Andre van Heerden

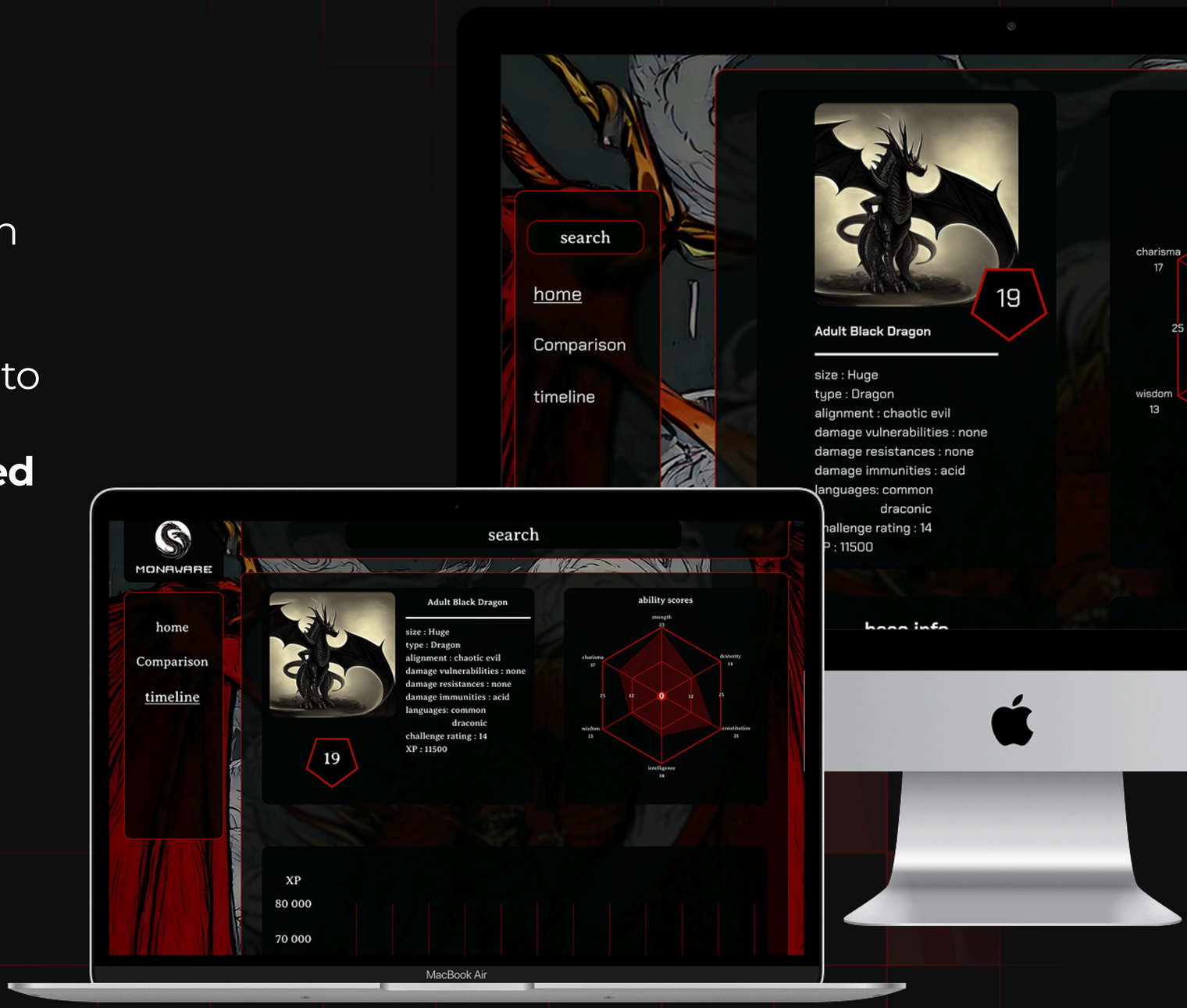
DV200 Portfolio presenation

MONAWARE

A DnD monster Data

What was the Task For the term

The goal of this project was to design and build a **React web application** that visualizes data from an external **API** using **Chart.js**. The app needed to include asynchronous data retrieval with **Axios**, a clear **component-based UI**, and **functional navigation** between multiple pages.



Key requirements where:

Dashboard Landing Page

A **Landing Page** showcasing an overview of the dataset with contextual information and a custom-designed UI element.

Comparison Interface

A **Comparison Page** displaying two data objects using a **Bar**, **Pie**, and **Radar** or **Polar Area chart**.

Timeline Visualization (Optional)

An optional **Timeline Page** with a **Line chart** showing trends across multiple properties.

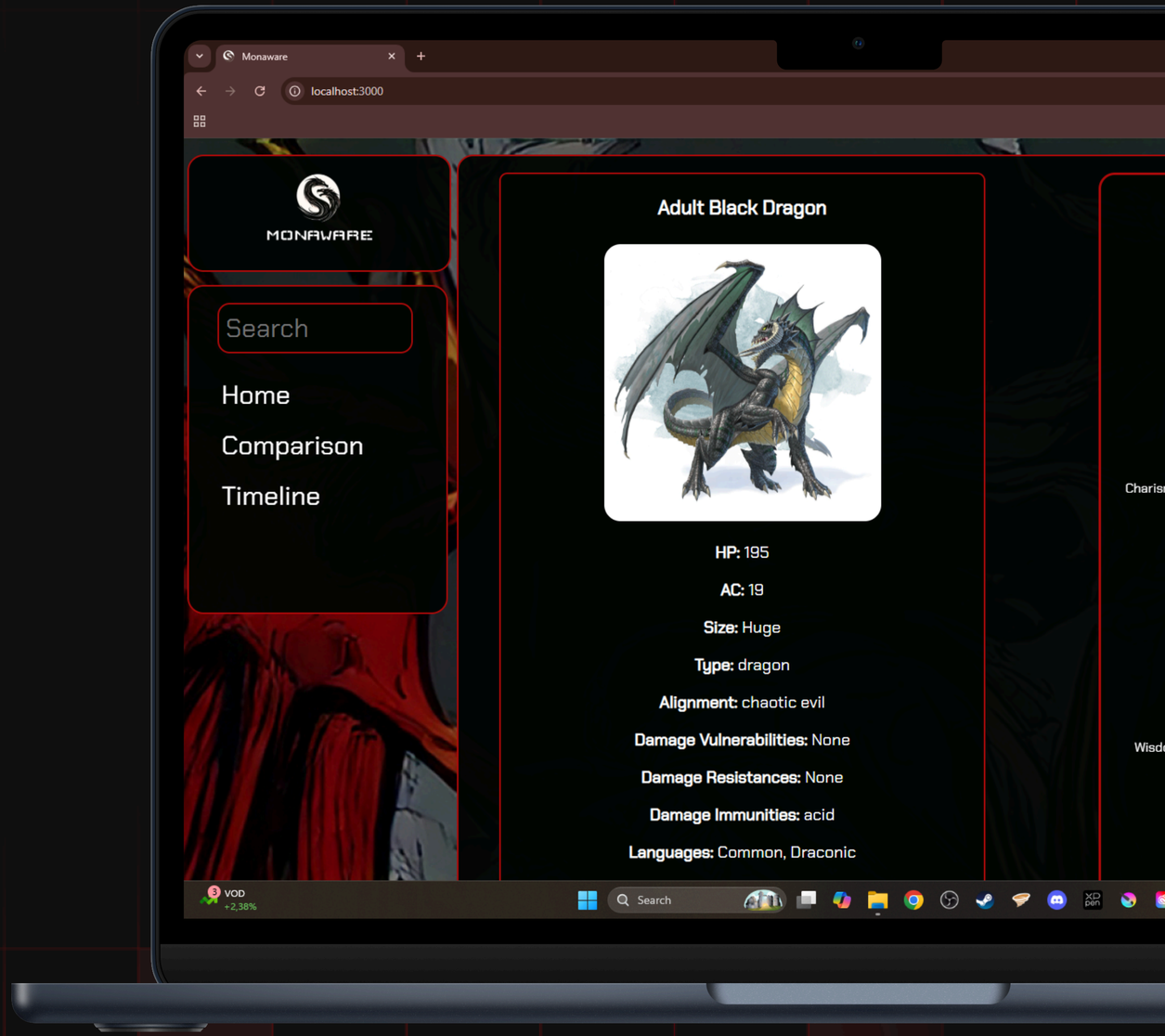
Component-Based Navigation

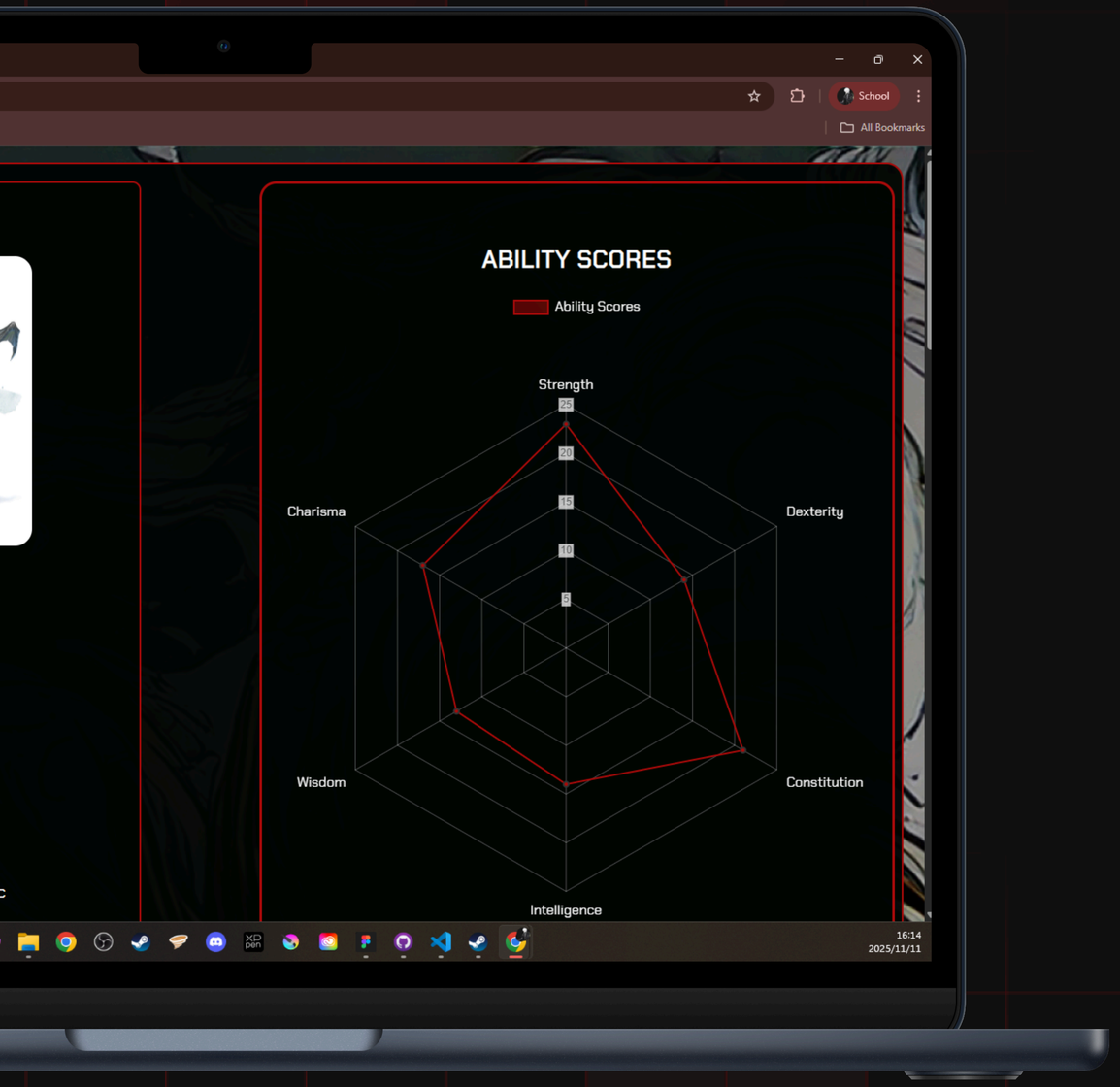
Implementation of **routes, reusable components**, and a **visually** appealing dashboard layout.

What is Monaware

Monaware is a web app built using the D&D 5e SRD API to explore detailed monster data from Dungeons & Dragons 5th Edition.

- Focuses on the monsters database with stats like abilities, skills, HP, AC, and attacks.
- Allows users to compare two monsters side by side for easier analysis.
- Enhances visuals by sourcing monster images from a D&D 5e SRD Discord archive.



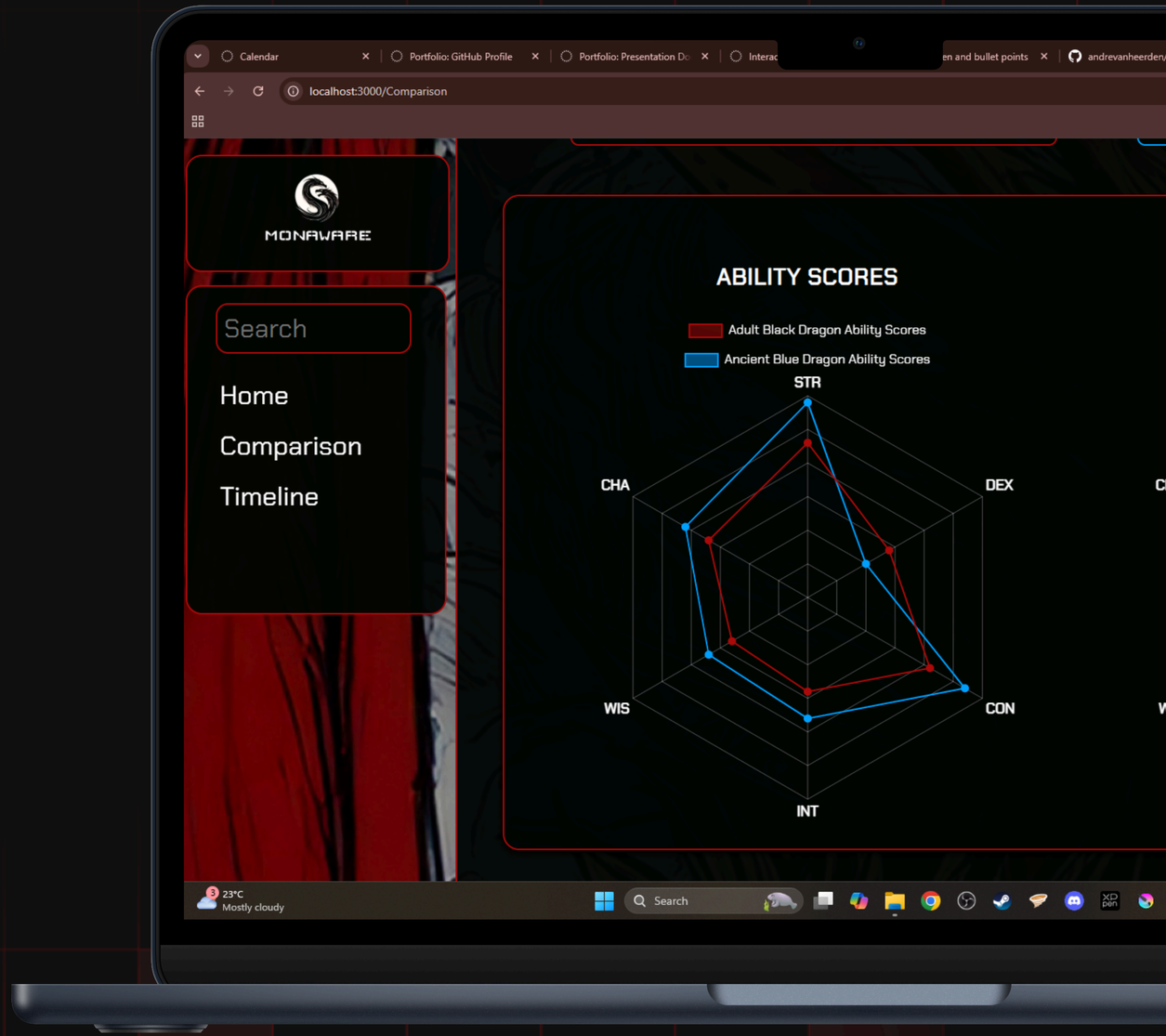


What problems does Monaware solve

- **Quick Access to Monster Stats** – DMs and players can instantly view all relevant monster information like abilities, skills, hit points, armor class, and attacks, without flipping through multiple rulebooks.
- **Side-by-Side Monster Comparison** – Easily compare two monsters at once to analyze differences in stats, skills, movement, and challenge rating. Ideal for planning encounters or balancing gameplay.
- **Improved Gameplay Efficiency** – Overall, Monaware saves time and reduces manual calculations, allowing DMs and players to focus more on gameplay and storytelling.

What is D&D 5e SRD API

The **D&D 5e SRD API** provides structured data from the Dungeons & Dragons 5th Edition System Reference Document (SRD). It includes information on monsters, spells, equipment, classes, and abilities, allowing developers to access official D&D content programmatically for apps, tools, or websites.



visit the api : https://www.dnd5eapi.co/

Live demo and important code overview

API Fetching with Axios

Here’s where I fetch monster data from the D&D 5e API using Axios. It’s an asynchronous call that retrieves a monster’s stats when the user searches or selects one.

I use React state to store the data, and error handling ensures users get feedback if a monster isn’t found.

This same logic repeats for the second monster, so both sides of the comparison update instantly when users choose new creatures.

```
const fetchDataset1 = async (monsterIndex) => {
  if (!monsterIndex) {
    setError1("Please enter a valid monster name.");
    return;
  }
  setLoading1(true);
  setError1(null);
  try {
    const response = await axios.get(`https://www.dnd5eapi.co/api/monsters/${monsterIndex}`);
    setCardDetails1([response.data]);
    onDataset1Change(monsterIndex);

    // Set and preload the image
    const url = getMonsterImage(monsterIndex);
    setImageUrl1(url);
    setImageLoading1(true);
    setImageError1(false);

    const img = new Image();
    img.src = url;
    img.onload = () => setImageLoading1(false);
    img.onerror = () => {
      console.warn(`Failed to preload image for monster: ${response.data.name}`);
      setImageError1(true);
      setImageLoading1(false);
    };
  } catch (err) {
    console.error("Error fetching dataset 1:", err);
    setError1("Monster not found. Please try again.");
  } finally {
    setLoading1(false);
  }
};
```

Fixing the API’s Images

Since the official D&D API doesn’t have 50% of the monster images, I added my own custom image dataset that I found from a community source on Discord.

The code you see here “the getMonsterImage() function” takes the monster name from the API and compares it to my local dataset.

It cleans up the name by removing dashes and spaces, then tries several matching methods to find the right image.

If it finds a match, it displays that monster’s image and if not, it shows a default placeholder image instead.

This approach allowed me to give each monster a proper visual without relying on the limited API.

```
const getMonsterImage = (monsterIndex) => {
  if (!monsterIndex) return noIMG;

  // Clean up the monster index for matching
  const cleanIndex = monsterIndex.toLowerCase().replace(/-/g, ' ').trim();

  // Try different matching strategies
  const matches = monsterImages.sample.filter(entry => {
    const cleanDescription = entry.description.toLowerCase().replace(/-/g, ' ').trim();
    return (
      cleanDescription === cleanIndex || // exact match
      cleanDescription.includes(cleanIndex) || // description contains our index
      cleanIndex.includes(cleanDescription) || // our index contains description
      cleanDescription.replace(/\s+/g, '') === cleanIndex.replace(/\s+/g, '')
    );
  });

  // Return the first match if found, otherwise use noIMG
  return matches.length > 0 ? matches[0].imageUrl : noIMG;
};
```

Chart.js Implementation

Here is the Radar chart setup using Chart.js.
I take the API data for each monster and map their ability scores into the chart.

Each monster gets its own dataset with unique colors and styling, so users can easily compare stats side by side.

This dynamic setup updates automatically whenever a new monster is selected, making comparisons simple and visual.

```
// Ability Scores Data
const abilityData = {
  labels: ['STR', 'DEX', 'CON', 'INT', 'WIS', 'CHA'],
  datasets: [
    {
      label: `${monster1.name} Ability Scores`,
      data: [
        monster1.strength,
        monster1.dexterity,
        monster1.constitution,
        monster1.intelligence,
        monster1.wisdom,
        monster1.charisma,
      ],
      borderColor: 'rgba(171, 14, 11, 1)',
      borderWidth: 2,
      pointBackgroundColor: 'rgba(171, 14, 11, 1)',
      fill: true,
      backgroundColor: 'rgba(171, 14, 11, 0.5)',
    },
    {
      label: `${monster2.name} Ability Scores`,
      data: [
        monster2.strength,
        monster2.dexterity,
        monster2.constitution,
        monster2.intelligence,
        monster2.wisdom,
        monster2.charisma,
      ],
      borderColor: 'rgb(0, 162, 255, 1)',
      borderWidth: 2,
      pointBackgroundColor: 'rgb(0, 162, 255, 1)',
      fill: true,
      backgroundColor: 'rgba(0, 162, 255, 0.5)',
    },
  ],
};
```

Routing and Navigation

I used React Router to manage navigation between the Home, Comparison, and Timeline pages.

The Routes component maps URLs to components, while the Navbar provides easy navigation.

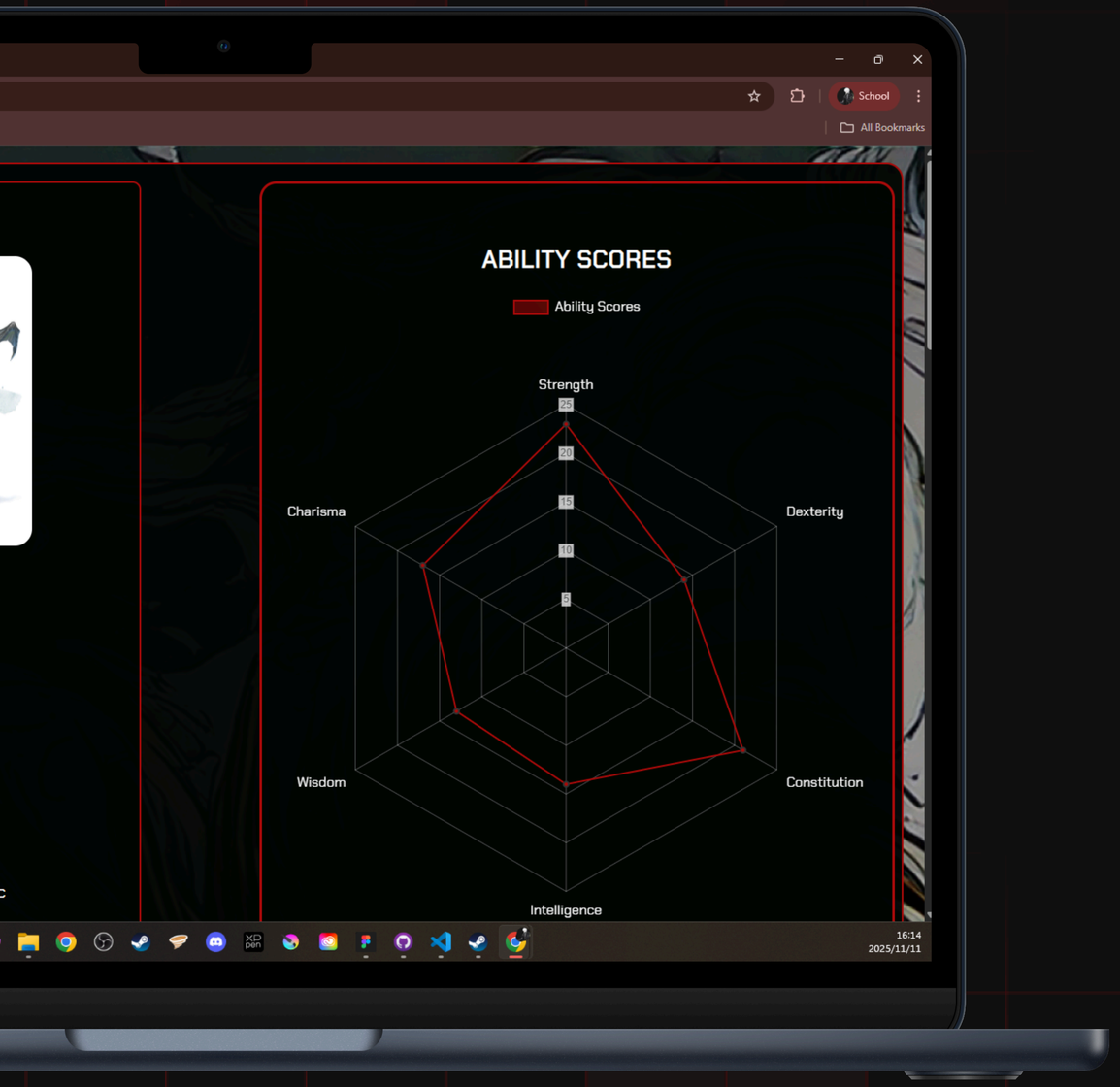
This setup makes the app feel structured and seamless for users.

```jsx
import React, { useState } from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Navbar from "./Components/Navbar";
import Home from "./Components/Home";
import Timeline from "./Components/Timeline";
import Comparison from "./Components/Comparison";
import "@fontsource/chakra-petch"; // Defaults to weight 400
import "@fontsource/chakra-petch/400.css"; // Specify weight
import "@fontsource/chakra-petch/400-italic.css"; // Specify weight and style

function App() {
  const [selectedMonster, setSelectedMonster] = useState("adult-black-dragon");

  return (
    <Router>
      <div className="App">
        <Navbar onMonsterSelect={setSelectedMonster} /> {/* Pass the state updater */}
        <Routes>
          <Route path="/" element={<Home selectedMonster={selectedMonster} />} />
          <Route path="/Comparison" element={<Comparison />} />
          <Route path="/Timeline" element={<Timeline selectedMonster={selectedMonster} />} /> {/* Pass
        </Routes>
      </div>
    </Router>
  );
}

export default App;
```



Challenges

Handling Missing API Images:

- About 70% of the images from the API were unavailable.
- To solve this, I visited the API's Discord community and found that another user had compiled a list of monsters with corresponding image URLs.
- I converted this list into a React import file, allowing the images to display correctly in the app's cards.

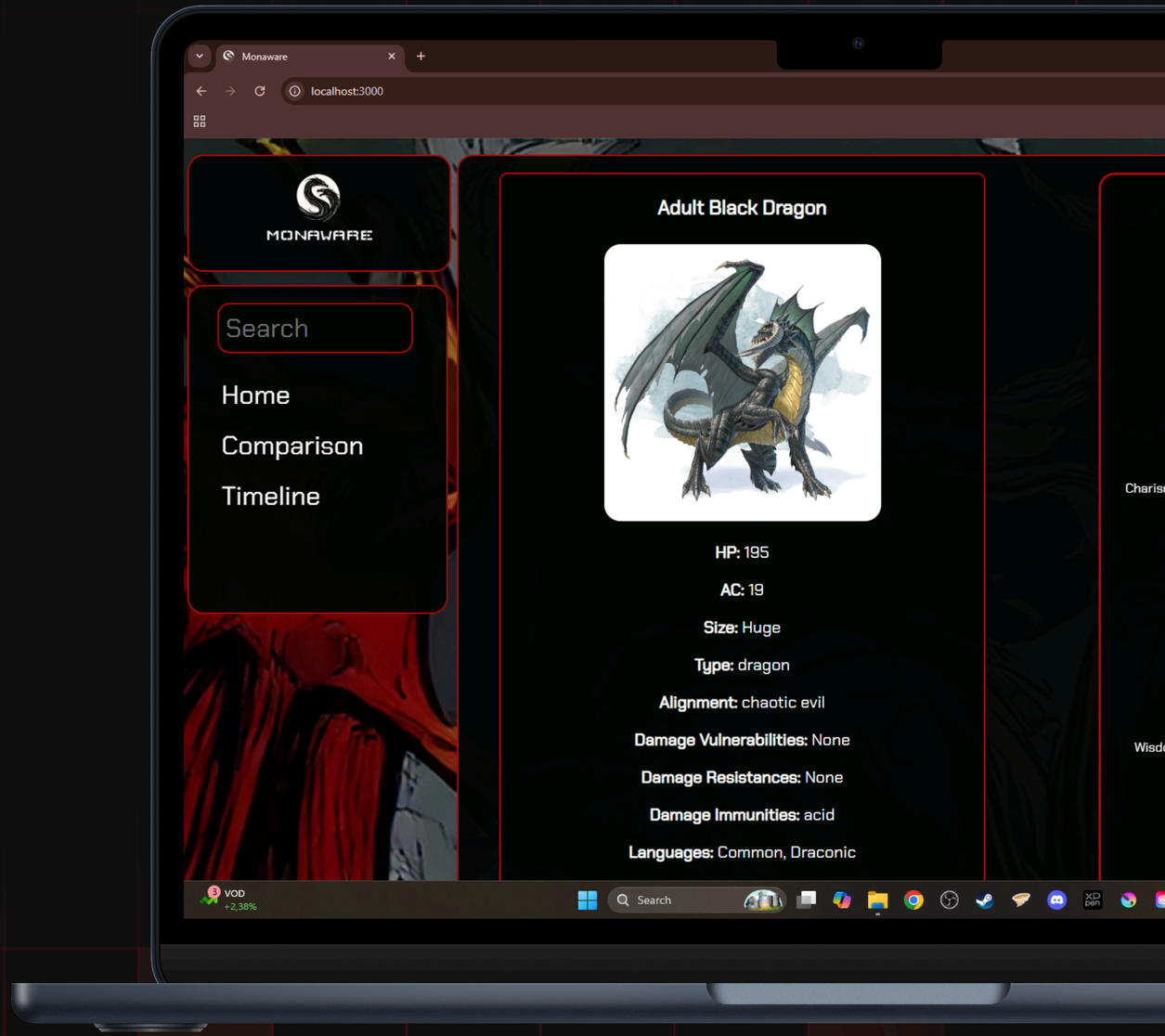
Storing Timeline Bar Chart Data:

- The timeline bar chart required persistent data storage.
- I created a storage.js file to save this data locally in JSON format, ensuring the chart could render accurately and maintain state across sessions.

Conclusion

This project was my first experience working with Node.js and React, and I found it extremely enjoyable. I loved the freedom React provides for building dynamic, component-based interfaces, and I particularly enjoyed working with charts and data visualization to display complex information.

Completing this full API project on my own was a valuable learning experience, and I appreciated the creative freedom in designing the UI and UX. I look forward to future projects where I can build on these skills, refine my workflow, and continue exploring the possibilities of React and Node.js.

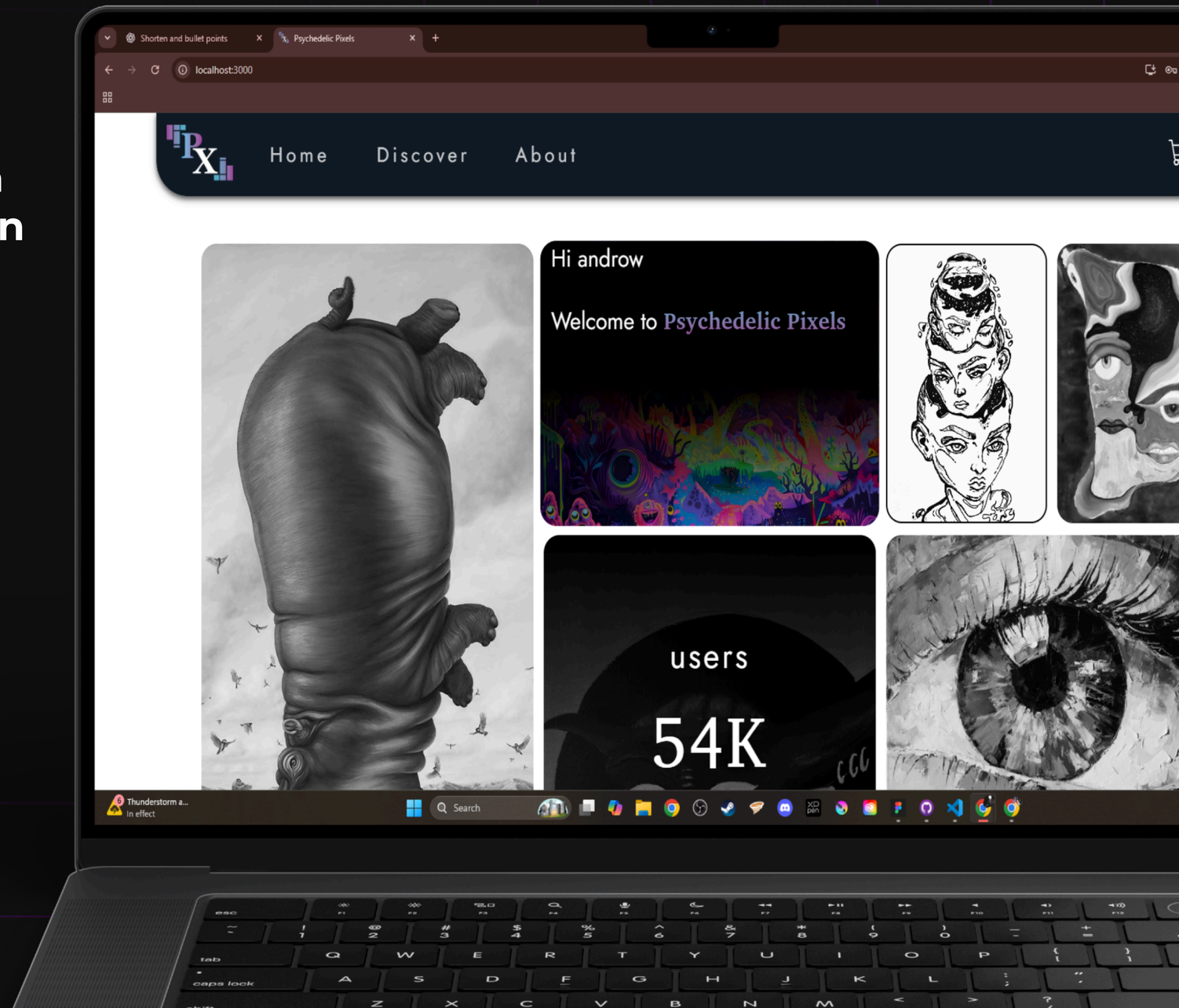


Psychedelic Pixels

online art store

What was the Task For the term

The goal of this project was to design and develop a **React web application** with a strong **brand identity** and functional **user authentication system**. The app needed to demonstrate both design and technical skills while implementing additional features beyond the standard brief.



Key requirements where:

Visual Identity & Branding

Creating a theme and brand identity with logos, icons, and cohesive UI styling.

User Interaction & Engagement

Implementing functionality for entering, commenting, and approving user submissions based on the app’s theme.

Secure Authentication System

Adding authentication with registration, login, and possible role-based access for admin approval.

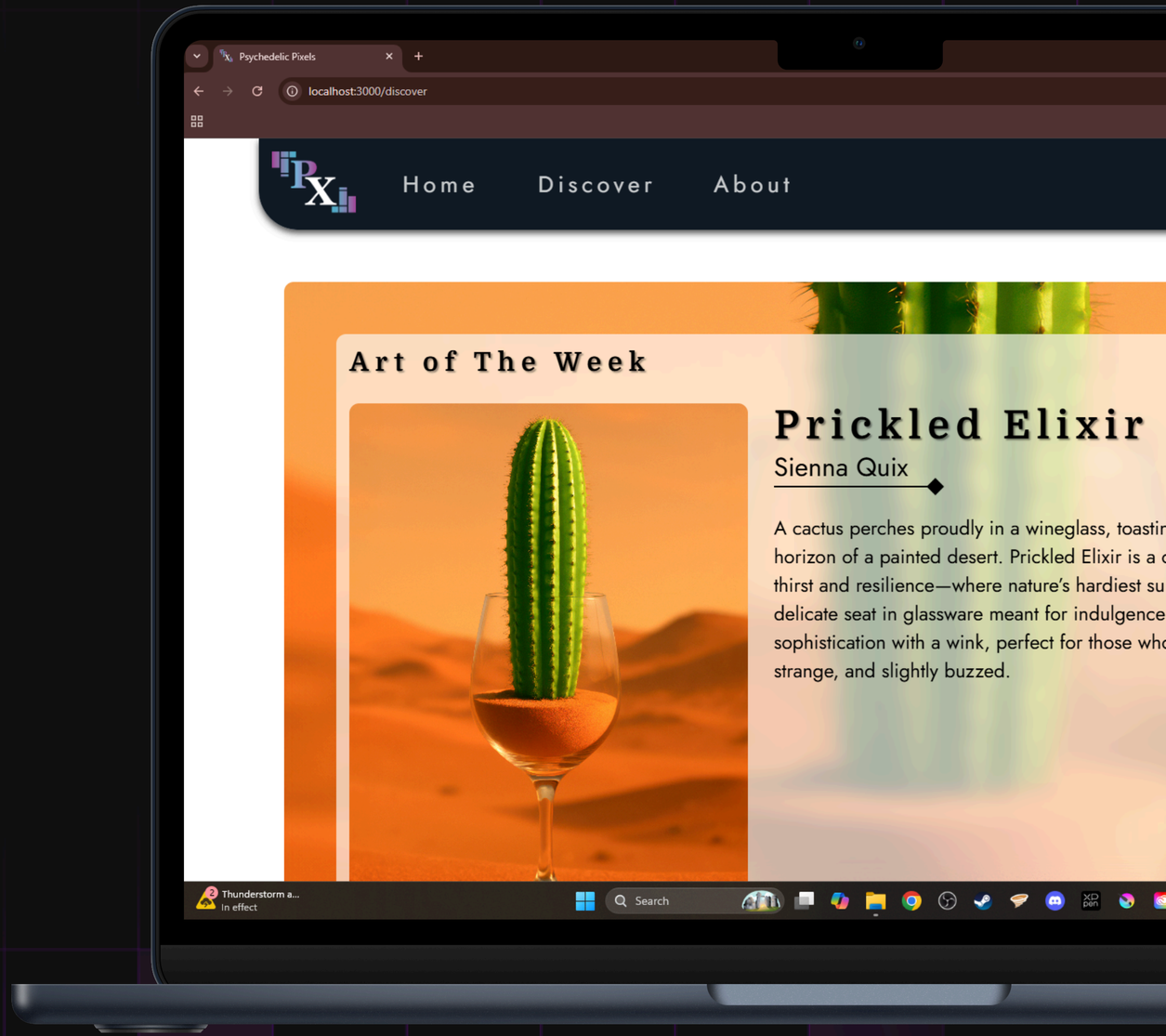
Innovative Feature Expansion

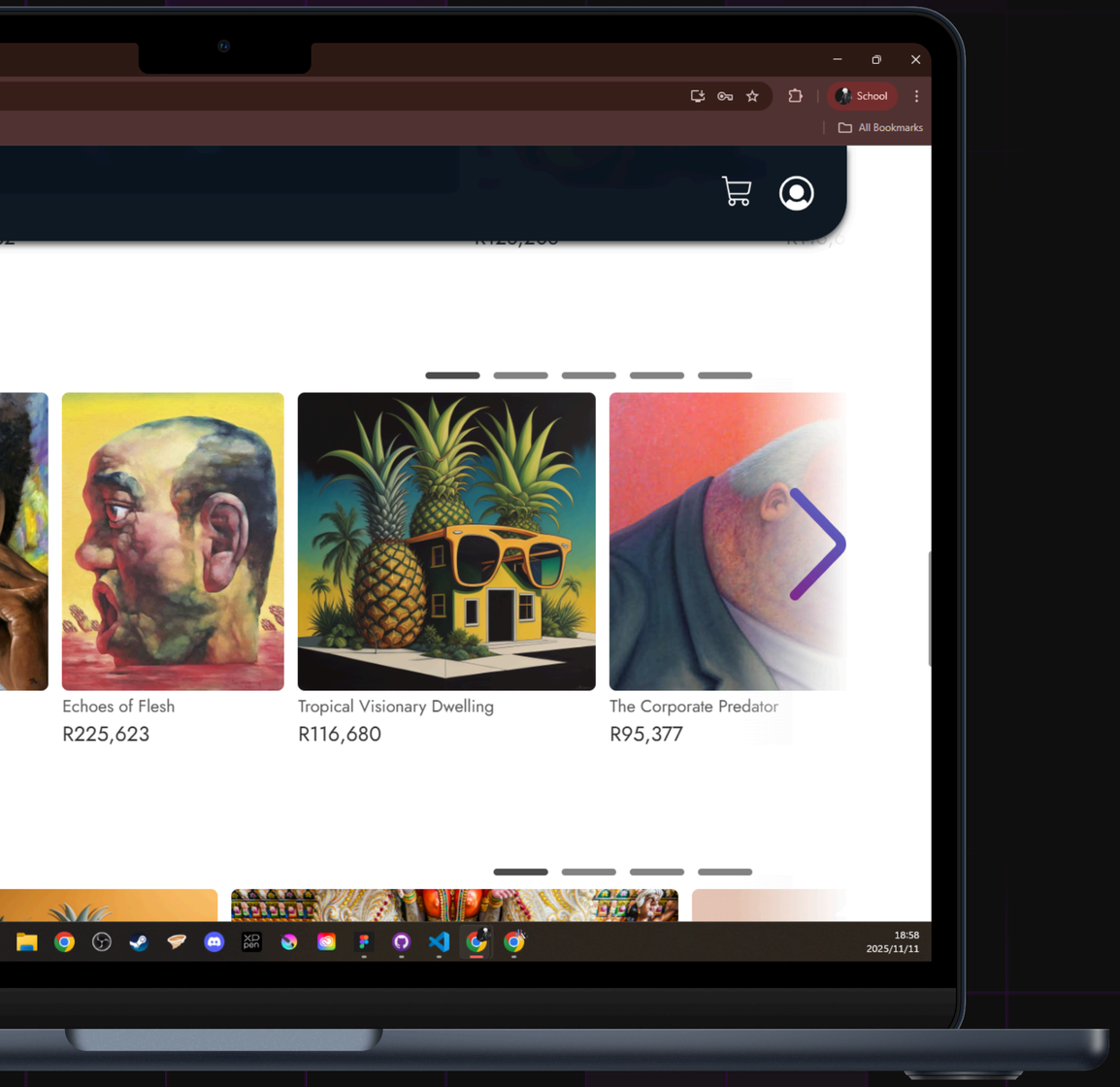
Developing at least one unique feature outside the lecture content (e.g., filtering, notifications, or progress indicators).

What is Psychedelic Pixels

Psychedelic Pixels is an e-commerce platform that celebrates surreal and imaginative artwork.

- Showcases unique paintings, sculptures, and creative pieces.
- Focuses on bold, unconventional, and thought-provoking art.
- Connects eccentric artists with curious collectors and art lovers.
- Encourages creativity beyond traditional galleries.





What problems does Psychedelic Pixels solve

- **Accessible Platform for Artists –** Both emerging and established artists can easily showcase and sell their work, reaching collectors without relying on traditional galleries.
- **Direct Artist-to-Collector Connection –** Buyers can explore unique, imaginative artwork and interact with the creators directly, encouraging engagement and support for unconventional art.
- **Creative Freedom & Quality Control –** The platform allows artists to post freely while admin moderation ensures submissions maintain quality, creating a trustworthy marketplace.

What did each team member do

Andre

- Log in and sign up page
- Authentication page
- user data security
- role Assignment
- Admin checker
- user profile page
- about page
- piano for auth

Gaby

- checkout pages
- admin pages
- toast popups
- add new artwork
- inventory page

KB

- home page
- discover page
- single view page
- carousels
- ui hover effects
- footer

Live demo and important code overview

Security and Password Management

This part of the backend handles secure registration and login.

When a user signs up, their password is **hashed using bcrypt** before being stored in MongoDB — so the database never saves plain text passwords.

Then, during login, bcrypt's **compare()** checks the entered password against the stored hash.

This makes brute-force attacks or leaks far less dangerous because actual passwords can't be retrieved. It's a simple but essential security layer in any authentication system.

```
// --- Register User ---
router.post('/register', async (req, res) => {
  const { username, email, password, role } = req.body;

  // 1. Hash the password before saving
  const saltRounds = 10;
  const hashedPassword = await bcrypt.hash(password, saltRounds);

  // 2. Save new user with hashed password
  const newUser = new User({ username, email, password: hashedPassword, role });
  await newUser.save();

  res.status(201).json({ message: "User registered securely" });
});

// --- Login Check ---
router.post('/check', async (req, res) => {
  const { username, password } = req.body;

  // 1. Find user
  const user = await User.findOne({ username });
  if (!user) return res.status(401).json({ message: "User not found" });

  // 2. Verify password using bcrypt
  const passwordMatch = await bcrypt.compare(password, user.password);
  if (!passwordMatch) return res.status(401).json({ message: "Incorrect password" });

  res.status(200).json({ message: "Login successful" });
});
```

User Authentication & Authorization

Here, I implemented full authentication and authorization using **bcrypt, JWT, and role-based access control**.

When users sign up, their passwords are **hashed securely** before being stored in MongoDB.
Each user receives a **JWT token** after login, which verifies their identity and role when accessing protected routes.

If a route is marked as admin-only, my **requireAdmin** middleware checks the JWT — only users with the **admin role** can continue.

On the front end, users log in through my custom interface, then complete a **second authentication step**: playing a 7**-note piano sequence** sent to their email.
This adds a creative 2FA layer, blending security and music in a unique way.

Together, these parts make a complete system for secure access, role separation, and creative verification.

```
// --- User Registration ---
router.post('/register', async (req, res) => {
  const { username, email, password, role, adminToken } = req.body;

  // Assign admin role only with valid invite token
  let userRole = "customer";
  if (role === "admin" && adminToken === process.env.ADMIN_INVITE_TOKEN) {
    userRole = "admin";
  }

  const hashedPassword = await bcrypt.hash(password, 10);
  const newUser = new User({ username, email, password: hashedPassword, role: userRole });
  await newUser.save();

  res.status(201).json({ message: "User registered successfully" });
});

// --- User Login ---
router.post('/login', async (req, res) => {
  const { username, password } = req.body;
  const user = await User.findOne({ username });
  if (!user) return res.status(401).json({ message: "User not found" });

  const passwordMatch = await bcrypt.compare(password, user.password);
  if (!passwordMatch) return res.status(401).json({ message: "Incorrect password" });

  // JWT gives each user a signed access token
  const token = jwt.sign(
    { id: user._id, username: user.username, role: user.role },
    process.env.JWT_SECRET,
    { expiresIn: '1d' }
  );

  res.json({ token, user });
});

// --- Admin-Only Route Example ---
router.get('/admin/some-data', authenticateJWT, requireAdmin, (req, res) => {
  res.json({ secret: "Admin-only data" });
});
```

Admin & Role Management

This part of my app handles user permissions and access control.

When a user logs in, their role ‘admin’ or ‘Customer’ is stored in session storage.
The navbar checks that role and only displays admin-exclusive buttons if the user is verified as an admin.

This ensures that sensitive pages, such as product management or comment moderation, can’t be accessed by normal users.

I used conditional rendering and session-based user data to manage this securely

```
const [user, setUser] = useState(
  JSON.parse(sessionStorage.getItem("user"))
);

return (
  <>
    <button onClick={() => navigate("/profile")}>My Profile</button>
    {user && user.role === "admin" && (
      <>
        <button onClick={() => navigate("/adminForm")}>Product Form</button>
        <button onClick={() => navigate("/stocklist")}>Stocklist</button>
        <button onClick={() => navigate("/comments")}>Flagged Comments</button>
      </>
    )}
  </>
);
```

User Profile Editing

This part of my apps code handles profile editing.

When a user submits changes, the backend receives a **PUT** request with the new username or profile picture.

It updates the MongoDB record using **findOneAndUpdate** and returns the updated user data.

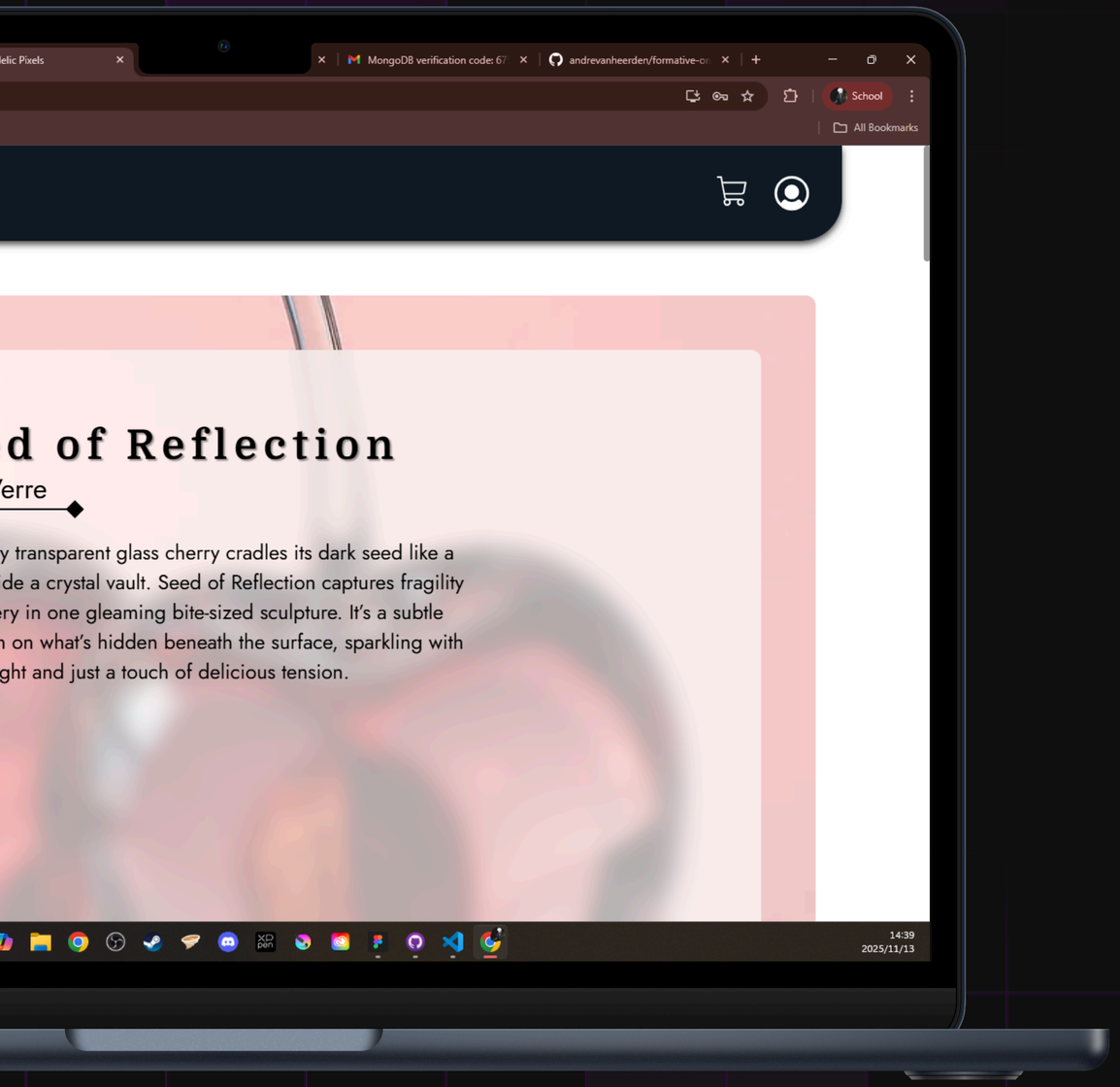
This ensures that users can securely edit their own profile info, and the app immediately reflects those changes.

```
// Route to update current user info
router.put('/me', async (req, res) => {
  const username = req.query.username || req.body.username;
  if (!username) return res.status(400).json({ message: "Username required" });

  try {
    const update = {};
    if (req.body.newUsername) update.username = req.body.newUsername;
    if (req.body.email) update.email = req.body.email;
    if (req.body.profilePic) update.profilePic = req.body.profilePic;

    const user = await User.findOneAndUpdate({ username }, update, { new: true });
    if (!user) return res.status(404).json({ message: "User not found" });

    res.json(user);
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```



Challenges

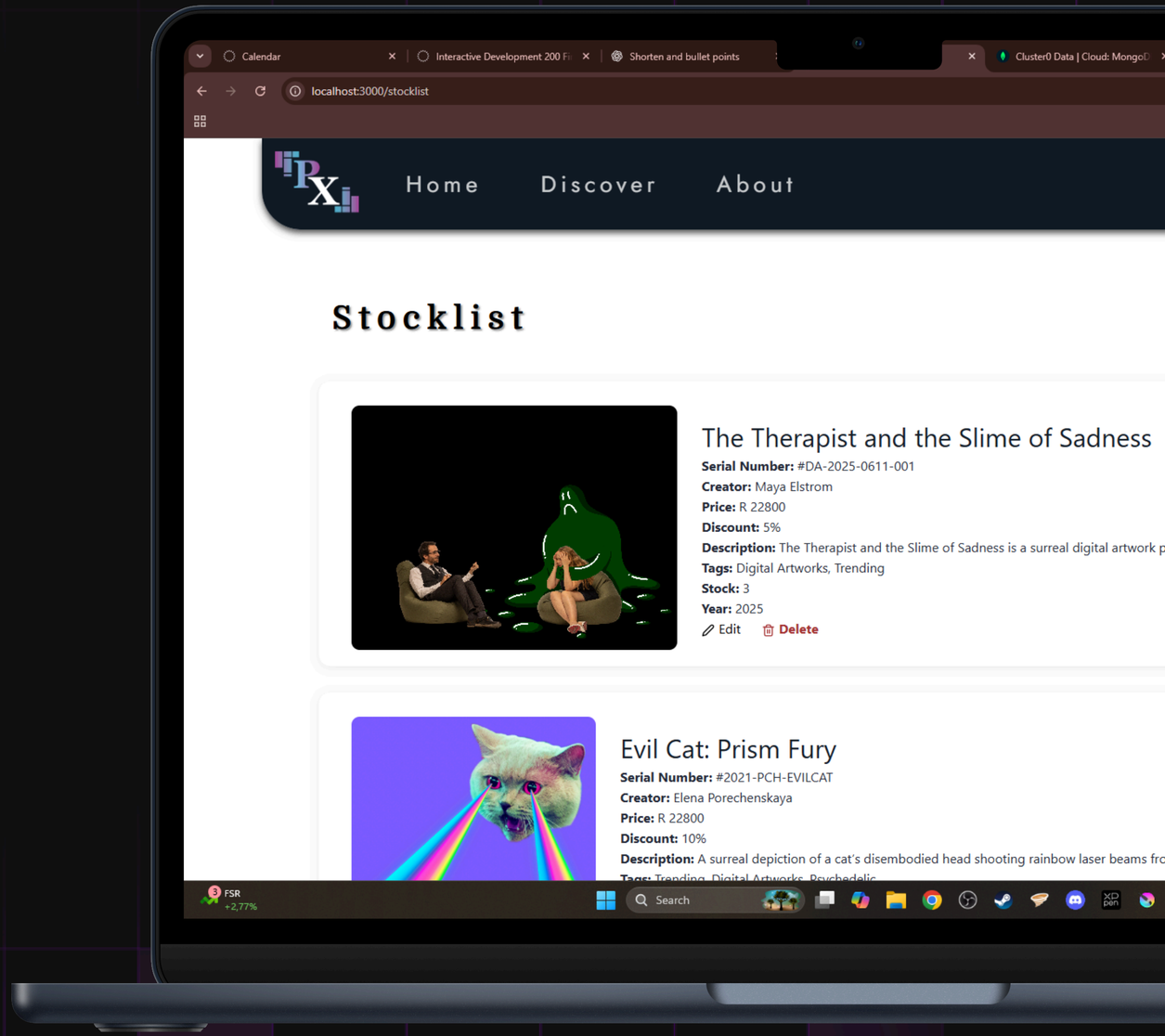
Designing a Unique Authentication Flow – Creating an authentication system that was both secure and visually unique was a major challenge. I wanted it to send a code like traditional auth systems but still feel original. After some brainstorming, I decided to integrate a piano-style input system inspired by online tutorials, which allowed users to enter codes creatively.

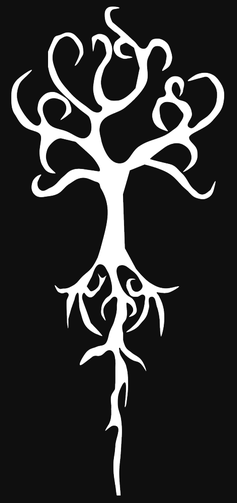
Sending Verification Notes via Email – Implementing the email-based code delivery was tricky. After researching, I discovered that the Google account used must have two-factor authentication enabled to send emails successfully, which resolved the issue.

Conclusion

Working on this project with my group members was an excellent experience. Our team collaborated really well. Gaby and KB are both great coders, and we balanced each other's strengths and weaknesses perfectly.

This project was a valuable learning experience that helped me grow not only as a developer but also in understanding how to approach coding as part of a team. I truly enjoyed the process, learned a lot about teamwork and project coordination, and would gladly work with this group again in the future.





YGGDRASIL

DnD campaign manager

What was the Task For the term

The goal of this project was to design, build, and deploy a full-stack web application using **React** for the frontend, **Node.js** and **Express** for the backend, and **MySQL** as the database. The application needed to demonstrate strong technical and design skills while implementing a complete **CRUD system, user authentication, and SEO optimization** to simulate a real-world production environment.

Key requirements where:

Theme & Branding

Create a unique visual identity with cohesive UI design, logo, icons, and appropriate imagery that reflect the app’s concept.

Authentication System

Add secure authentication with registration and login, supporting role-based access and optional multi-factor verification for creativity.

Feature Expansion

Go beyond basic requirements by adding innovative or advanced functionality that enhances user interaction or application depth.

CRUD & Database Functionality

CRUD & Database Functionality
Implement full CRUD operations with a normalized MySQL database (3NF), clear ERDs, and secure prepared statements using advanced SQL queries.

SEO & Optimisation

Integrate SEO best practices, meta tags, analytics, and performance optimization to ensure visibility and smooth user experience.

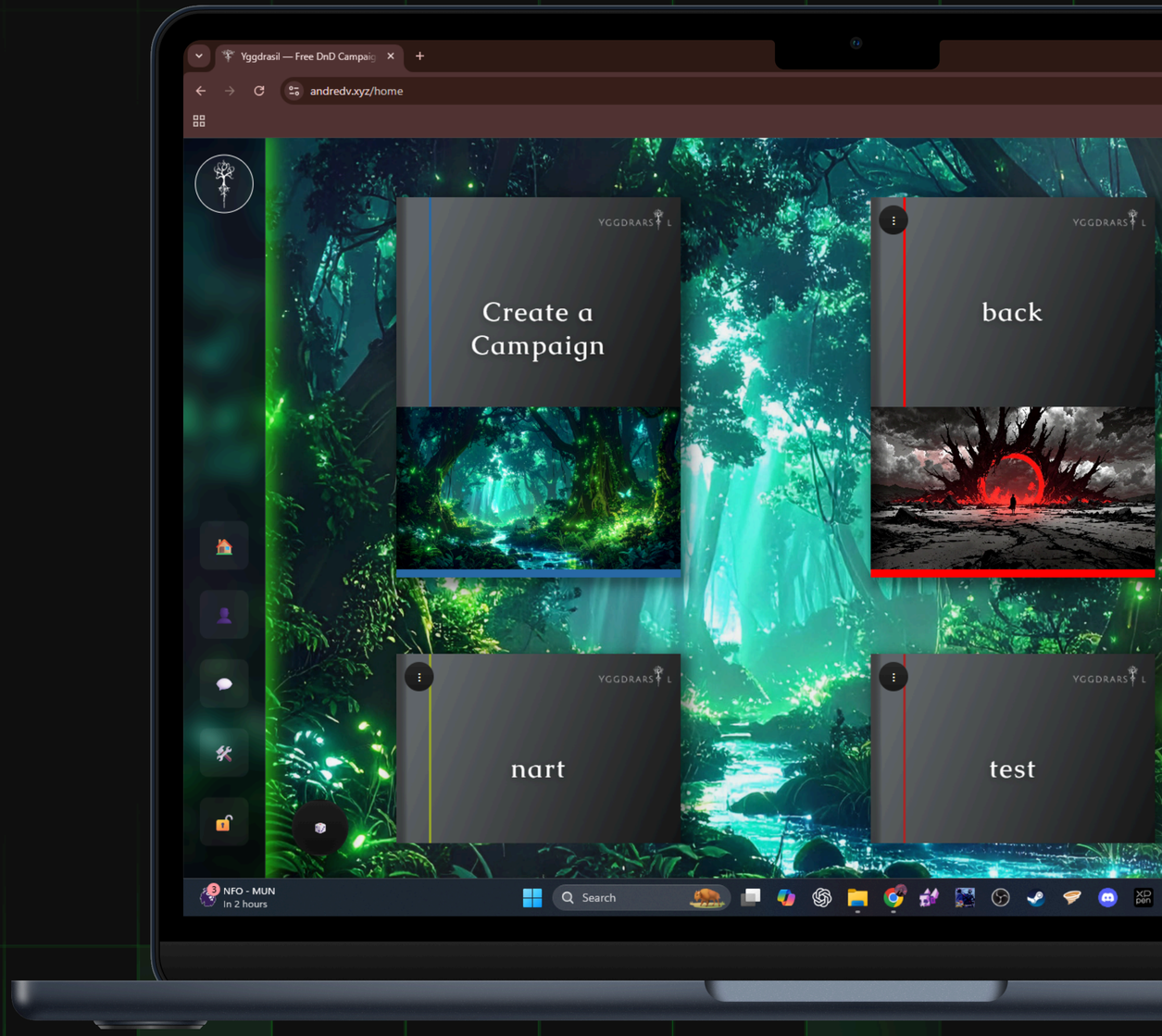
Deployment

Deploy both frontend and backend to cloud platforms such as AWS, Vercel, or Render, ensuring proper database connection, CORS handling, and environment security.

What is Yggdrasil

Yggdrasil is an all-in-one platform for Dungeon Masters and D&D players, designed to streamline campaigns.

- Simplifies running and managing adventures for newcomers and veterans.
- Features a game-style system to create classes, spells, weapons, items, monsters, characters, and more.
- Focuses on accessibility, flexibility, and intuitive design.
- Offers a paywall-free, clutter-free alternative to existing tools like D&D Beyond.



What problems does Yggdrasil solve

Problems Yggdrasil Solves:

- Limits on creativity due to paywalls and restricted customization in other D&D platforms.
- Difficulty managing campaigns freely and efficiently.
- Lack of accessible tools for designing unique characters, encounters, worlds, classes, spells, and items.
- Cluttered or confusing interfaces that slow down gameplay.

What was used to build Yggdrasil

Frontend: React

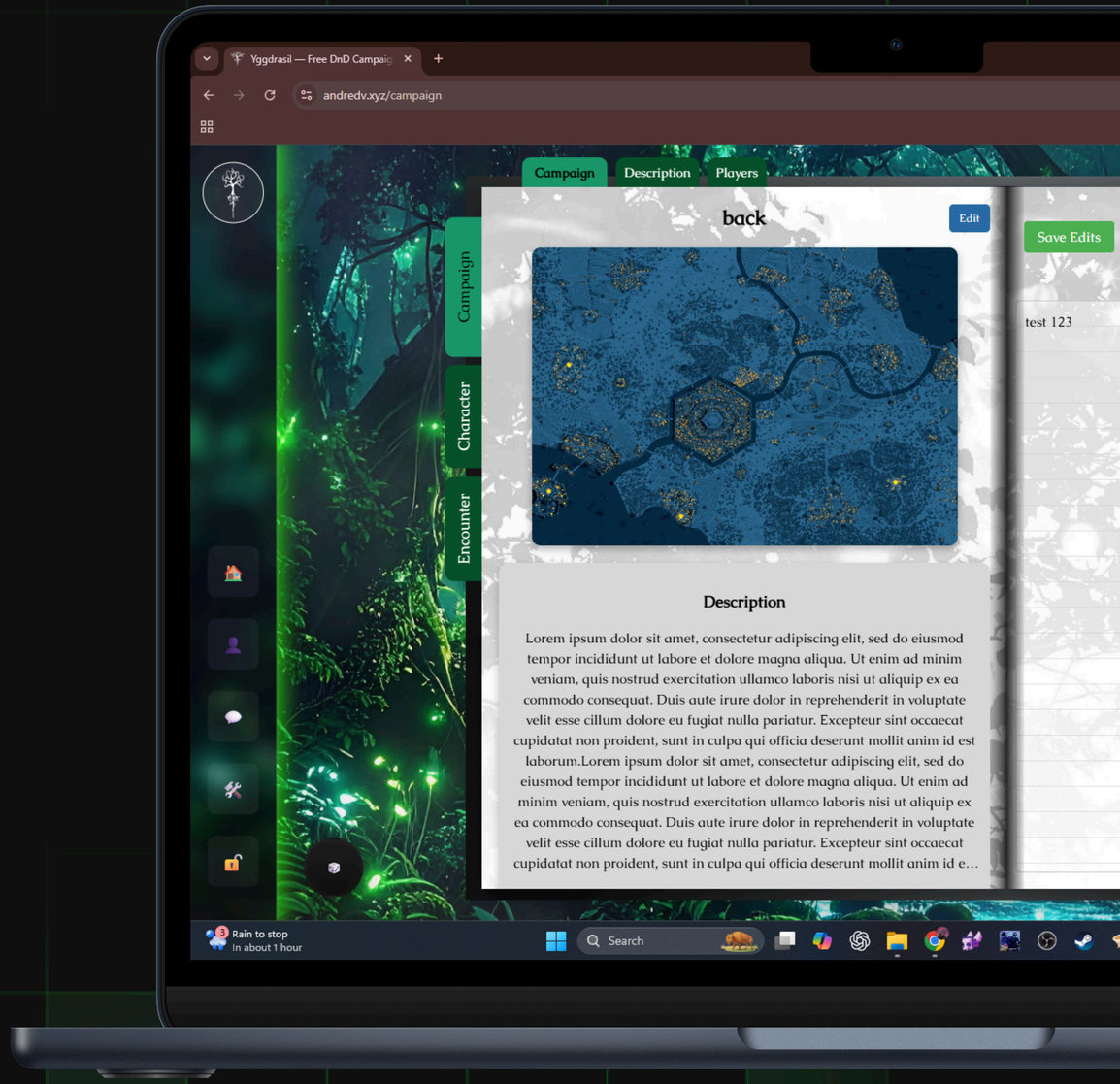
- Modern JavaScript library for dynamic, responsive UIs
- Ideal for interactive features like character builders and live campaign previews

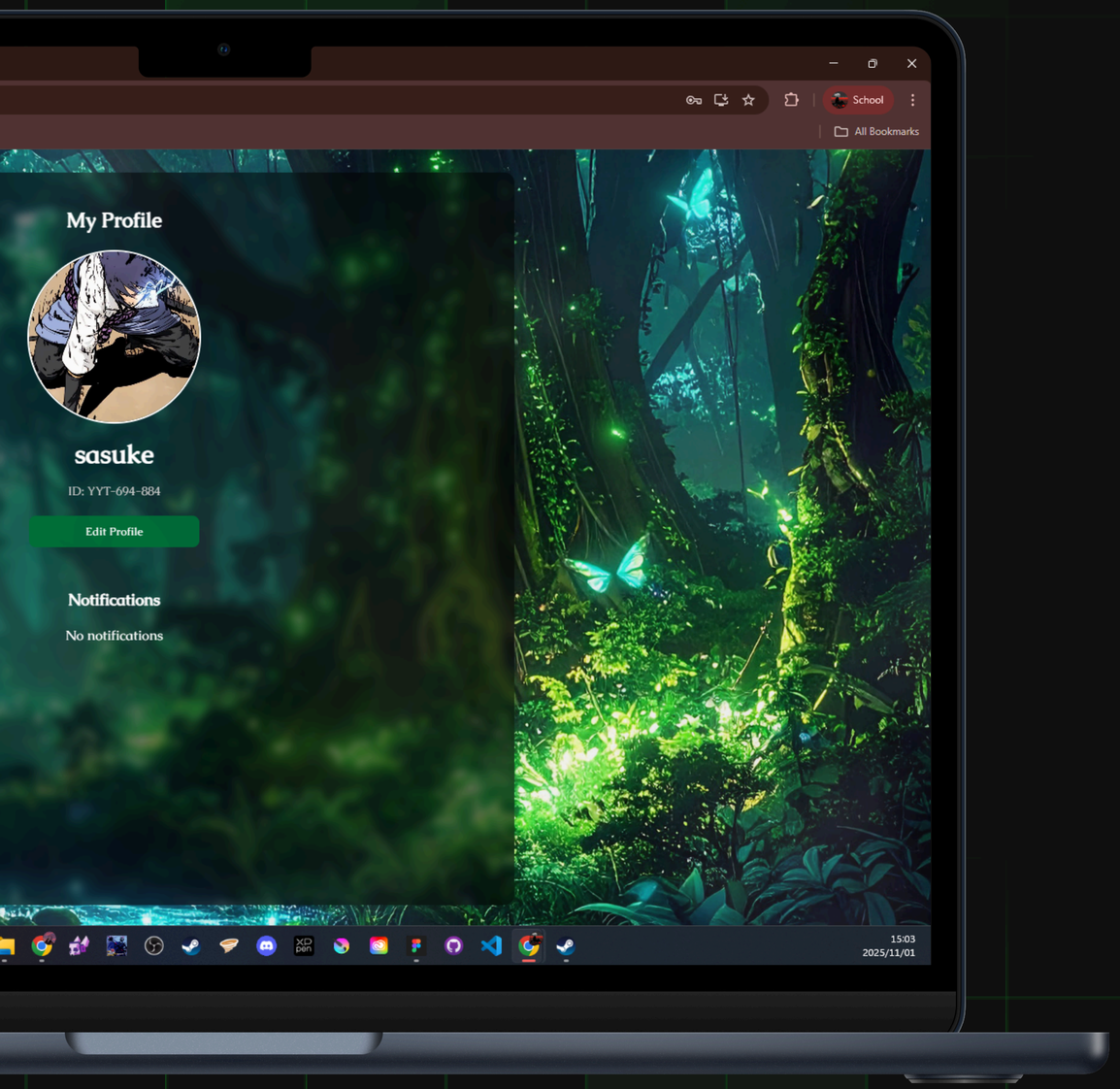
Backend: Node.js & Express

- Fast, scalable server-side runtime (Node.js)
- Simplified routing and API handling (Express)
- Supports real-time gameplay features

Database: MySQL

- Reliable relational database for structured D&D data
- Handles complex queries and maintains data integrity
- Manages relationships between characters, stats, spells, and inventories





What was used to Deploy Yggdrasil

Frontend: Vercel

- a fast and developer-friendly hosting platform built specifically for modern JavaScript frameworks like React.
- Vercel automates builds, provides instant deployments, and makes hosting frontend apps simple and reliable.

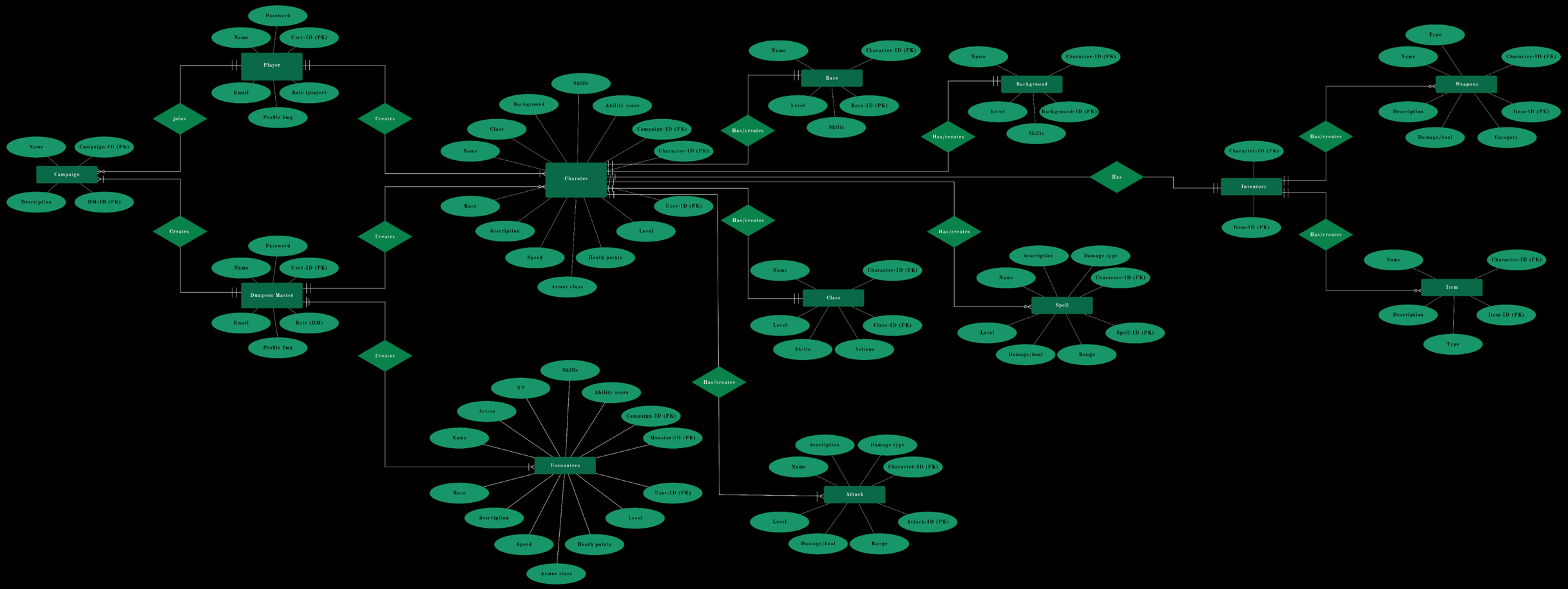
Backend: Render

- a platform that hosts web servers and APIs with automatic scaling, easy environment variable management, and smooth continuous deployment.

Database: Alwaysdata

- a hosting provider that offers managed MySQL databases, making it easy to store and manage persistent application data.

Entity-Relationship Diagram (ERD)



Live demo and important code overview

Fully deployed website : https://andredv.xyz/

Primary Key Generation and Naming Convention

This code demonstrates how I implemented a consistent primary key naming convention for users.

Each user is assigned a **user_id** in the format **AAA-000-001**, which is automatically generated and guaranteed to be unique.

This approach not only prevents any collisions in the database but also makes it easier to track and reference users across the backend logic.

Using a readable and standardized key structure ensures clarity when managing relationships between tables and improves maintainability as the application grows.

```
// Generate a random user_id like AAA-000-001
function generateUserId() {
  const letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  const randomLetter = () => letters[Math.floor(Math.random() * letters.length)];
  const randomNumber = () => String(Math.floor(Math.random() * 1000)).padStart(3, "0");
  return `${randomLetter()}${randomLetter()}${randomLetter()}-${randomNumber()}-${randomNumber()}`;
}

// Ensure unique user_id
async function generateUniqueUserId() {
  let user_id;
  let exists = true;
  while (exists) {
    user_id = generateUserId();
    const [rows] = await pool.query("SELECT user_id FROM users WHERE user_id = ?", [user_id]);
    exists = rows.length > 0;
  }
  return user_id;
}
```

Role-Based Access Control

This component ensures that only users with the **super_admin** role can access certain pages.
When the component mounts, it fetches the current user's role from the backend using the stored token.

If the role is not **super_admin**, the user is immediately redirected to the home page. Otherwise, the component renders its children, giving access to the restricted content.

This approach provides secure, centralized role-based access control, preventing unauthorized users from even seeing sensitive admin pages, rather than just hiding buttons in the UI.

It's a best practice for building secure, scalable applications with React.

```jsx
import React, { useEffect, useState } from "react";
import { Navigate } from "react-router-dom";
import API from "../../../api";

const RequireSuperAdmin = ({ children }) => {
  const [role, setRole] = useState(null);
  const [loading, setLoading] = useState(true);
  const token = localStorage.getItem("token");

  useEffect(() => {
    const fetchUser = async () => {
      if (!token) return setLoading(false);
      try {
        const res = await API.get("/api/users/me", {
          headers: { Authorization: `Bearer ${token}` }
        });
        // Normalize role just like Navbar
        setRole(res.data.role?.toLowerCase().replace(' ', '_'));
      } catch (err) {
        console.error(err);
      } finally {
        setLoading(false);
      }
    };
    fetchUser();
  }, [token]);

  if (loading) return <p>Loading...</p>;
  if (role !== "super_admin") return <Navigate to="/home" replace />;
  return children;
};

export default RequireSuperAdmin;
```

Securing User Authentication and Data

This snippet demonstrates how we **securely handle user authentication.**

During signup, user passwords are hashed with bcrypt before being stored in the database, so even if the database is compromised, raw passwords aren't exposed.

During login, the hashed password is compared against the input, ensuring credentials are validated securely. After verification, a JWT token is generated containing the user's ID and role, which allows secure, token-based authentication for subsequent requests.

This approach keeps sensitive user data safe, enforces role-based access, and prevents unauthorized access to protected routes.

It's a standard, secure practice for modern web applications.

```
// Signup – hashing the password
const hashedPassword = await bcrypt.hash(password, 10);
await createUser({ user_id, username, email, password: hashedPassword, role });

// Login – validating password and generating token
const match = await bcrypt.compare(password, user.password);
if (!match) return res.status(400).json({ error: "Invalid credentials" });

const token = jwt.sign(
  { user_id: user.user_id, role: user.role },
  process.env.JWT_SECRET,
  { expiresIn: "1h" }
);

res.json({ message: "Login successful", token, user });
```

API Routing Setup

This code snippet shows how I set up the API routing for encounters in my backend.

Each route corresponds to a CRUD operation: creating a new encounter, fetching encounters by campaign ID, retrieving a specific encounter by its unique ID, updating, and deleting.

I've structured the routes to follow RESTful conventions, which makes the API predictable and easy to maintain.

By separating the routes and controllers, the backend remains modular, clean, and scalable.

```
const express = require("express");
const router = express.Router();
const encounterController = require("../controllers/encounterController");

// Routes
router.post("/", encounterController.create);
router.get("/:campaignId", encounterController.getByCampaign);
router.get("/id/:id", encounterController.getById);
router.put("/:id", encounterController.update);
router.delete("/:id", encounterController.delete);

module.exports = router;
```

SQL Query

This code snippet shows how I insert **encounters** into the database.

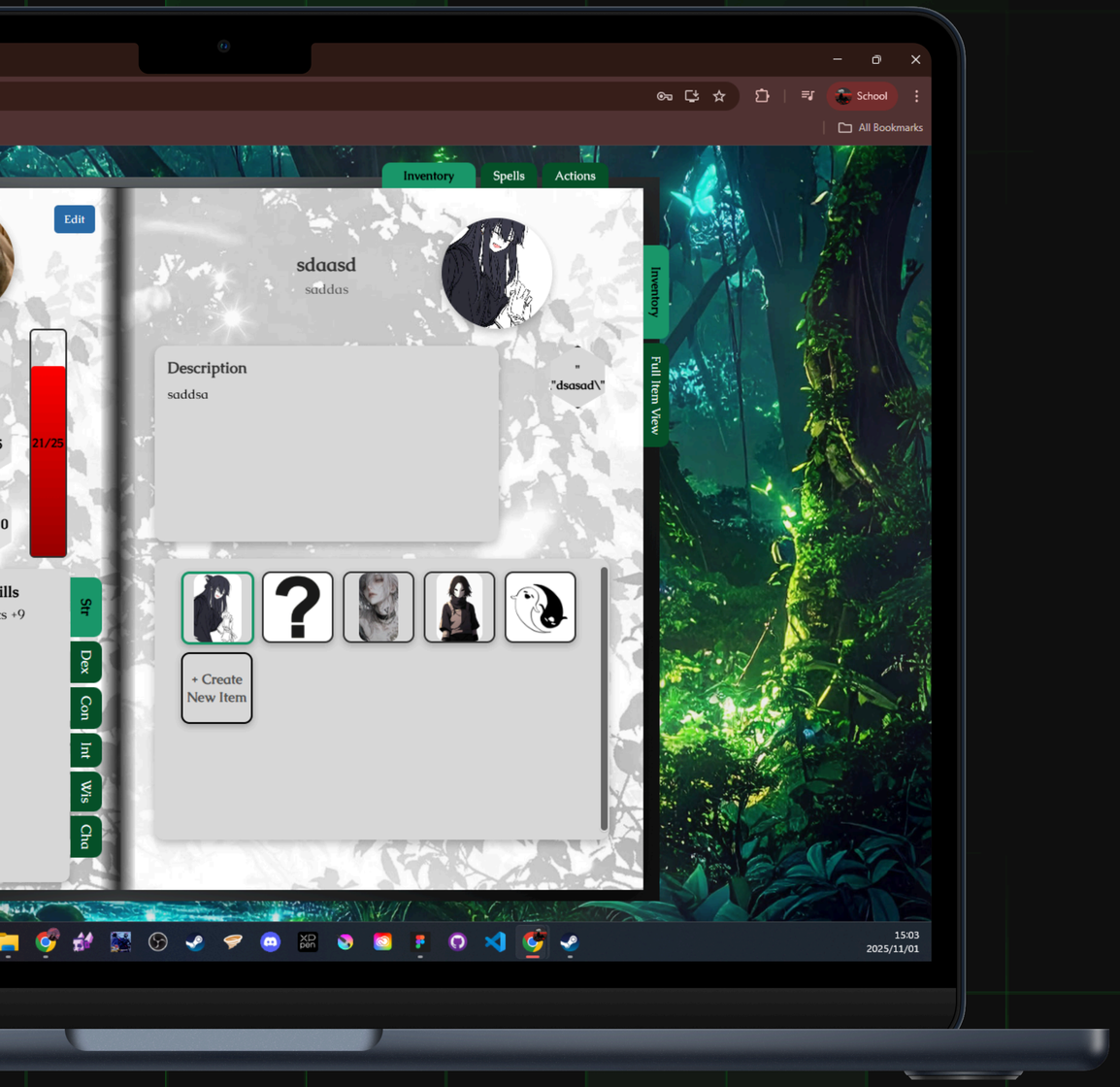
Each field in the encounters table corresponds to attributes from the ERD, including general encounter stats, ability scores, skills, and race-specific details. You'll notice that arrays, like proficiencies and tools, are stored as JSON strings to keep the schema flexible.

This query ensures all required fields are properly saved, and by using parameterised **queries (?)**, we prevent SQL injection, keeping the data secure and consistent.

```
const createEncounter = async (data) => {
  const sql = `
    INSERT INTO encounters (
      encounter_id, campaign_id, encounter_name, encounter_img,
      encounter_AC, encounter_level, encounter_speed,
      encounter_current_HP, encounter_max_HP,
      encounter_ability_score_str, encounter_ability_score_dex,
      encounter_ability_score_con, encounter_ability_score_int,
      encounter_ability_score_wis, encounter_ability_score_cha,
      skill_modefed_1, skill_modefed_2, encounter_dec,
      race_name, race_dec, race_skill_modefed_1, race_skill_modefed_2,
      race_proficiencie_languages, race_proficiencie_tools
    ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
  `;

  const values = [
    data.encounter_id, data.campaign_id, data.encounter_name, data.encounter_img,
    data.encounter_AC, data.encounter_level, data.encounter_speed,
    data.encounter_current_HP, data.encounter_max_HP,
    data.encounter_ability_score_str, data.encounter_ability_score_dex,
    data.encounter_ability_score_con, data.encounter_ability_score_int,
    data.encounter_ability_score_wis, data.encounter_ability_score_cha,
    data.skill_modefed_1, data.skill_modefed_2, data.encounter_dec,
    data.race_name, data.race_dec, data.race_skill_modefed_1, data.race_skill_modefed_2,
    JSON.stringify(data.race_proficiencie_languages || []),
    JSON.stringify(data.race_proficiencie_tools || [])
  ];

  const [result] = await pool.execute(sql, values);
  return result;
};
```



Challenges

While building Yggdrasil, I faced several challenges that helped me grow as a developer. Time management was a major difficulty, as balancing multiple large projects often led to rushed “crunch time” sessions. This taught me the value of better planning and prioritisation.

Switching from MongoDB to MySQL was another challenge, especially with self-hosting and configuring XAMPP. After troubleshooting connection issues, I gained a stronger understanding of database setup and stability.

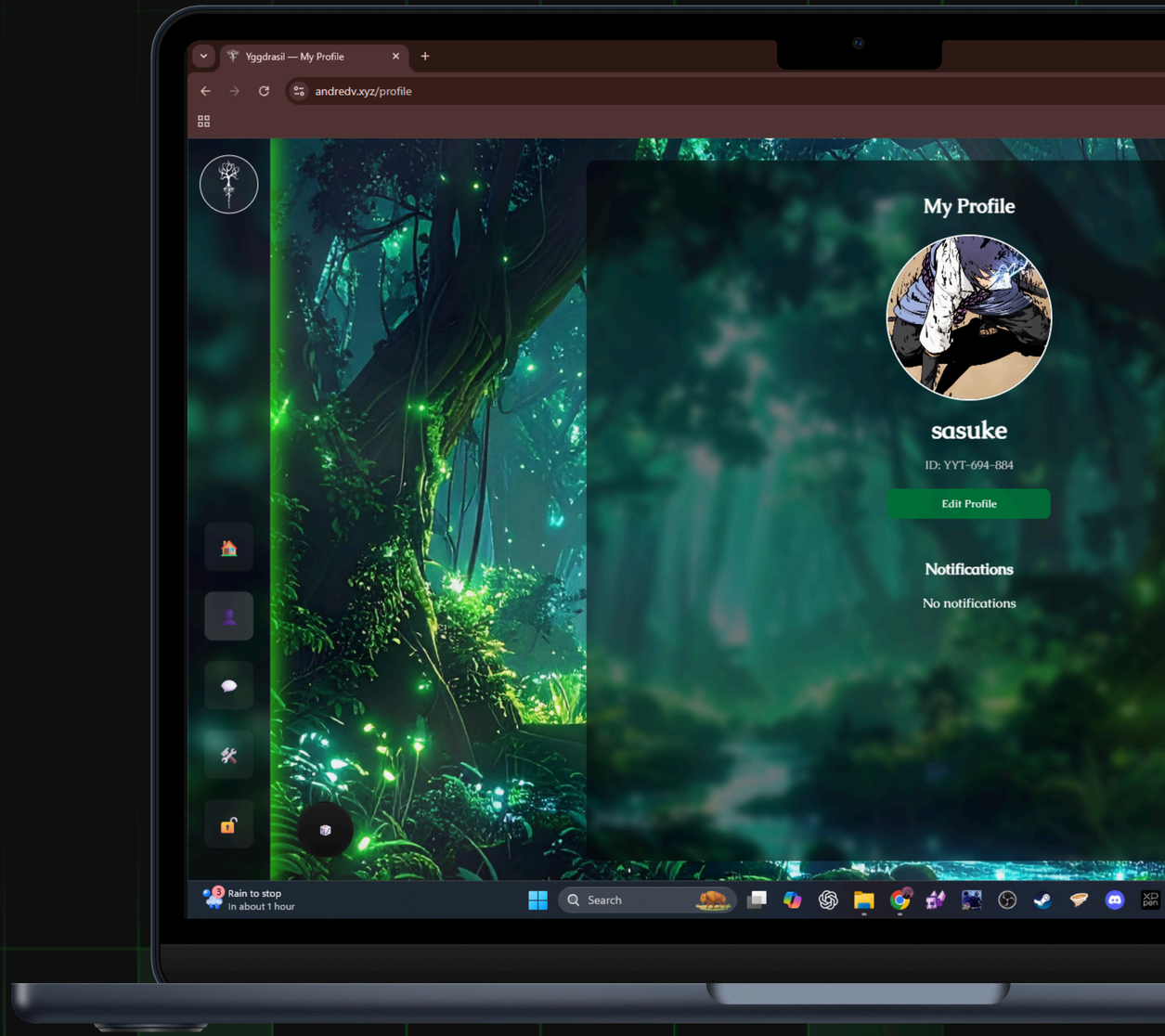
Deployment also proved tricky. Platforms like Azure were powerful but not always intuitive, making tasks like linking the frontend and backend or managing environment variables confusing at first. Working through this improved my confidence with hosting and deployment environments.

Overall, these challenges strengthened my technical skills and reinforced the importance of planning, problem-solving, and persistence.

Conclusion

This project was the most challenging one I've ever worked on, both in workload and complexity. Throughout the process I learned a huge amount—from time management and deployment, to database development, routing, and problem-solving.

Every part of Yggdrasil pushed me to think deeper, whether it was designing a book-style website with no scrolling, or researching the best technologies for deployment. These challenges strengthened my skills as a developer and showed me how much I've grown in planning, persistence, and building complete full-stack systems.



Thank you for your time