CS 319 Object-Oriented Software Engineering

Fall 2024

S2T1

D4

MeetBilkent

Sümeyye Acar 22103640

Ömer Edip Aras 22203238

Ömer Bıçakcıoğlu 22103294

Muhammed Furkan Başıbüyük 22102101

Table of Contents

1. Design Goals

Performance

Rationale: The system must handle multiple concurrent users efficiently, especially during peak periods like high school fairs or tour registrations.

Tradeoff: Achieving high performance may increase development complexity (e.g., through caching or database optimizations), but this is critical for delivering a seamless user experience.

Scalability

Rationale: As the university's outreach efforts to students increase, the system should be able to handle a growing number of users and events without significant changes to the architecture.

Tradeoff: Implementing scalable features, such as load balancers or database sharding, may increase costs and require additional development effort. However, scalability is necessary for the system's longevity.

Maintainability

Rationale: A clear separation of concerns between the backend and frontend ensures that the system is easy to extend and debug over time.

Tradeoff: Focusing on maintainability might slow down initial development as best practices like modularization and proper documentation are enforced. However, this ensures smoother updates and ease of coding for developers with less-to-none experince

User Experience (UX)

Rationale: The system is aimed at prospective students, so creating a responsive, intuitive, and visually appealing interface is essential for engagement and ease of use.

Tradeoff: Enhancing UX may extend the development timeline due to iterative testing and refinements, but it is a crucial part of user satisfaction in terms of usability.

Security

Rationale: Protecting user data, including personal and login credentials, is important to maintain trust and comply with data protection regulations.

Tradeoff: Implementing strong security measures (e.g., encryption, token-based authentication) may slightly reduce performance due to encryption process. Nevertheless, security is prioritized to protect sensitive information.

2. Connectors

- ***SQLAlchemy: FastAPI backend - PostgreSQL database***
  SQLAlchemy provides simplified queries, data retrieval, and schema management via its ORM layer for efficient interactions between FastAPI and PostgreSQL.
- ***Axios: React frontend - FastAPI backend***
  Axios connects the React frontend to the FastAPI backend by sending HTTP requests to REST API endpoints and receiving responses asynchronously.
- ***REST APIs: React frontend - FastAPI backend***
  REST APIs connect the React frontend with the FastAPI backend by providing standardized endpoints for data exchange and CRUD operations.
- ***Postman: Test REST API endpoints***
  Postman enabled us to test backend endpoints without any frontend or user interface.

3. Architectural Style

We use **Layered Architecture** in development because it clearly and logically organizes the system. It separates the system into three main layers:

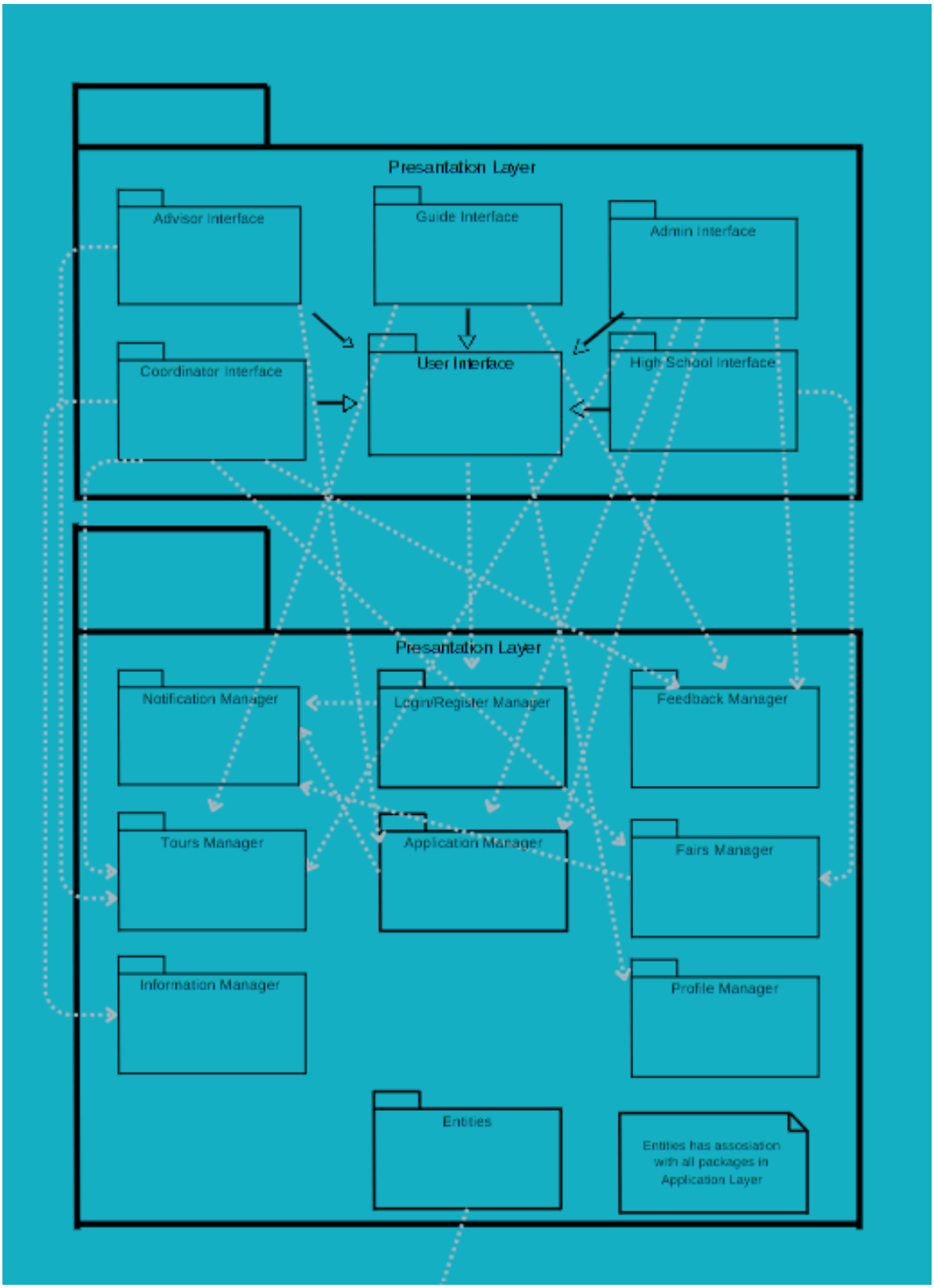
- **Frontend (React)**: Responsible for the user interface and interaction with the user.
- **Backend (FastAPI)**: To implement the business logic that moderates interactions between the front end and the database.
- **Database (PostgreSQL)**: Handles data persistence using PostgreSQL.

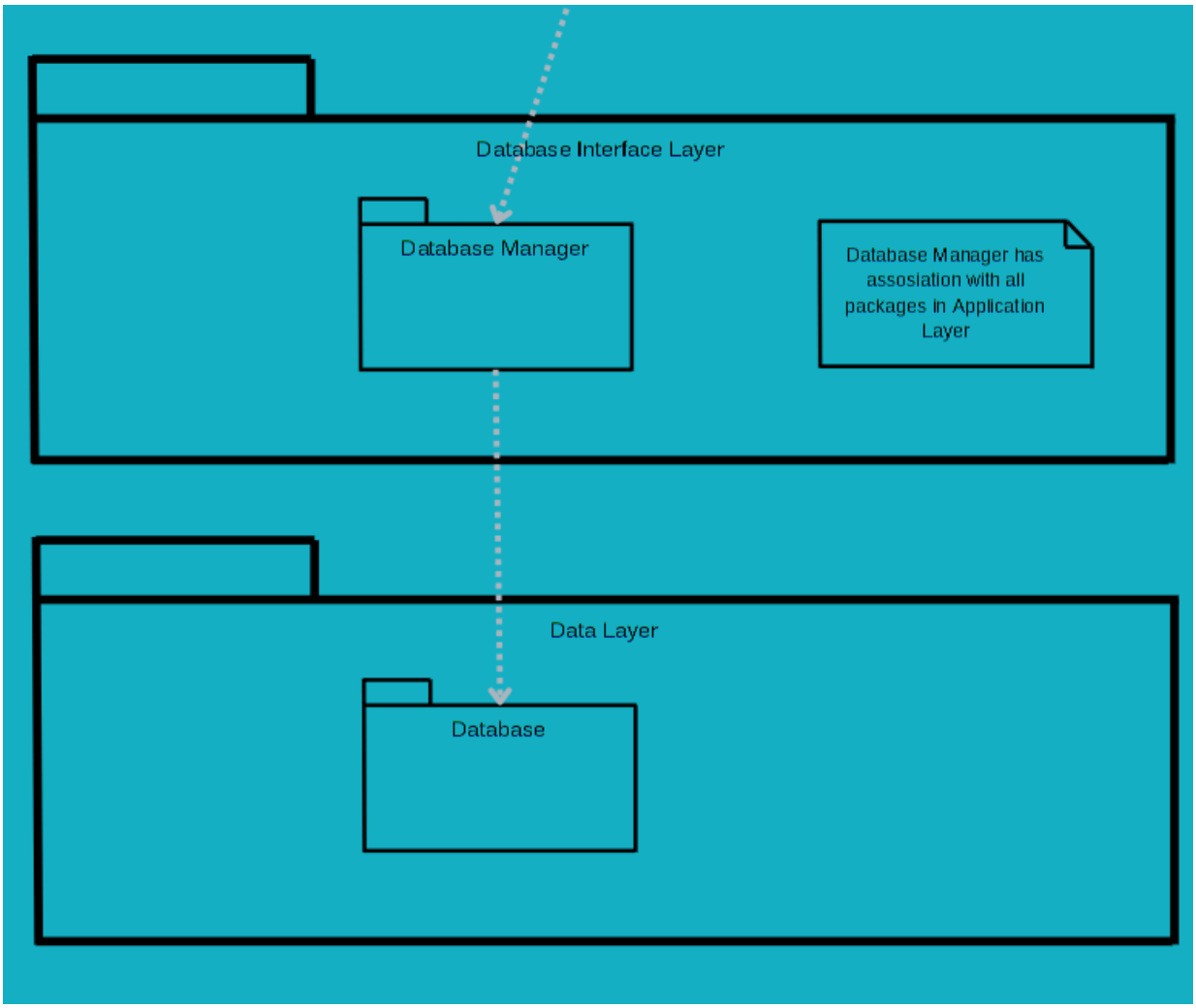
We chose this structure because of its simplicity and maintainability. Each layer is responsible for a specific role, making the system modular and easier to manage. This makes it possible for different team members to work on separate parts of the application without interfering with each other. Thanks to this structure, when one layer has an error or is changed, we can manage the relevant layer without disrupting the structure of the different layers.
When the system transitions to deployment, it will evolve into a **Client-Server Architecture**. In this setup:

- **The client (React)** will run on users' devices, such as browsers, and handle user interactions.
- **The server (FastAPI & Database)** will process requests from the client and handle data-related operations by communicating with the database.
-

We chose this architecture for deployment because it fits the distributed nature of modern web applications, allowing the front and back end to scale independently. Tools such as Reverse-proxy and WSGI servers(NGINX, Gunicorn, etc.) are easily integrated to manage data efficiently and enable smooth performance in that structure. This setup will enhance security by adding layers like reverse proxy protection, TLS encryption, and rate limiting, which makes the system flow secure and efficient.

4. Subsystem Decomposition Diagram




Database Interface Layer
Database Manager
Database Manager has assosiation with all packages in Application Layer
Data Layer
Database