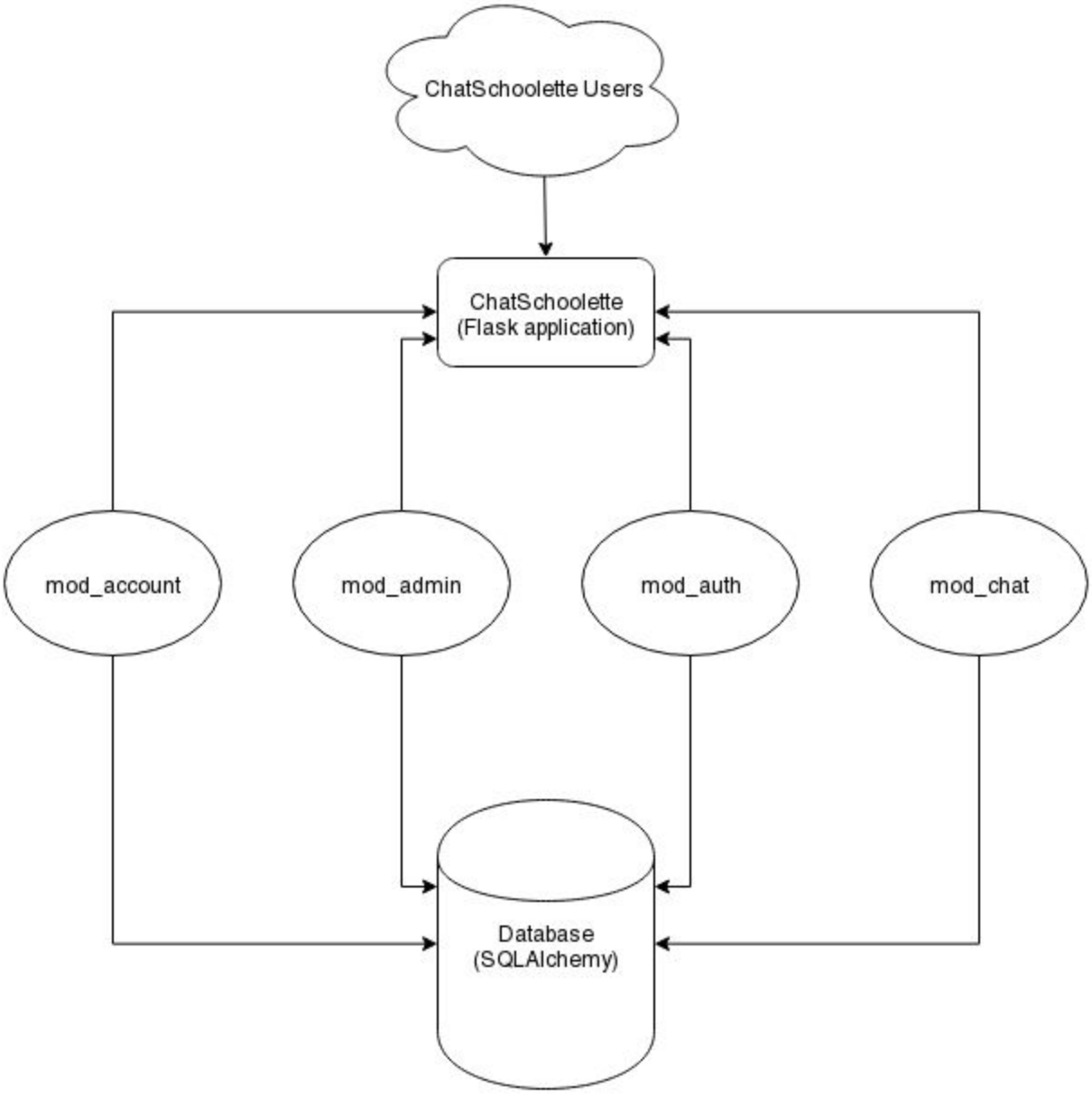
ChatSchoolette

Incremental and Regression Testing

Team 1: Logan Gore, Kyle Rodd, Sang Rhee, GeonHee Lee, Stephen Hong, Tyler Springer

Classification of Components



ChatSchoolette is comprised of four major modules: ACCOUNT, ADMINISTRATION, AUTHORIZATION, and CHAT.

The ACCOUNT module stores all of the user's data. It has methods for retrieving and updating a user's profile; this includes items like the user's profile picture, profile description, list of interests, gender, and birthdate.

The next module we have is ADMINISTRATION. This module defines what administrator accounts are authorized to do. This module defines functions for things like listing ALL site users, banning users, reading a user's chat logs to determine if they've been behaving inappropriately, or viewing a user's profile. This module is necessary to make sure ChatSchoolette remains a clean and fun environment for students to make new friends.

The AUTHORIZATION module defines all of the functions for determining if a user really is who he or she says they are. This module has functions for logging users in and out, resetting lost passwords, and loading user's into sessions so cookies can be tracked for functions like "Remember Me."

The last module is CHAT. Unsurprisingly, ChatSchoolette relies heavily upon the CHAT module to function well as a website. The CHAT module defines the functions for users to request a chat partner, list their chatting preferences, and (obviously) connecting to a chat session.

All of our modules are tied together by the Python-Flask framework. If you've never seen Flask before, it basically intuitively ties everything in a project together as long as the individual Python files are defined within a module. As I just outlined before this, this is what we've done. As a result, any change to an individual submodule is automatically propagated throughout the site with no other changes required within the code. Because we are using Python-Flask, we never have to have the modules talk to each other. Moreover, our components should NOT interact with each other directly, EVER. Any resource needed by a module should be retrieved directly from the database or through the Flask API. This is the point of the modularization of the project.

We use SQLAlchemy for our database operations. This is an incredibly powerful package that automatically defines all INSERT, UPDATE, DELETE, etc. queries for you as long as you define classes that extend "db.Model". This leads us to not ever having to write or rewrite SQL code as all changes are automatically picked up by SQLAlchemy. Furthermore, if we ever want to move from SQLITE to another database, we just redefine the Database URI and SQLAlchemy can migrate everything for us.

Our group used bottom-up testing to develop ChatSchoolette. We wanted to get the individual sub-modules and methods working before we integrated them into the site as a whole. Since websites are relatively simple things to code (compared to the actual algorithms USED by the website), we weren't too worried in making sure that things like loading "/index.html" was loading a page in the correct format for the user. It was much more important for us to ensure that the individual functions that created, retrieved, updated, and deleted our data were working correctly and efficiently. Because of this, we decided to write all of our test cases from a bottom-up style of testing.

MODULE: ACCOUNT

Incremental Testing

#	Description	Severity	How to Correct
1	When pressing the register button, causes an error on the back end.	1	Change backend code to allow for proper registration.
2	Text boxes allow for unlimited characters to be entered and has no filtering.	2	Set filters and limit text length on front end.
3	Any file type is allowed to be uploaded into profile picture.	2	Set a filter to check and allow only for appropriate file extensions.

Regression Testing

#	Description	Severity	How to Correct
1	When adding the new options for gender, caused an error when invoking server for forgetting to add attributes to table in database.	2	Add choices to table in database.
2	When adding filters for text fields, used wrong regex to filter characters.	3	Change regex to allow characters appropriate for text fields.
3	When adding interests text field, no string parsing done and accepted whole string as one interest.	2	Add code to parse strings and store in database.
4	Checking ranges when user POSTs for a select field, we made range always have only one option.	1	Set ranges to each appropriate select field to allow for choices.
5	After allowing for edit of profile page, if no changes are made to select field, stores empty value in database.	2	If no changes were made to select field then just do nothing for that attribute's stored value of the table.

MODULE: ADMINISTRATION

Incremental Testing

#	Description	Severity	How to Correct
1	Block user button should not permanently delete the user	2	Create attribute on user table on database to check block history
2	When viewing banned user lists, warned users should not show up	3	From block history attribute, we differentiated banned user and warned user
3	Warning email should not be sent with no content	1	Add message body on the SMTP header when sending a warning email

Regression Testing

#	Description	Severity	How to Correct
1	Since one attribute has been added to user table, previous queries did not work correctly	1	add new attribute on insert and modify queries
2	block history attribute causes that even normal users are classified as warned user	2	classified them into 3 levels; normal, warned and blocked
3	Warning email content was too generic that it sounds like everybody did same inappropriate action	1	We keep track of inappropriate action that the user did, and send the related warning email

MODULE: AUTHORIZATION

Incremental Testing

#	Description	Severity	How to Correct
1	Passwords stored in plain text in database, impeding security	1	Add a password hash when setting

			passwords
2	Any string considered valid for School email address	1	Add filter requiring the '@' character and '.edu' suffix
3	One email could be used to generate multiple profiles	2	User is only validated if email is not currently in the database
4	Could register without filling out all attributes	2	Don't allow page redirection unless all attributes are filled out accurately

Regression Testing

#	Description	Severity	How to Correct
1	Check password fails since the plain text and hashed passwords are different	2	In the validate password function, compare the hashed versions of both passwords
2	After hashing string, password couldn't fit in the allotted password space in the database	2	Size of the strings in the password column was increased to fit the whole password.
3	Page would only reload if not all attributes were filled out	2	Error messages were added by fields that weren't filled out

MODULE: CHAT

Incremental Testing

(Tests we should pass by adding a new feature)

#	Description	Severity	How to Correct
1	User has no way of specifying chat partner preferences	2	Add form user can submit to list chat preferences

2	Routing method for chat returns a 404	2	Create new api_rule with Flask to route users to create chat subchannel
3	No security checks on joining a chatroom	1	Make sure only authorized users can load a chat page for a certain chatroom

Regression Testing

(Tests we broke after adding a new feature)

#	Description	Severity	How to Correct
1	After allowing user to specify chat preferences, no regards were made to these preferences (still random matching)	2	Before matching users, check that all fields of match preferences conform to user specifications
2	Upon updating routing tables for chatrooms, default chat page (preference submission) broke due to ambiguous rule	2	Check all application api_rules to see where conflict existed and correct routing tables
3	After security checks, user is able to sit in a room alone without a chat partner if partner never connects	3	Create a button to allow user to edit preferences if no matches are made