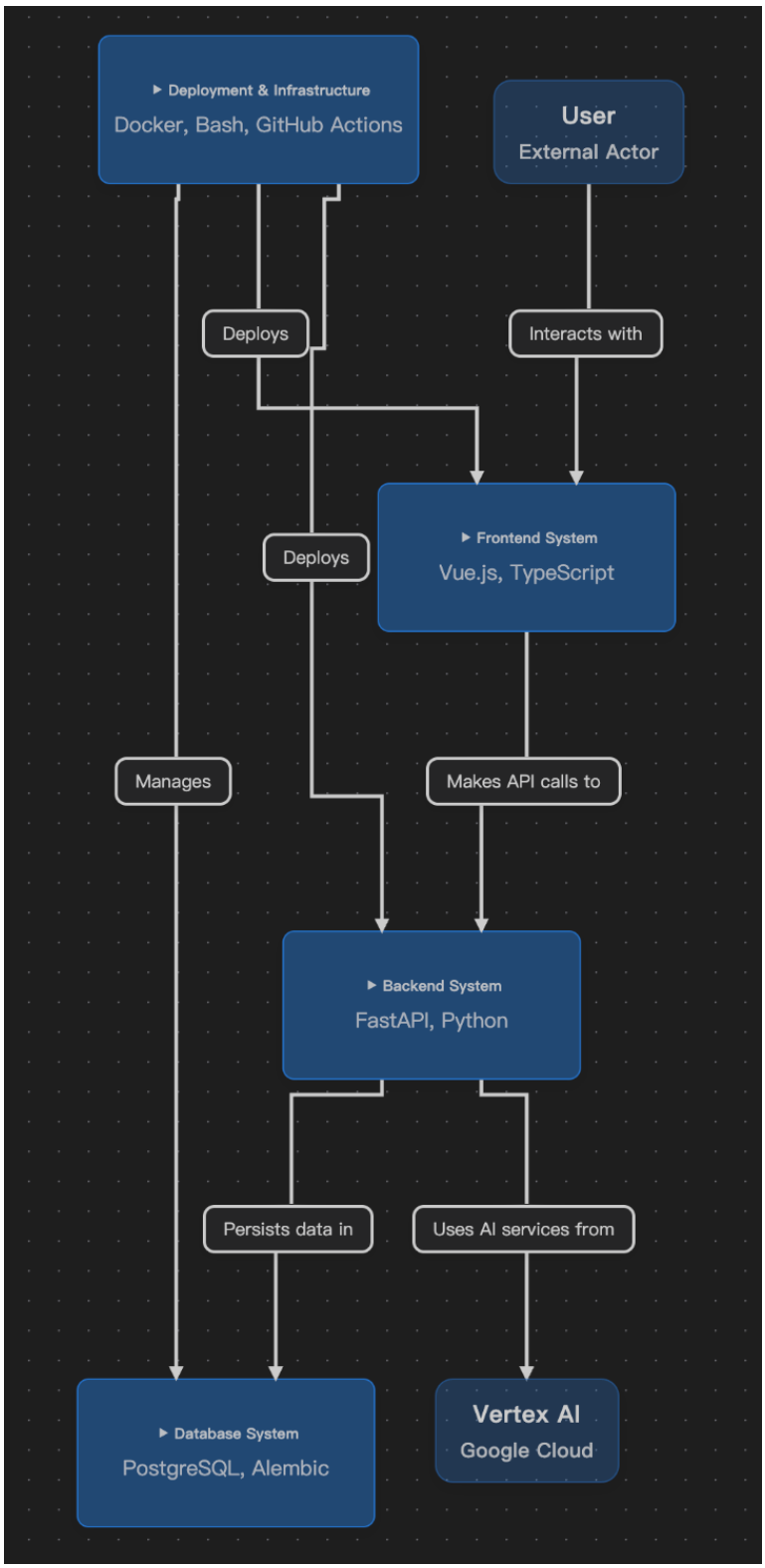
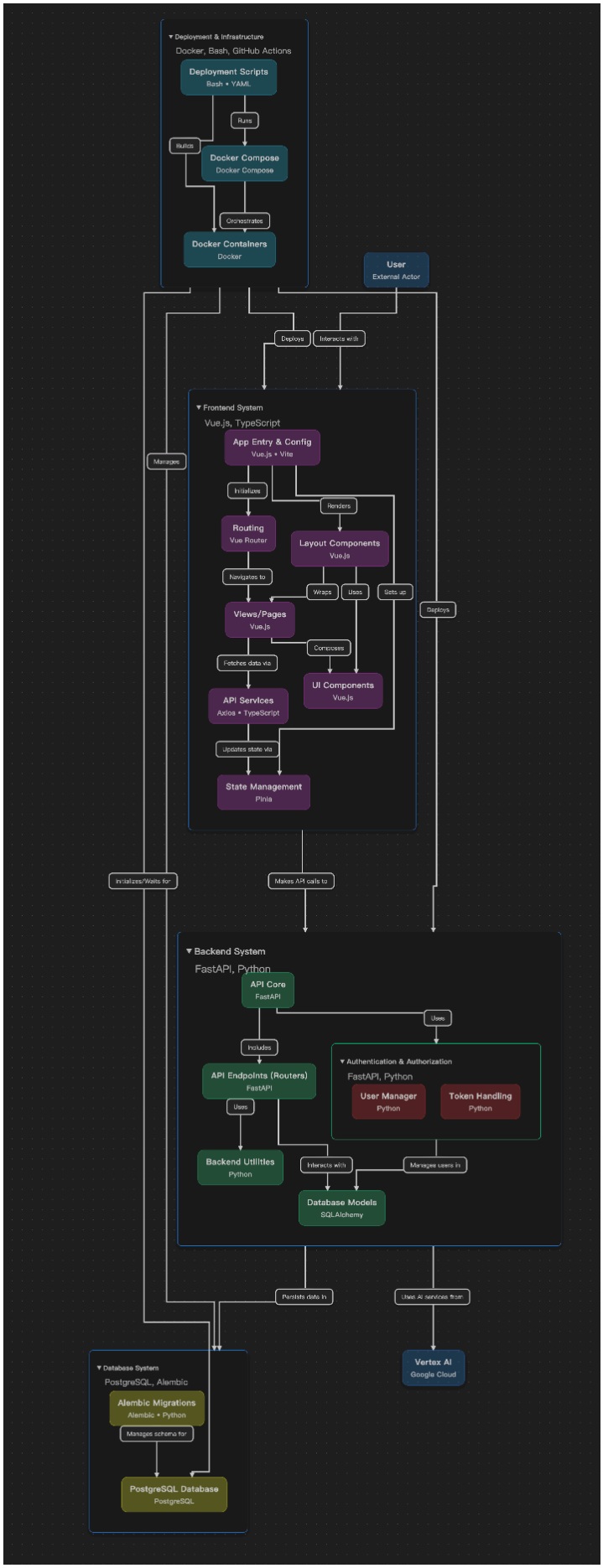
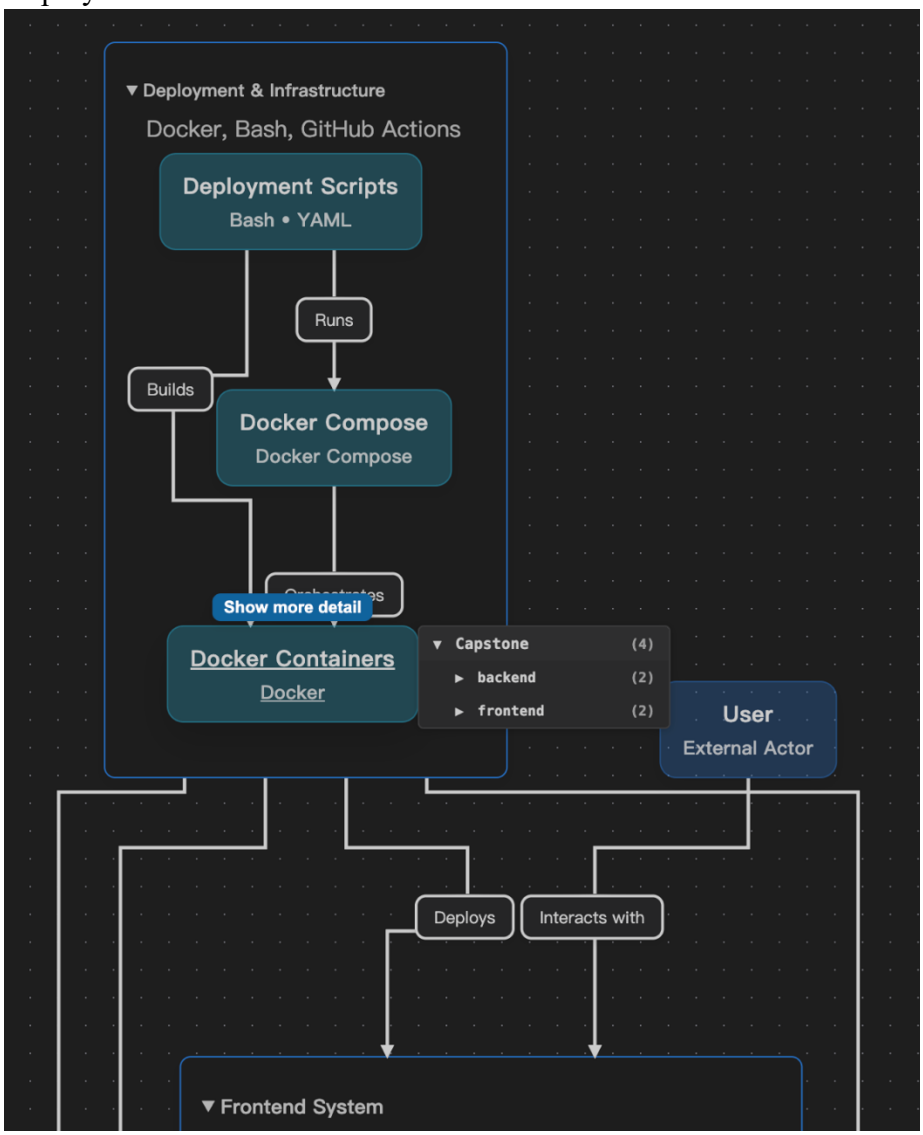
Architecture Diagram

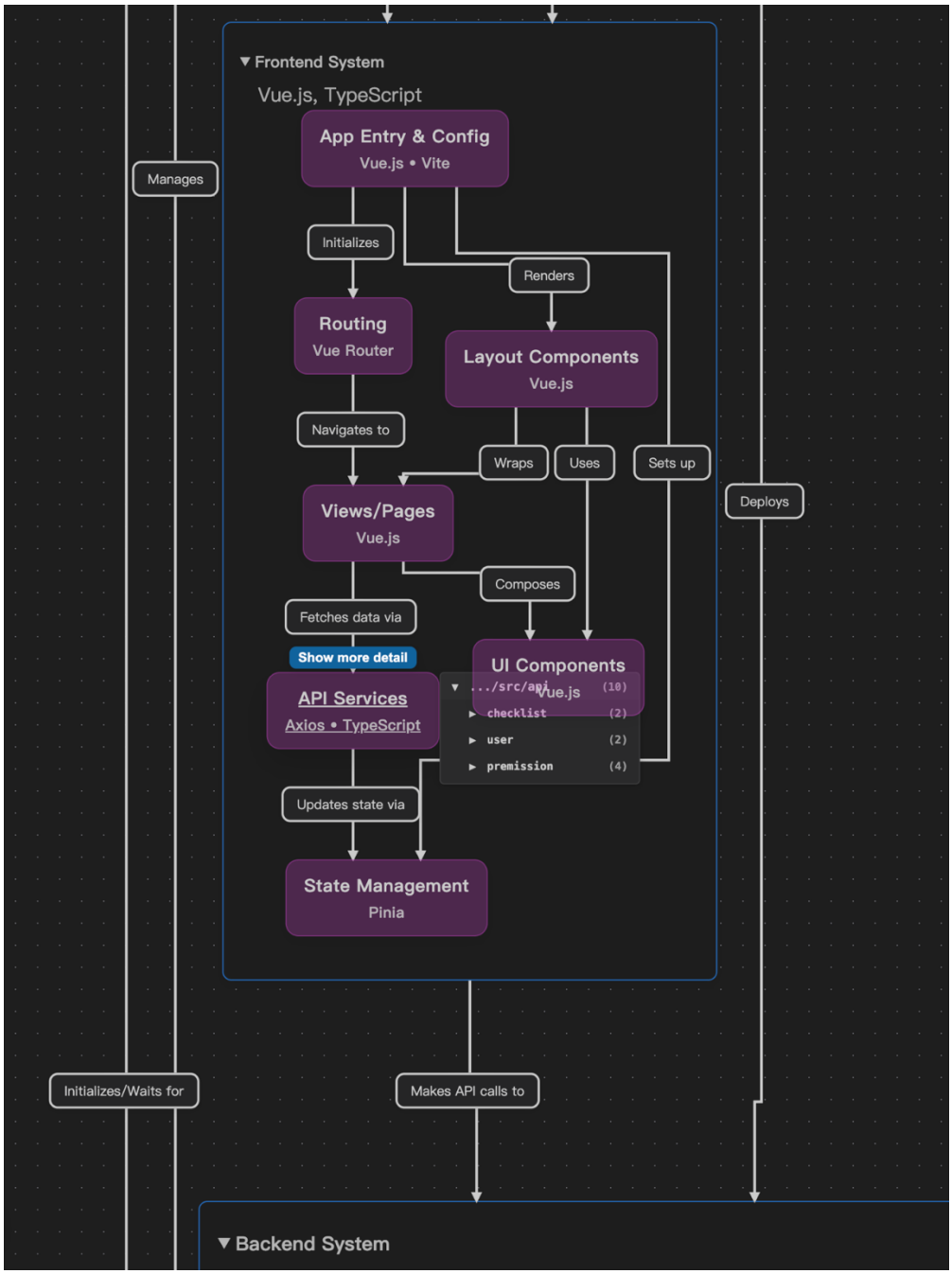



Deployment & Infrastructure
Docker, Bash, GitHub Actions
Deployment Scripts
Bash • YAML
Runs
Builds
Docker Compose
Docker Compose
Orchestrates
Docker Containers
Docker
User
External Actor
Deploys
Interacts with
Frontend System
Vue.js, TypeScript
App Entry & Config
Vue.js • Vite
Manages
Initializes
Renders
Routing
Vue Router
Layout Components
Vue.js
Navigates to
Wraps
Uses
Sets up
Views/Pages
Vue.js
Deploys
Composes
Fetches data via
UI Components
Vue.js
API Services
Axios • TypeScript
Updates state via
State Management
Pinia
Initializes/Waits for
Makes API calls to
Backend System
FastAPI, Python
API Core
FastAPI
Uses
Includes
Authentication & Authorization
FastAPI, Python
API Endpoints (Routers)
FastAPI
User Manager
Python
Token Handling
Python
Uses
Backend Utilities
Python
Interacts with
Manages users in
Database Models
SQLAlchemy
Persists data in
Uses AI services from
Database System
PostgreSQL, Alembic
Vertex AI
Google Cloud
Alembic Migrations
Alembic • Python
Manages schema for
PostgreSQL Database
PostgreSQL

Details:
Deployment:

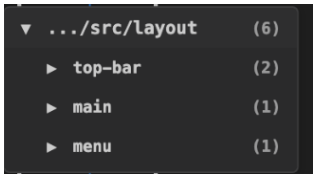


Frontend:

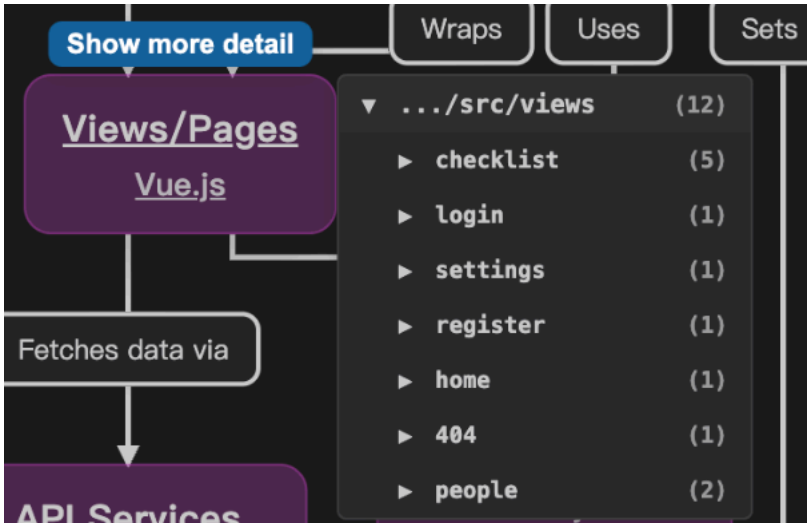


Layout Components:

For “Layout Components “on the pages:

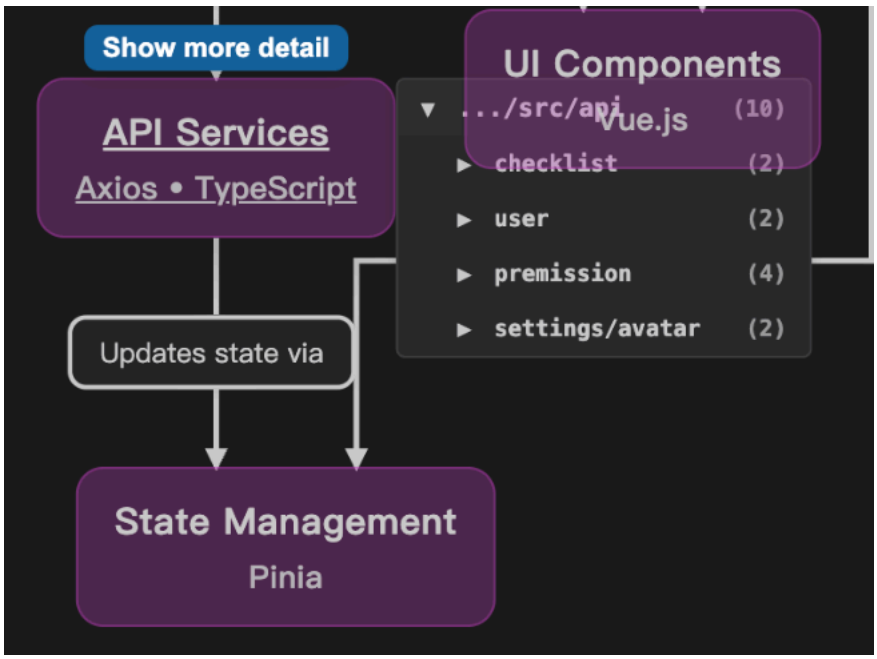


View, Pages:



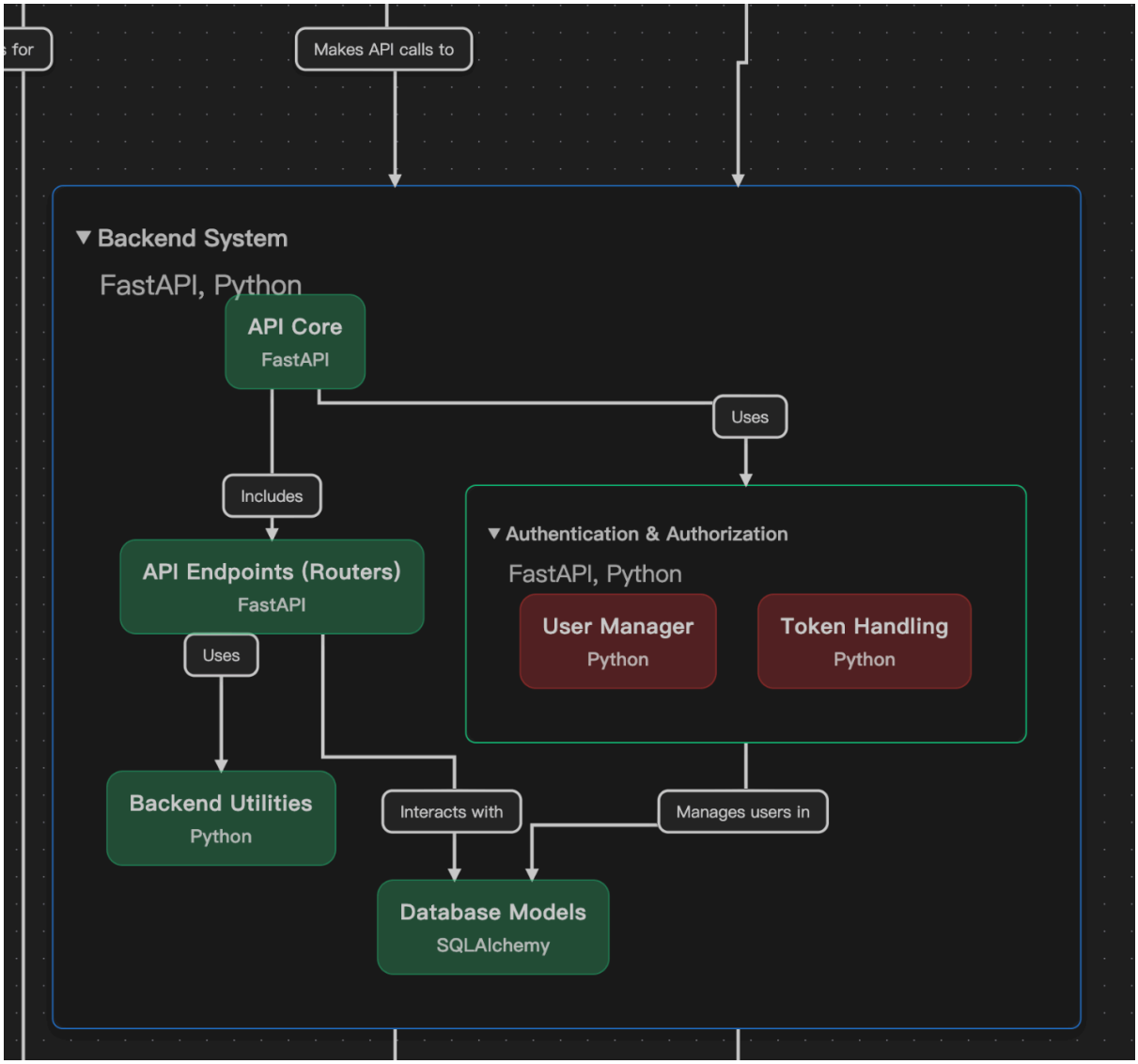
The pages views for all pages in the project.

API Services:
Call backend APIs

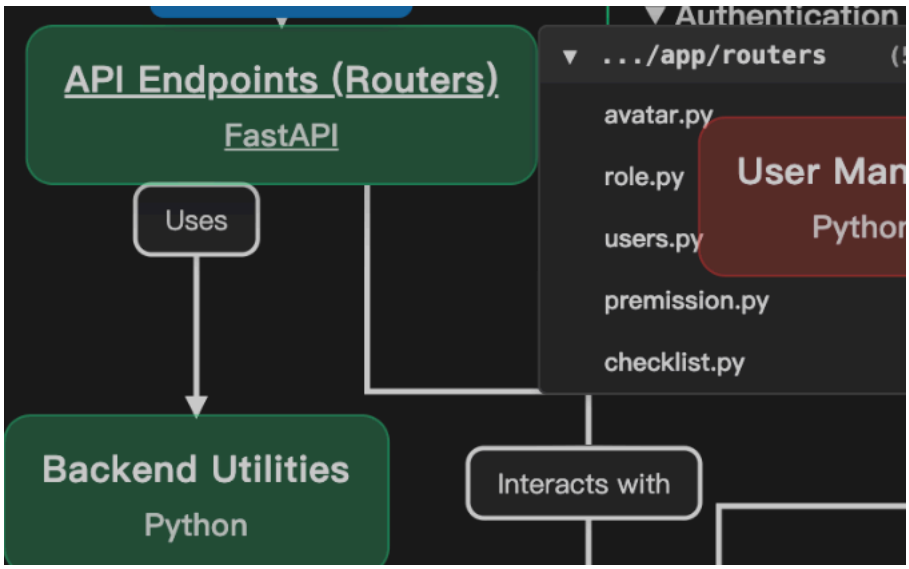


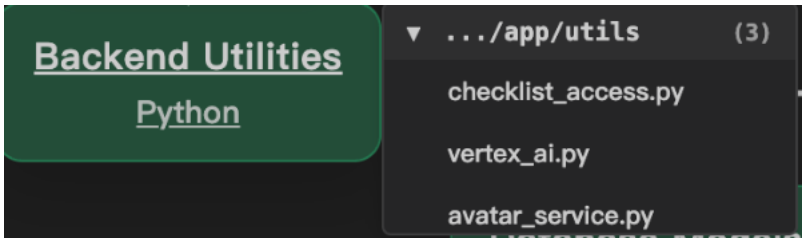
State Management:
Pinia, hold user info, dynamic routing info

Backends:



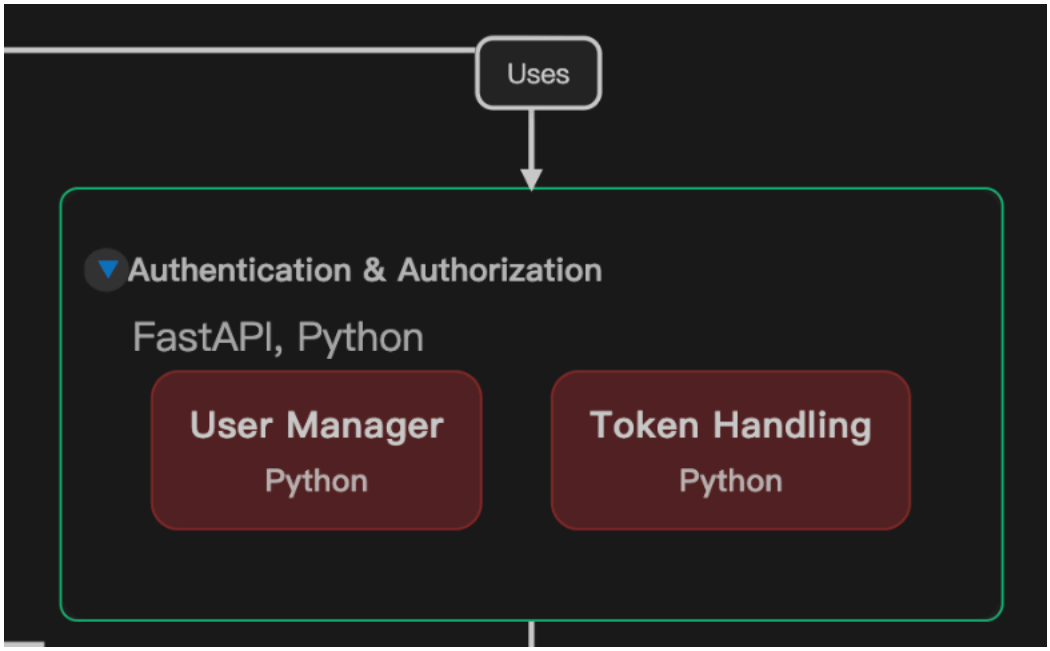
API Endpoints:





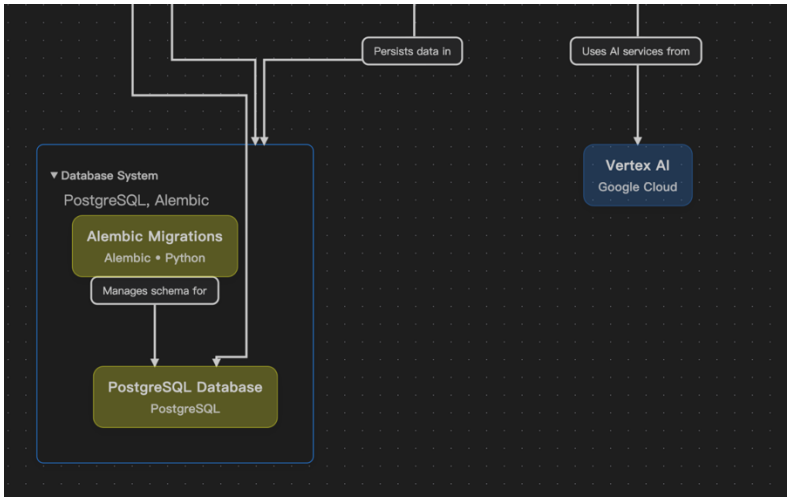
APIs for all functions and processing frontend requests, communicate with backend.

FastAPIUsers



For authorization and authentication

PostgreSQL Database / AI services



Hosting/deployment environments (local, staging, production).

Docker Compose Overview

Docker Compose serves as the central configuration for the application's services, networks, and volumes. Its primary responsibilities include:

- Service Definition: Declaring each service (e.g., `backend`, `frontend`, `db`) as a distinct container, specifying its image, build context, ports, volumes, and environment variables.
- Network Configuration: Establishing internal networks for inter-service communication, ensuring services can discover and communicate with each other securely.
- Volume Management: Defining persistent storage for data (e.g., database data) to ensure data is not lost when containers are stopped or removed.
- Environment Management: Providing a consistent way to manage environment-specific configurations for development and production.

Environment-Specific Configuration

The project utilizes two main Docker Compose files to manage different environments:

- **[docker-compose.yml]**(Capstone/docker-compose.yml): Configures the application for the **development environment**.
- **[docker-compose.prod.yml]**(Capstone/docker-compose.prod.yml): Configures the application for the **production environment**.

Development Environment Configuration

`db` Service:

- Purpose: Provides a PostgreSQL database instance for the backend service.
- Internal Parts: Uses the `postgres:15-alpine` Docker image.
- External Relationships: Exposes port `5432` to the host machine for direct access (e.g., via database clients) and is accessible by the `backend` service via the internal Docker network.
- Configuration Details:
    - - `image: postgres:15-alpine`
      - `ports: - "5432:5432"`
      - `volumes: - db:/var/lib/postgresql/data`
      - `environment`: Sets `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`.

`backend` Service:

- Purpose: Runs the FastAPI backend application.
- Internal Parts: Builds from the backend/Dockerfile, mounts the local backend code as a volume for live reloading.

- External Relationships: Depends on the `db` service, connects to it using the `db` hostname. Exposes port `8000` to the host.
- Configuration Details:
    - - `build: ./backend`
      - `ports: - "8000:8000"`
      - `volumes: - ./backend:/app`
      - `env_file: .env`
      - `depends_on: - db`
      - `command: bash -c "sh wait-for-postgres.sh db && alembic upgrade head && uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload"`

`frontend` Service:

- Purpose: Runs the Vue.js frontend application.
- Internal Parts: Builds from the frontend/Dockerfile, mounts the local frontend code as a volume.
- External Relationships: Exposes port `5173` (Vite's default dev port).
- Configuration Details:
    - - `build: ./frontend`
      - `ports: - "5173:5173"`
      - `volumes: - ./frontend:/app`
      - `env_file: .env`
      - `command: npm run dev -- --host 0.0.0.0`
    -

Production Environment Configuration

`db` Service:

- Purpose: Provides a PostgreSQL database instance for the backend service.
- Internal Parts: Uses the `postgres:15-alpine` image.
- External Relationships: Does not expose port `5432` to the host directly. Only accessible internally by the `backend` service.
- Configuration Details:
    - - `image: postgres:15-alpine`
      - `volumes: - db:/var/lib/postgresql/data`
      - `environment`: Loads from `.env.prod`.

`backend` Service:

- Purpose: Runs the FastAPI backend in production.
- Internal Parts: Builds from backend/Dockerfile (or uses pre-built). Uses Dockerfile.migrate for migrations.
- External Relationships: Depends on `db`. Exposes port `8000` to the host.
- Configuration Details:
    - - `build: ./backend`
      - `ports: - "8000:8000"`
      - `env_file: .env.prod`
      - `depends_on: - db`
      - `command: bash -c "sh wait-for-postgres.sh db && alembic upgrade head && gunicorn app.main:app --workers 4 --worker-class uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000"`

`frontend` Service:

- Purpose: Serves the compiled Vue.js frontend.
- Internal Parts: Builds from frontend/Dockerfile.prod and serves with Nginx.
- External Relationships: Exposes port `80` to the host.
- Configuration Details:
    - - `build: ./frontend`
      - `ports: - "80:80"`
      - `env_file: .env.prod`

Integration with Other System Components

Docker Compose integrates with other system components by:

- Containerization: Each major component (`backend`, `frontend`, `db`) is encapsulated in its own container with respective Dockerfiles.
- Networking: Services communicate over an internal Docker network. For example, backend connects to db via hostname `db`.
- Volume Management: PostgreSQL data is persisted via a named volume (`db`).
- Environment Variables: Config loaded from `.env` (development) and `.env.prod` (production).
- Dependency Management: `depends_on` ensures correct order; `wait-for-postgres.sh` ensures db readiness.
- Build Process: Custom service images are built from contexts (`backend`, `frontend`).