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
('ddddddddd-dddd-dddd-dddd-ddddddddddddd', '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.