



# **Electronic Arts**

## **Junior Full-Stack Software Engineer**

### **Coding Challenge Documentation**

***Prepared by:***

Shijie Gan

[shijiegan.gs@gmail.com](mailto:shijiegan.gs@gmail.com)

+6012-6383016

## TABLE OF CONTENTS

1.	Introduction .....	1
2.	Requirements Overview.....	2
2.1.	Original Requirements & Assumptions (from Specification).....	2
2.2.	How This Implementation Meets the Requirements .....	3
3.	System Design .....	5
3.1.	Sequence Diagram of Registration .....	5
3.2.	Sequence Diagram of Login.....	6
3.3.	Data Models .....	7
3.4.	API Endpoints Summary.....	9
4.	Implementation Summary .....	12
4.1.	Authentication Flow Overview .....	12
4.2.	Security Implementation.....	12
5.	Testing .....	14
5.1.	Testing Overview .....	14
5.2.	Test Structure.....	15
6.	How to Run Locally.....	16
7.	Step-by-Step Guidelines & Snapshots .....	17
7.1.	Registration .....	17
7.2.	Login .....	20

## **1. INTRODUCTION**

This document outlines the design and implementation strategy for a user login system developed using React for the frontend, TypeScript with Express for the backend, and Supabase PostgreSQL as the storage solution. This project was created as part of the Electronic Arts Junior Full-Stack Software Engineer coding challenge.

The goal of the project is to build a secure, functional, and extensible login feature that supports user registration and authentication via email/username, password, and a dynamic code sent to the user's email. The system is designed with a strong focus on best practices, clean architecture, and thorough testing to ensure a production-ready experience.

This document was prepared by Gan Shijie as part of the Electronic Arts coding challenge.

## 2. REQUIREMENTS OVERVIEW

This section outlines the original coding challenge requirements provided by Electronic Arts and how they are addressed or extended in this implementation.

### 2.1. Original Requirements & Assumptions (from Specification)

As stated in the challenge:

- The user login page is the focus of the challenge.
- Users must be able to register and log in using username, password, and a dynamic code received by phone.
- You are responsible for the full implementation, including frontend, backend, and storage.
- The following technical and functional expectations apply:
  - You may design the UI/UX as you see fit.
  - Security is sensitive due to the nature of login functionality.
  - Unit tests are required.
  - TypeScript is preferred for the backend, though Java is acceptable.
  - React is preferred (but optional) for the frontend.
  - SQLite or any other easy-to-switch storage may be used.
  - Only the user creation and login features need to be production-ready; other website functions are not in scope.
  - Performance is not a priority—focus is on functionality.
  - You can assume the user has already received the dynamic code during login.
  - You can assume secure communication between client and server.

Submissions must include:

- All source code and documentation in a single ZIP file.
- A README file for quick local setup and testing.
- Optional but encouraged: design documentation, diagrams, and UI snapshots.

The submission will be evaluated on:

- Functionality and completeness
- Best practices and clean code style
- Flexibility and extensibility
- High test coverage
- Quality documentation

## 2.2. How This Implementation Meets the Requirements

### User Registration:

- Users register using email, username, and password.
- Upon registration, an OTP is generated in the backend, stored in the database, and logged to the console (simulating email delivery).
- The user must verify their email using the OTP.
- Once verified, the user is automatically logged in.

### Login Process:

- Users can log in using either their email or username, along with their password.
- If the user's email is not yet verified, the login process initiates the same email verification flow as during registration. An OTP is generated and logged to the backend console, and upon successful verification, the user is automatically logged in.
- If the email is already verified, the system requires the user to enter a login OTP code as a second step of authentication. An OTP is generated, stored in the backend, and logged to the console.

### Dynamic Code Handling:

- The original spec assumes OTP is sent via phone. This implementation uses email OTP, simulated by backend console logs.

### Security:

- Passwords are securely hashed before being stored in the database.
- OTP verification is required for both email validation and as a second authentication factor during login for verified users, enhancing login security.
- JWT-based authentication is used to secure protected routes and manage user sessions after login.
- All OTPs are securely generated and stored in the database. Email delivery is not implemented but simulated via console output for testing. Although actual email delivery is not implemented, the OTP system is fully functional and designed in a way that it could be easily extended to support real email delivery.

### Technology Stack:

- Frontend: React
- Backend: Express.js with TypeScript

- Database: Supabase PostgreSQL, allowing easy migration and relational queries

#### Testing:

- Jest unit tests are implemented for the backend, covering all major features and API endpoints.

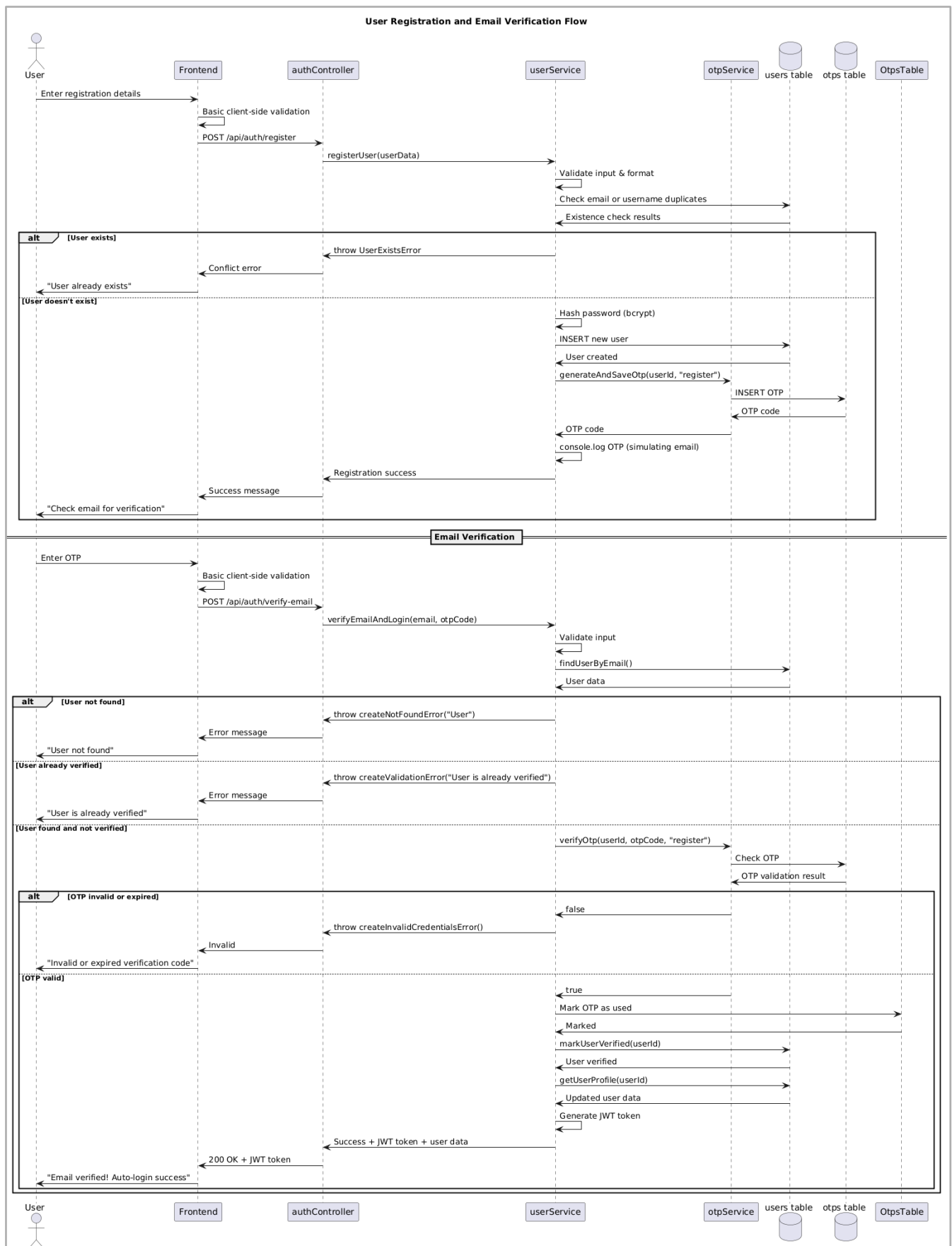
#### Documentation:

- This document includes design, system structure, and run instructions.
- Screenshots of UI pages are also included.

### 3. SYSTEM DESIGN

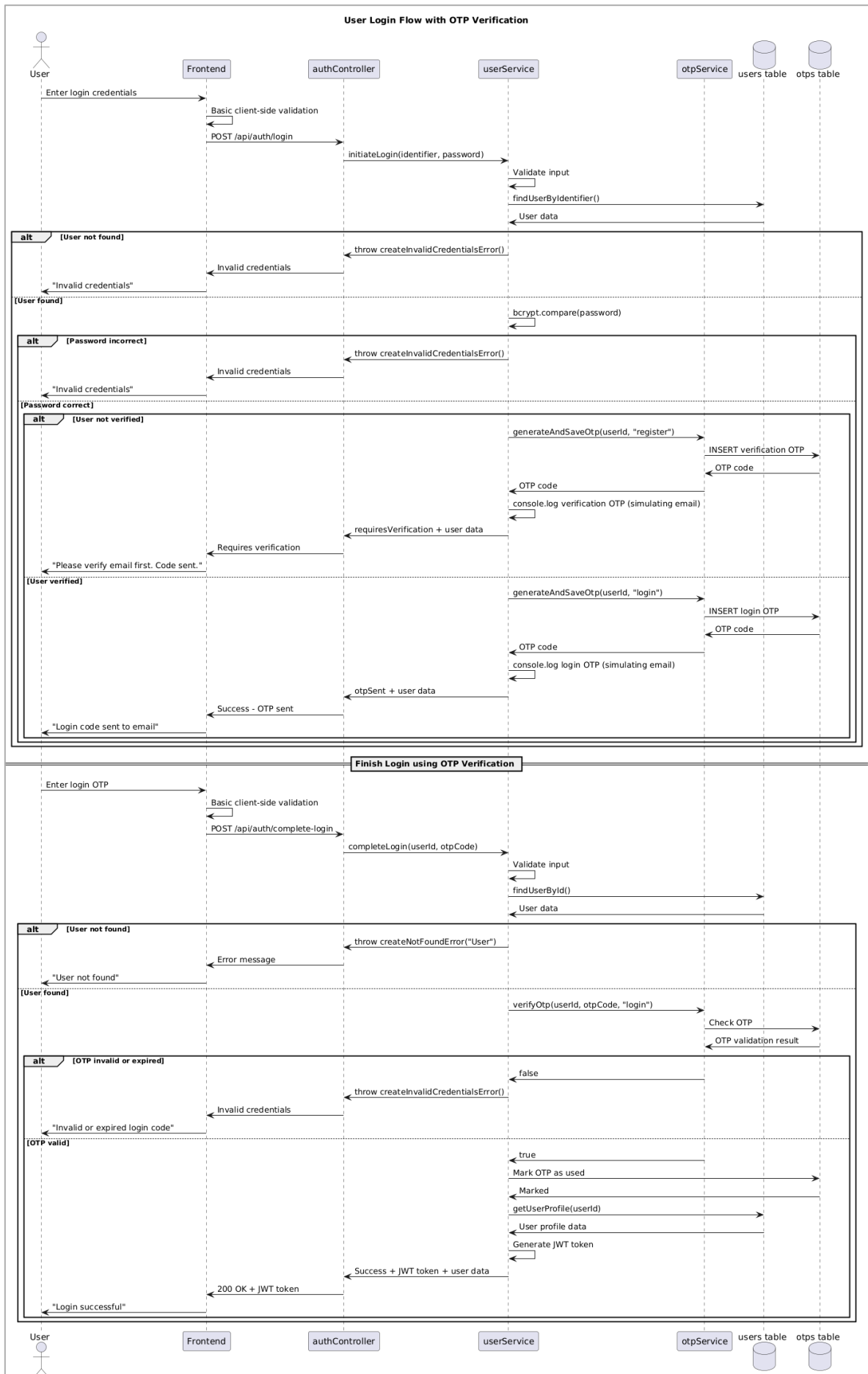
### 3.1. Sequence Diagram of Registration

The sequence diagram below illustrates the flow of **user registration and email verification**.



## 3.2. Sequence Diagram of Login

The sequence diagram below illustrates the flow of **user login** and **login code authentication**.

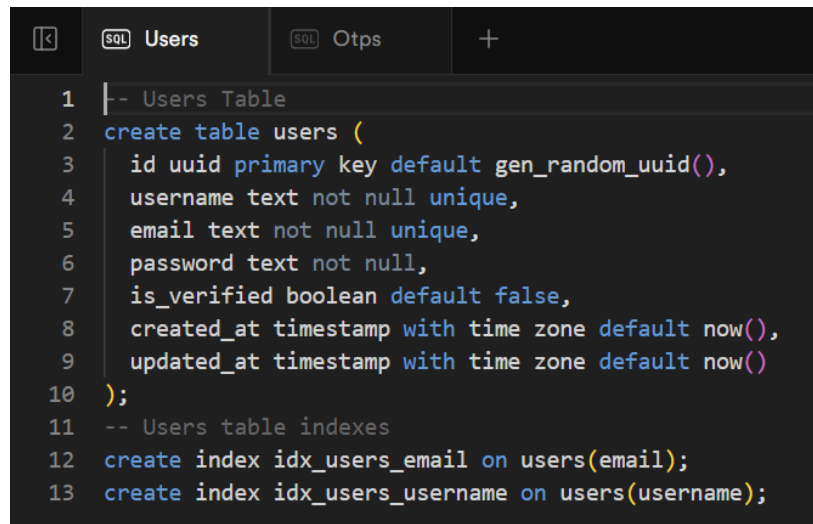




### 3.3. Data Models

This system uses two primary tables: users and otps, designed to support user registration, email verification, and secure login with OTP.

#### Users Table

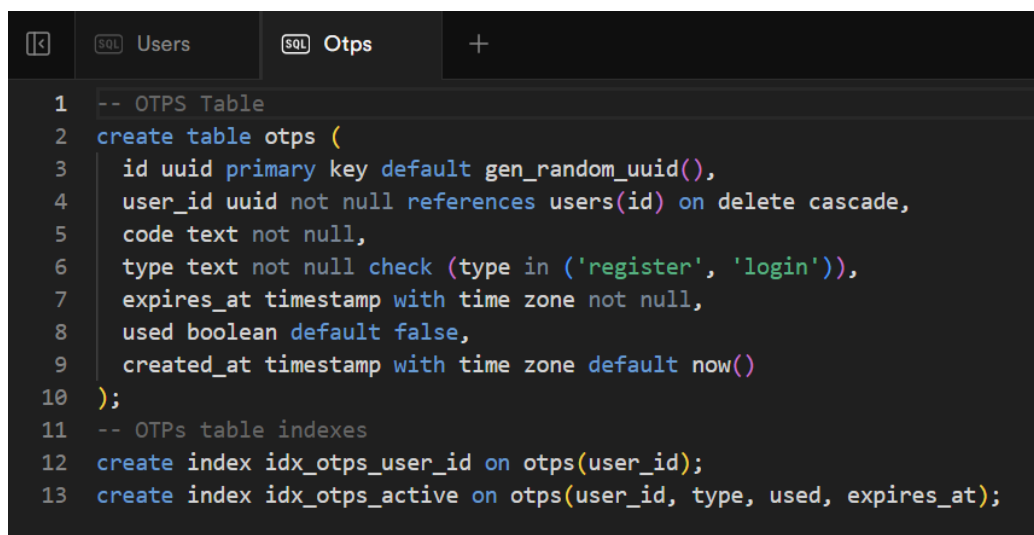
A screenshot of a SQL editor interface showing the creation of the 'Users' table. The interface has tabs for 'Users' and 'Otps', and a '+' button. The code is as follows:

```
1  -- Users Table
2  create table users (
3      id uuid primary key default gen_random_uuid(),
4      username text not null unique,
5      email text not null unique,
6      password text not null,
7      is_verified boolean default false,
8      created_at timestamp with time zone default now(),
9      updated_at timestamp with time zone default now()
10 );
11 -- Users table indexes
12 create index idx_users_email on users(email);
13 create index idx_users_username on users(username);
```

- The users table stores essential account information:
- id: Primary key (UUID).
- username and email: Unique identifiers for login and communication.
- password: Hashed password (securely stored).
- is\_verified: Indicates whether the user's email has been verified.
- Timestamps: created\_at and updated\_at track user lifecycle.

Indexes are added on email and username to optimize lookup and enforce uniqueness.

#### OTPs Table

A screenshot of a SQL editor interface showing the creation of the 'OTPs' table. The interface has tabs for 'Users' and 'Otps', and a '+' button. The code is as follows:

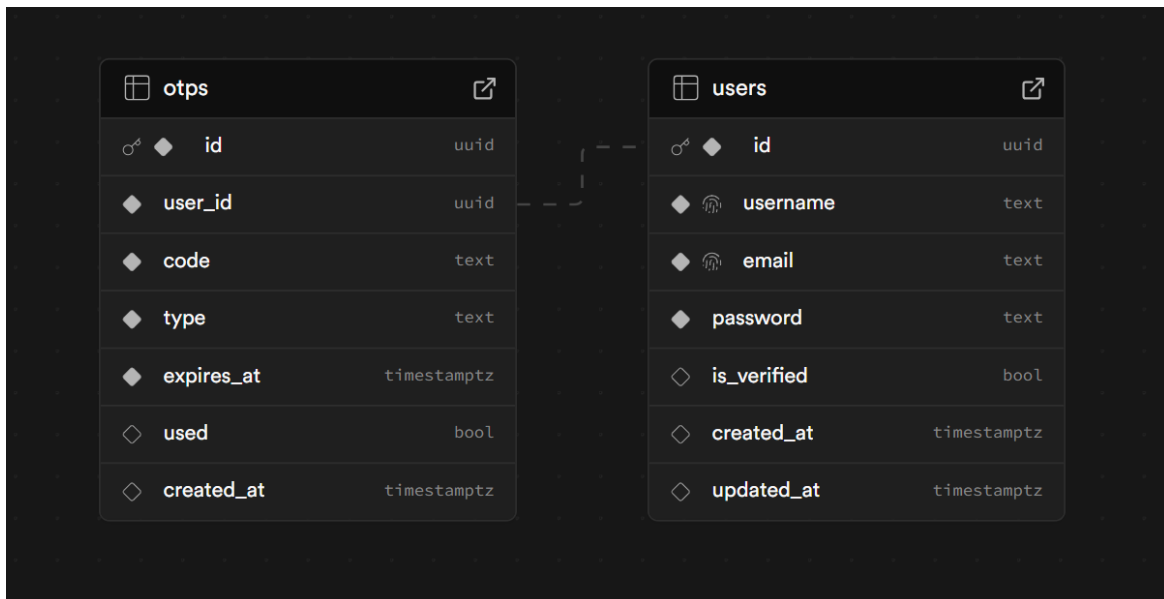
```
1  -- OTPS Table
2  create table otps (
3      id uuid primary key default gen_random_uuid(),
4      user_id uuid not null references users(id) on delete cascade,
5      code text not null,
6      type text not null check (type in ('register', 'login')),
7      expires_at timestamp with time zone not null,
8      used boolean default false,
9      created_at timestamp with time zone default now()
10 );
11 -- OTPs table indexes
12 create index idx_otp_user_id on otps(user_id);
13 create index idx_otp_active on otps(user_id, type, used, expires_at);
```

- The otps table handles time-limited one-time codes used for email verification and login:
- id: Primary key (UUID).
- user\_id: Foreign key referencing the associated user.
- code: The actual OTP value.
- type: Either 'register' or 'login', to distinguish the OTP purpose.
- expires\_at: Timestamp for expiration.
- used: Boolean flag to prevent reuse.
- created\_at: Timestamp for OTP creation.

Indexes support efficient queries for active OTPs by user and purpose.

### Entity Relationship Overview

- A user can have multiple OTPs, each tied to a specific purpose (register or login).
- OTPs are linked to users via the user\_id foreign key, with ON DELETE CASCADE to clean up associated OTPs when a user is removed.



### 3.4. API Endpoints Summary

The authentication system provides secure user registration, email verification, and two-step login using OTP. The following endpoints define the core backend API functionality.

#### POST /api/auth/register

Description: Registers a new user account.

Request Body:

```
{
  email: string;
  username: string;
  password: string;
}
```

Response: 201 Created

```
{
  success: boolean;
  data: {
    userId: string;
    message: string;
  }
}
```

---

#### POST /api/auth/verify-email

Description: Verifies the user's email address using OTP. Automatically logs the user in upon successful verification.

Request Body:

```
{
  email: string;
  otpCode: string;
}
```

Response: 200 OK

```
{
  success: boolean;
  data: {
    token: string;
    user: object;
    message: string;
  }
}
```

---

### POST /api/auth/login

Description: Initiates the login process. If the user's email is verified, an OTP is generated for two-step authentication.

Request Body:

```
{
  identifier: string; // email or username
  password: string;
}
```

Response: 200 OK

```
{
  success: boolean;
  data: {
    userId: string;
    message: string;
  }
}
```

---

### POST /api/auth/complete-login

Description: Completes the login process by verifying the OTP sent to the user's email.

Request Body:

```
{
  userId: string;
  otpCode: string;
}
```

Response: 200 OK

```
{
  success: boolean;
  data: {
    token: string;
    user: object;
    message: string;
  }
}
```

---

### POST /api/auth/resend-verification

Description: Resends the email verification OTP.

Request Body:

```
{
  email: string;
}
```

Response: 200 OK

```
{
  success: boolean;
  data: {
    message: string;
  }
}
```

---

### POST /api/auth/resend-login-otp

Description: Resends the login OTP email.

Request Body:

```
{
  email: string;
}
```

Response: 200 OK

```
{
  success: boolean;
  data: {
    message: string;
  }
}
```

---

### GET /api/auth/user

Description: Returns authenticated user information. Requires a valid JWT token.

Headers:

```
`Authorization: Bearer <token>`
```

Response: 200 OK

```
{
  success: boolean;
  data: {
    userId: string;
    email: string;
    username: string;
    isEmailVerified: boolean;
    createdAt: string;
    updatedAt: string;
  }
}
```

## 4. IMPLEMENTATION SUMMARY

### 4.1. Authentication Flow Overview

The authentication system implements a secure, multi-phase flow combining credential validation, email ownership verification, and two-factor login via OTP. The goal is to protect user accounts without compromising usability.

#### Registration Flow:

Register → Email Verification → Auto Login → JWT Token

1. User submits email, username, and password.
2. Server validates uniqueness and securely hashes the password.
3. An OTP is generated, stored in the database, and logged (simulating email delivery).
4. User verifies their email using the OTP.
5. Upon successful verification, the user is automatically logged in with a JWT token.

#### Login Flow:

Login → Verified ? Login OTP : Email Verification → Finish Login → JWT Token

1. User logs in using either email or username with their password.
2. If the email is not yet verified, the system initiates email verification as in the registration flow.
3. If the email is verified, a login OTP is generated and logged.
4. The user submits the OTP to complete login.
5. A JWT token is issued for authenticated access.

### 4.2. Security Implementation

Security is treated as a top priority, with safeguards at every layer of the system.

#### Multi-Layer Authentication

- Password-Based Authentication: User credentials are validated with securely hashed passwords.
- Email Verification: Ensures the user owns the email address provided at registration.
- Two-Factor Authentication (2FA): All verified users must enter an OTP during login.
- JWT Token Authorization: Grants access to protected routes without server-side session storage.

## **OTP Security Measures**

- Time-Limited Validity: OTPs expire after a short window (e.g., 5–15 minutes).
- One-Time Use: OTPs are marked as used after successful verification.
- Purpose-Specific Flow: OTPs are categorized as register or login, ensuring proper handling.
- Pluggable Email Delivery: Designed for easy integration with secure email providers.

## **JWT Token Security**

- Digitally Signed Tokens: Prevent token tampering.
- Token Expiry: Limits exposure if compromised.
- Authorization Header: All protected routes require Bearer <token>.
- Middleware Enforcement: Centralized token validation across routes.

## **Input Validation & Error Handling**

- Input Sanitization: Applied at the service layer to prevent injection attacks.
- Centralized Error Handling: Reduces code repetition and risk of info leakage.
- Consistent Responses: All errors return standard formats.
- No Sensitive Exposure: Internal details are hidden from end-users.

## **Access Control**

- Public Routes: Registration, login initiation, OTP verification.
- Protected Routes: Authenticated user actions (e.g., profile access).
- Strict Authorization: JWT is mandatory for all protected endpoints.
- Least Privilege Principle: Users can only access their own data.

## **Session Management**

- Stateless Design: No server-stored sessions, reducing attack surface.
- Token-Based Authorization: Sessions controlled through JWT expiry.
- Auto-Logout: Token expiration enforces re-authentication.
- Device Isolation: Each login generates a unique token.

## **Email Security**

- Email Ownership Verification: Ensures only legitimate users can activate accounts.
- OTP Over Email: Verification codes are sent through a secure (simulated) email process.
- Resend Mechanism: OTPs can be resent to improve usability.

## 5. TESTING

### 5.1. Testing Overview

The backend was tested using both **automated unit tests (Jest)** and manual testing via local development. This ensures correctness, robustness, and reliability across all authentication flows.

- 21 unit tests implemented with Jest
- Overall coverage: 85.75%
- Manually verified using the frontend and API in a local dev environment

File	% Stmts	% Branch	% Funcs	% Lines
All files	87.46	74.86	87.14	85.75
controllers	89.47	100	85.71	88
authController.ts	89.47	100	85.71	88
dao	92.75	71.42	91.66	91.22
otpDao.ts	100	87.5	100	100
userDao.ts	90.38	67.64	88.88	88.37
middleware	84.84	75	100	83.33
authMiddleware.ts	84.61	100	100	83.33
errorHandler.ts	85	66.66	100	83.33
services	86.06	81.51	88.88	85.06
otpService.ts	95.23	90.9	100	94.73
userService.ts	84.72	80.55	85.71	83.7
utils	83.72	10	72.72	78.12
errorUtils.ts	76.66	0	62.5	68.18
jwtUtils.ts	100	50	100	100
otpUtils.ts	100	100	100	100
Test Suites: 1 passed, 1 total				
Tests: 21 passed, 21 total				
Snapshots: 0 total				
Time: 10.778 s, estimated 11 s				
Ran all test suites.				



## 5.2. Test Structure

Basic Flow Tests (18 tests):

- Registration Tests (3): Missing fields, duplicate email, successful registration
- Email Verification Tests (3): Missing fields, incorrect OTP, verification scenarios
- Login Tests (3): Missing fields, wrong password, login scenarios
- Complete Login Tests (3): Missing fields, wrong OTP, complete login flow
- Resend OTP Tests (3): Missing email, non-existent email, resend logic
- Get User Data Tests (2): Invalid or missing token, user retrieval

Error Handling Tests (2 tests):

- Handles 404 routes and malformed requests gracefully

Advanced Coverage Tests (2 tests):

- Comprehensive Flow Test (1): Full flow with real OTP capture
- OTP Validation Test (1): Validates expiration, reuse, and purpose type

Overall coverage:

- 85.75%

## 6. HOW TO RUN LOCALLY

For full setup and usage guidelines, refer to the README file. Below is a concise summary to quickly set up and run the application locally.

1. Extract the project zip file and open the project folder.
2. Install dependencies:

- Backend:

```
cd backend  
npm install
```

- Frontend:

```
cd ../frontend  
npm install
```

3. Start the backend server:

```
cd backend  
npm run dev
```

4. Start the frontend server (in a new terminal):

```
cd frontend  
npm run dev
```

5. Open your browser and go to <http://localhost:5173> to use the application.
6. Refer to the snapshots in the next section for a step-by-step guide to using the application in your browser.

## 7. STEP-BY-STEP GUIDELINES & SNAPSHOTS

### 7.1. Registration

1 Navigate to <http://localhost:5173/>

Guide Me

Welcome to Electronic Arts

Your gateway to the EA universe.

Login

Register



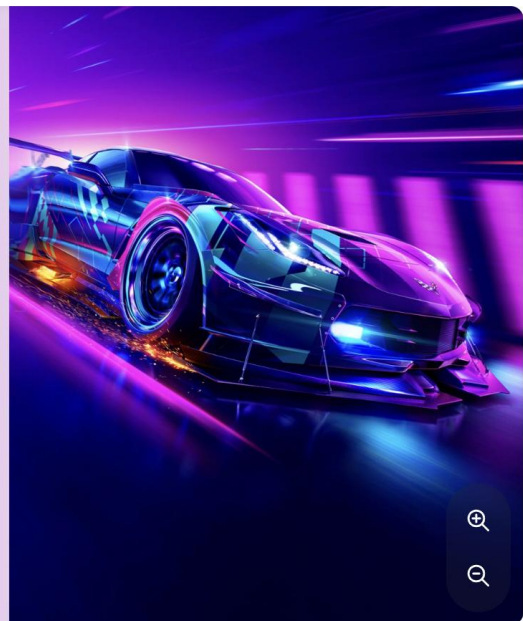
2 Click "Register"

Welcome to Electronic Arts

Your gateway to the EA universe.

Login

Register



3 Fill in your email address, username and password.

EA Electronic Arts

### Create Your EA Account

Email Address

EA Username

Password

Confirm Password

Create Account & Verify Email

Already have an account?

[Login here](#)

4 Click "Create Account & Verify Email"

Email Address

shijegan.gs@gmail.com

EA Username

shijegan

Password

\*\*\*\*\*

Your password must contain the following:

- ✓ 8-64 characters
- ✓ At least 1 lowercase letter
- ✓ At least 1 uppercase letter
- ✓ At least 1 number

Confirm Password

\*\*\*\*\*

Create Account & Verify Email

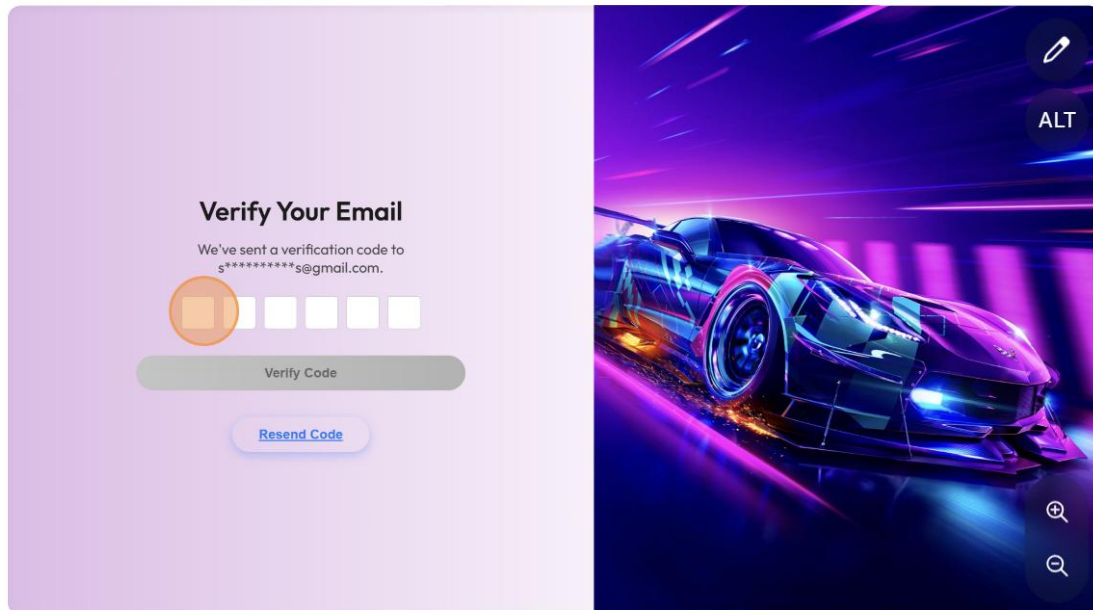
Already have an account?

[Login here](#)

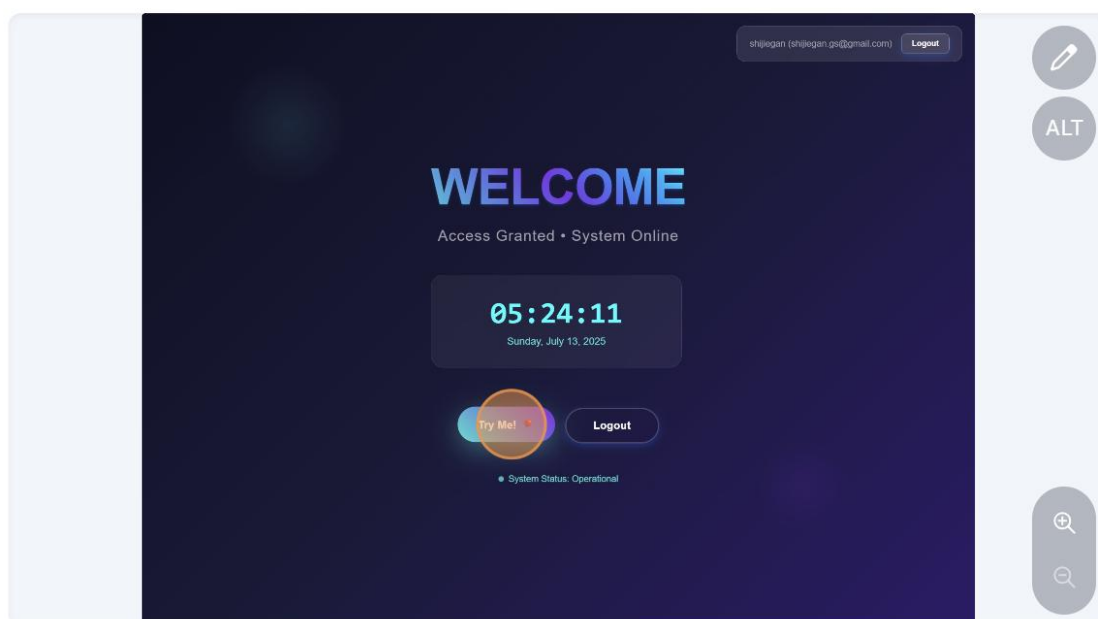
```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS  GITLENS  SQL HISTORY  TASK MONITOR

[OTP] Verification code sent to shijiegan.gs@gmail.com: 834414
```

5 Input the OTP that was logged to the backend console.

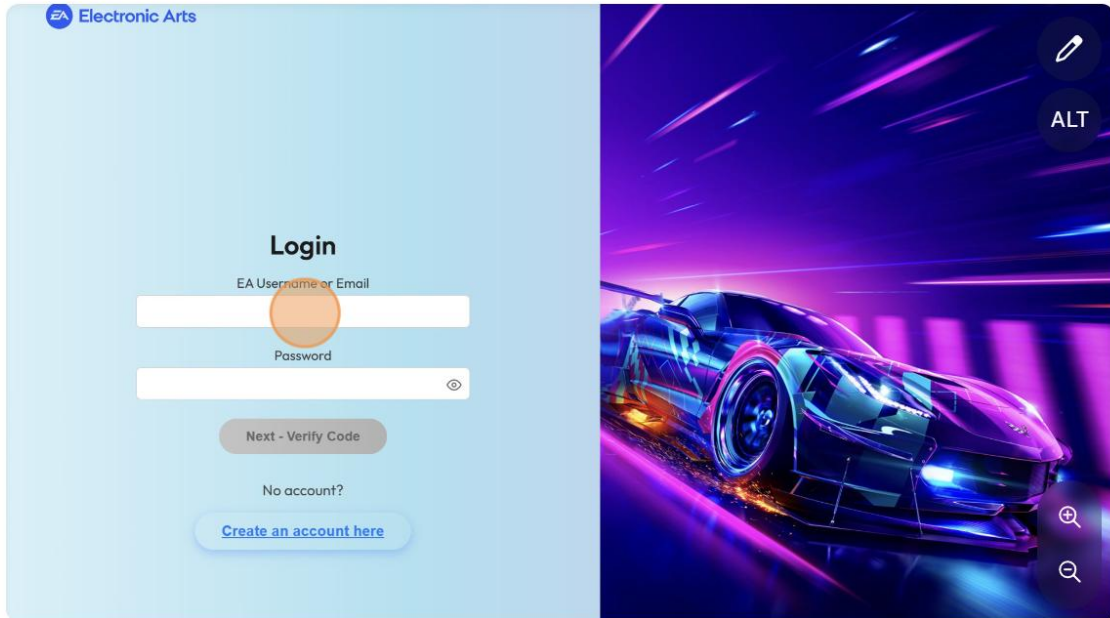


6 Welcome in, free to play around!



## 7.2. Login

- 1 Input your username or email, along with your password.



EA Electronic Arts

### Login

EA Username or Email

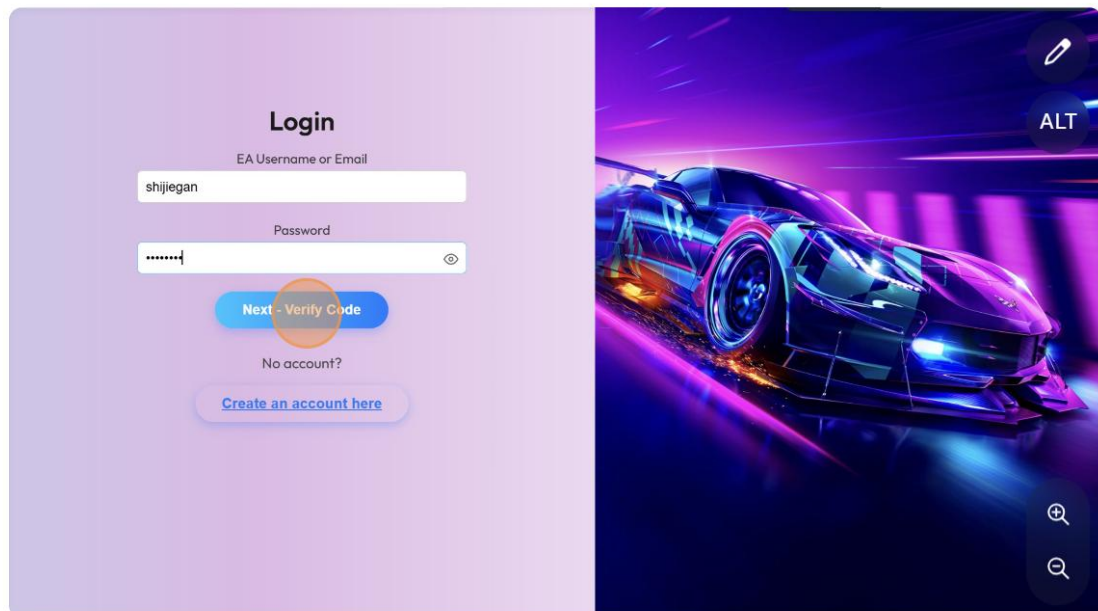
Password

Next - Verify Code

No account?

[Create an account here](#)

- 2 Click "Next - Verify Code"



### Login

EA Username or Email

Password

Next - Verify Code

No account?

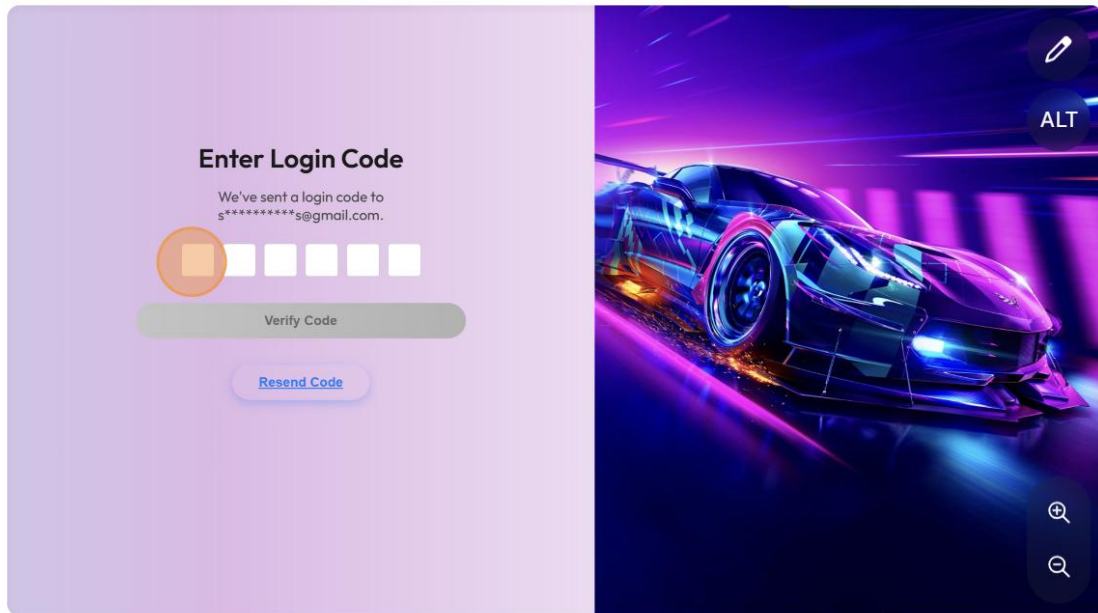
[Create an account here](#)



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS  GITLENS  SQL HISTORY  TASK MONITOR

[OTP] Login code sent to shijiegan.gs@gmail.com: 598700
```

3 Input the OTP that was logged to the backend console.



4 Hello again!

