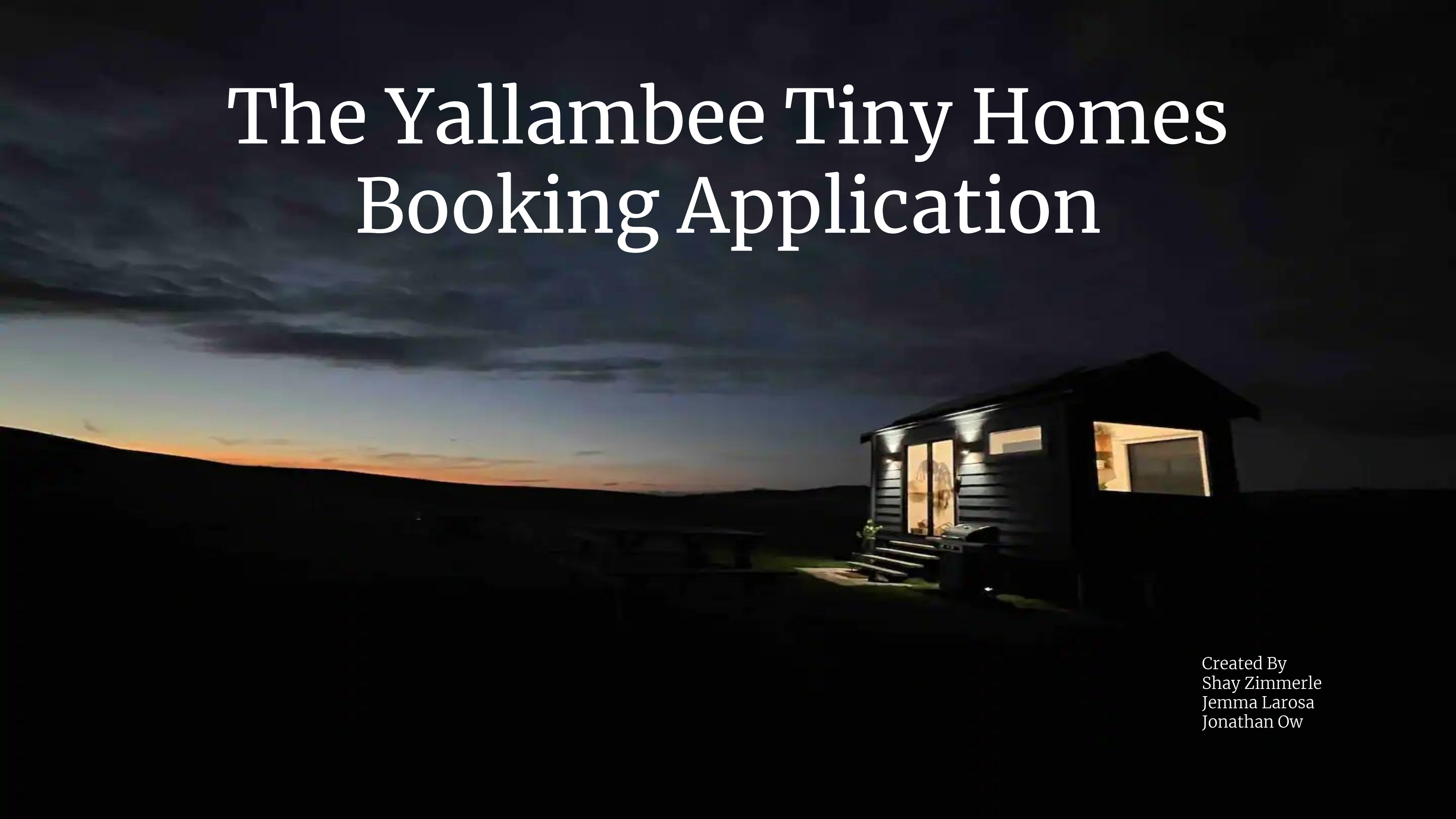


The Yallambee Tiny Homes Booking Application

Created By
Shay Zimmerle
Jemma Larosa
Jonathan Ow

Yallambee
TINY HOME

The Yallambee Tiny Homes Booking App is designed to provide a seamless platform for users to browse and book stays in a unique tiny home. This app offers control over bookings, allowing the client to provide a more personalised experience for their guests. Designed to empower guests to explore the Yallambee property, check availability, make bookings, and manage their reservations effortlessly, all in one place.

Main Features

Booking System

- Users can search for available tiny homes, view detailed property information, and make reservations.
- The system handles booking conflicts and ensures accurate availability updates.

Property Listings

- Property owners can list their tiny homes with detailed descriptions, photos, and pricing information.
- Users can filter properties based on various criteria, such as location and amenities.

User Authentication

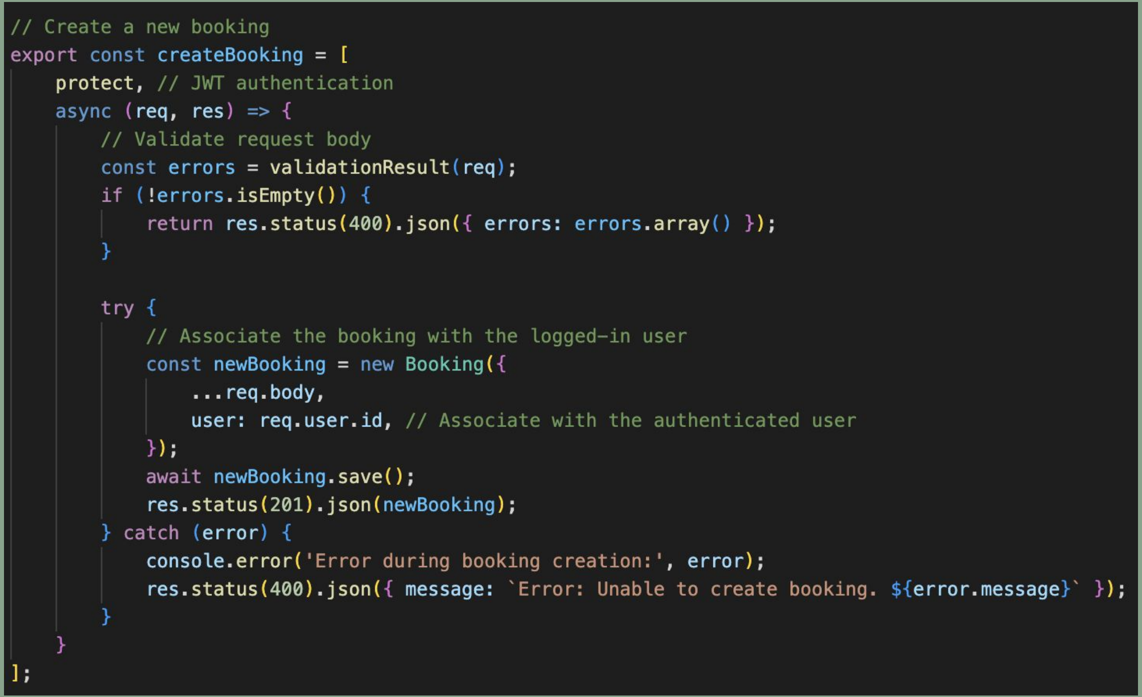
- Secure user registration and login processes.
- Users can manage their profiles, view booking history, and access exclusive features.

Admin Dashboard

- Administrators have access to a dashboard for managing bookings, properties, and user accounts.
- Includes functionality for approving or rejecting property listings and handling user reviews.

createBooking Code Example

We have chosen to highlight our createBooking function, which plays a vital role in our booking management system.
This function is responsible for handling the creation of new bookings, ensuring that all incoming data is securely validated, authenticated, and processed before being stored in the database.
It demonstrates integration of JWT authentication, input validation, and conditional logic, all working together to maintain the integrity and security of the booking system.



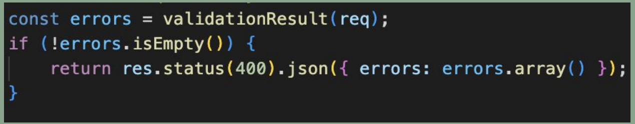
```javascript
// Create a new booking
export const createBooking = [
    protect, // JWT authentication
    async (req, res) => {
        // Validate request body
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array() });
        }

        try {
            // Associate the booking with the logged-in user
            const newBooking = new Booking({
                ...req.body,
                user: req.user.id, // Associate with the authenticated user
            });
            await newBooking.save();
            res.status(201).json(newBooking);
        } catch (error) {
            console.error('Error during booking creation:', error);
            res.status(400).json({ message: `Error: Unable to create booking. ${error.message}` });
        }
    }
];
```

Network Request and Conditional Statement

The **createBooking** function is essential for ensuring both security and data integrity within our booking system. It begins with a network request tied to JWT authentication, utilising the **protect** middleware as a security layer. This middleware intercepts incoming requests and verifies the validity of the JWT token provided by the user. If the token is missing or invalid, the request is blocked and access is denied, ensuring that only authenticated users can interact with the booking API.

Following successful authentication, the function moves on to validating the input data using the **validateUser** array, powered by the **express-validator** library. This middleware enforces strict rules to ensure the email format is correct, the username meets a minimum length of three characters, and the password is at least six characters long. It can also validate the date of birth if provided.

The function then employs a critical conditional statement to handle input validation before any database operations occur. Using **express-validator**, the function checks for any validation errors:



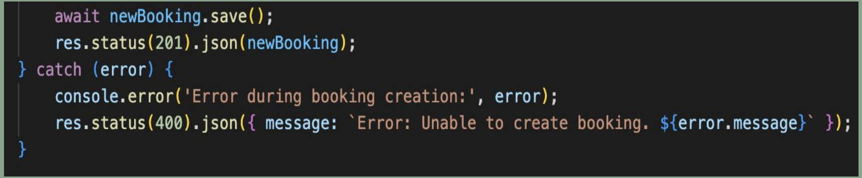
```
const errors = validationResult(req);
if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
}
```

This line of code checks whether the input data meets the required standards, such as correct format and necessary fields. If any validation errors are detected, the function immediately returns a 400 status with a detailed error message, effectively preventing the creation of a booking with invalid data. This step is crucial for maintaining data integrity and ensuring that only valid and meaningful information is processed by the application.

Database Operation

Once the data is validated, the function performs a database operation by creating a new instance of the Booking model and saving it to the database. The function first associates the booking with the authenticated user by including the user's ID (**req.user.id**) in the booking data. This association is important for tracking which user made the booking, providing a clear link between the user and their booking records.

The function then performs a database operation which POST's the new booking to the database using - **await newBooking.save()**. This database operation stores the booking data making it retrievable for future use and reference. As seen in the below example.



```
    await newBooking.save();
    res.status(201).json(newBooking);
} catch (error) {
    console.error('Error during booking creation:', error);
    res.status(400).json({ message: `Error: Unable to create booking. ${error.message}` });
}
```

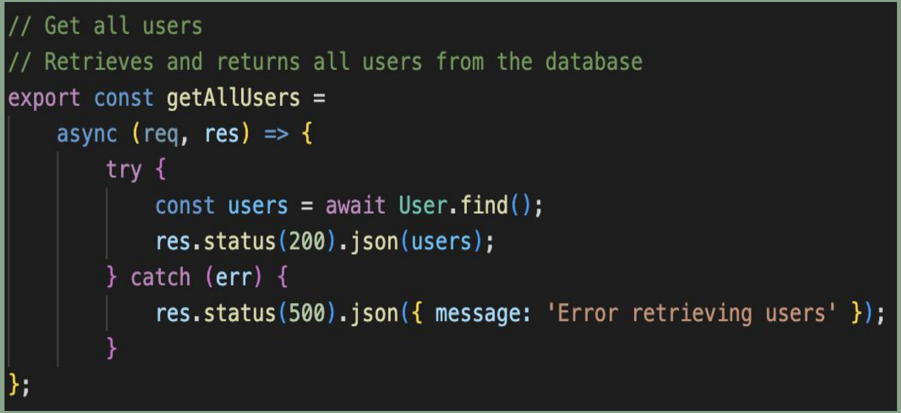
Finally, the function executes the booking creation process within a **try…catch** block to handle potential errors. If an error occurs during the database operation or any other part of the process, the function catches and logs the error, then returns a 400 status with an explanatory message to the user. This error handling is essential to ensure that the application can gracefully manage unexpected issues without crashing while also providing feedback that is both informative to the user and useful for debugging.

Challenges

1. In the development of our booking application, we encountered a challenge with the admin panel not displaying all users, which was tied to the network request used to fetch user data. The fetchUsers function, which sends a GET request to the /api/users endpoint, plays a crucial role here. This function includes a conditional statement that checks if the response is successful before processing the data.

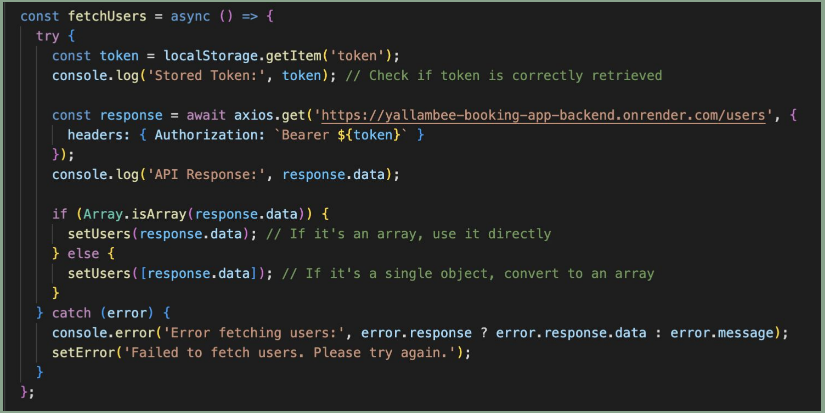
On the backend, the route responsible for fetching users involves database operations to retrieve user data. Ensuring that this route was correctly implemented was essential for resolving the issue. Additionally, while the frontend code does not explicitly show a loop, the process of displaying users involves iterating over the user data to render it in the UI. The function call setUsers updates the state with the fetched data, ensuring the admin panel displays the correct information.

To resolve the issue, we reviewed and revised both the backend route and the frontend API integration. Backend documentation was consulted to confirm the route setup, while frontend debugging tools like Chrome DevTools were used to trace API requests and verify data handling. This approach effectively addressed the challenge and ensured that the admin panel correctly displayed the user data.



```javascript
// Get all users
// Retrieves and returns all users from the database
export const getAllUsers =
    async (req, res) => {
        try {
            const users = await User.find();
            res.status(200).json(users);
        } catch (err) {
            res.status(500).json({ message: 'Error retrieving users' });
        }
};
```

Backend



```javascript
const fetchUsers = async () => {
  try {
    const token = localStorage.getItem('token');
    console.log('Stored Token:', token); // Check if token is correctly retrieved

    const response = await axios.get('https://yallambee-booking-app-backend.onrender.com/users', {
      headers: { Authorization: `Bearer ${token}` }
    });
    console.log('API Response:', response.data);

    if (Array.isArray(response.data)) {
      setUsers(response.data); // If it's an array, use it directly
    } else {
      setUsers([response.data]); // If it's a single object, convert to an array
    }
  } catch (error) {
    console.error('Error fetching users:', error.response ? error.response.data : error.message);
    setError('Failed to fetch users. Please try again.');
  }
};
```

Frontend

Resources and Resolution

Frontend Debugging Tools Utilised

Frontend debugging tools, specifically Chrome DevTools, were pivotal in diagnosing and resolving issues with the admin panel user display functionality. The **Network tab** in Chrome DevTools was particularly instrumental in tracing API requests made by the **fetchUsers** function. By accessing DevTools and navigating to the Network tab, we could monitor all network activity related to the **/users** endpoint. This feature allowed us to filter requests, review headers and verify that the correct URL and authorization token were being used. Importantly, it allowed us to examine the response data to ensure it was returned in the expected format, such as a JSON array of user objects. This helped identify discrepancies in the response format which were crucial in pinpointing if the was related to either the backend or frontend.

Additionally, the **Console tab** in Chrome DevTools was vital for error logging and debugging. It provided a real-time view of logs, errors, and warnings generated by the application. By incorporating **console.error** statements within our error handling blocks, we were able to capture and review detailed error messages related to API requests. The Console also helped with interactive debugging, allowing us to run JavaScript commands and inspect the variables directly. This functionality proved helpful for debugging data flow issues and ensuring that response data was processed and displayed correctly in the admin panel. Together, these tools enabled effective troubleshooting, ensuring that the data handling was accurate and that the user interface functioned as intended.

Backend Documentation Utilised

Backend documentation played a crucial role in the resolution, it ensured that our applications API routes were implemented correctly and meet all necessary requirements. It provided a comprehensive guide on the structure of API routes, expected data formats, and authentication protocols. When addressing issues with API integration, such as the admin panel failure to display user data, consulting this documentation was essential. It confirmed that the API route was set up correctly, adhered to security protocols, and handled data as specified.

For instance, when the **fetchUsers** function makes a GET request to the **/users** endpoint, the backend documentation outlines the following critical details:

- **Route Definition**
  The endpoint **/users** uses the GET method to retrieve a list of all users from the database.

- **Data Formats**
  The request does not require a body but must include an Authorization header. The response should be a JSON array of user objects, each containing fields like **id**, **username**, and **email**.

- **Authentication Requirements**
  The request must include a valid JWT token in the Authorization header to authenticate and authorise the request.

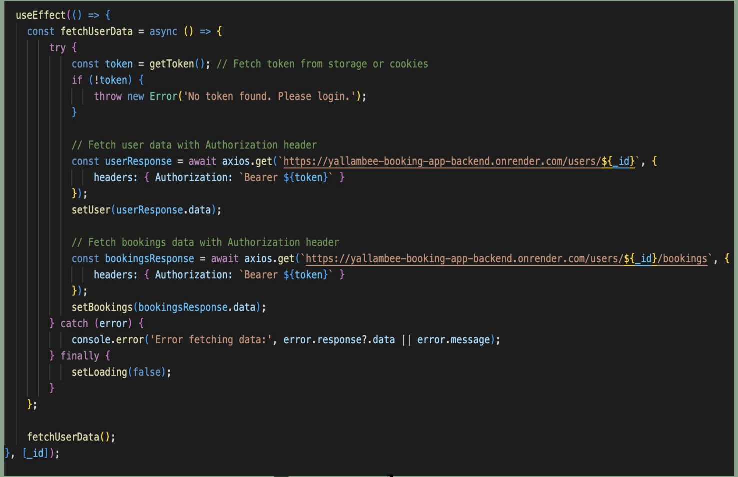
By reviewing this documentation this ensured that the **/users** route was correctly implemented and secured. This allowed us to verify that the endpoint was set up to handle GET requests, the response format matched the frontend's expectations, and the required Authorization header was properly included in frontend requests. This thorough review and adherence to documentation ensured that the backend route functioned as intended and provided the correct data for the frontend display.

Challenges

2. During the development of our booking application, we encountered a challenge with the profile page failing to display user data, despite using an admin bearer token that should have granted access to all users and their bookings. The fetchProfileData function, which is crucial for handling network requests to endpoints like /users and /users/:id/bookings, faced significant issues. This function involves making network requests and managing authentication tokens to ensure proper access. The problem was identified not with the IDs or API endpoints but with the token validation process.

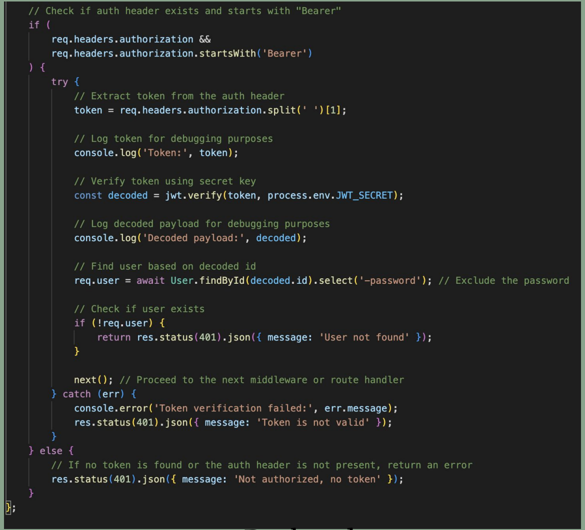
Although the token was valid and permissions were set correctly, the console log repeatedly showed the error message: "token not valid." This indicated that the issue was related to how the token was being validated rather than its authenticity or user permissions.

To resolve this issue, we reviewed the backend routes and consulted the API documentation to confirm that the token validation process was correctly implemented. Frontend debugging tools and error logs were used to pinpoint issues with token handling. By ensuring proper token validation on the backend and verifying the correct integration of these routes on the frontend, we were able to address the problem and enable the profile page to fetch and display user data correctly.



```
useEffect(() => {
  const fetchUserData = async () => {
    try {
      const token = getToken(); // Fetch token from storage or cookies
      if (!token) {
        throw new Error('No token found. Please login.');
      }

      // Fetch user data with Authorization header
      const userResponse = await axios.get(`https://yallambee-booking-app-backend.onrender.com/users/${_id}`, {
        headers: { Authorization: `Bearer ${token}` }
      });
      setUser(userResponse.data);

      // Fetch bookings data with Authorization header
      const bookingsResponse = await axios.get(`https://yallambee-booking-app-backend.onrender.com/users/${_id}/bookings`, {
        headers: { Authorization: `Bearer ${token}` }
      });
      setBookings(bookingsResponse.data);
    } catch (error) {
      console.error('Error fetching data:', error.response?.data || error.message);
    } finally {
      setLoading(false);
    }
  };

  fetchUserData();
}, [_id]);
```

Frontend



```
  // Check if auth header exists and starts with "Bearer"
  if (
    req.headers.authorization &&
    req.headers.authorization.startsWith('Bearer')
  ) {
    try {
      // Extract token from the auth header
      token = req.headers.authorization.split(' ')[1];

      // Log token for debugging purposes
      console.log('Token:', token);

      // Verify token using secret key
      const decoded = jwt.verify(token, process.env.JWT_SECRET);

      // Log decoded payload for debugging purposes
      console.log('Decoded payload:', decoded);

      // Find user based on decoded id
      req.user = await User.findById(decoded.id).select('-password'); // Exclude the password

      // Check if user exists
      if (!req.user) {
        return res.status(401).json({ message: 'User not found' });
      }

      next(); // Proceed to the next middleware or route handler
    } catch (err) {
      console.error('Token verification failed:', err.message);
      res.status(401).json({ message: 'Token is not valid' });
    }
  } else {
    // If no token is found or the auth header is not present, return an error
    res.status(401).json({ message: 'Not authorized, no token' });
  }
};
```

Backend

Resources and Resolution

Backend and Token Handling

Checking Token Retrieval
We began by verifying that the frontend was correctly retrieving the admin bearer token from storage and including it in the headers of API requests. Ensuring that the token was sent with each request was crucial for accessing protected routes. This step confirmed that the token, which was vital for user authentication and authorization, was being utilised properly in API requests.

Backend Validation
Next, we focused on the backend, where we reviewed the token validation logic. This involved checking that the token validation middleware was correctly implemented and functioning as expected. We made sure that the backend was accurately parsing and validating the token against the required standards to ensure that access to the requested resources was properly controlled.

Console Logs and Code Review

Console Logs
To further diagnose the issue, we examined the console logs for additional error details. The logs provided valuable insights into where the problem might be occurring, including unexpected error messages or warnings related to token validation. This information was crucial for pinpointing the exact nature of the issue.

Code Review
Through a thorough code review, we discovered that the issue stemmed from a misconfiguration in how the token was being processed or validated. Despite the token itself being valid, this misconfiguration led to token validation failures. Identifying this misconfiguration was key to resolving the issue.

PRs and Fixes

Backend Fixes
We addressed the identified token validation issues by submitting a pull request (PR) to the backend. This PR included updates to the token validation logic, ensuring that the token was processed correctly and addressing any authentication and authorization problems.

Front End Adjustments
On the frontend, necessary adjustments were made to ensure that the token was correctly included in API requests. These updates were essential for aligning the frontend with the revised backend logic and ensuring proper token utilisation in fetching user data.

By addressing both backend and frontend aspects, we successfully resolved the token validation issues, enabling the profile page to accurately fetch and display user data. This comprehensive approach ensured that the system’s authentication and data retrieval processes functioned as intended.