Tarpaulin API Architecture and Design Document

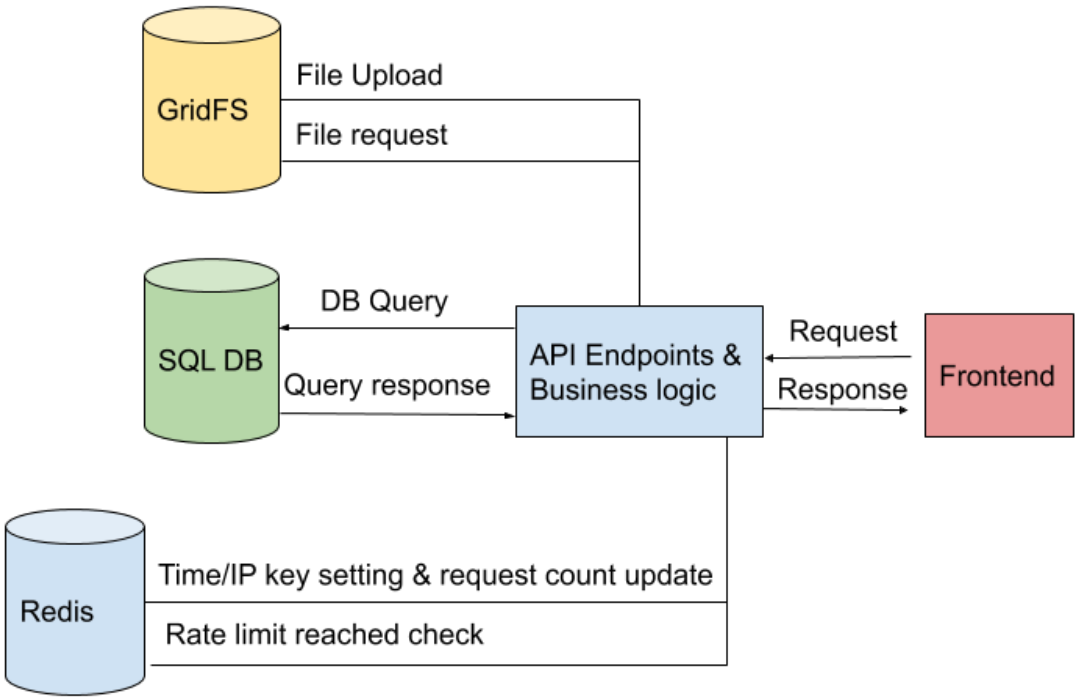
Group 32

- Khuong Luu (luukh)
- Phi Luu (luuph)
- Aidan Grimshaw (grimshaa)

GitHub repository: https://github.com/osu-cs493-sp19/final-project-do-it-later.

1. API Architecture Diagram

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



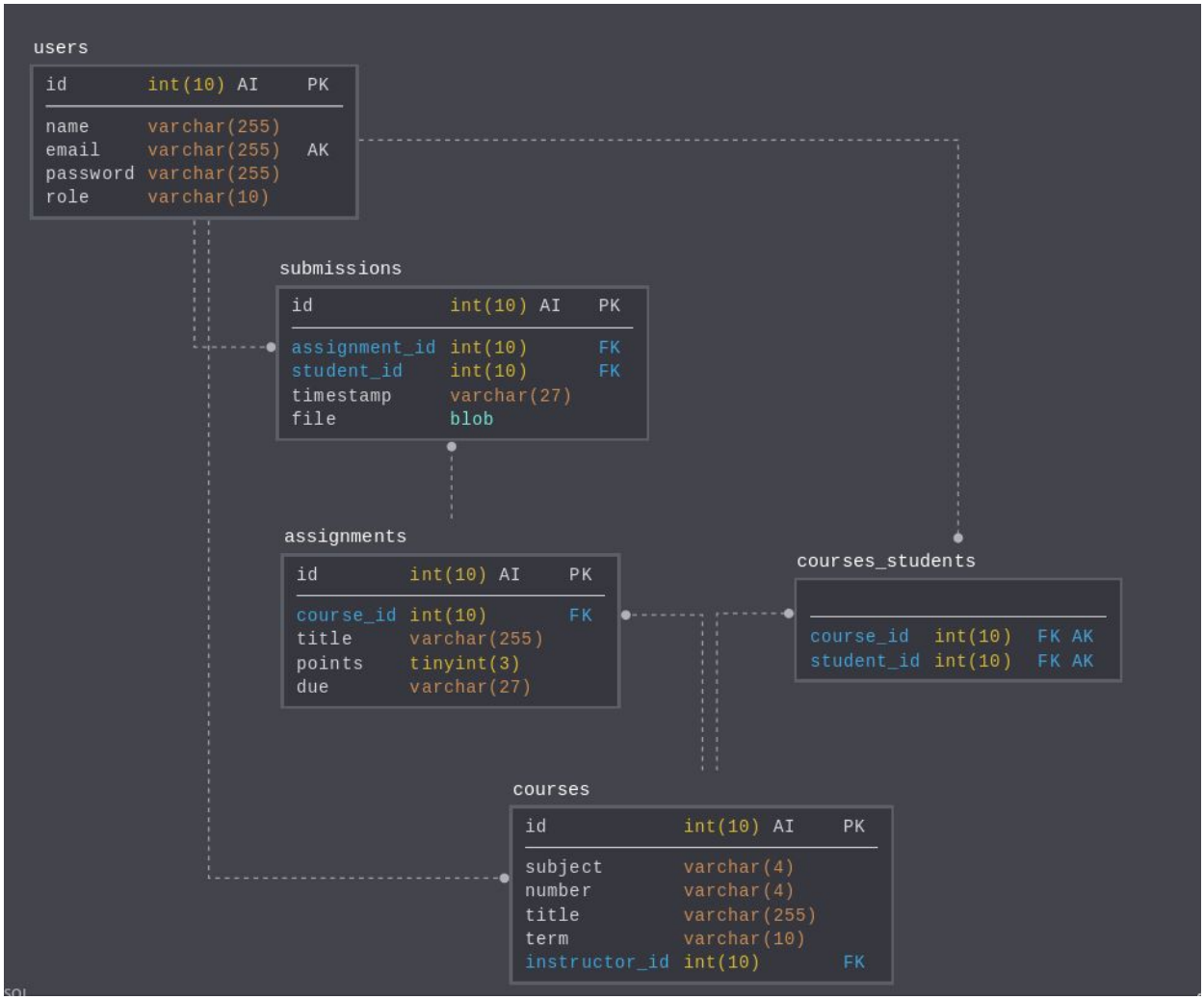
The front-end sends requests to the API endpoints that are exposed by the server. The backend server takes these requests, performs authentication on the requests, and sends corresponding queries to the SQL database. The SQL database server responds with info to the backend server. The backend server performs transformations on the data and sends it back to the frontend. Examples of these transformations include adding or removing info to responses and aggregating multiple SQL queries and responses.

The API server interacts with the MongoDB GridFS server whenever an assignment submission is submitted or requested. It pulls the binary file data from the DB and sends it to the client or vice versa.

The API server interacts with the Redis server to check whether a given IP has sent too many requests. It sends a combination of the IP and the timestamp of the first request as a key, and then subsequent requests by that IP are incremented to the value corresponding to the key in Redis. Once the value is over a certain amount, the rate limit has been reached and the server stops responding with info the user/frontend requested.

2. API Data Layout

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



There are 3 core tables in the SQL database: `courses`, `users`, and `assignments`. These core tables are joined by the `submissions` and `courses_students` tables, which represent relationships users-assignments and users-courses, respectively. `courses_students` is a pure joiner table, where `submissions` also includes information about the submission.

3. API Design Reflection

This describes why we chose the architecture and data layout we choose and reflect on what parts of our design worked well and what parts we would change, and how, if we were to implement the same project again.

- The architecture of the `/users` endpoints was overall good, pretty similar to the endpoints we made in earlier assignments. If we had to implement these endpoints again, we would try to aggregate all of the detailed user information for users/instructors/students into one SQL query rather than multiple, to increase efficiency.

- We found potential abuse issue in the OpenAPI documentation/requirement. In `POST /assignments/{id}/submissions`. The documentation requires `student_id` to be in the request body. I believed can be abused by users. A student could submit submission as another student by putting their IDs into the request body because we use `req.body.student_id` instead of the id of the currently logged in user (`req.authenticatedUserId`). To fix this, we should: (1) Remove `student_id` in the POST request for Submission; (2) Use `req.authenticatedUserId` to store into the database instead.
  Link to internal Security advisory discussion on GitHub (GitHub's new feature in beta)

- It took us quite some time to implement pagination and filtering by URL parameters. The paging code was similar to what we have learned in this class. The optional filtering parameters made the problem more challenging, but we solved it but tweaking the database queries such that the data was fetched before pagination (as opposed to paginating directly using the database query like in class).

- For the `POST /courses/{id}/students` endpoint, we decided to make the response a bit different from the initial OpenAPI documentation. Instead of just showing the success/failure status of the request, we implemented the response such that it would return a list of successfully enrolled student IDs and a list of successfully un-enrolled student IDs. The reason we did this was that the provided IDs in the requests might not belong to the students, and we wanted the app to ignore non-student IDs instead of complaining 400 errors about them.

- In addition user's ID, we included the user's role (either as an admin, an instructor, or a student) to the JWT authentication payload. As a result, We made the validation process of the role of the current user much quicker.

- We are glad that we decided to store binary data to a separate database because based on the context of this application (a course management portal as an "alternative to Canvas"), users don't always read/write from/to binary data all the time, if not to say way less than other relational queries such as CRUD the courses and assignments. Also, another rationale is that for binary data, it's better to use an object storage engine like MongoDB GridFS instead of MySQL's BLOB storage because the former performs a lot better with large binary files and is actually intentionally designed to do so just like Amazon S3 object storage, while MySQL is better suited for relational data [1]. We are aware of an option of using the Linux file system itself to manage files in a separate machine like Flickr did (or still does) [2] Doing that requires complicated network file system mounting configuration that we're not familiar enough to try. Another rationale is that storing file submissions in the main MySQL database will quickly fill up the size of the database.

- After implementing the Submission API with MongoDB GridFS as data storage for binary file *and* the submission data, we realize we could have done better to separate submission file binary data plus its file metadata (`filename`, `contentType`, etc.) to MongoDB and the corresponding submission data (`assignment_id`, `student_id`, `timestamp`, etc). to MySQL. Doing this will increase the complexity of operation in our API service but it would make more sense to store relational data to MySQL only and binary data/object to MongoDB only. The limitation of the current implementation is that whenever some tasks in the API service need something from the MySQL database and just something about the Submission data, not the binary file itself, it will query both MySQL and, necessarily, MongoDB.

- We didn't have enough time to implement Data Replication for both MySQL and MongoDB.

- We didn't have enough time to deploy multiple images to Google Cloud with Google Cloud Build and Travis CI.

- We wished we have crafted more adequate, concrete collaboration policy. One issue is that some changes on a feature branch are not related to the current feature. Instead of making a new branch and propose that unrelated changes to master, we made those unrelated changes to the branch we're currently working on. This issue later caused us some un-negligible amount of time to resolve.

References

- [1] https://docs.mongodb.com/manual/core/gridfs/#when-to-use-gridfs
- [2] http://highscalability.com/flickr-architecture