Armstrong Number Verification Web Application

December 17, 2024

BHAGYA SURESH KUMAR

Table of Contents

1 Introduction

The focus of this application is to verify Armstrong numbers, manage user-specific data, and provide meaningful insights through a well-integrated system. Armstrong numbers, also known as narcissistic numbers, hold mathematical significance as they are numbers that are equal to the sum of their own digits each raised to the power of the number of digits. Identifying these numbers programmatically while ensuring seamless data management is a key objective of the system.

The application offers the following core functionalities:

- **Verification of Armstrong numbers:** Users can input a number to check whether it is an Armstrong number. .
- **User-specific data management:** Armstrong numbers verified by users are stored securely in a database. This allows users to view their previously verified numbers.
- **Global data display:** The system provides an overview of all users and their verified Armstrong numbers.

To achieve these objectives, a modern full-stack technology stack has been adopted. The user interface is built using **React**, a popular JavaScript library for developing dynamic and responsive frontend applications. React enables smooth navigation, clean user interactions, and an intuitive design. On the server side, **GoLang** (also referred to as Go) is utilized for its performance, concurrency support, and simplicity. GoLang efficiently handles API requests, processes logic for Armstrong number verification, and interacts with the database layer. For reliable and structured data storage, **MySQL** is employed as the database. MySQL ensures efficient data retrieval, storage, and management, supporting both user-specific and global data queries.

By combining mathematical computations with modern software development practices, the application not only verifies Armstrong numbers but also manages and displays user-specific data in an organized manner. The detailed system design, implementation, and performance evaluation will be discussed in the subsequent sections.

2 Project Features

The application incorporates several features to ensure a seamless and efficient user experience. The functionalities provided are as follows:

- **User Registration:**
  - Allows users to create accounts using their email.
  - Basic form validation ensures that user inputs are valid and complete.
- **Armstrong Number Verification:**
  - An input field is provided for users to enter a number.
  - A check button allows users to verify whether the entered number is an Armstrong number.
  - Verified Armstrong numbers are saved to the database for future retrieval.
- **User Dashboard:**
  - Displays a list of Armstrong numbers verified by the logged-in user.
  - Pagination is implemented to handle large datasets efficiently.
- **Global Users Dashboard:**
  - Displays a list of all registered users along with their verified Armstrong numbers.
  - Filtering options, such as showing only positive or negative numbers, are provided.
  - Pagination is implemented to manage large datasets seamlessly.
- **Error Handling:**
  - Input validation is performed on both the frontend and backend to ensure data integrity.
  - User-friendly error messages guide users when invalid inputs or unexpected errors occur.
- **Bonus Features:**
  - Users can input a range (e.g., 10-100), and the page returns the Armstrong numbers within that range.

The above features collectively ensure that the application meets its objectives while providing a smooth and intuitive interface for users.

3 System Design

The system design focuses on creating a robust and efficient architecture for verifying Armstrong numbers, storing and retrieving user-specific data, and providing global insights. This section details the database schema, API endpoints, and overall system structure.

3.1 Database Schema

The database schema is designed to support both user-specific and global data management efficiently. It consists of two main tables: **Users** and **Numbers**.

- **Users Table:**
  - `user_id` (Primary Key): A unique string identifier for each user.
  - `name`: The user's full name.
  - `email` (Unique): The user's email, used for registration and login.
  - `created_at`: Timestamp indicating when the user account was created.
- **Numbers Table:**
  - `id` (Primary Key): A unique identifier for each record in the numbers table.
  - `user_id` (Foreign Key): References the `user_id` in the Users table to associate numbers with a specific user.
  - `number`: The number entered by the user, regardless of whether it's Armstrong or not.
  - `result`: Stores "positive" if the number is Armstrong and "negative" if it is not.
  - `created_at`: Timestamp of when the number was entered and stored.

The database structure ensures that each number entered by a user, along with its result, is linked to that user, allowing for easy retrieval of the data and verification.

3.2 API Endpoints

The backend provides a set of API endpoints that allow interaction with the frontend. These endpoints include user management, Armstrong number verification, and data retrieval for both individual users and global users.

- **User Authentication and Management:**
  - `POST /users`: Handles user authentication and registration. If the user is new, it enters the user details (including a unique `user_id`, which is a unique string) into the database.

- **Armstrong Number Verification:**
  - `POST /verify`: Takes the number entered by the user, checks whether the user has already entered that number before. If not, it checks whether the number is Armstrong, and then inputs the number details (positive or negative result), `user_id`, and `created_at` fields into the `Numbers` table.
- **User-Specific Data Retrieval:**
  - `GET /getuserdet`: Receives the `user_id` from the frontend (via `UserContext`) and checks the `Numbers` table to return the total number of searches, total positives, and total negatives for the user.
- **Global Users Data Retrieval:**
  - `GET /getremainingusers`: Returns the total number of searches, `user_id`, total positives and negatives, and the `name` of all users other than the one who is logged in.
- **User Numbers Data Retrieval:**
  - `GET /getusernumbers`: Takes the `user_id` from the frontend (via the URL) and returns all the numbers entered by the user, along with the result ("positive" or "negative") and the `created_at` field, which the frontend displays.

These API endpoints facilitate the core functionalities of the application, including user management, Armstrong number verification, and the retrieval of user-specific or global data.

3.3 System Overview

The frontend communicates with the backend through API requests. The user interacts with the frontend application, which collects input data, sends requests to the backend for verification and storage, and retrieves relevant data for display.

- **Frontend (React):** The React application provides an interactive interface, including forms for user registration, number verification, and dashboards for displaying Armstrong numbers.
- **Backend (GoLang):** The Go server handles API requests, performs the Armstrong number verification, and interacts with the MySQL database to store and retrieve data.
- **Database (MySQL):** The MySQL database stores user and number data, supporting efficient queries, filtering, and pagination.

This architecture ensures that the system is scalable, responsive, and able to handle large datasets effectively while providing a seamless user experience.

4 Implementation Details

This section outlines the implementation of the system, covering frontend and backend components, CORS handling, and database queries.

4.1 Frontend Implementation

The frontend of the system is built using React, providing an interactive interface for user registration, Armstrong number verification, and data display. Below are the key components and sample code placeholders for each:

- **login.js:** Handles user authentication and login.

```
import React, { useState, useEffect } from "react";
import { useUser } from "../context/UserContext"; // Access the user context
import { useNavigate } from "react-router-dom"; // Import useNavigate

const LoginForm = () => {
  const { setUserDetails, user } = useUser(); // Access the user context and setUserDetails function
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const navigate = useNavigate(); // Hook for navigation

  useEffect(() => {
    // Clear the user context when the login page loads
    console.log("Resetting UserContext");
    setUserDetails({ user_id: null, name: "", email: "" });
    localStorage.removeItem("user_id");
    localStorage.removeItem("name");
    localStorage.removeItem("email");
  }, []); // Runs only when the component mounts

  const handleLogin = async (e) => {
    e.preventDefault();

    console.log("Name entered:", name);
    console.log("Email entered:", email);

    // Input validation
    if (!name.trim()) {
      alert("Name is required");
      return;
    }
    if (!email.trim()) {
      alert("Email is required");
      return;
    }

    try {
```

```
      const response = await fetch("http://localhost:8080/users", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ name, email }),
      });

      const data = await response.json();
      console.log("Backend response:", data);

      if (response.ok) {
        // Update UserContext and localStorage with the new user_id
        setUserDetails({
          user_id: data.user_id, // Use the unique `user_id`
          name: data.name,
          email: data.email,
        });

        console.log("UserContext updated:", {
          user_id: data.user_id,
          name: data.name,
          email: data.email,
        });

        // Persist user details in localStorage
        localStorage.setItem("user_id", data.user_id);
        localStorage.setItem("name", data.name);
        localStorage.setItem("email", data.email);

        alert("User successfully logged in!");
        navigate("/home"); // Navigate to the Armstrong check page
      } else {
        alert(data.error || "Login failed");
      }
    } catch (error) {
      console.error("Error during login:", error);
      alert("An error occurred. Please try again later.");
    }
  };

  return (
    <div className="flex items-center justify-center min-h-screen bg-custom-blue">
      <div className="md:w-2/6 md:h-2/3 bg-white p-8 rounded-lg shadow-lg">
        <form onSubmit={handleLogin}>
          <h2 className="text-4xl font-bold mb-4 text-center
```

```
text-custom-blue">Welcome</h2>
          <div className="mb-4">
            <label htmlFor="name" className="block text-xl font-bold text-custom-blue">
              Name
            </label>
            <input
              type="text"
              id="name"
              name="name"
              value={name}
              onChange={(e) => setName(e.target.value)} // Updates `name` state
              className="mt-1 block w-full px-4 py-2 bg-gray-200 text-gray-900 rounded-md focus:outline-none focus:ring-2 focus:ring-custom-blue focus:border-transparent"
            />
          </div>
          <div className="mb-4">
            <label htmlFor="email" className="block text-xl font-bold text-custom-blue">
              Email
            </label>
            <input
              type="email"
              id="email"
              name="email"
              value={email}
              onChange={(e) => setEmail(e.target.value)} // Updates `email` state
              className="mt-1 block w-full px-4 py-2 bg-gray-200 text-gray-900 rounded-md focus:outline-none focus:ring-2 focus:ring-custom-blue focus:border-transparent"
            />
          </div>
          <div className="flex justify-center">
            <button
              type="submit"
              className="bg-custom-blue text-xl text-white font-bold py-2 px-4 rounded shadow hover:bg-opacity-90 focus:outline-none focus:ring focus:ring-blue-500"
            >
              LOGIN
            </button>
          </div>
        </form>
      </div>
    </div>
```

```
  );
};

export default LoginForm;
```

Listing 1: login.js

checkArm.js: A page for verifying if a number is Armstrong.

```
import React, { useState } from "react";
import { useUser } from "../context/UserContext"; // Access user data from the context
import thumbsUp from "../static/thumbsUp.gif";
import thumbsDown from "../static/thumbsDown.gif";
import NavBar from "./navbar"; // Import NavBar component

const CheckNumber = () => {
  const { user } = useUser(); // Access user data from the context
  const [number, setNumber] = useState("");
  const [result, setResult] = useState("");

  const handleCheck = async () => {
    if (!/^\d+$/.test(number)) {
      alert("Please enter a valid numeric value");
      return;
    }

    const num = parseInt(number, 10);

    // Check if the number is Armstrong
    const isArmstrongNumber = isArmstrong(num);
    const resultText = isArmstrongNumber ? "positive" : "negative";

    try {
      // Send the number, user id, and result to the backend
      const response = await fetch("http://localhost:8080/verify", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          user_id: user.user_id, // Pass the UID from context
          number: num,
          result: resultText, // Send the result as positive or negative
        }),
      });

      if (response.ok) {
```

```
        const data = await response.json();
        console.log("Backend Response:", data);
        // Set result based on the backend response
        setResult({
          message: isArmstrongNumber
            ? `Congrats! Number ${num} is an Armstrong number.`
            : `Number ${num} is not an Armstrong number.`,
          gif: isArmstrongNumber ? thumbsUp : thumbsDown,
        });
      } else {
        const error = await response.json();
        alert(error.error || "An error occurred");
        setResult({
          message: `An error occurred while checking the number.`,
          gif: thumbsDown,
        });
      }
    } catch (error) {
      console.error("Error:", error);
      alert("An error occurred. Please try again.");
    }
  };

  const isArmstrong = (num) => {
    const digits = num.toString().split("").map(Number);
    const sum = digits.reduce((acc, digit) => acc + Math.pow(digit, digits.length), 0);
    return sum === num;
  };

  return (
    <div className="bg-gray-100 min-h-screen flex flex-col"> {/* Ensured full height with min-h-screen */}
      <NavBar /> {/* Fixed NavBar at the top */}

      <div className="flex flex-col items-center justify-start flex-1 pt-28"> {/* pt-28 to ensure spacing for NavBar */}
        <div className="bg-custom-blue text-white p-8 px-0 rounded-lg shadow-md w-[30%] pb-15 h-[40%] flex flex-col opacity-90 items-center justify-center">
          {/* Increased height and added flex-grow to parent container */}
          <div className="flex flex-col justify-between h-full w-full p-8 px-0">
            <div className="flex flex-col items-center">
              <h1 className="text-2xl font-bold mb-4 text-center pb-2">Let's Check a number</h1>
              <label htmlFor="number" className="block text-2xl mb-2 text-centerpb-5">
```

```
                Enter a number
              </label>
              <input
                id="number"
                type="text"
                value={number}
                onChange={(e) => setNumber(e.target.value)}
                className="w-[70%] p-2 rounded bg-white text-custom-blue text-center text-xl border focus:outline-none focus:ring-2 focus:ring-blue-500"
              />
            </div>
            <button
              onClick={handleCheck}
              className="mt-4 bg-white text-xl font-bold text-custom-blue px-4 py-2 rounded hover:text-bold transition self-center"
            >
              Check
            </button>
          </div>
        </div>

        {result && (
          <div className="flex items-center pt-5 w-[20%] justify-center space-x-0">
            <img src={result.gif} alt="Result" className="w-[120px] h-[100px]" />
            <span className="text-xl text-center font-bold text-custom-blue">{result.message}</span>
          </div>
        )}
      </div>
    </div>
  );
};

export default CheckNumber;
```

Listing 2: checkArm.js

userDetails.js: Displays user-specific details like searches, positives, and negatives.

```
import React, { useEffect, useState } from "react";
import { useNavigate } from "react-router-dom"; // Import useNavigate from react-router-dom
import UserBox from "./userBox";
import { useUser } from "../context/UserContext";
import NavBar from "./navbar";

const UserDetails = () => {
  const [userDetails, setUserDetails] = useState(null);
```

```
const [otherUsers, setOtherUsers] = useState([]); // Initialize as an empty array
const { user } = useUser();
const navigate = useNavigate(); // Initialize the navigate function

useEffect(() => {
  if (user && user.user_id) {
    console.log(`Attempting to fetch user details for ID: ${user.user_id}`);

    fetch(`http://localhost:8080/getuserdet?user_id=${encodeURIComponent(user.user_id)}`, {
      method: "GET",
      headers: {
        Accept: "application/json",
      },
    })
      .then((response) => response.json())
      .then((data) => {
        console.log("User Details fetched:", data);
        setUserDetails(data);
      })
      .catch((error) => console.error("Error fetching user details:", error));

    // Fetch remaining users
    fetch(`http://localhost:8080/getremainingusers?user_id=${encodeURIComponent(user.user_id)}`, {
      method: "GET",
      headers: {
        Accept: "application/json",
      },
    })
      .then((response) => response.json())
      .then((data) => {
        console.log("Other Users fetched:", data);
        setOtherUsers(data); // Store the list of other users
      })
      .catch((error) => console.error("Error fetching other users:", error));
  } else {
    console.warn("User ID is not available. Cannot fetch details.", user);
  }
}, [user]);

// Ensure that `otherUsers` is an array before attempting to use `.map()`
if (!Array.isArray(otherUsers)) {
  console.error("otherUsers is not an array", otherUsers);
```

```
    return <div>Loading...</div>;
  }

  const handleUserClick = (userID) => {
    // Navigate to the dynamic user details page with the user ID
    navigate(`/user-details/${userID}`);
  };

  if (!userDetails) {
    return <div>Loading...</div>;
  }

  return (
    <div className=" min-h-screen text-white flex flex-col">
      {/* Navbar at the top */}
      <NavBar />

      {/* Main content container */}
      <div className="flex flex-col bg-custom-blue items-center justify-start flex-grow pt-20 px-10">
        <h1 className="text-4xl font-semi-bold mb-8">Hey {user.name}, Let's meet our users</h1>

        {/* Current user details */}
        <div className="flex items-center w-full pt-10 pb-8 border-b-[0.1px] border-white/40">
          <span className="mr-8 pr-10 font-bold text-xl">You</span>
          <UserBox
            name={userDetails.name || user.name}
            searches={userDetails.searches}
            positives={userDetails.positives}
            negatives={userDetails.negatives}
            onClick={() => handleUserClick(user.user_id)} // Pass the user ID to handle the click
          />
        </div>

        {/* Other users */}
        <div className="flex flex-wrap justify-start w-full gap-6 pt-10">
          {otherUsers.length > 0 ? (
            otherUsers.map((otherUser) => (
              <UserBox
                key={otherUser.user_id}
                name={otherUser.name}
                searches={otherUser.searches}
                positives={otherUser.positives}
                negatives={otherUser.negatives}
                onClick={() => handleUserClick(otherUser.user_id)}
```

```
              />
            ))
          ) : (
            <p>No other users found.</p>
          )}
        </div>
      </div>
    </div>
  );
};

export default UserDetails;
```

Listing 3: userDetails.js

userPage.js: Displays a table which has details about the numbers entered by the user like result, date, and time.

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

const UserPage = () => {
  const { user_id } = useParams();
  const [userDetails, setUserDetails] = useState(null);
  const [numberDetails, setNumberDetails] = useState([]);

  const [currentPage, setCurrentPage] = useState(1);
  const [totalPages, setTotalPages] = useState(0);
  const [totalCount, setTotalCount] = useState(0);
  const [pageSize, setPageSize] = useState(5); // Default to 5 rows per page
  const [filter, setFilter] = useState("all"); // New filter state

  // Fetch user and number details
  const fetchUserDetails = useCallback(() => {
    if (user_id) {
      console.log(`Fetching details for user ID: ${user_id}`);

      // Fetch user details
      fetch(
        `http://localhost:8080/getuserdet?user_id=${encodeURIComponent(user_id)}`,
        {
          method: "GET",
          headers: { Accept: "application/json" },
        }
      )
        .then((response) => response.json())
        .then((data) => {
          console.log("User Details fetched:", data);
          setUserDetails(data);
```

```
        })
        .catch((error) => console.error("Error fetching user details:", error));

      // Fetch paginated and filtered number details
      fetch(
        `http://localhost:8080/getusernumbers?user_id=${encodeURIComponent(
          user_id
        )}&page=${currentPage}&page_size=${pageSize}&filter=${filter}`, // Include filter in query
        {
          method: "GET",
          headers: { Accept: "application/json" },
        }
      )
        .then((response) => response.json())
        .then((data) => {
          console.log("Number Details fetched:", data);
          setNumberDetails(data.number_details);
          setTotalPages(data.total_pages);
          setTotalCount(data.total_count);
        })
        .catch((error) =>
          console.error("Error fetching user number details:", error)
        );
    } else {
      console.warn("No user_id found");
    }
  }, [user_id, currentPage, pageSize, filter]);

  useEffect(() => {
    fetchUserDetails();
  }, [fetchUserDetails]);

  // Pagination handlers
  const handleNextPage = () => {
    if (currentPage < totalPages) {
      setCurrentPage(currentPage + 1);
    }
  };

  const handlePrevPage = () => {
    if (currentPage > 1) {
      setCurrentPage(currentPage - 1);
    }
  };

  // Page size change handler
  const handlePageSizeChange = (e) => {
```

```
    setPageSize(parseInt(e.target.value, 10));
    setCurrentPage(1); // Reset to the first page when the page size changes
  };

  // Filter change handler
  const handleFilterChange = (e) => {
    setFilter(e.target.value);
    setCurrentPage(1); // Reset to the first page when the filter changes
  };

  if (!userDetails) {
    return <div className="text-center text-lg mt-10">Loading user details...</div>;
  }

  return (
    <div className="p-4 max-w-5xl mx-auto">
      <div className="flex items-center justify-between mb-20 mt-20">
        <button
          onClick={() => window.history.back()}
          className="text-custom-blue font-bold flex items-center"
        >
          <span className="mr-2 font-bold">&larr;</span> Back
        </button>

        <h1 className="text-3xl font-bold text-custom-blue mx-auto">
          Searches by {userDetails.name}
        </h1>

        {/* Filter Dropdown */}
        <select
          value={filter}
          onChange={handleFilterChange}
          className="border text-custom-blue font-bold rounded px-2 py-1"
        >
          <option value="all">All</option>
          <option value="positive">Positives</option>
          <option value="negative">Negatives</option>
        </select>
      </div>

      <div className="overflow-x-auto">
        <table className="table-auto border-separate border w-full">
          <thead>
```

```
            <tr className="bg-custom-blue text-white">
              <th className="px-4 py-2 text-left text-xl rounded-tl-lg">NUMBER</th>
              <th className="px-4 py-2 text-xl text-left">RESULT</th>
              <th className="px-4 py-2 text-xl text-left">DATE</th>
              <th className="px-4 py-2 text-left text-xl rounded-tr-lg">TIME</th>
            </tr>
          </thead>
          <tbody>
            {numberDetails.map((entry) => (
              <tr key={entry.id} className="bg-custom-blue">
                <td className="border px-4 py-2 text-white">{entry.number}</td>
                <td className="border px-4 py-2 text-white">{entry.result}</td>
                <td className="border px-4 py-2 text-white">
                  {new Date(entry.created_at).toLocaleDateString()}
                </td>
                <td className="border px-4 py-2 text-white">
                  {new Date(entry.created_at).toLocaleTimeString()}
                </td>
              </tr>
            ))}
          </tbody>
        </table>

        {/* Pagination Controls */}
        <div className="flex justify-between items-center mt-4">
          <div className="flex items-center">
            <label htmlFor="pageSize" className="mr-2 font-bold text-custom-blue">Show:</label>
            <select
              id="pageSize"
              value={pageSize}
              onChange={handlePageSizeChange}
              className="border text-custom-blue font-bold rounded px-2 py-1"
            >
              <option value={5}>5</option>
              <option value={10}>10</option>
              <option value={25}>25</option>
              <option value={50}>50</option>
            </select>
          </div>

          <div className="text-center font-bold text-custom-blue
```

```
              ">
            Page {currentPage} of {totalPages} (Total {totalCount} entries)
          </div>

          <div className="flex space-x-2">
            <button
              onClick={handlePrevPage}
              disabled={currentPage === 1}
              className={`px-4 py-2 border rounded ${
                currentPage === 1
                  ? 'bg-gray-200 font-bold text-custom-blue cursor-not-allowed'
                  : 'bg-custom-blue text-white font-bold hover:opacity-90'
              }`}
            >
              Previous
            </button>
            <button
              onClick={handleNextPage}
              disabled={currentPage === totalPages}
              className={`px-4 py-2 border rounded ${
                currentPage === totalPages
                  ? 'bg-gray-200 font-bold text-custom-blue cursor-not-allowed'
                  : 'bg-custom-blue text-white font-bold hover:opacity-90'
              }`}
            >
              Next
            </button>
          </div>
        </div>
      </div>
    </div>
  );
};

export default UserPage;
```

Listing 4: userPage.js

4.2 Backend Implementation with Database Queries

The backend is built using Go, which handles API requests for user authentication, Armstrong number verification, and data retrieval. Below are the handler functions corresponding to each route.

- **POST /users**: Handles user authentication and registration.

```go
package handlers

import (
	"backend/config"
	"backend/models"
	"encoding/json"
	"net/http"
	"regexp"
	"strings"
	"time"

	"github.com/google/uuid"
)

func LoginUser(w http.ResponseWriter, r *http.Request) {
	var input struct {
		Name  string `json:"name"`
		Email string `json:"email"`
	}
	if err := json.NewDecoder(r.Body).Decode(&input); err != nil {
		http.Error(w, "Invalid request payload", http.StatusBadRequest)
		return
	}

	// Validate inputs
	if strings.TrimSpace(input.Name) == "" {
		http.Error(w, "Name is required", http.StatusBadRequest)
		return
	}
	if strings.TrimSpace(input.Email) == "" {
		http.Error(w, "Email is required", http.StatusBadRequest)
		return
	}

	// Validate email format using regex
	emailRegex := `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`
	matched, err := regexp.MatchString(emailRegex, input.Email)
	if err != nil || !matched {
		w.Header().Set("Content-Type", "application/json")
		w.WriteHeader(http.StatusBadRequest)
		json.NewEncoder(w).Encode(map[string]string{"error": "Invalid email format"})
		return
	}

```

```go
        // Check if the user already exists
        var existingUser models.User
        if err := config.DB.Where("email = ?", input.Email).First(&existingUser).Error; err == nil {
                // Return the existing user's details
                w.Header().Set("Content-Type", "application/json")
                w.WriteHeader(http.StatusOK)
                json.NewEncoder(w).Encode(map[string]interface{}{
                        "message": "User already exists",
                        "user_id": existingUser.UserID, // Send unique user_id
                        "name":    existingUser.Name,
                        "email":   existingUser.Email,
                })
                return
        }

        // Generate a unique UserID
        newUserID := uuid.New().String()

        // Add new user to the database
        newUser := models.User{
                UserID:    newUserID,
                Name:      input.Name,
                Email:     input.Email,
                CreatedAt: time.Now(),
        }

        if err := config.DB.Create(&newUser).Error; err != nil {
                http.Error(w, "Error creating user", http.StatusInternalServerError)
                return
        }

        // Return the new user's details
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusCreated)
        json.NewEncoder(w).Encode(map[string]interface{}{
                "message": "User created successfully",
                "user_id": newUser.UserID, // Send unique user_id
                "name":    newUser.Name,
                "email":   newUser.Email,
        })
}
```

Listing 5: GoLang Login User Handler

- **POST /verify**: Verifies if a number is Armstrong and stores the result.

```go
package handlers

import (
    "backend/config"
    "backend/models"
    "encoding/json"
    "net/http"
    "time"
)
```

```go

// VerifyArmstrong checks if a number is Armstrong and adds it to the database
func VerifyArmstrong(w http.ResponseWriter, r *http.Request) {
    // Decode JSON request body
    var data struct {
        UserID string `json:"user_id"` // Use string to match UUID format
        Number int    `json:"number"`
        Result string `json:"result"` // Result will be either "positive" or "negative"
    }

    if err := json.NewDecoder(r.Body).Decode(&data); err != nil {
        http.Error(w, "Invalid request payload", http.StatusBadRequest)
        return
    }

    // Validate inputs
    if data.UserID == "" {
        http.Error(w, "User ID is required", http.StatusBadRequest)
        return
    }
    if data.Number <= 0 {
        http.Error(w, "Invalid number", http.StatusBadRequest)
        return
    }

    // Check if the number already exists for the user
    var existingEntry models.ArmstrongNumber
    if err := config.DB.Where("user_id = ? AND number = ?", data.UserID, data.Number).First(&existingEntry).Error; err == nil {
        // If the entry exists, return a conflict response
        w.WriteHeader(http.StatusConflict)
        json.NewEncoder(w).Encode(map[string]string{
            "error": "Number already checked by this user",
        })
        return
    }

    // Add new entry to the database
    newEntry := models.ArmstrongNumber{
        UserID:    data.UserID, // Store the user_id as string
        Number:    data.Number,
        Result:    data.Result, // Store the result (positive or negative)
        CreatedAt: time.Now(),
    }

    if err := config.DB.Create(&newEntry).Error; err != nil {
        http.Error(w, "Error saving entry", http.StatusInternalServerError)
        return
    }

    // Send a successful response with the new entry
```

```go
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(newEntry)
}
```

Listing 6: GoLang Armstrong Verification Handler

- **GET /getusernumbers**: Returns the details of the numbers entered by the users like result and timesstamp of when the number was entered.

```go
package handlers

import (
        "backend/config"
        "encoding/json"
        "log"
        "net/http"
        "strconv"
)

// NumberDetails represents a single entry in the armstrong_numbers table
type NumberDetails struct {
        ID        uint    `json:"id"`
        Number    int     `json:"number"`
        Result    string  `json:"result"`
        CreatedAt string  `json:"created_at"`
}

// PaginatedNumberDetailsResponse contains paginated number details
type PaginatedNumberDetailsResponse struct {
        NumberDetails []NumberDetails `json:"number_details"`
        TotalCount    int64           `json:"total_count"`
        Page          int             `json:"page"`
        PageSize      int             `json:"page_size"`
        TotalPages    int             `json:"total_pages"`
}

// GetUserNumbers retrieves paginated numbers entered by the user
func GetUserNumbers(w http.ResponseWriter, r *http.Request) {
    log.Println("Received request to fetch user numbers")

    // Get user ID from query parameter
    userID := r.URL.Query().Get("user_id")
    log.Printf("Received user_id: %s", userID)

    if userID == "" {
        log.Println("Missing user ID")
        http.Error(w, `{"error":"Missing user ID"}`, http.StatusBadRequest)
        return
    }

    // Parse pagination parameters
    pageStr := r.URL.Query().Get("page")
    pageSizeStr := r.URL.Query().Get("page_size")
    filter := r.URL.Query().Get("filter") // New filter parameter

```

```go
    // Default values
    page := 1
    pageSize := 5

    // Convert page and page_size to integers
    if pageStr != "" {
        if p, err := strconv.Atoi(pageStr); err == nil && p > 0 {
            page = p
        }
    }
    if pageSizeStr != "" {
        if ps, err := strconv.Atoi(pageSizeStr); err == nil && ps > 0 && ps <= 100 {
            pageSize = ps
        }
    }

    // Calculate offset
    offset := (page - 1) * pageSize

    // Prepare response struct
    var response PaginatedNumberDetailsResponse

    // Query for total count
    var totalCount int64
    countQuery := "SELECT COUNT(*) FROM armstrong_numbers WHERE user_id = ?"
    countParams := []interface{}{userID}

    // Apply filter if present
    if filter == "positive" || filter == "negative" {
        countQuery += " AND result = ?"
        countParams = append(countParams, filter)
    }

    config.DB.Raw(countQuery, countParams...).Scan(&totalCount)

    // Calculate total pages
    totalPages := (int(totalCount) + pageSize - 1) / pageSize

    // Query for paginated data
    var numberDetails []NumberDetails
    dataQuery := `
        SELECT
            id,
            number,
            result,
            created_at
        FROM armstrong_numbers
        WHERE user_id = ?
    `
    dataParams := []interface{}{userID}

    if filter == "positive" || filter == "negative" {
        dataQuery += " AND result = ?"
        dataParams = append(dataParams, filter)
    }

```

```go
    dataQuery += " ORDER BY created_at DESC LIMIT ? OFFSET ?"
    dataParams = append(dataParams, pageSize, offset)

    err := config.DB.Raw(dataQuery, dataParams...).Scan(&numberDetails).Error

    if err != nil {
        log.Printf("Error querying armstrong_numbers table for user_id %s: %v", userID, err)
        http.Error(w, `{"error":"Internal server error"}`, http.StatusInternalServerError)
        return
    }

    // Populate response
    response = PaginatedNumberDetailsResponse{
        NumberDetails: numberDetails,
        TotalCount:    totalCount,
        Page:          page,
        PageSize:      pageSize,
        TotalPages:    totalPages,
    }

    // Send the response
    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(response); err != nil {
        log.Printf("Error encoding response: %v", err)
        http.Error(w, `{"error":"Internal server error"}`, http.StatusInternalServerError)
    }
}
```

Listing 7: GoLang GetUserNumbers Handler with Pagination

4.3 CORS Handling

Cross-Origin Resource Sharing (CORS) is a security feature implemented by browsers to restrict how resources on a web server can be requested from another domain. The backend is configured with the ‘github.com/rs/cors‘ middleware to allow the frontend to make requests from a different origin.

```go
corsHandler := cors.New(cors.Options{
    AllowedOrigins:   []string{"http://localhost:3000"},
    AllowedMethods:   []string{"GET", "POST", "PUT", "DELETE", "OPTIONS"},
    AllowedHeaders:   []string{"Content-Type", "Authorization"},
    AllowCredentials: true,
}).Handler(r)
```

Listing 8: CORS Handler Setup in GoLang

5 Challenges and Solutions Implemented

5.1 User Identification Across Pages

Challenge: Maintaining user identification across different pages to ensure the user is authenticated throughout their session.

Solution: Implemented a `userContext` using the Context API in React. The `userContext` holds the user's credentials such as `name`, `user_id`, and `email`, and is accessible throughout the application to manage user authentication and authorization across different pages.

5.2 Preventing Unauthorized Access Through URL Manipulation

Challenge: A user trying to access the home page by manipulating the URL without logging in could bypass the authentication process.

Solution: Implemented a `PrivateRoute` component in React. This component checks if the user's `name`, `user_id`, and `email` fields are not null in the `userContext`. If any of these fields are null, the user is redirected to the login page.

5.3 Handling Invalid Inputs

Challenge: Ensuring valid user input, particularly for fields such as email and numeric input, to prevent issues during data processing.

Solutions:

- **Email Validation:** Implemented regular expressions (regex) to validate the email pattern, ensuring it follows a standard format (e.g., `user@example.com`).
- **Numeric Input Validation:** Implemented checks to ensure that only numbers are entered for fields that require numeric input, such as Armstrong number verification.

5.4 Handling Large Tables with Pagination

Challenge: Displaying large datasets such as Armstrong numbers in a user-friendly manner without overwhelming the interface or degrading performance.

Solution: Implemented pagination for the Armstrong numbers table. The table is split into pages, with each page displaying a limited number of records, allowing users to navigate through large datasets efficiently. Pagination is implemented both on the frontend (React) and backend (Go) to ensure smooth user experience and server efficiency.

6 Testing and Performance

6.1 Testing

To ensure the functionality and reliability of the application, manual testing for all features was conducted. Each feature was tested to verify its behavior under various scenarios, ensuring correct data processing and interaction between different components. The testing process involved checking:

- User registration and login flow.
- Armstrong number verification and storage.
- Pagination functionality.
- Input validation for fields like email and numbers.
- Proper routing and navigation between pages.

6.2 Performance Optimizations

To ensure optimal performance, the following optimizations were implemented:

- **Efficient API Pagination Logic:** Pagination was optimized to handle large data efficiently. The backend queries were optimized to return only the required set of results based on the current page and page size, thus reducing the amount of data transferred and improving API response times.

7 Future Enhancements

The following enhancements are planned for future versions of the project to improve functionality, scalability, and user experience:

- **Email Verification:** Implement email verification during user registration to ensure valid email addresses. A verification email will be sent to the user to confirm their email before they can log in.
- **Password Authentication:** Introduce password authentication for user accounts to improve security. This will involve adding password fields during user registration and login, using secure hashing algorithms to store passwords in the database.
- **Enhanced Pagination and Filtering:** Enhance the pagination and filtering logic to support more complex queries, such as sorting by multiple fields or filtering by additional parameters like date range or result type (positive/negative).
- **Internationalization and Localization:** Implement internationalization (i18n) to support multiple languages. This would involve translating the user interface and handling various date and number formats based on the user's region.

8 Conclusion

In this project, a robust web application was developed to verify Armstrong numbers, manage user data, and display results efficiently. Key features of the project include:

- **User Context:** Implemented userContext using the Context API to manage user details across different pages, ensuring a seamless experience.
- **Protected Routes:** Utilized private routes to restrict access to certain pages based on user credentials, enhancing security by preventing unauthorized access.
- **Pagination:** Implemented pagination to handle large datasets efficiently, ensuring smooth navigation through large tables of Armstrong numbers.
- **CORS Implementation:** Configured CORS settings to ensure secure communication between the frontend and backend, allowing specific origins and enabling safe cross-origin requests.
- **Input Validation:** Applied input validation techniques like regex for email verification and checks for valid numeric inputs, ensuring data integrity and preventing errors.

These features collectively ensure that the application is user-friendly, secure, and capable of handling large amounts of data efficiently, providing a solid foundation for further enhancements and scalability.

Appendix: UI Screenshots

Login Page

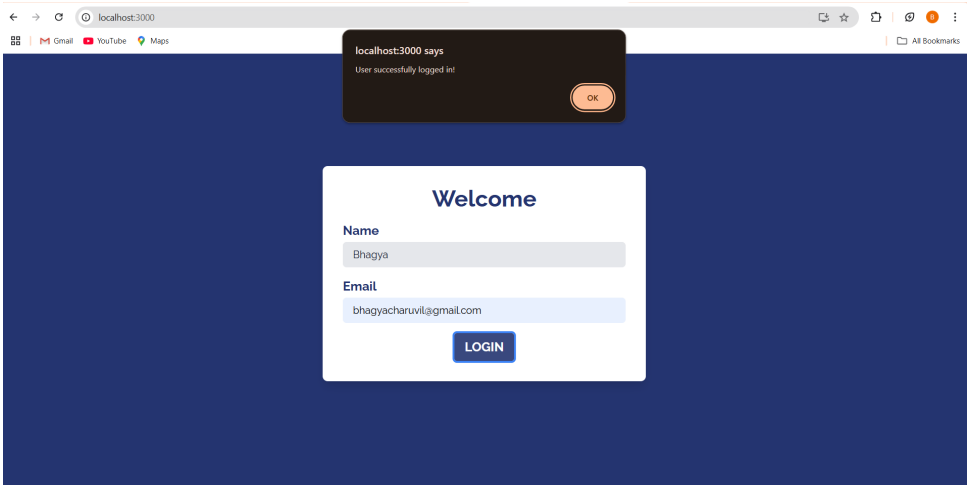

Figure 1: Login Page

Home Page

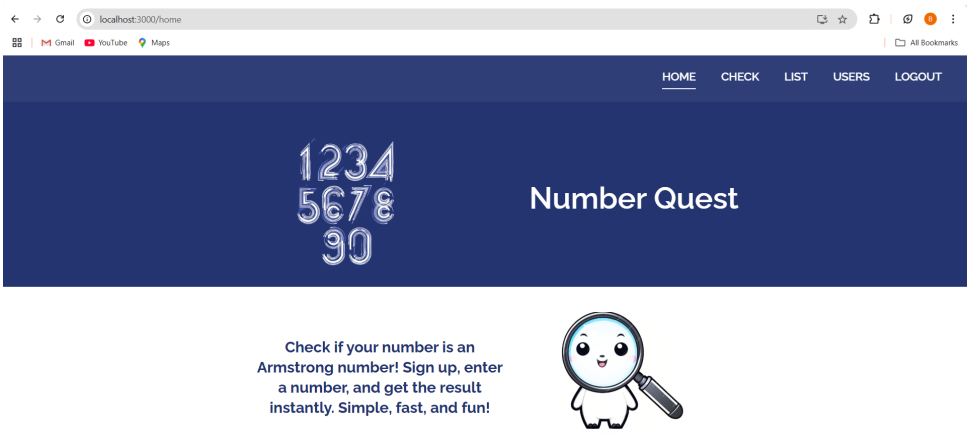

Figure 2: Landing Page

Verification Page

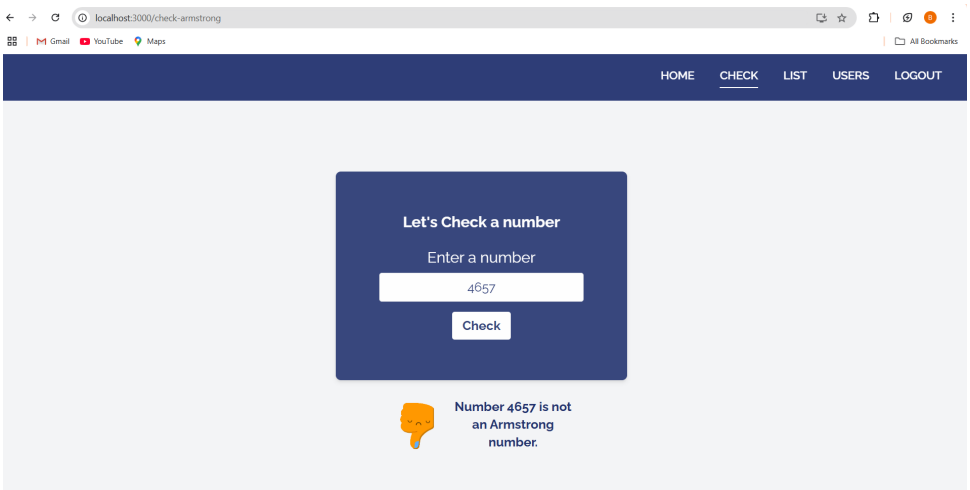

Figure 3: Verification page

User Details Page

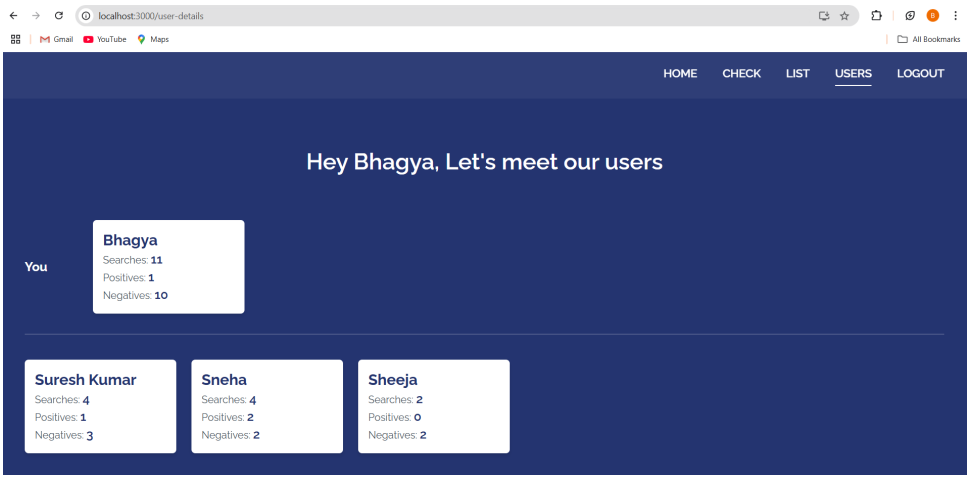

Figure 4: Page showing user details

Number Details Page

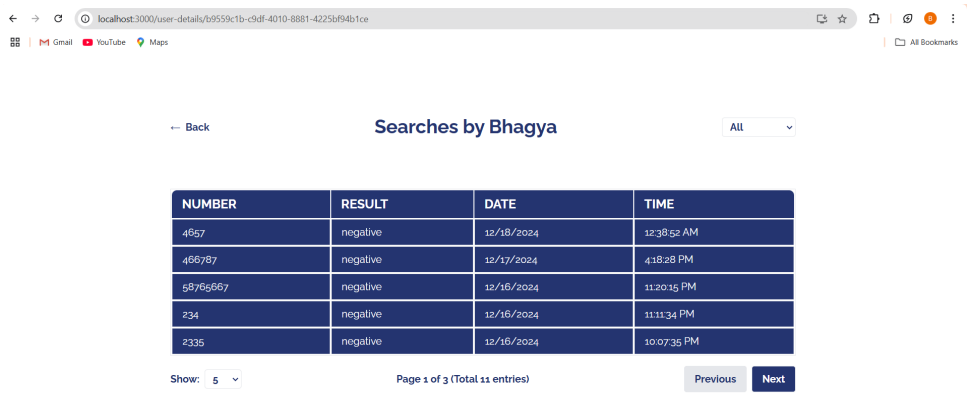

Figure 5: Table showing details of numbers entered by a user with pagination

Armstrong Nos within range page

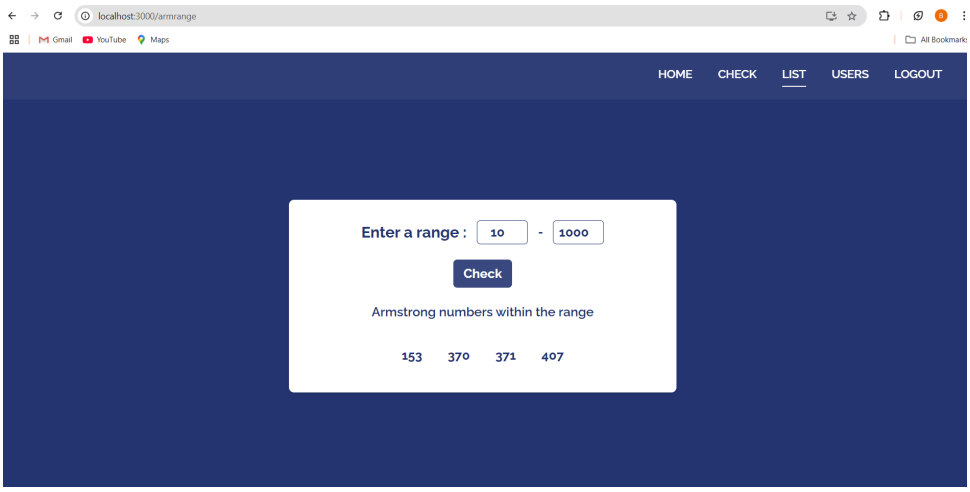

Figure 6: Page that returns the list of armstrong nos within range 10-1000