Venture Pipeline – Authentication Backend Design & Technical Contribution

Prepared by: Team UI/UX & Backend (Co‑lead)  
Date: 2025-08-29

# 1. Overview

This document presents a concise backend design for the Venture Pipeline Management System's authentication workflow.  
It includes a production-lean schema, table-by-table descriptions, a verification token flow, a realistic seed dataset,  
and optional Prisma models. It is intended to demonstrate technical contribution to the team's backend work and to act  
as a starting point for implementation.

## Goals

* Support secure user onboarding (sign up) and login (credentials).
* Store only password hashes (bcrypt/argon2), never plaintext passwords.
* Provide robust, expiring, one-time email verification tokens.
* Support reset-password tokens using the same token infrastructure.
* Track sessions (revocable) and security events (audit log).

## Assumptions

* Primary database: PostgreSQL 14+ (recommended).
* Token/Session storage uses SHA‑256 hashes of the plaintext tokens (never store plaintext tokens).
* Applications run behind HTTPS; secure, HTTP-only cookies are recommended for sessions.

# 2. Relational Schema (PostgreSQL)

The schema is designed for security, maintainability, and clarity. It includes organizations (optional), users, verification tokens, sessions, and an audit log.

## 2.1 Tables & Purpose

### organizations

* id (UUID, PK) – Tenant/group identifier.
* name (TEXT) – Organization name.
* domain (TEXT, optional) – Email or web domain for mapping/branding.
* created\_at (TIMESTAMPTZ) – Creation timestamp.

### users

* id (UUID, PK) – User identifier.
* org\_id (UUID, FK) – Optional link to organizations.
* email (TEXT, unique CI) – Login identifier (case-insensitive uniqueness).
* password\_hash (TEXT) – Argon2/bcrypt hash. Never store plaintext.
* first\_name / last\_name (TEXT) – Profile attributes.
* role (TEXT) – Authorization role: 'admin' or 'member'.
* email\_verified\_at (TIMESTAMPTZ) – Non-null when email is verified.
* created\_at / updated\_at (TIMESTAMPTZ) – Audit timestamps.

### verification\_tokens

* id (UUID, PK) – Token row id.
* user\_id (UUID, FK) – Target user.
* type (TEXT) – EMAIL\_VERIFY | PASSWORD\_RESET.
* token\_hash (TEXT, unique) – SHA‑256 of plaintext token.
* expires\_at (TIMESTAMPTZ) – Expiry to limit token lifetime.
* consumed\_at (TIMESTAMPTZ) – Non-null after first successful use.
* created\_at (TIMESTAMPTZ) – Creation timestamp.

### sessions

* id (UUID, PK) – Session id.
* user\_id (UUID, FK) – Owner user.
* session\_token\_hash (TEXT, unique) – SHA‑256 of random session token.
* ip (INET) / user\_agent (TEXT) – Context data for security review.
* created\_at / expires\_at / revoked\_at – Lifecycle fields.

### audit\_log

* id (UUID, PK) – Audit row id.
* user\_id (UUID, FK) – Nullable to allow anonymous events.
* action (TEXT) – e.g., LOGIN\_SUCCESS, EMAIL\_VERIFIED.
* details (JSONB) – Structured metadata.
* ip (INET) – Source IP if available.
* created\_at (TIMESTAMPTZ) – Event time.

## 2.2 SQL: Create Tables

Below is the PostgreSQL DDL to create the schema:

-- Enable extensions (UUIDs & crypto helpers)  
CREATE EXTENSION IF NOT EXISTS pgcrypto;  
  
-- Organizations (optional; for multi-tenant or grouping)  
CREATE TABLE organizations (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 name TEXT NOT NULL,  
 domain TEXT,  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
-- Users  
CREATE TABLE users (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 org\_id UUID REFERENCES organizations(id) ON DELETE SET NULL,  
 email TEXT NOT NULL,  
 password\_hash TEXT NOT NULL, -- store bcrypt/argon2 hash  
 first\_name TEXT,  
 last\_name TEXT,  
 role TEXT NOT NULL DEFAULT 'member' CHECK (role IN ('admin','member')),  
 email\_verified\_at TIMESTAMPTZ,  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),  
 updated\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
-- Case-insensitive unique email via index  
CREATE UNIQUE INDEX users\_email\_unique\_ci ON users (LOWER(email));  
  
-- Verification tokens (for email verify & password reset)  
CREATE TABLE verification\_tokens (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 user\_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,  
 type TEXT NOT NULL CHECK (type IN ('EMAIL\_VERIFY','PASSWORD\_RESET')),  
 token\_hash TEXT NOT NULL, -- SHA-256 hex of the plaintext token  
 expires\_at TIMESTAMPTZ NOT NULL,  
 consumed\_at TIMESTAMPTZ,  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE UNIQUE INDEX verification\_tokens\_token\_hash\_key ON verification\_tokens(token\_hash);  
  
-- Sessions (backed by secure, HTTP-only cookies or bearer tokens)  
CREATE TABLE sessions (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 user\_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,  
 session\_token\_hash TEXT NOT NULL, -- SHA-256 hex of the random session token  
 ip INET,  
 user\_agent TEXT,  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),  
 expires\_at TIMESTAMPTZ NOT NULL,  
 revoked\_at TIMESTAMPTZ  
);  
  
CREATE UNIQUE INDEX sessions\_token\_hash\_key ON sessions(session\_token\_hash);  
  
-- Audit log for security / compliance  
CREATE TABLE audit\_log (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 user\_id UUID REFERENCES users(id) ON DELETE SET NULL,  
 action TEXT NOT NULL, -- e.g., LOGIN\_SUCCESS, LOGIN\_FAILED, EMAIL\_VERIFIED  
 details JSONB,  
 ip INET,  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);

## 2.3 SQL: Seed Data

-- Organizations  
INSERT INTO organizations (id, name, domain)  
VALUES ('11111111-1111-1111-1111-111111111111', 'Acme Ventures', 'acme.example');  
  
-- Users (password\_hash values are placeholders: replace with real bcrypt/argon2 hashes)  
INSERT INTO users (id, org\_id, email, password\_hash, first\_name, last\_name, role, email\_verified\_at) VALUES  
('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaa1', '11111111-1111-1111-1111-111111111111', 'alice@acme.example',  
 '$2b$12$examplehash\_for\_Alice\_replace\_in\_prod', 'Alice', 'Nguyen', 'admin', now()),  
('bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbb2', '11111111-1111-1111-1111-111111111111', 'bob@acme.example',  
 '$2b$12$examplehash\_for\_Bob\_replace\_in\_prod', 'Bob', 'Rahman', 'member', NULL),  
('cccccccc-cccc-cccc-cccc-ccccccccccc3', '11111111-1111-1111-1111-111111111111', 'charlie@acme.example',  
 '$2b$12$examplehash\_for\_Char\_replace\_in\_prod', 'Charlie', 'Saha', 'member', now());  
  
-- Email verification token for Bob (plaintext DEV token = 'verify-bob-9A7F-ABC')  
INSERT INTO verification\_tokens (id, user\_id, type, token\_hash, expires\_at) VALUES  
('dddddddd-dddd-dddd-dddd-dddddddddddd', 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbb2', 'EMAIL\_VERIFY',  
 encode(digest('verify-bob-9A7F-ABC', 'sha256'), 'hex'), now() + interval '3 days');  
  
-- Password reset token for Alice (plaintext DEV token = 'reset-alice-2025-XYZ')  
INSERT INTO verification\_tokens (id, user\_id, type, token\_hash, expires\_at) VALUES  
('eeeeeeee-eeee-eeee-eeee-eeeeeeeeeeee', 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaa1', 'PASSWORD\_RESET',  
 encode(digest('reset-alice-2025-XYZ', 'sha256'), 'hex'), now() + interval '1 day');  
  
-- One active session for Alice  
INSERT INTO sessions (id, user\_id, session\_token\_hash, ip, user\_agent, expires\_at) VALUES  
('ffffffff-ffff-ffff-ffff-ffffffffffff', 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaa1',  
 encode(digest('session-alice-ABC123', 'sha256'), 'hex'), '203.0.113.10', 'Chrome 124 on macOS',  
 now() + interval '7 days');  
  
-- Audit examples  
INSERT INTO audit\_log (user\_id, action, details, ip) VALUES  
('aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaa1', 'LOGIN\_SUCCESS', '{"method":"password"}', '203.0.113.10'),  
('bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbb2', 'SIGNUP\_SUBMITTED', '{"email":"bob@acme.example"}', '198.51.100.2');

## 2.4 SQL: Verification Helpers

-- Verify a user's email by plaintext token (provided via link):  
-- :provided\_token is the token from the email (plaintext). Store only its SHA-256 in DB.  
  
WITH t AS (  
 SELECT vt.\*, u.id AS uid  
 FROM verification\_tokens vt  
 JOIN users u ON u.id = vt.user\_id  
 WHERE vt.type = 'EMAIL\_VERIFY'  
 AND vt.token\_hash = encode(digest(:provided\_token, 'sha256'), 'hex')  
 AND vt.consumed\_at IS NULL  
 AND vt.expires\_at > now()  
)  
UPDATE users u  
SET email\_verified\_at = COALESCE(u.email\_verified\_at, now()),  
 updated\_at = now()  
FROM t  
WHERE u.id = t.uid  
RETURNING u.id AS user\_id;  
  
-- Mark the token as consumed (one-time use)  
UPDATE verification\_tokens  
SET consumed\_at = now()  
WHERE token\_hash = encode(digest(:provided\_token, 'sha256'), 'hex')  
 AND type = 'EMAIL\_VERIFY'  
 AND consumed\_at IS NULL;

# 3. Prisma Models (Optional)

If your project uses Prisma, the following models mirror the SQL schema:

// schema.prisma (Prisma + PostgreSQL)  
datasource db {  
 provider = "postgresql"  
 url = env("DATABASE\_URL")  
}  
  
generator client {  
 provider = "prisma-client-js"  
}  
  
model Organization {  
 id String @id @default(uuid())  
 name String  
 domain String?  
 createdAt DateTime @default(now())  
 users User[]  
}  
  
model User {  
 id String @id @default(uuid())  
 orgId String?  
 org Organization? @relation(fields: [orgId], references: [id])  
 email String @db.Text  
 passwordHash String @db.Text  
 firstName String?  
 lastName String?  
 role Role @default(MEMBER)  
 emailVerifiedAt DateTime?  
 createdAt DateTime @default(now())  
 updatedAt DateTime @default(now())  
  
 sessions Session[]  
 tokens VerificationToken[]  
  
 @@unique([email]) // Normalize to lowercase in app code for CI uniqueness  
}  
  
enum Role {  
 ADMIN  
 MEMBER  
}  
  
model VerificationToken {  
 id String @id @default(uuid())  
 userId String  
 user User @relation(fields: [userId], references: [id], onDelete: Cascade)  
 type TokenType  
 tokenHash String @db.Text  
 expiresAt DateTime  
 consumedAt DateTime?  
 createdAt DateTime @default(now())  
  
 @@unique([tokenHash])  
}  
  
enum TokenType {  
 EMAIL\_VERIFY  
 PASSWORD\_RESET  
}  
  
model Session {  
 id String @id @default(uuid())  
 userId String  
 user User @relation(fields: [userId], references: [id], onDelete: Cascade)  
 sessionTokenHash String @db.Text  
 ip String?  
 userAgent String?  
 createdAt DateTime @default(now())  
 expiresAt DateTime  
 revokedAt DateTime?  
  
 @@unique([sessionTokenHash])  
}  
  
model AuditLog {  
 id String @id @default(uuid())  
 userId String?  
 user User? @relation(fields: [userId], references: [id], onDelete: SetNull)  
 action String  
 details Json?  
 ip String?  
 createdAt DateTime @default(now())  
}

## 3.1 Prisma Seed (TypeScript)

// prisma/seed.ts  
import { PrismaClient } from '@prisma/client'  
import { createHash } from 'crypto'  
  
const prisma = new PrismaClient()  
const sha256 = (s: string) => createHash('sha256').update(s).digest('hex')  
  
async function main() {  
 const org = await prisma.organization.upsert({  
 where: { id: '11111111-1111-1111-1111-111111111111' },  
 update: {},  
 create: { id: '11111111-1111-1111-1111-111111111111', name: 'Acme Ventures', domain: 'acme.example' }  
 })  
  
 const alice = await prisma.user.create({  
 data: {  
 id: 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaa1',  
 orgId: org.id,  
 email: 'alice@acme.example',  
 passwordHash: '$2b$12$examplehash\_for\_Alice\_replace\_in\_prod',  
 firstName: 'Alice',  
 lastName: 'Nguyen',  
 role: 'ADMIN',  
 emailVerifiedAt: new Date()  
 }  
 })  
  
 const bob = await prisma.user.create({  
 data: {  
 id: 'bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbb2',  
 orgId: org.id,  
 email: 'bob@acme.example',  
 passwordHash: '$2b$12$examplehash\_for\_Bob\_replace\_in\_prod',  
 firstName: 'Bob',  
 lastName: 'Rahman',  
 role: 'MEMBER'  
 }  
 })  
  
 await prisma.verificationToken.create({  
 data: {  
 id: 'dddddddd-dddd-dddd-dddd-dddddddddddd',  
 userId: bob.id,  
 type: 'EMAIL\_VERIFY',  
 tokenHash: sha256('verify-bob-9A7F-ABC'),  
 expiresAt: new Date(Date.now() + 3 \* 24 \* 3600 \* 1000)  
 }  
 })  
  
 await prisma.session.create({  
 data: {  
 id: 'ffffffff-ffff-ffff-ffff-ffffffffffff',  
 userId: alice.id,  
 sessionTokenHash: sha256('session-alice-ABC123'),  
 ip: '203.0.113.10',  
 userAgent: 'Chrome 124 on macOS',  
 expiresAt: new Date(Date.now() + 7 \* 24 \* 3600 \* 1000)  
 }  
 })  
  
 await prisma.auditLog.create({  
 data: { userId: alice.id, action: 'LOGIN\_SUCCESS', details: { method: 'password' }, ip: '203.0.113.10' }  
 })  
}  
  
main().finally(() => prisma.$disconnect())

# 4. Minimal Email Verification Handler (Node/TypeScript + pg)

This snippet demonstrates how to verify a plaintext token from an email link, mark it consumed, and log the event.

// Minimal verification handler (Node/TypeScript + pg)  
import { createHash } from 'crypto';  
import { pool } from './db'; // your pg Pool instance  
  
const sha256 = (s: string) => createHash('sha256').update(s).digest('hex');  
  
export async function verifyEmailToken(plaintextToken: string) {  
 const client = await pool.connect();  
 try {  
 await client.query('BEGIN');  
  
 const { rows } = await client.query(  
 `SELECT vt.user\_id  
 FROM verification\_tokens vt  
 WHERE vt.type = 'EMAIL\_VERIFY'  
 AND vt.token\_hash = $1  
 AND vt.consumed\_at IS NULL  
 AND vt.expires\_at > now()  
 FOR UPDATE`,  
 [sha256(plaintextToken)]  
 );  
  
 if (!rows.length) throw new Error('Invalid or expired token');  
 const userId = rows[0].user\_id;  
  
 await client.query(  
 'UPDATE users SET email\_verified\_at = COALESCE(email\_verified\_at, now()), updated\_at = now() WHERE id = $1',  
 [userId]  
 );  
  
 await client.query(  
 'UPDATE verification\_tokens SET consumed\_at = now() WHERE token\_hash = $1 AND type = $2 AND consumed\_at IS NULL',  
 [sha256(plaintextToken), 'EMAIL\_VERIFY']  
 );  
  
 await client.query(  
 "INSERT INTO audit\_log (user\_id, action, details) VALUES ($1, 'EMAIL\_VERIFIED', '{"source":"email\_link"}')",  
 [userId]  
 );  
  
 await client.query('COMMIT');  
 return { ok: true, userId };  
 } catch (e) {  
 await client.query('ROLLBACK');  
 throw e;  
 } finally {  
 client.release();  
 }  
}

# 5. API Endpoints (Suggested)

* POST /api/auth/signup – Create user, store password hash, emit EMAIL\_VERIFY token & send email.
* POST /api/auth/login – Validate credentials, create session, set secure cookie.
* POST /api/auth/logout – Revoke current session.
* POST /api/auth/verify-email – Accept token, mark user verified, consume token.
* POST /api/auth/request-password-reset – Create PASSWORD\_RESET token & email it.
* POST /api/auth/reset-password – Accept reset token & new password, consume token, rotate sessions.

# 6. Security Considerations

* Always hash tokens (SHA‑256) and passwords (argon2id or bcrypt) server‑side.
* Use HTTPS and HTTP‑only, Secure cookies for sessions.
* Enforce token expiry and single use (set consumed\_at).
* Normalize and unique‑index email on LOWER(email).
* Rate limit login, signup, and token endpoints; log failures to audit\_log.
* Rotate/Invalidate sessions on password reset and account‑sensitive changes.

# 7. Runbook (Quickstart)

1. Create database & run DDL: apply schema\_postgres.sql (DDL + seed + helpers).
2. Set server secrets: DB URL, bcrypt/argon config, email gateway creds.
3. Hook UI submit to backend endpoints (signup/login/verify).
4. Implement email service to deliver verification and reset links.
5. Add monitoring dashboards for audit\_log and session anomalies.

# 8. Technical Contribution Summary

Designed a secure, production-lean authentication schema; implemented verification token workflow; prepared seed dataset; outlined API contracts and security controls; and supplied Prisma models for teams using ORM. This document and the attached scripts are ready for integration with the existing frontend.