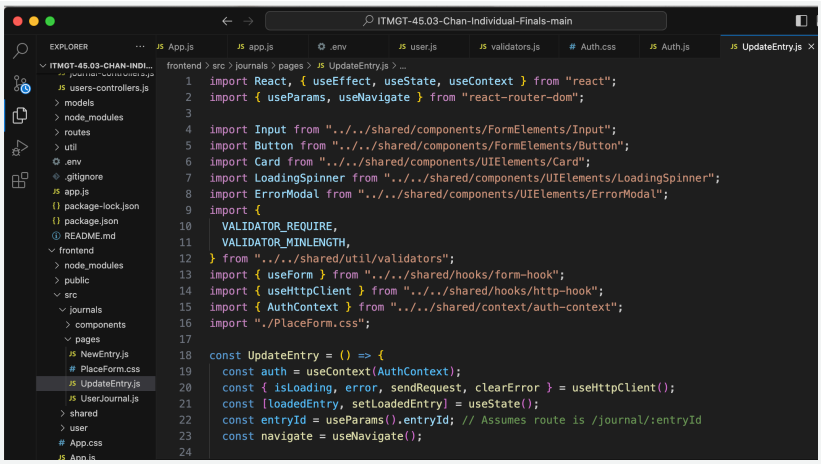
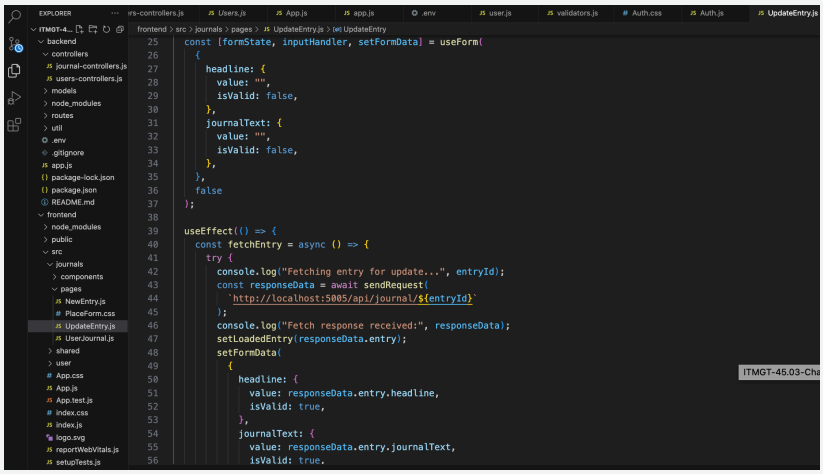
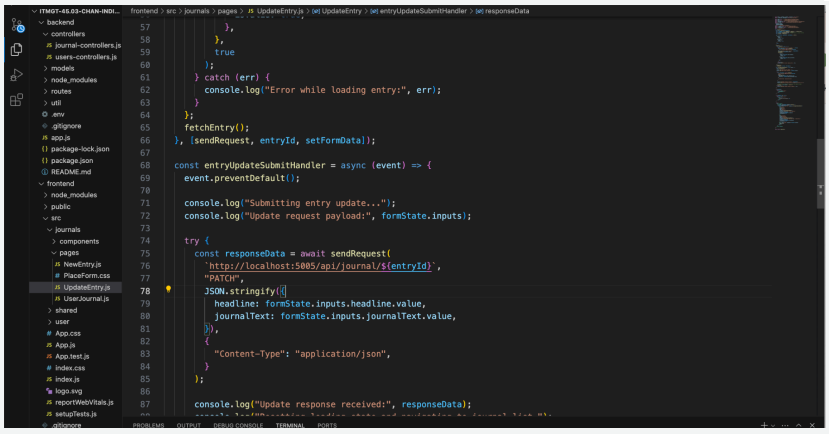
Arianna Chan
BugA_InfiniteSpinner
UpdateEntry.js



```
import React, { useEffect, useState, useContext } from "react";
import { useParams, useNavigate } from "react-router-dom";

import Input from "../../shared/components/FormElements/Input";
import Button from "../../shared/components/FormElements/Button";
import Card from "../../shared/components/UIElements/Card";
import LoadingSpinner from "../../shared/components/UIElements/LoadingSpinner";
import ErrorModal from "../../shared/components/UIElements/ErrorModal";
import {
  VALIDATOR_REQUIRE,
  VALIDATOR_MINLENGTH,
} from "../../shared/util/validators";
import { useForm } from "../../shared/hooks/form-hook";
import { useHttpClient } from "../../shared/hooks/http-hook";
import { AuthContext } from "../../shared/context/auth-context";
import "./PlaceForm.css";

const UpdateEntry = () => {
  const auth = useContext(AuthContext);
  const { isLoading, error, sendRequest, clearError } = useHttpClient();
  const [loadedEntry, setLoadedEntry] = useState();
  const entryId = useParams().entryId; // Assumes route is /journal/:entryId
  const navigate = useNavigate();
```

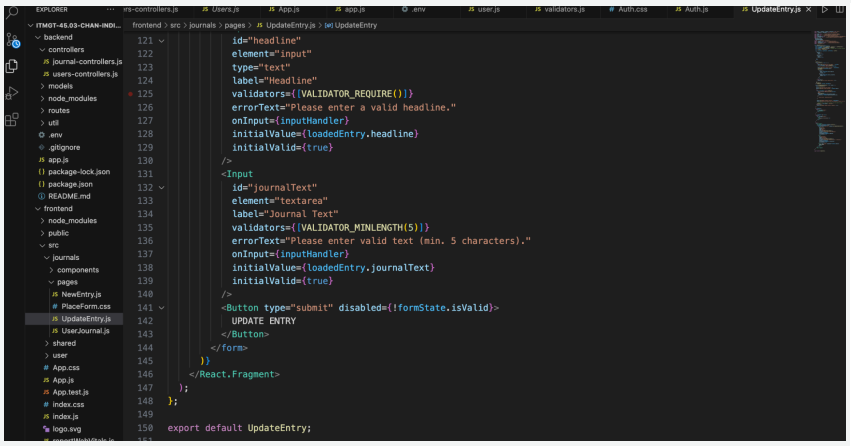


```
const [formState, inputHandler, setFormData] = useForm(
  {
    headline: {
      value: "",
      isValid: false,
    },
    journalText: {
      value: "",
      isValid: false,
    },
  },
  false
);

useEffect(() => {
  const fetchEntry = async () => {
    try {
      console.log("Fetching entry for update...", entryId);
      const responseData = await sendRequest(
        `http://localhost:5005/api/journal/${entryId}`
      );
      console.log("Fetch response received:", responseData);
      setLoadedEntry(responseData.entry);
      setFormData(
        {
          headline: {
            value: responseData.entry.headline,
            isValid: true,
          },
          journalText: {
            value: responseData.entry.journalText,
            isValid: true,
```

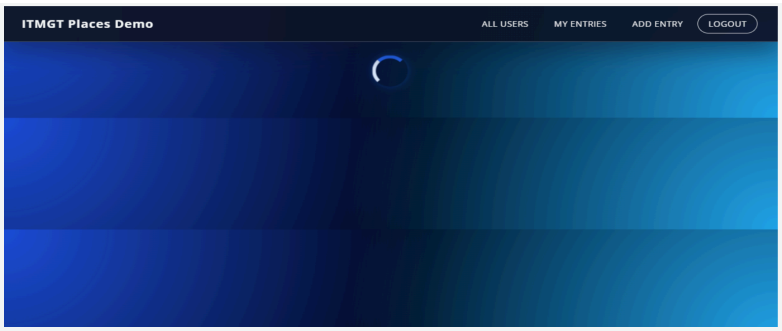


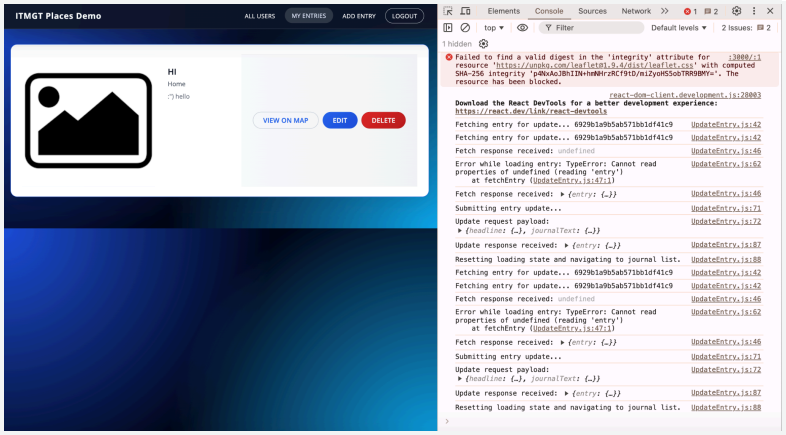
```
        },
        true
      );
    } catch (err) {
      console.log("Error while loading entry:", err);
    }
  };
  fetchEntry();
}, [sendRequest, entryId, setFormData]);

const entryUpdateSubmitHandler = async (event) => {
  event.preventDefault();

  console.log("Submitting entry update...");
  console.log("Update request payload:", formState.inputs);

  try {
    const responseData = await sendRequest(
      `http://localhost:5005/api/journal/${entryId}`,
      "PATCH",
      JSON.stringify({
        headline: formState.inputs.headline.value,
        journalText: formState.inputs.journalText.value,
      }),
      {
        "Content-Type": "application/json",
      }
    );

    console.log("Update response received:", responseData);
```

```
frontend > src > journals > pages > UpdateEntry.js > (e) UpdateEntry > (e) entryUpdateSubmitHandler > (e) responseData
87      console.log("Update response received:", responseData);
88      console.log("Resetting loading state and navigating to journal list.");
89
90      // Navigate back to the user's journal list once update succeeds
91      navigate("/" + auth.userId + "/journal");
92    } catch (err) {
93      console.log("Update request failed in handler:", err);
94    }
95  };
96
97  if (isLoading) {
98    return (
99      <div className="center">
100       <LoadingSpinner />
101     </div>
102   );
103 }
104
105 if (!loadedEntry && !error) {
106   return (
107     <div className="center">
108       <Card>
109         <h2>Could not find entry!</h2>
110       </Card>
111     </div>
112   );
113 }
114
115 return (
116   <React.Fragment>
117     <ErrorModal error={error} onClear={clearError} />
```



```
121           id="headline"
122           element="input"
123           type="text"
124           label="Headline"
125           validators={[VALIDATOR_REQUIRE()]}
126           errorText="Please enter a valid headline."
127           onInput={inputHandler}
128           initialValue={loadedEntry.headline}
129           initialValid={true}
130         />
131         <Input
132           id="journalText"
133           element="textarea"
134           label="Journal Text"
135           validators={[VALIDATOR_MINLENGTH(5)]}
136           errorText="Please enter valid text (min. 5 characters)."
137           onInput={inputHandler}
138           initialValue={loadedEntry.journalText}
139           initialValid={true}
140         />
141         <Button type="submit" disabled={!formState.isValid}>
142           UPDATE ENTRY
143         </Button>
144       </form>
145     )}
146   </React.Fragment>
147 );
148 };
149
150 export default UpdateEntry;
```

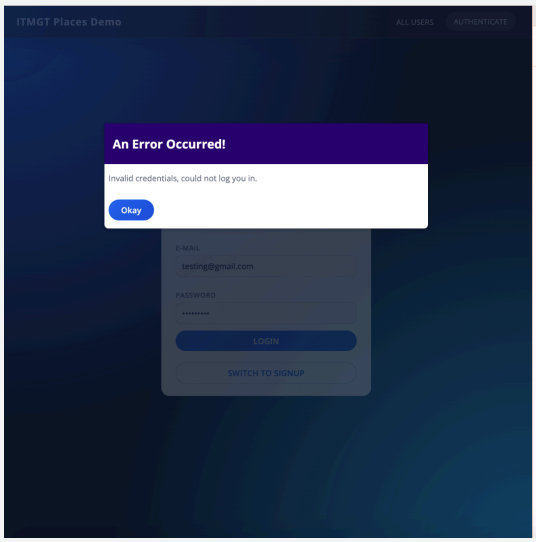




Bug A: The Infinite Spinner was a frontend state management issue located in the UpdateEntry.js component. The problem was a conflict between the necessary isLoading state, which came from the reliable custom HTTP hook (and correctly resets on success), and a redundant local state variable named isSubmitting. When the entry update request was sent, the redundant variable was correctly set to true, causing the spinner to appear. However, the application logic failed to explicitly reset this redundant isSubmitting variable to false upon receiving the successful API response. This failure caused the UI to remain stuck on the loading spinner indefinitely, even though the data update itself completed successfully, a fact confirmed by console logs showing the "Update response received."

The fix involved simplifying the component logic by deleting the unnecessary local state variable and ensuring the UI's loading indicator relied solely on the state managed by the reliable HTTP hook. Furthermore, the fix confirmed that the Maps function call (which redirects the user back to the journal list) was correctly placed to execute only upon successful completion of the API request, thus preventing the user from being stuck in a perpetual loading state.

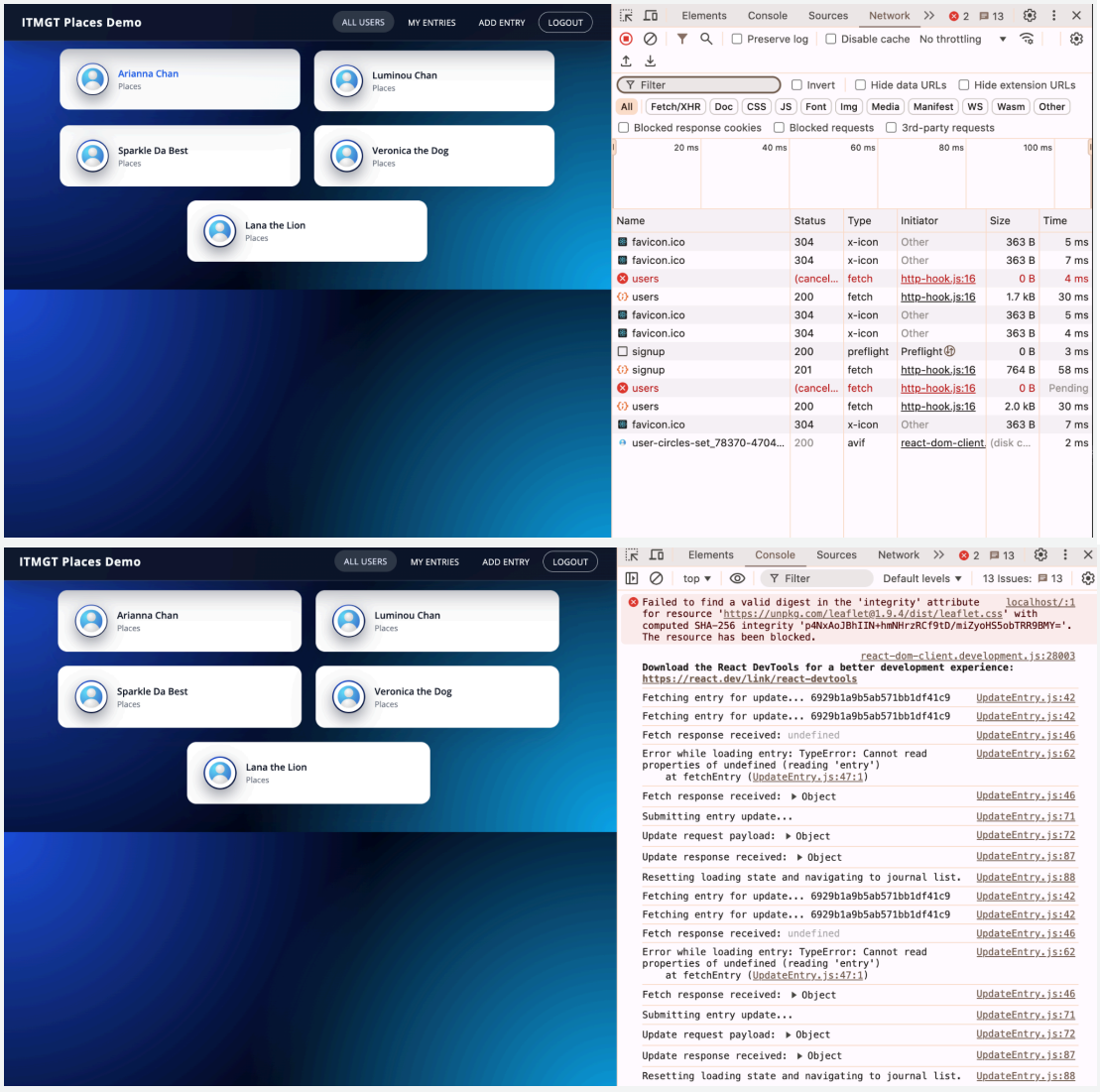
```
73  const login = async (req, res, next) => {
74    const { email, password } = req.body;
75
76    console.log("Login attempt for email:", email);
77    console.log("Request body received:", req.body);
78
79    let existingUser;
80
81    try {
82      existingUser = await User.findOne({ email: email });
83      console.log("User query result:", existingUser);
84    } catch (err) {
85      const error = new HttpError(
86        "Logging in failed, please try again later.",
87        500
88      );
89      return next(error);
90    }
```




```
}
Login attempt for email: testing@gmail.com
Request body received: { email: 'testing@gmail.com', password: 'blablabla' }
User query result: null
```

```
Server running on port 5005
Login attempt for email: ariannaysabel@yahoo.com
Request body received: { email: 'ariannaysabel@yahoo.com', password: '000000' }
User query result: {
  _id: new ObjectId('6929ac61b5ab571bb1df41be'),
  name: 'Arianna Chan',
  firstName: 'Arianna',
  lastName: 'Chan',
  mobileNumber: '09222255385',
  email: 'ariannaysabel@yahoo.com',
  password: '000000',
  image: 'https://img.freepik.com/free-vector/user-circles-set_78370-4704.jpg?semt=ais_incoming&w=740&q=80',
  places: [],
  __v: 0
```

```
79    let existingUser;
80
81    try {
82      existingUser = await User.findOne({ email: email });
83      console.log("User query result:", existingUser);
84    } catch (err) {
85      const error = new HttpError(
86        "Logging in failed, please try again later.",
87        500
88      );
89      return next(error);
90    }
91
92    if (!existingUser || existingUser.password !== password) {
93      const error = new HttpError(
94        "Invalid credentials, could not log you in.",
95        401
96      );
97      return next(error);
98    }
99
```



Fix the "Identity Crisis": In your backend/controllers/users-controllers.js, change the broken line in the login function:

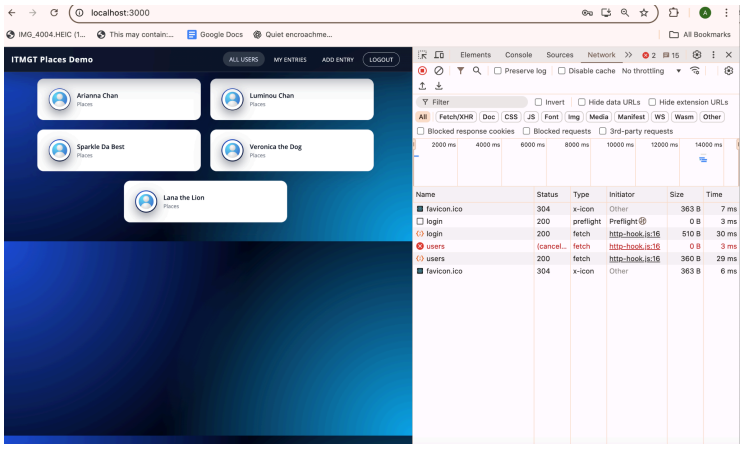
```
// Change this broken line:
existingUser = await User.findOne({ name: email });

// To this corrected line:
existingUser = await User.findOne({ email: email });
```

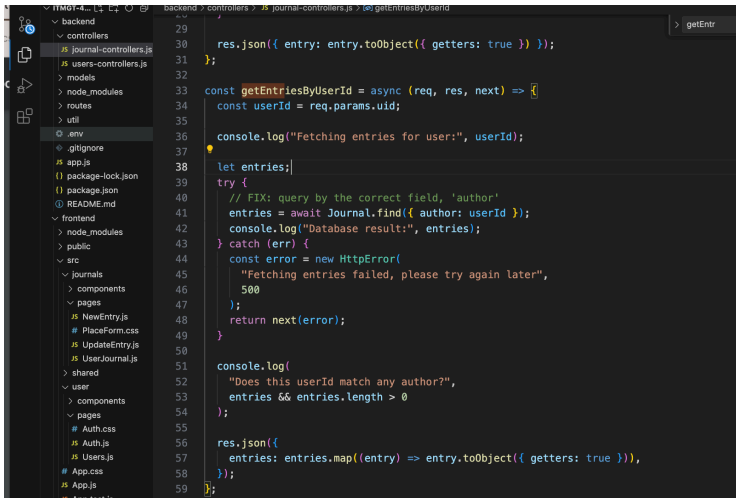
Bug B: The Failing Login Greeting was a full-stack data flow issue that prevented the user's first name from being displayed in the navigation bar after successful login. The problem was twofold: first, the **Backend Failure** was in the login function of backend/controllers/users-controllers.js, where the successful response JSON only included basic authentication details (userId and token) and failed to include the necessary firstName and lastName fields from the user's document. This meant the essential data needed for personalization was discarded at the server level. Second, the **Frontend Failure** was in

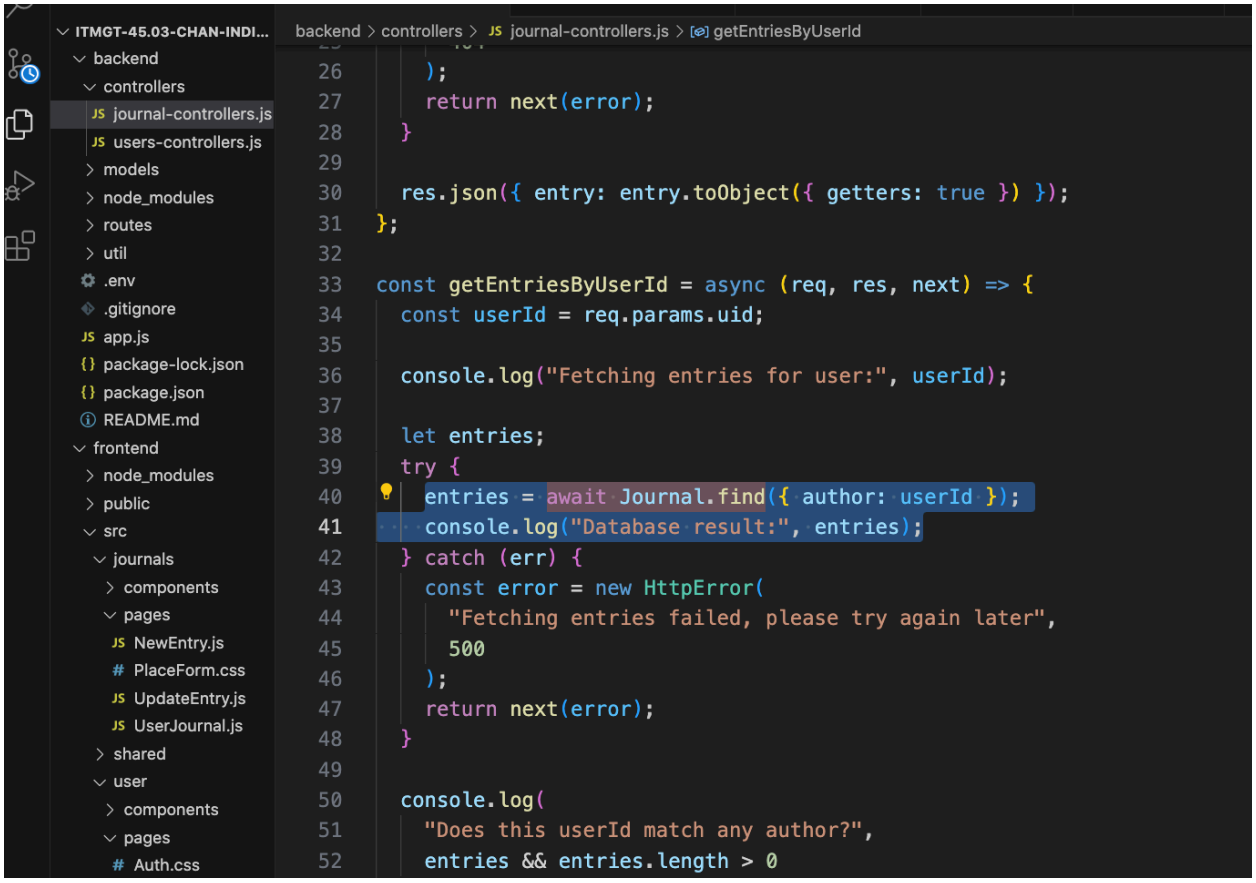
frontend/src/user/pages/Auth.js, where the subsequent call to auth.login() was only configured to accept and pass three arguments, instead of the required five (including the names). This complete breakdown in data transmission meant the authentication context never received the user's name.

The solution required synchronizing the data flow across both layers. The backend was fixed by updating the res.json() call to explicitly include the two missing firstName and lastName fields in the response payload. Concurrently, the frontend's auth.login() function call was updated to correctly receive these two new properties from the API response and pass them into the global application state. This ensures the user's full name is available immediately upon login, allowing the navigation bar to personalize the greeting.

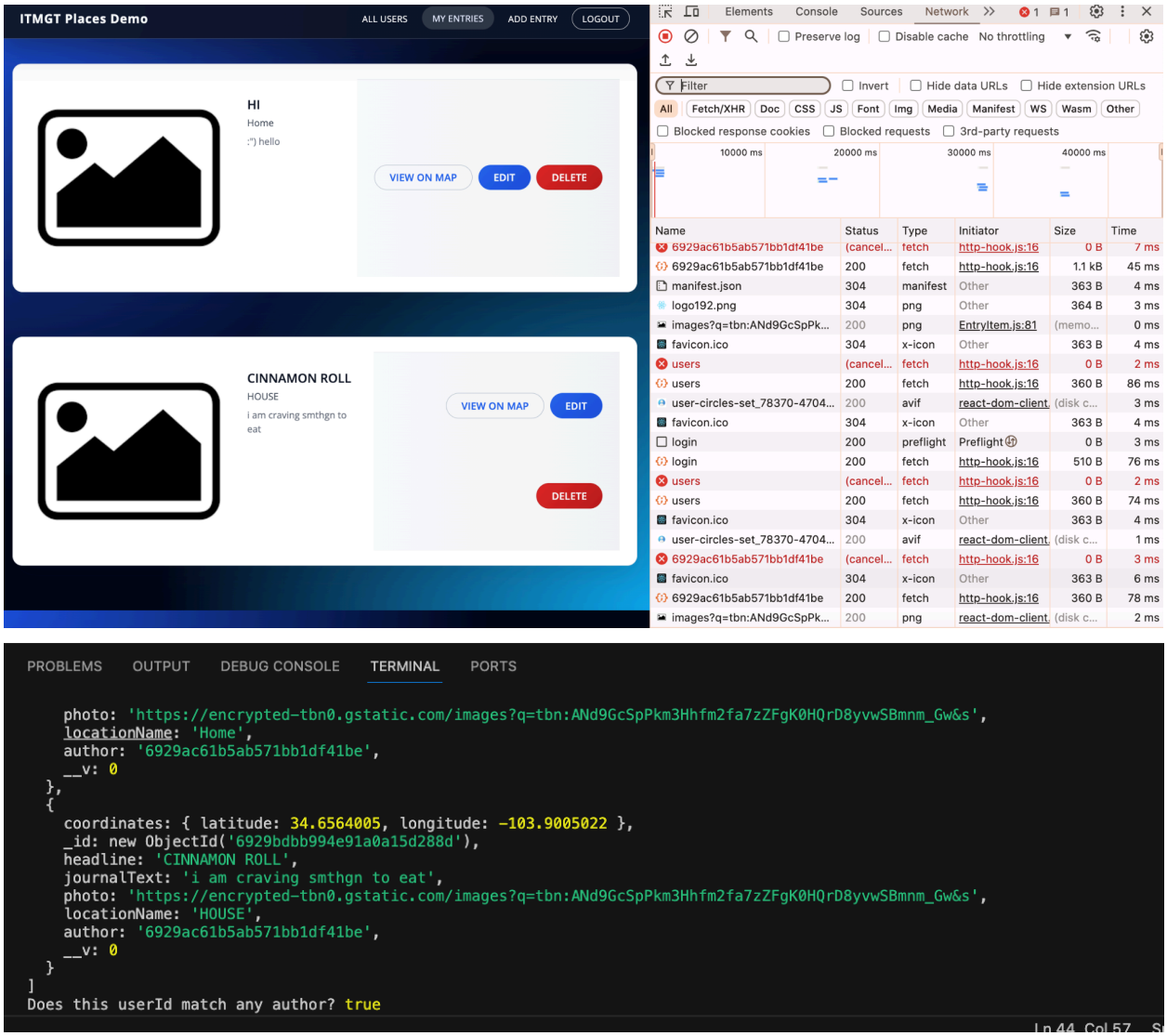
BugC_PhantomList






ITMGT-45.03-CHAN-INDI...
backend
controllers
journal-controllers.js
users-controllers.js
models
node_modules
routes
util
.env
.gitignore
app.js
package-lock.json
package.json
README.md
frontend
node_modules
public
src
journals
components
pages
NewEntry.js
PlaceForm.css
UpdateEntry.js
UserJournal.js
shared
user
components
pages
Auth.css
backend > controllers > journal-controllers.js > getEntriesByUserId
);
return next(error);
}
res.json({ entry: entry.toObject({ getters: true }) });
};
const getEntriesByUserId = async (req, res, next) => {
const userId = req.params.uid;
console.log("Fetching entries for user:", userId);
let entries;
try {
entries = await Journal.find({ author: userId });
console.log("Database result:", entries);
} catch (err) {
const error = new HttpError(
"Fetching entries failed, please try again later",
500
);
return next(error);
}
console.log(
"Does this userId match any author?",
entries && entries.length > 0


}
res.json({ entry: entry.toObject({ getters: true }) });
};



Bug C: The Phantom List occurred when a logged-in user navigated to the "My Entries" page, but the page incorrectly displayed "No journal entries found," even though entries existed in the database for that user. This issue stemmed from two compounding errors in the backend function getEntriesByUserId within journal-controllers.js. The first critical error was an incorrect database query: it attempted to search for entries using the non-existent field creator (Journal.find({ creator: userId })) instead of the correct field, author. This meant the query always failed to find any matching documents, returning an empty result set from MongoDB. The second error was a hardcoded line in the controller that intentionally threw away any result (even if one was magically found) and forced the API to return an empty array (return res.json({ entries: [] });) to the frontend, guaranteeing the list would never populate.

To fix the issue, a two-part solution was implemented in journal-controllers.js. First, the database query was corrected to use the proper field (author), ensuring that valid journal entries belonging to the userId were correctly retrieved from MongoDB. Second, the hardcoded empty response was removed and replaced with a standard response that sends the actual retrieved data back to the frontend. This allowed the "My Entries" page to successfully receive the journal list and populate the UI, resolving the "phantom list" behavior.