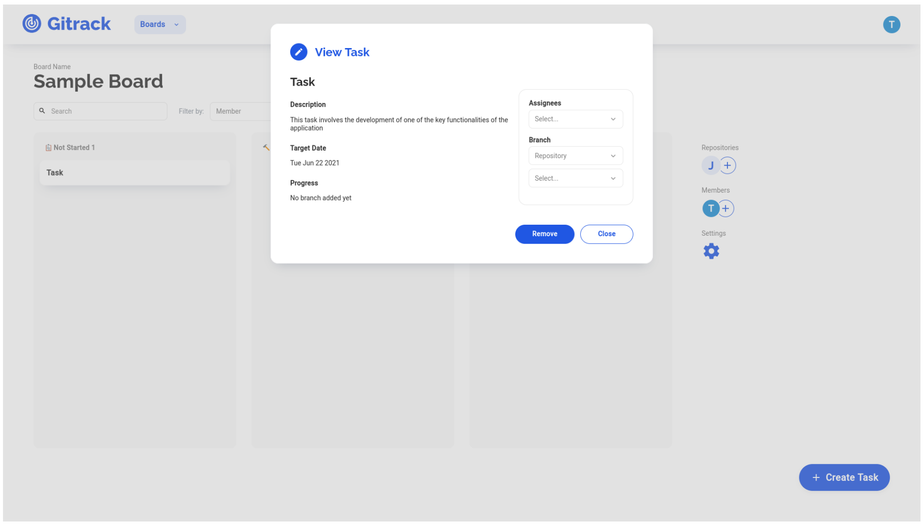


Figure 16: Task Card in Not Started Column



Task

Description

This task involves the development of one of the key functionalities of the application

Target Date

Tue Jun 22 2021

Progress

No branch added yet

Assignees

Select...

Branch

Repository

Select...

Figure 17: View Task Modal

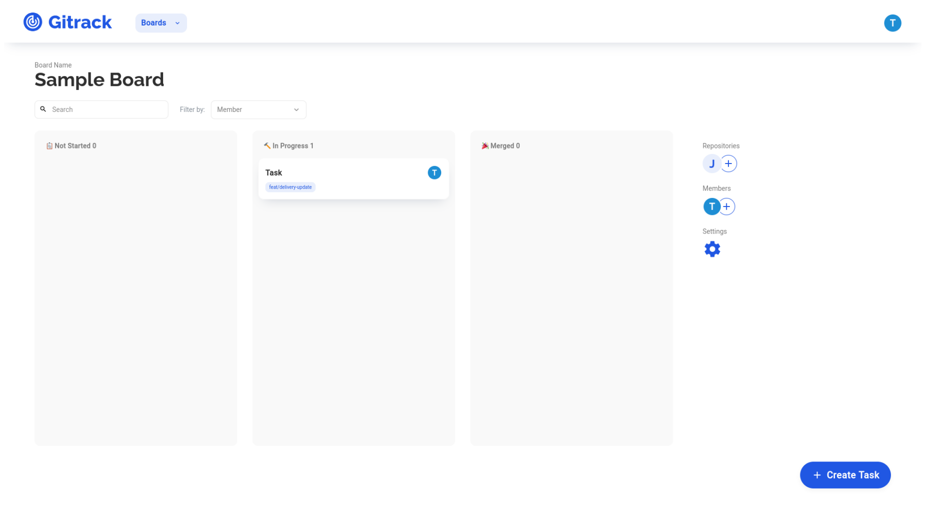


Figure 18: Task Card in In Progress Column

2.13.1 Description

The Task Card component represents a task in the project. When the task is first created, the task is placed in the Not Started column and the Task Card would only display the task's title, but once the task's information is updated, such as the assignees and the linked branch, it would be updated to display those as well. Once the task is linked to a branch, it would automatically be transferred to the In Progress column, and once the branch linked to it is merged, it will then be transferred to the Merged column. In order to update the task's information, the user would have to click the Task Card to open the View Task modal.

The View Task modal allows the user to view more information about the task, as well as modify information that would most likely be updated throughout the task's lifetime such as its assignees and the branch it is linked to. It also gives the user the ability to delete the task when the need arises.

2.13.2 Frontend

Task Card The Task Card, from the name itself, uses a simple Card component with custom design elements. It displays the essential information of each task: the task's title, the icon of the member assigned, indicators for the deadline, and the branch linked to the card. If the cursor is hovered above the task card, the card's color changes to a light blue from the normal white.

Everytime the card is rendered, the state of the task's progress relative to the deadline is checked. Using the current date and the target date of the task, it is determined whether the task has gone beyond the deadline without being completed or merged. If the current date is beyond the task date, a warning icon is then displayed next to the title of the card and the card's border color is changed to red.

View Task The View Task modal is opened once the Task Card is clicked. This modal is larger in size and shares design elements with other modals. At the bottom right of the modal is the Remove and Cancel buttons. The Remove button in blue opens the Remove Task Modal while the Cancel button closes the View Task Modal. Clicking outside of the modal also effectively closes the modal.

Much like in Task Card, once the modal is opened it immediately checks the task's progress relative to the deadline. If the current date is past the task deadline, a warning icon is displayed next to the Target Date.

Remove Task The Remove Task modal is opened once the Remove button on View Task modal is clicked. It shares the same UI elements as other modals of the same size except the content simply asks "Are you sure you want to remove this task?"

Once the Yes button is clicked to confirm the removal, the `remove()` function is called from `TaskService`. This is a DELETE request to `/task/$taskId` that sends the taskId as the body. If the delete is successful, `refreshBoardTasks()` is called and the modal is closed.

Setting Assignees To the right of the View Task modal, a card houses the dropdown menus for the list of members that could be assigned to the task. Once the View Task modal is opened, it immediately checks if there are members assigned to the task already. From the dropdown menu, the user can simply click on the name of the member they would like to assign to the task.

Once the modal is closed, if there are developers selected and no members have been assigned to the task before, the `assign()` function is called from `TaskService`. This is a POST request to `/task/$taskId/assign-task` which sends the `task_id`, `board_id`, and `assignee_ids`. If there were already assigned members beforehand and there is another developer selected, the `updateAssignees()`

function is called from `TaskService` which is a PUT request to `/task/$taskId/assign-task` that sends the same data. If the request response is successful, the function `refreshBoardTasks()` is called and the modal is closed.

Connect Branch Right under the assignees dropdown menu are the dropdown menus for the repository and branch that could be assigned to the task. If there is no repository assigned to the task yet, the repositories linked to the board are all listed. Once the user selects a repository, the specific branches in that repository are then displayed in the dropdown menu for branches. Once a branch is selected or if there already was a selected branch before and a new branch is selected, the `connect()` function from `TaskService` is called. This is a PATCH request that sends the `repo_id` of the selected branch, the name of the selected branch, and the `taskId`. This updates the branch connected to the specific task.

2.13.3 Backend

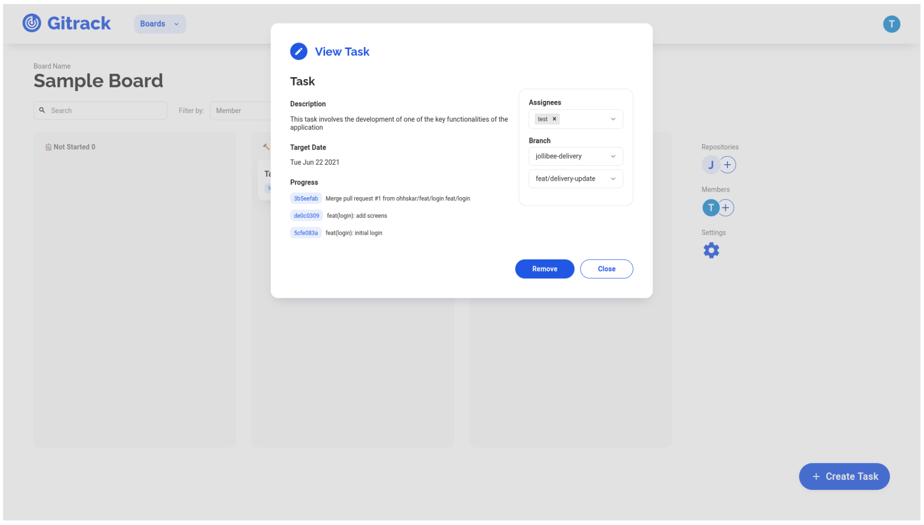
View Task From the `/board/:id(\\d+)/tasks` endpoint used in the Main Dashboard (see Section F), an array of objects containing all the task information in the board is obtained and the values of the properties of these objects are displayed in the frontend.

Remove Task This is implemented through a function `removeTask()` which accepts the parameter `id`. Through this `id`, the `id` in the *Tasks* table it corresponds to is deleted. If the result.changes is zero (0), it means such a task does not exist, throwing an error of `TASK_NOT_FOUND`. Otherwise, the `id` is returned. If unsuccessful, it throws an error. The `removeTask()` is then called in the `tasks/:id(\\d+)` endpoint. This endpoint receives a DELETE request which accepts `id` from request parameters, and `userId` from the user. This also returns a JSON with keys `id`, `column_id`, `board_id`, `title`, `target_date`, and `error_message`. It first checks if the task exists. If it exists, then checks if the user has permissions through the `getPermissions()` function. After, the `removeTask()` is then called. If successful, returns a JSON of the task, and `null` value for `error_message` key.

Set Assignees This is implemented through the function `setAssignees()` which accepts the parameters `taskId`, and `assigneeIds`. It is then followed by a try catch in which `assigneeIds` are individually inserted into the `Assignees` table along with the `taskId`. If it fails, it is then caught, and an error `INSERT_FAILED` is thrown. This function is then called in the `tasks/:id(\\d+)/assign-task` endpoint. This endpoint receives a POST request which accepts `id` from request parameters, `userId` from the user, and `board_id, assignee_ids` from the request body. This also returns a JSON with keys `board_id`, `task_id`, `assignee_ids`, and `error_message`. It first checks if the user has permissions through `getPermissions()` function. If the user has permission, then the `setAssignees()` is then called. If successful, returns a JSON with `boardId` value for `board_id`, `id` value for `task_id` key, `assigneeIds` value for `assignee_ids` key, and `null` value for `error_message` key.

Connect Branch This is implemented through function `connectBranch()` which accepts the parameters `task_id`, `branch`, and `repoId`. The row of the `taskId` in the *Tasks* table is then updated with the repository ID (`repoId`) and branch (`branch`) it corresponds to. If unsuccessful, an error message of `CONNECTION_FAILED` is thrown. The `connectBranch()` function is then called in the `tasks/:id(\\d+)/connect` endpoint. This endpoint receives a PATCH request which accepts `id` from request parameters, `userId` from the user, and `repo_id`, `branch_name` from the request body. This also returns a JSON with keys `name`, `repo_id`, and `error_message`. It first checks if the `id`, `repoId`, and `branchName` are undefined. If at least one is `undefined`, an error is thrown. Then, the `connnectBranch()` is then called. If successful, returns a JSON with `branchName` value for name `key`, `repoId` value for `repo_id` key, and `null` value for `error_message` key.

2.14 Task Card: View Commits



View Task

Task

Description

This task involves the development of one of the key functionalities of the application

Target Date

Tue Jun 22 2021

Progress

3b5eefab Merge pull request #1 from ohhskar/feat/login feat/login

de0c0309 feat(login): add screens

5cfe083a feat(login): initial login

Assignees: test

Branch: jollibee-delivery

feat/delivery-update

Figure 19: View Task Modal with Commits

2.14.1 Description

The View Commits component is only available to task cards whose tasks are linked to a branch in GitHub. The task card can then display the three latest commits to the branch linked to the task.

2.14.2 Frontend

Once the View Task Modal is opened and a branch has already been linked to the task, or when the developer has just finished linking the task to a branch, the component would then proceed to call the `commits()` function from `GithubService`. The `commits()` function is a GET request to `/github/$repoId/commits` that retrieves the list of commits given a `repo_id` and `branch_name`. This function would also be called whenever the developer decides to link a different branch. Once the server returns the list of commits, the component would then only display the three latest commits from that branch.

2.14.3 Backend

This is implemented directly at its endpoint. It receives a GET request which accepts `id` from the request parameter, and accepts `userId` from the user. Three variables are declared with null values, namely `authToken`, `fullName`, and `rep`. It first checks if the `id` is `undefined`, and if it is, then responds with 400 status code, returns `null` value for `id` and `title`, and an `error_message` of `MISSING_REPO_ID`. If `id` is defined, it then proceeds to try and call `getGithubToken()` function to get an authentication token for the user. If unsuccessful, it responds with 403 status code, returns `null` value for the branches, and an `error_message` from the function. If successful, it then tries and calls `getRepository()` function. If it fails, it responds with 400 status code, returns `null` value for the branches, and an `error_message` from the thrown error of the `getRepository()` function. Otherwise, the returned string of the `getRepository()` function which is stored in the `fullName` variable is split with '/' and is assigned to the `rep` variable. After, POST GitHub endpoint `/repos/owner/repo/commits` is called to get the commits, and then stored in `data`. If successful, `data` is then mapped to get the required elements from the list of commits which are the hashes, url, and commit message. It then returns a JSON of the mapped commits, and `null` value of `error_message`. If the request failed, the error status is checked. If error status is 401, `removeGithubToken()` is called, then it responds with 403 status code, returns `null` value of `repos`, and an `error_message` of `NOT_GITHUB_AUTHENTICATED`. If not 401, then it responds with 403 status code, returns null value of repos and the error from the recent try catch.

2.15 Automatic Movement of Cards

2.15.1 Description

A key differentiator of GiTrack over other project management tools is the ability to automatically move cards depending on the git status of the accompanying branch. If a task has a connected branch, this task will be automatically moved to the next column: "In Progress". Once a feature branch has been completed and merged to the main branch, the task will be automatically moved to the final column: "Completed".

2.15.2 Backend

There are two ways the backend moves the cards. The first method is done when a card is moved from "To Do" to "In Progress". The movement of a card of this type is done when a branch is connected to the task. As such, this functionality piggybacks on the `task/id/connect` endpoint. When this endpoint is called, the function `connectBranch()` is called. This function is described in full on Section K. One specific change this function has is updating the `column_id` column on a task. It sets this `column_id` to 1 which indicates that this task is now on the "In Progress" column.

The second method handles the instance where a card moves from "In Progress" to "Completed". This is done by detecting the moment a branch is merged and checking if that branch is connected to a task. To accomplish this a webhook is attached to a repository once it has been connected. This is done by the `board/id/connect` endpoint. The full functionality of this endpoint is described in Section H. A part of the functionality of this endpoint is calling the POST GitHub endpoint `/repos/owner/repo/hooks` with a config that indicates that the application is interested in receiving all events that correspond to pull requests. GitHub sends this information to the application by sending a POST request to `/github/payload`. This endpoint checks the type of payload that GitHub has sent. If it is a `pull_request` payload, the endpoint checks if that specific event is closing and merging a pull request. If so, it calls the function `moveTaskByBranchAndRepo()` which has `branchName`, `repoId`, and `columnId` as arguments. `branchName` specifies the full branch name, `repoId` is the ID of the repository where the pull request was merged, and `columnId` is the column where that task is to be moved. In this case, `columnId` was set to 2, which means that the task should be moved to the "Completed" column.