2022

CS348 Project Milestone 2 Report

ANDREW ZANG, ZHONGQI YUE

Special Note:

1. The black colored text is from milestone 1. All the text colored in blue is from milestone 2.
2. In the feature part (R6-R11) of this report, part d is to display testing each feature from users' perspective. For backend SQL queries and expected output, please see part b and part c of each feature.
3. Message and note are used interchangeably here because we decided to add more functionality to enable real-time messages in the application.

Table of Contents

R1 2

R2 3

R3 4

R4 5

R5 6

R6 8

R7 14

R8 18

R9 23

R10 26

R11 29

R17 31

Link to GitHub repository: 31

R1

This application is a mock support desk for digital devices. It can be considered as a support website for a repairing company that deals with users' digital devices. Users can register with their email, name and other personal information to get support for their damaged devices like cell phones, laptops and tablets. The company and platform support many different kinds of devices like Apple, Samsung or Huawei products. To create a ticket, a user can simply log in and use the create ticket function to create a ticket and put in their device information and detailed messages about how they would like to receive support. After that, there will be staff replying to their ticket and giving support as necessary.

We will first start with some toy dataset with some mock users, products and tickets. In the beginning, the developers (Zhongqi Yue and Andrew Zang) of this application will be the admin staff of the website. Anyone with the deployed link should be able to create their own account and become a user of the website. After registering, users can create tickers, leave notes and messages for the staff. Users will be able to see all history of their tickets and messages, as well as the status of all tickets. Staff will be able to log in to staff portal and reply to users as necessary. After a problem is resolved, ticket can be closed by the user or the staff.

R2

In this application, we utilized React (TypeScript) framework for frontend and Node/Express (TypeScript) for backend. We used relational database MySQL ~~on our local machines~~ on cloud GCP SQL. We changed to GCP SQL because it is more convenient for us to collaborate during development. Since the data is in cloud environment, we do not need to reinsert same data to everyone's laptop repetitively to be consistent. We can all use the same data on cloud, and this saves us lots of time during development, as well as helps reducing error and unnecessary bugs during development phase.

Compared to Milestone 1, we also utilized docker to containerize backend components. This is done through docker images and containers. I will first explain what is happening behind the scene. Containers will be created by using a docker image. The container will run the docker image and backend component will be running in the container. Note that the backend components will automatically connect to GCP SQL during this process. This is very helpful during development because developers do not need to worry about lots of infrastructure and environment setup, which saves a lot of time. Now with this set up, developers can work on the frontend directly in the docker container, with everything ready for them.

The development OS is MacOS ~~Monterey~~ Ventura due to a very recent Apple OS update.

R3

We would first create a sample dataset manually. We will create 4 CSV files for our mock users, products, tickets and notes. Then we use Node.js to read in the four CSV files into the program, connect to MySQL database and populate the database automatically using Sequelize ORM with Node in TypeScript.

We have migrated database to GCP SQL to achieve data persistent on cloud. The database has two instances, one for sample dataset and one for production dataset. We used the same way to populate the sample dataset.

R4

As mentioned in previous section, the database is in a database instance we created in GCP SQL. The production data has the same schema as the sample dataset, but with much more records.

Since our dataset heavily depends on user input, it is hard for us to obtain such a big dataset by actual users' inputs. Thus, we wrote a python script to randomly generate such a large dataset. The script directly inserts those records into the production database. More details of how the script can be found in *productionDataGenerator.py* in backend folder of our repository.

Now in production dataset, we have about 20,000 users, 36 product records, 10,000 tickets and 1,000 notes. In the product table, we only have 36 unique combinations of input, so that's why we have only 36 records in this table.

R5

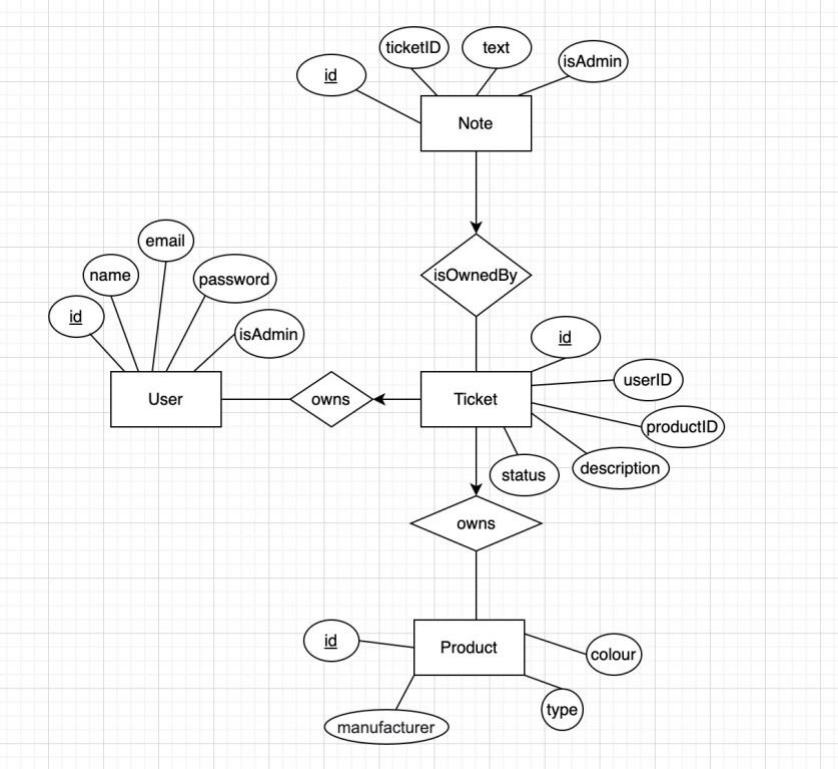
We have 4 tables in the database, User, Product, Ticket and Note respectively.

R5a

In User table, we assume user email is unique and password is hashed using external libraries.
Foreign keys:

- Ticket(userID) references User
- Ticket(productID) references Product
- Note(ticketID) references Ticket
- User(email) is unique

R5b



From the above E/R diagram, we see that we have 4 entity lists in total. They are User, Product, Note and Ticket. In each entity, id is the primary key.
In User entity, we have user id, user name, user email, user password (for authentication) and if user is admin or not. In Ticket entity, we see that each ticket has its own id, the userID meaning which user it belongs to, the productID meaning which product this ticket is for, a description of the problem and status of the ticket. Here the status could be one of open, closed or archived.
We see that the user entity forms a many to one relation with ticket entity. This means that each user can own many tickets, but one ticket can only belong to one user.
In note entity, we see that we have an id for each note, a ticket id meaning which ticket this note belongs to, the text in the note and if this note is from admin.

Then, we see that we have a many-to-one relation between note and ticket. A note can only be related to one ticket and a ticket can have many notes associated with it.
Finally, we see that product also has a many-to-one relation with ticket entity. Each ticket can only be used to handle one product, but a product can appear in many different tickets. For example, we can have two tickets that are both dealing with same product like Apple iPhone.

R5c

Direct translation will result in 3 more tables, compared to the answer below. They are the 3 tables for the 3 relationship sets in the E/R diagram. However, the foreign keys section defined R5a above will merge these 3 extra tables to the 4 tables below. So, in the end, we are left with the table schema below.

1. User (id, name, email, password, isAdmin)
2. Product (id, manufacturer, type, colour)
3. Ticket (id, userID, productID, description, status)
4. Note (id, ticketID, text, isStaff)

Foreign keys are discussed above in R5a.

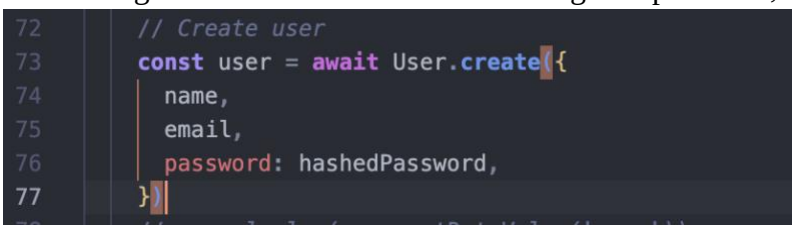
R6

R6a

When a new user visits our support website, the user can choose to register before submitting any tickets. A user would simply click on the Register button, and we have an event handler for that button. The event handler will call the backend API to capture all the information the user inputs and send them back. The backend API will then check/validate user input and insert a new user into the User table in our database.

R6b

ORM for inserting a user to the User table according to input name, email and password.

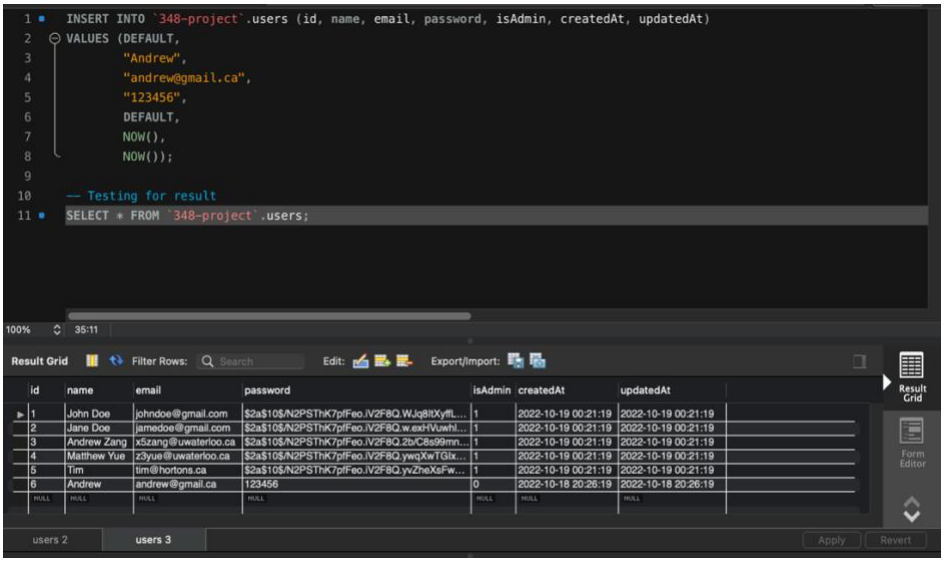
- 

```
72    // Create user
73    const user = await User.create({
74      name,
75      email,
76      password: hashedPassword,
77    })
```

SQL Query for inserting an Example user:

```
INSERT INTO `348-project`.users (id, name, email, password, isAdmin, createdAt,
updatedAt)
VALUES (DEFAULT,
        "Andrew",
        "andrew@gmail.ca",
        "123456",
        DEFAULT,
        NOW(),
        NOW());
```

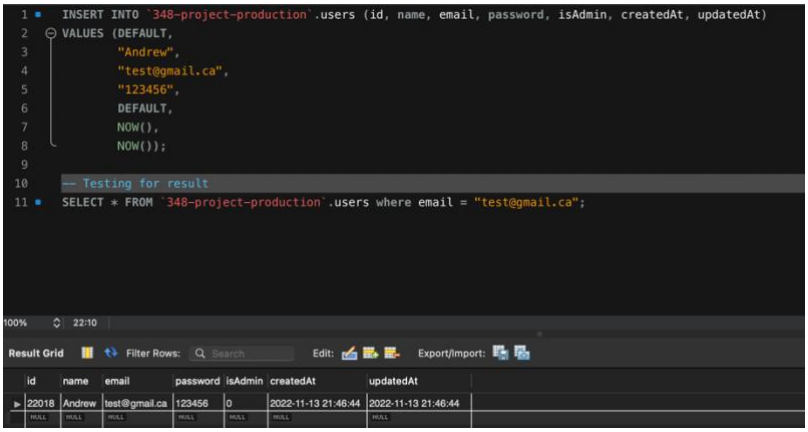
Expected output for inserting this data tuple (“Andrew”, “andrew@gmail.ca”, “123456”, “isAdmin”) on sample database.


INSERT INTO `348-project`.users (id, name, email, password, isAdmin, createdAt, updatedAt)
VALUES (DEFAULT,
"Andrew",
"andrew@gmail.ca",
"123456",
DEFAULT,
NOW(),
NOW());
-- Testing for result
SELECT * FROM `348-project`.users;
100%
35:11
Result Grid
Filter Rows:
Search
Edit:
Export/Import:
id
name
email
password
isAdmin
createdAt
updatedAt
1 John Doe johndoe@gmail.com $2a$10$/N2PSThK7pfFeo.iV2F8Q.WJq8ltXyffL... 1 2022-10-19 00:21:19 2022-10-19 00:21:19
2 Jane Doe janedoe@gmail.com $2a$10$/N2PSThK7pfFeo.iV2F8Q.w.exHVuwhl... 1 2022-10-19 00:21:19 2022-10-19 00:21:19
3 Andrew Zang x5zang@uwaterloo.ca $2a$10$/N2PSThK7pfFeo.iV2F8Q.2bfC8s99mn... 1 2022-10-19 00:21:19 2022-10-19 00:21:19
4 Matthew Yue z3yue@uwaterloo.ca $2a$10$/N2PSThK7pfFeo.iV2F8Q.ywqXwTGlx... 1 2022-10-19 00:21:19 2022-10-19 00:21:19
5 Tim tim@hortons.ca $2a$10$/N2PSThK7pfFeo.iV2F8Q.yvZheXsFw... 1 2022-10-19 00:21:19 2022-10-19 00:21:19
6 Andrew andrew@gmail.ca 123456 0 2022-10-18 20:26:19 2022-10-18 20:26:19
NULL NULL NULL NULL NULL NULL NULL
Result Grid
Form Editor
users 2
users 3
Apply
Revert

R6c

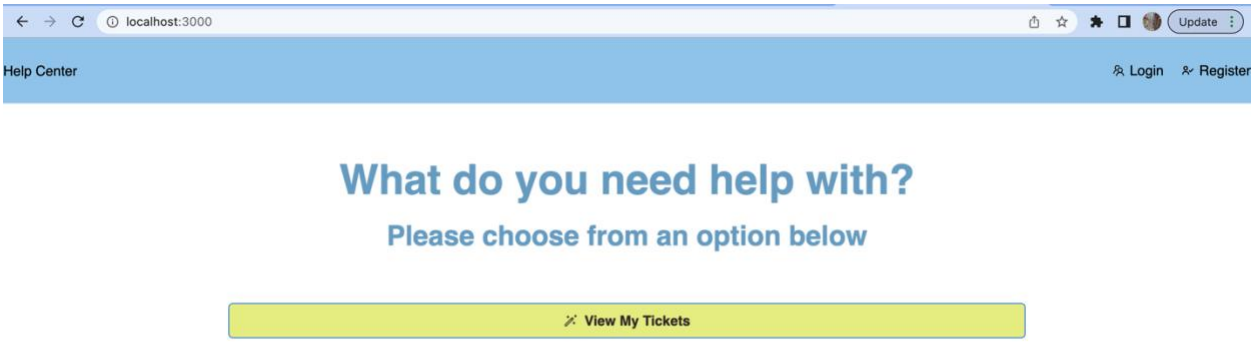
Insertion operation can be optimized by inserting multiple rows of records at one time. For our case, every time a user tries to register an account, they only register one account at a time. Therefore, we cannot use this optimization strategy. However, using the same query that used in 6b, we get a run time of 0.044 seconds. Performance is still fast in this case.

SQL query and expected output:



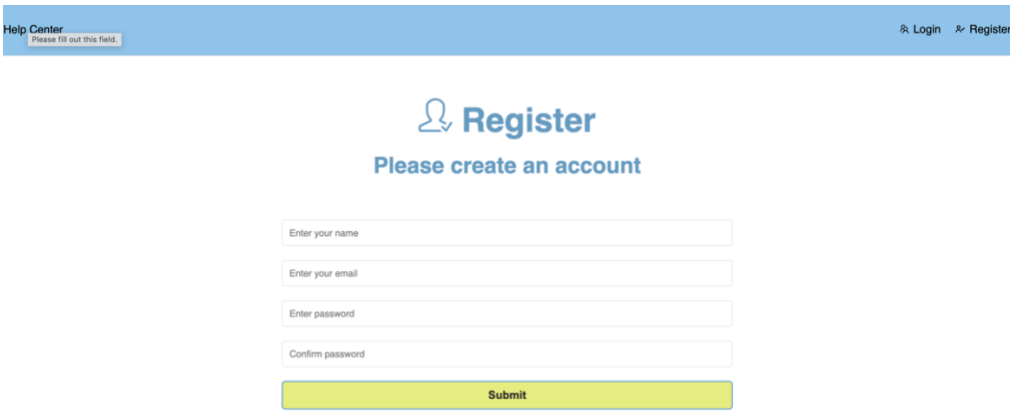
R6d

Below is the screen user will see when they visit our webpage



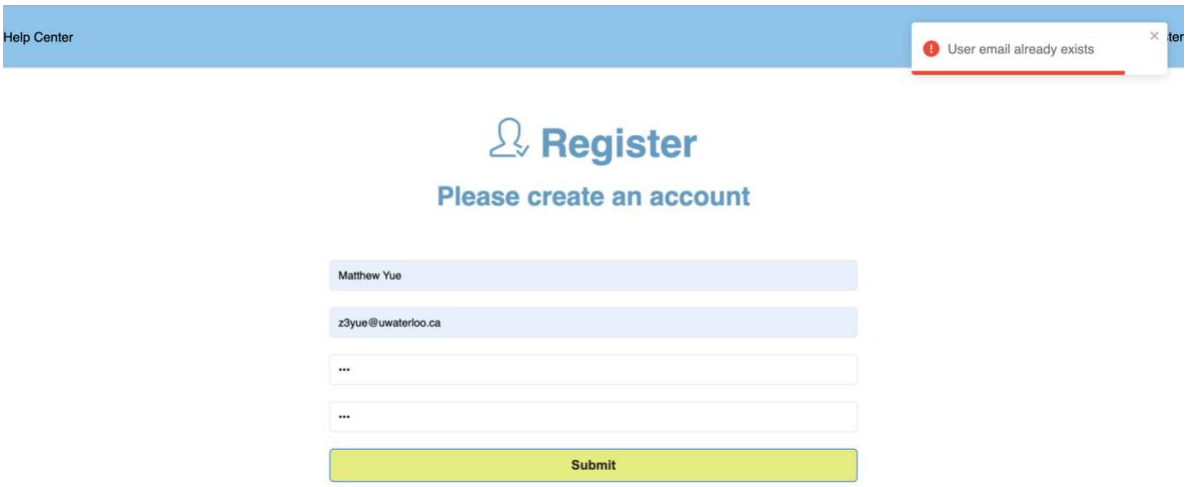
At this point, if user is not logged in, users will see the login and register option in the upper right corner. If user tries to click on “view my tickets” at this moment without logging in, users will be redirected to log in page.

To register, users need to click on “Register” button in the upper right part of the screen. After clicking, users will see below screen.



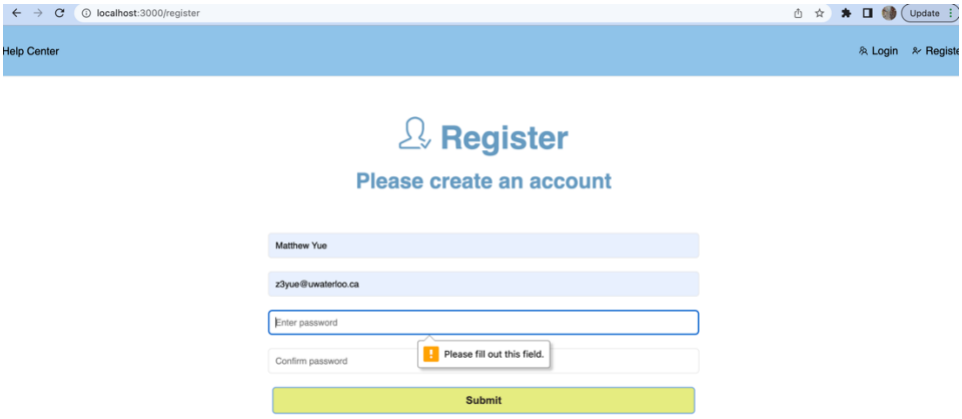
Users will need to input their name, email and two passwords that match with each other. If user fails to input any of this information, webpage will display corresponding error message. If user has already existed/registered in our database, corresponding error message will also show.

Error 1: User already existed (Using my own email and my email is already registered)

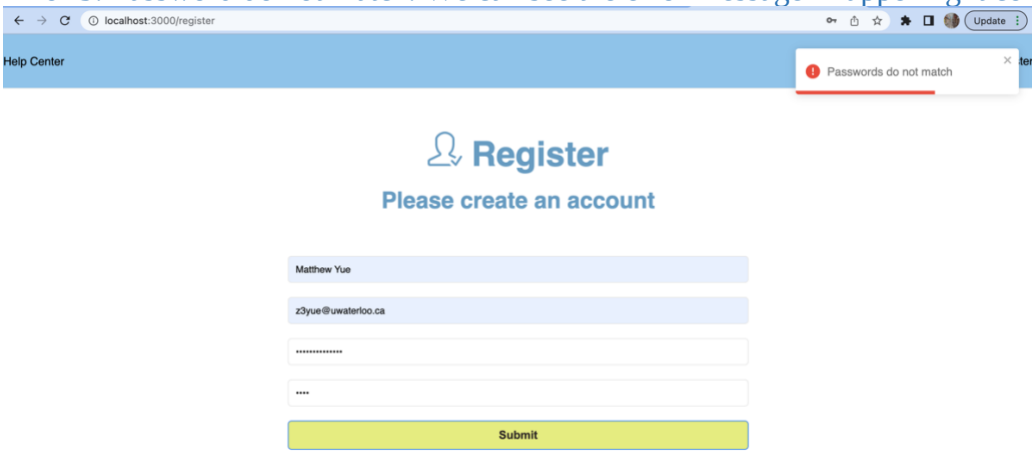


We see that registration is not successful and error message is displayed in the upper right corner.

Error 2: Leave some fields empty. If we click the submit button at this moment, it will not allow submission and the below message will display.

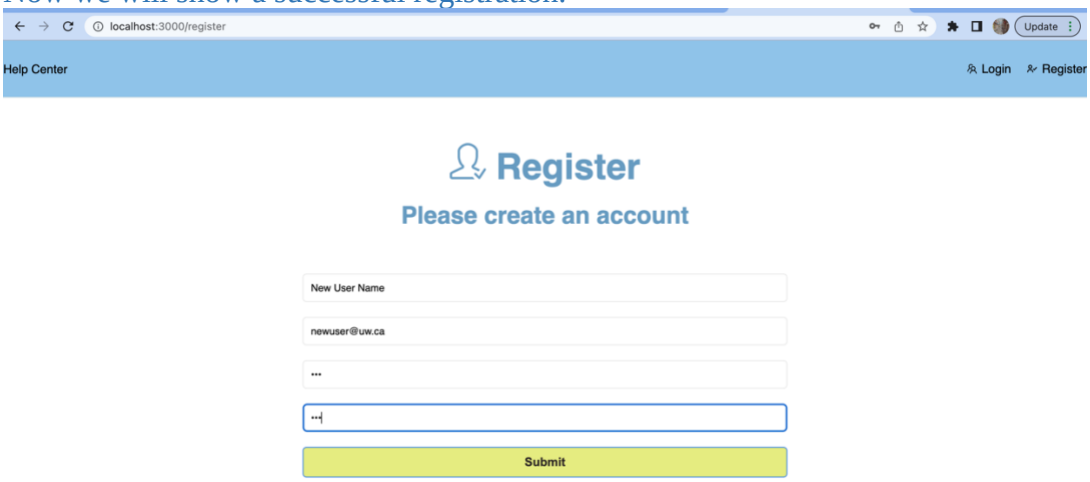


Error 3: Password do not match. We can see the error message in upper right corner.

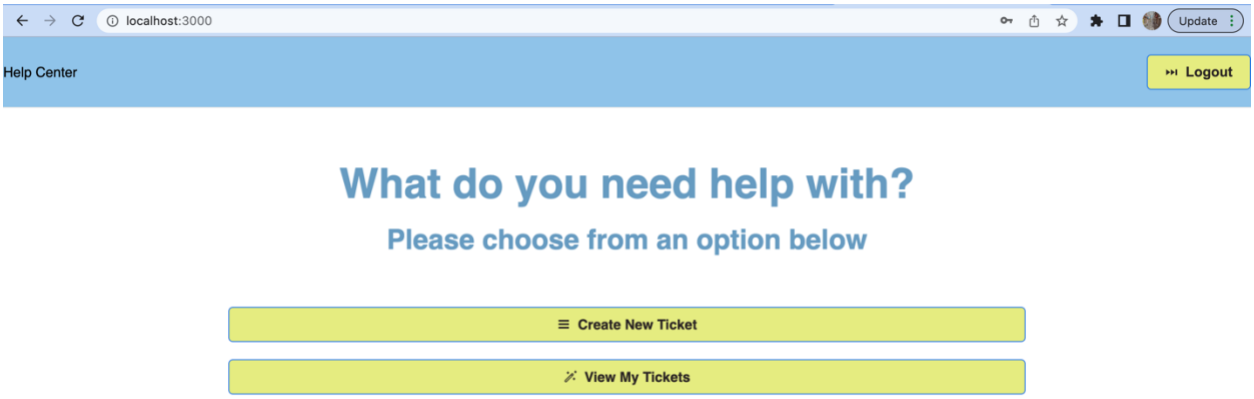


There are also other checkings done during registry. Please see /frontend/src/pages/register.tsx for more details.

Now we will show a successful registration.



After clicking on “submit” button, registration will be successful and users will be logged in and directed to the following page, which supports more functionalities for logged in user.



R7

R7a

When an existing user visits our page, the user will need to log in first before he/she can view his/her tickets and status of the ticket. So a user will need to input its user email and password. After clicking the log in button, the event handler for that button will call the backend API to look for that email in the database. If the email is not found, it will throw an error showing to the user. If it is found, we will compare the password that user inputs with the password stored in database. If it matches, we will authenticate the user.

R7b

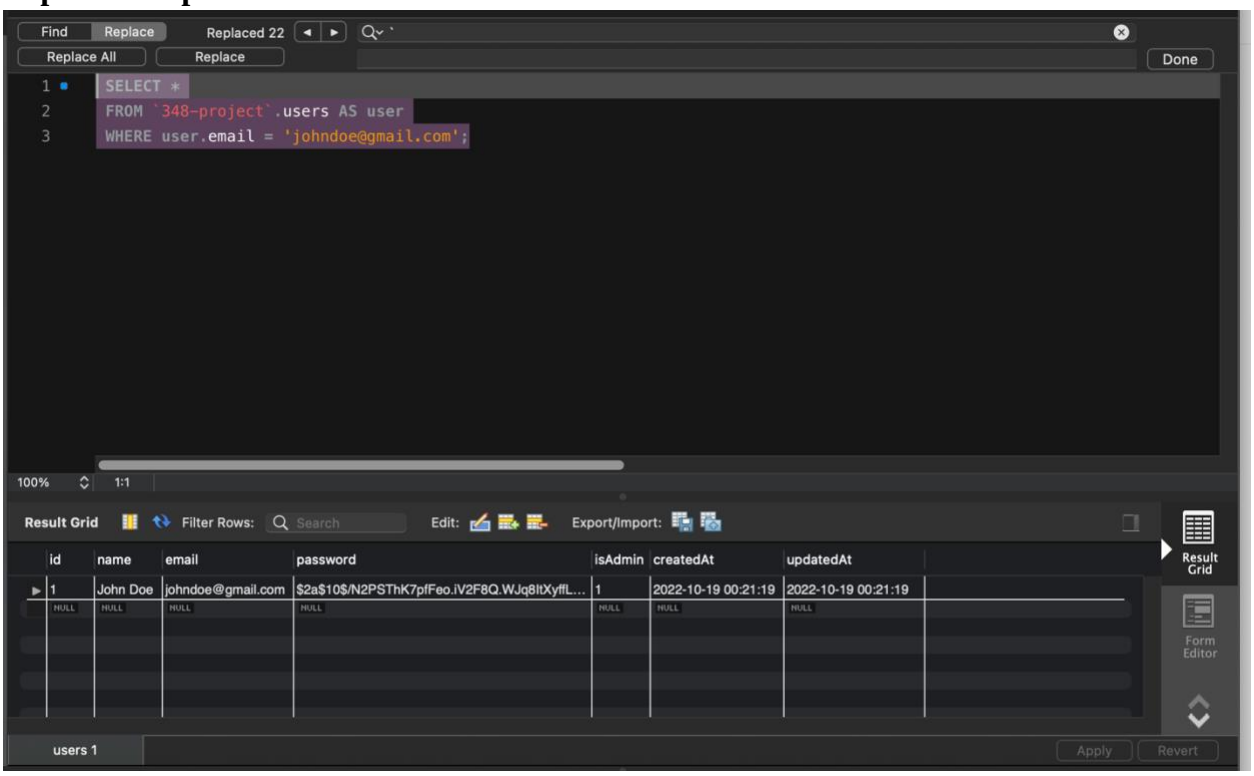
ORM for checking if a user exists in the User table according to email.

-
```
const user = await User.findOne({
  where: {
    email: email,
  },
})
```

SQLQuery for checking if “johndoe@gmail.com” exists in database:

```
SELECT *
 FROM `348-project`.users AS user
 WHERE user.email = 'johndoe@gmail.com';
```

Expected output:



R7C

Within the production data, we have more than 20 thousand user records. The following query runs in 0.031 seconds.

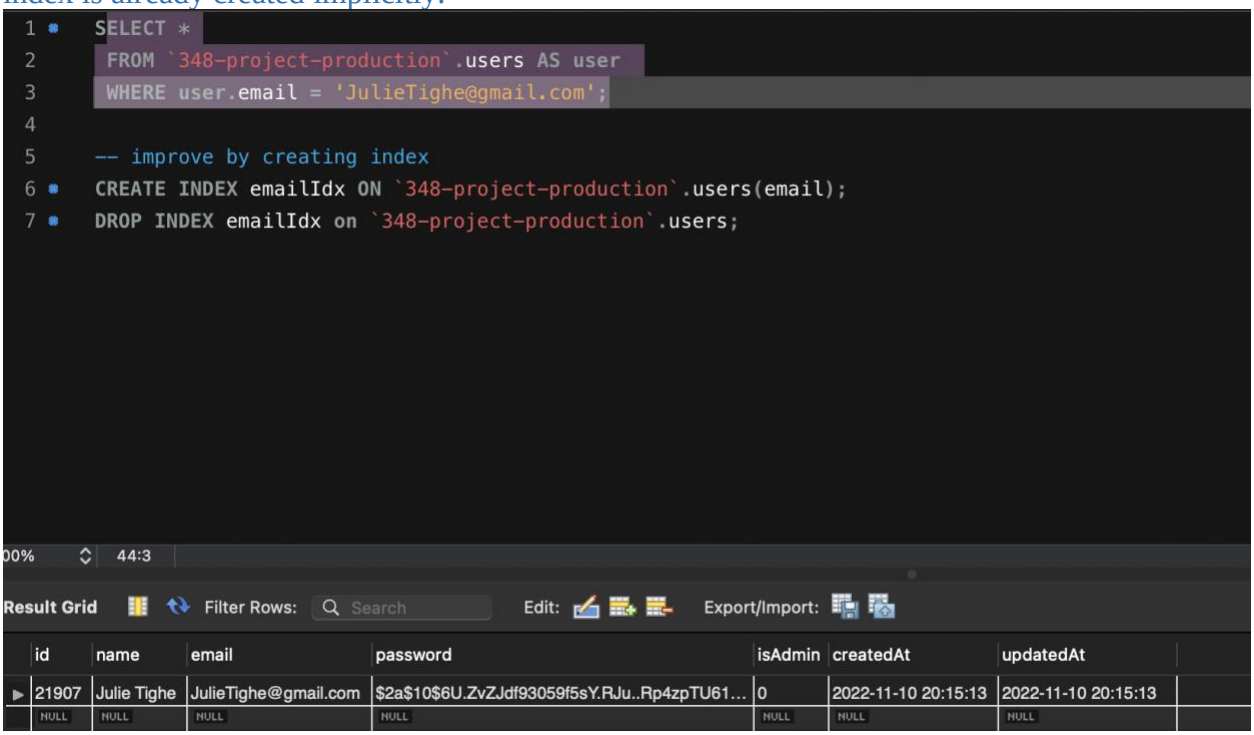
```
SELECT *
 FROM `348-project-production`.users AS user
 WHERE user.email = 'JulieTighe@gmail.com';
```

To improve the performance, we created an index on user email column, using the following query.

```
CREATE INDEX emailIdx ON `348-project-production`.users(email);
```

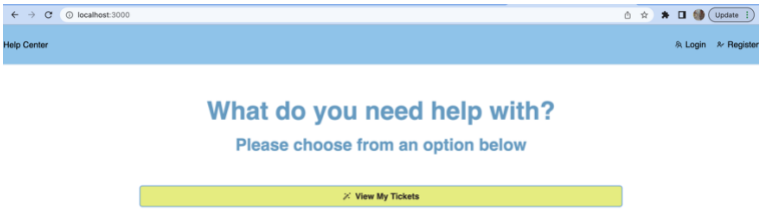
Then we ran the same query again, which has a runtime of 0.030 seconds, which has a small improvement of 0.001 seconds.

The minor change in performance is due to the emails are unique within user table. Thus, an index is already created implicitly.

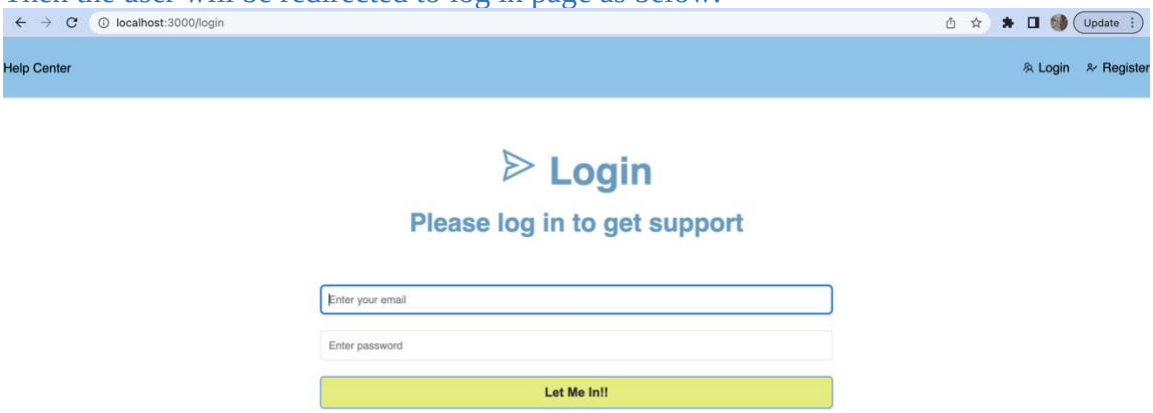


R7d

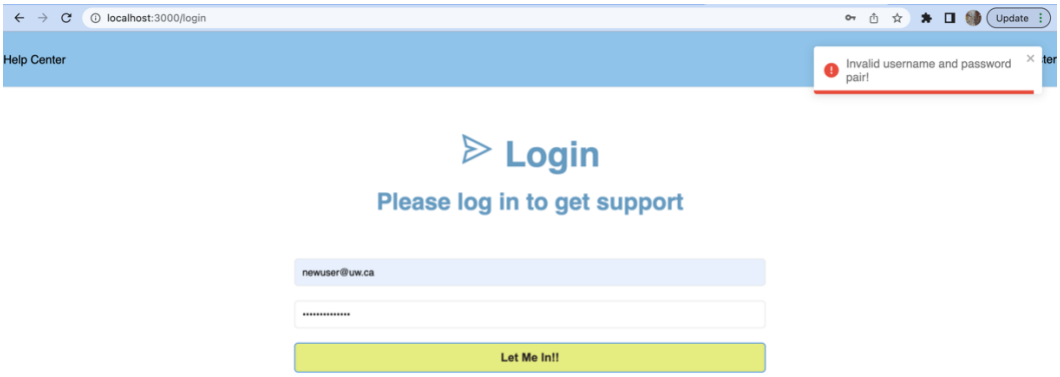
To log in, users need to click on the log in button in the upper right corner.



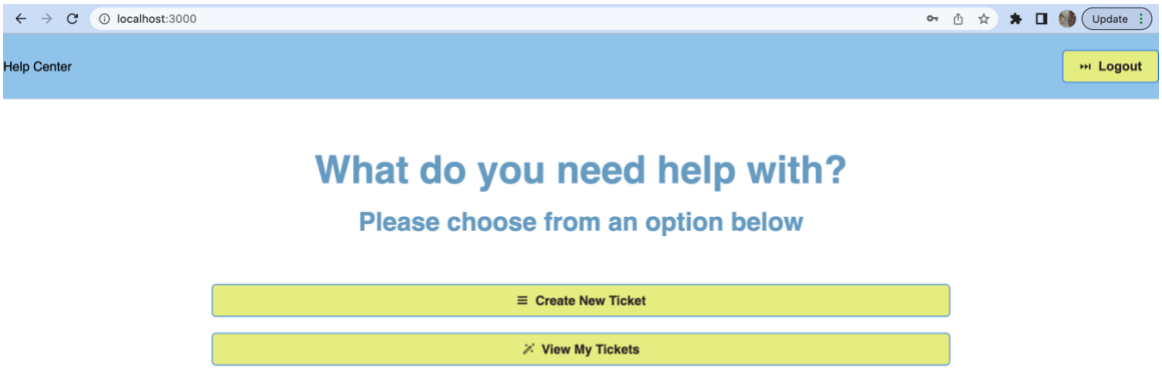
Then the user will be redirected to log in page as below.



Let's log in the user we just created above in R6d!
If the username and password do not match, we will not allow logging in and error message will display. We see the error message in top right corner.



If user gives the correct password, it will be successful, and users will be redirected to their homepage to view or create their tickets. We also notice that after logging in, the button in the upper right corner becomes log out, not log in and register anymore.


localhost:3000
Update
Help Center
Logout
What do you need help with?
Please choose from an option below
Create New Ticket
View My Tickets

R8

R8a

For logged in users, users have the option to update a ticket's information. For example, if a user's Macbook updated OS after submitting the initial ticket, the user might want our staff to know that. So, the user might want to edit the existing tickets. Users can do this by clicking the Edit button in the ticket. The button event handler will call the backend API to update corresponding field in the database.

R8b

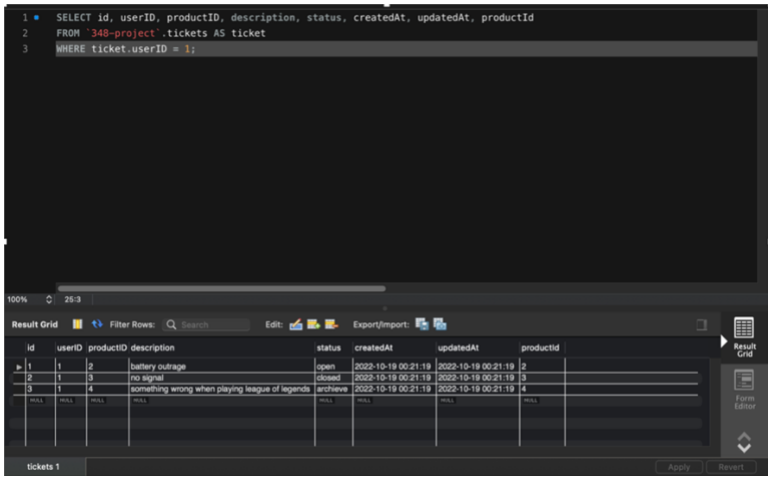
ORM for updating a ticket in Ticket table.

```
226     Ticket.findByPk(req.params.id)
227       .then((ticket) => {
228         if (!ticket) {
229           res.status(404)
230           throw new Error('Ticket not found')
231         }
232
233         if (ticket.getDataValue('userID') !== userID) {
234           res.status(401)
235           throw new Error('Not Authorized')
236         }
237         return ticket.update({
238           description: updatedDescription,
239           status: updatedStatus,
240         })
241       })
242       .then((result) => {
243         console.log('UPDATED TICKETS')
244         res.status(200).json(result)
245       })
246       .catch((err) => console.log(err))
247   }
248 )
```

SQL Query for update a ticket by its ID:

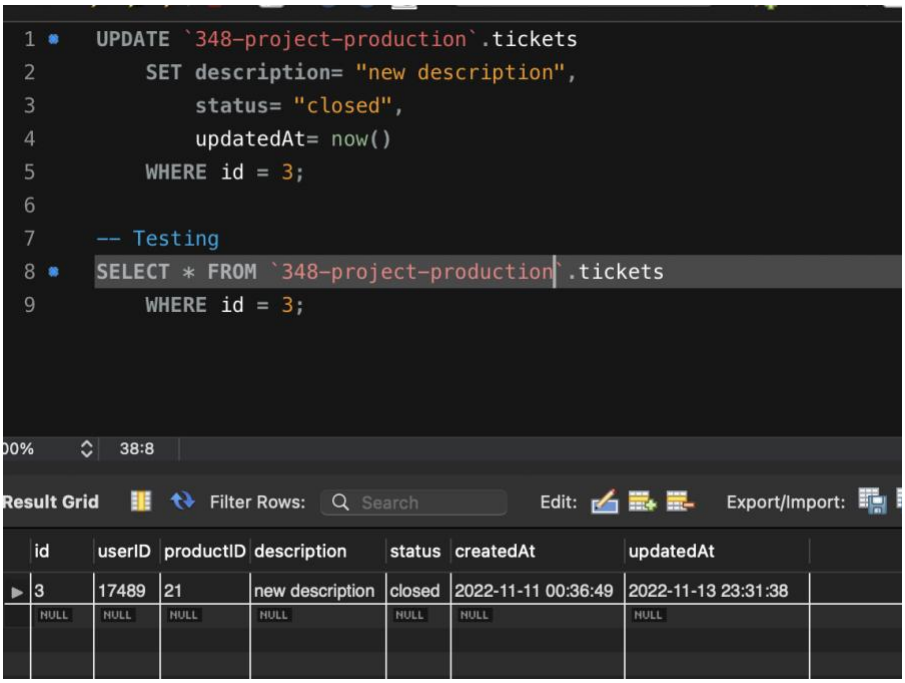
```
UPDATE `348-project`.tickets
    SET description= "new description",
        status= "closed",
    updatedAt= now()
    WHERE id = 3
```

Expected Output:


1 SELECT id, userID, productID, description, status, createdAt, updatedAt, productId
2 FROM `348-project`.tickets AS ticket
3 WHERE ticket.userID = 1;
100% 25:3
Result Grid
Filter Rows:
Search
Edit:
Export/Import:
id userID productID description status createdAt updatedAt productId
1 1 2 battery outrage open 2022-10-19 00:21:19 2022-10-19 00:21:19 2
2 1 3 no signal closed 2022-10-19 00:21:19 2022-10-19 00:21:19 3
3 1 4 something wrong when playing league of legends archieve 2022-10-19 00:21:19 2022-10-19 00:21:19 4
NULL NULL NULL NULL NULL NULL NULL NULL
Result Grid
Form Editor
tickets 1
Apply
Revert

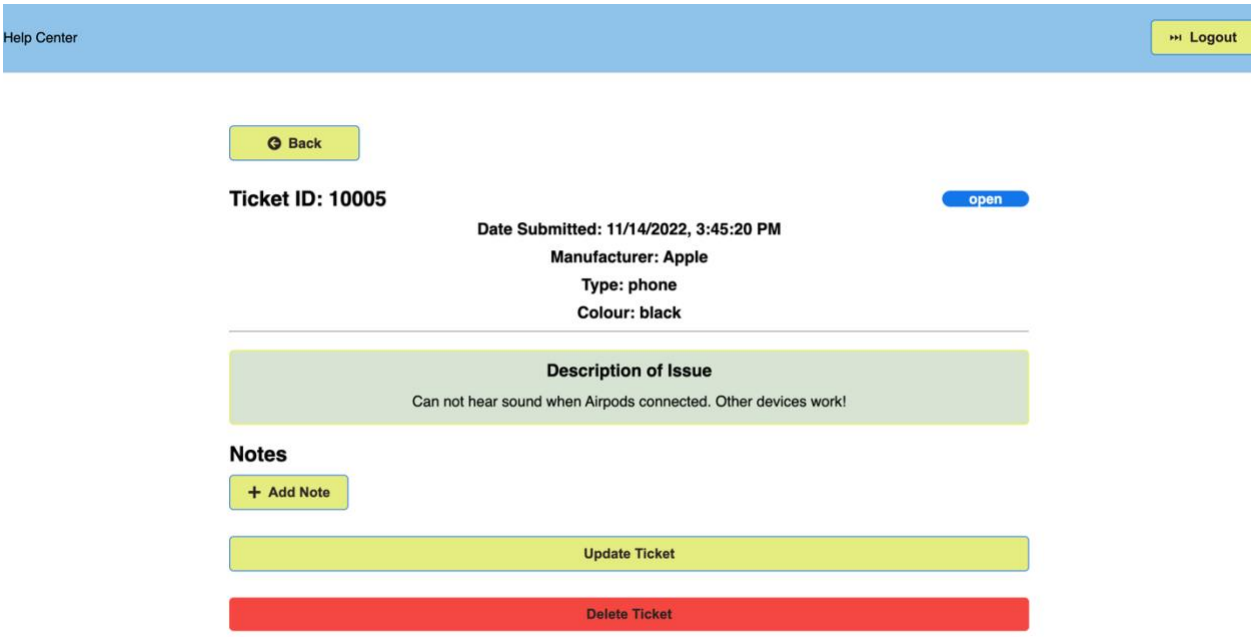
R8c

There are not much of optimization we can do on production database in this particular functionality. We tried to come up with several different logical plan and execution plan. Then, we first wrote raw SQL in the workbench to compare these plans. Sadly, there is not much difference. Hence, we finally wrote ORM and let the compiler to choose the best SQL queries. One possible solution would be removing index on the column to be updated. Since the column to be updated is neither a primary key or unique key, and we did not create any index on it, therefore, the query itself is already being optimized. We find the record by its id which is the primary key, and there is an implicit index on it once we created the database. It has a runtime of 0.031 seconds, which is considered as good performance.

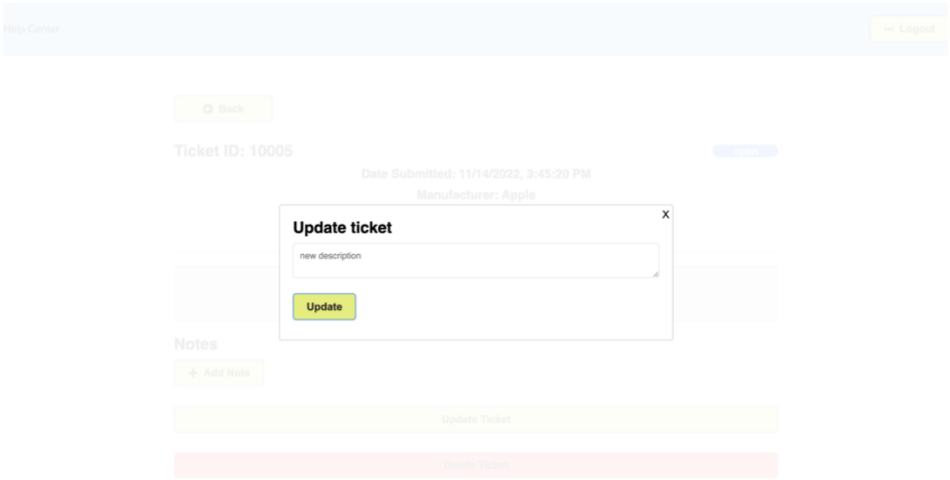


R8d

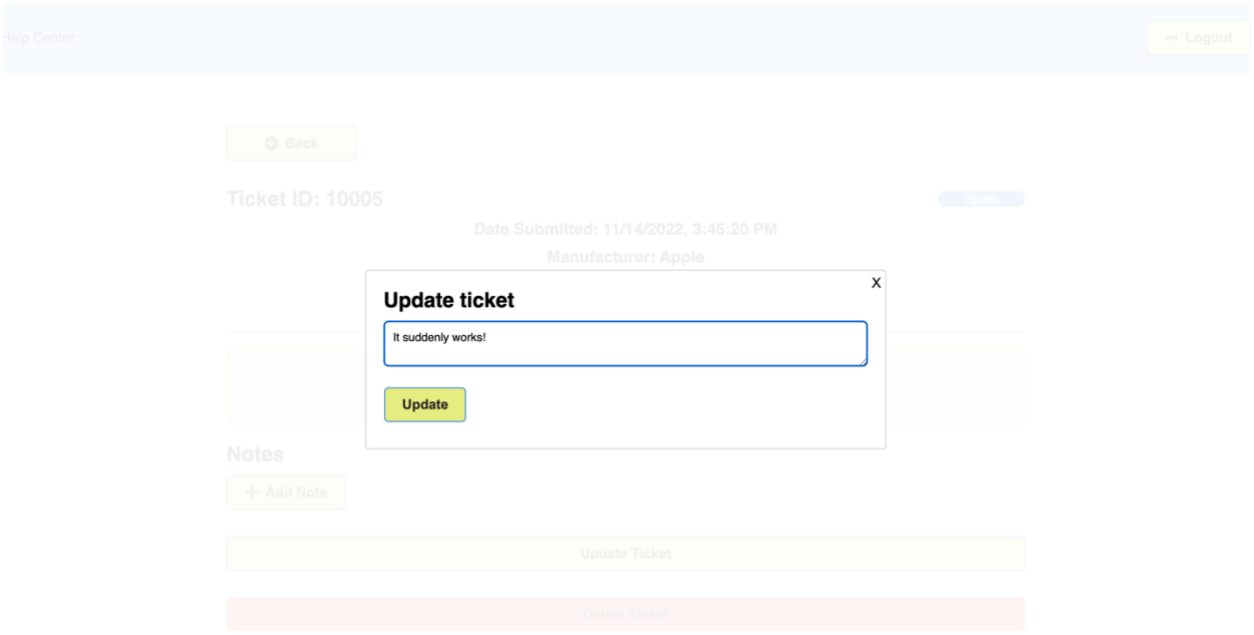
Users are able to see the below page for a particular ticket they submitted.



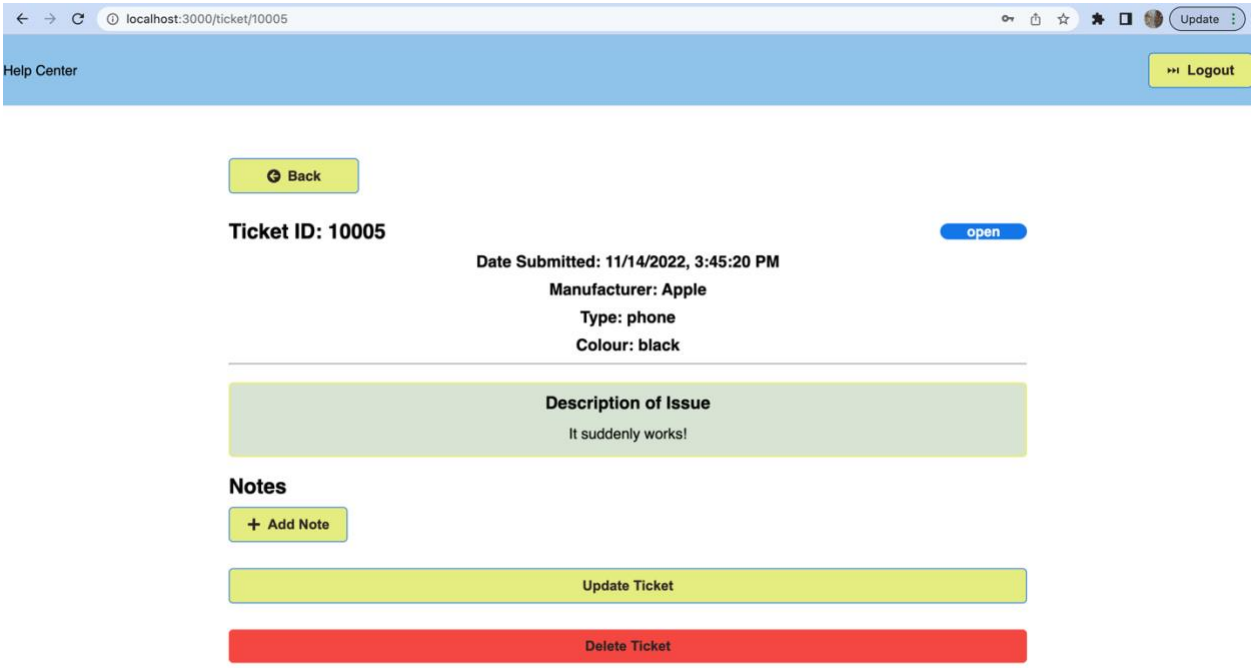
Now users have the option to update the ticket description if anything changes. Users can click on the update ticket button below, and a new window will open for user to input new description of the issue.



Now users can type in the new description and click update like below.



Now, click Update. We can see that the description is updated.



R9

R9a

The user will have the option to delete a ticket. We have a button near each ticket and the user can click on the delete button to delete a ticket. After the user clicks on the button, it will call the backend API to perform delete operation in database. After the deletion completes, we will give user a new page without the deleted ticket.

R9b

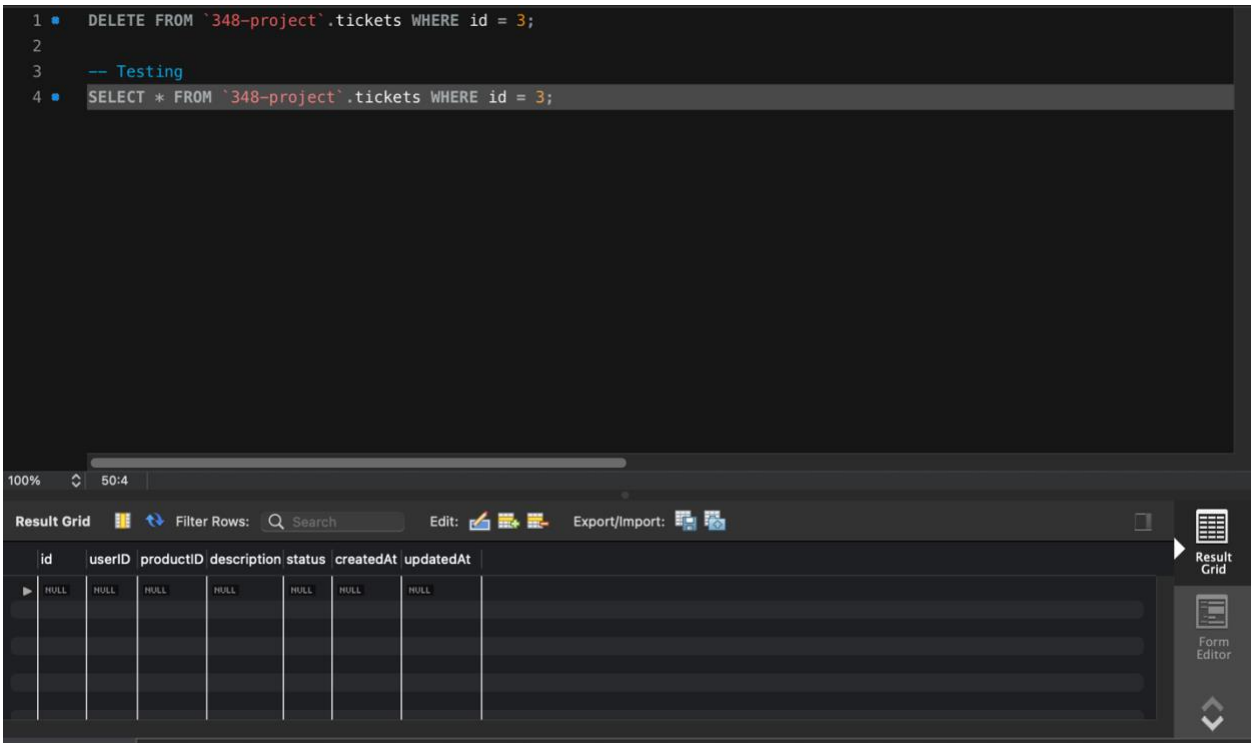
ORM for deleting a ticket in Ticket table.

```
182     const ticket = await Ticket.findByPk(req.params.id)
183
184     if (!ticket) {
185       res.status(404)
186       throw new Error('Ticket not found')
187     }
188
189     if (ticket.getDataValue('userID') !== userID) {
190       res.status(401)
191       throw new Error('Not Authorized')
192     }
193
194     await ticket.destroy()
```

SQL Query for Deleting ticket with ID of 3:

DELETE FROM `348-project`.tickets WHERE id = 3;

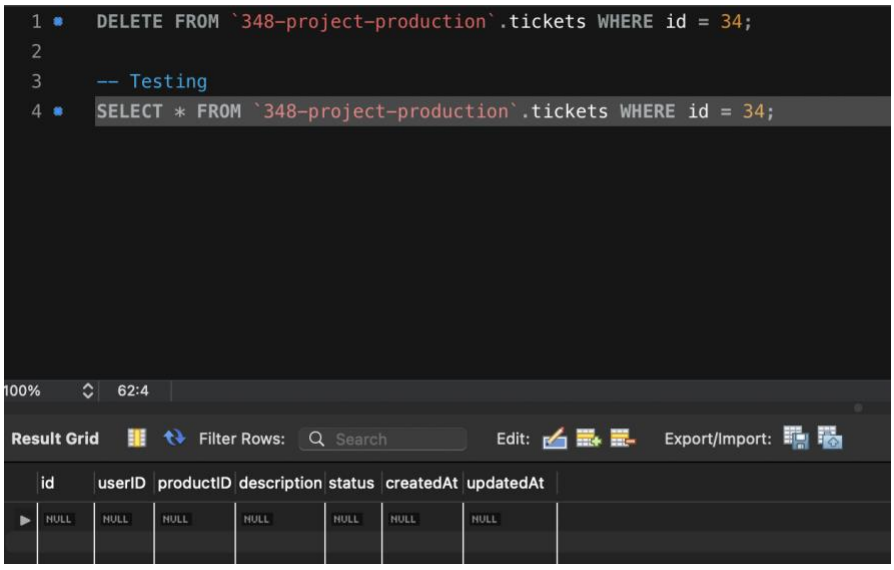
Expected Output:



R9c

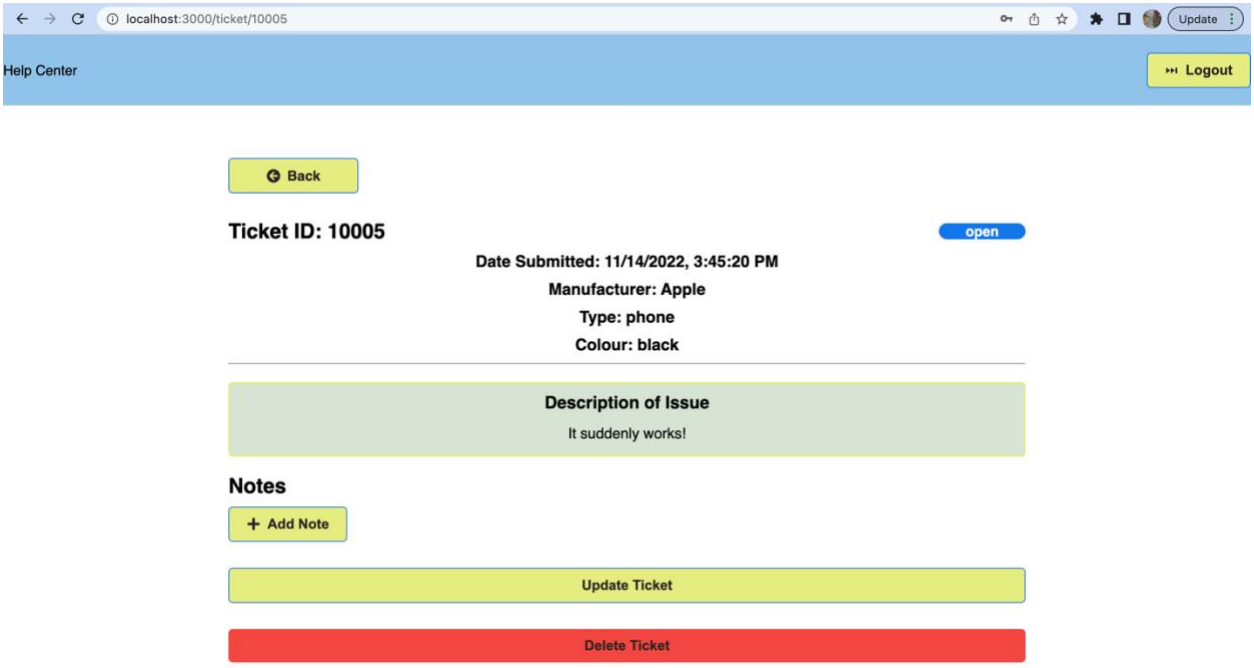
Same as before, we search the ticket for deletion using its id which is ticket's primary key. Thus, there are not much optimization we can do here as well. In production database, the same query has a runtime of 0.037 seconds, which is considered as fast.

The query and output are shown below:

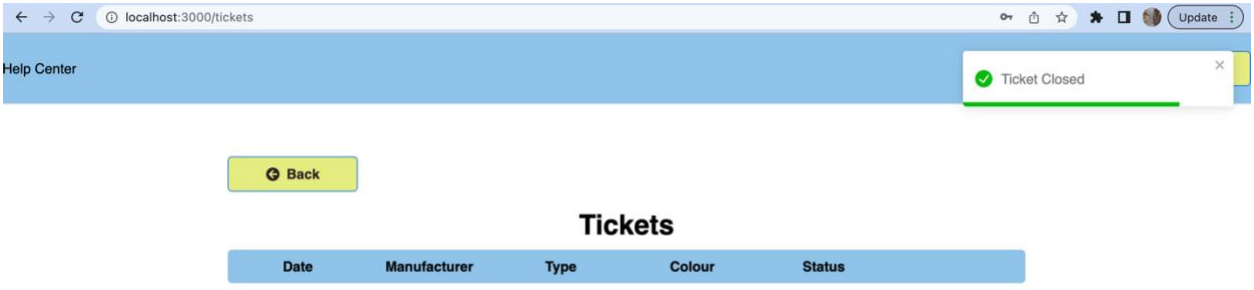


R9d

In the ticket page, there is a delete button in the bottom. If the user's issue is fixed, user can choose to delete the ticket by clicking the delete button below.



Now, clicking the delete button will delete this ticket. We see that ticket is closed and there is no ticket for this user.



R10

R10a

The user will have the option to get all tickets that are created by the user. We have a button in the home page and the user can click on the “view all tickets” button to get to such page. After the user clicks on the button, it will call the backend API to fetch all tickets that belongs to this user in database. After the API call completes, we will give user a new page with all open tickets created by the user.

R10b

ORM for fetching tickets in Ticket table.

```
42    const tickets = await Ticket.findAll({
43      where: {
44        status: 'open',
45        userID: userID,
46      },
47      include: [User, Product],
48    })
```

The ORM generated SQL is shown below:

```
SELECT `ticket`.`id`, `ticket`.`userID`, `ticket`.`productID`, `ticket`.`description`,
`ticket`.`status`, `ticket`.`createdAt`, `ticket`.`updatedAt`, `user`.`id` AS `user.id`, `user`.`name`
AS `user.name`,
`user`.`email` AS `user.email`, `user`.`password` AS `user.password`, `user`.`isAdmin` AS
`user.isAdmin`,
`user`.`createdAt` AS `user.createdAt`, `user`.`updatedAt` AS `user.updatedAt`,
`product`.`id` AS `product.id`, `product`.`manufacturer` AS `product.manufacturer`,
`product`.`type` AS `product.type`, `product`.`colour` AS `product.colour`,
`product`.`createdAt` AS `product.createdAt`, `product`.`updatedAt` AS `product.updatedAt`
FROM `348-project-sample`.`tickets` AS `ticket`
LEFT OUTER JOIN `348-project-sample`.`users` AS `user` ON `ticket`.`userID` = `user`.`id`
LEFT OUTER JOIN `348-project-sample`.`products` AS `product` ON `ticket`.`productID` =
`product`.`id`
WHERE `ticket`.`status` = 'open' AND `ticket`.`userID` = 1;
```

Expected Output:

id	userID	productID	description	status	createdAt	updatedAt	user.id	user.name	user.email	user.password
1	1	2	battery outrage	open	2022-11-10 15:18:36	2022-11-10 15:18:36	1	John Doe	johndoe@gmail.com	$2a$10$vUgaEP/dL7pVfM0vVG1GDOYL6MAR...

user.isAdmin	user.createdAt	user.updatedAt	product....	product.manufactu...	product.ty...	product.col...	product.createdAt	product.updatedAt
1	2022-11-10 15:18:36	2022-11-10 15:18:36	2	apple	phone	silver	2022-11-10 15:18:36	2022-11-10 15:18:36

R10c

Due to the conditions we choose to fetch the list of tickets, we can add an index on ticket's status and userID. Before creating the index, the query takes 0.035 seconds to finish. After creating the index, the query takes in 0.030 seconds which is some good performance improvement.

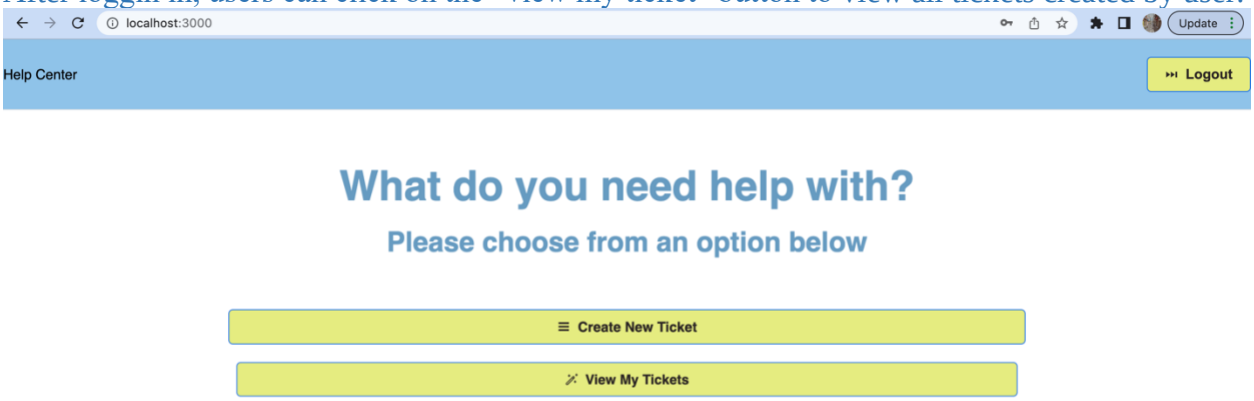
The query for creating such an index is:
CREATE INDEX statusUserID on `348-project-production`.tickets(status, userID);
Expected output for Production database:

id	userID	productID	description	status	createdAt	updatedAt	user.id	user.name	user.email	user.password
10001	4	31	2222222	open	2022-11-13 20:17:37	2022-11-14 00:55:28	4	Matthew Yue	z3yue@uwaterloo.ca	$2a$10$6U.ZvZJdf93059f5sY.RJuo2Rlo9xi0bt5...
10002	4	19	f	open	2022-11-13 20:23:39	2022-11-14 00:27:15	4	Matthew Yue	z3yue@uwaterloo.ca	$2a$10$6U.ZvZJdf93059f5sY.RJuo2Rlo9xi0bt5...
10004	4	1	vthjk	open	2022-11-14 00:56:35	2022-11-14 00:56:35	4	Matthew Yue	z3yue@uwaterloo.ca	$2a$10$6U.ZvZJdf93059f5sY.RJuo2Rlo9xi0bt5...

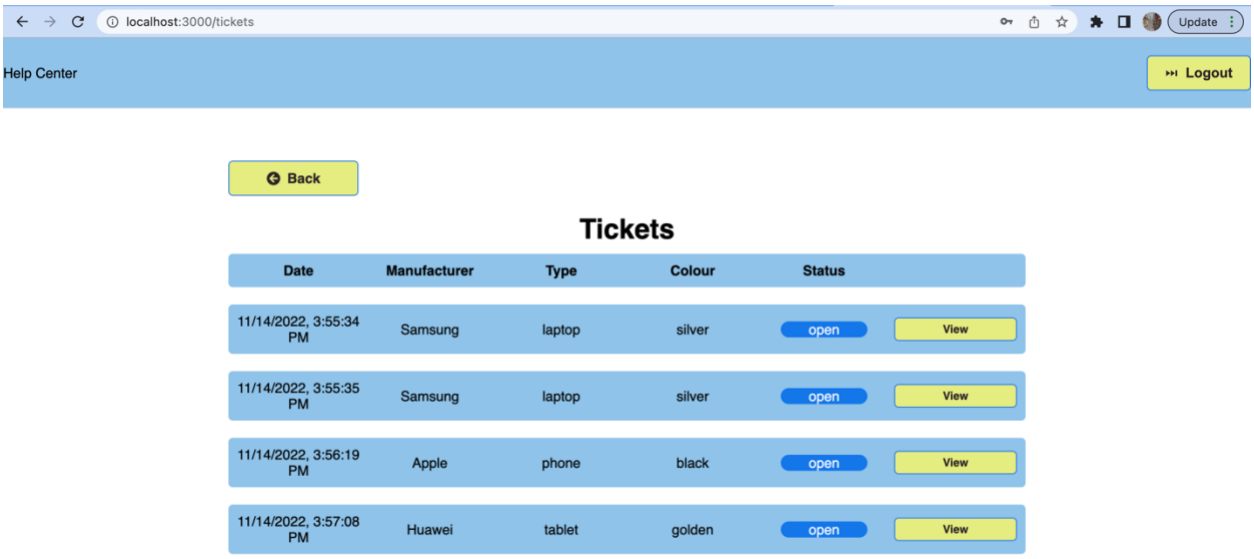
user.isAdmin	user.createdAt	user.updatedAt	product....	product.manufactu...	product.ty...	product.col...	product.createdAt	product.updatedAt
0	2022-11-10 16:47:49	2022-11-10 16:47:49	31	Huawei	phone	golden	2022-11-10 20:17:34	2022-11-10 20:17:34
0	2022-11-10 16:47:49	2022-11-10 16:47:49	19	Samsung	phone	golden	2022-11-10 20:17:33	2022-11-10 20:17:33
0	2022-11-10 16:47:49	2022-11-10 16:47:49	1	Apple	laptop	silver	2022-11-10 20:17:32	2022-11-10 20:17:32

R10d

After loggin in, users can click on the "view my ticket" button to view all tickets created by user.



After clicking the "view my tickets" button, we can see the below page. This user created 2 tickets for his/her Samsung laptop, 1 ticket for apple phone and 1 ticket for Huawei tablet.


localhost:3000/tickets
Update
Help Center
Logout
Back
Tickets
Date
Manufacturer
Type
Colour
Status
11/14/2022, 3:55:34 PM
Samsung
laptop
silver
open
View
11/14/2022, 3:55:35 PM
Samsung
laptop
silver
open
View
11/14/2022, 3:56:19 PM
Apple
phone
black
open
View
11/14/2022, 3:57:08 PM
Huawei
tablet
golden
open
View

R11

R11a

Users can click on the view button to see each individual ticket. Once the user navigate to a specific ticket page, the user will see details of the ticket and a list of notes associated with the ticket. As the user enters such a page, the frontend will call the backend API to fetch all notes associated with this ticket in database. We need the notes to be ordered ascendingly by its created time.

R11b

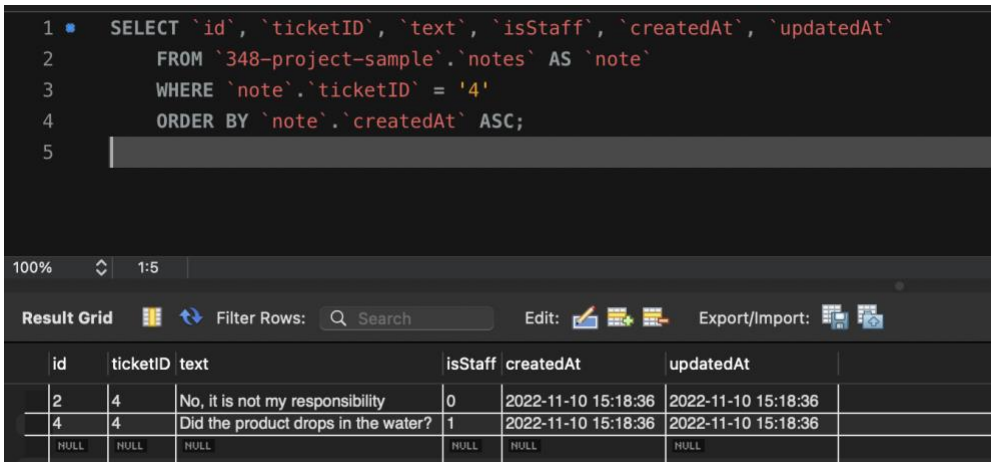
ORM for fetching notes in Notes table.

```
46      const notes = await Note.findAll({
47        where: {
48          ticketID: req.params.ticketId,
49        },
50        order: [['createdAt', 'ASC']],
51      })
```

The ORM generated SQL is shown below:

```
SELECT `id`, `ticketID`, `text`, `isStaff`, `createdAt`, `updatedAt`
    FROM `348-project-sample`.`notes` AS `note`
    WHERE `note`.`ticketID` = '4'
    ORDER BY `note`.`createdAt` ASC;
```

Expected Output:



R11c

```
1  SELECT `id`, `ticketID`, `text`, `isStaff`, `createdAt`, `updatedAt`
2      FROM `348-project-production`.`notes` AS `note`
3      WHERE `note`.`ticketID` = 10002
4      ORDER BY `note`.`createdAt` ASC;
5
```

100% 30:2

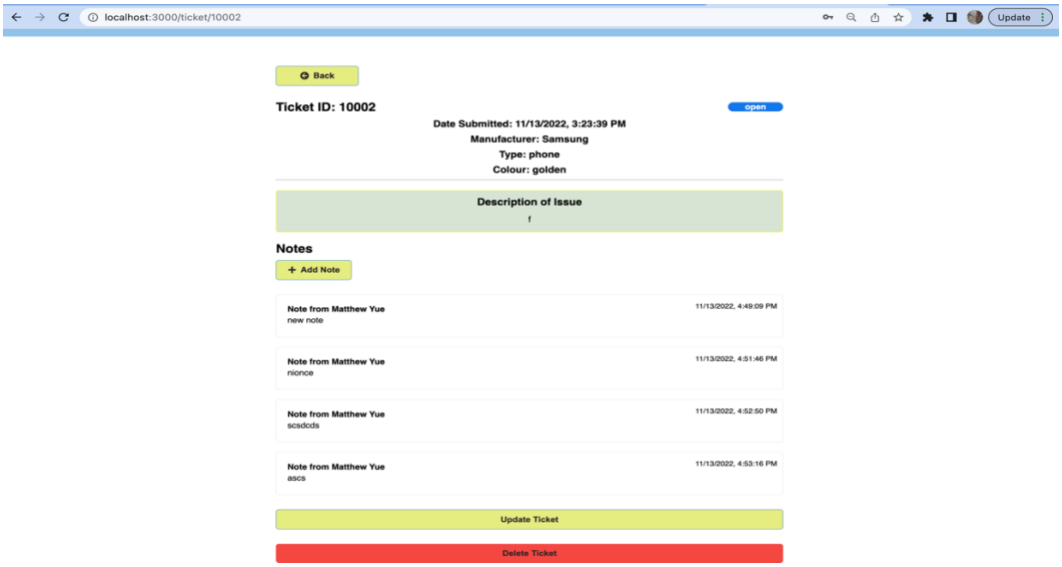
Result Grid | Filter Rows: Search | Edit: | Export/Import:

id	ticketID	text	isStaff	createdAt	updatedAt
1001	10002	new note	0	2022-11-13 21:49:09	2022-11-13 21:49:09
1002	10002	nionce	0	2022-11-13 21:51:46	2022-11-13 21:51:46
1003	10002	scsdcds	0	2022-11-13 21:52:50	2022-11-13 21:52:50
1004	10002	ascs	0	2022-11-13 21:53:16	2022-11-13 21:53:16
NULL	NULL	NULL	NULL	NULL	NULL

Testing in production dataset yields a runtime of 0.031 seconds, which is already fast enough. We can still further improve performance by adding index on ticketID.
Index is created as below:
CREATE INDEX ticketIDIdx on `348-project-production`.`notes`(ticketID);
However, this still yields the same runtime as the performance is not being affected by a lot. Creating such an index is not worth it.

R11d

After clicking on the view button, users can see details of the ticket and notes section in the bottom. Users can leave message to staff using this method.



R17

Andrew Zang:

1. Set up MySQL database and connect to it
2. prepared all code in backend folder
3. filled in R6b, R7b, R8b and R9b sections in this report.

M2:

1. main contributor to backend code
2. Set up cloud GCP SQL and containerize backend code
3. Generate production data
4. Finished backend part of this report

Zhongqi Yue:

1. prepared all other sections in this report
2. worked on frontend code for setup
3. Final formatting and error checking

M2:

1. Main contributor of frontend code
2. Finished all remaining part of this report
3. Error checking and format checking

Link to GitHub repository:

Link