OPEN WINDOW

Exhibito

Project Presentation

Interactive Development 200 Term 2

Kayla Posthumus
231096

Iné Smith
221076

Frederick Beytell
231374

Wolf Botha
21100255

Our Group

Kayla Posthumus

231096

Fun Fact:
I used to play classical guitar in high school.

Iné Smith

221076

Fun Fact:
I grew up on a farm in North West and went to a school where we were only 5 children in Gr. 7

Frederick Beytell

231374

Fun Fact:
I ride a motorcycle and am a secret metalhead, but I'm about as quiet, calm and collected as you can be.

Wolf Botha

21100255

Fun Fact:
DNA test results found that I'm 5% of Middle Eastern descent.

Exhibito

Project Introduction

Exhibito is a web service that serves as a dynamic platform where arthouses can post and manage their exhibition events.

This project was built on the foundation of the MERN stack and implements technologies like Axios and React Router. The project placed a strong emphasis on implementing security measures which included encryption and decryption and user validation.

We were required to implement user roles that give different types of users unique permissions. These roles consist of the standard, moderator and arthouse roles.

Frederick Beytell

Tech Stack

Essential front-end languages

- HTML
  - Purpose: Structure and Content
  - Function: HTML is the standard markup language used to create the basic structure and content of a web page.
- CSS
  - Purpose: Presentation and Styling
  - Function: CSS is used to control the presentation, formatting, and layout of the web pages HTML elements, such as colors, fonts, spacing, and positioning.
- JavaScript
  - Purpose: Interactivity and Functionality
  - Function: JavaScript is a programming language used to create dynamic and interactive effects and perform tasks on web pages.

Frederick Beytell

Tech Stack

MERN Stack

- MongoDB
  - Role: Database
  - Function: MongoDB is a NoSQL database that stores data in flexible, JSON-like documents. It is used for storing and retrieving data. We use this for storing and retrieving data necessary for the site's operations.
- Express.js
  - Role: Web Application Framework
  - Function: Express.js is a Node.js web application framework that provides a robust set of features for web and mobile applications. It is used for building the backend of the application and handling HTTP requests.
- React
  - Role: Frontend Library
  - Function: JavaScript library for building user interfaces. It is used to create reusable UI components and to efficiently update and render the user interface.
- Node.js
  - Role: JavaScript Runtime Environment
  - Function: Node.js allows developers to run JavaScript on the server side.

Frederick Beytell

Tech Stack

Other technologies

- Tailwind CSS
  - Purpose: Utility-First CSS Framework
  - Function: Tailwind CSS is a highly customizable, utility-first CSS framework that provides a set of low-level utility classes to build custom designs directly in your HTML.
- Axios
  - Purpose: HTTP Client for Making API Requests
  - Function: Axios is a promise-based HTTP client used with React and other frontend frameworks to make asynchronous HTTP requests to interact with APIs. It simplifies sending GET, POST, PUT, DELETE, and other HTTP requests and handling responses.
- React Router
  - Purpose: Client-Side Routing Library for React
  - Function: React Router enables the creation of single-page applications with dynamic routing, allowing users to navigate between different views or pages without reloading the entire page. React Router manages the URL and renders the corresponding components.

Project Purpose

Purpose & Goal

The purpose of this project is to develop a dynamic and user-friendly web application using the MERN stack, aiming to completely transform how people discover, explore, and purchase art exhibitions. By bridging the gap between art enthusiasts and exhibitions, our platform will provide an intuitive digital solution where users can effortlessly browse, comment on, and purchase tickets for art exhibitions. Additionally, galleries and art hosts will be empowered to manage and publish exhibitions with ease, showcasing the effectiveness of contemporary web development tools in creating significant digital solutions.

Project Requirements

Project Requirements with the Brief

Our project required developing a full-featured web application using the MERN stack (MongoDB, Express.js, React, Node.js) to manage user accounts, art exhibitions, and ticket purchases. Key requirements included user authentication, secure data storage, seamless frontend-backend communication, and an intuitive user interface. Users can browse, purchase tickets, and comment on exhibitions, while art houses can publish and manage their events.

To meet these requirements, we implemented secure user authentication, designed user-friendly interfaces for browsing, purchasing and adding events, and ensured smooth data operations. Our application is responsive and accessible across devices, fulfilling the brief's expectations and delivering a platform that effectively connects art lovers with exhibitions.

Iné Smith

User Types

Standard User

Art enthusiasts, art buyers, individuals who wish to book & attend art exhibitions.

Art House Employee

The Art House Employees are associated with specific art houses and manage events for their respective houses.

Administrator

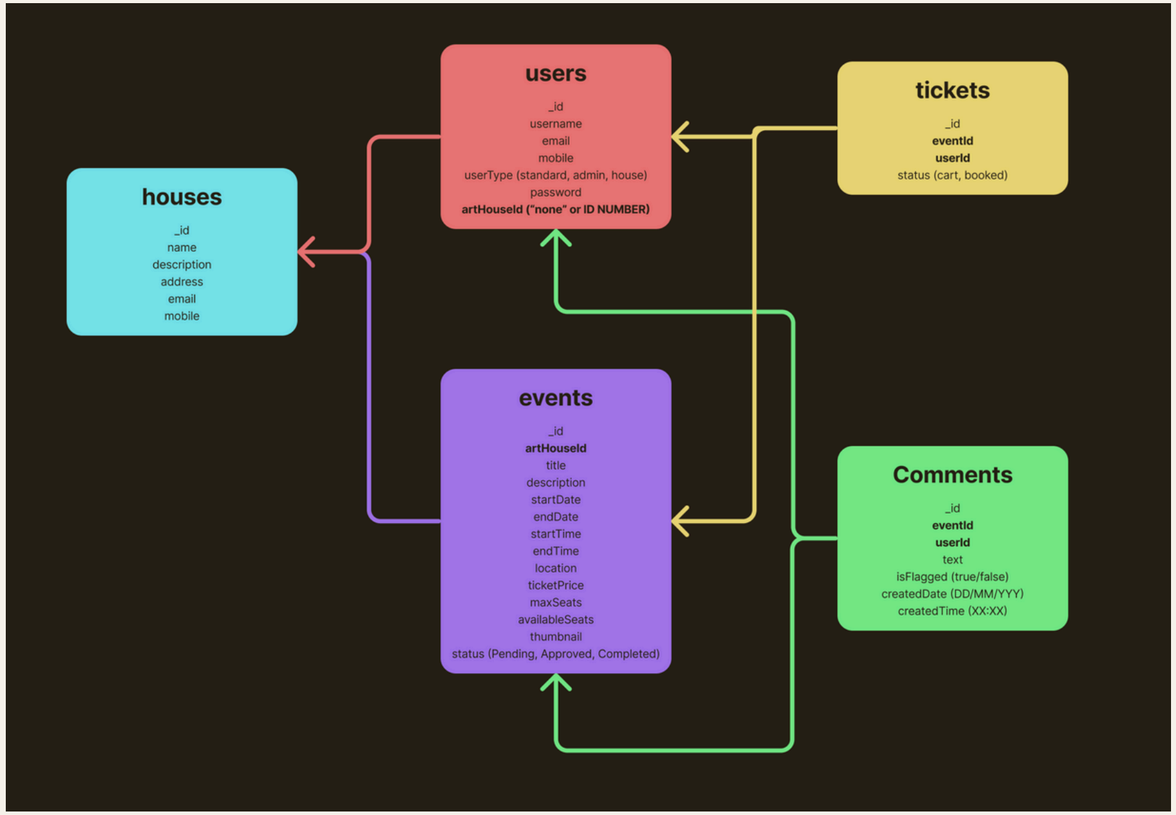
Admins have full control over the platform and its data.

Kayla Posthumus

Website Functionality

Data Flow in the application

The system facilitates interactions between users (standard, admin, house) associated with art houses. Users can book events hosted by these houses, with events containing details like title, dates, and ticket info. Bookings track date and ticket count. Users can comment on events, with each comment linked to a specific event and user, ensuring seamless information flow across users, events, houses, bookings, and comments.



Kayla Posthumus

Data Interaction I

Users and Houses

- House Users: Each house user is linked to a specific art house via artHouseId. This association allows house users to manage events for their respective art houses.
- Admins and Standard Users: Admins and standard users have no specific house association.

Houses and Events

- Events are linked to a specific art house via artHouseId. This relationship ensures that each event is associated with the house that hosts it.
- House users manage these events, they can create events for their Art House

Kayla Posthumus

Data Interaction II

Users and Bookings

- Standard users can make bookings for events. Each booking record in the bookings table includes a userId to link the booking to the user and an eventId to link it to the event.
- Bookings are recorded with details such as the Event Title and booking date.

Events and Comments

- Events can have multiple comments associated with them. The comments table records each comment with a reference to the eventId.
- This allows users to engage with events by sharing feedback and discussions.

Kayla Posthumus

Data Interaction III

Users and Comments

- Users can post comments on events. Each comment in the comments table includes a userId to link it to the user and an eventId to link it to the event.
- Comments include details such as the text of the comment, and the date and time of posting.

Kayla Posthumus

Challenges & Solutions I

Async Operations

Handling Asynchronous operations like API calls to get details of events, usernames or art houses proved difficult at times.

Solution

By using async/await syntax with try & catch blocks to handle errors. We also implemented loading states whilst data was being retrieved from the data base.

Database Design

Designing a database schema to handle all the interactions between users, events, comments, art houses and tickets was a complex task.

Solution

The use of MongoDB with the MongoDB VS-code extension made interaction and setup much easier, by easily viewing the data in a convenient manner.

Challenges & Solutions II

Responsive Design

Ensuring that Exhibito was responsive at different screen sizes took a long time and was challenging.

Solution

Using a combination of custom CSS-classes, React-bootstrap and TailwindCSS expedited the process and made the process more efficient.

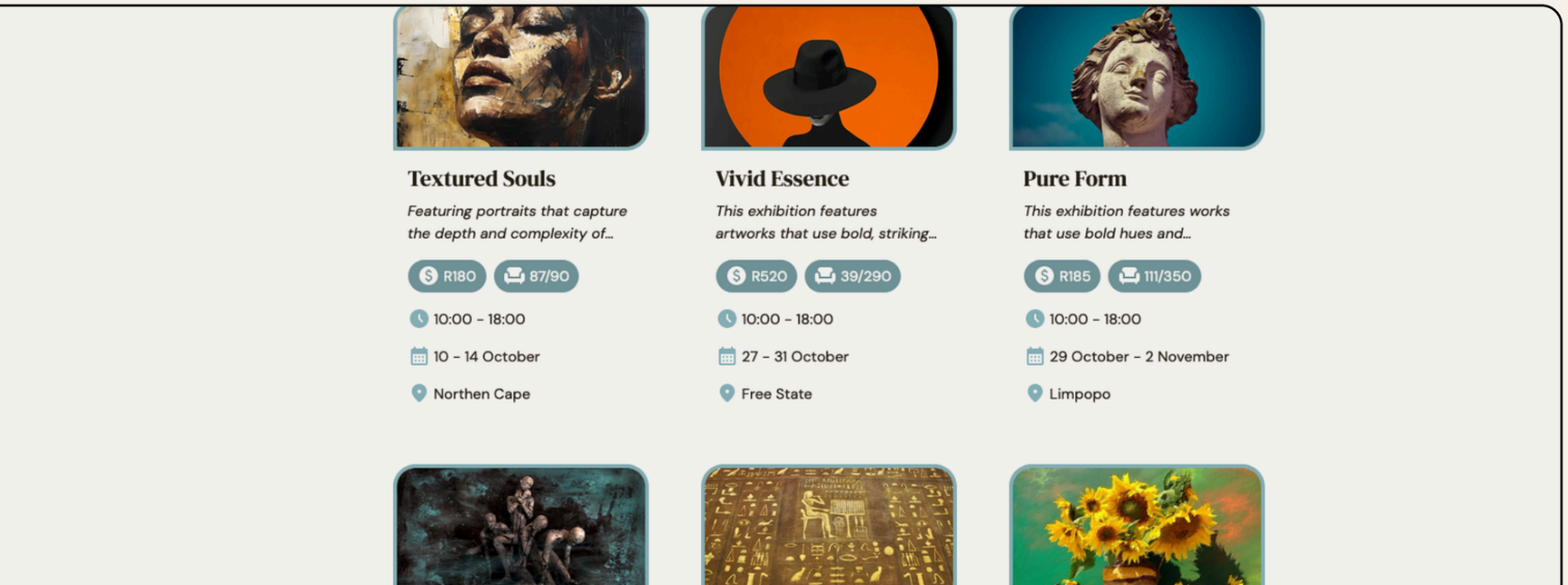
Wolf Botha

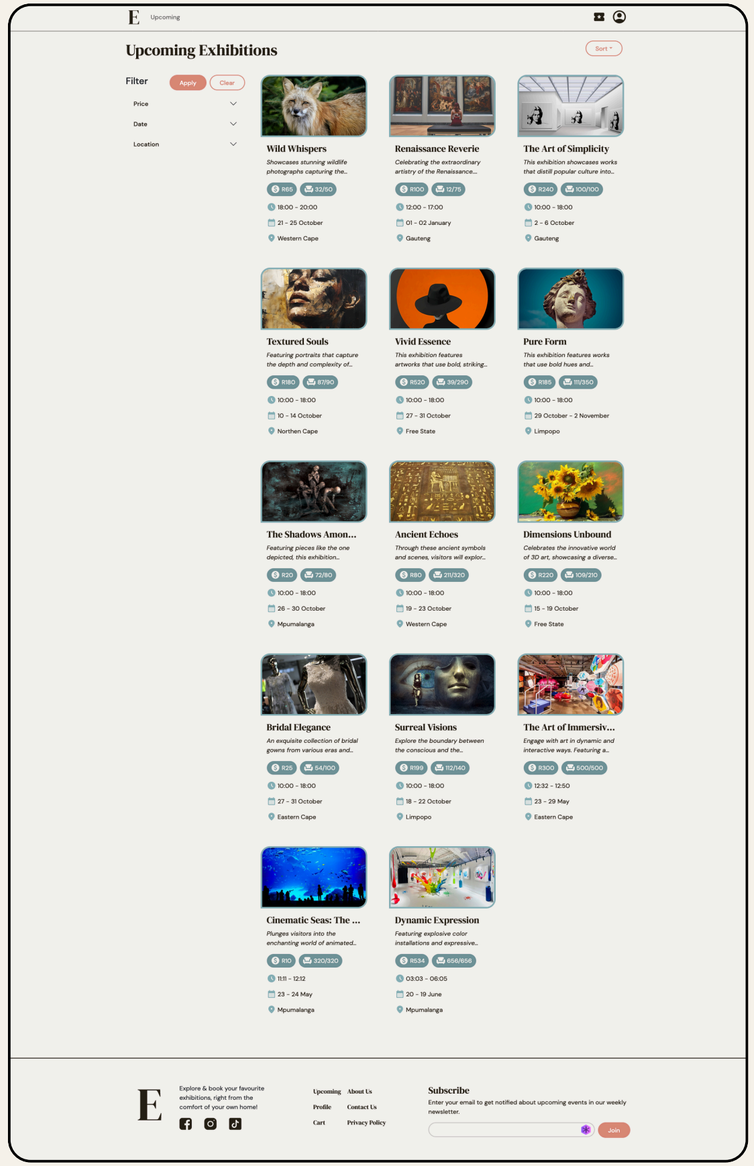
Code Explained A1

Populating a list from MongoDB: Upcoming Events

```
// STATES
const [events, setEvents] = useState([]); // Events
const [defaultAllEvents, setDefaultAllEvents] = useState([]); // Default Order of all events
```

The events are stored in a state ‘events’ that will be displayed.
The ‘defaultAllEvents’ is the original order of all events for resetting filters.





Wolf Botha

Code Explained A2

Populating a list from MongoDB: Upcoming Events

```
// On Page Load, get events data from MongoDB and set to state "events"
useEffect(() => {
  fetchApprovedEvents();
}, []);

// Get events from MongoDB, but filter to only show "Approved" events.
const fetchApprovedEvents = () => {
  getAllEvents()
    .then((data) => {
      // Filter Pending events
      const filteredEvents = data.filter((event) => event.status === "Approved");
      setEvents(filteredEvents);
      setDefaultAllEvents(filteredEvents);
    })
    .catch((error) => {
      console.log(error);
    });
};
```

When the page ‘loads’ or mounts, the **fetchApprovedEvents** function is called.

It fetches all the events from MongoDB, which are then filtered to only show “Approved” events (approved by the admin), and are then set to the state variables of ‘events and ‘defaultAllEvents’.

Wolf Botha

Code Explained A3

Populating a list from MongoDB: Upcoming Events

```
{/* Event Exhibition Cards */}
<Col xs={12} md={6} lg={9}>
  <Row>
    {/* Generate (map) All events from MongoDB to an EventCard in a column (for s
    {events.length > 0 ? (
      events.map((event) => (
        <Col xs={12} lg={6} xl={4} key={event._id}>
          <EventCard
            key={event._id}
            eventIdNum={event._id}
            thumbnail={event.thumbnail}
            title={event.title}
            desc={event.description}
            ticketPrice={event.ticketPrice}
            avSeats={event.availableSeats}
            maxSeats={event.maxSeats}
            startTime={event.startTime}
            endTime={event.endTime}
            startDate={event.startDate}
            endDate={event.endDate}
            location={event.location}
          />
        </Col>
      ))
    ) : (
      <div className="w-full flex flex-col justify-center items-center mt-12 ">
        <h4 className="font-body mb-2">No events match your filters.</h4>
        <PrimaryBtn label="Clear Filters" onClick={clearFilters} />
      </div>
```

The ‘events’ array is then mapped over to render each event using the “EventCard” component. Each property of the event object in the event array is passed to the event card (which will use the data to generate a card).

If the event array’s length is less than 0 (in other words, if there are no events due to filters being applied), a message displays stating “No events match your filters”, alongside a button to clear the filters.

Wolf Botha

Code Explained B1

```jsx
import Button from "react-bootstrap/Button";

function PrimaryBtn(props) {
  return (
    <Button
      onClick={props.onClick}
      className={`bg-scarlet-melody-BASE hover:bg-scarlet-melody-40% border-2 border-scarlet-melody-BASE rounded-full px-4 font-body ${props.className}`}
      type={props.type}
    >
      {props.label}
    </Button>
  );
}

export default PrimaryBtn;
```

Event handlers (like the expected behaviour when clicking a button)

- The component accepts **props** as an argument, allowing it to receive properties passed down from the parent component.
- **“onClick={props.onClick}”:** Assigns the click event handler passed in via the onClick prop. This means that when the button is clicked, the function provided in the onClick prop will be executed.
- **“className”**: Applies a set of CSS classes to the button for styling.
- **“type={props.type}”:** Sets the button type attribute (e.g., submit, button, etc.) based on the prop’s type.
- **“{props.label}”:** Displays the button label text provided via the label prop.

Frederick Beytell

Code Explained B2

```
const handleSubmit = async (event) => {
  event.preventDefault();
  const { email, password } = formData;

  try {
    const response = await axios.post("http://localhost:3001/users/login", {
      email,
      password,
    });
    const token = response.data.token;
    sessionStorage.setItem("token", token); // Store the token
    const decodedToken = jwtDecode(token);
    const userRole = decodedToken.userType;
    setMessage("User successfully Logged In!");

    // Redirect based on user role
    if (userRole === "admin") {
      navigate("/admin");
    } else if (userRole === "house") {
      navigate("/home");
    } else {
      navigate("/home");
    }
  } catch (error) {
    setMessage(`Invalid email or password.`);
  }
};

return (
```

User Sign-In Process

This code snippet handles the user sign-in process, sends the login credentials to the server, stores the received token, decodes it to get the user role, sets a success message, and redirects the user based on their role.

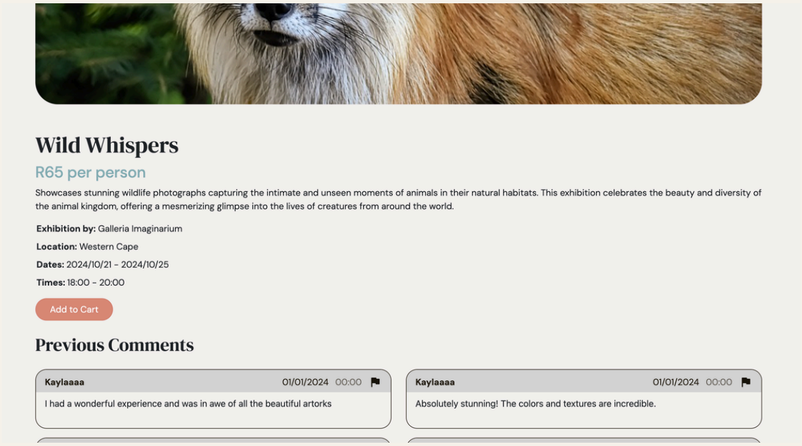
- Axios to send a **POST request** to the server at the specified URL (http://localhost:3001/users/login) with the user's email and password. **“await”** pauses the execution until the server responds.
- The response object contains the server's response. **“response.data.token”** extracts the token from the response data. **“sessionStorage.setItem("token", token)”** stores the token in the session storage, making it accessible for the duration of the session.
- **“jwtDecode(token)”** decodes the JWT token to extract its payload, which contains user information. **“const userRole = decodedToken.userType”** retrieves the userType from the decoded token, indicating the user's role (e.g., admin, house).
- **“setMessage”** is a state updater function that sets a message indicating the user has logged in successfully.
- The conditional block checks the user's role and uses navigate to redirect them to the appropriate page.
- The catch block captures any errors that occur during the login process.

Frederick Beytell

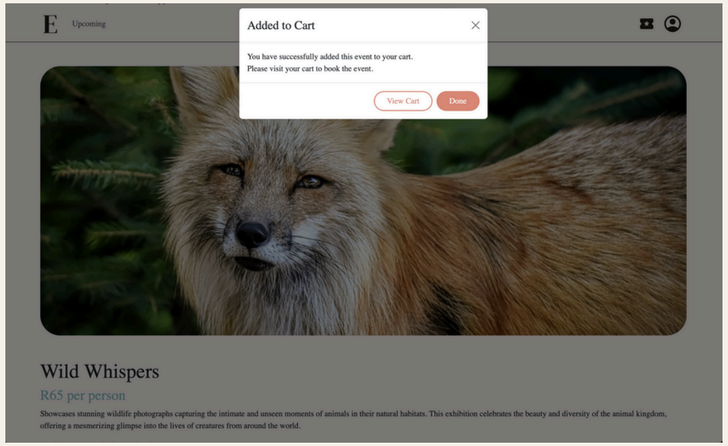
Code Explained C1

Adding an Event to your Cart

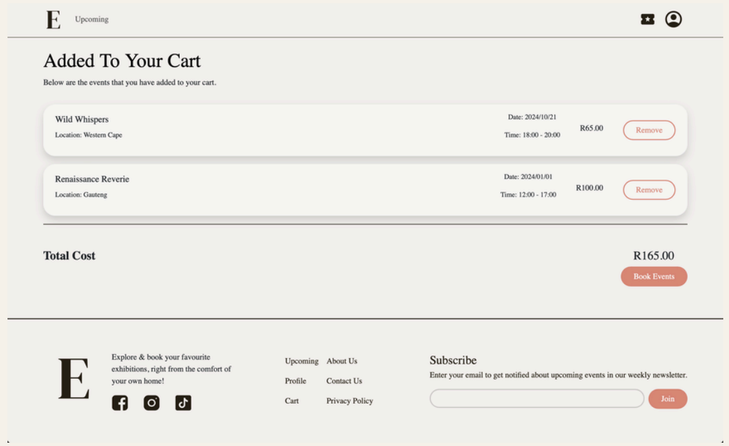
Visual Representation



Step 1: When a user wants to buy a ticket, they will click the “Add to Cart” button.



Step 2: A pop-up notification (modal) will appear that tells the user that the ticket has been added to the cart successfully. They can then continue browsing or go to the cart.



Step 3: Users can view all the tickets they have added in their cart.

Iné Smith

Code Explained C2

Adding an Event to your Cart

Code Explanation: Clicking the “Add to Cart” button will initially open a modal.

```
<Modal show={showModal} onHide={() => setShowModal(false)}>
  <Modal.Header closeButton>
    <Modal.Title className="font-display">Added to Cart</Modal.Title>
  </Modal.Header>
  <Modal.Body className="font-body">
    You have successfully added this event to your cart. <br />
    Please visit your cart to book the event.
  </Modal.Body>
  <Modal.Footer>
    <SecondaryBtn label="View Cart" onClick={handleToCart} />
    <PrimaryBtn label="Done" onClick={() => setShowModal(false)} />
  </Modal.Footer>
</Modal>
```

The modal is designed to inform users that an event has been successfully added to their cart and to prompt them to either view their cart or complete the action.

Iné Smith

Code Explained C3

Adding an Event to your Cart

Code Explanation: Added tickets shown in cart.

```
useEffect(() => {
  const fetchCartTickets = async () => {
    const token = sessionStorage.getItem("token");
    if (token) {
      const decodedToken = jwtDecode(token);
      const userId = decodedToken.userId;

      try {
        const tickets = await getTicketsByStatus(userId, "cart");
        setCartTickets(tickets);

        const eventPromises = tickets.map((ticket) => getEventById(ticket.eventId));

        const events = await Promise.all(eventPromises);
        setCartedEvents(events);

        console.log(events);
        // Calculate the total cost
        const cost = events.reduce((acc, event) => {
          // Check if eventId and ticketPrice exist
          return acc + event.ticketPrice;
        }, 0);
        setTotalCost(cost);
      } catch (error) {
        console.error("Error fetching cart tickets:", error);
      }
    }
  };
```

This piece of code demonstrates the use of the ‘useEffect’ hook and an asynchronous function, ‘fetchCartTickets’, to retrieve and manage the cart tickets for a logged-in user.

Decode Token extracts the ‘userId’, which is the user that’s logged in.

Maps over the tickets to fetch details for each event using their eventId. Underneath that, we used ‘Promise.all’ to resolve all event detail promises and sets the events in the state using ‘setCartedEvents’.

The event detail is reduced to calculate the total cost of the tickets in the cart and then sets the total cost in the state using setTotalCost.

Iné Smith

Code Explained D1

Populating the comments based on the eventId

This part of the EventInfoPage renders the CommentsCard component, passing eventId and refreshComments as props to manage the comments display and refresh functionality.

```jsx
<div className="container">
  <h2 className="font-display mt-4">Previous Comments</h2>
  <div className="row mt-2">
    <div>
      <CommentsCard eventId={eventId} refreshComments={refreshComments} />
    </div>
  </div>
</div>
```

Kayla Posthumus

Code Explained D2

Fetching and Filtering Comments

The getTheComments function fetches all comments for the given eventId, filters out flagged comments, and updates the state with the safe comments. It also fetches user data for each comment.

```
// Refresh the comments
const getTheComments = async () => {
  try {
    const response = await axios.get(`/comments?eventId=${eventId}`);

    // Filtering the safe comments
    const safeComments = response.data.filter((comment) => !comment.isFlagged);
    setFComments(safeComments);

    // Fetch user data for each comment and store it
    const userPromises = safeComments.map((comment) => fetchUserData(comment.userId));
    await Promise.all(userPromises);

    // All comments
    setComments(response.data);
  } catch (error) {
    console.error("Error fetching comments:", error);
  }
};
```

Kayla Posthumus

Code Explained D3

Fetching Comments Based on eventId

The useEffect hook in this code runs a side effect whenever eventId or refreshComments changes, as specified in the dependencies array [eventId, refreshComments], ensuring the effect re-executes when either variable updates. Within the hook, there's a conditional check if (eventId) to verify that eventId has a value before proceeding. If true, it calls the getTheComments function, which is tasked with fetching the comments associated with the current event from a data source, such as an API. This ensures that the displayed comments are always up-to-date with the latest event context.

```
// When the eventID changes, get the comments for that event
useEffect(() => {
  if (eventId) {
    getTheComments();
  }
}, [eventId, refreshComments]);
```

Kayla Posthumus

Code Explained D4

Rendering the fecthed Comments

The code snippet maps over the fComments array to dynamically generate a styled container for each comment, displaying the commenter's username (or "Deleted User" if the user is deleted), the comment's creation date and time, and the comment text. Additionally, it includes an interactive flag button that, when clicked, triggers the handleFlagComment function with the specific comment's ID, allowing users to report comments. This approach ensures that all relevant comment details are presented clearly and interactively within the UI.

```
return (
  <div className="w-full flex flex-wrap justify-between">
    {fComments.map((comment) => (
      <div key={comment._id} className="rectangle bg-canvas-white-BASE w-[49%] mt-3">
        <div className="top-section flex items-center justify-between pt-1 px-3 font-body text-ink-silho
          <div className="font-bold">{users[comment.userId]?.username || "Deleted User"}</div>
          <div className="flex items-center space-x-3">
            <div>{comment.createdDate}</div>
            <div className=" text-ink-silhouette-40%">{comment.createdTime}</div>
            {/* Flag Button Icon */}
            <div className="mx-2 cursor-pointer" onClick={() => handleFlagComment(comment._id)}>
              <svg
                xmlns="http://www.w3.org/2000/svg"
                height="24px"
                viewBox="0 -960 960 960"
                width="24px"
                className=" fill-ink-silhouette-70% hover:fill-scarlet-melody-BASE"
              >
                <path d="M280-400v240q0 17-11.5 28.5T240-120q-17 0-28.5-11.5T200-160v-600q0-17 11.5-28.5T2
              </svg>
            </div>
          </div>
        </div>
        <p className="pt-2 px-3 mb-8">{comment.text}</p>
      </div>
    ))}
  </div>
);
```

Kayla Posthumus

Kayla Posthumus

Code Explained D5

Creating a new Comment

State Initialization for New Comment: In the EventInfoPage component, a state variable newComment is initialized to store the data of the new comment being created.

```
const [newComment, setNewComment] = useState({
  eventId: eventId,
  userId: "",
  text: "",
  isFlagged: false,
  createdDate: "",
  createdTime: "",
});
```

Fetching User Data and Setting State: When the page loads and the eventId changes, the user data is fetched and stored in the user state variable.

```
  // Fetch the current user data
  const token = sessionStorage.getItem("token");
  if (token) {
    const decodedToken = jwtDecode(token);
    getUserById(decodedToken.userId)
      .then((userData) => {
        setUser(userData);
        if (userData.userType === "house") {
          setIsHouse(true);
        }
      })
      .catch((error) => {
        console.error("Error fetching user data:", error);
      });
  }
}, [eventId]);
```

Code Explained D6

Creating a new Comment

Creating a New Comment Function: The createComment function constructs the new comment data and sends it to the backend using the addNewComment service function.

```
// Creating a new comment
function createComment() {
  // If logged in user does not exist, stop the comment-creation process
  if (!user) return;

  const commentData = {
    eventId: eventId,
    userId: user._id,
    username: user.username,
    text: newComment.text,
    isFlagged: false,
    createdDate: getCurrentDate(),
    createdTime: getCurrentTime(),
  };

  addNewComment(commentData).then(() => {
    setRefreshComments((prev) => !prev); // Toggle between false & true
  });
}
```

```
<div className="container mt-3">
  <h2 className="font-display">Leave a Review</h2>
  <NewComment
    onPostClick={createComment}
    newComment={newComment}
    setNewComment={setNewComment}
  />
</div>
```

Passing Props to NewComment Component: The NewComment component is used for rendering the UI for adding a new comment. The createComment function, newComment state, and setNewComment setter function are passed as props.

Kayla Posthumus

Code Explained D7

Creating a new Comment

Handling the New Comment Form
The NewComment component handles the form input and updates the newComment state as the user types in their comment. It calls createComment when the "Post Review" button is clicked.

```
function NewComment({ onPostClick, newComment, setNewComment }) {
  const handleChange = (e) => {
    setNewComment({
      ...newComment,
      [e.target.id]: e.target.value,
    });
  };

  const handlePostReviewBtnClick = (e) => {
    e.preventDefault();
    onPostClick();
  };
```

```
  addNewComment(commentData).then(() => {
    setRefreshComments((prev) => !prev); // Toggle between false & true
  });
}
```

Refreshing Comments: After the new comment is added successfully, the refreshComments state is toggled to trigger the re-fetching of comments in the CommentsCard component.

Kayla Posthumus

Live Demonstration

We will now do a step-by-step walkthrough of the website, highlighting all the features & functionalities.

Any Questions?

Thank you

OPEN WINDOW

Kayla Posthumus
231096

Iné Smith
221076

Frederick Beytell
231374

Wolf Botha
21100255