EE 547 – Applied and Cloud Computing

Project Report

Aravindh Soundararajan, Jacqueline Liu, Lifuling Wei

December 13, 2022

Project Title: Mood-themed Social Media Platform (InstaMood)

Figure 1: LOGO

Contents

1 Summary / Description

This project's objective is to provide a platform that users can use as an online personal notebook as well as social media to interact with friends. The websites are composed of different web pages, each representing a mood that the user wants to present. Every mood has a web page designed for it.

We do not intend to solve a specific problem or target certain audiences, but to build something artistic, relaxing, and interesting. Anyone is welcome to use the website and explore its functionalities.

2 Timeline - Retrospective

Monday of the Week	Progress
Nov. 14th	• Brainstorming & Planning
Nov. 21st	• Backend: schema.graphql & server set-up • Frontend: Log in, Sign Up, Mood Menu & client set-up
Nov. 28th	• Backend: resolvers • Frontend: Development of remaining pages
Dec. 5th	• Frontend & Backend Connection • Debugging & Testing • Report
Dec. 12th	• Report, deploy, demo slides & video

Figure 2: Timeline - Retrospective

3 Architecture

Our project interfaces over GraphQL and apollo react-hooks. It is a full-stack service with its corresponding UI and back-end components, all hosted in Amazon Elastic Compute Cloud (EC2) instance. Our back-end server is programmed in NodeJS with GraphQL API, and a MongoDB database. We designed nine distinct web pages as listed below. The combination of static assets and dynamic content is handled via React, such as $useState$ and $useEffect$. Web pages are linked and navigated via react-router-dom. Figure 3 and 4 show the diagrams of the overall architecture and user flow.

3.1 Home/Log-in

Views User Login

Features Session validation

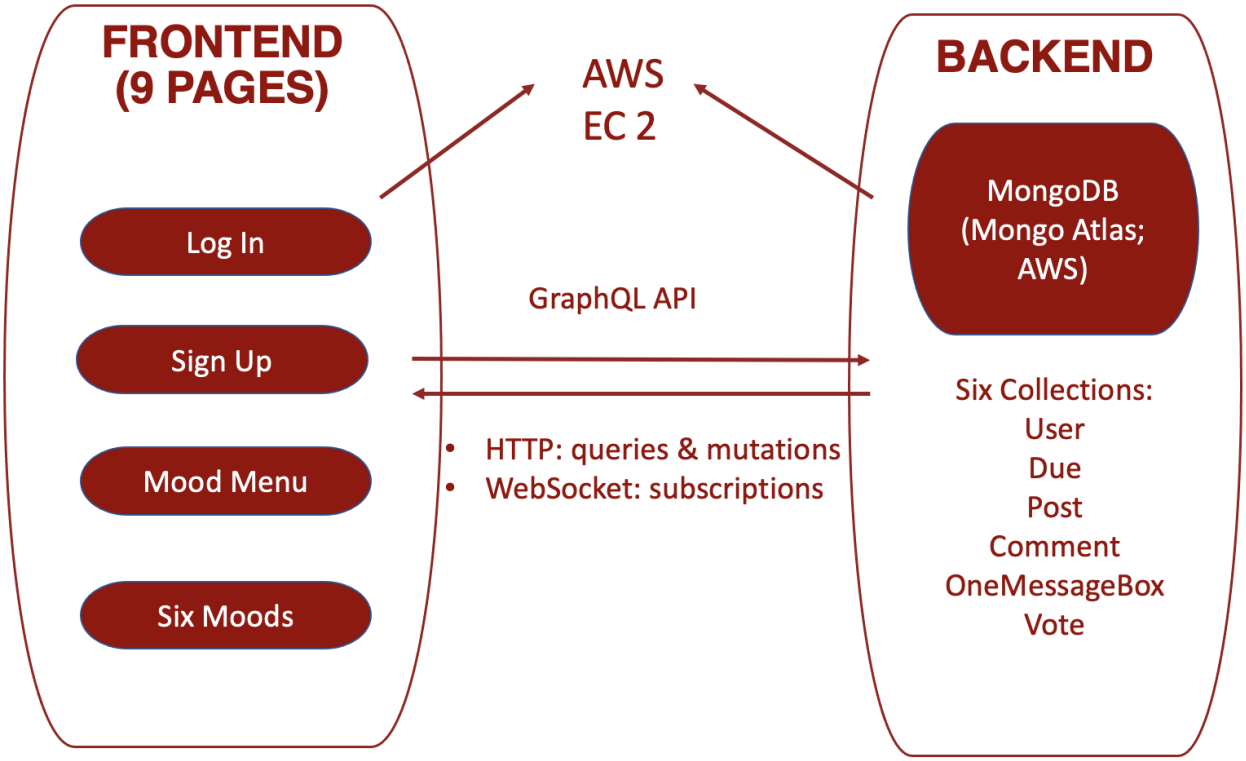


Figure 3: Overall Architecture

3.2 Sign-up

Views Forms to be filled out with personal and account information

Features Session validation

3.3 Mood Menu

Views A menu of six items

Features -Pure front-end design
- Mood descriptions will show when hovered.

3.4 Mood1 - Loads of Deadlines but Only Wanna Be a Couch Potato

Views A blackboard with a place to type deadlines

Features - Private space
- Each item is draggable and random-colored.
- Will automatically show previous items if any
- Can add or delete any number of items and will immediately rerender accordingly

3.5 Mood2 - Extreme Sleepiness

Views An animation of swinging light bulb that can be switched on and off via clicking

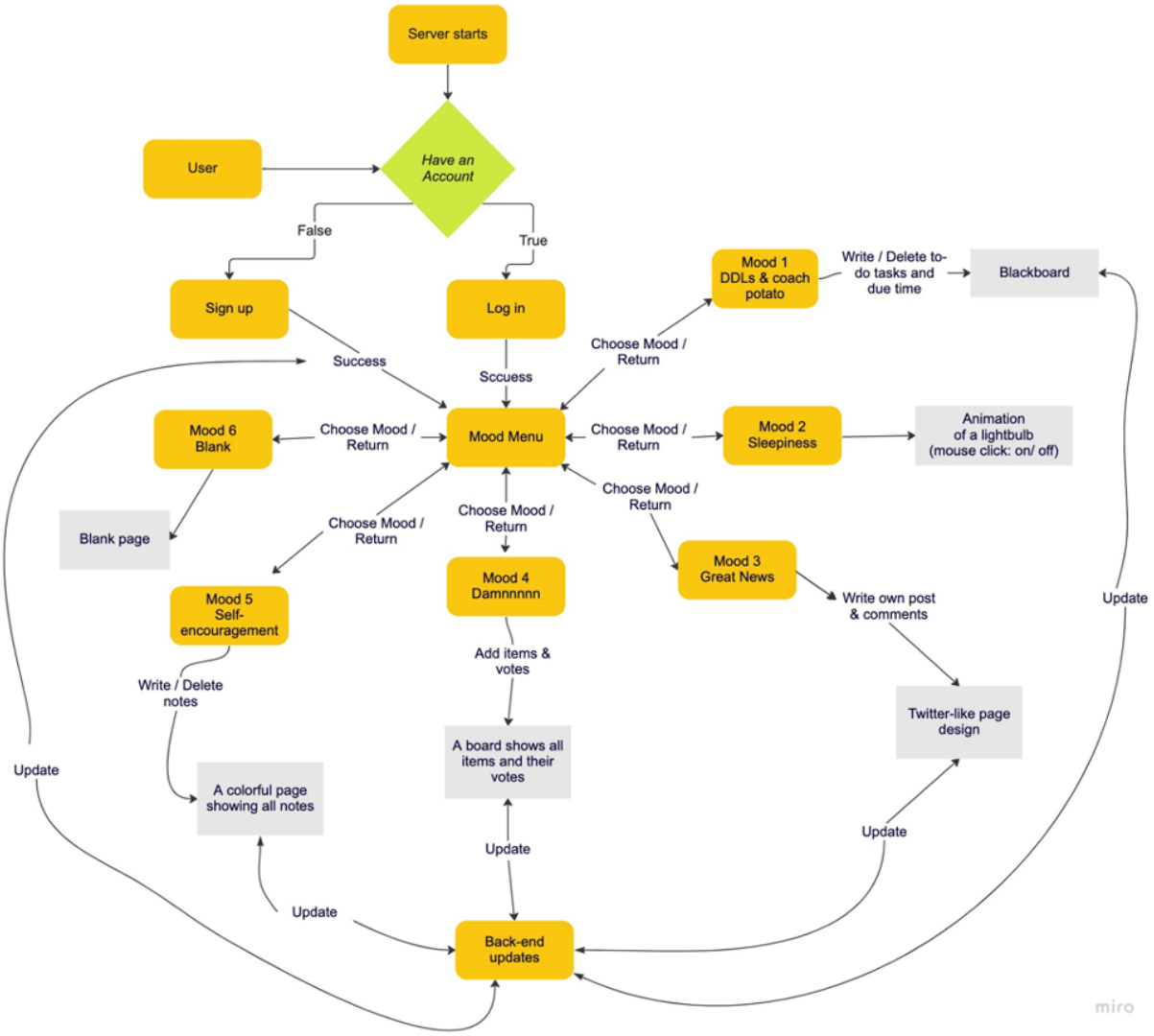


Figure 4: User Flow Diagram

Features - Private space
- Pure front-end design

3.6 Mood3 - Some Great News

Views Twitter-like design

Features - Shareable space
- Each user can have only one post but multiple comments.
- Any user's updates will simultaneously display on all other users' pages.

3.7 Mood4 - Damnnnnn

Views A bulletin board with a place to type and vote B.S. stuff

Features - Anonymously shareable space
- Each user can create multiple items anonymously and give votes to multiple items.
- Items can be deleted only if they are created by the user deleting them.
- Any user's updates will simultaneously display on all other users' pages.

3.8 Mood5 - I Can Do It

Views A page where users can add/delete any number of yellow notes

Features - Private space
- Each note is draggable.
- Will automatically show previous notes if any
- Adding/Deleting a note will immediately rerender accordingly.

3.9 Mood6 - Blank

Views A very blank page with one sentence reading: "Mind Blanking turns the complicated to the simple. You embrace the entirety."

Features - Private space
- Pure front-end design

4 Implementation

This is a fairly front-end-intensive project. To design the web app, a number of frameworks were used for styling and dynamic content:

- antd (Ant Design): A React UI library for building rich, interactive user interfaces. We used components including Modal, Form, Input, and message for alerting messages.
- react-draggable: To add dragging movement to React elements.
- Bootstrap: To generate random colors.

In order to connect to the backend API, we set up Apollo GraphQL client as in Figure 5.

- The HTTP link is for queries and mutations.
- The WebSocket link is for subscriptions.
- With the *split* function, we can tell apart the types of GraphQL requests. We then send the requests to different endpoints accordingly.

For back-end and database, we didn't apply any decorator. We applied validators in the Sign-up page to fields including phone number, email, and password (See Figure 8).

4.1 Home/Log-in

The page consists of two main parts: header and body. The page header contains a navigator component on the left side, to switch between log-in and sign-up pages. For the body of the page, on the left side is our self-designed logo, whereas a basic login form sits on the right. Our log-in page with an example alert is shown below in Figure 6. Input validation is handled by GraphQL

```
import { ApolloClient, InMemoryCache, ApolloProvider, split, HttpLink } from '@apollo/client';
import { createClient } from 'graphql-ws';
import { GraphQLWsLink } from '@apollo/client/link/subscriptions';

const httpLink = new HttpLink({
  uri: 'http://localhost:5000/graphql'
});

const wsLink = new GraphQLWsLink(createClient({
  url: 'ws://localhost:5000/graphql',
}));

const splitLink = split(
  ({ query }) => {
    const definition = getMainDefinition(query);
    return (
      definition.kind === 'OperationDefinition' &&
      definition.operation === 'subscription'
    );
  },
  wsLink,
  httpLink,
);

const client = new ApolloClient({
  link: splitLink,
  cache: new InMemoryCache()
});
```

Figure 5: Apollo GraphQL Client

Query operation.
If a user forgets his/her username and password, he/she has to type the email that is associated with the account. If email can't be provided, he/she can call for help. (See Figure 7. This is just a simulation; no actual link will be sent.)

4.2 Sign-up

The layout is very similar to that of Log-in page, as described in section 4.1. Click on "InstaMood" in the header will return to Log-in page. There are two forms in the body: a form where a first-time user has to type his/her phone and email, and the other form where he/she has to type his/her unique username and password. The sign-up page is shown below in Figure 8. A successful registration should satisfy all those requirements in red and turn them into green.
All user data are stored in a MongoDB collection named 'users'. The user schema with all fields is shown below in Figure 9.

4.3 Mood-Menu

A successful log-in or sign-up will lead to this page, whose layout is very similar to that of the Log-in page, as described in section 4.1. Click on "InstaMood" in the header will return to the Log-in page. The current user's username is also displayed in the header. When the very centered button

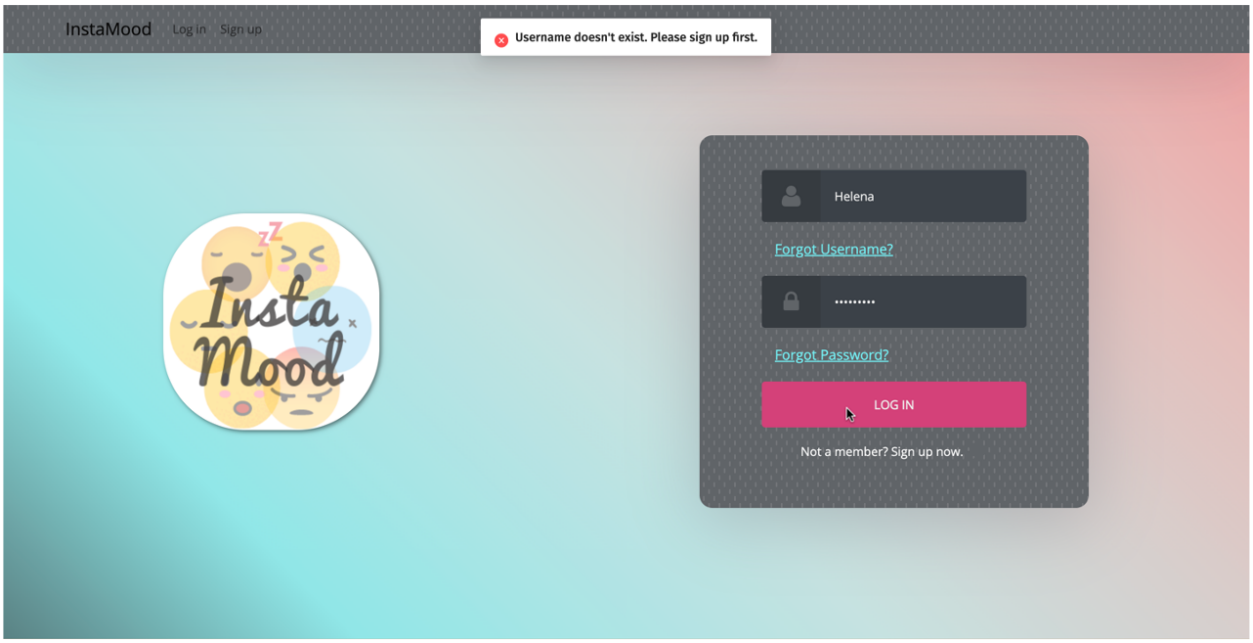


Figure 6: User Login Form and Input Validation

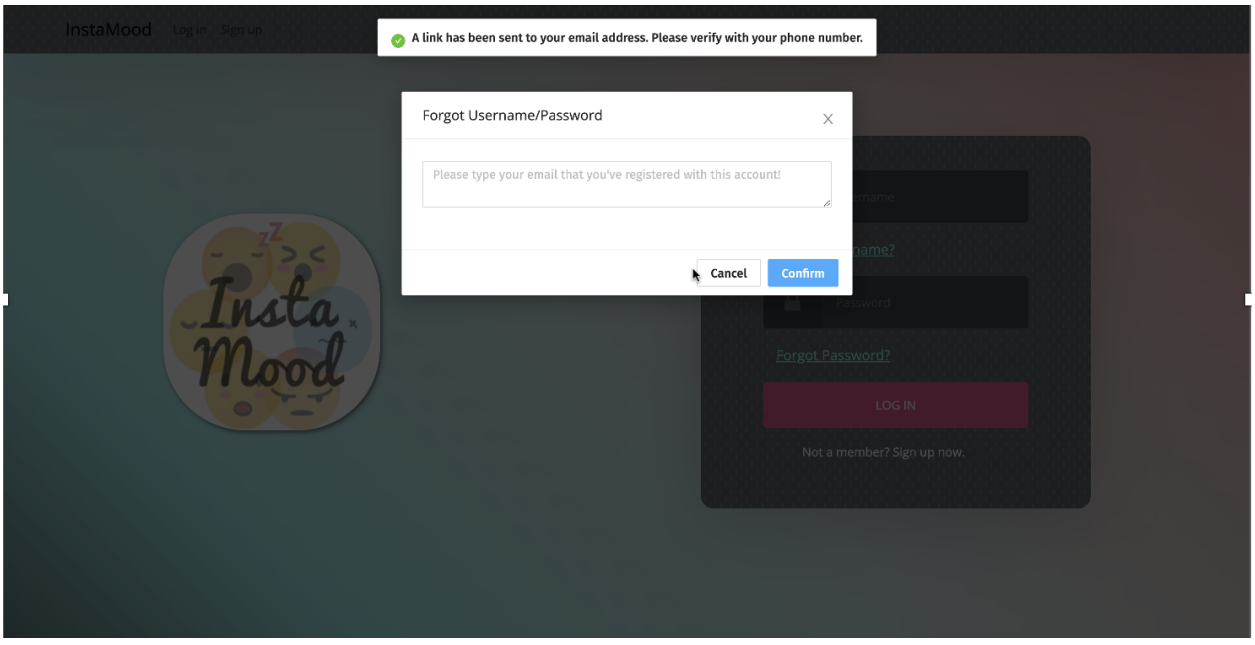


Figure 7: User Login: Forgot Username/Password

is clicked on, six other buttons will pop up around it, as shown below in Figure 10. Hovering upon buttons on the circumference will display descriptions about the corresponding mood; clicking on them will link to distinct pages. Click on the centered "x" will constrict all six buttons into the center.

The following sections describe each mood and web page in detail in a clockwise manner starting from the top.

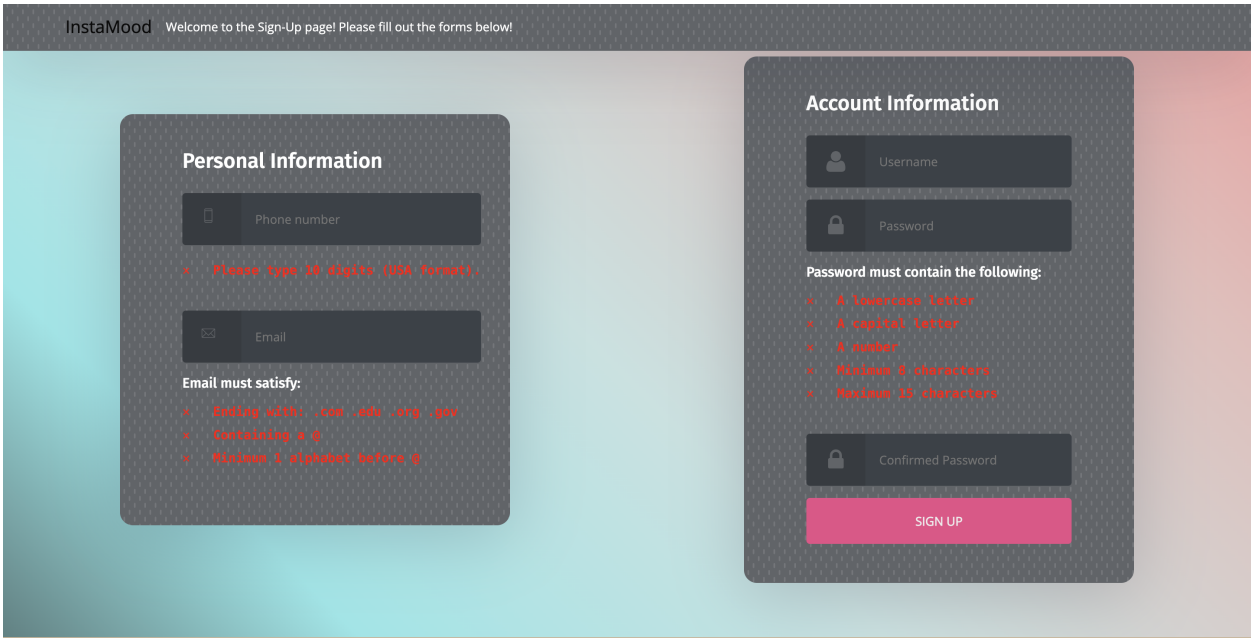


Figure 8: User Sign-up Page

User Schema	
Fieldname	Type
_id	ID!
name	String!
password	String!
phone	String!
email	String!

Figure 9: User Schema in MongoDB

4.4 Mood1 - Loads of Deadlines but Only Wanna Be a Couch Potato

Corresponding mood statement in Mood Menu: Let's keep track of your deadlines!
When this page loads, a fetch to the database is executed. The items are then loaded for display. It's a basic blackboard design as shown below in Figure 11. Users can type deadlines at the bottom and then click on the "Enter" button or press the "Enter" key on his/her keyboard. This action will check whether the typing fields are non-empty, store the data, and pop up an item on the screen, which is random-colored and draggable. Users can delete an item by clicking on the red button on the top-right.
All deadline data are stored in a MongoDB collection named 'dues'. The due schema with all fields is shown below in Figure 12.

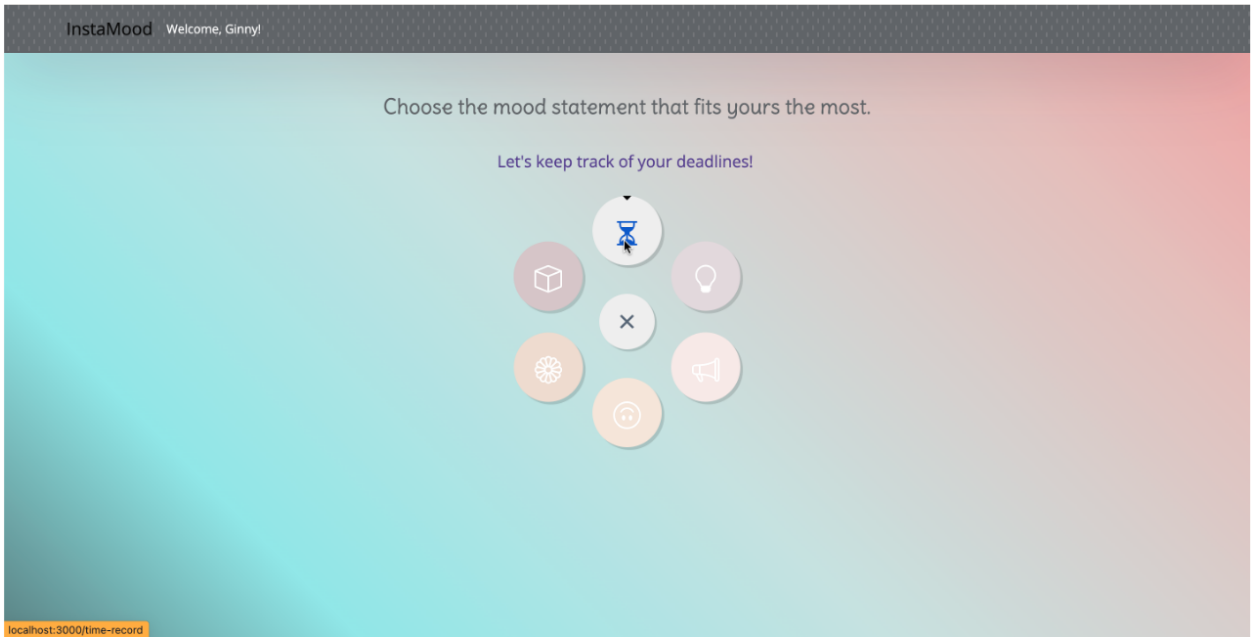


Figure 10: Mood Menu

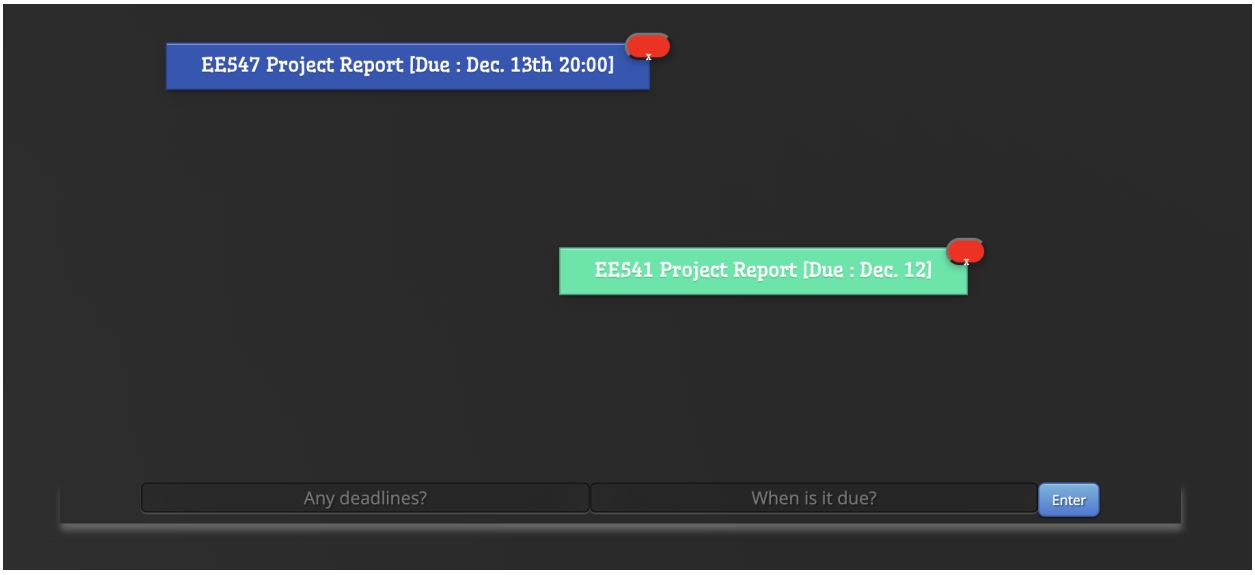


Figure 11: Web page for Mood 1

4.5 Mood2 - Extreme Sleepiness

Corresponding mood statement in Mood Menu: "Feeling sleepy?"
Users will see a swinging light upon entering this page, as shown in Figure 13a. Users are welcomed to click twice on the bulb to turn it off, as shown in Figure 13b, and click once again to turn it on.

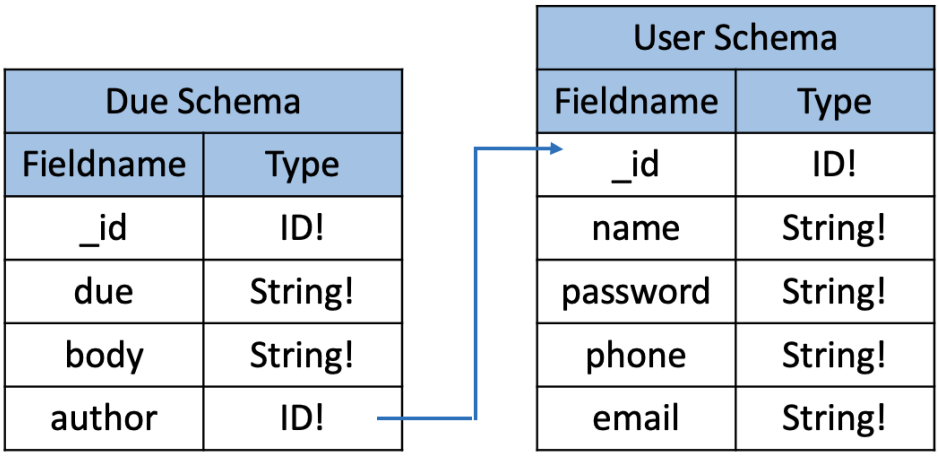


Figure 12: Due Schema in MongoDB

4.6 Mood3 - Some Great News

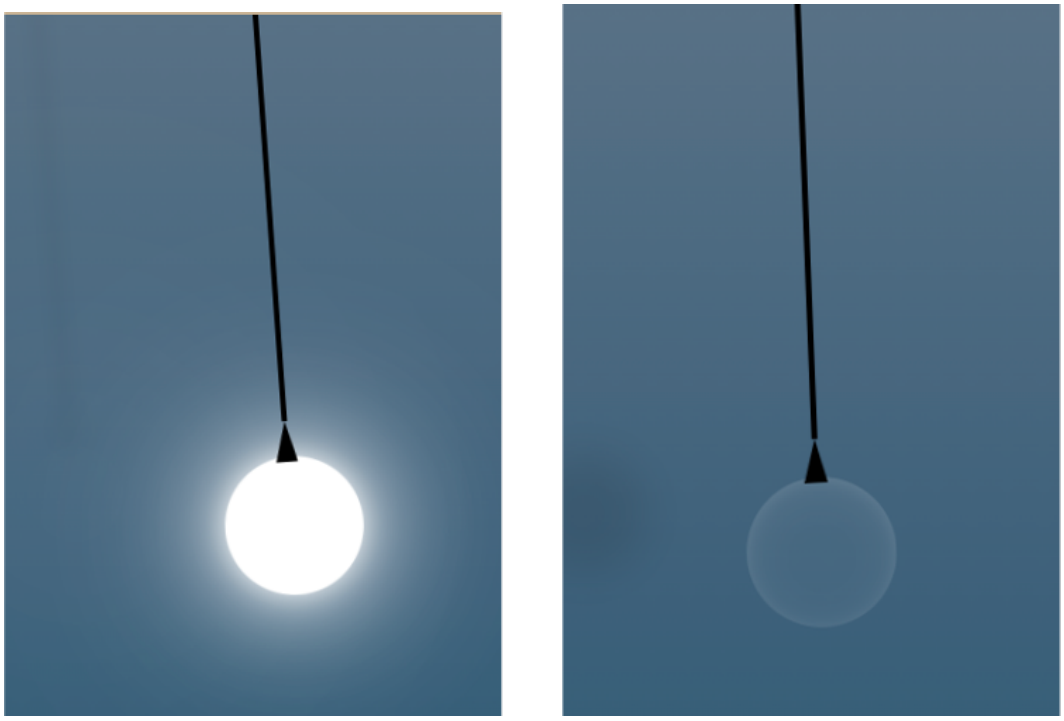
Corresponding mood statement in Mood Menu: "Share some great news!"
When this page loads, a fetch to the database is executed. The items are then loaded for display. Users can type something into the box on the top and click on the "Post" button. This action will check whether the text is non-empty, store the data, and render a post in the screen. The post contains a default picture, the author's username, the text that has been typed, the time when the post is generated, and a zero that indicates zero comments. The post is viewable by all registered users. Any registered user, including the author, can add any number of comments to any viewable post. Comments will then be immediately displayed, and the number of comments will increase, as shown in Figure 14. Note that each registered user is allowed to have only one post, so if he/she types something again and click on the "Post" button, his/her previous post will be replaced by the new one, and previous comments will also be cleared.
All post data are stored in a MongoDB collection named 'posts'. The post schema with all fields is shown below in Figure 15.

4.7 Mood4 - Damnnnnn

Corresponding mood statement in Mood Menu: "Shitty things happened?"
When this page loads, a fetch to the database is executed. The items are then loaded for display. Users can type any amount of B.S. stuff on the top and it will be displayed immediately and anonymously as shown in Figure 16. All items are viewable to any registered user. Any registered user can provide any number of votes but can only delete items created by themselves.
All the above data are stored in a MongoDB collection named 'votes'. The vote schema with all fields is shown below in Figure 17.

4.8 Mood5 - I Can Do It

Corresponding mood statement in Mood Menu: "Encourage yourself!"
When this page loads, a fetch to the database is executed. The items are then loaded for display. Users can add any number of notes by clicking on the green button "Add" on the top-right of the



(a) Light On (Click Once) (b) Light Off (Click Twice)

Figure 13: Animation design for Mood 2

web page. Each note is draggable. When hovering upon a note, a red "remove" button will appear; users can delete a note by clicking on it. (See Figure 18)
All data of notes are stored in a MongoDB collection named 'onemessageboxes'. The schema with all fields is shown below in Figure 19.

4.9 Mood6 - Blank

Corresponding mood statement in Mood Menu: "Feeling empty?"
Upon entering this page, users will see words popping up one-by-one gradually into a sentence as shown in Figure 20.

5 Planned Unimplemented Features

- User sign-out
- A deletion confirmation box before any item is truly deleted.
- A place where users can upload their headshots.
- Enabling users to upload images/videos aside from text when posting.
- Post deletion, post reshare, and comment deletion.
- Mechanism of friends and privacy policies.
- Chatroom

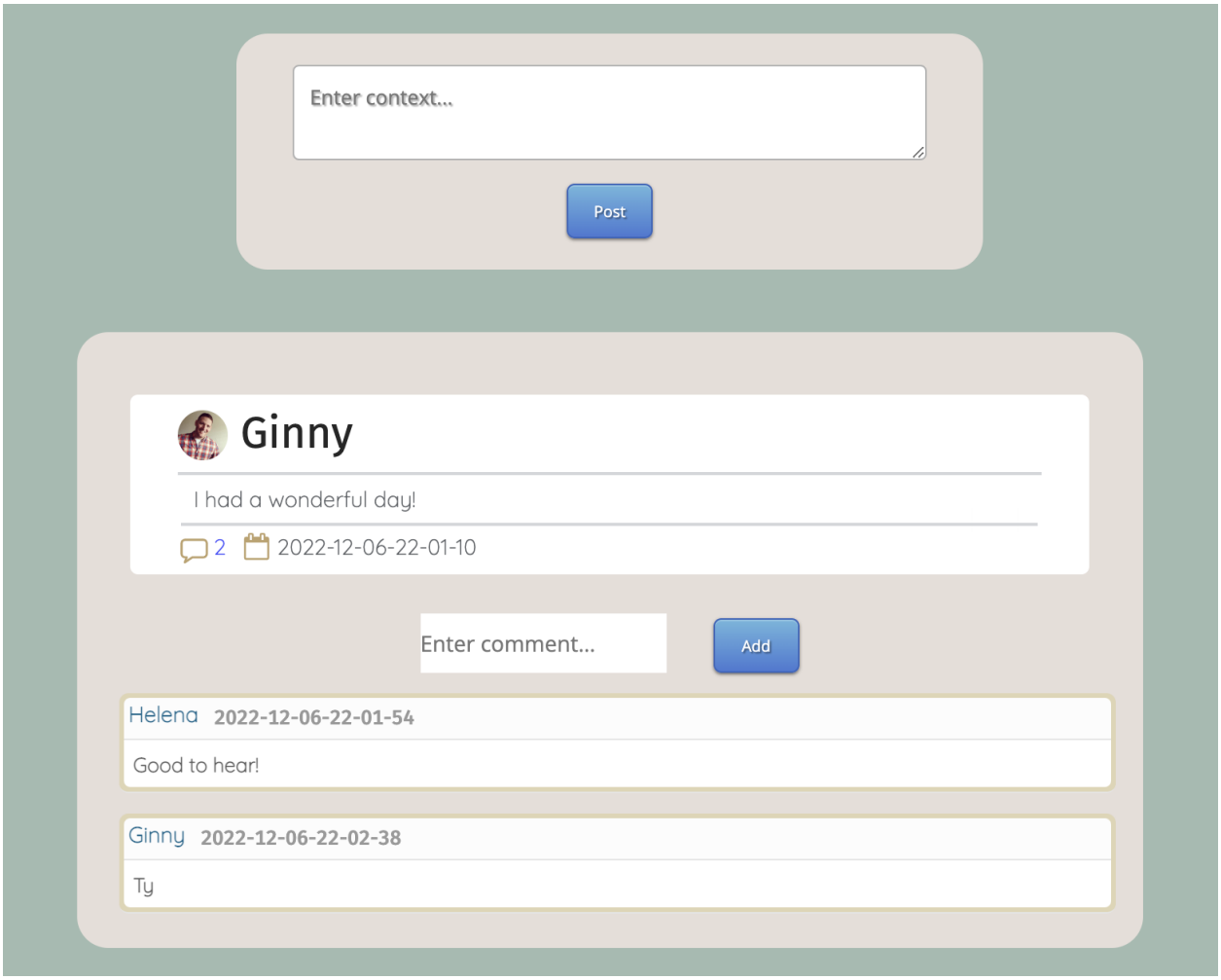


Figure 14: Design for Mood 3

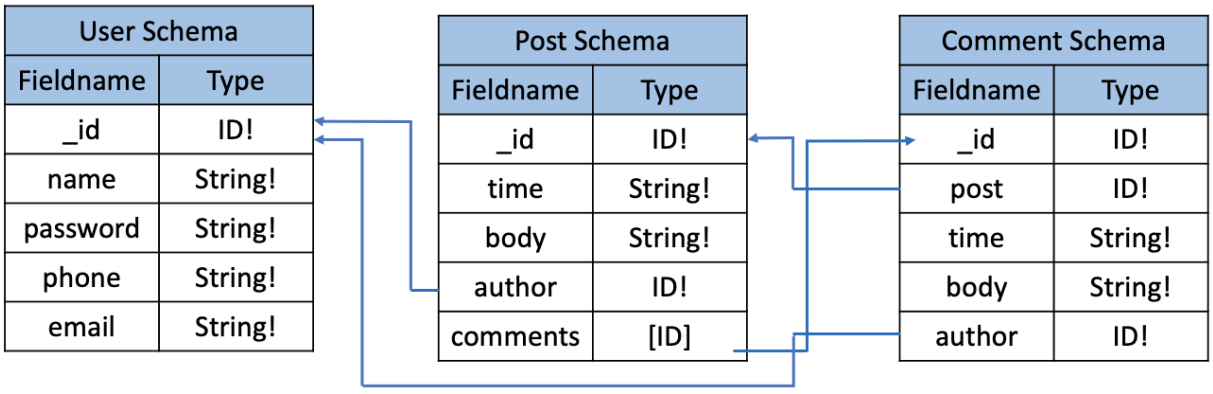


Figure 15: Post Schema in MongoDB

- Background music

Figure 16: Web page for Mood 4

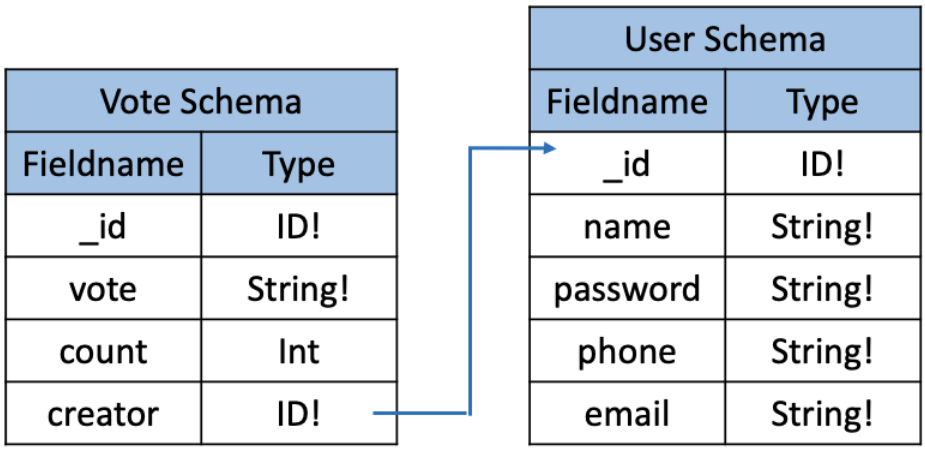


Figure 17: Vote Schema in MongoDB

6 Conclusions

First, to answer what we fundamentally misunderstood before starting the project, our response is the implementation took far more time than we would imagined or planned. For instance, we thought the log-in/sign-up thing would take about three hours to complete, but then we spent almost three days getting everything right.

Thanks to GraphiQL, the front-end and the back-end development can be undertaken simultaneously and independently once we had a rough layout design and logistics. The most intractable part lies in integrating them together via apollo. There were countless frustrating moments when back-end database worked fine but front-end wasn't rendering properly.

Figure 18: Web page for Mood 5

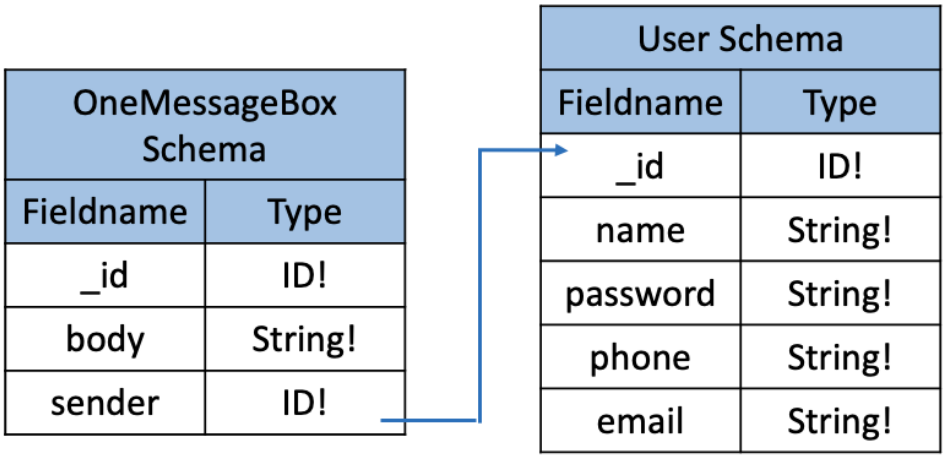


Figure 19: OneMessageBox Schema in MongoDB

However, we believe that's still a rewarding process because we're getting better at debugging, and seeing our project growing more and more stable and completed was really uplifting. We learned a lot while carving out front-end design, which was pretty fun, and got a lot familiar with GraphQL while implementing our logistics and schema.
Another major challenge is that since we're building a web service, there are actually endless details we can take care of. It's fairly challenging to decide which should be implemented first so that our project won't look crappy eventually given a limited time frame.

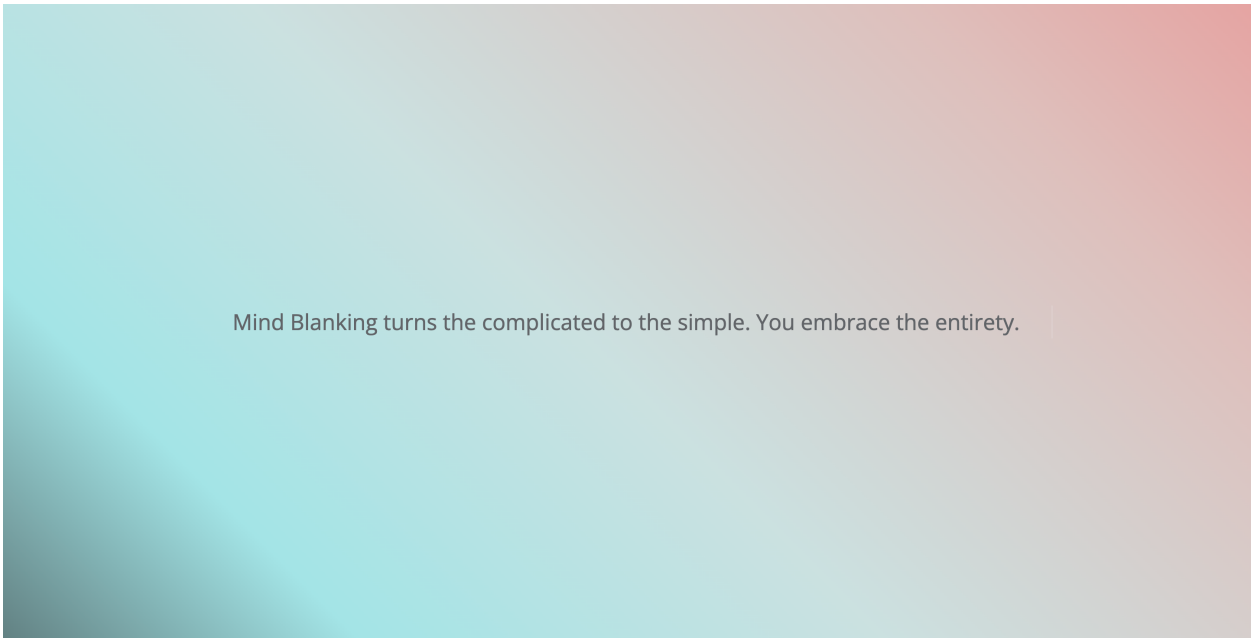


Figure 20: Web page for Mood 6

7 References, Tutorials, Codebases, Documentation, and Libraries

- Frontend: antd, bootstrap, apollo, react-router-dom, reactjs, graphql-ws
- Backend: mongoose, nodejs, express, express-graphql, graphql, graphql-ws. ws, cors, dotenv-defaults
- React tutorial
- @apollo/react-hooks
- Amazon Elastic Compute Cloud (Amazon EC2)

8 Compute Needs

- **We use MongoDB Atlas as our database.** MongoDB Atlas is a fully-managed cloud database that handles all the complexity of deploying, managing, and healing your deployments on the cloud service provider (we chose AWS).
- **We use Amazon EC2 to host the application.** We created the Amazon EC2 instance to host our project, link to the project: `3.101.146.208`.

9 Team Roles

The following is the rough breakdown of roles and responsibilities we plan for our team:

- Lifuling: Backend server design and APIs. Primary: GraphQL resolvers and database. Secondary: Assist with HTML.

- Aravindh: Primary: HTML page design, review the node.js code. Secondary: Set up the project on AWS.
- Jacqueline: Client UX and UI. Primary: build client side interface for API and create rough pages with dynamic server content. Secondary: Assist backend coding

All team members will work on the final presentation, slides, and report.

10 Appendix: GraphQL API

10.1 schema.graphql

```
type Query {
  users(name: String): [User!]
  signIn(name: String!, password: String!): SignInType
  posts(author: String): [Post!]
  dues(author: String!): [Due!]
  onemessageboxes(sender: String!): [Message!]
  votes(creator: String): [Vote!]
}

type Mutation {
  createUser(data: CreateUserInput!): SignUpType
  checkUser(email: String!): CheckUserType

  createPost(data: CreatePostInput!): Post!
  createComment(data: CreateCommentInput!): Comment!

  createDue(data: CreateDueInput!): Due!
  deleteDue(_id: ID!, author: String!): Due!

  createOneMessage(sender: String!, body: String!): Message!
  deleteOneMessage(_id: ID!, sender: String!): Message!

  createVote(data: CreateVoteInput!): Vote!
  updateVote(data: CreateVoteInput!): Vote!
  deleteVote(_id: ID!, creator: String!): Vote!
}

type Subscription {
  user: UserSubscriptionPayload!
  due(author: String!): DueSubscriptionPayload!
  onemessagebox(sender: String!): OnemessageboxSubscriptionPayload!
  post: PostSubscriptionPayload!
  vote: VoteSubscriptionPayload!
}

type User {
  name: String!
```

```
38    password: String!
39    phone: String!
40    email: String!
41  }
42
43  type Friend{
44    _id: ID!
45    name: String!
46    today: Int!
47  }
48
49  type Message {
50    _id: ID!
51    sender: Author!
52    body: String!
53  }
54
55  type Due {
56    _id: ID!
57    due: String!
58    body: String!
59    author: Author!
60  }
61
62  type Post {
63    _id: ID!
64    time: String!
65    body: String!
66    author: Author!
67    comments: [Comment!]
68  }
69
70  type Comment {
71    _id: ID!
72    post: Author!
73    time: String!
74    body: String!
75    author: Author!
76  }
77
78  type Author {
79    name: String!
80  }
81
82  type Vote {
83    _id: ID!
84    vote: String!
85    creator: Author!
```

```
  count: Int!
}

type UserSubscriptionPayload {
  mutation: MutationType!
  data: User!
}

type PostSubscriptionPayload {
  mutation: MutationType!
  data: Post!
}

type DueSubscriptionPayload {
  mutation: MutationType!
  data: Due!
}

type OnemessageboxSubscriptionPayload {
  mutation: MutationType!
  data: Message!
}

type VoteSubscriptionPayload {
  mutation: MutationType!
  data: Vote!
}

input CreateUserInput {
  name: String!
  password: String!
  phone: String!
  email: String!
}

input UpdateUserInput {
  name: String!
  friends: [String!]
  mood: [Int!]
  today: Int!
}

input CreatePostInput {
  body: String!
  author: String!
}

input CreateDueInput {
```

```
134   due: String!
135   body: String!
136   author: String!
137 }
138
139 input CreateCommentInput {
140   postId: ID!
141   postAuthor: String!
142   body: String!
143   author: String!
144 }
145
146 input DeleteCommentInput {
147   type: Int!
148   postId: Int!
149   postAuthor:String!
150   commentId: Int!
151   author: String!
152 }
153
154 input CreateVoteInput {
155   vote: String!
156   creator: String!
157 }
158
159 enum MutationType {
160   CREATED
161   UPDATED
162   DELETED
163   ADDED_COMMENT
164   DELETED_COMMENT
165 }
166
167 enum SignInType {
168   SUCCESS
169   USER_NOT_FOUND
170   USER_EXISTS
171   NOT_MATCH
172 }
173
174 enum SignUpType {
175   SUCCESS
176   USER_EXISTS
177 }
178
179 enum CheckUserType {
180   EMAIL_EXIST
181   EMAIL_NOT_EXIST
```

```
}
```

10.2 Query.js

```
import db from './db';
const mongoose = require('mongoose');
const Query = {
  async users(parent, {name}, context, info) {
      if (!name) {
        return await db.UserModel.find({}).
          collation({locale: "en"}).
          sort({name : 1}).
          select('-password')
      }
      let ul = await db.UserModel.find({name: name}).select('-password')

      console.log("===Query===")
      console.log(ul)
      return ul
  },

  async signIn(parent, {name, password}, context, info){
    try{
      if(!name || !password) return;

      let user = await db.UserModel.findOne({name: name}).select('name password');
      console.log("===Query:SignIn===");
      console.log(user);

      if(user) {
        if (user.password === password) return "SUCCESS";
        else return "WRONG_PASSWORD";
      } else {
        return "USER_NOT_FOUND";
      }
    } catch(e) {return "USER_NOT_FOUND";}
  },

  async posts(parent, {author}, context, info) {
    try{
      const post = await db.PostModel.find({});
      if (!post) throw ("Post Not found")
      console.log("===Query:Posts===");
      console.log(post);
      return post
    } catch(e) {console.log(e)}
  },

```

```
45    async dues(parent, {author}, context, info) {
46      try{
47        if (!author) throw new Error("Missing author.");
48        const user = await db.UserModel.findOne({name: author});
49        const due_post = await db.DueModel.find({author: user});
50        if (!user || !due_post) throw ("User or post not found.");
51        console.log("===Query:Due Posts===");
52        console.log(due_post);
53        return due_post;
54      } catch(e) {console.log(e)}
55    },
56
57    async onemessageboxes(parent, {sender}, context, info){
58      try{
59        if (!sender) throw new Error("Missing sender.");
60        const user = await db.UserModel.findOne({name: sender});
61        const message = await db.OneMessageBoxModel.find({sender: user});
62        if (!user || !message) throw ("User or message not found.");
63        console.log("===Query:Message===");
64        console.log(message);
65        return message;
66      } catch(e) {console.log(e)}
67    },
68
69    async votes(parent, {creator}, context, info){
70      try{
71        const vote = await db.VoteModel.find({});
72        if (!vote) throw ("Vote Not Found")
73        console.log("===Query:Votes===");
74        console.log(vote);
75        return vote;
76      } catch(e) {console.log(e)}
77    },
78  }
79
80  export default Query;
81
```

10.3 Mutation.js

```
1  import pubsub from './pubsub';
2  import db from './db';
3  const mongoose = require('mongoose');
4  mongoose.set('useFindAndModify', false);
5
6  async function newUser(db, data){
7      return await new db.UserModel(data).save();
8  }
```

```
9
10 const Mutation = {
11     async createUser(parent, { data }, context, info) {
12         try{
13             if (!data.name) {throw new Error('Missing username');}
14             if (!data.password) {throw new Error('Missing password');}
15             if (!data.phone) {throw new Error('Missing phone');}
16             if (!data.email) {throw new Error('Missing email');}
17             const existing = await db.UserModel.findOne({ name: data.name});
18             if (existing) {return "USER_EXISTS"}
19             const userdata = {
20                 name: data.name,
21                 password: data.password,
22                 phone: data.phone,
23                 email: data.email
24             };
25
26             const user = await newUser(db, userdata);
27
28             console.log("===Mutation: createUser===")
29             console.log(user)
30             pubsub.publish('user', {
31                 user: {
32                     mutation: 'CREATED',
33                     data: user,
34                 },
35             });
36
37             return "SUCCESS"
38
39         }  catch(e){console.log(e)}
40     },
41
42     async checkUser(parent, { email }, context, info) {
43         try{
44             if (!email) {throw new Error('Missing email');}
45             const existing = await db.UserModel.findOne({ email: email});
46             if (existing) {return "EMAIL_EXIST"}
47             else {return "EMAIL_NOT_EXIST"}
48
49         }  catch(e){console.log(e)}
50     },
51     async createDue(parent, {data}, context, info) {
52         try {
53             const {due, body, author} = data;
54             const user = await db.UserModel.findOne({name: author});
55
56             if (!user) { throw new Error('User not found'); }
```

```
57              const due_post = {
58                  due: due,
59                  body: body,
60                  author: user
61              };
62              await new db.DueModel(due_post).save();
63              pubsub.publish('due'+`${author}`, {
64                  due: {
65                      mutation: 'CREATED',
66                      data: due_post,
67                  },
68              });
69              console.log("===Mutation: createDue===");
70              console.log(due_post);
71              return due_post;
72
73          }catch(e) {console.log(e)}
74      },
75      async deleteDue(parent, {_id, author}, context, info) {
76          try{
77              const user = await db.UserModel.findOne({name: author})
78              if(!user) throw('User not found')
79              const due = await db.DueModel.findOne({_id: _id, author: user});
80              if(!due) throw('Due not found')
81
82              await db.DueModel.deleteOne({_id: _id, author: user});
83
84              pubsub.publish('due'+`${author}`, {
85                  due: {
86                      mutation: 'DELETED',
87                      data: due,
88                  },
89              });
90
91              return due
92          } catch(e) {console.log(e)}
93      },
94      async createPost(parent, {data}, context, info) {
95          try {
96              const {body, author} = data;
97              const user = await db.UserModel.findOne({name: author});
98
99              if (!user) { throw new Error('User not found');}
100             var tzoffset = (new Date()).getTimezoneOffset() * 60000; //offset in milliseconds
101             var localISOTime = (new Date(Date.now() - tzoffset)).toISOString().slice(0, -1);
102             var timeStamp = localISOTime.substring(0, 10) + '-' + localISOTime.substring(11, 1
103             const post = {
104                 time: timeStamp,
```

```
105                 body: body,
106                 author: user,
107                 comments: []
108             };
109             await db.PostModel.updateOne({author: user},post,{upsert: true});
110             pubsub.publish('post', {
111                 post: {
112                     mutation: 'CREATED',
113                     data: post,
114                 },
115             });
116             console.log("===Mutation: createPost===")
117             console.log(post);
118             return post;
119 
120         }catch(e) {console.log(e)}
121     },
122     async createComment (parent, {data}, context, info) {
123         try {
124             const {postId, postAuthor, body, author} = data;
125             const postUser = await db.UserModel.findOne({name: postAuthor})
126             const commentUser = await db.UserModel.findOne({name: author})
127             if (!postUser) throw (`${postAuthor} not found`)
128             if (!commentUser) throw (`${author} not found`)
129 
130             const post = await db.PostModel.findOne({_id:postId, author: postUser})
131             if(!post) throw ("Post not found")
132 
133             var tzoffset = (new Date()).getTimezoneOffset() * 60000; //offset in milliseconds
134             var localISOTime = (new Date(Date.now() - tzoffset)).toISOString().slice(0, -1);
135             var timeStamp = localISOTime.substring(0, 10) + '-' + localISOTime.substring(11, 13
136 
137             const comment = new db.CommentModel({
138                 post: post,
139                 time: timeStamp,
140                 body: body,
141                 author: commentUser
142             })
143             await comment.save()
144             await db.PostModel.updateOne(
145                 {_id:postId, author: postUser},
146                 {$push: {comments: comment}}
147             );
148             console.log("===Mutation: createComment===")
149             pubsub.publish('post', {
150                 post: {
151                     mutation: 'ADDED_COMMENT',
152                     data: {
```

```
153                         id: post.id,
154                         time: post.time,
155                         body: post.body,
156                         author: post.author,
157                         comments: [comment]
158                     },
159                 },
160             });
161             return comment
162 
163         } catch (e) {console.log(e)}
164     },
165     async createOneMessage(parent, {sender, body}, context, info) {
166         try{
167             const user = await db.UserModel.findOne({name: sender})
168             if(!user) throw ("User not found")
169             const message = new db.OneMessageBoxModel({
170                 sender: user,
171                 body: body
172             })
173             await message.save();
174             pubsub.publish('message'+`${sender}`, {
175                 onemessagebox: {
176                     mutation: 'CREATED',
177                     data: message,
178                 },
179             });
180             console.log("===Mutation: CreateOneMessage===");
181             console.log(message);
182             return message;
183 
184         }  catch(e) {console.log(e)}
185     },
186 
187     async deleteOneMessage(parent, {_id, sender}, context, info) {
188         try{
189             const user = await db.UserModel.findOne({name: sender})
190             if(!user) throw('User not found')
191             const message = await db.OneMessageBoxModel.findOne({_id: _id, sender: user});
192             if(!message) throw('Message not found')
193 
194             await db.OneMessageBoxModel.deleteOne({_id: _id, sender: user});
195             pubsub.publish('message'+`${sender}`, {
196                 onemessagebox: {
197                     mutation: 'DELETED',
198                     data: message,
199                 },
200             });
```

```
            console.log("===Mutation: deleteOneMessage===");
            console.log(message);
            return message;
        } catch(e) {console.log(e)}
    },

    async createVote (parent, {data}, context, info) {
        try{
            const {vote, creator} = data;
            const User = await db.UserModel.findOne({name: creator});
            if(!User) throw ("User not found")
            const Vote_exist = await db.VoteModel.findOne({vote: vote, creator: User});
            if(Vote_exist) throw ("This story has been made; share some other stories!!!")
            const newVote = new db.VoteModel({
                vote: vote,
                creator: User,
                count: 0
            })
            await newVote.save();
            pubsub.publish('vote', {
                vote: {
                    mutation: 'CREATED',
                    data: newVote
                },
            });
            return newVote;
        }catch(e) {console.log(e)}
    },

    async updateVote (parent, {data}, context, info) {
        try{
            const {vote, creator} = data;
            const User = await db.UserModel.findOne({name: creator});
            if(!User) throw ("User not found")
            const Vote_exist = await db.VoteModel.findOne({vote: vote, creator: User});
            if(!Vote_exist) throw ("Story doesn't exist!!!")
            const c = Vote_exist.count + 1
            const newVote = {
                vote: vote,
                creator: User,
                count: c
            }
            await db.VoteModel.updateOne({vote: vote, creator: User},newVote,{upsert: true});
            pubsub.publish('vote', {
                vote: {
                    mutation: 'UPDATED',
                    data: newVote
                },
```

```
            });
            return newVote;
        }catch(e) {console.log(e)}
    },

    async deleteVote (parent, {_id, creator}, context, info) {
        try{
            const User = await db.UserModel.findOne({name: creator});
            if(!User) throw ("User not found")

            const vote = await db.VoteModel.findOne({_id: _id, creator: User});
            if(!vote) throw ("Vote not found");

            await db.VoteModel.deleteOne({_id: _id, creator: User});
            pubsub.publish('vote', {
                vote: {
                    mutation: 'DELETED',
                    data:vote
                },
            });
            return vote;
        }catch(e) {console.log(e)}
    },
};

export default Mutation;
```

10.4 Subscription.js

```
import pubsub from './pubsub';

const Subscription = {
    user: {
      resolve: (payload) => {return payload.user},
      subscribe: () => {return pubsub.asyncIterator('user')}
    },
    due: {
      resolve: (payload) => {return payload.due},
      subscribe: (parent, {author}, context, info) => {
        return  pubsub.asyncIterator('due'+`${author}`);
        }
    },
    post: {
      resolve: (payload) => {return payload.post;},
      subscribe: () => { return pubsub.asyncIterator('post'); }
    },
    onemessagebox: {
      resolve: (payload) => {return payload.onemessagebox},
```

```
20        subscribe: (parent, {sender}, context, info) => {
21          return  pubsub.asyncIterator('message'+`${sender}`);
22          }
23      },
24      vote: {
25        resolve: (payload) => {return payload.vote},
26        subscribe: () => {return pubsub.asyncIterator('vote')}
27    },
28  };
29  export default Subscription;
```