ChatGPT

Project Overview

Build a **single full-stack web application** using FastAPI (Python) as the backend API and React+Vite as the frontend. The app will showcase common intranet features: user login/logout, user profile management, an admin panel for user management, personal access token (PAT) management, and a sample data entity with full CRUD. Authentication will support both traditional username/password (via a secure form and JWTs) and LDAP (Active Directory) login, with eventual SAML/OIDC SSO support. The demo will use Docker Compose on a Linux server (in a private datacenter on Azure GCC High) to deploy backend, frontend, and database containers. The database can be **SQLite** for simplicity (a single file) [1], with the option to switch to SQL Server in production (FastAPI's SQLModel/SQLAlchemy supports SQL Server) [1]. We will follow industry best practices for security, testing, and CI/CD. A reference point is the official "Full Stack FastAPI" template, which uses FastAPI + React + Vite + Docker and already implements many needed features (secure auth, JWT, Docker Compose, PyTest, Playwright, etc.) [2] [3].

Technology Stack

- **Backend:** FastAPI with Python 3.10+. Use Pydantic/SQLModel (built on SQLAlchemy) for data models and validation [4] [1]. FastAPI's security utilities will handle password hashing and JWT issuance [3]. Manage environment settings via Pydantic's `BaseSettings` (loading from `.env` files) [5].
- **Frontend:** React (with TypeScript) bootstrapped by Vite for fast development. Use a UI library (e.g. Tailwind CSS or a component library like Chakra UI) for styling [6]. The frontend will interact with the backend API (using fetch/axios or auto-generated clients).
- **Database:** SQLite for the demo (file-based, no external DB setup) [1]. Define models for Users, Tokens, Items, etc., and use Alembic for migrations. The FastAPI tutorial notes SQLModel works with SQLite or any SQL database (including SQL Server) [1]. SQL Server can be used in production (it's supported by SQLAlchemy). IT will handle backups of the production SQL Server.
- **Authentication:** Username/password (with secure hashing) and LDAP. FastAPI supports OAuth2/JWT flows out-of-the-box [3]. For LDAP, we'll use the `ldap3` library to verify credentials against AD (for example, via HTTPBasic credentials) [7]. SSO can be added later (e.g. Azure AD SAML using PySAML2) [8]. Implement role-based access: a special "admin" role in the database can access user-management endpoints.
- **Features:**
- *User Accounts:* Sign-up (or admin-create), login, logout, and profile editing. Implement email-based password reset. Store user info and hashed passwords in the DB.
- *User Management:* An **admin dashboard** where admins can list users, create/edit/delete accounts, assign roles, and manage user status (active/disabled).
- *Personal Access Tokens:* Allow users to generate API tokens (like GitHub PATs) tied to their account. Tokens should be stored securely (hashed) and limited by scope/expiry. (Tokens are essentially long-lived credentials, so treat them "like passwords" and require expiration/scopes [9].) The UI will let users create, view (once), and revoke tokens.
- *Example CRUD Entity:* Include a sample data model (e.g. "Items" or "Projects") with a simple frontend interface. This demonstrates database interaction: users can create/read/update/delete records they own.

- *API Documentation:* FastAPI's automatic docs (Swagger UI) will be available for testing the API endpoints.

Frontend (UI)

The **React** frontend (built with Vite) will provide intuitive UI pages for all functionality. For example, the login page could look like the reference below.

Example login screen UI (email/password form).

- Use React router to manage pages (login, dashboard, profile, admin panel).
- UI library: **Tailwind CSS** is a great choice for custom designs, or a component library like **Chakra UI/ Bootstrap** for quick layout [6]. The FastAPI template uses Chakra UI, but any responsive framework works.
- Vite's hot-reloading simplifies development. Include unit tests with Vitest (designed for Vite) or Jest+React Testing Library. (Vitest is highly recommended for new Vite projects [10].)

Example admin dashboard: user list and management.

- The admin dashboard will have tables (e.g. listing users, items). Admins can click "Create User" to open a form like below.

User creation form (admin panel).

- Each user's "settings" page (profile) allows editing personal info (name, password, etc.), and viewing their own PATs.

User profile page (update personal info, view tokens).

Authentication & Authorization

- **Basic Auth + JWT:** On login, the backend will check credentials (against SQLite or LDAP). On success, return a JWT (stored in a secure cookie or local storage) [3]. Use HTTPS (TLS) in production to protect tokens. FastAPI's `fastapi.security` and `oauth2_password_bearer` schemes make this straightforward.
- **LDAP:** If internal users should use AD credentials, integrate LDAP as shown in examples [7]. For instance, use FastAPI's HTTPBasic dependency to accept username/password and call `ldap3.Connection.bind()` against the domain controller. If bind succeeds, consider the user authenticated [7]. Failing authentication returns HTTP 401.
- **Roles:** Embed user roles in the JWT or check in DB on each request. Secure sensitive endpoints (like user management) with a dependency that ensures the "admin" role. FastAPI's dependency injection can enforce this easily.
- **SSO (future):** For single sign-on, we can use Azure AD (EntraID) via SAML or OAuth2/OIDC. For example, use PySAML2 to handle SAML redirects and responses [8], or use MSAL (Microsoft Authentication Library) for OIDC. This would allow users to log in with their corporate credentials in one click.

Database & Data Model

- Use **SQLModel** (or SQLAlchemy) to define tables: `User`, `PersonalAccessToken`, `Item`, etc. A `User` model includes fields like email, hashed_password, full_name, role, and LDAP flag. The `PersonalAccessToken` model stores token hashes, owner_id, creation/expiry timestamps, and scopes. The `Item` model can have simple fields (id, owner_id, name, description).
- For SQLite, use a session-local dependency (`Session(engine)`) and create tables at startup [1]. In production, configure the connection string for SQL Server (via ODBC or SQLAlchemy's MSSQL driver). Always apply migrations on upgrade – either by running Alembic as a separate step or on app startup (see best practices [11]).
- IT will handle backups of the production SQL Server. For our demo, SQLite is ephemeral and can be recreated.

Testing & Automation

- **Backend tests:** Use **PyTest** to unit-test all Python code and API endpoints. The FastAPI template includes PyTest setup [3]. Write tests for auth flows, CRUD operations, and security rules.
- **Frontend tests:** Use **Vitest** (recommended for Vite) or **Jest** with React Testing Library to unit-test React components and hooks. The Speakeasy guide recommends Vitest for most modern apps: "If you're starting a new project, use Vitest" [10].
- **End-to-end (E2E) tests:** Use **Playwright** (or Puppeteer) to simulate user flows in the browser. Playwright is preferred because it supports multiple browsers (Chromium, Firefox, WebKit) and has built-in test runner features [12] [6]. Write scripts for key scenarios: user login, profile update, admin user creation, and item CRUD. The FastAPI full-stack template uses Playwright for E2E tests by default [6].
- **Automation:** Integrate tests into CI: run PyTest and Vitest on each commit, and run Playwright tests after deployment to a staging container. This ensures new changes don't break login, data access, etc.

CI/CD and Deployment with Docker

- **Docker Compose:** Containerize the application. One image for the FastAPI backend, one for the React frontend (or serve frontend statically via Nginx), and one for the database (if using SQL Server, use Microsoft's Docker image; for SQLite, no separate DB container needed). The Compose file will link these services on a private Docker network. Use official base images (e.g. `python:3.13-slim` for FastAPI [13]) and configure health checks.
- **Dockerfile (backend):** Follow best practices [13]: copy `requirements.txt` first to leverage layer caching, run `pip install --no-cache-dir -r requirements.txt`, then copy app code. Use the exec form for CMD: e.g.

```
FROM python:3.13-slim
WORKDIR /code
COPY requirements.txt /code/
RUN pip install --no-cache-dir -r requirements.txt
COPY . /code
```

```
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80", "--proxy-headers"]
```

The `--proxy-headers` flag ensures FastAPI honors forwarded headers (trusting a reverse proxy) [14].

- **Environment & Config:** Use environment variables for all settings (database URL, SECRET_KEY, etc.) and load them via Pydantic BaseSettings [5]. Do not hard-code secrets. Keep a `.env.example` template (add actual `.env` to `.gitignore`). FastAPI's documentation recommends this approach [15].
- **Deploying:** On the target server, install Docker and Docker Compose. Push the Docker images to a registry (private or Azure Container Registry). On each new version, pull updated images and run `docker-compose up -d`. Document upgrade steps (e.g. stop old containers, run DB migrations, start new ones). Demonstrate an upgrade by changing the app version tag or code, and redeploy.
- **CI/CD:** Use **Jenkins** (or GitHub Actions) to automate builds. For example, on each Git push Jenkins could build and push Docker images, then SSH into the server to trigger `docker-compose pull` and `up -d`. Include automated tests in the pipeline before deployment. The FastAPI template shows GitHub Actions CI/CD, but in your environment Jenkins can serve a similar role [16].

Upgrade Strategy

- **Versioning:** Tag releases (v1.0, v1.1, etc.). Store Docker image tags accordingly. Keep database migrations in sync.
- **Migration:** On upgrade, run Alembic `upgrade head` to apply schema changes (see code snippet in best practices [11]). Ensure only one instance runs migrations to avoid race conditions.
- **Backup:** Since IT handles backups (SQL Server snapshots), coordinate release windows with backup schedules. For SQLite (demo), data loss is acceptable.
- **Rolling Back:** If an upgrade fails tests, simply revert to the previous image tag and configuration.

Constraints and Best Practices

- **Security:** Treat credentials and tokens like passwords. Store PATs hashed, enforce expirations and scopes (GitHub's docs note "PATs are like passwords" and recommend expirations) [9]. Always use HTTPS (TLS) in production – even internal apps should encrypt traffic. Do not run containers as root and minimize open ports. Use official base images (e.g. `python:slim`) and keep them updated.
- **Configuration:** Use the 12-factor App approach – all config (secrets, URLs) via environment variables [5]. Do not commit `.env` to Git. Use a `.env.example` for reference. FastAPI's Pydantic settings can auto-load from env files [5].
- **Logging & Monitoring:** Implement structured logging (JSON output) so logs can be aggregated. For example, configure Python's `logging.basicConfig` with a JSON formatter [17]. Use Docker's logging drivers (e.g. `max-size: 10m` on json-file) to rotate logs. Consider a centralized log service (ELK, Splunk, etc.).
- **Health Checks:** Include a simple "/health" endpoint for container health checks (returns 200 OK) [18]. The orchestrator (Docker) can use this to restart the app if it becomes unhealthy.
- **Dependencies:** Pin library versions in `requirements.txt` to avoid accidental breaking changes. Regularly run `pip-compile` or equivalent to update safely.

- **Testing in CI:** Don't rely on any external services in tests. Use SQLite or in-memory DB for unit tests. In CI/CD, run both backend and frontend tests (the template shows PyTest and Playwright) [19].
- **License:** The application code will be open-source with an **MIT license** (preferred). The FastAPI full-stack template itself is MIT-licensed [20], so you can fork or copy it under MIT as well. This license is compatible with in-house use and makes future sharing simpler.
- **Compliance (GCC High):** Since the target is Azure Government (GCC High), use FedRAMP-authorized images and libraries. Don't include any disallowed network calls. Use Azure AD (GCC) for identity if SSO is enabled.

By following this plan—with FastAPI, React/Vite, Docker Compose, and the outlined practices—we'll have a minimal yet complete demo app illustrating login, LDAP auth, user/admin workflows, PATs, a database CRUD, and a robust deployment pipeline. All key decisions (Tech Stack, testing tools, deployment) are aligned with current best practices and referenced examples [2] [13] [9].

Sources: FastAPI documentation and templates [2] [1]; LDAP/SAML integration examples [7] [8]; Docker and testing best-practices guides [13] [6] [9].

---

[1] SQL (Relational) Databases - FastAPI
https://fastapi.tiangolo.com/tutorial/sql-databases/

[2] [3] [4] [6] [16] [19] [20] GitHub - fastapi/full-stack-fastapi-template: Full stack, modern web application template. Using FastAPI, React, SQLModel, PostgreSQL, Docker, GitHub Actions, automatic HTTPS and more.
https://github.com/fastapi/full-stack-fastapi-template

[5] [11] [13] [14] [15] [17] [18] FastAPI Docker Best Practices | Better Stack Community
https://betterstack.com/community/guides/scaling-python/fastapi-docker-best-practices/

[7] GitHub - RetributionByRevenue/LDAP3-fastapi-auth-simple: LDAP3 fastapi auth in 44 lines | python active directory
https://github.com/RetributionByRevenue/LDAP3-fastapi-auth-simple

[8] Integrate Azure AD SSO with SAML for your FastAPI Application using PySAML2 - Anurag's Blog
https://anurag.codes/posts/azure-ad-pysaml2/

[9] Managing your personal access tokens - GitHub Docs
https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens

[10] Vitest vs Jest | Speakeasy
https://www.speakeasy.com/blog/vitest-vs-jest

[12] Playwright vs Puppeteer: Which to choose in 2025? | BrowserStack
https://www.browserstack.com/guide/playwright-vs-puppeteer