TABLE OF CONTENTS

Discussion of Changes

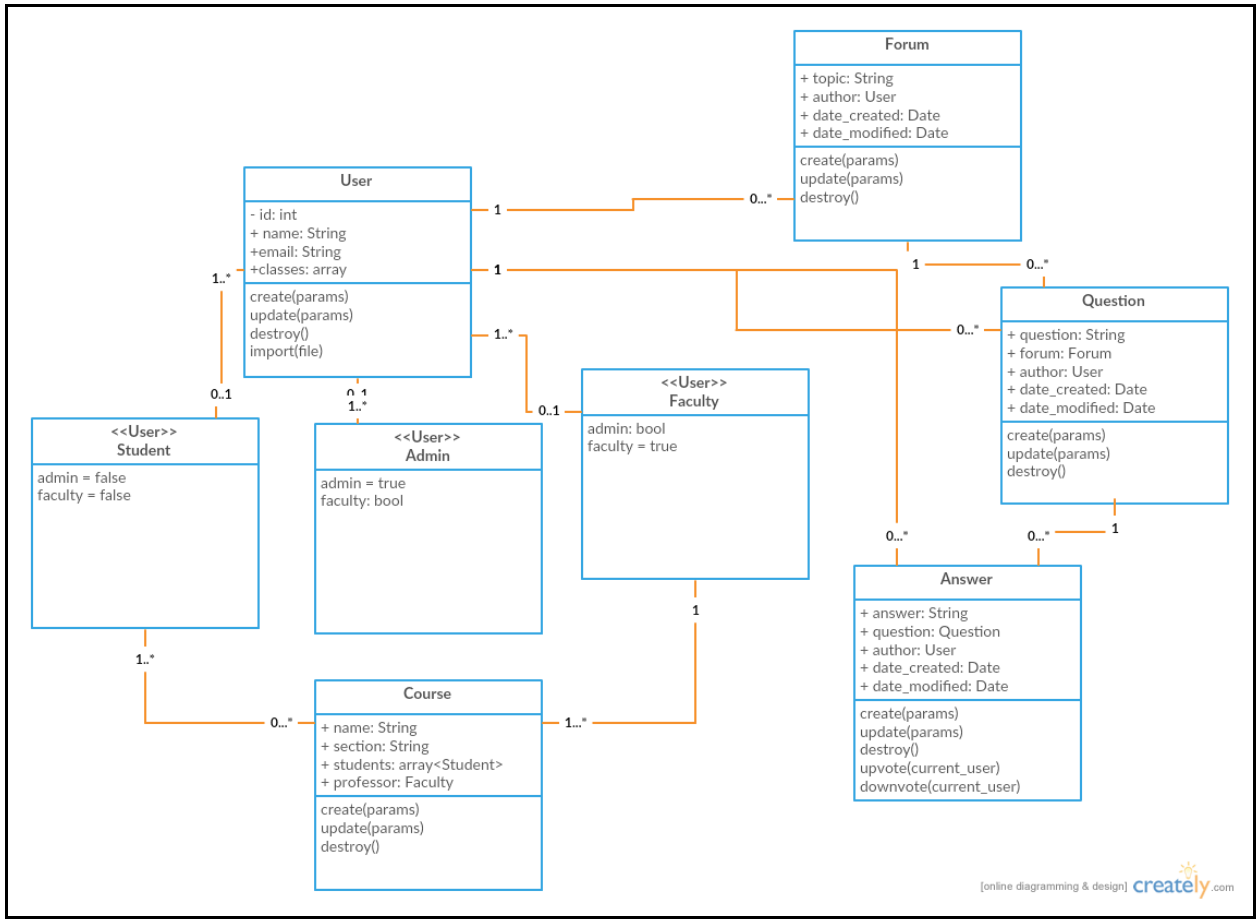
Generally speaking, issues were fixed and usability was improved during this iteration. For example, before this iteration, all the forums would be listed in a giant list, no matter how many existed. During this iteration, we added pagination, so that forums would be split across pages, if there are too many to show on one page. Similarly, this was done for the courses and users pages as well.

Also, links were added from a user's profile to their respective courses, and also from a course page to its respective students, for ease of navigation between these items. A search option was added to the navigation bar as well, so users can search for questions, so it's easier to find answers to questions they have.

A few bugs were fixed as well. For example, changing the type of a user wouldn't work on update, so now that feature is fixed. The buttons were also fixed on the edit course page, and buttons were added to the table of courses, so that you can edit/delete a course, without navigating to the course page first.

As an enhancement, a reCAPTCHA was added to the sign-in page to verify that users aren't robots, so that forums wouldn't be automatically populated by some hacker.

Design Class Diagram



All of the classes for the app represented objects in the domain model. It can be seen that all of these objects were listed in the original project description. Forums, questions, and answers had to be included since that's the main functionality of the app: to ask and answer questions of other students. Course needed to be added since students want to see what courses other students are taking, as well as see which students are in their courses. The different types of users had to be included for various authority purposes, as described in the project. For example, only the admin can delete posts. Originally, registrar was included as well. But as a team, we decided that the registrar should simply be included under admin, since their rights are essentially the same. There wasn't anything to be gained by splitting the two users into separate types, in our opinion. Not to mention, if a registrar gains administrative rights, they can also delete inappropriate posts, which means faster response time on removing those.

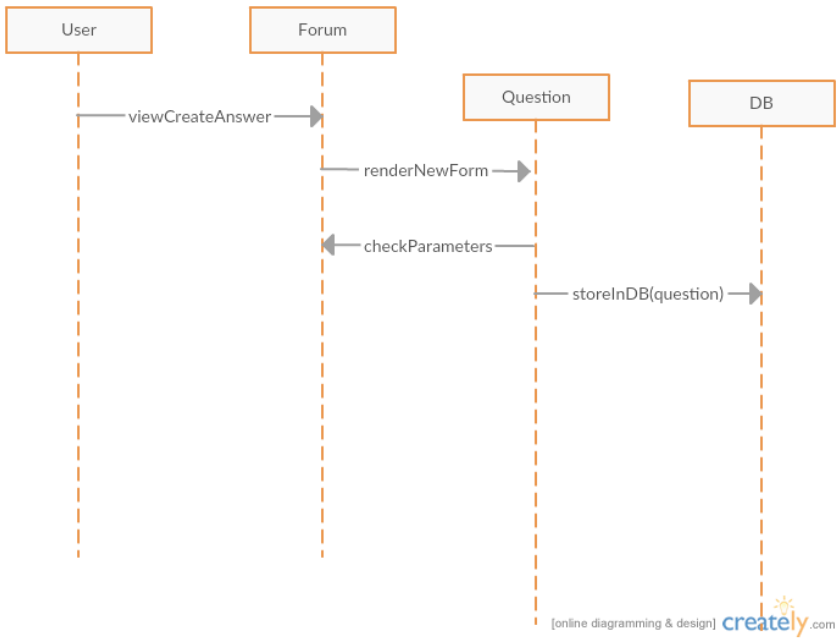
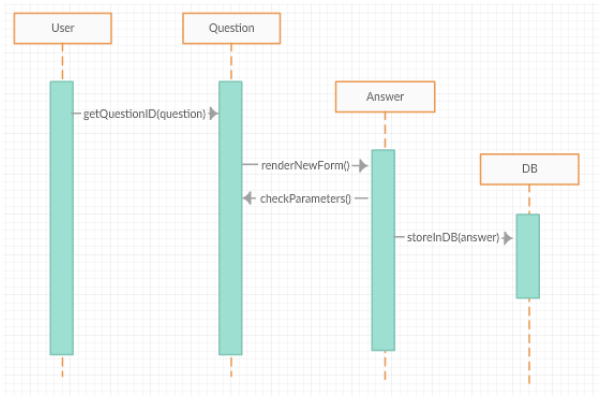
Originally, the user superclass wasn't included. However, it immediately became apparent that repeating so much information would be wasteful, which is why there is a user class, which has information that all users need, whereas the user types are simply

differentiated by boolean values. The rest of the relationships were determined by business (or should we say “school”) logic. For example, a forum should be able to contain multiple questions, since a forum is simply a topic which may encompass multiple questions. A question can have multiple answers, since some answers may be incorrect or may offer a different opinion. Also, classes generally contain multiple students (but at least one, or the class would be cancelled) and have a single faculty member as the professor. Although there are exceptions to the rule of a single professor, (for example, mobile development with 3 professors), generally, there is one professor who is considered the main professor, which we can umbrella the course under.

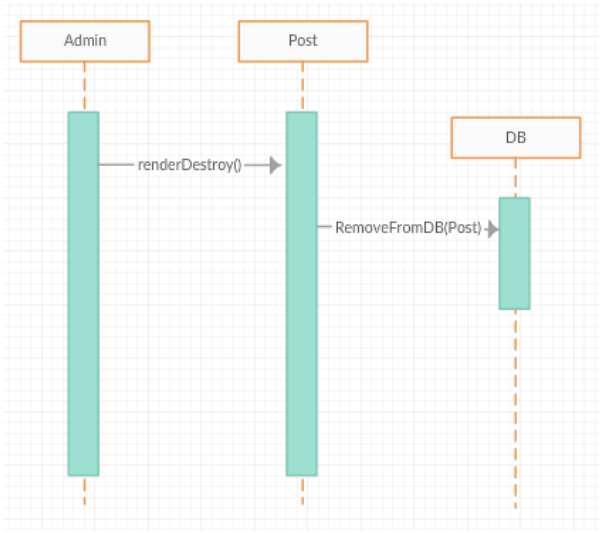
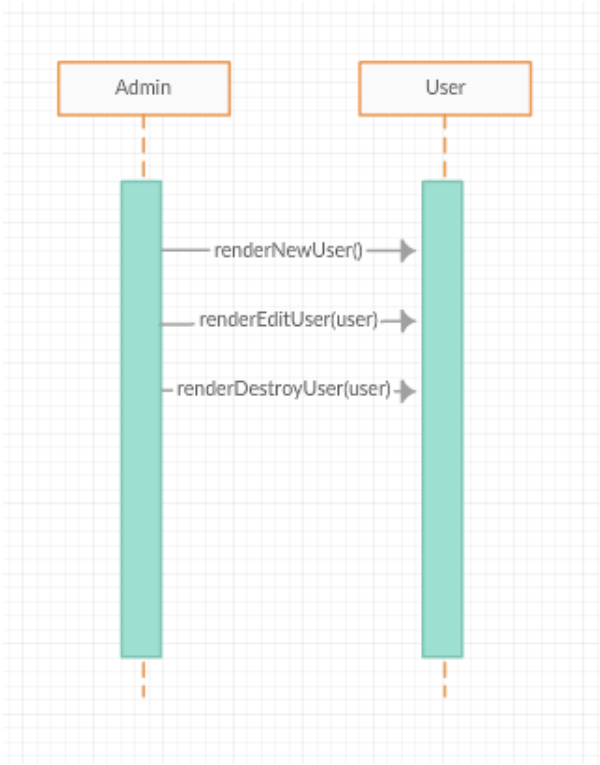
For all the classes, there really only needed to be the three main methods: create, update, and destroy, with the exception being answer with the upvote/downvote methods as well. However, those three methods encompass essentially the entire functionality of this app.

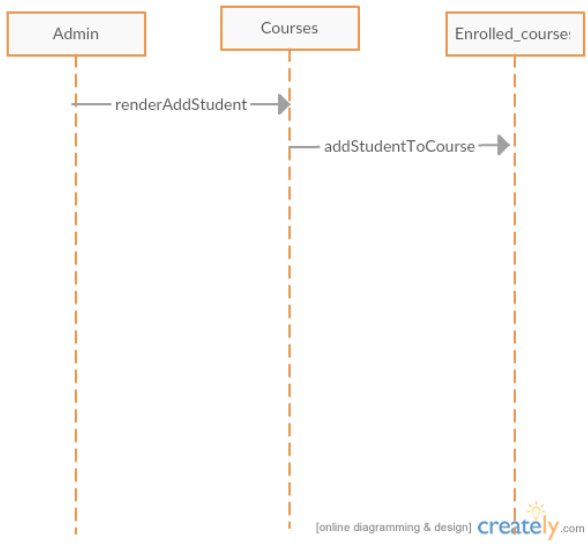
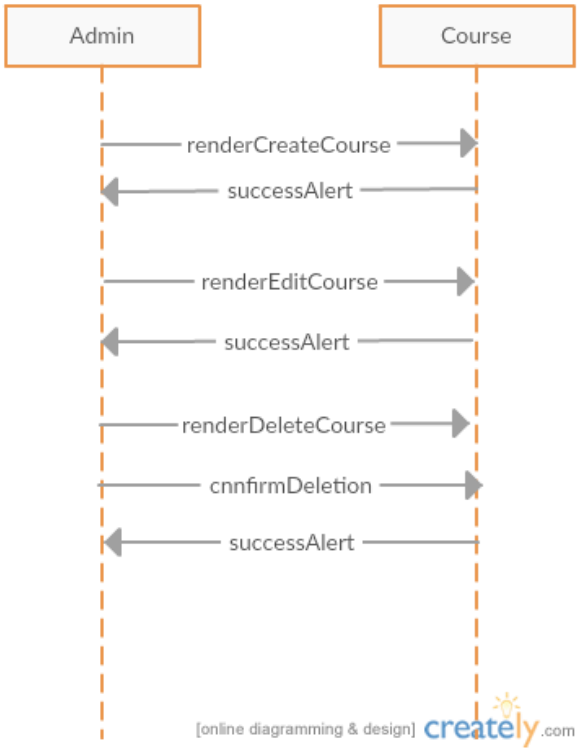
No classes were added during this iteration, since the main functionality was covered in previous iterations. However, a search controller was added for the search option.

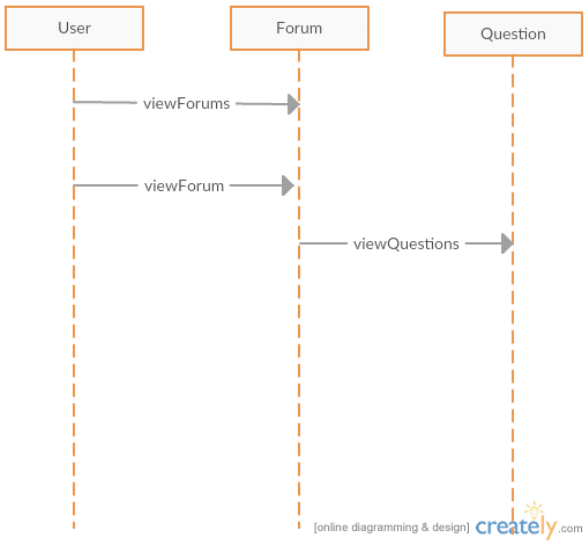
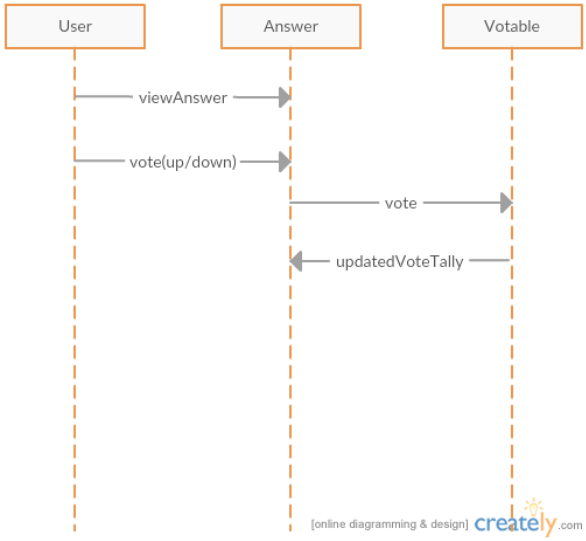
Design-level Sequence Diagrams

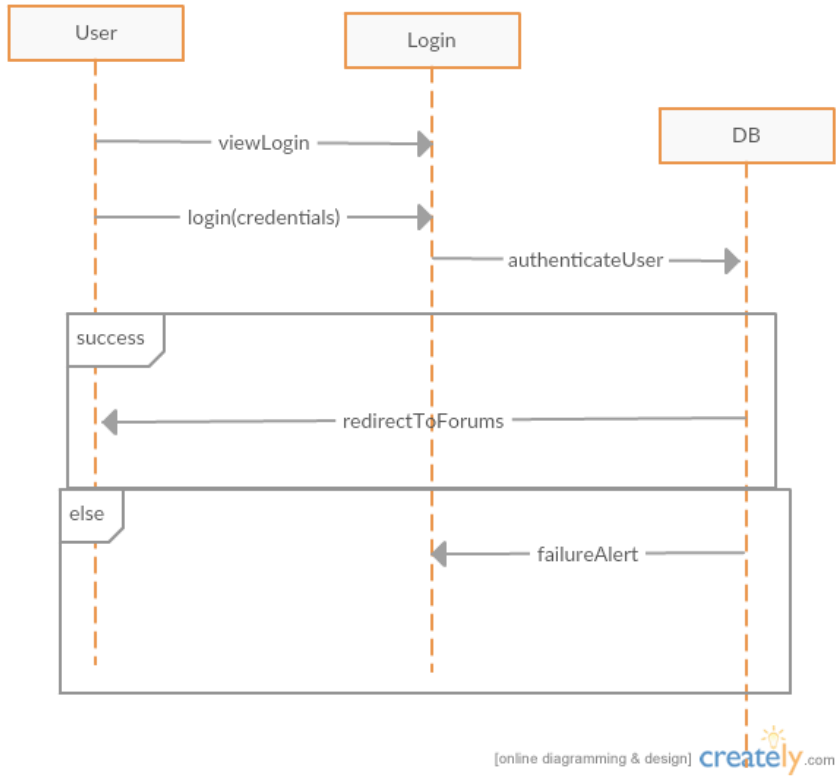
Post Question	
Post Answer	

5

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 a 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. The controller pattern is used for this, because what else 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. Similarly, a search controller was added during this iteration to search for items, because this task shouldn’t fall to any of the existing classes, since it’s possible that we would want the ability to search different fields, such as users or questions.

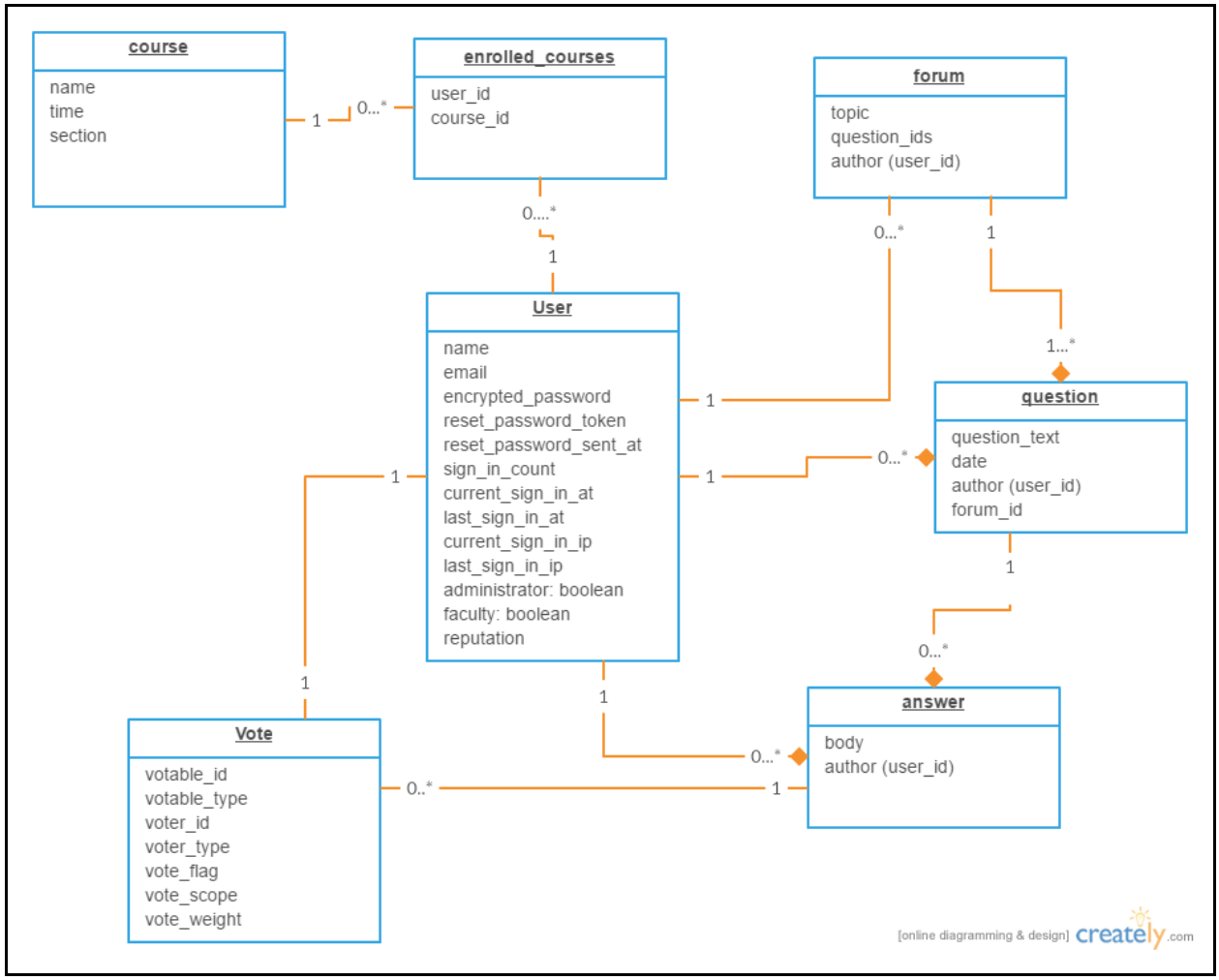
The users are an example of the polymorphism GRASP pattern. All of student, admin, and faculty are of the same type: user. However, the views of each user differs. For example, the edit/delete buttons for users and courses are only visible to the admin user. Therefore, the views for the admin must differ 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.

The GoF Template Method pattern was inherently used, as the application of the polymorphism, for the users. For example, the view of the questions page is essentially the same for all users: show a list of the questions related to a forum, which will link to their respective answers. This basic UI framework provides the template for the UI for all users. However, the edit/delete buttons only appear for admin users, or the user who created the question. Therefore, the system must insert those buttons for all admin users, or determine if a user created the question to display for non-admin users. This same method arises for all the UI pages which differ for different types of users (which, in fact, is every page except the forums page).

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 how 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 Google's reCAPTCHA to the users login page that has to pass before the current_user session is created.

Database Design



The database design originated with the main tables, which we assumed based on the project description: user, course, forum, question, and answer. These tables all correlate to a class as well. The database evolved in later iterations to include the vote table and enrolled_classes table, as well as extra columns in the user table. Besides those three changes, the database remained the same during the entire project, because the problem area was understood well enough at the beginning to get those tables correct initially.

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. This gem stores last_sign_in_at and last_sign_in_ip in the user table of the database for authentication.

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.

These are the database commands used during this entire project:

- `bin/rails generate scaffold User name:string email:string password:string type:string reputation:Integer`
- `bin/rails generate scaffold class name:string time:string section:string user_id:Integer`
- `bin/rails generate scaffold forum topic:string`
- `bin/rails generate scaffold question question_text:string date:string user_id:Integer forum_id:Integer`
- `bin/rails generate scaffold answer answer_text:string date:string user_id:Integer`

- `rails g migration add_user_type_to_user user_type:integer`
- `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`