TABLE OF CONTENTS

Discussion of Changes

Permissions were correctly reflected in the design decisions. The home page will show the forums, but when navigating to a specific forum, the user is redirected to a login page if they're not already logged in. Permissions were reflected in other areas as well, such as the users and courses pages. On the users page, only admin users can see the "add new student" and "edit/delete" options. Similarly on the courses page, answers page, and questions page, admin only can see the modify options. With the exception being that any user can modify the question, answer, or forum that they personally posted. Non-admin users are also limited to what they can modify on their account settings. Admin users can change their name, email, and password, but students, for example, can only modify their password.

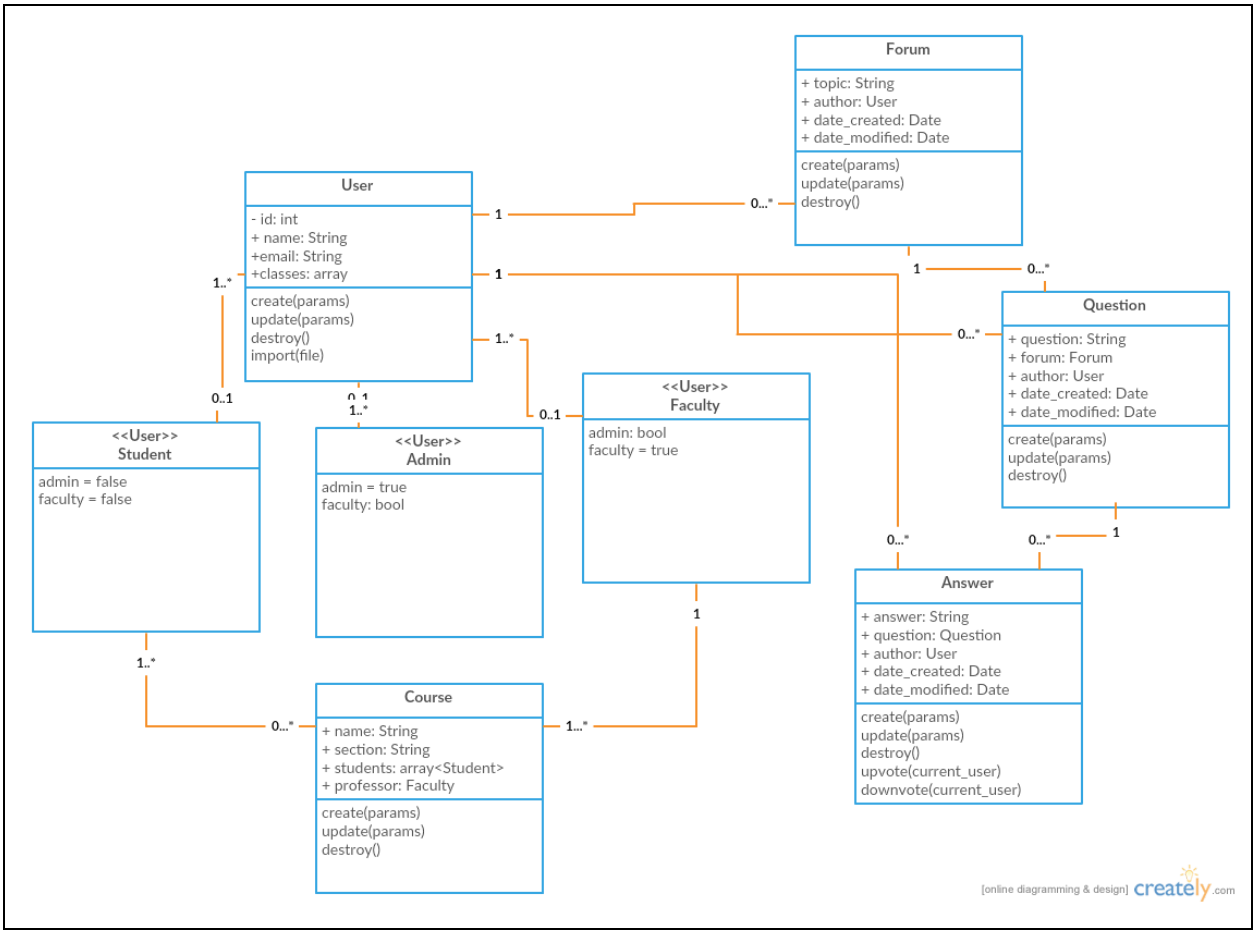
Voting on answers was also added. Users can upvote/downvote an answer, but only one or the other and only once. Therefore a user can't go through and upvote an answer multiple times. The number of upvotes and downvotes is shown on the buttons on the UI. This required an Ajax implementation so that the entire page wouldn't re-render, and instead just the buttons themselves re-rendered with the updated vote tally.

Adding questions and answers is also much easier now. Previously, the user had to manually input a question or forum id to link the fields. Now that is all taken care of in the backend. So a user can click a question, and immediately add an answer directly to that question, or click a forum and directly add a question to that forum.

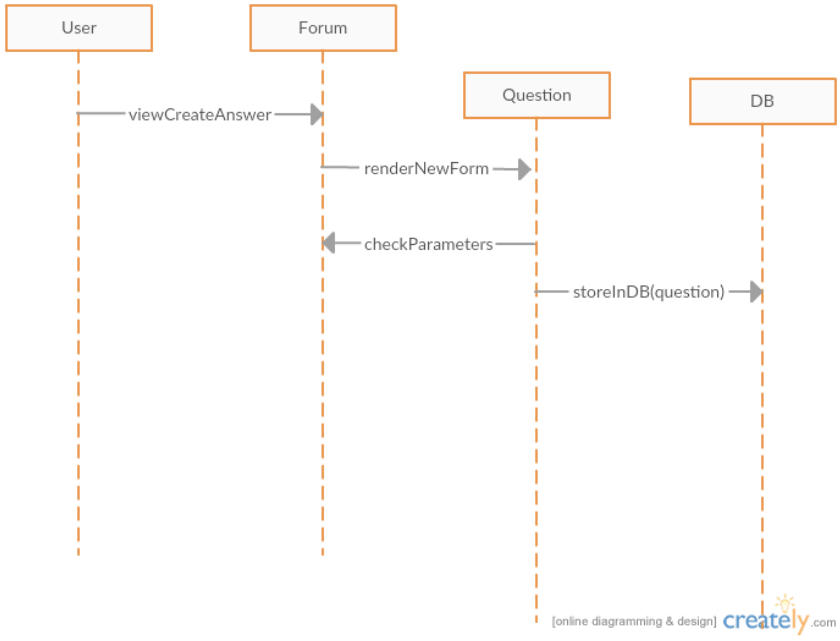
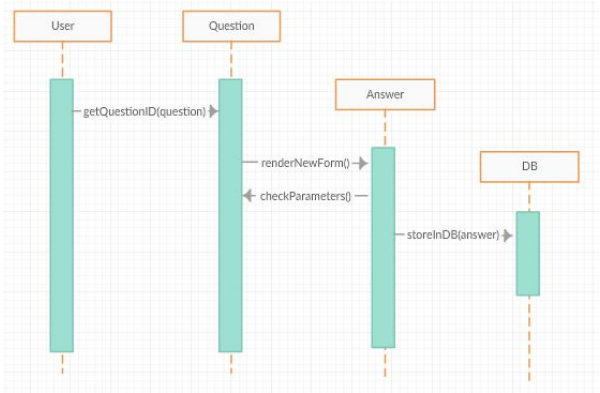
Alerts were also added. Therefore, when a user successfully signs in, a confirmation alert notifies them that they are signed in. Similarly, there are confirmation alerts for successful creation, update, and deletion of questions, answers, and forums. In addition, when a user tries to delete something (such as a user or course), a dialog box pops up, asking if they're sure on deleting that item. This helps to prevent admin users from accidentally deleting items. All fields across the app require data to be submitted so if a required field is left blank, the corresponding alert is displayed to the user.

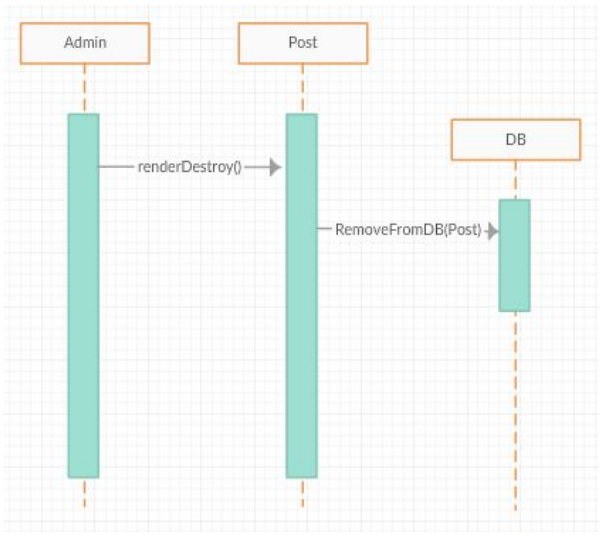
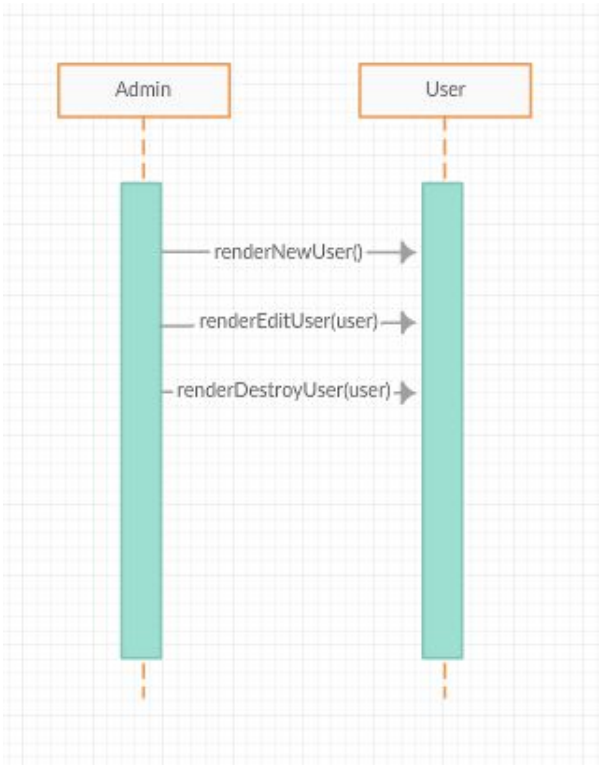
Finally, the UI was also modified. The elements are more unified with a NKU theme. A NKUNet logo was also added. It's also easier to read than the previous UI.

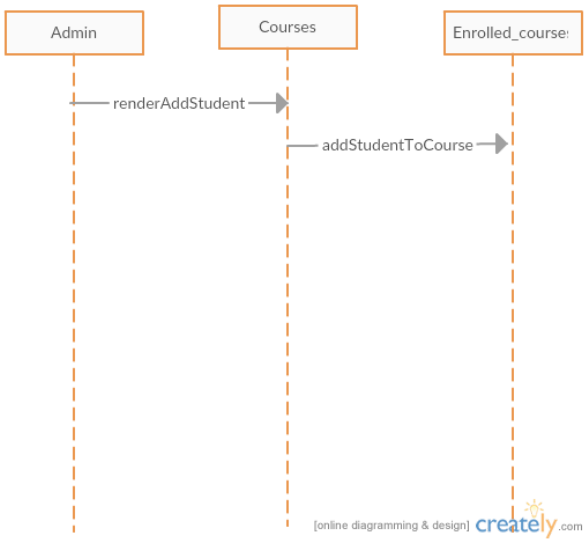
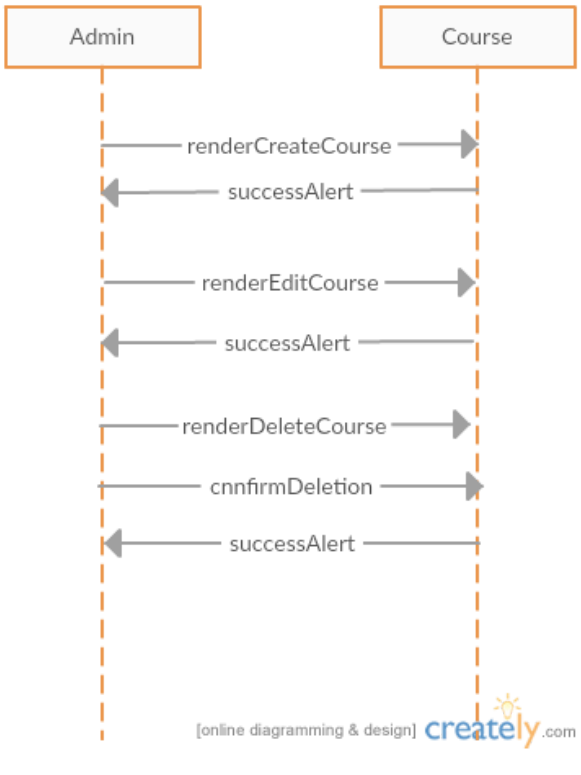
Design Class Diagram

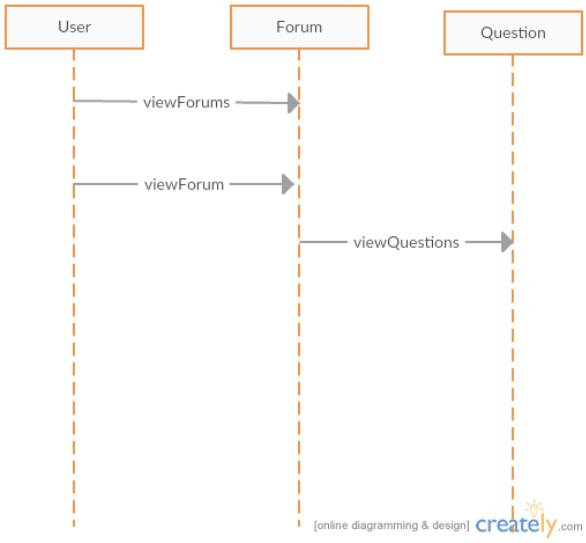
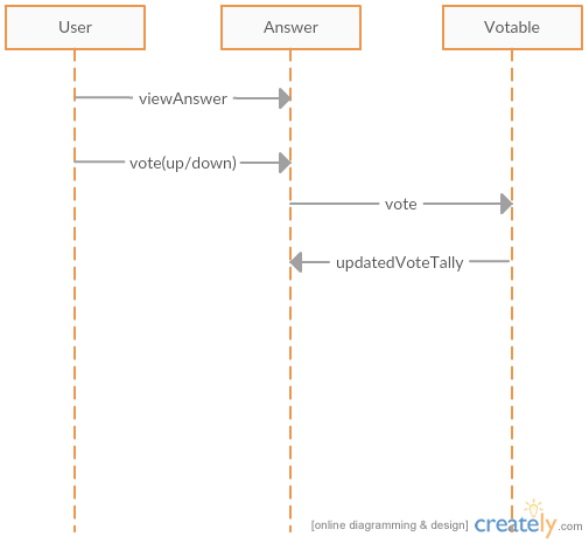


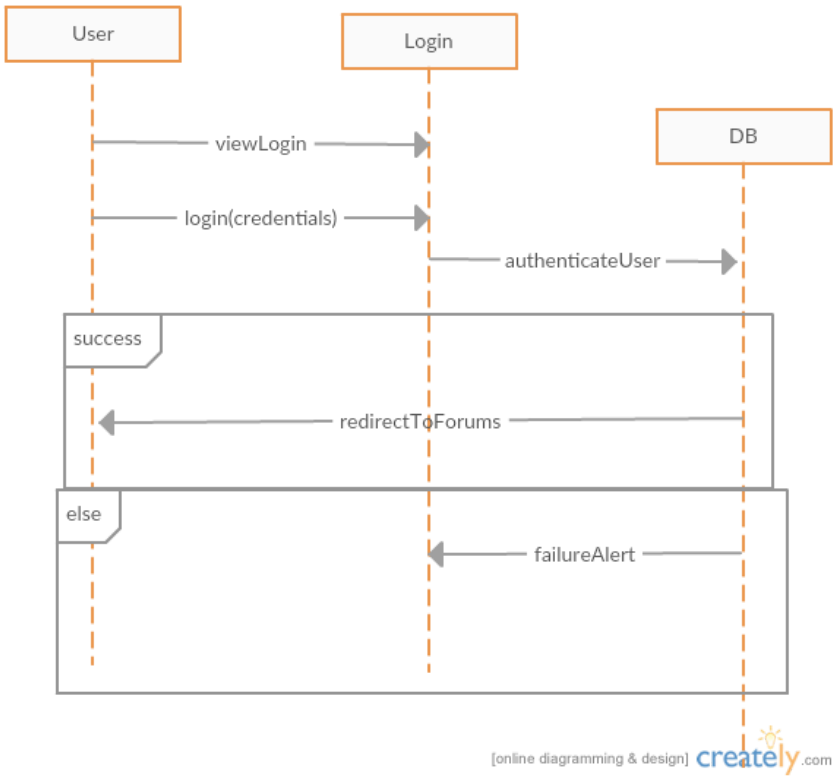
Design-level Sequence Diagrams

Post Question	
Post Answer	

Manage Post	
Manage User	

Add Student to Course	
Manage Course	

View Forum	
Vote	

Authenticate User	

Design Decisions

The classes were originally designed, so that they would remain coherent picture of the concepts behind the implementation. For example, a student may be enrolled in multiple classes, and a class may have multiple students enrolled. Some GRASP patterns were also used in the design of the system to determine how the different classes interact.

For example, the information expert pattern was used in how the forum and question pages are rendered. Forum should know all the questions related to it. Therefore, when you view a forum page, you can easily view all the questions associated with that forum. Similarly, question should know all answers related to it. Also, an answer should know all the votes associated with it. Therefore, the total tally will be visible on the answer page, as shown in the “Vote” sequence diagram.

The creator pattern was also used. User is responsible for creating forums. Although a user also semantically creates questions and answers (that is, the conceptual student creates a question), the responsibility for creating these falls to the class that contains these. For example, since a forum contains questions, a forum is responsible for creating questions. Similarly, since a question contains answers, the question is responsible for creating answers.

This hierarchical form of creating also supports low coupling, since a user shouldn’t have to know all the inner details of the question and answer. For example, previously, user had to know the forum id when they created a question or the question id when they created an answer. By allowing the forum to create the question and the question to create the answer, this information isn’t necessary for the user to know.

High cohesion is also maintained because none of the classes are very complex. Each class has very few methods.

Currently, the functionality includes a sign up option, which may or may not remain depending on the future vision of the project. The controller pattern is used for this, because what should be responsible for dealing with new users signing up? Semantically, it doesn’t make sense for any of the existing classes to handle this event. In fact, by making one of the existing classes handle it, then cohesion is lowered. Therefore, a registrations controller was specifically added to handle this.

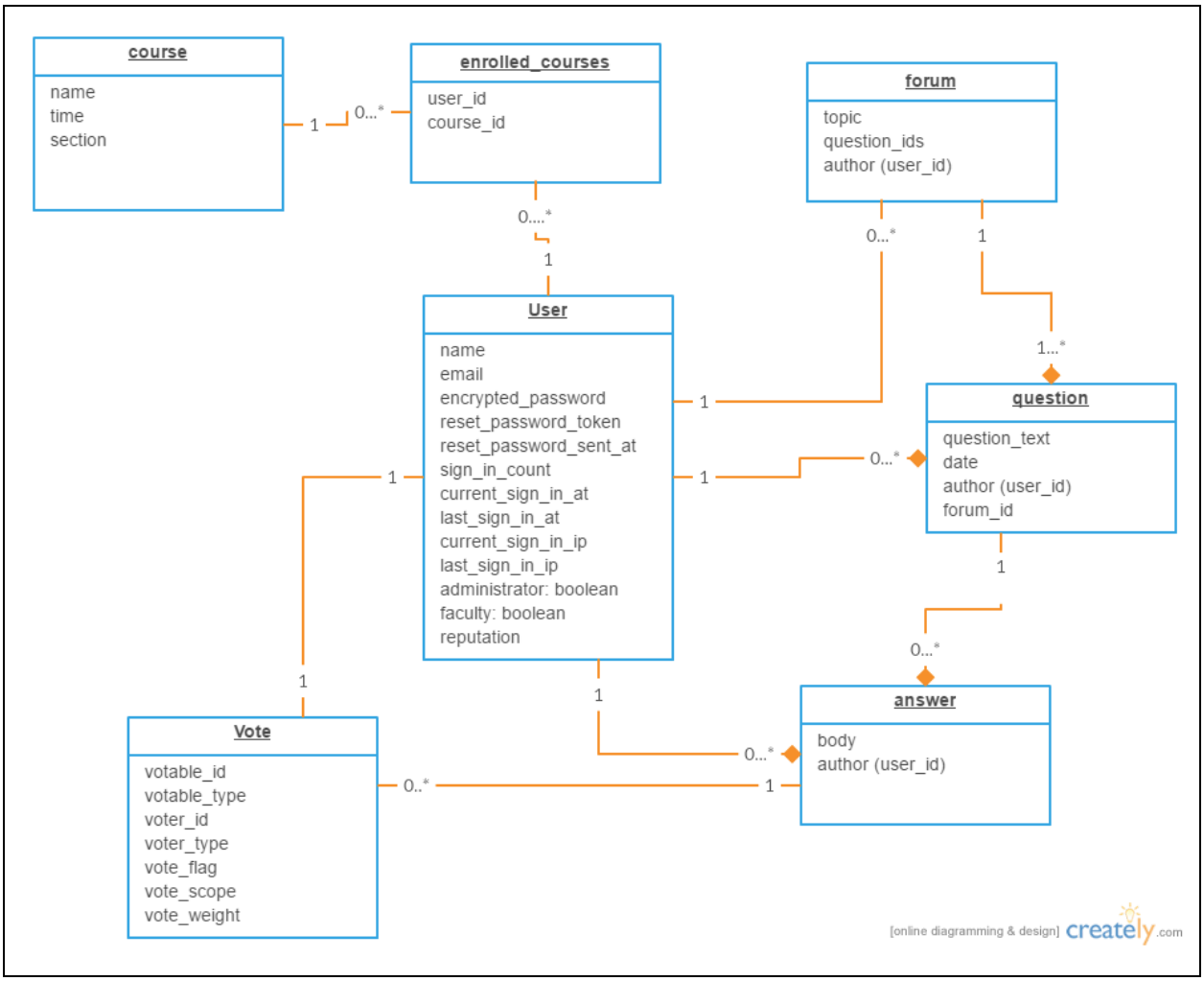
The users are an example of the polymorphism GRASP pattern. All of student, admin, and faculty are of the same type: user. However, the views of each user differs. For example, the edit/delete buttons for users and courses are only visible to the admin user. Therefore, the views for the admin must differ from the views for the student/faculty users. Similarly, the ability to add a course and register a student for a course should only be accessible to the admin users.

Other than those factors, this iteration was largely a start from scratch once we realized the flexibility and ease that various Ruby gems offered. Previously, there was a

lot of by-hand implementations of various backend handlers. By adding in the devise gem, this squared away most of the handling of user authentication. The gem only stores last_sign_in_at and last_sign_in_ip in the user table of the database but the other information is at a decent level of security. Some of the modules that are packed in within Devise that we use are recoverable, registerable, rememberable, trackable, timeoutable, validatable and lockable. There are also two other modules that are included which are confirmable (confirmation email) and omniauthable(used for multi-provider authentication). The passwords are hashed and stored locally without being accessible. Being apart of the main database authenticatable module is hww this is accomplished.

Ruby 5 handles a lot of the cookies, where they are stored in Ruby's Cookiestore, and sessions of users but Devise takes over for sessions. The session object is stored as current_user. Through this current_user object is how our app knows who is posting and or the type of user that is logged in such as faculty, student or admin. To add a bit more robustness to the app, we used Googles reCAPTCHA to the users login page that has to pass before the current_user session is created.

Database Design



The main changes in the database design were with the user and vote tables. The vote table needed to be added for up/down voting functionality on answers. The vote table and its columns are used by the acts_as_votable gem. Columns needed to be added to the user for authentication purposes with the devise gem.

In addition to what's shown on the database schema above, each table also has a column for created_at and updated_at to keep track of the times that an item was added to the database and when a row in the database is edited. These columns were automatically added to the tables during each migration.

Semantically, there is a many to many relationship between users and courses, since a course could have multiple students enrolled, while a student can be enrolled in multiple courses. Since this is a relational database, an additional table had to be added to join those two tables to represent this many to many relationship. This table was

called enrolled_classes. The function of the table was simply to include listings of pairs of student_ids to course_ids.

In addition to the commands used in the last iteration, the following rails commands were used to update the db for this iteration:

- `rails generate scaffold Votable user:references voter_type:string vote_flag:boolean vote_scope:string vote_weight:integer`
- `rails generate scaffold Enrolled_Courses user:references course:references`
- `rails generate migration User encrypted_password:string reset_password_token:string reset_password_sent_at:datetime remember_created_at:datetime sign_in_count:integer current_sign_in_at:datetime last_sign_in_at:datetime current_sign_in_ip:string last_sign_in_ip:string administrator:boolean faculty:boolean`
- `rake db:migrate`