General overview - The main goal of our design was a simple modular overview. We opted for this design as it allowed us to create our own files to edit, individually test components easily, and have a clear understanding of what does what. However, we took the criticism from Project 1 and opted for less modularity overall. Aside from that, we basically adapted our design from Project 1 to Project 2 and made the necessary modifications to fit the spec. The flowchart for the program follows below.

Welcome Screen
->Main menue
-> post question
-> Search for question
-> post answer
-> list answers
-> vote answer
->vote question
->logout
->exit program

- - -User guide - - -
 To build the document store, run the command “python3 CollectionGenerator.py”, and it should find and insert the content of 3 .json files into the mongo database 291db.
 To start the program that operates on the document store, run the command “python3 main.py (port number)”, where post number is the name of the port the database is on. On start, the user can enter a user id, in which a small report displaying the count of all of their questions and answers with the average scores for both, as well as the number of votes they have cast. From this point, the user is moved to the main menu.
 From the main menu, the user can choose to do one of four things: - Post a question, in which they are prompted for a title, body, and any number of tags including no tags. - Search for questions, in which they are prompted for search terms and are presented with a list of questions. From here, the user can select a post and be prompted for actions on said post (actions follow below). - Logout, in which they are taken to the beginning of the program. - Exit, in which the program shuts down. When the user selects a post after searching for questions, they are prompted to do one of four things:

Post an answer, in which the user is asked for the body of the answer. - List answers, in which the user is presented with a list of all the answers the question has. The user can select an answer from there and either vote for said answer or return to the main menu. - Vote for the question. - Return to the main menu. - - - Software design - - -
Generally, every file incorporates the use of Terminal.py and pymongo to function. The reason for this is that Terminal has a function getPort() that returns the port given at the execution of main.py, as well as methods like printCenter(string) and clear() that allow for printing to the centre of the window as well as clearing it. Pymongo is needed for database operations, so it is used in all interface files and some screen files like WelcomeScreen and main.

CollectionGenerator.py Handled opening, reading, and writing the contents of the three json files into the mongo database titled 291db. Storing each file in it's
own respective collection (Tags, Votes, and Posts)

Terminal.py allows for easy access to the database port, as well as shorthand commands for printing strings.

Menu.py is the shorthand way to offer a choice to users. The main feature of Menu is printScreen(), which prints out the options added to it by addMenuItem(). It then returns the number of choice the user made.

Post.py is the framework for adding posts to the database. It relies on DateTime for adding time. There are 4 important methods in post:
The first of these is add_post. This method allows for a post, whether question or answer, to be added to the database. For questions, this also uses the method for tags being added regardless of existence (The process for adding tags i

s below)
The second and third is get_pid and get_tagID. These methods generate a unique pid for posts and tags. This is done through a binary search combined with a method for finding an initial min and max number for the binary search. This binary search returns a unique ID for whatever needs it.
The last important method is __check_tag__. This is the method that takes all of the tags and either adds them to tags if they don't exist, or increments the tag counter if it does exist.

PostPrinter.py is the framework for printing out the contents of a post. PrintTitle is responsible for the printing, and all it takes is a post in the form of a dictionary.

Vote.py is responsible for adding votes to the database. There are 2 important methods:
The first is makeVote. This is responsible for letting a user vote depending on if they already voted. The functionality of that check takes place in the method userVoted.
The second is getUniqueID. This is the same function as in Post.py, where binary search is utilized to find a unique id.

AnswerListScreen.py is responsible for listing the answers to a question. The main method is printScreen, where it prints the list of answers. To do this, it uses the functions in AnswerList and AnswerListMenu, which are responsible for grabbing all of the answers, ordering them in a list, and moving the accepted answer to the top of the list if it exists.

MainMenuScreen.py incorporates Menu.py to display and return the main menu choice the user makes. The printScreen function just returns Menu.printScreen() with the 4 choices being the four main menu choices.

PostScreen is responsible for the UI of posting questions and answers. To accomplish this, it uses the framework from Post.py and two methods: printQuestionScreen and printAnswerScreen. printQuestionScreen prompts the user for a title, body, and optionally tags. It then passes that information to Post.py to be added to the database. For printAnswerScreen, it prompts the user for the answer, and then passes it to Post.py to be added to the database.

SearchForQuestionsScreen.py is responsible for searching for the questions and returning the post the user selects. To do this, it uses a collection of small methods. It uses the methods in SearchForQuestions to get all of the matching questions, SearchQuestionMenu for adding all of the questions to the menu, and SearchForQuestionScreen.printScreen() to give the user a UI to work with.

SelectedAnswerScreen is responsible for giving the user options on how to interact with the answer. It relies on an answer being passed to printScreen() for it to function, and it returns whether the user wants to vote for the answer or exit to main menu.

SelectedQuestionScreen is similar to SelectedAnswerScreen, except it's for questions. The user is prompted to select either posting an answer, listing all answers, voting for the question or exiting to main menu. The method printScreen() allows this to happen when a post is passed to it.

WelcomeScreen is responsible for handling users, as well as delivering the report to the user. It is the first screen shown when the program is launched, and returns the user ID if given or None if not given. This return is used in almost all other aspects of the program, specifically anything where the userID will be added to a document field. This includes posting questions and answers, as well as voting on a post.

All of this is incorporated in main where WelcomeScreen gives the uid used for PostScreen and Vote, menu and MainMenuScreen give the user the main menu options, PostScreen and Post allow the user to post a question and answer a question, SearchForQuestionScreen gives a post for SelectedQuestionScreen to work with, which allows for the post to be answered by PostScreen or voted on by Vote or have its answers listed by AnswerListScreen, which gives an answer to Vote to be voted on.

- - - Testing Strategy - - -

We used the strength of modular design to aid our testing. We tested each individual module within said module, an d then in main from full implementation. We did this for every module in our project, and then did a final run-throug h to test for any UI bugs by throwing whatever we could think of at each part of testing. We also did a final check fo r posts, votes, and tags being inserted correctly.

As an example of testing modules, while creating the questions portion of Post.py, I (Cameron) tested each individu al case in the "if __name__ == __main__" area of the file, and tested adding a post with no tags or uid, adding a pos t with a uid but no tags, adding a post with no uid but tags, and adding a post with both a uid and tags. In each of the se cases, I manually authored both the uid and tags, as well as the pid. From there, I created the unique pid method, a nd did a singular test to see if it worked while manually checking each entry. I then ran a for-loop in the "if __name_ _ == __main__" area of the file and created 100 questions, and check each one of them to see if the pid was unique. When that was done, I implemented it into PostScreen and tested it from there, doing nearly the exact same thing mi nus the for-loop. When that was finished, I implemented PostScreen in main and tested it from there to fix bugs. Wh en the testing was over, I knew that there weren't any bugs because I fixed them from the ground up.

- - Group Work Strategy - - -

Our strategy revolved around using Github as version control, and picking which modules we wanted to do, then test ing them ourselves until we found them to be bug free in our own testing. When modules were finished, we would p ush them to our own branch, then merge with the main branch. This way, we could always have the most up-to-date version ready for the other person to work/test with. We used a discord server to update each other on progress, coor dinate on what to work on, and help each other with problems. In terms of work spread, I (Cameron) completed Post .py, PostScreen.py, WelcomeScreen.py, and main.py. I also did all of the bug checking for main.py, helped Chase w ith the indexing of the database, and wrote up this current report (aside from the section about CollectionGenerator.p y). Chase wrote up the rest of the files, and completed the readme.txt file. Chase also did Part 1 in its entirety. We m utually feel the work spread was about 50/50, as we often helped each other with the more difficult parts of the proje ct. We both spent about 20 hours each on the project, mostly on creating and implementing to the spec.

- - - Assumptions - - -□
For our project, we made the assumptions that the Id's of posts, votes, and tags were exclusively numbers, and that t hose numbers were strings inside the database. We also assumed that there was no character limit for user ID's, nor passwords. Finally, I (Cameron) assumed that answerCount didn't need to be updated in the question when an answ er is given to said question. That requirement was not in the spec, nor the marking rubric. □ □ □