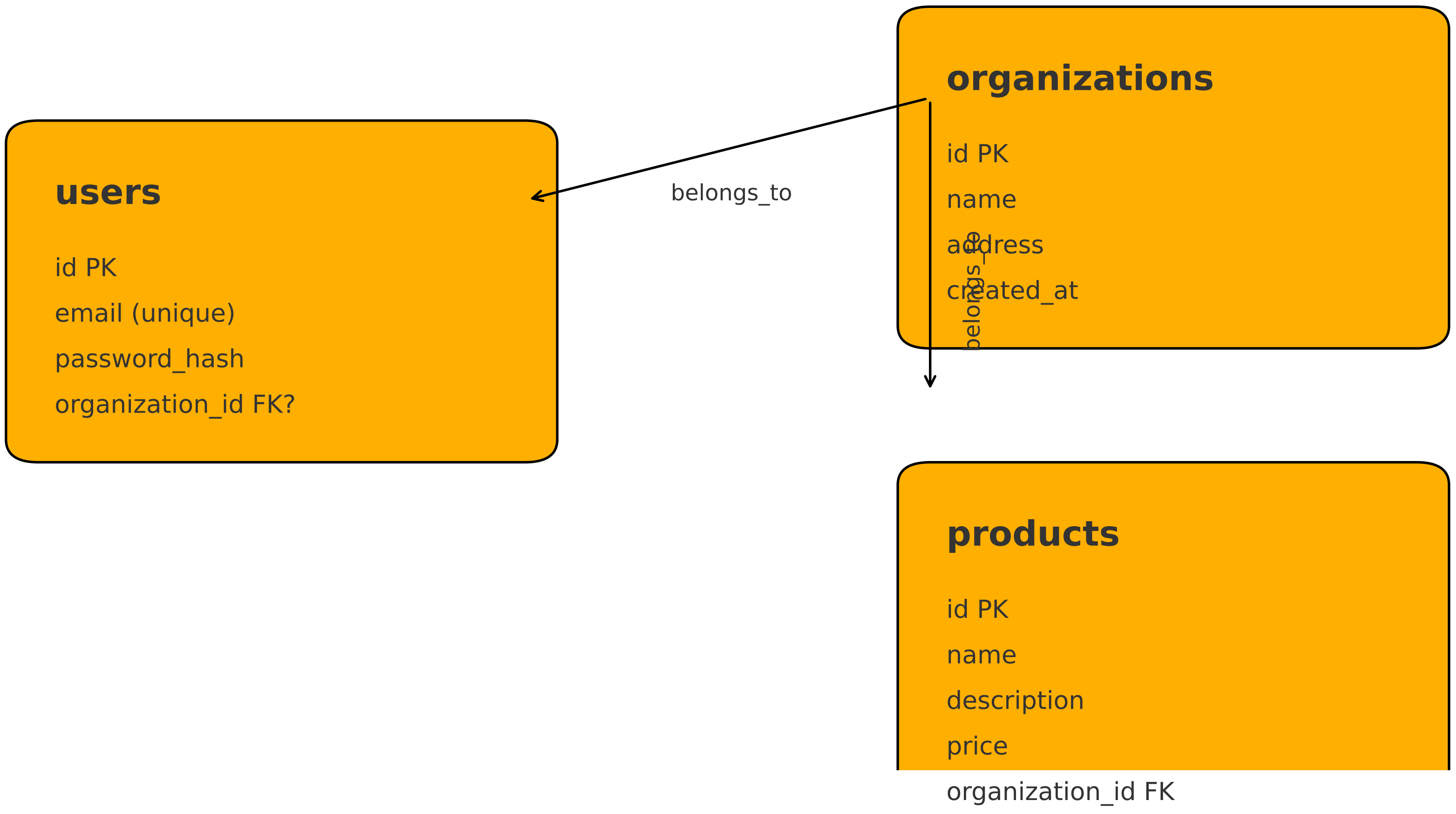
Auth Development History — Extended

This document contains an overview of the authentication development work, ERD, security architecture, and frontend token handling guide. Generated by ChatGPT on request.

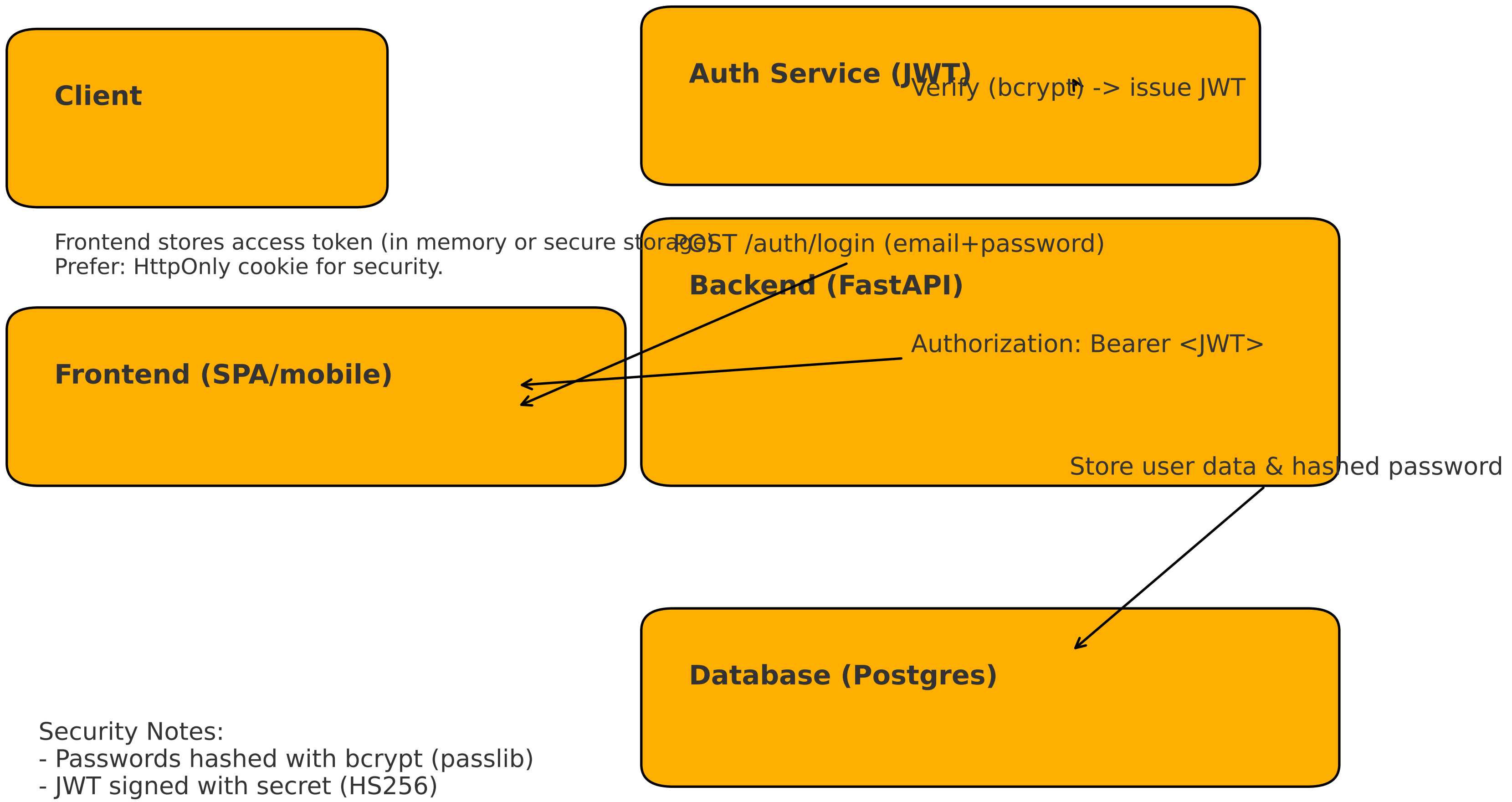
Development Summary

Summary of what we implemented (high-level): - Database: Postgres (Docker). Tables: users, organizations, products. - Backend: FastAPI with SQLAlchemy ORM. Declarative Base and session dependency. - Auth: Registration and login endpoints, password hashing with bcrypt (passlib), JWT issuance. - Security: Token-based auth (JWT), token validation dependency for protected routes. - Dev notes: resolved environment loading (.env), docker postgres vs local, fixed import cycles and model Base visibility. Errors encountered and fixes (highlights): - 'DATABASE_URL not defined' -> fixed by using load_dotenv(find_dotenv()) and printing for debug. - 'password authentication failed for user' -> fixed by restarting docker container and ensuring credentials matched .env. - 'column users.password_hash does not exist' -> run migrations / recreate tables; used SQLAlchemy Base.metadata.create_all() - bcrypt passlib backend warnings and 72-byte password limit -> truncate or ensure passwords shorter than 72 bytes and install correct bcrypt package.

Entity Relationship Diagram (ERD)



Security Architecture & Authentication Flow



Client
Auth Service (JWT)
Verify (bcrypt) -> issue JWT
Frontend stores access token (in memory or secure storage)
Prefer: HttpOnly cookie for security.
POST /auth/login (email+password)
Backend (FastAPI)
Frontend (SPA/mobile)
Authorization: Bearer <JWT>
Store user data & hashed password
Database (Postgres)

Security Notes:
- Passwords hashed with bcrypt (passlib)
- JWT signed with secret (HS256)
- Backend validates token on protected routes
- Use short-lived access tokens + refresh tokens
- Use HTTPS in production

Frontend Token Handling Guide

1) After successful login, the backend returns an access token (JWT). Example response: { "access_token": "<JWT>", "token_type": "bearer" } 2) Where to store the token (options):   - HttpOnly Secure Cookie (recommended for web SPAs): prevents JS access and mitigates XSS.   - In-memory (React state / Redux): safe from persistent XSS but lost on refresh; combine with refresh token.   - localStorage/sessionStorage: easy but vulnerable to XSS (use only if you understand the risks). 3) Using the token on requests: - Include header: Authorization: Bearer <JWT>   - Or rely on HttpOnly cookie with server-side session handling. 4) Refresh tokens: - Store refresh tokens more securely (HttpOnly cookie) and use an endpoint /auth/refresh to get new access tokens.   - Keep access tokens short-lived (minutes) and refresh tokens longer (days) with revocation capability. 5) Security tips: - Always use HTTPS in production.   - Implement CSRF protection if using cookies.   - Validate tokens and scopes server-side.   - Log token usage and provide token revocation for logout/compromise.

Additional Sections & Next Steps

- Full API design: endpoints, request/response examples, error codes. - Database schema details and migration plan (use Alembic). - ER diagram (included) and extended relationships (roles, permissions). - Security: refresh tokens, token revocation, rate limiting, audit logging. - Frontend: sample code snippets (React) for login, storing token, and refresh flow. - Deployment checklist: HTTPS, environment variables, secrets management, containerization. If you want, I can: - Add a formal corporate layout and branding. - Produce a longer, line-by-line chronology of each code edit since Day 1. - Export separate PNGs of diagrams for embedding elsewhere.