Zachary Stata

Spring 2023

Advisors:
Daniel Mossé, PhD
Nathan Ong

University of Pittsburgh
School of Computing and Information
Computer Science Master's Studies Project Final Report
Beyond Transcripts Back End Development

Introduction

Beyond Transcripts is a tool to be used in the monitoring and assessment of a student's grades as they change throughout a semester. To understand the use of this tool, one must first understand the idea behind it. Consider a hierarchy where courses are broken down into topics, where each topic may contain multiple concepts. Concepts in a class may be linked to each other in a logical hierarchy of their own. For example, consider a low level computer science course. An early topic that could be taught in that course would be data types. Within the data type topic, there may be concepts that students are taught, such as integers and strings. While the integer and string concepts may be independent of one another aside from sharing a topic, they both may be linked to a subsequent concept, such as arrays. Arrays may then feed into a concept about for-loops, which could then feed into a concept about recursion. In this manner, concepts link together to form Concept Progression Maps (CPM). A visual example of this small map can be seen below in Figure 1.

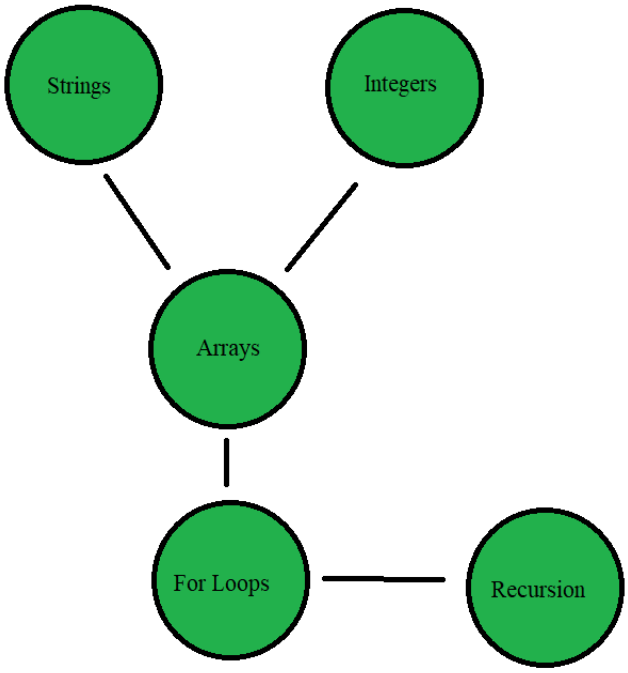

Figure 1. Example CPM

The main idea behind this tool is the dynamic creation and displaying of these CPMs for any given course. Student grades are to be calculated per concept, with a final goal of displaying the CPM with concepts colored to give an at-a-glance view of the student's performance in the class. For example, in Figure 1 we may assume that the student whose concept map is being displayed has scored well on all questions relating to these concepts since each concept is green. Had they scored lower on questions relating to a particular concept, it may be colored orange, yellow, or red. Instructors, advisors and the student themselves can use this view to better understand the student's performance on a concept-by-concept basis, allowing them to better focus their study time on concepts that may progress into further concepts, rather than those that are dead ends. In theory, this will enhance student performance within a course by allowing them to prioritize course concepts that will be built upon throughout the course.

Background

I joined this project at the beginning of January 2023. Our goal at the beginning of the semester was to develop a Minimum Viable Product (MVP) for the validation of the Beyond Transcripts idea. The first few weeks were spent finalizing the database design that would relate all of the entities needed for the storing and retrieving of information used in the creation of the CPMs. The finished Entity Relationship diagram can be seen below in Figure 2. Following the final edits of this design, my task for the remainder of the semester was to develop the backend of this project from scratch. This backend would have to expose API endpoints for a potential frontend to hit to both save and retrieve all necessary information in the creation and display of CPMs. The backend also is responsible for all database management associated with these tasks. While I have plenty of experience in Java development, this would be a new venture for me that

required the learning of new tools that I believe will be of great use to me in my future in software engineering. In this report, I will describe my time with the project, the documentation of the backend, and any knowledge/future work that I believe would be useful to anyone who may work on this project in the future.

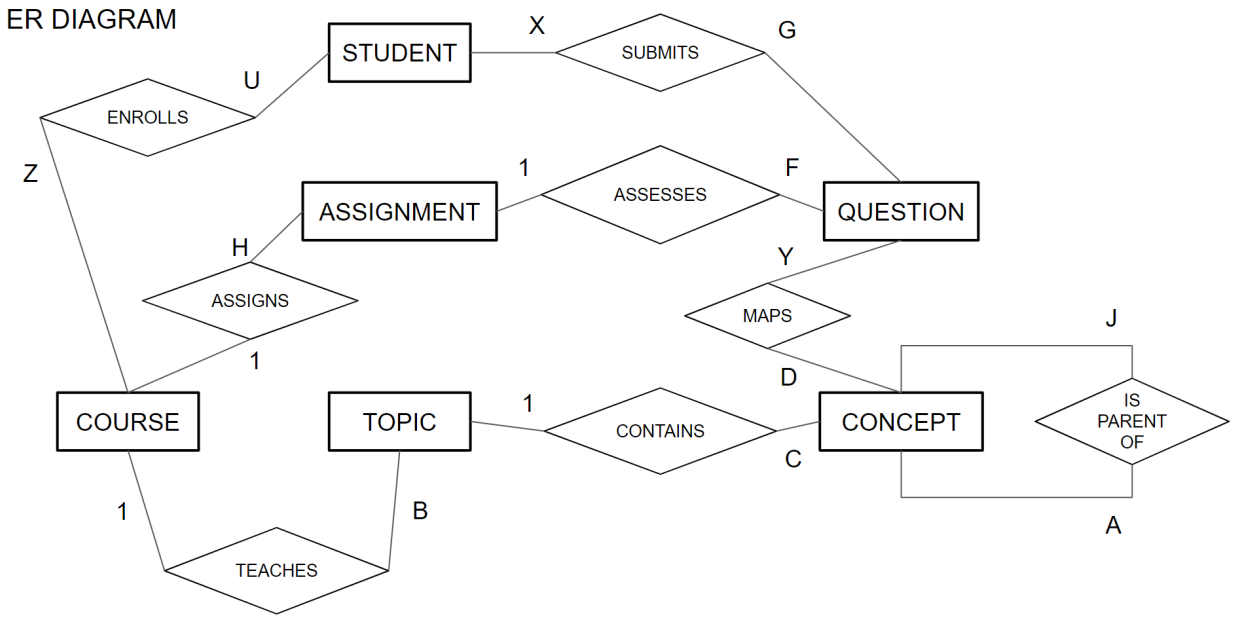

Figure 2. ER diagram used in saving CPM information.

Database Design

When I joined this team at the beginning of the semester, the focus was on finalizing the relational design for the database that I was to create. Plenty of this work was done before I joined the team, but nonetheless it is important to understand as it is integral information in the formation and function of the backend. In the above ER diagram contained in Figure 2, the rectangular boxes represent entities within the database while the diamonds represent the relationships between entities. For instance, a course entity has a one-to-many relationship with topic entities. This essentially translates to the fact that a singular course may contain many topics. Similarly, topic entities have a one to many relationship with concept entities, meaning

many concepts can be contained within a single topic. Perhaps a little more confusing to understand would be the concept entity's relationship with itself, which is many-to-many. This describes how within a Concept Progression Map, a concept may both have many parents and many children, representing logical prerequisites and following concepts.

These relationships are important because they allow us to reconstruct CPMs in the backend, along with the grade calculations that come with them, but that will be discussed further in the discussion of the API endpoints. For now, assume each individual entity and relationship has their own table in a database. Columns in the entity tables represent the entity's fields. For example, an instance in the database representing a single course entity would contain fields representing the course's name and description. A table containing the entity schemas and their respective fields can be seen below in Figure 3. Tables representing relationships such as Enrolls or Maps look somewhat different in the database. They simply contain the keys between the two entities that they connect. For example, an instance in the Enrolls table representing one of the classes I am currently in would be as simple as "zrs17" and "CS2750" for the course name and student ID fields.

STUDENT	student_id
QUESTION	qName, aName, courseName, qDescription, totalScore
ASSIGNMENT	aName, courseName, aWeight
CONCEPT	conName, conCourseName, conceptDescription, nodeVal
TOPIC	tName, courseName, tDescription
COURSE	cName, cDescription

Figure 3. Entity Schema. Underlined fields represent primary keys.

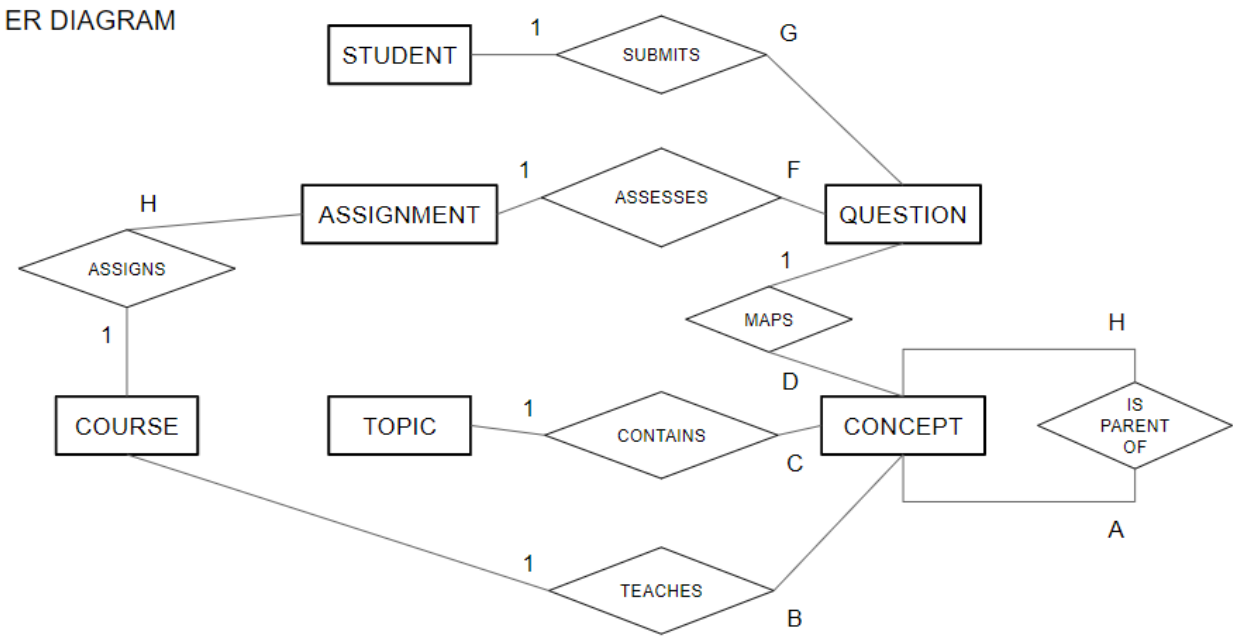

Figure 4. Early ER diagram.

With knowledge of how the Entity Relationship diagram works, we can now consider some of the design decisions that were made, along with the justifications behind them. An early version of the ER diagram can be seen above in Figure 4. It contains the necessary relationships for constructing CPMs from the entities, but without certain optimizations. The first major change we decided on as a group was the creation of a course-topic-concept hierarchy rather than the course-concept-topic entanglement seen in Figure 4. Not only does the change make the relationship between the three entities more intuitive, but it also helps from an optimization standpoint. To understand this, consider the situation in which the frontend wants to display a list of topics within a course. In the old setup, the only way of doing this was to find all of the concepts within the course and examine all of their topics, ignoring overlaps between concepts in the same topic. All in all, this means more table joins, and therefore less efficiency than would be the case when simply connecting the course and topic entities directly. While this may lead

one to question whether the same issue would appear with finding all the concepts in a course, simply including the course name inside the concept entity allowed me to bypass these joins entirely.

Another such instance of an optimization we made is the inclusion of the Enrolls relationship seen in the top left corner of Figure 2. In this case, we must consider how the front end functions. A block diagram depicting a rough layout to the application's functionality can be seen below in Figure 5. As seen on the far left side of the diagram, users, which in this case we assume to be instructors or advisors, enter the application from a Pitt Passport Login and are presented with courses and the students within them. This information, of course, is pulled from the backend, meaning that we must be able to relate students to courses. In the old ER diagram seen in Figure 4, this would have to be done by joining all of the Assigns, Assesses and Submits relation tables, rather than storing one simple Enrolls table between the course and student entities. This demonstrates a tradeoff between time and storage made in the diagram, but I believe it is a very important one given the high usage rate that the Enrolls table experiences due to the fact that the course-student relationship is always needed upon starting the application.

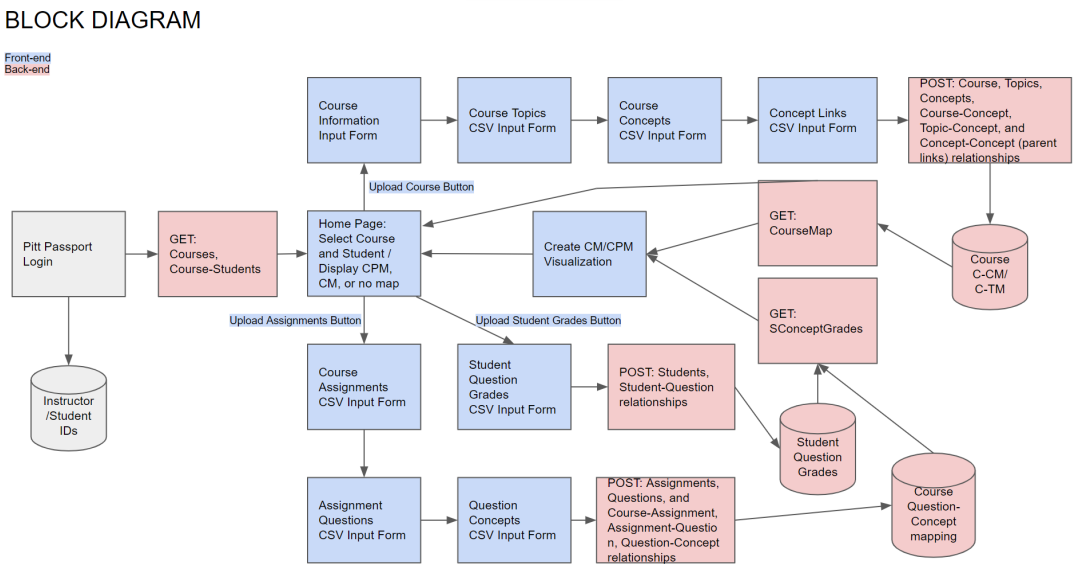

Figure 5. Block diagram depicting application workflow.

This entire stage of development was particularly new and impactful for me, as it required me not only to participate in the construction of a relational diagram given a problem space, but to consider the tradeoffs in the execution of its design. My previous academic experience with database design left me with the knowledge of how to construct and understand these sorts of diagrams, but not quite the rationale behind the decisions made. Therefore, before this project I likely would have been satisfied with the first iteration of the ER diagram. Given our goal of developing an MVP for this application, decisions such as reworking the diagram to include the Enrolls relation would have an actual impact on a product, meaning that such complexities cannot simply be waived off. In this way, I gained important knowledge from this stage that will stay with me should I encounter a similar problem in the future.

Backend Tools

Starting the backend development of this project also came with new challenges that I had not considered before. After all, I had never developed an API, and this would require the learning of new tools and techniques. The two main development tools that I found to be extremely useful in this venture were Spring Boot and Postman. Spring Boot is a framework for building Java applications quickly and easily. It is built on top of the Spring framework and offers a streamlined development experience for creating standalone, production-grade applications. Perhaps most useful is how it eliminates the need for developers to manually configure infrastructure components and provides a wide range of out-of-the-box features that can be easily customized to suit specific project requirements. It also includes an embedded web server, making it easy to build and deploy web applications, greatly simplifying my development process [1]. Overall, Spring Boot simplifies and accelerates Java development, making it a useful

choice for building robust and scalable applications, like what I wanted to do with this project. Setting it up was as simple as using the Spring Initializr tool [2]. For the sake of this project, I chose to use Spring 3.0.2 combined with Maven for dependency management and Java 17. Aside from those customization options, I simply filled in the project name metadata and let Spring Boot produce the startup code for me. For the time being this code had no function, but Spring Boot provided me with the tools to create and run a RESTful API with ease from here.

Postman assisted me on the other end of development, testing. It is a popular API development tool that simplifies the process of making API requests [3]. It provides a user-friendly interface for developers to create, test, and document APIs. With Postman, developers can easily send HTTP requests to APIs, view and analyze responses, and debug any issues that may arise. It also allows for the creation of automated tests, making it easier to ensure that APIs are working as expected. Additionally, Postman provides collaborative features that enable teams to work together on API development and testing. All of these features were particularly useful to me, as I was working on setting up the API calls at the same time that my development partner, Jacob Hoffman, was working on the frontend. In the finished MVP, his frontend would obviously call my APIs. In the meantime though, Postman allowed me to test these endpoints as if the frontend had called them, allowing for Jacob and I to work at the same time and seamlessly integrate once the work was done.

Backend Design

The backend is developed using the Model-View-Controller-Service (MVCS) design framework. This is a standardized way of designing a code base that I had never worked with in my academic career, but I have found it to be one of the most useful things that I learned this semester. In this framework, the model represents the data layer of the application, which

defines the entities and their relationships. In the context of this project, these would be the classes I designed for database entities such as concepts and the enrolls relationship. Each model is mapped to a table in our SQL database as well. The view represents the presentation layer, which handles user interface and input/output. This would refer mainly to our frontend and the way it sends input to our backend through API endpoints. The controller handles the logic of the application, handling user input and translating it into actions on the model and view. In this project, the controllers were classes containing API endpoints. Upon receiving input from the frontend, or Postman in my testing setup, the controller simply passes the input onto the service layer. The service layer provides a layer of abstraction between the controller and the model, encapsulating the business logic of the application. It acts as a mediator between these two layers, handling complex business logic and enabling better separation of concerns. Overall, the MVCS framework is a software design pattern that promotes modularity and code reusability, making it easier to develop and maintain complex applications such as this one. It allowed me to organize my code in such a way that debugging, as well as incrementally building, became much easier than I have ever experienced, and I would recommend it for anyone looking to build a backend to an application such as this. A small, visual abstraction of this framework can be seen below in Figure 6.

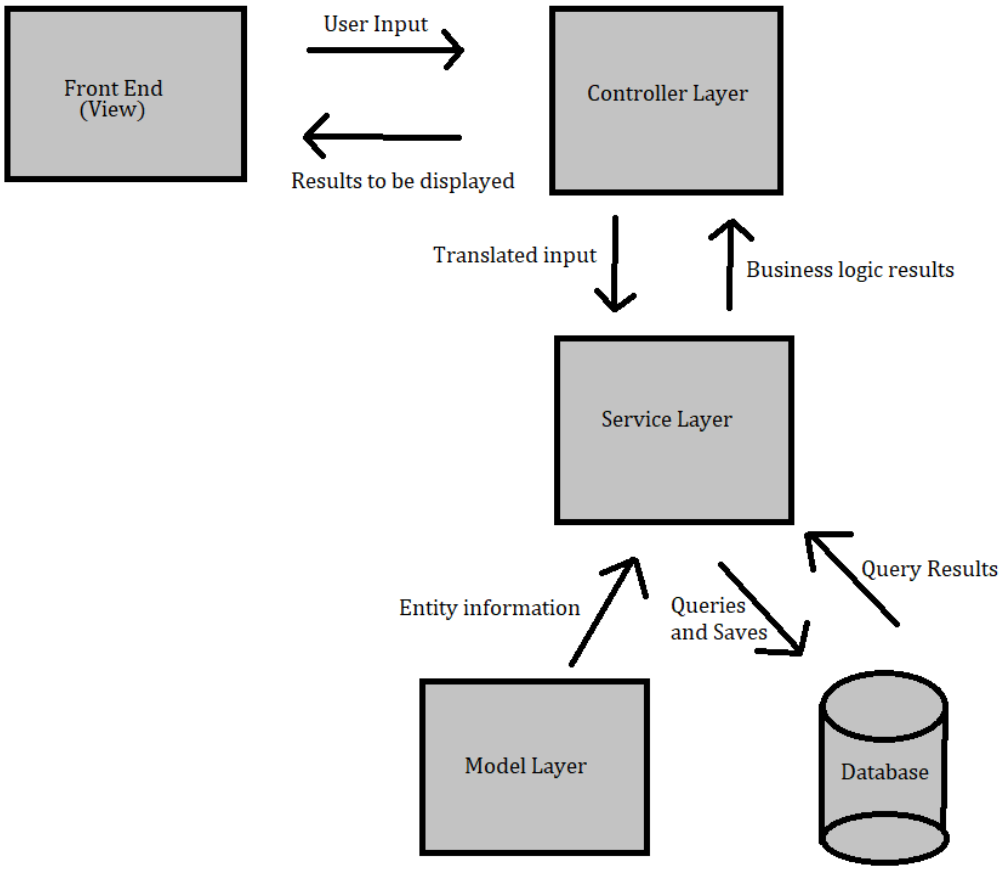

Figure 6. Visual abstraction of MVCS framework.

The benefits of this framework are best understood when working through an example. To pick one within the scope of Beyond Transcripts, let's assume a user would like to see the CPM for a specific course. This input is JSONified and sent to the backend through an API endpoint that is set up by the controller. The controller takes in this input and translates the JSON data to models representing the specified course, before passing this information onto the service layer. The service layer class has a specific function mapped to every endpoint supported by the controller. In our example, the service function will query the database for all topics, concepts, and links between concepts that exist for the given course. Using the model layer, it builds an object representing the desired CPM and passes this back to the controller layer. The

controller layer translates this object to JSON and sends it back to the frontend, where it will then be displayed to the user.

This framework functions similarly for any given API endpoint provided by the backend, all of which will be briefly described in a separate section below. Spring Boot helped tremendously in conjunction with the MVCS design framework, as it provided tools for exposing endpoints quickly and easily using function decorators. It also provides automatic translation of JSON to and from the models that I provided, making integrating my work with Postman and the actual frontend as easy as possible. I could not recommend this trio of development tools more heavily. Learning to use them was a fairly lengthy process early on in the development cycle, but after working through a tutorial, further work under this setup was an enjoyable process [4].

The final part about the backend development process I would like to point out is the database setup. For all development purposes, I have been using an H2 database that is built into Spring Boot. This means that the current database resets every time the backend is restarted. Earlier in the development process, Jacob and I worked on containerizing the backend using Docker. We found this to be a success at the time, allowing for it to be run separately from a database that would be containerized itself. It is currently set up to assume that the separate database would be a Postgres Docker image running on port 5432, should the switch from the H2 database be made via the application.properties file. This is entirely customizable via the docker-compose.yml file in the backend repository, and should provide a smooth transition into a production environment.

API Endpoints

At the end of the day, the main job of the backend is to provide a RESTful API setup for the frontend to access. A RESTful API is an architectural style for designing web services that

provide methods for creating, reading, updating, and deleting (CRUD) resources over HTTP. REST stands for Representational State Transfer, which means that the API transfers the state of a resource, like a CPM, in a representational format, such as how our CPM objects are sent back to the frontend using JSON. These types of APIs use a set of HTTP methods to interact with resources, though the ones utilized in our MVP are simply GET and POST. These methods indicate the specific purpose of the HTTP request: GET is used to retrieve a resource, and POST is used to create a new resource [5]. The design of the API endpoints hosted by the backend I created follows this framework of interacting with users. All such endpoints, followed by a brief description of their functionality can be found below. More in depth explanations can be found in the JavaDocs documentation found in the /docs folder of the backend, as well as in the thorough commenting in the backend code itself.

Note: all API endpoints start with a URL of "http//<server>:<port>/api/<specific_endpoint>".

- **/assignment/add -** POST endpoint for adding an array of assignments and their respective questions to the database for a given course.
- **/map/add/** - POST endpoint for saving all of the relevant components of a CPM (course, concept, topics, concept-concept links) into the database.
- **/map/get/** - GET endpoint for requesting a CPM from the database, given a specific course's name via URL request parameters.
- **/courses/get/** - GET endpoint for retrieving an array of all of the courses in the database.
- **/students/get/** - GET endpoint for retrieving an array of students, given a course name via URL parameters.

- **/grades/add/** - POST endpoint for posting an array of student grades for specific questions. These grades are saved in the database to be used later in the calculation of concept grades.
- **/conceptgrades/student/weighted/get/** - GET endpoint for retrieving a mapping of grades (weighted by assignment weight and question-concept distribution) to concepts for a given student and course.
- **/conceptgrades/student/unweighted/get/** - GET endpoint for retrieving a mapping of grades (where all questions in a concept are assumed to have equal weight) to concepts for a given student and course.
- **/conceptgrades/average/weighted/get/** - GET endpoint for retrieving a mapping of class-average grades (weighted by assignment weight and question-concept distribution) to concepts for a given course.
- **/conceptgrades/average/unweighted/get/** - GET endpoint for retrieving a mapping of class-average grades (where all questions in a concept are assumed to have equal weight) to concepts for a given course.

Future Work

As mentioned above, this semester's work has been to create a Minimum Viable Product, meaning that there is obviously plenty of room for growth in the future. In this section I will list some potential goals that I believe would be pertinent for future iterations to expand on.

- Integration with a sign on service such as the single-sign on (SSO) service used by Pitt. This, in theory, will provide the Beyond Transcripts frontend with relevant information, such as user IDs, to be used in querying the backend for CPMs and grades.

- Under the current application design, the backend is queried to return all courses that exist in the database. This is done in order to allow a user, such as an advisor, to pick a course and trigger a retrieval of the course's CPM and student list. This is fine in the MVP, but should this service be used in production, the database of courses will experience exponential growth. This would mean querying, sending, and using tremendously large lists of classes over HTTP, which is bound to induce problems. A more specific way of presenting a course list to users should be worked on. This would be a trivial query change in the backend, but currently there is no such information to specify on, such as professor or course semester. Perhaps this information could somehow be acquired on the frontend via the sign-in service.
- Persistent database integration needs to be completed and tested before this application can move into production. As mentioned in prior sections, we have set up the backend to function as a Docker-containerized microservice that can communicate with a similarly containerized Postgres database. While all of this was set up, development was conducted using Spring's internal H2 database, meaning some setup and testing may still be required with a persistent one.

Personal Takeaway

Getting a chance to work within this environment has truly been one of the most impactful experiences of my entire degree. Never before had I been introduced to tools like Spring Boot and Postman, nor had I ever used an MVCS design framework. I now know how extremely useful these are, and given what I have learned from this project, I will make sure to use them in my future as a software engineer. My time on this project has even given me the confidence to start thinking of fun applications of my own that I may want to develop some day.

I have often been disappointed in my Master's degree with how many classes equip me with theory more so than they do with development experience. With this project, I have finally found that experience that I had been craving.

Beyond implementation experience, this project has also equipped me with knowledge of what it is like to work within a team more so than my classes have. Early on in our weekly meetings, I was unsure of when to speak up, often thinking of things to say but without the confidence to voice them. As time went on, I learned to voice my opinions and talk out ideas as a group. As I start my career in the coming weeks, this learning experience may be the most valuable of all. I truly am grateful for my time with this project. It has been one of the most interesting and impactful experiences of my college career, and I certainly will not forget it. I want to thank Dr. Mossé and Nathan Ong for advising me throughout this project, and Jacob Hoffman for being a tremendously helpful partner along the way.

References

[1] Spring Boot | Why Spring, https://spring.io/why-spring

[2] Spring Initializr https://start.spring.io/

[3] Postman https://www.postman.com/

[4] JavaBrains Spring Boot Quick Start https://www.youtube.com/watch?v=msXL2oDexqw&list=PLqq-6Pq4lTTbx8p2oCgcAQGQyqN8XeA1x&ab_channel=JavaBrains

[5] RESTful API Tutorial https://restfulapi.net/