ZEBRA

Real-time collaborative project management tool for teams

Project Report

Team

Apoorv Saini (Team Lead)
Aditi Lal
Rohit Morey
Sahana CS
Nandana Yadla
Karthik Rumalla

To maintain the flow of the report we have written sections of work as each member did. Also **final work done by each member is shown at the end of the report.**

Overview

We are aware of the troubles small teams go through while working on projects (especially managing Software Development Lifecycle). There are a lot of specialized tools for managing the projects and timelines but their abundance has created a fragmentation and this is where our project, Zebra comes in.

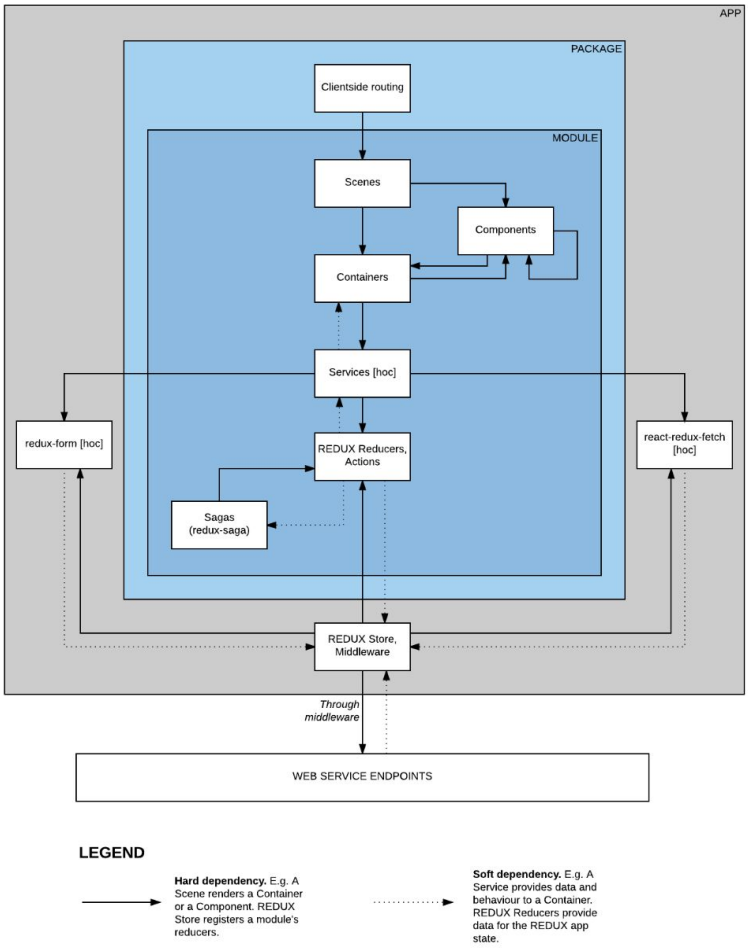
We picked the most essential parts and processes involved in SDLC and other projects and focused on making a single workplace for all team members to manage their projects.
We built a collaborative doc for all the members of the team to work on ideas, project report and specifications. The same doc provides the slack-like chat without jeopardizing the minimal distraction-free design of the workspace. On top of that we added Kanban tool, the most popular agile project management tool used by all major companies. And, the best part is all the components work in real time.

Architecture *(Apoorv Saini)*

For Zebra, we decided to go with multi-tier architecture. We built it keeping modern browsers and high concurrency in mind, the front-end we designed is a fat client i.e. a lot of logic and navigation is decoupled from the backend and the frontend (even though running in browser) can work offline (i.e. keep state even when the Backend is unreachable). Let's go deep into our architecture:

1. **Frontend architecture *(Apoorv Saini)***
   The front-end is built using **ReactJS** (https://reactjs.org/) + **Redux** (https://redux.js.org/) which the most modern frontend architecture used by a lot of prominent tech companies (the complete Facebook web-app is built using React JS).

React helped us properly divide the UI components, Actions and States, which helped in escalating the development is a faster and organized way.
I have used ES6 standard of Javascript. Following is the representation of our frontend architecture that I have built.



Due to Zebra's reactive nature, the client stores the state of the components and the **Thunk middleware**

(https://github.com/gaearon/redux-thunk) observes the states and broadcasts it to the components, updating the data or UI states. The whole frontend is written using Javascript and the components are complied in a JS bundle (cached on browser) which renders HTML for the browser as per user actions.
The **src/** folder at the root of the project contains the structure of the frontend.
The **reducers/** are responsible for maintaining the reactive states. Following is one of the reducer structure that I use for the Doc component.

```
import { loadUserProfile } from "../utils/apiUtils";

const initialState = {
  showCal: false,
  showGantt: false,
  projectId: "new",
  projectName: "",
  teamList: [],
  chatMessages: [],
  projects: [],
  taskList:[],
  inviteVisible: false,
  docData: {
    "entityMap": {},
    "blocks": [
      {
        "key": "ag6qs",
        "text": "",
        "type": "unstyled",
        "depth": 0,
        "inlineStyleRanges": [],
        "entityRanges": [],
        "data": {}
      }
    ]
  },
};
```

The **actions/** contains the constant actions that are binded with the components and are responsible for changing the states in reducers.
The **containers/** contains the separate modules which holds smaller reusable components. Containers themselves are not reusable.
The **components**/ are reusable components which use Thunk to bind themselves with the reducer states and are able to **Dispatch Actions.**
At the very core, we user React-Router to enable navigation which in turn utilizes HTML5 History API. The web app changes the contents/screens and the url without loading.

The app has minimalist UI and the 90% of the styles used are written by us. For fall-back we utilize Bulma CSS.

2. **Frontend Server *(Apoorv Saini)***
   Since our frontend is decoupled from the backend (business logic), the frontend is served using **Node.js** server. The server handles utilities and basic authentication for the client side as well serving the JS bundles as per request. I have mentioned, how to run the server in README file.

3. **Real-time/Event driven server *(Apoorv Saini)***
   While we were at it, we thought why not develop our own RTC service. We did that by abstracting Bayeux protocol (https://docs.cometd.org/current/reference/#_bayeux) and using Faye to write a Pub/Sub (publish/subscribe) interface over it. The server is

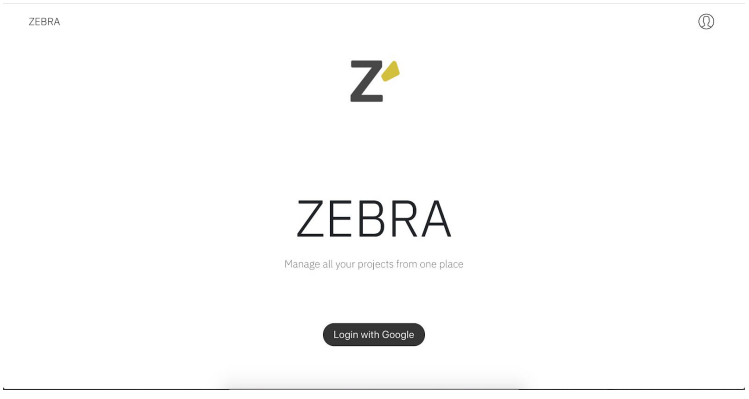
written in Javascript (Node.js) and we are running it on a medium instance on AWS. Every client connects to this server using websocket (ws://) and subscribe to his/her individual channel (channel name is provided by Application server after the user logs in). Any message from other client is handled by this server and broadcasted to all the subscribers of that channel in real time.

4. **Application Server *(Aditi Lal)***
   Written in **Python** and running on **Flask** server. The APIs are served by this server. The API endpoints provide the JSON data queried from the Database (MongoDB). The token-based basic authentication (explained later in this report) is knitted in every request that this server handles. Along with that, we have enabled CORS request for every endpoint as all the requests care originated from the client side (Javascript).

5. **Database Cluster *(Aditi Lal & Nandana & Sahana)***
   We decided to work with NoSQL DB and chose MongoDB. We are using MongoDB cluster running on Atlas (https://www.mongodb.com/cloud/atlas) to store user data, project information as well as the collaborative doc and Kanban data. For better data availability we have created 3 shards (1 Master + 2 Slaves).

Features

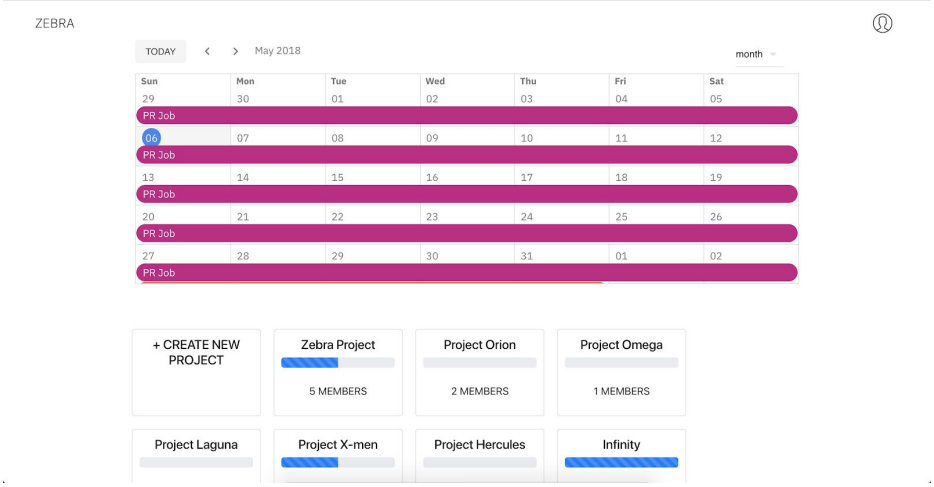
1. **Authentication/Google Login *(Apoorv Saini)***
   We rely on Google auth JS plugin for signing up and logging users in. After a user is authenticated by Google, we get their **email-id, name and auth token**. The frontend server then sends this data to Application server using API (/auth/ endpoint), where our Flask server checks if the email is already registered or not and generates a new auth token (and drops the Google's token). The new token is sent back to the frontend and is saved as cookie in browser. All the subsequent API calls require that token to be sent back along with request body so that the backend can authenticate each user and request before processing it.



2. **Dashboard *(Apoorv Saini)***

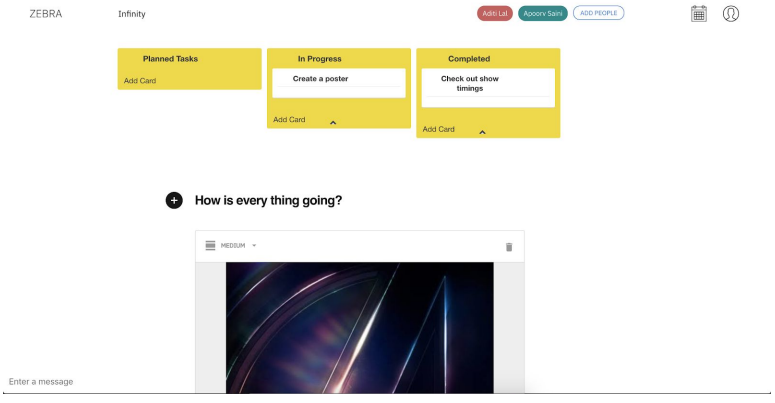
First thing that the user sees after logging in or signing in is the dashboard. The dashboard comprises of a calendar and all the projects the user is a part of. The dashboard API fetches all the tasks related to all the projects he is a part of and we show them on the calendar.

User also gets a clear visualization of how much the project is completed. We calculate it by dividing the number of tasks marked completed by total number of tasks in the project.

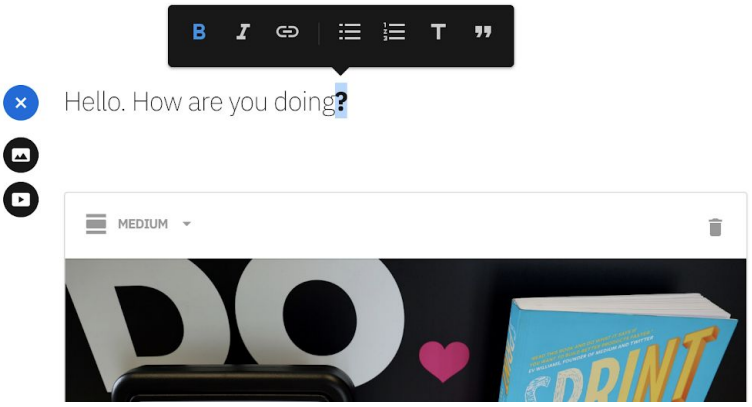


3. **Workspace/Doc *(Apoorv Saini)***

The main feature of Zebra. It is where all the collaboration and planning happen. The workspace consists of a real-time collaborative rich text editor, a chat feature and a Kanban tool (which also changes in real-time for every user who has the same doc opened)



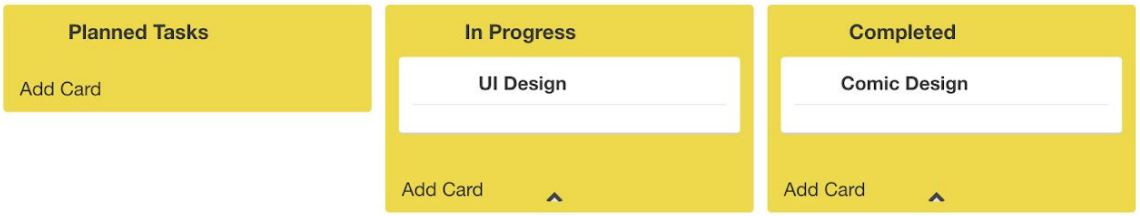
(Above) The kanban tool remains on the top and can be collapsed.



(Above) The text editor has rich features and text changes in real time if other members edit the same doc.

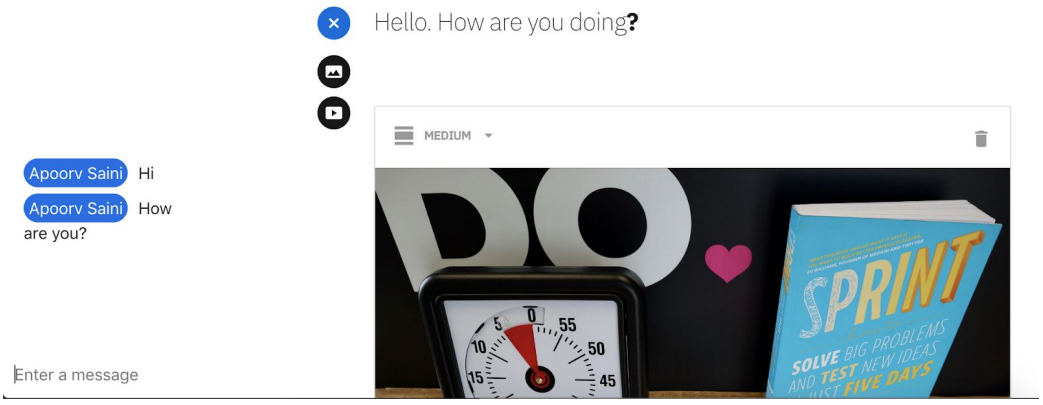
We have relied on Megadraft Editor (https://github.com/globocom/megadraft) to make rich text editing possible. Around the Megadraft we have created a component wrapper to extend its functionalities according to our needs.
After every 10 keyUp events or 10 seconds (whichever happens first) the component calls /doc_save API to auto save the user's progress on the doc. The content of the doc is stored using reducer locally in JSON format and during the above action, the JSON is parsed as string and sent to the server.

4. **KanBan *(Apoorv Saini)***
   Perhaps the most widely used Agile management method, we decided it to be the tool of choice on Zebra. Users can create tasks and drag and drop tasks in "In Progress" or "Completed" board to keep track of the project progress. The tool sits on top of the doc to maintain the visibility. Any changes made to the board is also reflected to other users in real-time.



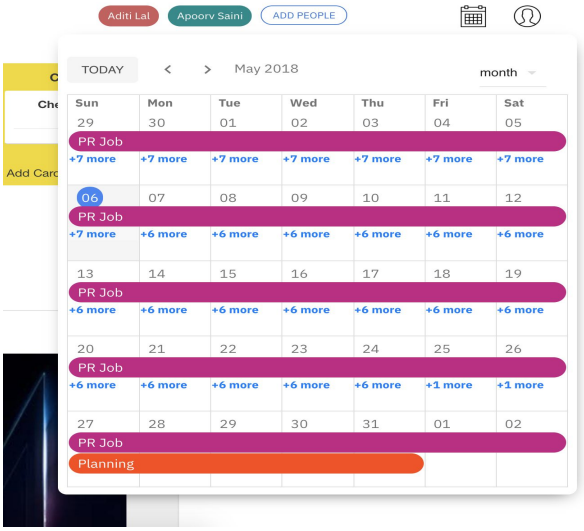
5. **Chat + Realtime *(Apoorv Saini [complete backend + frontend components])***
   Since, real-time holds the whole workspace features alive, we added chat feature too. We have created a minimalist UI and the chat also follows the same design pattern. It doesn't disturb the user experience and users can switch from editing the doc to

talking/discussing the project at any moment. You can refer to section 3 of **Architecture**, where I have explained how we have built the real-time frameowrk for our project (also read the README file along with this report)



6. **Calendar/Timeline** ***(Apoorv Saini [Extended functionality] & Rohit More [Basic function])***

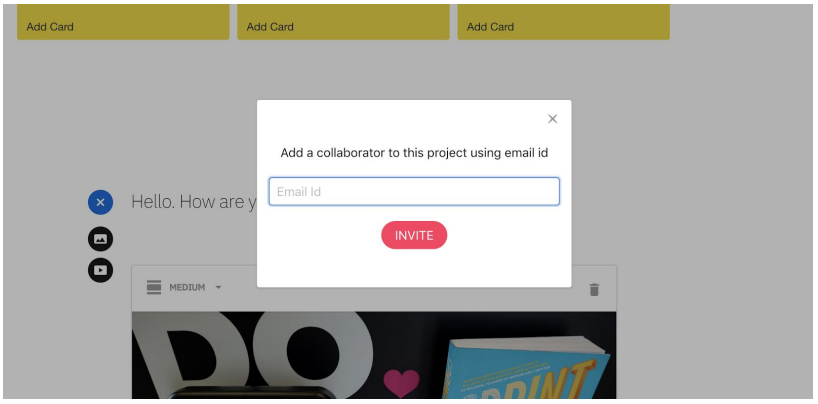
   The calendar component is available on dashboard as well as inside the project workspace (or Doc).



   We have explained the working behind the Calendar component in details later in the report.

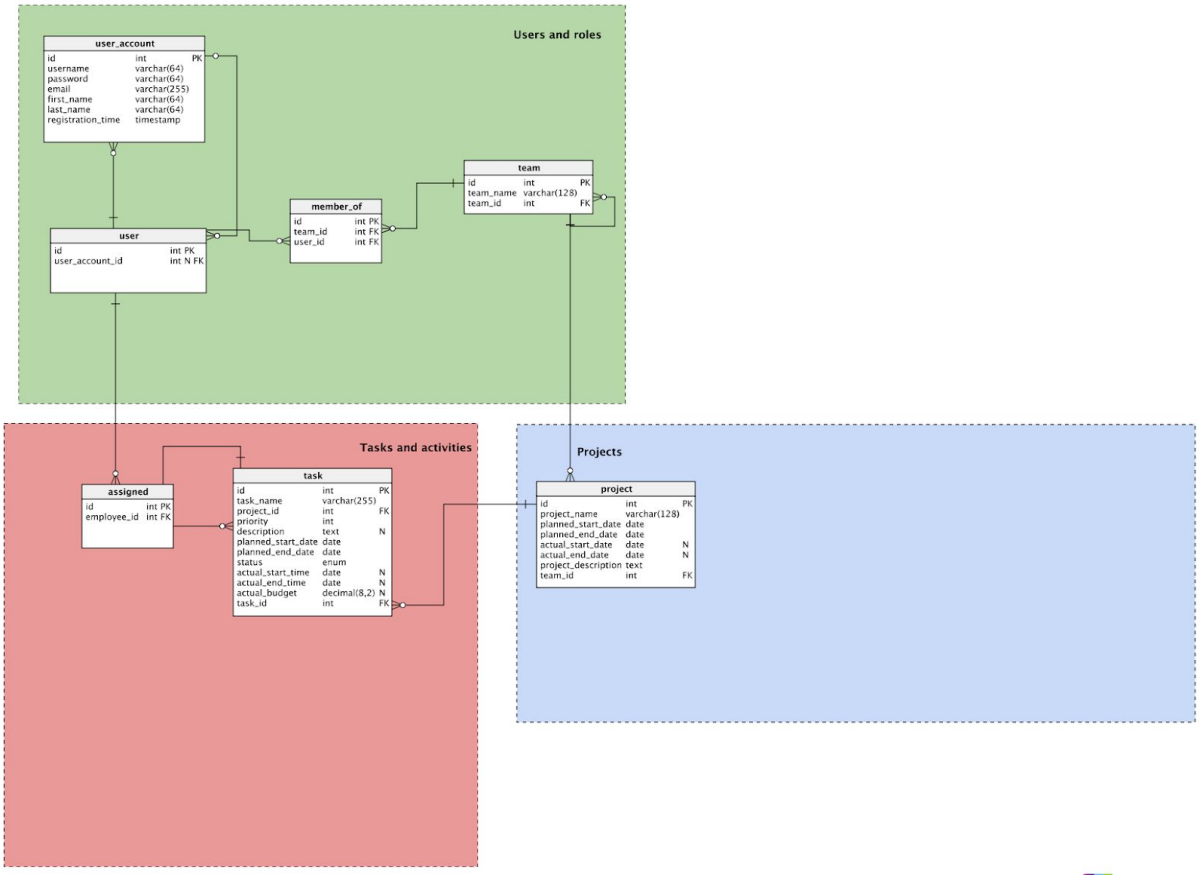
7. **Invite** ***(Apoorv Saini [Frontend] & Sahana [Backend])***

   Users can invite other members of Zebra by just clicking “Add People” button and entering their email id. The /invite API endpoint handles this request and updates the database for project members. The invited users then sees the new project on his/her dashboard.



DB Modelling *(Aditi Lal):*

We used MongoDB (non-relational DB) for our database implementation, but followed the following relational DB model to create our MongoDB model and implement DB normalization



Backend (APIs) ***(by Aditi Lal)***

The Zebra backend is built using APIs written in Python's Flask Framework and consisted of the following APIs for powering individual features of the application:

1. **/auth/sign_in (User Authentication API)**
   The user authentication for Zebra was built using Google's authentication API (as explained above). The user get auth token from Google and that token is sent to backend using this API endpoint along with email and name. We then check for the existing email id and create a new token (even if a user is already present, we delete the old token and create a new token) for the user and send it as a response.

2. **/new_project (Create new Project API)**
   This POST API is called when the user wants to create a new project/workspace on Zebra. Each project on Zebra is an individual document and denotes the workspace for that particular project consisting of tools specific to that project. The new_project API uses following POST request parameters:
   - creator_token: unique user token stored in the DB for the user who initiated the request to create the project.This token is fetched from the request and first verified from the users collection in the DB to make sure the user is valid and then proceed successfully further
   - project_name: the project name to be stored in the DB
   - doc_content: entire Doc content which needs to be stored in the form of a JSON (parsed as string) in the DBOn receiving the parameters the API if verifies the creator token successfully, proceeds with adding a new project entry in the collection and also saves the doc creation date/time in the same DB record

3. **/fetch_doc (get project details/ doc details):** This API is called anytime the doc is opened or refreshed in order to display current doc contents to the users. It uses the following POST request parameters:
   - user_token: unique user token stored in the DB for the user who initiated the request to view the project.This token is fetched from the request and first verified from the users collection in the DB to make sure the user is valid and then proceed successfully further
   - project_id_string: project id for which the contents have to viewed/fetched from the DB. This is a string but later is converted to MongoDB ObjectID using ObjectId() constructor
     API finally fetches the details from the project collections using this Project ID as the search parameter and returns all project details including the current project members, content and last updated information for the doc

4. **/save_project (Auto-save the doc contents):** This API is use to save doc contents in the DB and uses the following POST request parameters:
    - token: unique token to verify user
    - Project_id_string: represents the current project's unique DB id
    - doc_content: updated Doc contents to be saved in the DB

    Once the above parameters are received and the user token is verified in the DB, the doc contents are saved in the DB using the MongoDB update function on the projects Collection.

    If the update is successful - success as a response is sent, else an error is sent in the response to notify failure

5. **/create_task (create a new task from KanBan tool):** This API is used to create a new task for a particular project and uses the following request parameters:
    - token: unique token to verify user
    - project_id_string: represents the current project's unique DB id
    - task_name: name of the task being created
    - description: task description
    - start_date: start date set for the task
    - end_date: set end date for the task

      Once the above parameters are received and the user token is verified in the DB, a new task in added in the tasks collection in the MongoDB using the insert function call. A corresponding JSON response is sent denoting success or failure of the function.
6. **/user_dashboard (for calendar and Kanban) :** This API fetches all the user's dashboard details based on the following request parameter

- token: unique token to for the current user for which the dashboard details needs to be displayed

  Once the user token is received and the user token is verified in the DB, the user details are fetched from the DB including the projects user is a member of and tasks assigned to the user. These details are then sent as a JSON response in the following format which is used to populate user's dashboard, Kanban, and Calendar tools:
    - status: this represents the status of the API call- error at any stage will return the corresponding error message in the JSON parameters while a success will return success and the add an additional data parameter which contains the user's dashboard data
    - data: it consists of the user data in the below format:
        - projects: contains user's current projects and their corresponding details displayed in the JSON structure below. Also computes the project's progress depending on number of tasks completed for the project and returns as a % as progress to be displayed on user's dashboard

- tasks: all tasks assigned to the user along with their full details which are displayed on user's calendar and kanban

Structure of the JSON response :

{'status': success/failure, 'data': {'projects': 'project_id':str(project['_id']),'project_name':project['project_name'],'owner':owner_name,'members':count, 'progress':project_progress},{'tasks': 'task_id':str(task['_id']),'task_name':task['task_name'],'description':task['description'],'project_id':str(task['project_id']),'start_date':task['start_date'],'end_date':task['end_date'],'task_status':task['status']}}

7. /get_user_calendar_tasks : this API separately returns all tasks to be displayed on user's calendar. Takes the following POST request parameters:

- token: unique token of the current user for whom the taks need to be displayed

This API first verifies the user using the given token, and once the token is verified fetches all the tasks assigned to the user as a JSON response of the following form:

{'status': success/failure, 'data': {'tasks': 'task_id':str(task['_id']),'task_name':task['task_name'],'description':task['description'],'project_id':str(task['project_id']),'start_date':task['start_date'],'end_date':task['end_date'],'task_status':task['status']}}

Backend APIs (by Sahana)

INVITE API :

INVITE API is used to add a member to a particular project and uses the following request parameters :

Project_id : This id represents the project 's unique DB id to which the member who is being invited will be added to .

Email_id : This id represents the email id of the member who has to be added to a particular project .

Token: This represents a unique token to verify the user who will be adding the other members

Once the above parameters are received and the user token is verified in the Mongo DB, project_id is used to check if that projects exists in the database. After the project_id is verified , email_id is used to check if the member to be added exists in the database and if it does then that member is made a member of the project by adding to the project_member collection in the mongoDB using insert function call.

UPDATE_PROJECT API:

UPDATE_PROJECT API is used to make changes to name of the project . It uses the following request parameters .

Token: This represents a unique token to verify the user who will be updating the project.

Project_Id : This id represents the project 's unique DB id to which we are updating the name.

Name: The name to be updated for the particular project .

Once the above parameters are received and the user token is verified in the Mongo DB, project_id is used to check if that projects exists in the database. After the project_id is verified ,
the project is updated with the new name received from the request .
A corresponding JSON response is sent denoting the success or failure of the function .

DELETE_TASK API :
DELETE_TASK API is used to delete task from the task collection in the Mongo DB. It uses the following request parameters .
Token: This represents a unique token to verify the user who will be deleting the task .
Task_id: This id represents the task's unique DB id which is going to be deleted .
Once the above parameters are received and the user token is verified in the Mongo DB, task_id is used to check if that particular task exists . After the task_id is verified , the task is removed from the task collection in Mongo DB .

Calendar UI (Rohit More & Apoorv Saini)

In order to display a calendar with a list of agendas and task status, we have made use of a third part reactjs library called react-big-calendar. react-big-calendar is full featured Calendar component for managing events and dates. It uses modern flexbox for layout making it super responsive and performant. Leaving most of the layout heavy lifting to the browser. One important thing to note is that the default styles use height: 100% which means you should define an explicit height for the container. We have set the explicit height and added other styling changes by editing the css file found in the following location:
react-big-calendar/lib/css/react-big-calendar.css
We have made use of Moment.js for Date internationalization and localization as mentioned below:

```
import BigCalendar from 'react-big-calendar';
import moment from 'moment';

// Setup the localizer by providing the moment (or globalize) Object
// to the correct localizer.
BigCalendar.momentLocalizer(moment); // or globalizeLocalizer
```

By default, we have set the calendar view of our project as monthly view. We have obtained the list of events and agenda from our api's in json format and then passed that list to events to 'events' props in BigCalendar component.

```
const MyCalendar = props => (
  <div>
    <BigCalendar
      events={myEventsList}
      startAccessor='startDate'
      endAccessor='endDate'
    />
  </div>
);
```

The json format of the list of events obtained from our api's is as below:

```
var myEventsList = [{
    'title': 'Conference',
    'start': new Date(2018, 3, 14),
    'end': new Date(2018, 3, 15),
    desc: 'Conference 1'
    },
    {
        'title': 'Meeting',
        'start': new Date(2018, 4, 16, 10, 30, 0, 0),
        'end': new Date(2018, 4, 17, 12, 30, 0, 0),
        desc: 'Important meeting'
    },
    {
        'title': 'Lunch',
        'start':new Date(2017, 3, 12, 12, 0, 0, 0),
        'end': new Date(2017, 3, 12, 13, 0, 0, 0),
        desc: 'Power lunch'
    }]
```

In the above list, we have mentioned the title and description of events along with their start and end date respectively. By default, the key for start and end date should be 'start' and 'end' respectively. However, if you want to give your own key, you can effortlessly do so by passing your own keys in startAccessor and endAccessor props of BigCalender component. In the following snippet, We have kept the key for start date as 'startDate'

```
const MyCalendar = props => (
  <div>
    <BigCalendar
      events={myEventsList}
      startAccessor='startDate'
      endAccessor='endDate'
    />
  </div>
);
```

Finally, we have also enabled calendar view based on days, weeks and agenda which provides varied perspectives of different tasks which can be ultimately useful for a better understanding of different tasks performed by different team members.

Final Work Distribution