Player Registration and Authentication System

Project Overview

The goal is to create a **Player Registration and Authentication system** using Nginx as a reverse proxy to route RESTful API requests to an Authentication Service. The service will manage multiple authentication options and integrate security mechanisms like CAPTCHA, rate limiting, and email verification. A PostgreSQL database will store user credentials securely. The client will use pure JavaScript for interacting with the system.

High-Level Architecture

1. **Nginx (Reverse Proxy)**:

   - Acts as the entry point for client requests.
   - Routes RESTful API requests to the Authentication Service.
   - Implements rate limiting and forwards traffic securely using HTTPS.

2. **Authentication Service**:

   - A containerized service written in pure Ruby that handles:
     - Player registration (username/password).
     - Authentication (username/password, JWT, OAuth).
     - Token generation and validation.
     - Email verification.
     - Email notifications for password resets.
     - Token refresh for JWT to extend session duration.
   - Interacts with the PostgreSQL database.

3. **PostgreSQL Database**:

- Stores user data securely, including hashed passwords.
- Provides a scalable, reliable backend for user management.

4. **Client (JavaScript)**:

- Pure JavaScript to interact with the system via RESTful APIs.
- Implements registration and login forms.
- Displays a simple page with all the player's information stored in the database after successful login.
- Handles authentication flows (e.g., OAuth redirects).

Tasks Breakdown

1. Set Up the Environment

1. **Tools**:

- Use Docker Compose to manage containerized services.
- Choose a lightweight JavaScript library for the frontend if needed (e.g., vanilla JS or fetch API).

2. **Services**:

- Create Docker containers for:
  - Nginx
  - Authentication Service (Ruby)
  - PostgreSQL Database

---

2. Nginx Configuration

1. **Reverse Proxy**:

   - Route API requests (e.g., `/api/auth/register`, `/api/auth/login`) to the Authentication Service.

2. **Rate Limiting**:

   - Use the Nginx `limit_req` module to enforce rate limiting for API endpoints.

   Example Policy:

   - Limit registration attempts to 20 requests per minute per IP.

3. **Security**:

   - Configure HTTPS with self-signed certificates for local testing.

---

3. Authentication Service

1. **API Endpoints**:

   - **POST /api/auth/register**:
     - Handles player registration with username/password.
     - Validates input and checks for username uniqueness.
     - Sends an email verification link upon successful registration.
   - **POST /api/auth/login**:
     - Authenticates the user and generates tokens.
     - Supports:
       - JWT-based token generation and validation.
       - OAuth redirection for external login providers (e.g., Google).
   - **POST /api/auth/validate**:

  - Validates JWT or OAuth tokens to confirm authentication.
- **GET /api/auth/verify-email**:
  - Confirms email verification through a unique token sent via email.
- **POST /api/auth/reset-password**:
  - Sends a password reset email with a unique token.
- **POST /api/auth/update-password**:
  - Updates the password using the reset token.
- **GET /api/auth/player-info**:
  - Returns all stored information about the authenticated player.

2. **Password Hashing**:

   - Use bcrypt or Argon2 for securely storing passwords.

3. **Security Features**:

   - **CAPTCHA**:
     - Implement Google reCAPTCHA for registration and login endpoints.
   - **Error Messages**:
     - Return generic error messages for authentication failures to avoid revealing sensitive information.

4. **Email Verification**:

   - Send a unique link to the user's email address upon registration.
   - Store the verification token and its expiration in the database.
   - Activate the account only after the user verifies their email.

5. **Email Notifications for Password Resets**:

   - Implement an endpoint to trigger a password reset email.
   - Generate a unique token for password resets and store it in the database with an expiration time.

6. **Token Refresh for JWT**:

   - Implement an endpoint to refresh JWTs when nearing expiration.
   - Validate the existing token and issue a new one with an extended duration.

7. **Testing Multiple Authentication Methods**:

   - **Username/Password**:
     - Direct login with stored credentials.
   - **JWT**:
     - Stateless token-based authentication.
   - **OAuth**:
     - Use external providers for authentication (e.g., Google OAuth).

---

4. PostgreSQL Database

1. **Schema**:

   - Create a `users` table with the following fields:
     - `id` (Primary Key, UUID)
     - `username` (Unique, VARCHAR)
     - `email` (Optional, Unique)
     - `hashed_password` (VARCHAR)
     - `email_verified` (BOOLEAN, default: false)
     - `email_verification_token` (VARCHAR, nullable)
     - `email_verification_token_expiry` (TIMESTAMP, nullable)
     - `reset_password_token` (VARCHAR, nullable)
     - `reset_password_token_expiry` (TIMESTAMP, nullable)
     - `created_at`, `updated_at` (Timestamps)

2. **Integration**:

   - Use an ORM or raw SQL queries in the Authentication Service.

---

5. Frontend (Client)

1. **Landing Page**:

   - Implement a basic landing page that serves as the entry point for the application.
   - Provide links to the registration and login pages.

2. **Registration and Login Forms**:

   - Build simple HTML forms for:
     - Username/password registration.
     - Login via username/password.
     - OAuth login using "Login with Google" or similar buttons.

3. **RESTful API Calls**:

   - Use `fetch()` to communicate with the Authentication Service.

4. **Token Handling**:

   - Store JWT tokens securely in local storage or session storage.

5. **CAPTCHA Integration**:

   - Display Google reCAPTCHA widgets on the frontend for registration/login.

6. **Email Verification Flow**:

   - Redirect the user to a verification page after registration.
   - Confirm email verification by interacting with the `/api/auth/verify-email` endpoint.

7. **Player Information Page**:

   - After successful login, fetch and display all information about the player from the database on a dedicated page.

8. **Password Reset Flow**:

   - Provide a form to request a password reset via email.
   - Implement another form to update the password using the reset token.

Testing Plan

1. Functional Tests

- Test API endpoints with `curl` or a REST client like Postman:
  - Ensure registration and login endpoints behave as expected.
  - Verify tokens are generated and validated correctly.
  - Confirm email verification flow works properly.
  - Validate the `/api/auth/player-info` endpoint retrieves accurate player data.
  - Test password reset flow for both token generation and password update.
  - Test JWT refresh functionality to ensure extended session duration.

2. Security Tests

- Attempt brute-force attacks to confirm rate limiting works.

- Test password hashing with long or weak passwords.
- Validate CAPTCHA integration blocks automated bots.
- Confirm email verification prevents unverified accounts from logging in.
- Test password reset tokens for expiration and misuse.

3. OAuth Testing

- Use a sandbox account (e.g., Google Developer Console) to test OAuth login flow.

4. Frontend Testing

- Verify user flows in a browser:
  - Registration, login, email verification, password reset, and OAuth flows should complete without errors.
  - Token expiration should redirect the user appropriately.
  - Confirm player information is displayed accurately after login.