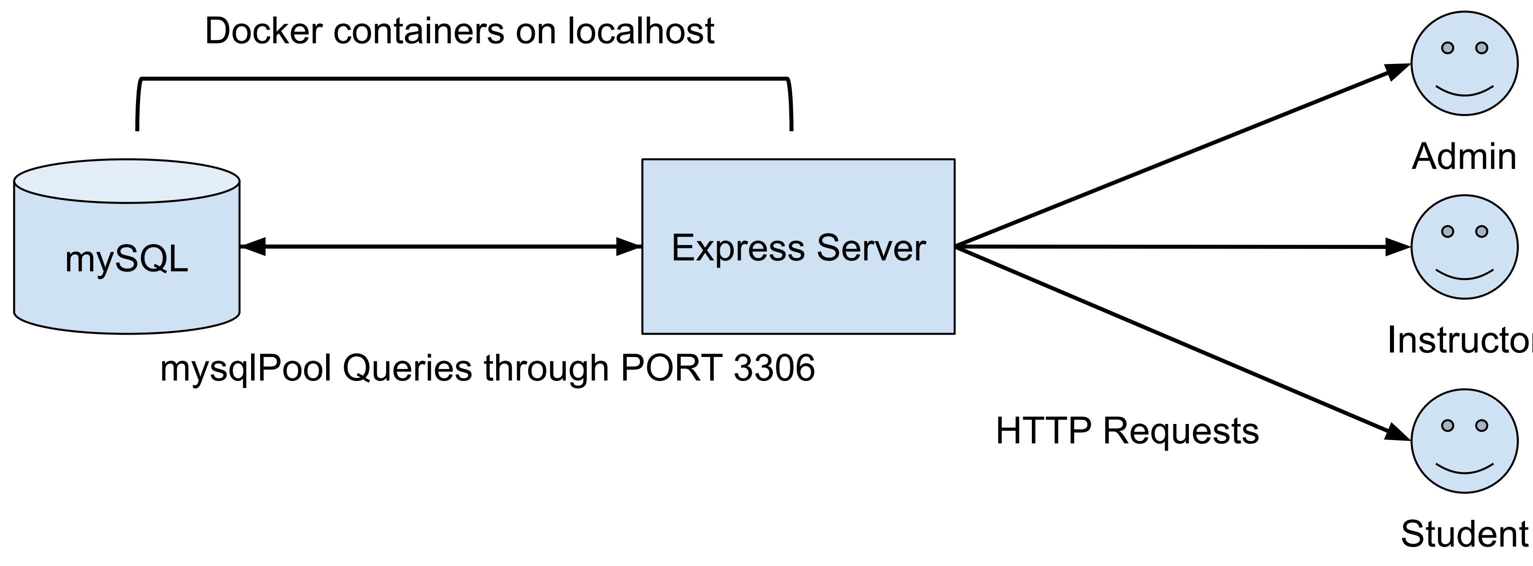
Final Project: Architecture Document

By Melissa Swearingen, Samson DeVol, Ian Snyder, Jonah Broyer

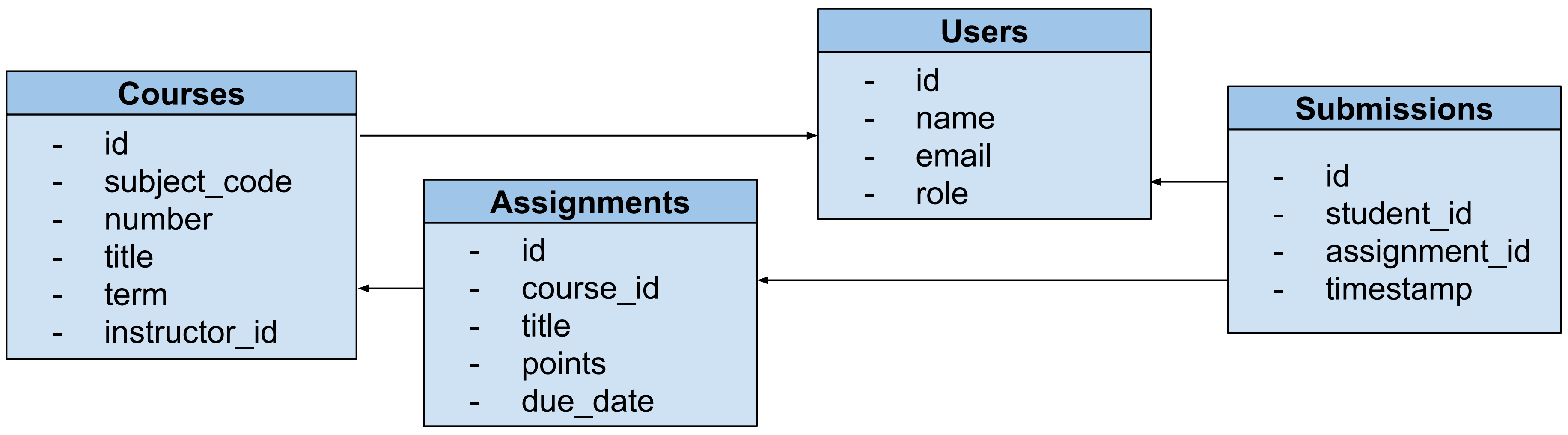
API Architecture Diagram

This should, at a high level, depict the major components of your API (e.g. API server(s), database server(s), etc.) and indicate the relationships between these components.


Docker containers on localhost
mySQL
Express Server
mysqlPool Queries through PORT 3306
HTTP Requests
Admin
Instructor
Student

API Data Layout

This should describe, either pictorially or in text, how application data is organized within your API database (e.g. data schema, links between entities, etc.).



API Design Reflection

The chosen architecture and data layout was designed to meet the requirements of an API system that manages users, courses, assignments, and submissions. Below we will reflect on the design and identify what worked well and what could be improved upon, as well as considerations for future changes.

What worked well:

1. Role-based User Model: The inclusion of user roles (admin, instructor, student) allows for fine-grained access control and differentiates permissions for each role. This design enables the API to enforce appropriate authorization and access restrictions.
2. Entity Relationships: The relationships between entities can be defined using foreign keys.
3. File Storage: The data layout includes a reference to the file associated with each submission. This allows files to be stored separately, reducing redundancy, and enabling efficient file management.

Areas for improvement:

1. Authentication and Security: The current design does not explicitly address authentication and security mechanisms. To enhance the API's security, additional considerations should be given to implementing secure authentication methods such as token-based authentication.
2. API Documentation: It would be beneficial to invest time in designing comprehensive API documentation, including clear guidelines on usage, request/response formats, and error handling. This would air developers using the API and foster ease of integration.
3. Event-Driven Architecture: For real-time updates and better integration with other systems, an event-driven architecture could be considered. This would allow components of the API to react to specific events and propagate changes across the system.

Overall, the chosen architecture and data layout provides a solid foundation for the API system. However, there is room for improvement in terms of security, scalability, and performance optimizations. By addressing these areas and considering modular, event-driven approaches, the API design can be further enhanced to meet evolving requirements.