Cs 6310 : Software Architecture & Design

Project 3: Teaming Up & Reflections on Project 2 towards Project 4

Simon Hunt	shunt31	shunt31@gatech.edu
Ryan McMahon	mcmahon6	mcmahon6@gatech.edu
John-Peter Reiland	jreiland6	john-peter.reiland@gtri.gatech.edu
Jesse Stills	jstills3	jstills3@gatech.edu

October 25, 2015

1. Overview

This report merges together four individual system designs from the members of Team 33.

2. Requirements Analysis

Functional Requirements — Student Role	
Func1	Log into the system (from a web browser) with their user ID and password.
Func2	Enter a (single) prioritized list of courses they wish to consider taking in the next semester
Func3	Enter the number of courses they wish to take.
Func4	Submit a query to the recommendation system, based on their current course selection.
Func5	View the results of their query, which will include the following: • For each selected class: o Class size vs. current demand. o Whether the student's seniority places him/her within the class capacity. • A recommendation for which courses to take.
Func6	View a history of previous recommendation queries and re-list the results.

Functional Requirements — Admin Role	
Func7	Log into the system (from a web browser) with their user ID and password.
Func8	View and/or modify the class enrollment limits on individual classes.
Func9	View and/or modify the TA pool.
Func10	View and/or modify the available courses for the semester.
Func11	View and/or modify the Professor—Course assignments.
Func12	View any student's course preferences, past and present.
Func13	View a history of preferences, aggregate.
Func14	View a history of computer generated recommendations.
Func15	Option to deploy constraints live, or in shadow mode.

Functional Requirements — System	
Func16	The core engine should consider the following when generating a solution: • List of available courses. • List of professors. • List of courses each professors can teach. • Mandatory course offerings *(there may be none)*. • Prioritized list of desired courses from each student. • Student status by seniority. • Current TA pool *(there may be none)*. • The TA course assignments *(there may be none)*. • The professor course assignments *(there may be none)*. • Course size limits, if any.
Func17	The core engine should be able to output the following: • List of courses offered. • List of student assigned to each offered course. • List of professors for each offered course. • List of TA's assigned to each offered course.
Func18	All student, course, administrative constraints, and recommendation data will be persisted.

Non-Functional Requirements	
Non-Func1	*Performance* – The users should get a response in a timely manner (absolutely < 1 minute).
Non-Func2	*Availability* – The system should be available when required without loss of service.
Non-Func3	*Security* – https will be used for the connection between the browser and the server. The use of SSL certificates should also be considered.
Non-Func4	*Usability* – The system should be easy and intuitive to use.
Non-Func5	*Scalability* – As the school grows the system will need to support more students.

Non-Functional Requirements	
Non-Func6	*Extensibility* – Schools can be subject to new requirements and regulations that the system may need to account for. The layered architecture and use of appropriate interfaces should allow the system to be extended for further functionality at minimal expenditure of effort.
Non-Func7	*Maintainability* – The system requires high availability, and in the event of a failure, will need to be repaired in a swift and timely manner. The system (back-end) will be written in Java, using object-oriented techniques, and good coding practices; the UI (front-end) will be written in Javascript.

User Interfaces

- The user interface will comprise a Web UI Single-Page Application (SPA) written in JavaScript.
- User interface should be accessible from any modern browser.
- User interface should provide a way to log into and out of the system.

User Characteristics

- The intended users of the application are Georgia Tech students and administrators.
- The level of technical expertise is unknown. It is assumed there are all ranges of technical expertise.
- Users will likely have a wide range of differing screen sizes.
- The users will likely access the system using a variety of web browsers.

Assumptions

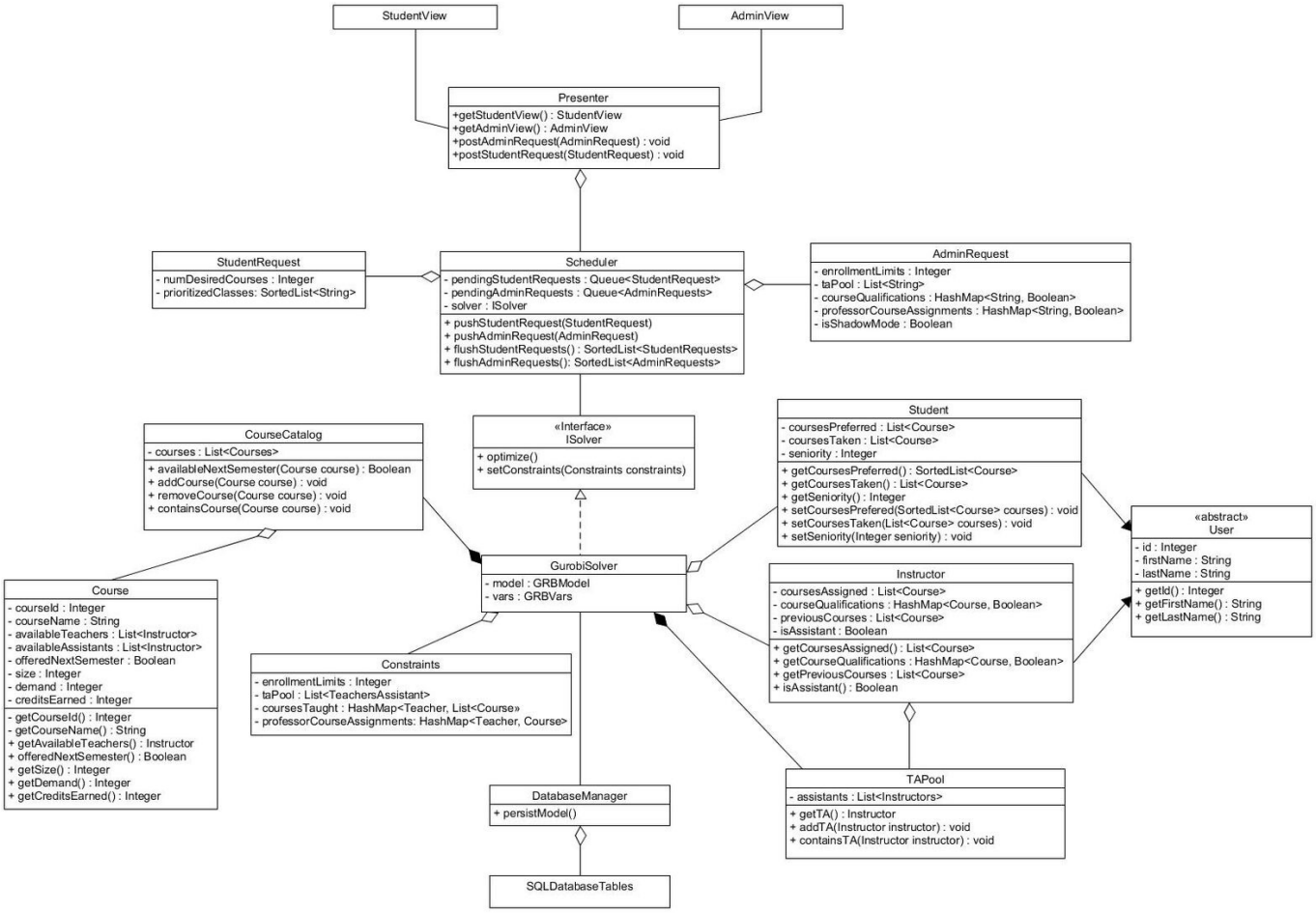
- The following information will be pre-loaded into the DB:
  - All course information.
  - Courses offered for the next semester.
  - Professors, their competencies, and their availability for the next semester.
  - The TA pool (including TA competencies and availability).
- Students and administrators are provided with a username and password to access the system.
- Another system exists for actually registering students for classes, and this system shares the same database as this student course assignment application.
- All students are able to access the system simultaneously.

Dependencies

- Dependency on *Gurobi* optimizer library.
- Dependency on external authentication server (for verifying user credentials).

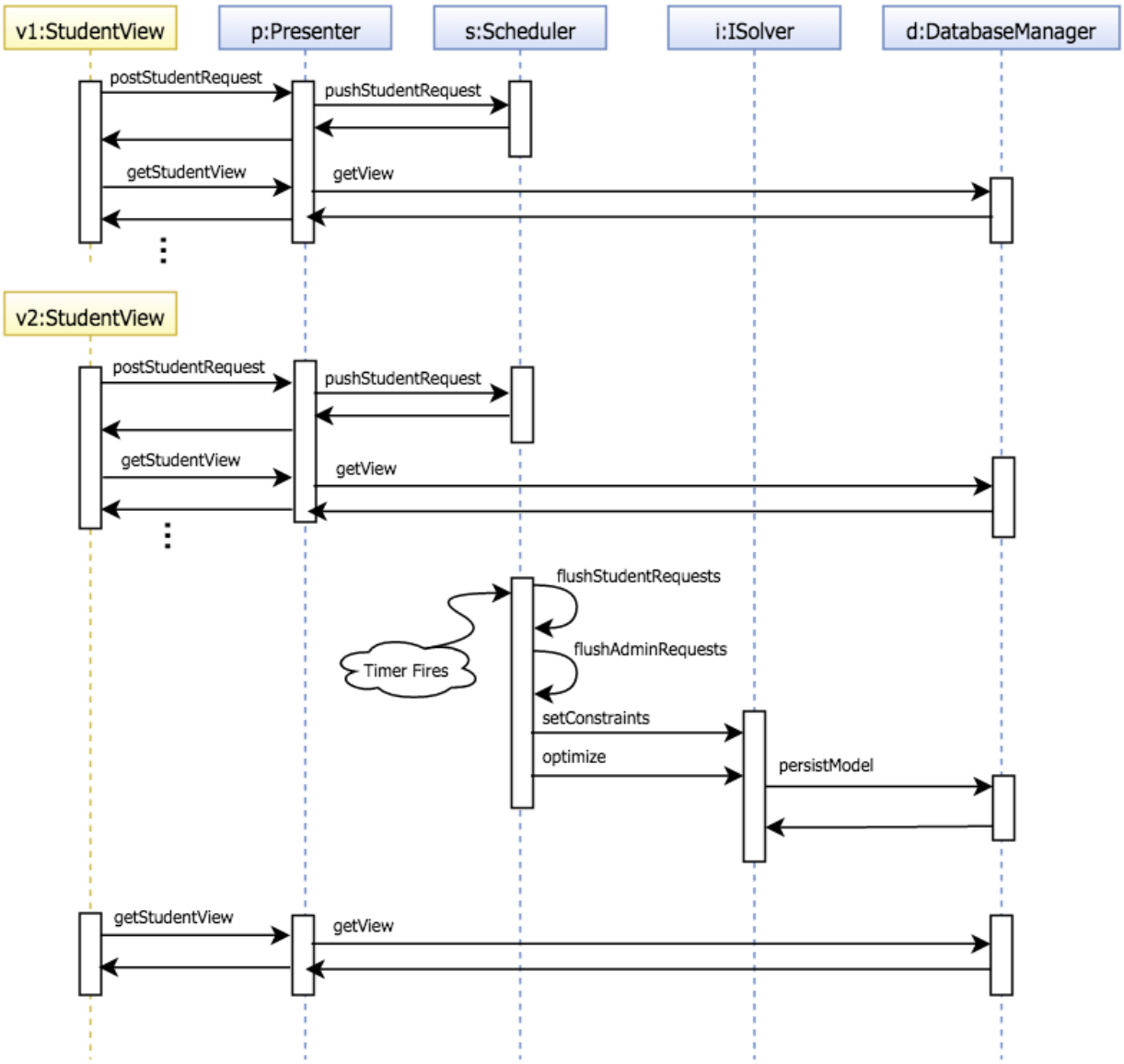
Regarding the functional and non-functional requirements, our team members came up with a very similar set of requirements. Some of the members focused more heavily on the user's interaction with the system when listing requirements, while others spent more time highlighting the requirements for the internal state of the system. Team members had a slightly different idea on what should was listed in regards to user characteristics, describing details of targeted users versus describing how the users will be using the system.

3. UML class-model design diagram



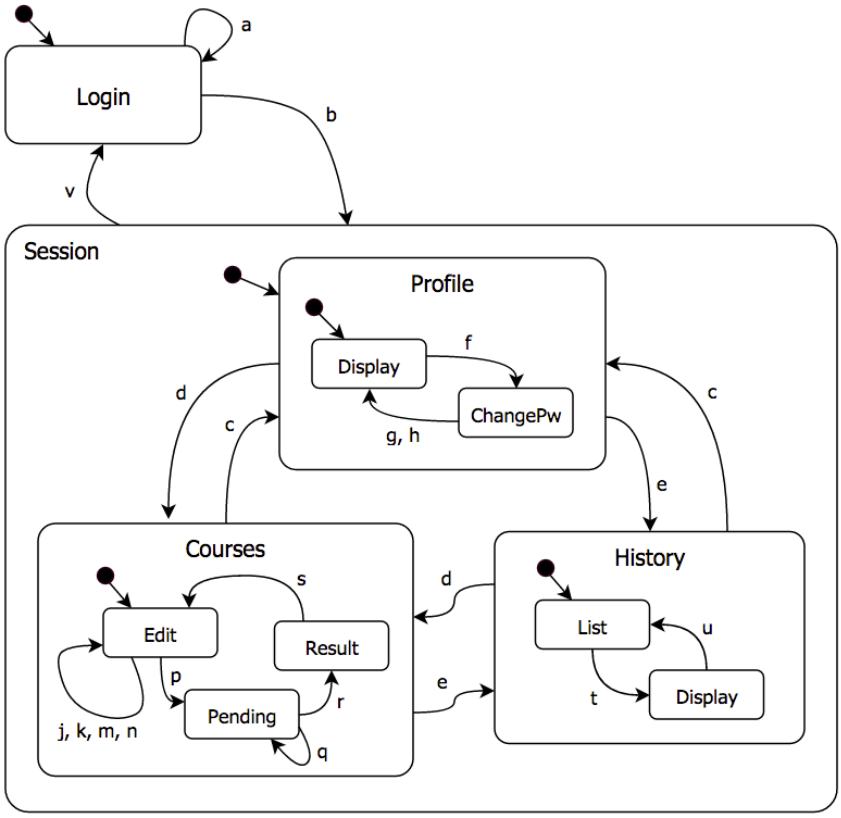
Two views are available to interact with the server, a *StudentView* and an *AdminView*. These two views interact with a *Presenter* class, which is responsible for handling requests from clients, and returning results from the solver. A *Scheduler* object is used to batch requests before being sent on to the solver. In our application, an *ISolver* interface allows for different solvers to be substituted, but our implementation will use a *GurobiSolver*. The *GurobiSolver* will work with classes representing course related objects, such as *Student*, *CourseCatalog*, and *Instructor*. These objects will represent the current state of the system in terms of inputs and persisted preferences. The solver persists the data using some API to the SQL database, represented as a *DatabaseManager*. The persisted layer is simplified here and represented as a *SQLDatabaseTables* class.

4. Interaction Diagram



The *student view* makes a REST call to the server, posting the request to the *presenter*. The presenter passes the request on to the *scheduler*. Note that multiple student views can submit requests. When an internal timer fires, the *scheduler* batches up the requests, translates them into constraints for the *solver*, then tells the *solver* to optimize. Once a solution is found, the *solver* persists the results via the *database manager*. Note that *student views* can request the data to be displayed at any time, by polling the *database manager*.

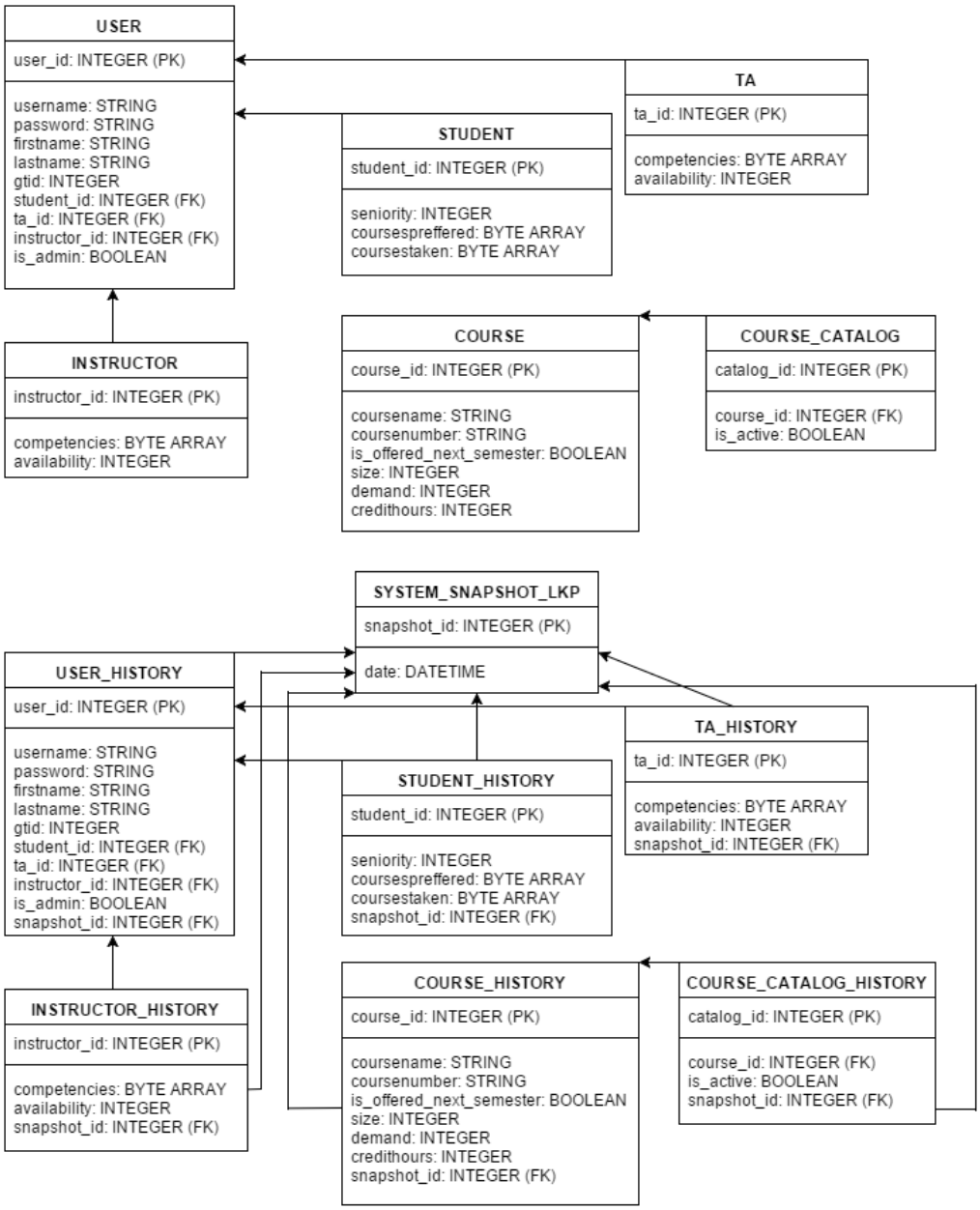
5. Statechart Diagram



Event	Description	Event	Description
a	Failed login	k	Add course to priority list
b	Successful login	m	Remove course from priority list
c	*Profile* tab selected	n	Drag course (change priority) in list
d	*Courses* tab selected	p	*Submit* button pressed
e	*History* tab selected	q	Poll server — no data
f	*Change Password* button pressed	r	Poll server — data available
g	Password changed	s	*New Query* button pressed
h	Change password canceled	t	*View Query* button pressed
j	Change number of courses	u	*Back to List* button pressed
		v	*Logout* button pressed

Statechart Event List

6. Logical Data Model

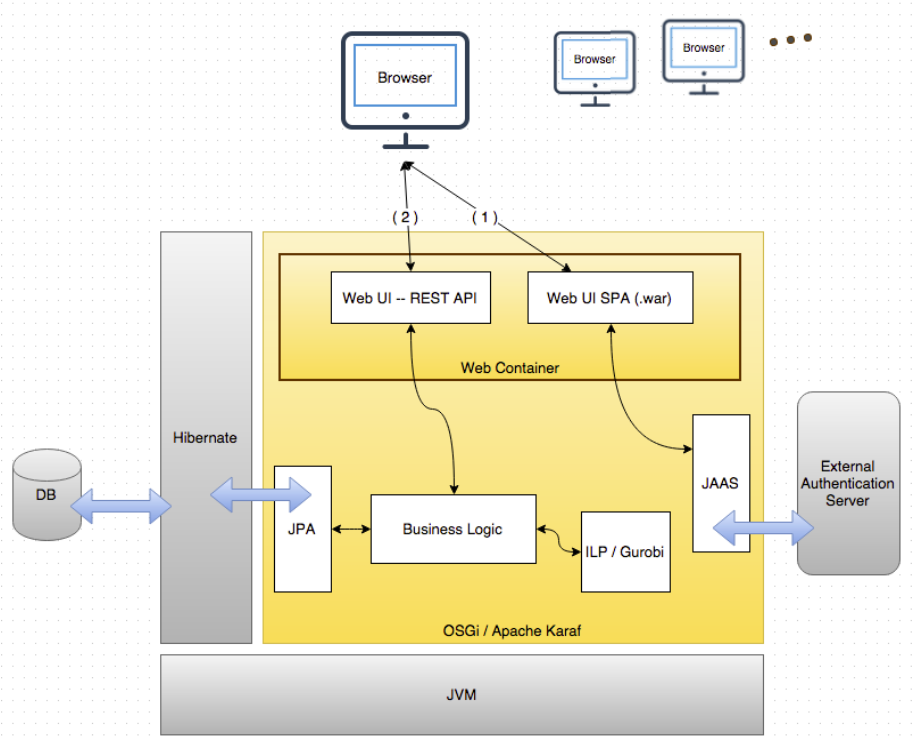


Some group members chose to represent data with tables for each of students, instructors, admins, and courses, plus tables to store historical information for each. Other members approached the logical data model with similar concepts but also opting to model additional application constructs and composite or derived structures as their own entities. In both cases, modeling historical data is a costly mechanism, but is required for the system to function as intended. The advantage to modeling out additional tables for the data model is faster and more

direct retrieval from the data store. The disadvantage with this approach is that storage requirements will be larger and time to update each of the additional entities can add up. On the other hand, taking the simpler approach of modeling only minimal structures plus their histories and depending on the data access layer to compose more complex structures for retrieval and use in the system can vastly simplify the logical data model. Disadvantages include a more extensive DAO and, in some cases, slower data retrieval. The benefits to the simplified model include speed of storage and overall simplicity, which can help developers interpret, manipulate, and reason about the data better.

7. Design Rationale

Architectural Style / Design Pattern



We decided to use a layered, object oriented, client-server architecture. A layered architecture allows for a separation of presentation, computation and persistence responsibilities. More specifically, the application provides a web service that serves up the UI via http.

A Java Authentication & Authorization Service (JAAS) will communicate with an external authentication system to validate user credentials. The client issues REST calls to the presentation layer of the server. The presentation layer transforms the data as needed to interact with the business logic layer. The business logic layer is responsible for all the computations and houses the core computational engine (Gurobi). The business logic layer interacts with the persistence layer to store the data in a database, and also sends the results back to the presentation layer where they are formatted and sent to the client.

All members of our team generally designed the same type of architecture. As mentioned above, a layered architecture allows for a separation of responsibilities, which is advantageous for simultaneous development due to the limited interaction between layers. Another advantage is that each layer can be distributed across different hardware to provide for greater scalability. In addition, each layer can be updated and maintained independently. A disadvantage to using RESTful calls to interact with the server, is that changes that have occurred in the system after a client has already received their results, will not be reflected unless the user refreshes the page. One team member proposed a possible alternative that utilizes an implicit invocation style. Each time a change in the results occur on the server, they are pushed to the clients via broadcasted events, providing a more responsive user experience. If time allows, such a solution may be added.

Persistence Mechanism

Our team all agreed that a relational database system would be optimal for this project. The developers' familiarity with the technology and ample documentation of best practices available will contribute to a smoother development timeline. A flat file system would be cumbersome and demand too much development overhead. The flexibility of a NoSQL database system is not necessary for this project. The disadvantages of a fixed schema leading to a less flexible data model do not outweigh the strengths of a relational database. Ultimately we all agreed that a relational database would be the most performant solution, offering the speed and structure necessary to store and retrieve data on users, courses, and the like for this application.

Algorithms / Data Structure

Each of the group's respective architectures involved storing data in intermediate objects to be used by the integer linear programming solver. These objects will aggregate information about students, courses, instructors, and other relevant objects in the application domain. These objects can be worked with independently, or as lists or hash maps depending on the objective at hand. These objects can be easily added or removed from the system depending on the inputs from users.

The majority of the computationally intensive work will be handled by the solver, by providing constraints and an objective function to calculate. Additionally, the design will include a batching mechanism which queues incoming requests and feeds them to the solver. Student inputs in a given batch will be weighted by seniority, measured by number of credits taken. If seniority is equal, GPA will be used as a tie breaker. If GPA is equal, then the user who first entered their input into the system will be given priority.

This choice of data structure and algorithms was to support the design of the business layer. Using objects and collections to represent classes, instructors, courses, and other domain objects will allow for a more intuitive and maintainable architecture. Batching input into the solver will allow for performance optimization and a satisfactory user experience.

8. Reflection

Collectively, I believe our team was successful in identifying the requirements of the application. I believe our overall structure is conveyed well in our class model, sequence and deployment diagrams. This will hopefully keep the number of ambiguities in our design low while we implement and develop our design. I think our architectural style is well suited for the project and the non-functional requirements that we identified. I think the technologies we chose play to our team's strength and are commonly used in modern development. I believe our user interface is straight forward and should be intuitive for the users of the system. Overall I think we have a solid design and we look forward to implementing it.