**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. The development OS is MacOS Monterey.

**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.

**R5**

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

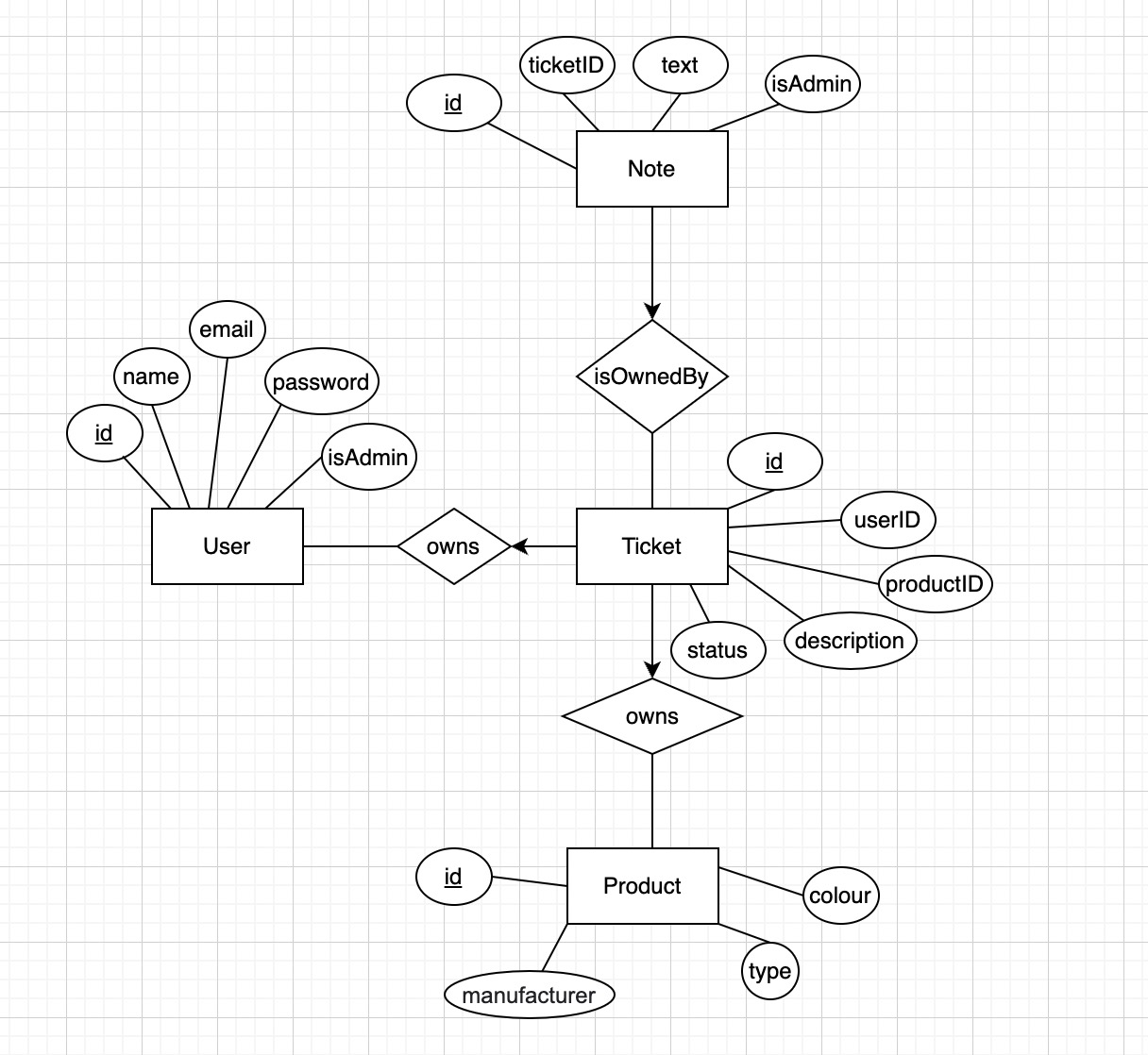
**R5a**

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

**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.

**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.

* Graphical user interface, text, application, website

  Description automatically generated

**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",

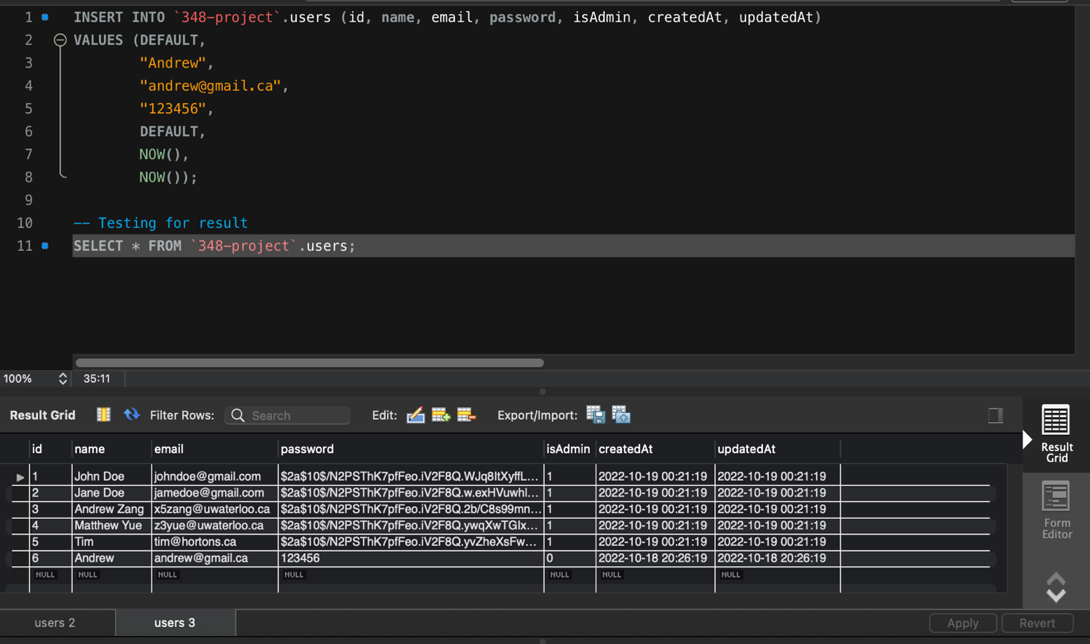
"123456",

DEFAULT,

NOW(),

NOW());

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



**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.

* Graphical user interface, text, application

  Description automatically generated

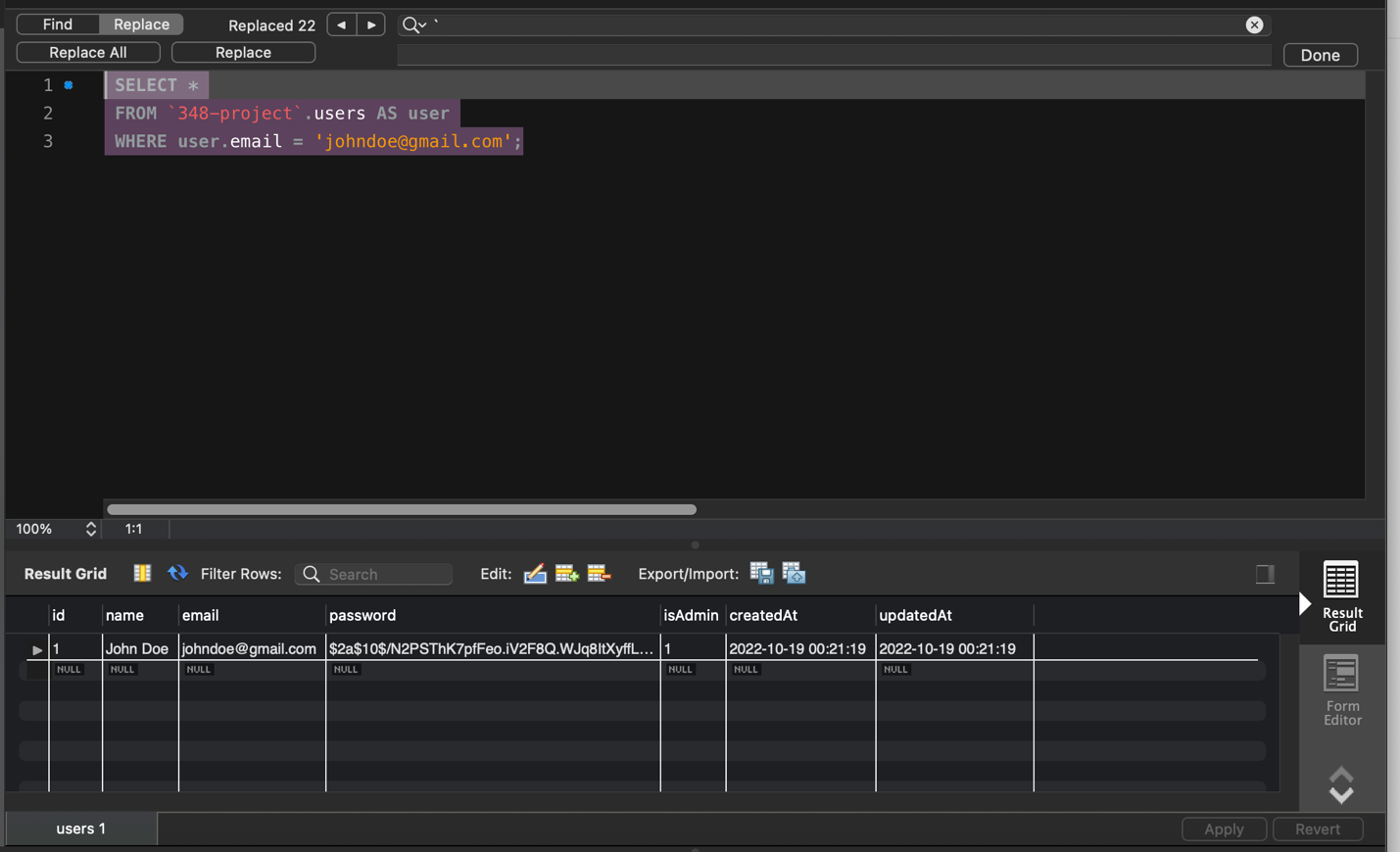
**SQLQuery** for checking if “johndoe@gmail.com” exists in database:

SELECT \*

FROM `348-project`.users AS user

WHERE user.email = 'johndoe@gmail.com';

**Expected output:**



**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.

Text

Description automatically generated

**SQL** Query for update a ticket by its ID:

UPDATE `348-project`.tickets

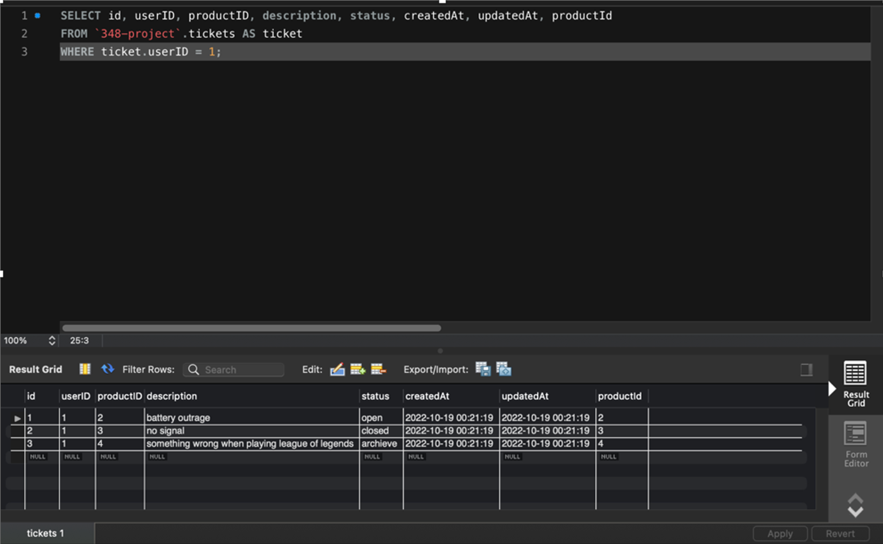
SET description= "new description",

status= "closed",

updatedAt= now()

WHERE id = 3

**Expected Output:**



**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.

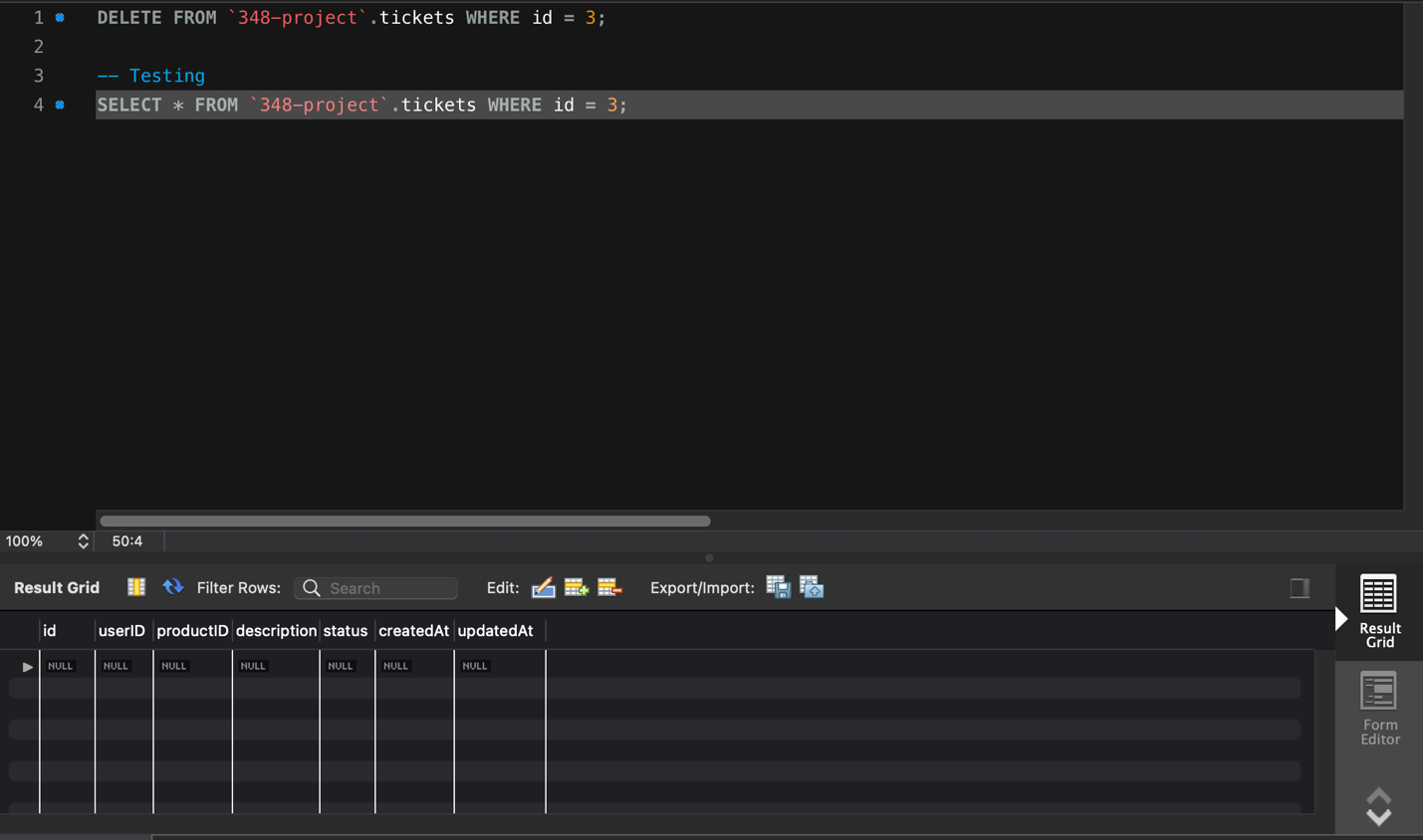
A screenshot of a computer

Description automatically generated with medium confidence

**SQL Query** for Deleting ticket with ID of 3:

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

**Expected Output:**



**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.

Zhongqi Yue:

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

**Link to GitHub repository:**

[**Link**](https://github.com/Zhongqi0402/cs348_project)