Lab8

Introduction

- The project is implemented as a modern web application based on FastAPI (Python) using asynchronous architecture. The server side is built using SQLAlchemy ORM and PostgreSQL for data management. The architecture follows the principles of the REST API with a clear division into endpoints of different domains (tests, materials, users, etc.). The authentication system is implemented using JWT tokens and bcrypt for password hashing. Database migrations are managed through Alembic, and interaction with the database is carried out through the asyncpg driver.

---

1. Basis for development.

- This software product is designed to automate the management of educational data within a school environment. The system covers key entities such as subjects, classes, schools, and users (students, teachers, and administrators), providing centralized storage and processing of academic information.
- The product addresses the need for a unified platform that enables structured management of subjects and classes, monitoring of academic performance, and transparent access to educational data for all stakeholders. The development is driven by the growing demand for digitalization in educational institutions and the necessity to streamline administrative workflows while improving data accuracy and analytical capabilities.
- The project is built using modern technologies: the frontend is implemented with Vue 3 and Vite, while the backend is developed using FastAPI with asynchronous interaction through SQLAlchemy. The architecture includes REST API standards, Pydantic-based data validation, and a layered structure (repositories, services, schemas) to ensure maintainability and scalability. Access control, error handling, and role-based authorization are also integrated.
- This project is developed as part of an educational initiative, aligned with coursework or a diploma project, and focuses on automating typical processes within an academic institution.

---

2. Purpose of development.

- The primary goal of the project is to create an integrated educational platform that streamlines the management of schools, users, academic classes, subjects, and digital learning materials. The system is designed to address key tasks such as user authentication and authorization, administration of educational entities, progress tracking, and test management. Target users include school administrators, teachers, students, and superadmins. The platform automates routine administrative processes, enhances data accessibility, and improves communication between stakeholders, thereby increasing operational efficiency and supporting modern educational needs.

---

3. Requirements for the program or software product.

Functional Requirements

Main user scenarios:

- Registration and authentication of users (students, teachers, administrators, super administrators).
- User management: viewing, filtering, creating, editing, activating/deactivating users.
- Management of educational organizations (schools, classes, subjects).
- Creation, editing, and publishing of learning materials (PDF, video, text, etc.).
- Conducting and taking tests, tracking attempts and results.
- Analytics of student performance and progress, as well as statistics by classes and subjects.
- Role-based access control to system functions (e.g., only the super administrator can create users).

Key system functions:

- REST API for all core entities: users, schools, classes, subjects, tests, materials.
- Asynchronous request processing (FastAPI + SQLAlchemy).
- JWT-based authentication and role-based authorization.
- Data filtering and pagination (e.g., user lists, subjects).
- Integration with a Vue.js (SPA) frontend with support for dynamic routing and components.
- Management of test attempts, including time limits and maximum number of attempts.

API/service behavior specifics:

- All major operations are implemented via REST endpoints with strict data validation (Pydantic).
- Protected routes enforce access checks and role verification.
- The API design supports extensibility: new material types, roles, and usage scenarios can be added.
- Errors and exceptions are handled consistently with appropriate HTTP status codes.

Non-functional Requirements

Performance:

- Asynchronous request handling (FastAPI, Async SQLAlchemy).
- Optimized database queries with pagination support.

Reliability:

- Input validation on all layers (Pydantic).
- Proper error and exception handling, with informative responses.
- Logging of key operations and failures.

Security:

- JWT authentication and secure password hashing (bcrypt).
- Role-based access restriction for API endpoints.
- Permission checks implemented at the endpoint level via dependency injection (Depends).
- CORS configuration for protection against unauthorized cross-origin requests.

Constraints:

- The system is designed for educational institutions (schools, universities).
- Supports only predefined roles and material types.
- Constraints on data structure (e.g., grade_level must be between 0 and 11).

Environment requirements:

- Server: Python 3.10+, FastAPI, SQLAlchemy, PostgreSQL.
- Client: Node.js, Vue.js 3+, Vite.
- Database: PostgreSQL.
- Deployment: supports both local and cloud hosting; CORS must be configured for frontend interaction.

---

4. Requirements for software documentation.

Types of Documentation Required

- **User Documentation:** Guides and manuals for end-users, including students, teachers, and administrators.
- **Technical Documentation:** Materials for developers, system administrators, and maintainers.
- **API Documentation:** Detailed description of REST endpoints, data models, authentication, and error handling.
- **System Architecture Documentation:** Diagrams and explanations of system components, data flows, and integration points.
- **SDLC Artifacts:** Requirements specifications, design documents, test plans, deployment guides, and maintenance procedures.

Contents of User Documentation

- Step-by-step instructions for registration, authentication, and role-based access.
- Descriptions of main user scenarios: managing classes, subjects, materials, and tests.
- Guidance on using analytics and progress tracking features.
- Troubleshooting section and FAQ.
- Visual aids: screenshots, diagrams, and sample workflows.

Contents of Technical Documentation

- System architecture overview, including backend, frontend, and database structure.
- API reference with endpoint descriptions, request/response formats, and authentication mechanisms.
- Data model definitions for all entities (users, subjects, materials, tests, etc.).
- Deployment and environment setup instructions.
- Maintenance procedures and update guidelines.
- Source code structure and conventions.

SDLC Artifacts to be Included

- Requirements specification documents.
- System and software design documentation.
- Implementation notes and code comments.
- Test plans, test cases, and results.
- Deployment and rollback procedures.
- Maintenance and support guidelines.

Requirements for Completeness, Accuracy, and Format

- Documentation must be comprehensive, covering all system features and scenarios.
- Information should be accurate, up-to-date, and reflect the actual implementation.
- Format must be clear, structured, and consistent, using headings, lists, and diagrams where appropriate.
- All documentation should be versioned and maintained alongside the codebase.
- Language must be formal and precise, suitable for academic and professional use.

---

5. Feasibility study.

Technical Feasibility

- **Technologies Used:**
  - Backend: Python 3.10+, FastAPI, SQLAlchemy (async), Alembic, PostgreSQL.
  - Frontend: Vue.js 3, Vite, Nuxt 4 (elFront), Pinia, TailwindCSS, Axios.
  - API: RESTful endpoints with JWT authentication and role-based access control.
- **Architecture:**
  - Modular backend with separated services, repositories, and schemas.
  - Asynchronous request handling for scalability.
  - Frontend SPA with dynamic routing and state management.
  - CORS and proxy configuration for secure API access.
- **Realism:**
  - All technologies are mature, widely supported, and suitable for educational platforms.
  - The architecture supports extensibility (adding new entities, roles, materials).

Economic Feasibility

- **Time Estimate:**
  - Initial development: 3-4 months (backend, frontend, basic analytics).
  - Testing and deployment: 1 month.
- **Effort:**
  - Team: 2 backend developers, 2 frontend developers, 1 QA, 1 DevOps.
- **Resources:**
  - Cloud hosting for backend and database.
  - Minimal hardware requirements for client devices.
  - Open-source libraries reduce licensing costs.

Operational Feasibility

- **Usage:**
  - Designed for schools, teachers, and students to manage classes, subjects, materials, and progress.
  - Role-based access ensures secure and relevant functionality for each user type.
- **Support:**
  - Modular codebase allows easy updates and bug fixes.
  - Documentation and README files for setup and deployment.
  - Automated migrations (Alembic) for database changes.
  - Frontend and backend can be deployed independently.

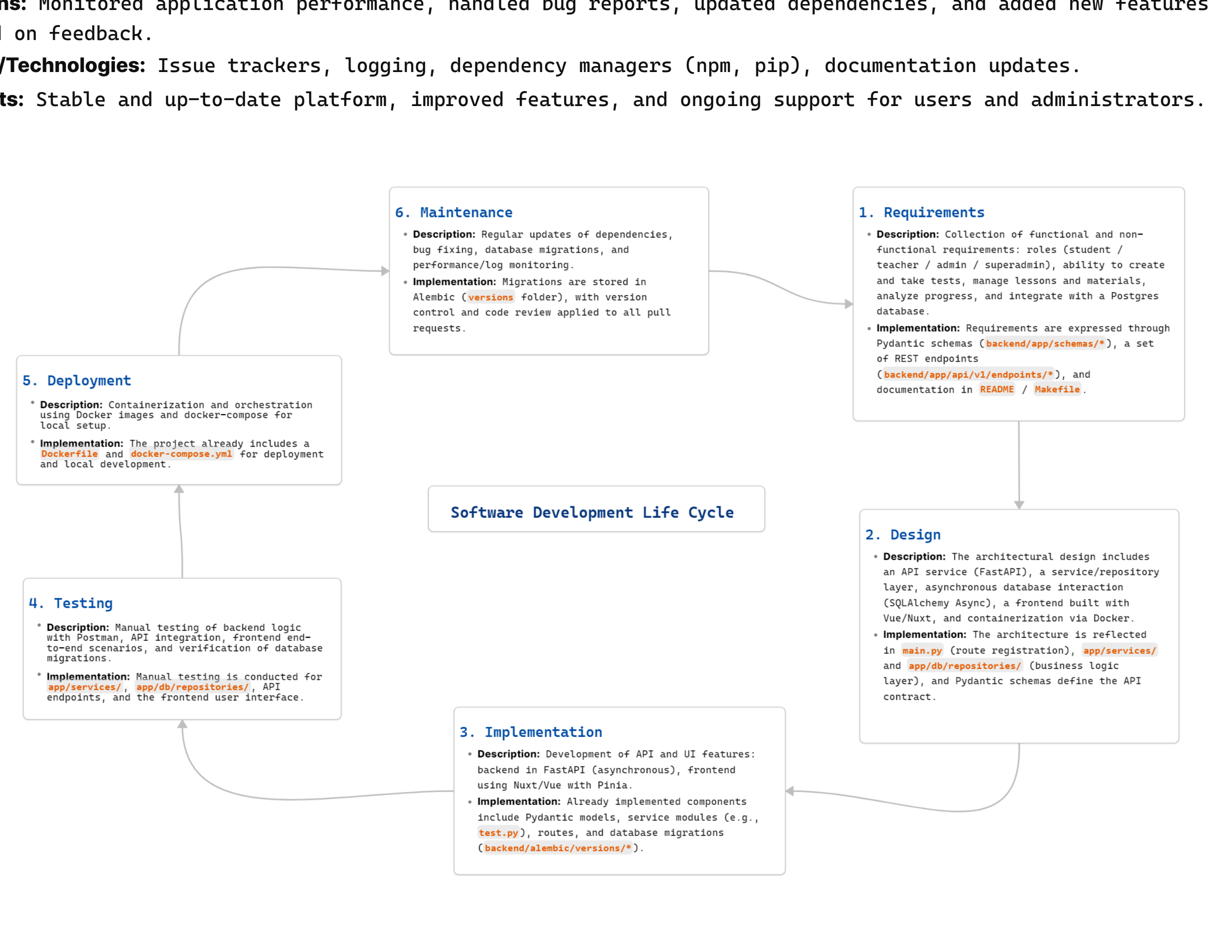
Risks and Mitigation

1. **Data Security Breach**
   - Mitigation: JWT authentication, password hashing, CORS, role checks in endpoints.
2. **Performance Bottlenecks**
   - Mitigation: Asynchronous DB queries, pagination, optimized endpoints.
3. **Integration Issues (Frontend/Backend)**
   - Mitigation: Strict API contracts (Pydantic schemas), CORS/proxy setup, automated tests.
4. **Scalability Limits**
   - Mitigation: Stateless backend, cloud-ready deployment, modular architecture.
5. **User Error or Misuse**
   - Mitigation: Input validation, informative error messages, role-based restrictions.

---

6. Stages and stages of development.

1. **Requirements**
   - **Actions:** Collected and analyzed requirements for an educational platform supporting tests, lessons, materials, analytics, and user management.
   - **Tools/Technologies:** Meetings, documentation, user stories, Notion.
   - **Results:** Defined functional and non-functional requirements, created initial feature list and user roles (student, teacher, admin).
2. **Design**
   - **Actions:** Designed system architecture, database schema, API endpoints, and user interface wireframes.
   - **Tools/Technologies:** ER diagrams, Figma for UI mockups, Alembic for migration scripts, Pydantic for schema design.
   - **Results:** Database models for entities (Test, Lesson, Material, User, etc.), RESTful API structure, Vue component structure, and navigation routes.
3. **Implementation**
   - **Actions:** Developed backend (FastAPI, SQLAlchemy), frontend (Vue 3, Vite), and integrated API endpoints. Implemented core features: test management, lesson materials, analytics, authentication.
   - **Tools/Technologies:** Python, FastAPI, SQLAlchemy, Alembic, Vue 3, Vite, Axios, Vue Router, Vuex, Git for version control.
   - **Results:** Working backend and frontend applications, CRUD operations for main entities, user authentication, and role-based access.
4. **Testing**
   - **Actions:** Manual and automated testing of API endpoints, frontend components, and user flows. Validated data integrity and business logic.
   - **Tools/Technologies:** Pytest, Vee-Validate, Yup, browser dev tools, Postman.
   - **Results:** Verified functionality, fixed bugs, ensured stable releases, and improved user experience.
5. **Deployment**
   - **Actions:** Deployed backend and frontend applications to local and production environments. Configured CORS, environment variables, and build scripts.
   - **Tools/Technologies:** Vite build, Docker (optional), cloud hosting (optional), GitHub Actions (optional).
   - **Results:** Accessible platform at designated URLs, automated build and deployment pipelines, production-ready builds.
6. **Maintenance**
   - **Actions:** Monitored application performance, handled bug reports, updated dependencies, and added new features based on feedback.
   - **Tools/Technologies:** Issue trackers, logging, dependency managers (npm, pip), documentation updates.
   - **Results:** Stable and up-to-date platform, improved features, and ongoing support for users and administrators.



---

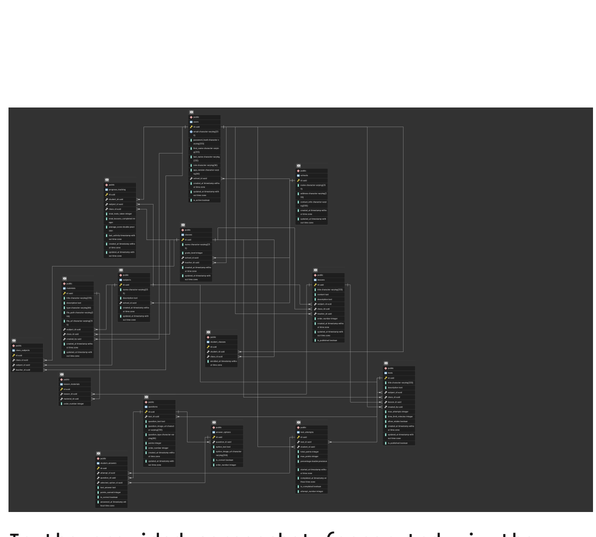
7. Order of control and acceptance.

1. **System Functionality Verification**
   System operability is verified through manual and automated functional testing. Core modules (API, frontend, authentication, content management) must match the approved requirements.
2. **Acceptance Criteria**
   - All functional requirements fully implemented.
   - Mandatory test scenarios passed, including edge cases.
   - No critical or high-severity defects.
   - Performance indicators meet expected benchmarks.
   - Frontend and backend properly integrated.
3. **Mandatory Tests**
   - Unit and integration tests for backend logic and API endpoints.
   - End-to-end scenarios for key user workflows.
   - Security checks for authentication, authorization, and data protection.
   - Data integrity validation across system components.
   - Cross-browser and device compatibility verification.
4. **Requirements for Stability and Security**
   - Stable operation under expected load.
   - Accurate and consistent data processing.
   - No unauthorized access to sensitive operations.
   - Graceful handling of invalid input.
   - Dependencies must be current and free of known vulnerabilities.
5. **Final Acceptance Format**
   Final acceptance is performed by the Customer or Commission and includes:
   - Review of test reports and defect logs.
   - Live demonstration according to acceptance scenarios.
   - Signing of the acceptance certificate upon successful verification.
   - Recording any remaining issues and agreeing on resolution timelines.

---

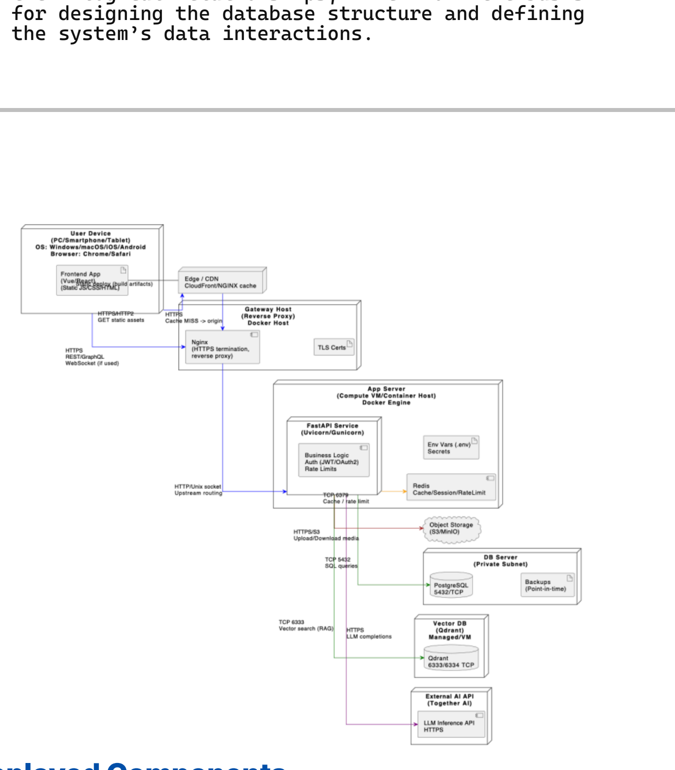
8. Applications.

ERD



In the provided screenshot (generated via the pgAdmin DBMS), the system's Entity-Relationship (ER) model is presented.
The diagram highlights the core data entities and their logical relationships, which form the basis for designing the database structure and defining the system's data interactions.

Deployment Diagram



Description of Nodes and Deployed Components

1. **User Device (Client Device)**
   Purpose: Entry point for the user. Runs the frontend application in the browser.
   Deployed: Static frontend files (JS/CSS/HTML) are loaded into the browser from the CDN/gateway.
2. **Edge/CDN**
   Purpose: Caching and fast global delivery of static assets, reducing load on the backend.
   Deployed: Frontend build artifacts (static files). Handles TLS termination or proxies traffic further.
3. **Gateway Host (Nginx)**
   Purpose: HTTPS termination, reverse proxy, routing to FastAPI, request size limits, CORS, gzip compression.
   Deployed: Nginx with TLS certificates.
4. **App Server (FastAPI Service)**
   Purpose: Business logic, JWT/OAuth2 authentication, API endpoints, integrations with the database and vector search, LLM calls.
   Deployed: FastAPI container (Uvicorn/Gunicorn), environment variables/secrets (.env), optional Redis for caching and rate limiting.
5. **Database Server (PostgreSQL)**
   Purpose: Storage of transactional data (users, lessons, tests, attempts, results, progress analytics).
   Deployed: PostgreSQL with backups (snapshot/PITR).
6. **Vector DB (Qdrant)**
   Purpose: Vector storage for RAG-based search over materials (questions, answers, notes, library content).
   Deployed: Qdrant (managed or VM), ports 6333/6334.
7. **External AI API (Together AI)**
   Purpose: LLM inference over HTTPS for generating answers, explanations, and hints in the learning chat.
   Deployed: External service (not hosted on our infrastructure).
8. **Object Storage (S3/MinIO, optional)**
   Purpose: Storage of media files (lesson images, attachments, exports).
   Deployed: Storage bucket accessible from the backend via the S3 API.