Sprint 3

For

TigerTix Development

Prepared by Jonah Colestock and Chris Skelly
Clemson University

Version 0.1
November 18, 2025

Table of Contents

Revision History

Name	Date	Reasons for Change	Version
Jonah Colestock, Chris Skelly	November 18, 2025	Initial version	0.1

Testing Strategy

Test Cases

Several unit tests related to the use cases of the website were created. These tests were created using Jest, and include tests for loading the website, switching to registering from the login screen, logging in using a dummy account, using the ticket confirmation, buying a ticket, and using the chatbot to book tickets.
Alongside the unit tests, manual testing was also done for logging in, registering, purchasing tickets, using the confirmation system, using the chatbot, and using the speech recognition system.

Expected Results

The test for loading the website checks if the login header text is visible to the user. The test for registering simulates a user clicking on the button to switch to registering and checking if the register textboxes are then visible. The login test enters test account information into the email and password boxes, then simulates clicking the login button and checks if the main page is visible. The test for using ticket confirmation simulates a user clicking on one of the “Buy Ticket” buttons and checking if the “Yes” and “No” buttons are then visible. The test for buying a ticket repeats the same process as the previous test, but additionally stores the current ticket amount, simulates buying a ticket, then checks if the ticket amount decreases. The test for the chatbot simulates clicking the chatbot button, entering a prompt, submitting the prompt, then checking if the confirmation appears.

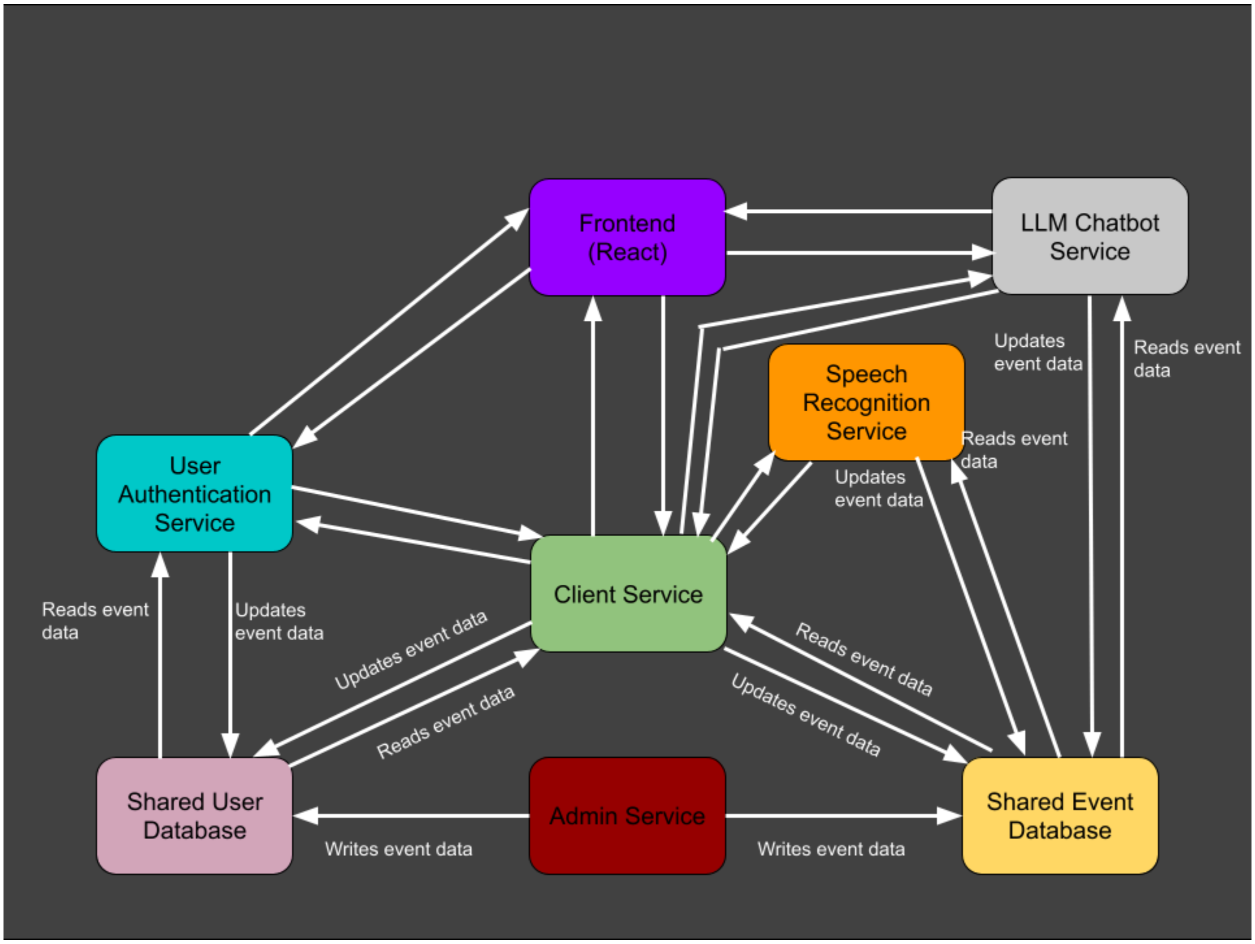
Actual Results

While most of the unit tests were mostly smooth to implement, testing the login and chatbot functionality using unit testing was a more challenging task because it would repeatedly run into an issue where server calls were not being recognized.
Manual testing of the confirmation system helped find a bug where the confirmation would appear on every event when one event was being purchased from.

Test Log

A brief test log can be found here: Test Log

Software Architecture



Accessibility

The front end website application supports the ability to navigate between buttons via pressing the Tab key. This makes it possible to navigate the website without the use of a mouse. Also, a speech recognition service allows for speaking commands without having to type in information, and a text-to-speech service voices text on the screen. Through these combined features, this offers those with difficulties in vision the ability to navigate and use the application.

Security Justification

```
app.post("/api/register", async (req, res) => {
  console.log("🔥 Incoming /api/register body:", req.body);
  const { username, password } = req.body;
  if (!username || !password) return res.status(400).json({ message: "Username and password required" });

  db.get("SELECT * FROM savedaccounts WHERE username = ?", [username], async (err, row) => {
    if (err) return res.status(500).json({ message: err.message });
    if (row) return res.status(400).json({ message: "User already exists" });

    const hashedPassword = await bcrypt.hash(password, 10);

    db.run(
      "INSERT INTO savedaccounts (username, password) VALUES (?, ?)",
      [username, hashedPassword],
      function (err2) {
        if (err2) return res.status(500).json({ message: err2.message });
        res.status(201).json({ message: "Registration successful" });
      }
    );
  });
});
```

The code example above shows the authentication function for user registration. After ensuring that the user's credentials are valid, the program hashes the password using bcrypt, and saves that encrypted password in the shared database. This is important because if someone breaks into the database, they will still be unable to access users' passwords. Furthermore, below is an example of the JSON Web Token system used for authentication.

```
const SECRET_KEY = 'my_evil_super_secret_key';

app.post("/api/login", (req, res) => {
  const { username, password } = req.body;

  db.get("SELECT * FROM savedaccounts WHERE username = ?", [username], async (err, user) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!user) return res.status(400).json({ error: "Invalid username or password" });

    const match = await bcrypt.compare(password, user.password);
    if (!match) return res.status(400).json({ error: "Invalid username or password" });

    const token = jwt.sign({ username: user.username}, SECRET_KEY, { expiresIn: '30m' });
    res.cookie("token", token, { httpOnly: true });
    res.json({ message: "Login successful" });
  });
});
```

In this example, you can see the process of logging in. The system stores a secret key, and when the user logs in a new token is created for them using their username and the secret key. The program stores this token in a web cookie, and it is set to expire 30 minutes after it is created. The token is helpful for authenticating user actions without constantly needing to make potentially-costly server calls.

One possible vulnerability in the system is the fact that the JWT key is stored directly in the source code. If some malefactor was to access the program's code, they would be able to replicate JWTs with only a person's username and therefore fool the authentication system. This could be fixed by storing the key somewhere hidden in an environment file.