A) The application allows for users to interact with other users by posting questions and answers. The user, once registered and logged in, has the option to post a question for other users to answer, or they can choose to search a post themselves. If they choose to search a post, they can provide one or more keywords to filter the post results in order to narrow down their search. After that, the user can perform various post actions on the post that they have selected, such as giving the poster a badge, posting their answer to a post (if it is a question), etc.

B) In **welcomeScreen()**, the user will be given an interface where they choose if they are “registered” or “not registered”. If they are registered, they will be redirected to **loginScreen()**. If not, they will be redirected to the **registerScreen()**.

In **loginScreen()**, the user will be asked to provide their uid and password (which will be hidden via getpass). If uid or password does not match a value in the *users* table, then the user does not exist. The function will loop depending if the user wants to try again. If they do not want to try again, it goes back to the **welcomeScreen().** If login is successful, user will be redirected to **sysFunc()**

In **registerScreen(),** the user can provide uid, name, city, pwd via input. The provided uid will be checked by helper function, **isUniqueUser()**. This function will return true if the uid is NOT in the database, else false. The user can try again to input a unique uid.

In **sysFunc(),** the user can choose to either post a question or search for posts. This mimics the interface of the welcome screen.

In **postQuestion()**, this is simply the interface where the user can create a post. Each post is added to the posts and questions table. The system will generate a random 4 digit number via random module. If the random number exists as a pid in the *posts* table, then it runs and gets a new random number again. This will loop until a unique number is generated.

In **keyword_search()**, the user is able to provide one or more keywords which will be stored in a list. For each keyword that they enter, they will be given the option to enter another keyword. Based on the keywords given, post results will be displayed along with information about the post. The user selects a post that is displayed by entering the number (eg. enter ‘1’ to select post result 1), or they can type in ‘s’ to see more posts (only if that option is displayed for them, which indicates that more posts can be displayed). The postID that the user has selected will be returned and can be used in the post actions. If no posts are displayed, the user will be asked again to enter keywords until a postID is returned.

post_action(), The only input is the pid of the post for which we may perform a post action on. This function displays the Screen which contains all the post actions available to the user for this post. It prompts for the user's choice of action (prompting over and over until a valid option is chosen) and calls the corresponding function for the action chosen by the user passing in the pid as input. If the user decided to take a post-action after the post-action is finished we return to this function which then asks the user if they want to perform another post action or not. If they choose to perform another post action we start this function again with the same pid and if not we go to the system functionalities screen.

In **mark_accepted()**, the user will receive an error message if they are not a privileged user or the post that they have selected is a question. Otherwise, the user will be notified if an accepted answer already exists, and then the user will be asked whether or not they would like to mark the post that they have selected as the accepted answer.

In **give_badge()**, the user will receive an error message if they are not a privileged user. Otherwise, they will be displayed all the possible badge options, and the user chooses which badge to give to the poster by entering in the number beside the badge option. The user cannot give any sort of badge to the same poster on the same day.

vote(), this function takes in as input the pid of the post for which the user wants to add a vote. It then makes sure the post exists in our database and that the user has not already voted on this post. If both of these are true then it adds a vote for our post to the database and assigns it a new and unique vno. With this post-action completed we return to post_action()

tag(), this function takes in as input the pid of the post for which the user wants to add a tag. It then makes sure the current user is a privileged user (by checking that our user id appears in the privileged table) and that the post exits. Then it prompts our user for the tag it would like to add. Based on their response it adds a tag for this post to our database where the content of the tag comes from our user's input. With this post-action completed we return to post_action()

edit(), this function takes in as input the pid of the post for which the user wants to edit. It then makes sure the current user is a privileged user (by checking that our user id appears in the privileged table) and that the post exits. Then it prompts our user whether they would like to edit the title of the post and/or the body. Based on their response the user is then prompted for the new title and/or new body (based on their previous choices). It then updates our post to reflect the new title and/or body. With this post-action completed we return to post_action()

postAnswer(), this takes in a pid of a question post (assertion was made beforehand that post is a question). This will simply take in the input for the title and the body of said answer post. This also follows the generation of random unique pid as we've seen on **postQuestion()**.

C) The testing strategy involves checking if the user inputs are valid, by checking to see if their input matches with one of the options that we have provided for them. Other cases may involve exception handling in order to prevent the application from crashing and some tests to make sure that program is responding correctly to user inputs. Here are the major cases we considered for testing.

Cases to consider and handle when registering a user, the user enters an already existing uid. This can be solved by allowing the user to try again via a while-loop.

Cases to consider and handle when creating a post (question or answer), if randomly generated pid is already existing. This can be solved via a while-loop, as the system randomly generates a 4-digit number until it has one that does not exist in the database.

Cases to consider and handle when searching for posts includes the user not entering a valid keyword, the user enters valid keywords but no posts can be found, the user enters the same keyword but written in a different way (eg. life vs LiFe), the user enters valid keywords but no posts can be found, the user does not select a post result that is displayed (eg. the user has the choice of selecting post1 or post2, but they enter post3), the user wishes to see more posts even though all the possible posts have been displayed, the user has the option to choose a post that is displayed, or they can choose to see more, but they enter an input that does not belong to either one of these options.

Cases to consider and handle when giving the poster a badge includes the user giving a badge (regardless if it is the same badge or not) to the same poster on the same day (this gives a unique constraint error which can be handled by providing an error message), and the user logged in is not a registered user.

Cases to consider and handle when marking an answer post as the accepted answer includes the answer post that the user has selected to mark as the accepted answer to a question is already the accepted answer, and the user logged in is not a registered user.
Cases to consider when choosing which post action to take: The user chooses any of ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’. The user is taken to the correct post-action screen. The user enters ‘exit’. The program exits. The user enters ‘EXit’. The program exits. These two tests are repeated for ‘logout’ as well. The user chooses a non-valid option. The user is told this is a non-valid option and they are prompted once more.

Cases to consider when voting: A vote already exists for this post from our User. This vote should be rejected. User adds a vote to a post for which they have never voted. This vote should be recorded in the database.

Cases to consider when adding a tag: A non-privileged user tries to create a tag. The tag should instantly be rejected. A tag already exists for this post with the exact same tag. Adding our tag should be rejected. A tag already exists for this post with the same tag and different case (ie richard vs RIChaRd). Adding our tag should be rejected. The user enters ‘exit’ when prompted for a tag. The program should stop running. The user enters ‘EXit’ when prompted for a tag. The program should stop running. The same two tests done for ‘exit’ should now be done with ‘logout’ except this time the user should be logged out. A tag does not exist for this post with the same tag and the same tag does not exist with different case (ie richard vs RIChaRd). This tag should be added to the database.

Cases to consider when editing a post: A non-privileged user tries to edit a post. The edit should instantly be rejected. The user enters ‘exit’ or ‘EXit’ when prompted for whether or not they would like to edit the title or body or when they are prompted for either the new title or new body. The program should stop running. The tests done for ‘exit’ should now be done with ‘logout’ except this time the user should be logged out. The user enters a non-valid choice when prompted for whether or not they would like to edit the title or body. They are told this is a non-valid option and are prompted again. The user enters ‘yes’ for editing the title and ‘no’ for editing the body. Then enters anything when prompted for the new title. The post should be edited where the only field edited is the title which is now set to the ‘anything’ the user entered. The reverse for when the user enters ‘no’ to edit the title and ‘yes’ to edit the body.

D) After reading the specification, we divided up the parts into 9 tasks which ensures that each member has the same number of tasks to do. 8 of these tasks are from the system’s specification, and the other task is the login system itself. The goal of our first meeting was to divide these tasks up among us.

Elena was in charge of three of the system functionalities, which includes “Search for posts”, “Post action - Mark as the accepted”, and “Post action-Give a badge”. Out of these three system functionalities, “search for posts” required the most time to complete and was finished last, which was approximately a week (where 3-6 hours was spent each day). The first code implementation did not work as expected since adding up the total keyword count was difficult to implement, and the placeholder ? could only correspond to one value rather than more than one (i.e. ...LIKE ?, (keyword1 OR title like keyword2)). In the second attempt, nested dictionaries were considered (eg. {keyword1: {pid1: ___, ...poster1: ____,....poster3: ___}). However, adding the total keyword count by using the nested dictionary also did not work. A third approach was taken that involved the use of a list of tuples and dictionaries, which ended up working as expected. The other two system functionalities took approximately 2-3 days to complete.

Isaias was responsible for creating the login system, and two of the system functionalities, which are “Post a question” and “Post action-Answer”. The first thing Isaias did was create a draft diagram to show the basic function calls that would be made throughout the program. Laying out the login system should provide an organization as to how the other functions would be called. Isaias tried to make his functions as small as possible, abstracting functionalities as much as possible. This includes checking if a uid or pid is unique, retrieving user data and basic logout/exit. These helper functions were used in other functions created by the other team members as well. The implementation of all these functions took over 8 hours (aggregated).

Richard was in charge of the remaining three system functionalities, which are “Post action-Vote”, “Post action-Add a tag”, and “Post Action-Edit”. Going into these tasks Richard had a feeling that all three would take a similar amount of time since while the “Post action-Vote” talk had less work (since it didn't require the user to be privileged) therefore he gave himself 4 hours per task spread over 3 days. In fact this is not how it ended up going. The first task took substantially longer than expected. This is because Richard underestimated how long it would take to go over the lab slides and understand how to work SQL inside of python. In addition he decided that it would be easier for him (and the rest of the team) to create functions to drop tables, create tables, insert data and then one function which combined all three called create database. This helped a great deal as it made creating the database and deciding what should be in it very easy thus helping a lot in figuring out bugs and testing. In the next night of working on the project Richard Expected added delays again however based on the knowledge he had gained the night before, and the work he had already done, the next two tasks were much easier to get done and Richard finished them both in the same day. Then he had to incorporate his work into the larger project which was a little difficult to try and figure out how everyone else’s code flowed and what he needed to add in once he got a hang of this he created the display post actions function which Displays the Screen which contains all the post actions available to the user. Then prompts the user to enter their choice of action and takes the correct action accordingly. He then used his previously created functions to deal with his three post-actions. Then he received some feedback from the group and made changes accordingly.

In order to keep the project on track, a Github repository was created in order to keep our files organized in one place. Meetings were scheduled from time to time where members of our team would update one another on our progress and ask for help from other members if needed (which can also be done through issues on Github).

Decisions we have made which is not in the project specification or any coding you have done beyond or different from what is required: Richard decided to create functions to drop tables, create tables, insert data and then one function which combined all three called create database. This made it so that we could easily create new databases with the input we wanted to see in our new database easily customizable and visible. This function could be commented out if we wanted to work with a pre existing database instead of a new one. This was useful for debugging and for testing.