# Competa Arena User Service — Development Blueprint

## Table of Contents

---

## Purpose & Scope

The User Service is responsible for all aspects of user management on Competa Arena. It ensures secure registration, authentication, profile management, role assignment, account status, and user analytics. All other services interact with it to authenticate users and retrieve public user info.

---

## Core Functionalities

- **User Registration** with strong password policy and email verification
- **Authentication** (JWT-based)
- **Profile Management** (view/update info, upload/change avatar)
- **Role Management** (user, creator, admin; creators via application/approval)
- **Account Status** (active, suspended, blocked)
- **Password Management** (reset/change with email)
- **Public Profiles** (some info visible to all)
- **Creator Applications** (users apply, admin approves)
- **Audit & Analytics** (last login, applications, suspensions, etc.)

---

## API Endpoints Specification

| Endpoint | Method | Auth | Role | Description |
|---|---|---|---|---|
| `/register` | POST | No | - | Register a new user |

| Endpoint | Method | Auth | Role | Description |
|---|---|---|---|---|
| `/login` | POST | No | - | Authenticate and receive JWT |
| `/verify-email` | POST | No | - | Verify email with token |
| `/forgot-password` | POST | No | - | Send password reset link |
| `/reset-password` | POST | No | - | Reset password with token |
| `/me` | GET | Yes | All | Get own profile |
| `/me` | PUT | Yes | All | Update own profile |
| `/me/password` | PUT | Yes | All | Change password (with current password) |
| `/me/avatar` | POST | Yes | All | Upload/change profile photo |
| `/me/apply-creator` | POST | Yes | user | Request creator role |
| `/users/{username}` | GET | Yes/No* | - | Public user profile |
| `/users/creator-applications` | GET | Yes | admin | List pending creator applications |
| `/users/{username}/role` | PUT | Yes | admin | Assign/revoke roles |
| `/users/{username}/suspend` | PUT | Yes | admin | Suspend/reactivate user |
| `/users/{username}/block` | PUT | Yes | admin | Block (ban) user |

*Public endpoint can be restricted for privacy if desired.

**All endpoints validate input with Pydantic models.**

---

## Data Models

### Pydantic User Model (Example)

“‘python name=app/models/user.py from pydantic import BaseModel, EmailStr, constr from typing import Optional from datetime import datetime

class UserBase(BaseModel): username: constr(min_length=3, max_length=20, regex=r”[1]+")$email : EmailStrname : constr(min_length = 2, max_length = 50)country : constr(min_length = 2, max_length = 50)gender : constr(regex = "^{(}male|female|other)$“)

class UserCreate(UserBase): password: constr(min_length=8, max_length=128)

class UserPublic(BaseModel): username: str name: str country: str avatar_url: Optional[str] role: str standing: Optional[int] ranking: Optional[int]

class UserInDB(UserBase): id: int password_hash: str profile_photo_url: Optional[str] role: str creator_application_status: str # ‘none’, ‘pending’, ‘approved’, ‘rejected’ status: str # ‘active’, ‘suspended’, ‘blocked’ email_verified:

[1] a-zA-Z0-9_

bool last_login: Optional[datetime] created_at: datetime updated_at: datetime

~~~
---

## Implementation Details

### FastAPI App Setup

- Use FastAPI for async endpoints & easy OpenAPI docs.
- Use SQLAlchemy ORM for Postgres DB.
- Use Alembic for migrations.
- Use passlib/bcrypt for password hashing.
- Use PyJWT for JWT generation & verification.
- Use Python's email libraries (or SendGrid) for emails.
- Store avatars in object storage (or static folder for dev).
- Use Pydantic for all input/output validation.

### Password Policy Logic Example

```python name=app/utils/password.py
import re

def strong_password(password, username, email):
    if len(password) < 8:
        return False
    if username.lower() in password.lower() or any(part in password.lower() for part in emai
        return False
    if not re.search(r"[A-Z]", password):
        return False
    if not re.search(r"[a-z]", password):
        return False
    if not re.search(r"[0-9]", password):
        return False
    if not re.search(r"[^A-Za-z0-9]", password):
        return False
    return True
~~~

---

## Directory Structure

```
app/
  main.py               # FastAPI entrypoint
  models/               # Pydantic & SQLAlchemy models
     user.py
```

```
schemas/             # Pydantic schemas (request/response)
   user.py
api/                 # API routers
   users.py
db/                  # DB session, connection, setup
   base.py
   crud.py
utils/               # Utilities (auth, password, email)
   auth.py
   password.py
   email.py
tests/               # Pytest test cases
   test_users.py
static/              # Avatars (if using local storage)
```

---

## Sample Test Cases

"'python name=app/tests/test_users.py import pytest from fastapi.testclient import TestClient from app.main import app

client = TestClient(app)

def test_register_user_success(): payload = { "username": "johnny", "email": "johnny@example.com", "name": "John Doe", "country": "Rwanda", "gender": "male", "password": "Str0ng!Passw0rd" } response = client.post("/register", json=payload) assert response.status_code == 201 assert response.json()["message"] == "Registration successful, please verify your email."

def test_register_user_weak_password(): payload = { "username": "johnny", "email": "johnny@example.com", "name": "John Doe", "country": "Rwanda", "gender": "male", "password": "johnny123" } response = client.post("/register", json=payload) assert response.status_code == 400 assert "password" in response.json()["detail"]

def test_login_unverified_email(): payload = { "username": "johnny", "password": "Str0ng!Passw0rd" } response = client.post("/login", json=payload) assert response.status_code == 403 assert response.json()["detail"] == "Email not verified."

def test_apply_creator(): # Assume user is logged in with token token = "Bearer exampletoken" response = client.post("/me/apply-creator", headers={"Authorization": token}) assert response.status_code == 200 assert response.json()["message"] == "Creator application submitted." "'

---

## Views & Usage Scenarios

### Registration & Email Verification

- **User submits info** → receives verification email.
- **User clicks link** → `/verify-email?token=...` endpoint → email marked as verified.

### Login

- **User logs in** with username/email & password.
- **JWT token returned**; required for all protected actions.

### Profile Management

- **User fetches profile** via `/me`.
- **User updates info** (excluding username/email).
- **User uploads/changing avatar** via `/me/avatar`.

### Creator Application

- **User applies to be creator** via `/me/apply-creator`.
- **Admin reviews** via `/users/creator-applications` and approves/rejects.

### Admin Actions

- **Admin suspends/blocks users** via `/users/{username}/suspend` or `/block`.
- **Admin assigns roles** via `/users/{username}/role`.

---

## Best Practices & Security

- **Never store plain text passwords.** Always hash with bcrypt.
- **JWT for stateless authentication**; verify on every request.
- **Validate all inputs** with Pydantic before DB insert/update.
- **Rate limit** registration, login, password reset endpoints.
- **Audit log** all admin actions (role changes, suspensions).
- **Use HTTPS in production** for all endpoints.

---

## Notes for Developers

- Use environment variables for secrets (DB, JWT, email).
- Write unit and integration tests for all endpoints.
- Document API endpoints with FastAPI/OpenAPI docs.

- Use Alembic for DB migrations.
- Structure code for easy extension (e.g., future social login).

---

**If you follow this blueprint, your User Service will be robust, scalable, and ready for integration with the broader Competa Arena platform!**