

SUBJECT: Project Report for BookwormDB

CANVAS GROUP: E4

GITHUB REPO: <https://github.com/UT-SWLab/TeamE4>

TEAM INFORMATION:

Name	EID	GitHub Username
Jaelyn Bethea	jnb2634	jaelynbethea
Carlos Borja	cab6523	valerium-dev
Mrugank Parab	mp44675	mp44675
Santhosh Saravanan	sks3648	santhosh2000

1.0 INTRODUCTION

This document serves to reflect all of the work our team has completed for our EE461L team project. In this document, we provide an overview of the project and discuss the certain requirements associated with the project. Furthermore, this document outlines the design for our project from both front-end and back-end perspectives.

2.0 PROJECT DESCRIPTION

This project focused on creating a working, deployed app that emulated IMDB and provided useful information books, authors, and publishers. The name of the deployed app is called BookwormDB. BookwormDB was created for book lovers all around the world to explore new books and learn more about authors and publishers. Users of BookwormDB are not just for book readers. Future authors can find out more about possible publishing companies, students may be in need of a book for class, or a parent might want to check the content rating of a book their child is reading. Future authors can only find certain information about books from top authors and be able to view the content ratings of a particular book of interest, if they are interested in checking it out at a local library or going to a bookstore to purchase the book. Future companies who are interested in working with well-featured publishers can look at the authors and books

which have been written by these publishers and could schedule a meeting with them by learning about their headquarters and looking at Wikipedia and other sites for more information.

2.1 Requirements

Our team created a use case diagram to help us better understand how to design and implement the web application. To facilitate development, user stories were the main driver behind the project requirements. Each phase of the project included at least five user stories to implement. Table 1, Table 2, and Table 3 in section 2.1.2 show each user story, a description of the story, and the time measurements our team spent completing each story.

2.1.1 Use Case Diagram

The purpose of the use case diagram shown in Figure 1 below is to provide a high level view of BookwormDB as a system. The actors represent potential users of BookwormDB and the possible actions they have as users. For example, if a librarian visits our web app, then they may potentially want to find out more information on a book or they could be interested in the reviews of the book to see if they want to add it to their library. While the use case diagram lists different actors, all actors have the same potential actions/options when using BookwormDB. For example, a parent may also want to find out more information about a book. The use case diagram was very beneficial in ensuring that our team was designing BookwormDB in such a way that users could complete their interactions.

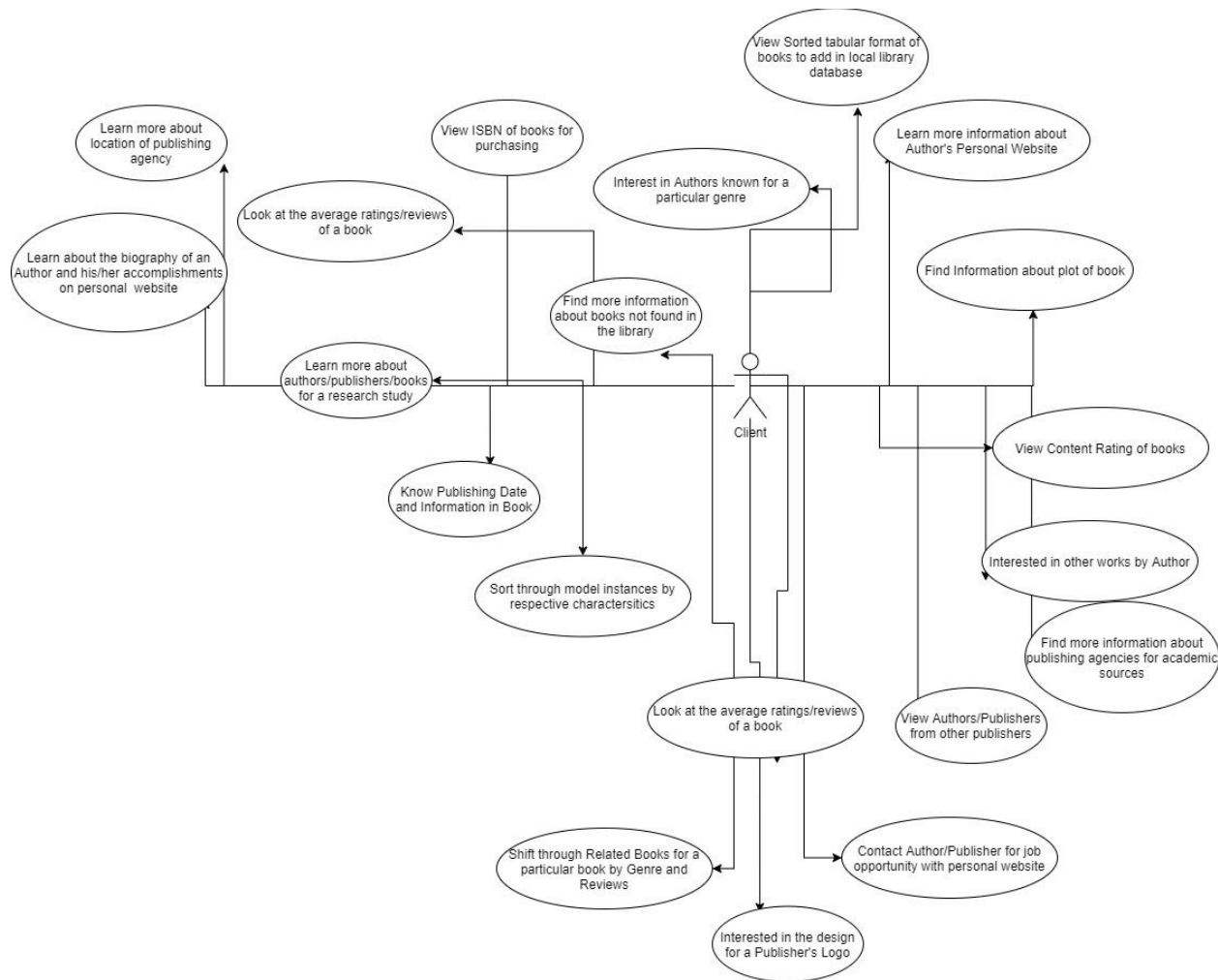


Figure 1. Use Case Diagram

2.1.2 User Stories

Table 1. Phase I User Stories

<i>User Story [Team Members]</i>	Description and Assumptions	Estimated Time	Actual Time
<i>As a book reader, I want to be able to find information about the plot of the book I am interested in</i> <i>[Jaelyn Bethea]</i>	The purpose of BookwormDB is to provide information that would be deemed useful for the users. For a book reader, the plot of the book would fall into the useful information	1.5 hrs	1 hr

	category. It was assumed that one of the first things a book reader does when discovering a new book is to read the plot to see if the book interests them.		
<i>As a user, I can easily navigate to each of the model pages so I can find what I'm looking for</i> <i>[Jaelyn Bethea]</i>	It has been shown that easy navigation is one of the most useful website features. It allows users to quickly find the content they are looking for. Easy navigation will also promote more users to use BookwormDB because users of all ages and backgrounds can understand the navigation. It is assumed that the easier the navigation on a website, the more likely a user will return to use BookwormDB again.	3.5 hrs	4.25 hrs
<i>As a parent, I want to see the content rating of a book so I know if it's appropriate for my children.</i> <i>[Carlos Borja]</i>	It is assumed that not every user who interacts with BookwormDB will be a book reader. A parent may want to buy a book for their child (a person who loves to read books) but before buying the book, they want to check the content rating to ensure that a book is child-appropriate.	50 min	48 min
<i>As a book reader, I want to find other books that an author wrote</i> <i>[Jaelyn Bethea]</i>	The purpose of BookwormDB is to provide information that would be deemed useful for the users. For a book reader, it is assumed that other books that an author wrote would fall into the useful information category. If a	2 hrs	3.5 hrs

	reader thoroughly enjoyed the book by an author, they are more likely to discover other books written by the same		
<i>As an author, I want to find publishers with works in my genre so that I can potentially get published by them.</i> <i>[Carlos Borja]</i>	BookwormDB was created to extend to audiences beyond book readers. The motivation behind BookwormDB was to also create useful information for authors and publishers as well.	65 min	2 hrs
<i>As a book reader, I want to easily navigate to many authors of books I have read so I can better understand what I am reading.</i> <i>[Carlos Borja]</i>	The purpose of BookwormDB is to provide information that would be deemed useful for the users. Easy navigation is a tool that helps achieve this goal. It is assumed that the more concise and more clear the navigation is layed out, the quicker a book reader can find the information that they are looking for.	35 min	20 min

Table 2. Phase II User Stories

User Story [Team Members]	Description and Assumptions	Estimated Time	Actual Time
<i>As a developer, I want the website to load content from a database so I can generate content dynamically.</i> <i>[Jaelyn Bethea, Carlos Borja]</i>	Using a database allows us to only request the minimum amount of data needed to have a functional webpage resulting in faster load times. No assumptions were made for this user story.	1.5 hrs	1 hrs
<i>As a website user, I want pages with lots of content to use pagination so I can easily navigate through the webpage's content.</i> <i>[Jaelyn Bethea, Carlos Borja]</i>	Pagination speeds up the loading times for our website. It also limits the content a user can see. In websites with a large amount of content, the user could be easily overwhelmed. We've assumed that the website user has visited pages where pagination is used.	3.5 hrs	4.25 hrs
<i>As a user with multiple devices, I want the website to be thoroughly tested for correct functionality so all my devices will be able to use all website features.</i> <i>[Jaelyn Bethea, Carlos Borja]</i>	Users will visit our website on a wide range of devices, so we needed to ensure that the functionality would work regardless of platform. For this phase, we assumed that the website would only be accessed through a desktop browser.	1.5 hrs	4.0 hrs
<i>As a book cover designer, I want to see book covers for inspiration for future designs.</i> <i>[Carlos Borja]</i>	When accessing a book's page, the user will be able to see the book's cover. This is a picture provided by GoogleBooks API with a thumbnail tag in their JSON response.	0.5 hrs	1 hr
<i>As a reviewer, I want to find related publishers</i>	When accessing a publisher's page, the user	1 hrs	2 hrs

<p><i>associated with my genre of interest so that I can contact authors and publishers with their bio page to learn more about future publications.</i></p> <p><i>[Jaelyn Bethea]</i></p>	<p>will be able to view the authors and books that they have published. This concept will also be applied to the authors' and books' pages. This is done by having two arrays of objects in each of the documents. Each of the arrays contain information about the other two collections that correspond to the current document. For example, a publisher document will have an array of authors that they have published and an array of books they have published. Each element in the two arrays will have the author's name and ID of the author in the author's collection. The same concept applies for the books array.</p>		
<p><i>As a designer, I want to be able to organize the database in a straightforward format that my team members and other developers can understand.</i></p> <p><i>[Mrugank Parab/ Santhosh Saravanan]</i></p>	<p>We organized the database in a simple format that allows for the front end to easily parse information about each model. Each collection has documents that have a consistent set of data entries across all documents. This is done by ensuring that each data entry is either filled with information or empty (for now). When the front end wants to get information about each of the documents, they will be provided a JSON file to easily parse the data for the website.</p>	1 hrs	2 hrs
<p><i>As a developer, I want to</i></p>	<p>MongoDB provides</p>	1 hrs	3 hrs

<p><i>be able to write effective unit tests to test back-end CRUD operations so my data is always usable by the front-end.</i> <i>[Mrugank Parab/ Santhosh Saravanan]</i></p>	<p>different functions that correspond to each of the CRUD operations. This is done with the Python unittest library. When MongoDB gets a create request, it will rarely raise an exception, but to account for this case, the creation function returns a result class that has a <code>inserted_count</code> attribute which corresponds to how many documents were inserted. If this number is the number it's supposed to be, that means the creation passed. The same concept applies to deletion (where we read the <code>deleted_count</code>) and updating (where we read the <code>modified_count</code>). For reading, the <code>find_one</code> function takes in arguments to find the key's value and returns if the document exists</p>		
--	---	--	--

Table 3. Phase III User Stories

User Story [Team Members]	Description and Assumptions	Estimated Time	Actual Time
<i>As a book reader, I want the ability to see similar works of books that I have already read</i> [Jaelyn Bethea]	It was assumed that a book reader would want to know more about a book than just the author and plot of the book. Another informative and useful attribute to most book readers are similar works. It is also assumed that a reader if a reader enjoys a specific book, they are more inclined to enjoy one that is fairly similar. To incorporate these assumptions, similar books was an attribute our team included on our book-instance pages. If a reader see's a book they might be interested in, they can search through our website for more information.	0.5 hr	1 hr
<i>As a book reader, I want the ability to search for a specific book so that I can find out more information about the book (plot, author, etc)</i> [Jaelyn Bethea, Carlos Borja]	It was assumed that typically book readers will not want to spend time browsing page by page for their book of interest. Instead, they would want a fast and easy way to search if a book they are interested in exists within our website. This explains the purpose of the search bar. A book reader is able to simply type in what book they are looking for, hit search, and see what results pop up. This will save our user time and will add to a friendly user experience.	3.5 hrs	4.25 hrs

<p><i>As an avid reader of fictional books, I want the ability to filter out the nonfictional books so I can focus on browsing fictional books I might be interested in</i></p> <p><i>[Jaelyn Bethea]</i></p>	<p>Not only is the purpose of Bookworm DB to provide relevant information for book users, authors, etc, but it is to provide that information in an efficient and user friendly way. If a user is looking for a specific genre of books, it is assumed that the user does not have the time nor wants to spend time looking page after page for the genre they want. To make sure this does not happen, we implemented filter functionality so that a user can more easily find what they are looking for.</p>	<p>3.5 hrs</p>	<p>4 hrs</p>
<p><i>As a book reviewer, I would want to see other books and authors who worked in conjunction with a publisher.</i></p> <p><i>[Santhosh Saravanan]</i></p>	<p>Readers are generally interested in other sources of literature which are similar to the genres they are accustomed to. If a user is looking for similar works, they likely wouldn't like to use the search feature to find relevant books by sorting. They can simply use the title of a recommended book and find it more easily with the implemented filter functionality of the search interface.</p>	<p>10 hrs</p>	<p>4 hrs</p>
<p><i>As a book reader, I want to be able to click on a link on the book's page such that I can find out more about its genre.</i></p> <p><i>[Mrugank Parab/ Santhosh Saravanan]</i></p>	<p>Readers are generally interested in genres which cater to a whole new world of books waiting to be discovered. Knowing the general genres/themes of what a reader is into may help them find other relevant works under the</p>	<p>6hrs</p>	<p>4hrs</p>

	same genre which pique their interest.		
<i>As a developer I want to be able to give filters to unit tests so that I can refine my searches and deletions. [Mrugank Parab]</i>	Adding filters to the unit tests allows for effective debugging and creating of test suites to check for more expected outcomes and conditions.	1 hr	2hrs

2.2 Design

2.2.1 Front-End

The front-end uses Bootstrap for styling and interactive features like popovers. Using Bootstrap as our styling framework allows our pages to remain uniform in colors, fonts, and spacings. Additionally, using Bootstrap's built-in classes, we can reduce the UI development time drastically. In addition to keeping UI consistent, we also had to find a way to facilitate content generation. Jinja templates served this purpose and were used for our model and instance pages. The use of templates helped keep the UI structure uniform across all pages of our website.

Now, there are two queries that can be made through the front-end: one for a type of model page, and one for an instance of one of our model types. The query for a model page is a collection of instance queries since we show a user-defined number of instances in our model pages. By default, the pagination for the model pages shows 10 instances. However, the user has the ability to select how many instances the front-end should show. The sequence diagram in Figure 2 shows how data is served to the front-end after a user has made a request for a type of instance.

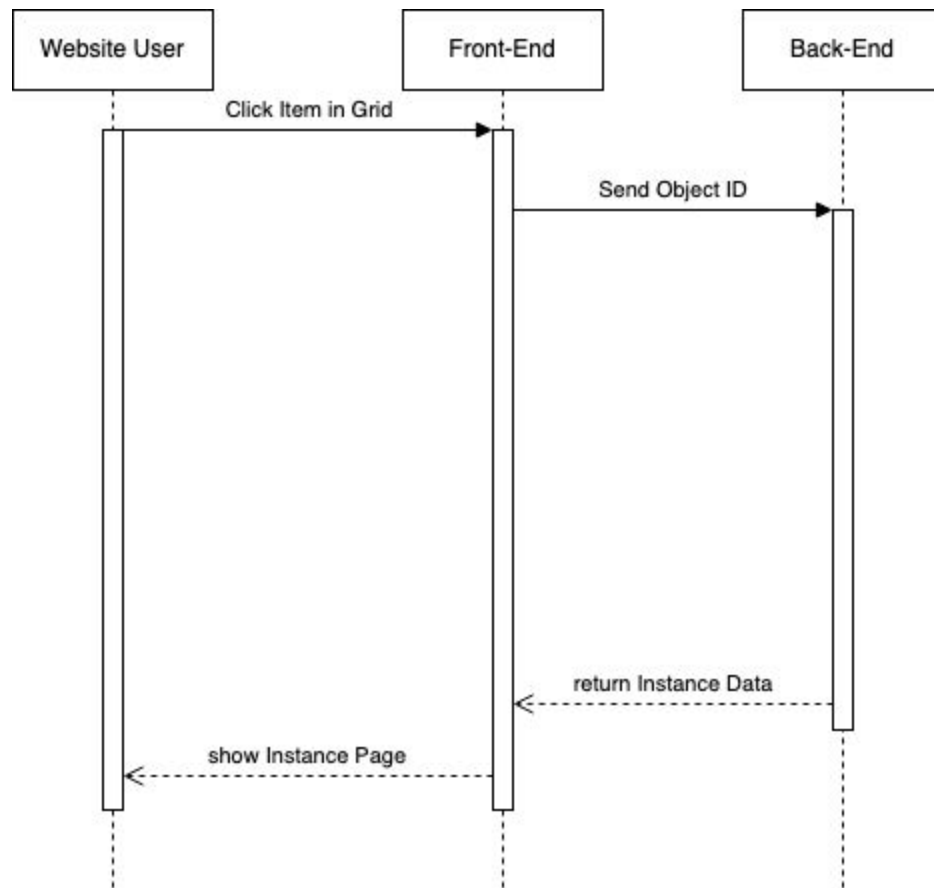


Figure 2. Sequence diagram showing request for model instance

The main difference between each model instance request is the database from which the object is obtained. Otherwise, the sequence remains the same among all three types of model instances.

2.2.2 Back-End

The database that we decided to go with is MongoDB. Our database has 3 collections, one for each of the following: Authors, Publishers, and Books. Each of the collections have documents that have attributes that correspond to the ones shown in Figure 3. All of the documents have a consistent number of attributes across each of their respective collections. Each of the collections's documents are linked to each other in other collections through an array of dictionaries. For example, each author has two arrays for books and publishers. The books array consists of dictionaries that have the name of a book made by the author and the ID associated with that book in the Books collection. The same model applies for the publisher array, except

it's the name of the publisher and the ID associated with that publisher in the Publisher collection. The same is done for the books documents (an array for Authors and an array for Publishers) and the publisher documents (an array for Books and an array for Authors). In regards to images, previews, and embeds, we obtain that information from GoogleBooks and use the link associated with the respective information that's hosted by GoogleBooks.

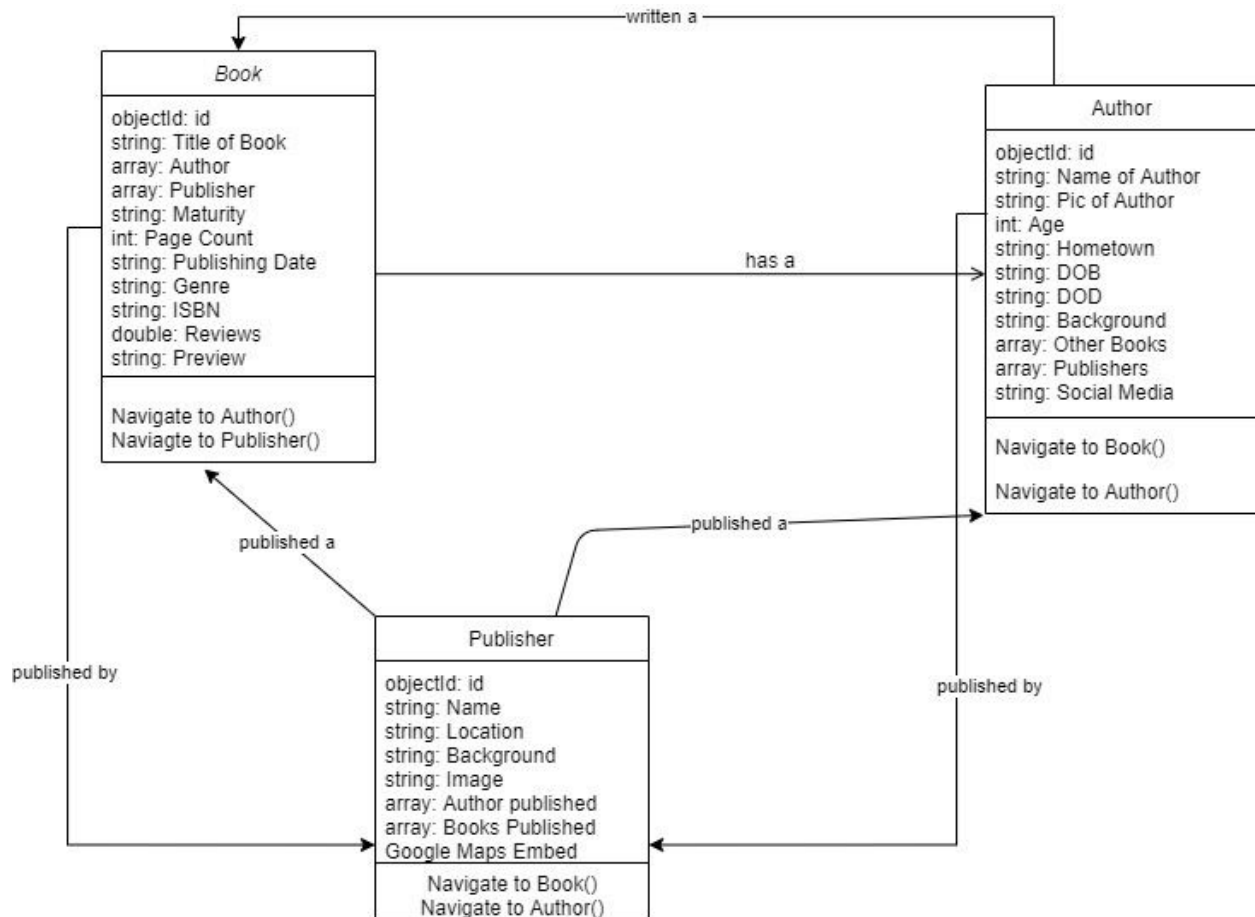


Figure 3. Class diagram for model instance

2.2.3 Information Hiding

For the front-end, information hiding was used to prevent the user from sending an invalid input to our program. Initially, API requests used GET requests to implement model page navigation. We selected this request method to facilitate testing during Phase II. However, we anticipated

that having our request information visible to the user could lead to them attempting to use invalid values for the request. As such, we changed the method used in requests to POST. The values we send are still visible if the HTML source is read; however, it is much less likely that a user would access this information.

For the back-end, information hiding was utilized through the hiding of object IDs of each instance to prevent the user from being able to click on different object IDs to go into other instances without permission [eg. clicking on a publishing object ID and being taken to the underlying structure of the book instance]. We mainly also hid the propagation of the top authors that we used to prevent users from being able to tamper with setting and changing the respective object IDs and other attributes of the publisher, author, and book instance. We designed a Facade pattern framework to hide the complexities of our instances from the client and give an easier mock-up implementation for clients to interact with the website and encapsulate the private methods by reducing their scope in our design.

2.2.4 Design Pattern

One design pattern our project incorporates is the Model-View-Controller framework. The Model component consists of our models: books, authors, and publishers. The View component consists of all the HTML files that displays all the model instances, website homepage, etc. Lastly, the app.py is the Controller component. The user interacts with the Controller to manipulate one of the Model components, which in turn updates the View component that displays what the user sees. One of the biggest advantages of using MVC was that modification did not affect our entire model. For example, during each phase of the project our team was altering and changing the user interface. Since our Model component did not rely on our View component, the modifications our team made had no impact on that component. Another advantage of the MVC pattern was that it kept our code fairly organized and aided in a faster development process. One team member was able to focus on the Controller component, while another focused on the View component. The UML diagram below in Figure 4 provides a visual of how the MVC architecture was implemented in our project.

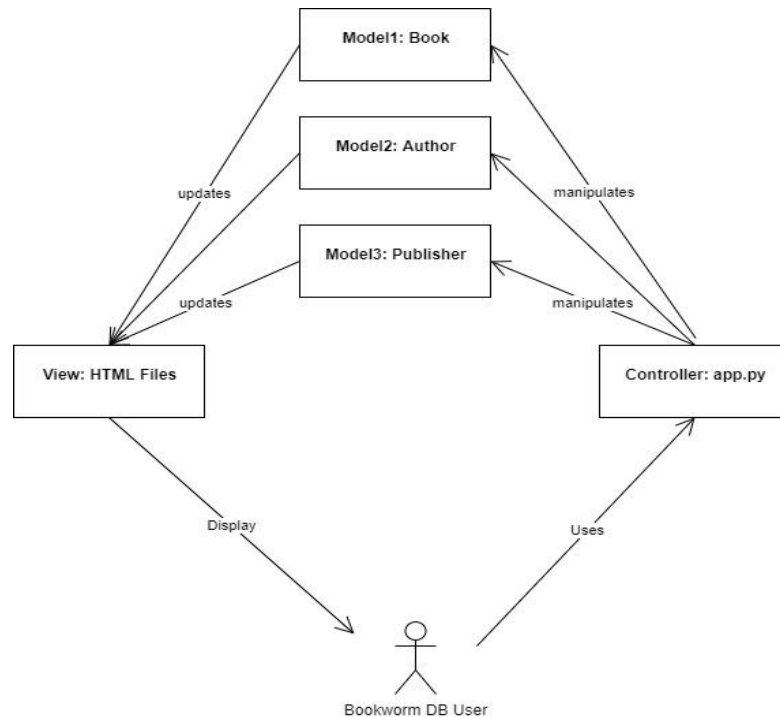


Figure 4. UML MVC Diagram

There were two major drawbacks to using the MVC pattern. First, the complexity of our application quickly rose since each component needed to be modular. The increased modularity allowed for quick individual changes but increased latency for passing information from one component to another. Second, MVC encourages asynchronous work which sounded great in theory. Unfortunately, we found ourselves dedicating significant time to getting our components to work together which ultimately slowed down our development speed.

For the backend, we decided to go with the facade pattern for the code that filled up our database. This pattern allows us to make an interface that is being implemented by our model classes. We also have a controller class that calls the methods in the model classes. Figure 5 is a UML diagram showing our implementation of this model.

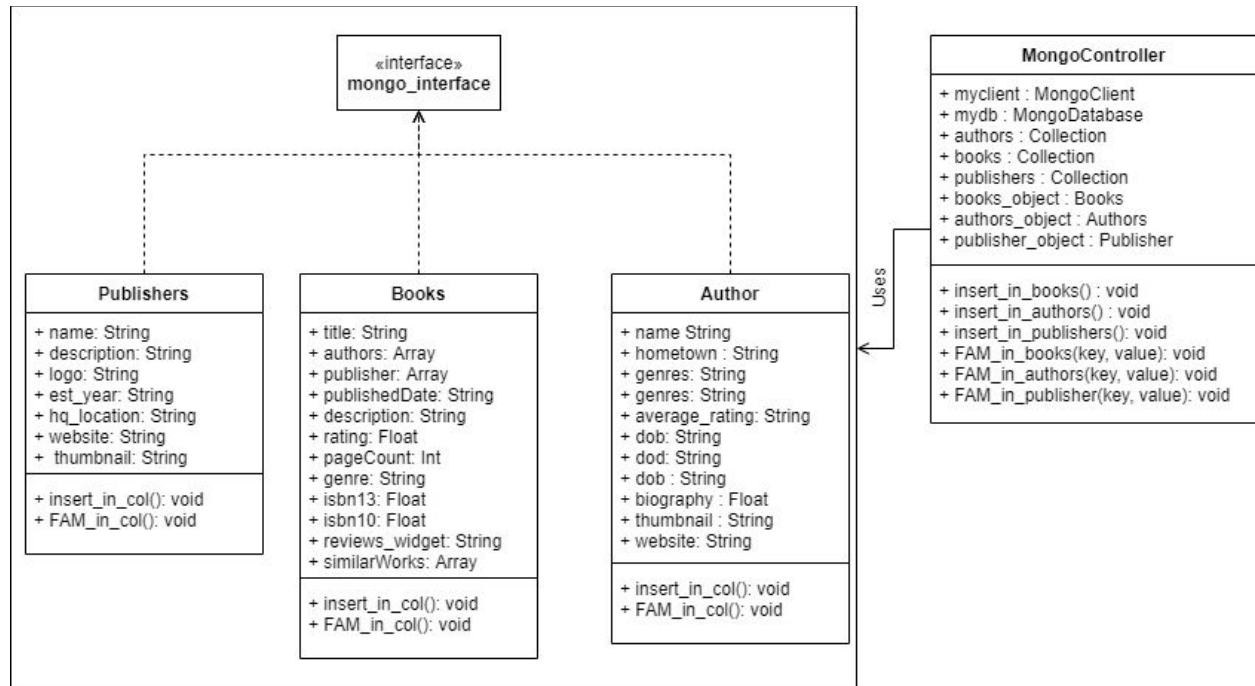


Figure 5. UML Facade Pattern Diagram

This pattern allows us to hide dirty looking code that's used by the `find_and_modify` method that is given by Pymongo. This method takes in a lot of flags and parameters that took up three lines. The idea behind this was that all the models needed to be inserted and updated in the respective collections in our database. So we made an interface that defines the insert and `find_and_modify` method. These two methods are then being implemented in each of the three classes with their own respective collections. So our main file to fill up the database, `reformed_books.py` uses the `MongoController` class to call the methods to insert or modify the documents in each of the respective collections.


```

if(foundIndex == -1):
    publishers.find_and_modify(query={'name': str(author.books[i].publisher)},
                              update={"$push": {'authors': authorEntry}}, upsert=False,
                              full_response=True)
    publishers.find_and_modify(query={'name': str(author.books[i].publisher)},
                              update={"$push": {'books': bookEntry}}, upsert=False,
                              full_response=True)
else:
    publishers.find_and_modify(query={'name': str(publishers_name_list[foundIndex])},
                              update={"$push": {'authors': authorEntry}}, upsert=False,
                              full_response=True)
    publishers.find_and_modify(query={'name': str(publishers_name_list[foundIndex])},
                              update={"$push": {'books': bookEntry}}, upsert=False,
                              full_response=True)
authors.find_and_modify(query={'name': str(author.name)},
                        update={"$push": {'books': bookEntry}}, upsert=False,
                        full_response=True)
authors.find_and_modify(query={'name': str(author.name)},
                        update={"$push": {'publishers': publisherEntry}}, upsert=False,
                        full_response=True)

```

Figure 6. UML Facade Pattern Diagram

Figure 6 shows us code before this design pattern. We realized that these lines are lines that are used by all of our models. The interface is shown in Figure 7 that is being implemented by all the models. Figure 8 shows the controller that controls the functions for find and modify and insert.

```

1 class mongo_interface:
2     #interface class defining insert and find_and_modify methods
3     def insert_in_col(self, collection):
4         pass
5
6     def FAM_in_col(self, collection, key, value):
7         pass

```

Figure 7. Facade Pattern Interface

```

import pymongo

class MongoController:
    myclient = ""
    mydb = ""
    authors = ""
    books = ""
    publishers = ""

    books_object = ""
    authors_object = ""
    publisher_object = ""

    def __init__(self, booksObject, authorsObject, publisherObject):
        myclient = pymongo.MongoClient("mongodb+srv://Santhosh_Saravanan:Divya2004@cluster0.rh1w0.mongodb.net/test")

        self.mydb = myclient["book_worm_database"]
        self.authors = self.mydb["Golden_Authors_Mrugank_copy"]
        self.books = self.mydb["Golden_Books"]
        self.publishers = self.mydb["Golden_Publishers"]
        self.books_object = booksObject
        self.authors_object = authorsObject
        self.publisher_object = publisherObject

    def insert_in_books(self):
        self.books.insert_in_col(self.books)

    def insert_in_authors(self):
        self.authors.insert_in_col(self.authors)

    def insert_in_publishers(self):
        self.publishers.insert_in_col(self.publishers)

    #fam is find and modify
    def FAM_in_books(self, key, value):
        self.books_object.FAM_in_col(self.books, key, value)

    def FAM_in_authors(self, key, value):
        self.authors_object.FAM_in_col(self.authors, key, value)

    def FAM_in_publishers(self, key, value):
        self.publisher_object.FAM_in_col(self.publishers, key, value)

```

Figure 8. Facade Pattern Controller

Some advantages of using this model is it makes our code more organized by simply calling a one-line function. This is because we have functions in the interface that define these functions for each of the models. There aren't really any serious disadvantages to using this pattern because it makes our code more organized, but one disadvantage can be that if we were to make another model, we would have to define the methods that's in the interface in each of the new models that we create.

2.2.5 Refactoring

Phase IV was where our team did our large scale refactoring. One of the refactoring our team did was renaming variables. While this was a rather simple refactoring method, it had a great impact on readability. When completing each project phase, more focus was placed on making sure the implementation was correct rather than making sure the variable names made sense. As a result, looking over some of the code after each phase was a bit confusing (even for the person who wrote the code). The situation was even worse when a member would ask for help, but extra time

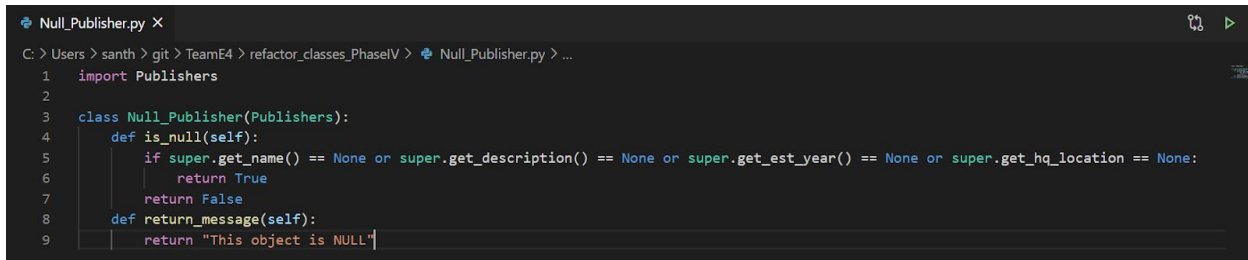
had to be taken out to explain what each variable meant before the code logic could even be explained. Changing the variable names not only improved readability but aided in a better development process.

In addition to poorly named variables, our team had a lot of duplicated code. Initially, it was often easier and faster to copy and paste code that we knew worked when it was needed. However, repeatedly copying and pasting made our code longer and made us more prone to mistakes. To reduce the chance for mistakes, we extracted a method from the repeated code lines. The extraction shortened the length of our code, and it helped increase readability. One method that we extracted was a `truncated_<model>(instance)` method. It creates a shortened version of our models which are used to speed up loading times. The extraction reduced our lines of code by ~27%. The code snippet in Figure 9 shows the definition for one of the truncated methods.

```
# Sample method
def truncated_book(book):
    temp = dict()
    temp['_id'] = book['_id']
    temp['title'] = book['title']
    temp['authors'] = book['authors']
    temp['genre'] = book['genre']
    temp['rating'] = book['rating']
    temp['thumbnail_url'] = book['thumbnail']
    return temp
```

Figure 9. Extracted truncated method for book instances

For the Back-end team, to combat our confusing code, one particular mode of refactoring we used was the introduction of the null object. Given that Python doesn't have the concept of null references, we were essentially forced to write null checks for checking attributes of our instances. This led to a lot of unhandled exceptions and we found it wasteful to create multiple object instances with null parameters, but rather decided to create a null object encapsulation certain null behavior characteristics expected in a publisher, book, or author object. Figure 10 shows the implementation of the pattern for a publisher object. The use of this refactoring exponentially reduced the number of if-else conditions, duplicated code, and dead code.



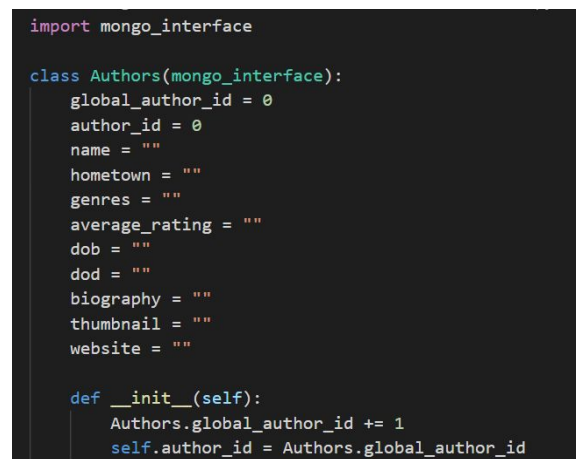
```

1 import Publishers
2
3 class Null_Publisher(Publishers):
4     def is_null(self):
5         if super.get_name() == None or super.get_description() == None or super.get_est_year() == None or super.get_hq_location == None:
6             return True
7         return False
8     def return_message(self):
9         return "This object is NULL"

```

Figure 10. Encapsulation of Expected Null Behavior for Publisher Instance

To reduce dependency between classes, we used the move refactoring method. For instance, our reformed_authors class had a class definition called Authors. After looking at the logs, it turned out that most of the data received and handled by the API requests seemed to target another class called Authors which was responsible for a lot of the tasks involving the back-end support for the authors instance page. A class definition there and migrating the method calls to this class would make a lot more sense, and would significantly reduce method calls from our reformed_authors class to our Authors class. We could then reduce the scope of the reformed-authors to only focus on hiding the main list of authors from the user, deleting defective authors based on inaccurate or inadequate information, and streamlining the navigation amongst the different models.



```

import mongo_interface

class Authors(mongo_interface):
    global_author_id = 0
    author_id = 0
    name = ""
    hometown = ""
    genres = ""
    average_rating = ""
    dob = ""
    dod = ""
    biography = ""
    thumbnail = ""
    website = ""

    def __init__(self):
        Authors.global_author_id += 1
        self.author_id = Authors.global_author_id

```

Figure 11. Move method Refactoring for Authors Object

2.3 Models

The three models our team created were: Books, Authors, and Publishers. The connectedness between our models are seen in the fact that each book was written by an author(s) and published

by some company. Each author has written a book and has used a publishing company to publish that book. Lastly, publishers contain information about the individual authors and their books in an attribute section. The relationship between a publisher and a book is many to one as a publisher is responsible for publishing many books written by different authors. Usually these books feature different genres and many authors are known for certain genres of books they write. Each model page uses pagination, so a user can see how the total number of pages a model has and easily navigate to a page of interest.

On each page of our Book models pages, 10 entries of books are listed along with a picture of the book, the book title, genre, and rating. Each book entry is clickable and will navigate a user to the book-instance page. On a book-instance page, the book's title, author, publisher, rating, plot, content rating, word count, etc is displayed. Further down the instance page are reviews from those who read the book as well as a short preview/excerpt from the book. The data for our Book model and instance pages were sourced from Google Books API and Goodreads API.

Our Author model pages consist of 10 entries of authors per page. Each author entry is accompanied with a picture of the author, author's name, and the author's hometown. From the start of Phase III, the back-end team is thinking of adding in a link to the author's website and a content rating related to the author's book. The data for our Author model pages were mainly sourced from the Wikipedia API, Google Books API, and the Goodreads API. Pagination is also apparent as only a certain number of authors are being displayed with the ability to get some attributes while hovering over a particular author's image. Also, given the century in which the author is known for, we might be able to post a social media link to his or her social media account. We understand that certain authors may not have had access to the technological age of the 2000s and may not have a social media link, so we might just stick with a link to their personal website, if we are able to find any through an existing API or web-scraping. We will also post a list of clickable book title objects from a particular author under the other books written by the author section. Each author name will also feature a short description of the author involving his/her name and the hometown and a short description about their accomplishments.

Lastly, our Publisher model page consists of 10 entries of publishers per page. Each publisher entry is accompanied with the publishing company's logo, company name, and company

location. Each publisher entry is clickable and will navigate a user to a publisher-instance page. The publisher-instance page consists of the attributes that were on the model page with the addition of company background information and a Google maps embed. The data for our Publisher model pages and instances were primarily sourced from the Google Books API and the Goodreads API, primarily for the name, description, list of authors, and list of books. For the headquarters instance and logos, a Selenium API and the Google Search API were crucial in fetching the correct information related to these sub-fields.

2.4 Testing

2.4.1 Phase II

2.4.1.1 Front-End

The front-end team used Selenium to test the GUI. For each model page, we did testing centered on pagination functionality. One test involved ensuring that the number of entries showing on the first page of our model pages was the correct number of entries. For example, when clicking on the Books model page, it should show the first 10 book entries (1-10). Then when clicking on page two, it should show the next 10 books entries (11 - 20). However, this test only sufficed from page 0 to n-1 where n is the number pages. Our last page has the option of displaying 10 entries or it could display less. Regardless, our HTML code will show it has 10 entries even when it is not. In addition to testing the entries text, we also created a test to ensure that when navigating to the next page or to the previous page, the user ended up at the right page number. This test also ensures pagination controls remain functional. The next and previous page buttons were tested from the first page, last page, and from an arbitrary page. Our test cases can be found in the testing folder on our Github repository. In the future, we plan to create unit tests for our splash page, and searches, sorts, and filters across all models.

2.4.1.2 Back-End

The backend team tested the basic CRUD operations such as creating a valid document, getting a document's information, updating a document, and deleting a valid document. Initially we had trouble with inserting a document because we wanted to insert a document that only has all the information about a book/author/publisher. So we fixed this by taking input from the user about

the different values to put into an individual document and testing whether the document was inserted successfully. The read test takes input from the user to search the database based on the title of the document. If the document exists, the test passes. The update test takes input from the user to search the collection to find a document. The test also takes input for which key to update and fails if the collection doesn't have the document, or the key doesn't exist. The deletion test takes input from the user to find the document to delete. The test then deletes the document and fails if the document wasn't deleted or if the document doesn't exist. Like front-end tests, our testing code is located inside the testing folder.

2.4.2 Phase III Testing

Information for phase III testing will be added during development.

2.4.2.1 Front-End

The front-end team used Selenium to test the GUI. More specifically we created a test that focused on filtering and pagination. Because filtering limited the results we show to users, our team had to modify the pagination code or else pagination would show negative page numbers. Now, when the number of page results is less than or equal to 7, the pagination will just display all pages instead of the usual display that involved the first 4 pages, then ellipses, then the last 4 pages. Testing consisted of testing the number of entries, the next page button, and the previous page button. In addition to those forms of testing, we also tested for when a filter results only produced 1 page. In this case, the next and previous page buttons should be disabled and the only page number pagination should show is page 1. The testing for the filtering can be found in the testing folder of the filterui branch on our team's GitHub.

2.4.2.2 Back-End

Concerning unit tests for the back end, we decided to refine the tests more by allowing the tester to add more filters. When using a test, we allow the tester to give it extra filters for the CRUD operations. This is especially important for the search and filter feature on our website because when the user wants to search for something, we allow the user to use filters for a more refined search. The reason we did this for the read and delete operations is because we wanted to make

sure that the user can find something more designed to their specifications. The reason we did this for deletion is to make sure that we as developers can delete documents that are more specific to our filters. Essentially when MongoDB is given a dictionary for the find() function, it will return all of the documents that match that key and value. So to have a more refined read and delete, we allow the user to give the test more keys and values. These keys and values get added to a dictionary and sent to the delete or read function.

3.0 TOOLS AND FRAMEWORKS

For web applications, there is no shortage of tools and frameworks to develop a functional and reliable application. For our project, we decided to use Python and Flask to serve our website.

For the back-end, we used a NoSQL document-oriented database program called MongoDB which uses a JSON-like structure to format documents. This helped update records and create records quite intuitively, also giving us search queries to locate certain entries by their id or other attribute label. We also used Python as our main programming language because Python was quite intuitive to use in conjunction with fetching data from our database. For populating the database, a majority of our APIs had very good Python API wrappers, which made it quite easy to make a chain of method calls with a series of put, post, and update REST API commands. All of these commands can be executed by a series of easy-to-use commands from the user interface and the user doesn't have to know the underlying call flow graph within an internal API's method calls. With respect to the IDE environment, Pycharm has remained an excellent choice in importing several packages through our APIs even though we figured it wouldn't be recognized by the interpreter. With debugging logs and the feature of adding breakpoints, we were able to debug a lot of our issues and it was exciting to see our method calls affect the database in real-time.

For revamping our models due to lack of information about a good majority of authors in Phase III, we used Beautiful Soup and Pandas for use in gaining more information about popular authors in our database. The Google Books API proved to be helpful when getting information for a certain book. This is because it allows us to search a book by various tags such as publisher, book title, isbn, authors, etc. In addition, we could also add various filters for example, finding books that have an ebook link, paid, or free. The GoodReads API was helpful in being able to

find Authors and information associated with that author. The GoodReads API documentation showed us a way to use a GET request on a url that returns an xml file when passed in an author's name. This returned an ID which allowed us to get information about that author like a biography, date of birth, etc. The Wikipedia API allowed us to get a wikipedia page when provided a search term. This API also allowed us to get the summary section of the page which in our database is the biography or the description for the model. This API also allowed us to get all the HTML elements of the page and parse them using BeautifulSoup. BeautifulSoup is a Python web-scraping library. This allowed us to get information on the table that's almost on every wikipedia page to get information that we couldn't find using the other APIs.

4.0 CONCLUSION

This report serves to outline the details of our team's deployed web app, BookwormDB. Through the development of BookwormDB, our team encountered struggles, gained knowledge, and learned what process worked and what did not.

4.1 Things We Learned

4.1.1 Phase I

The front-end team was able to learn more about Bootstrap 4 and really understand the inner workings of their HTML classes and CSS stylesheets. They were also able to learn about UI design. In specific, they learned about the effective use of whitespace to provide a user interface that is great for any device.

The backend team had the opportunity to explore different APIs and how to call information by using them. They also learned about using JSON files to parse data very quickly and easily. We also learned that we need to make our time management skills more efficient. We learned that although we are communicating well, we tend to be rather vague on what to work on when the time's come. However, when being specific on the things to work on, we are able to efficiently and quickly finish tasks without having doubts or misunderstandings.

4.1.2 Phase II

The front-end team was able to learn a lot about pagination. This was the first time anyone on the front-end used pagination. Front-end was able to gain a deeper understanding surrounding the design and functionality of pagination and how to use it effectively within our website. Phase II also highlighted the importance and complexities of the user-interface. When it comes to the interface, we realized that it was important to consider design layout from a variety of different perspectives. What may appear visual appealing from the iPad layout, may look horrendous and cluttered on the iPhone layout. Front-end learned how to keep these different aspects in mind when designing the user interface.

The back-end team had the opportunity to weigh the costs and benefits of using certain APIs and important web-scraping techniques to gather information not available to the use of the APIs. We also learned a lot of mongoDB, concerning python, especially when learning to create collections, update certain entries, delete certain records, or know the different types of data that can be stored in a particular record. We also learned how to unit test our code effectively and how to communicate efficiently with the front-end team on the structure of how certain data parameters should be stored in a database, where it can be a list, dictionary, tuple, set, etc. We were able to fully utilize our time management skills in delivering our product, but there are still many obstacles to overcome in terms of communication as we tend to be rather vague about certain requirements that need to be met and not updating our Github regularly. We were able to combat these problems and use the issue tracker and commit history to keep track of our latest updates in code. However, despite these obstacles, we were still able to finish our tasks without running into a lot of issues and misunderstandings.

4.1.3 Phase III

The front-end team learned a lot about implementing sorting, filtering, and searching interface and functionality. This was the first time anyone on our team attempted to create such functionality. Within our process of figuring out how to implement these functions, our eyes were opened to the importance of designing for the user. More thought was put into how to balance our team objectives (sorting, filtering, and searching) and what works best for user

interaction. Our coding was no longer solely focused on meeting the requirements, but it now included how to meet those requirements with the thought of our users in mind.

The back-end team learned immensely about utilizing Python scripts in conjunction with MongoDB and exploring new commands for organizing and filtering documents in the database. We learned immensely about testing frameworks and were able to fix an immense amount of the troubleshooting problems due to helpful forum posts on the APIs and on StackOverFlow. This was also the first time we learned about creating a script and utilizing the API method calls from an updated version of the GoodReads API and how to use OAuth to access methods with additional functionalities. A lot of experimentation had to be done to fetch the correct URLs and to understand how the respective book/author/publisher object was created and the most efficient ways to extract information from them, convert it to utf-8 encoding, and store it in the database for get calls from the front-end team. More thought was put into how to efficiently sort and search for the back-end team to implement these features, but the challenging part was to efficiently think about how our audience would benefit from searching through our website.

4.1.4 Phase IV

The front-end team learned about the importance and impact of refactoring. Even the simplest refactoring methods like renaming variables increased the readability of our code. Through refactoring, we reduced the lines of code for one file by 27%. We also learned more about the MVC framework. While the tutorial we had in class somewhat covered this framework, we were able to learn more from actually implementing it within our project.

The back-end team learned immensely about how to write effective test cases and modularize our Python scripts to the refactoring and design patterns. We also learned quite a bit simply from rewatching the lecture videos about the common design/refactoring patterns and making direct applications of that material in an effort to fix our spaghetti code :). We used Piazza, StackOverflow, and other sources of documentation to learn more about practical examples of these refactorings and figuring out how we can make a similar impact with our current codebase. This was also the first time we were able to make real-life applications of our software testing

class to figure out when and where API calls were being made, how objects were instantiated, and whether our refactored code was passing our test cases from Phase III. Lastly, We were able to effectively manage our time and resources to fix most of our errors from Phase III, refactor our main design classes appropriately, and implement a new design pattern for our core classes to increase modularization and reduce latency speed.

Due to the design patterns, we learned that planning out the patterns on paper proved to be helpful. We understood why design patterns are so helpful in making sure that we have clean code that is very organized and well maintained. In addition, after learning about more design patterns online, we found out that there were many ways in achieving our goal of making sure that the database has complete documents by using these design patterns. After reading through the code for the second time for this phase, we realized that it was very messy and having to modify it would be very difficult. So using design patterns during this implementation would prove to be extremely helpful.

4.2 Things We Struggled With

4.2.1 Phase I

During our work in Phase 1 of this project, our backend team struggled with finding relevant information to use for the publishers. We learned that when it comes to publishers, the only information that the APIs we have right now gave us were the books that each publisher has published. However, we decided that in order to solve one of the user stories from an author's perspective, we wanted more information rather than just books. We solved this by finding a Wikipedia API that gives a lot more information on authors and publishers. We will still have to explore this in order to use it to its full potential. We have done some exploration on the Wikipedia API and we are able to fetch the summary of our respective publishers in string format, but additional work may lie in getting the necessary headers from the Wikipedia API.

Another struggle our team faced was time management. While we were able to successfully submit the requirements for Phase1, there was a bit of a time crunch. This was mainly due to the

fact that we underestimated the amount of time needed to complete Phase 1 of this project. Coinciding with lack of proper time management, our team consists of members who have never interacted (besides the lab tutorials) with some of the technologies like Flask, AWS, and Bootstrap. Because of this, for some members, it took some time to get comfortable with the environment and realize what certain errors meant or what the proper syntax should like.

In addition, our team struggled with picking the appropriate models to meet the team requirements of having a certain amount of instances. Prior to planning for these instances, we were quite lost in coming up with potential ideas to showcase in our website. We then realized that several ideas for our models involved time scheduling, but we were able to come up with a more practical topic, involving book titles, publishers, and authors. We also had an uphill battle coming up with the proper implementations of the APIs and several permission issues with their use to fetch data.

Another struggle our team faced was communication within a virtual environment. Zoom and Slack meetings are a decent substitute for group collaboration and efficient communication, but it will never really substitute in-person interactions between teammates where certain concepts can be understood faster or a solution can be derived quickly. Also, many members have time-consuming courses and academic organizations and their meeting times clashed a bit with our meetings, so rescheduling these virtual meetings quickly became a hassle.

Lastly, a struggle the front-end team dealt with was the platform for deploying their final website and it was between AWS and Heroku. AWS had a lot of user-friendly documentation and tutorials, but these resources required a lot of previous research and some experience using them to truly understand their purpose. Because of the depth of knowledge needed to fully take advantage of AWS, there were a lot of integration and cloud errors associated with the launch of our website. After evaluating other options, we realized that the deployment process was very straightforward if we used Heroku. This service allowed us to have a testing framework to debug issues and prototype new changes in our static website while the upcoming phases were graded. Now, AWS has better flexibility and scalability versus Heroku. In addition, AWS is more cost effective than Heroku, so we'll have to migrate our website during the next phase.

4.2.2 Phase II

During our work in Phase 2 of this project, our backend team struggled with finding relevant information to use for our models. We quickly realized that the Wikipedia API, Goodreads API, and the Google Books API weren't sufficient in giving us the information required for our models. In order to solve the problem of getting logos, for instance, for our database, Santhosh learned how to use the Selenium API to make a script that would click on the first image to pop-up with a google search on each publisher and was able to extract the image URL from that to be used by the front-end team. We realized that the Goodreads API had a straightforward user interface for fetching basic information about less-known authors, but a plethora of information was found for authors that were better-known. For books, however, Goodreads API and the Google Books API were incredibly useful in filling and exploring this in order to use it to its full potential. We have done some exploration on the Goodreads API and we are able to fetch the key attributes for our books collection and discover other "extra features" to be fetched.

Another struggle the back-end team had was we didn't have a lot of incoming knowledge about MongoDB prior to working on phase 2, besides the tutorial. The tutorial was useful for certain operations to be done on a database, but we required more advanced operations to format our database for the utilization from the front-end. This required more reading on our end and understanding of the documentation and the creation of test scripts to get a feel for how these new operations would impact our database. Also, creating clones of our databases for back-ups definitely took a while because of the sheer size of the data consumed. We are considering significantly reducing our size of the models data in the subsequent phases so that the data can be more manageable and more time could be dedicated to testing and utilization of other features.

In addition, our team struggled with testing with Selenium for front-end. None of the members on front-end had extensive (or any) prior experience with Selenium. Although we were giving the tutorial for testing, it helped provide some footing, but not that much when it came to creating the test without a template to follow. The challenge we faced was figuring out to assign ids and what elements we needed to test. We also couldn't generalize the testing for all of our models.

Another struggle our team faced was pagination. While Bootstrap was helpful in implementing the basic pagination design, the functionality of it was more complicated to execute. For example, our books model had over 1,000 books and we displayed 10 books per page. While our pagination showed the different pages since there were so many pages that a user could only access certain ones. In short, we had to figure out how to truncate the pagination bar and make it more user friendly. The process took a lot of strategic thinking and a lot of trials.

Another struggle the back-end team faced was specifically regarding the population of the authors database with our APIs. With regards to the Goodreads API, there was only information related to famous authors in the 20th and 21st century which was only a small subset of our massive collection of authors. The GoogleBooks API had a lot of incorrect information pertaining to our authors and quickly lost credibility as a source for authors, even though it was excellent for books. After an email with Dylan, we have decided to populate the database with about 200 famous authors and then try to update our publisher and books database accordingly. Making sure that these authors have active social feeds and are well-known in the literary world is necessary for including our sub-fields for each publisher, book, and author instance. We will talk to Dylan during office hours on Monday for more guidance on this present issue.

Lastly, a major struggle our team faced during this phase was communication. While communication within our sub-teams of front-end and back-end were fine, the overall communication between front-end and back-end was lacking in this phase. There were many instances in which the progress of our project in phase II was delayed because of miscommunication or requirements not being detailed enough. For example, some of the information stored in the database did not match the front-end code. Part of this struggle was due to the fact that our team cannot meet face to face and therefore communication needs to be more precise and clear. However, this struggle was also due to the fact that as a team, we failed to properly communicate between our sub-teams. It was not until closer to the deadline date did we begin checking in more frequently with each other and updating each other on each respective process.

4.2.3 Phase III

One thing our team struggled with this phase was implementing the filtering functionality. While our team could figure out how to filter attributes at a time, our team could not figure out how to filter several attributes at once. Our intention was to implement a stacked filtering system, but we could not quite figure the implementation out. Filtering multiple attributes at the same time is an improvement our team is still focused on.

Our back-end team also struggled with picking a certain subset of authors with sufficient information on books and their publishers. A lot of these publishers don't quite exist anymore, simply due to these entities merging with other publishers, or simply going out of business eventually. The implementation of filtering a publisher's status by web scraping Wikipedia pages to check if a company is still active and hasn't been merged into another big publishing entity. We also used a trial-by-error basis on fetching the correct sub-headings on a Wikipedia website, because the API continually fails to find data on certain publishers. We then had to filter out the defective publishers and only fetch books where the publishers were known in the reading community. Even so, there are some books without an established book cover, which I have replaced with a grey G character icon instead, for now.

Another thing that our team struggled with is making sure that the search functionality worked as close to Google as possible. So, when given no filters, it will search the collection for the document's title/name. If filters are given, it will search based on those given filters. The default configuration for the find function is very elementary so we needed to research ways to make sure that the search function worked well. We learned that the find function takes in flags. The flag that we needed to use is the "\$regex" flag. This works the same way as the contains method in python. Essentially when using this flag, if the document has the search string given to it within the document's title/name, it will return the document. In addition, another flag we used is the options tag that gives us the ability to search for documents with case insensitivity. We did this because we don't want the user to worry about capitalizing their searches. We also had trouble deciding on a design for the GUI. Initially we wanted to display results for the search results across all the databases on one page. For example, if the user searched for Penguin, it is a publisher, and also a book. In this case, it would show the results for the book with the title Penguin, the publisher with the title Penguin, and no author is named Penguin so it would be

empty. This would all be shown on one page. But then we decided that this would be too cluttered for the user and we decided to go for an approach where the user would click on tabs corresponding to the collection to view the search results for one specific collection.

In addition to implementing our filter functionality, our front-end team also struggled maintaining pagination with our filter results. When implementing pagination for our model pages, we had multiple pages because each model page had multiple instances. However, when filtering specific attributes, our total amount of pages were significantly smaller, making the pagination code we wrote in Phase II no longer valid. Using the pagination code from Phase II on our filtering results displayed negative page numbers. In order to fix this, our front-end team had to create a different pagination layout for when the total number of pages was less than or equal to 7.

Lastly, our team struggled with GitHub and merging our branches properly. The plan for the front-end during this phase was to work separately in different branches for filtering, searching, and sorting, and then merge the code together once filtering, searching, and sorting were working in their respective branches. However, our team kept running into merging issues. More specifically, the pagination for one of our model pages would not work properly but the remaining model pages (which essentially were coded the same) worked perfectly fine. In the end, we could not figure how to merge the branches so we manually moved the files we needed.

4.2.3 Phase IV

One thing the back-end team struggled with this phase was debugging our refactored code. We never had much use for the debugger on VS Code, because our way of checking for the expected output was to deploy the app and check the layout of the website. Unfortunately, we had to use the debugger to set breakpoints for common Endpoint Exceptions within our back-end system, due to the copy-paste design pattern of our handling of the back-end tasks for the instance pages. Nevertheless, it took us quite a while to look into how to effectively use the debugger for our needs. We were eventually able to increase our development speed especially when creating test objects using our modularized class main methods to check instances.

The back-end team also struggled with rebasing and merging our branches. We kept running into detached HEAD issues when trying to pull our changes and when using the git branch command, we were unable to commit our changes to the branch. Nevertheless, we did have to reclone our local copy of the repository on Github and reupload our changes to the branch. When this error occurred the second time, we decided to revisit our Git tutorials and StackOverflow to figure out that we could abort our git merges, resolve our conflicts, and then do a git pull. We also learned how to revert our changes using the git reset command and view the history of recent commits , but we accidentally did a reset--hard as per a solution and nearly lost everything on our branch. Luckily, we had a local backup of that particular branch and were able to clone it. We lost a lot of development speed concerning our commits of new features into our branch for Phase IV.

In addition to rebasing and merging, our team struggled with implementing design patterns. The set up of the project was not along the lines of object oriented programming in the fact that we did not really have classes. Due to this fact, it was rather difficult implementing a design pattern. We kept trying to mold our code into design patterns and it was rather difficult. What further aided this struggle was the mindset of not wanting to change the structure of the code we have been working for this whole semester. The code we wrote was familiar and trustworthy and it felt like we were changing something that wasn't broken.

Furthermore, refactoring was challenging. It took major effort to determine what refactoring was appropriate for our code base. In particular, method extraction was hard to figure out. We were struggling to decide whether the method should be part of a class or a standalone method. Fortunately, we decided on extracting the duplicate code into standalone methods. While this resulted in three new, separate methods, our code base was significantly improved by their implementation. Ideally, we would have implemented the method in its respective class, but that proved to be a massive challenge given the struggles mentioned previously.

The back end also struggled with finding a good design pattern to use for our code. This is because we mainly used a long script that did the large portion of filling up the database. Initially, it was all just messy code with a lot of variables that represented the Models' data. So

we first started out by making specific classes for our models. In addition, initially finding a good and helpful design pattern was hard to find. But after reading through the powerpoints and online sites, we found out that using the Facade Pattern was perfect with the use of our Model classes. Using the Facade Pattern we were able to easily implement the classes with the Facade which was used by the client (reformed_books.py).

4.3 Things That Worked Well

4.3.1 Phase I

Considering our current environment, our teamwork is working really well. So far, all teammates show up to meetings and communicate issues or concerns effectively. In addition to this, the diverse knowledge and ability to communicate provides a strong support system where each team member helps each other at a moment's notice. Furthermore, the way we decided to split up the work went well. We had two members working on the front-end development while the remaining two members started working on the backend development. The splitting of the groups allowed for our development to come together more quickly and simultaneously work on parts of the project due now and in the future. By doing this, our hope is to save time when moving on to the next phase of the project.

Another thing that worked really well is the Google Books API. We feel like this API is very easy and straightforward to use. In addition to that, we found that using Postman for quickly testing out API calls is very useful as making API calls is simple when using Postman. In addition to Google Books, GitHub was also a component that worked really well. Github allowed us to code separately, while at the same time keeping other members notified of changes to the code. Github made it easy to save and share each stage of the development of our application.

An API that was suggested by our teammate which turned out to be very useful was the Wikipedia API. Previously, we couldn't find a way to retrieve some biographical data about our authors and publishers and were considering the use of web-scraping to obtain this kind of data. After some research, the back-end team found a very interactive and clean Python wrapper

encapsulating the Wikipedia API's methods which proved quite flexible and lightweight to use. A method call gives us the summary of a particular page, but more research will be required to obtain other sub-headings related to publisher instances.

The front-end team really benefited from the Bootstrap framework. There were certain behaviors that were not expected. However, the framework's flexibility allowed Carlos and Jaelyn to quickly identify alternate ways to achieve the same result. For example, our Books page initially used Bootstrap's `.card-columns`. Unfortunately, using this class heavily restricted our layout which affected the responsiveness of the website. Thankfully, the front-end team was able to use the grid system in the framework to implement a better, more responsive layout. The resulting layout was consequently used throughout many of the website's pages. Overall, this change shaved off time from some of the tasks the front-end team had to perform.

Finally, the back-end team really benefited from the online tutorials regarding MongoDB in Python. Learning the differences between the NoSQL and SQL databases and working through some tutorials helped make the process of creating a server instance hosting our database a lot simpler when considering our application as a whole. Santhosh was also able to dive further into advanced tutorials regarding the use of the database and was able to play around with the key and value pairs and how to fetch them from a Python application. This helped give the back-end team a long-term perspective on how to support the dynamic callback mechanism of our database into our website, which will be crucial for the upcoming phases when the data will no longer be hard-coded.

4.3.2 Phase II

One of the things that worked well during this phase was our incorporation of Jinja. Jinja has been extremely useful when it came to looping through the information in our database and getting that information to display nicely in our HTML files. The syntax was fairly easy to figure out and was not too challenging for the lesser experienced members on the team to become more comfortable with it.

In addition, one thing that worked really well was pagination. Outside of functionality (which was the difficult part) implementing the user interface for pagination was fairly straightforward. The Bootstrap documentation was the biggest help in making the user interface implementation run smoothly. The documentation listed the various options for pagination as well as attributes associated with pagination. We decided to have pagination on the top of our page as well as the bottom. Once we figured out how the user interface worked for the top on one of the model pages, the same code was replicated for the pagination for our remaining two model pages.

Another thing that worked really well was the utilization of the GoodReads API. We feel like this API was very easy and straightforward to use after getting help from one of the publishers of the API library on how to look up information of authors on their database. There are also similar works and similar genres mapped to each author, which are certain points of interest to consider when thinking about phase III and adding more attributes to an author object. Also, the GoodReads API can give us more “secret” information if we know the OAuth Key and the author’s numerical id in the database. Previously we had trouble using this feature with the GoodReads API because we weren’t able to use the python GoodReads API to get a numerical ID that was connected to the Author/Book. But we quickly learned by asking some questions on the GoodReads API forum that a way to get the ID connected to the Author/Book was by using a GET request that is on the GoodReads Documentation that the Python GoodReads library fails to implement. This way, we then use the Python GoodReads library to get the author connected to the ID gotten from the GET request and get various information such as birth, death, and biography.

Another thing that worked really well in the back-end section of the website was the Wikipedia API. Even though sometimes, we weren’t able to find certain things on the website as Wikipedia doesn’t have a consistent set of data for each of their articles, we were able to easily grab the necessary information. For example, the Wikipedia API has a search feature that returns a list of related search results. This way we could find if there was an entry in the list that matches the thing we are looking for. In addition, the API had a function that allowed us to get a biography or a summary of the thing we are searching for. This became our “about” or “biography” tag for our publisher and author documents respectively. Lastly, we were able to get the page in an html

format and use beautiful soup to look for a table called “infobox vcard” which is the table on the right side of almost every wikipedia page that gives you information like birthday, death day, and other quick facts. This part was fairly easy to implement as we had to only worry about whether or not the table’s row equals the tag we are searching for. If the tag exists, then we get the information associated with the row. This was especially helpful for authors that didn’t have a lot of information on their GoodReads page and also for publishers.

Finally, the back-end team really benefited from the online tutorials regarding MongoDB in Python. Santhosh was able to dive further into Selenium and learn about the auto click feature in the Selenium API, which was beyond the knowledge scope required for the database.

Santhosh also benefited from the forums on Goodreads and other APIs for providing help on how to fetch certain bits of data. This helped implement the dynamic callback mechanism of our database and to aid the front-end team with programmatically piping data to be shown on our website.

4.3.3 Phase III

One thing that worked well this phase was implementing the filtering interface. We decided to design our filtering interface similar to that of Amazon, where the filtering list would be located on the left hand side of the model pages. A user will click a drop down arrow and select the attributes that they would like to filter by. Once the user finished clicking all the attributes they wanted, the user would click the submit button that would then pull up the filtering results page. Implementing this design consisted of using a drop down option of cards. Each card represents an attribute and each card allows a user to check which sub-attribute a user is looking for.

In addition to the filtering interface, our IDE, PyCharm, has been working very well. PyCharm has allowed us to pull directly from our GitHub repository, keeping everyone up to date with the latest code and changes. PyCharm has also been beneficial in the fact that it points out errors in our code, mainly syntactical errors. It also has a debugger tool which the front-end and back-end team could test incremental changes in our code to see the effects on our deployed website. For example, if we are missing a colon or if our spacing is off, PyCharm has been a decent IDE that allows our team to code efficiently.

Another thing that has worked well for our team was our user stories. While the user stories took a while to get used to, creating the stories each phase were extremely helpful. Not only did the user stories help our team refocus after each project phase, but it helped us keep track of which tasks were completed and which tasks still needed to be accomplished for each respective phase. Our user stories also shed some light on how to visualize the general expectations of people who are using our website and their collective goals with using our services. We generally used these stories to stay on track and to commit clean and efficient code.

As for the back end, we decided on a newer method that worked really well when it came down to remake our database. We learned that we needed to start off with the authors. Initially, we thought that since the publishers are at the top level, we decided to start off with finding information about the publishers first, and then find books made by the publishers, and lastly the authors. Although, this method didn't work because there wasn't a lot of information about authors. Therefore, we decided to find information about authors first. We did this by going through a list of famous authors. This method ensured that Wikipedia had plenty of information for most of those authors because they were famous. We also realized that all books had plenty of information so we didn't need to worry about that. The only thing we needed to worry about is making sure that the publishers had all the information that we needed.

Additionally with this method, we used authors that had all the information we needed. Then we got a list of books that those authors had made. Then we found the publishers of each of those books and checked to see if their publisher had all the information that we needed. For example, let's say Author A had books {A', B', C'}. And the publishers would be described by double quotes "", so we have {A' -> A'', B' -> B'', C' -> C''}. Now out of these, the publisher C'' doesn't have all the information that we need. Therefore, we don't consider the book C' anymore, and since A'' and B'' had all the information that we needed, we add those to the respective databases and move on to the next author. Another case we had to deal with is where all three publishers A'', B'', and C'' don't have all the information that we needed. In this case, we delete the author and all the books and publishers associated with it, and move on to the next author. This method

ensured that we have at least one author that had one book that had one publisher where each of the documents had all the information that we needed it to have.

4.3.4 Phase IV

One thing that worked well this phase was the use of pair-programming for the first time on the back-end team. We were a bit confused on how to implement a new design pattern for our current hierarchy of our class instances, but through teamwork and bouncing ideas off of one another, we were able to quickly fix syntactical and logic errors when looking through the debugging console output together. We were able to complete our main UML diagrams of how the new features might be integrated into our repatterning scheme and were able to directly implement with little to no errors, due to our thorough planning and discussion prior to the implementation. Previously, we would work on back-end features separately, but it seems that for this phase, we were able to understand the requirements of the facade design pattern and implement in successfully while considering encapsulation of the instance information and keeping in mind of what the clients should expect with interacting with the GUI built by the front-end.

In addition to the incredibly useful debugger on PyCharm, we started to experiment with some of the plugins and were enthralled with the MongoDB Plugin and LiveEdit HTML and CSS code. These tools proved to be an invaluable source in checking for updates in our production website in real-time with back-end changes, as well as easy communication with our databases. It usually took a while for our database instances to load up on MongoDB Compass and having this plugin installed helped streamline our process of adding and changing elements relatively quickly in our database. Incremental changes in our back-end could be seen on our end, rather than requesting the front-end team to make periodic deployments via Heroku to see our changes to the deployed website. Overall, it was a great resource to use and we were also able to circumvent a lot of the connection issues we rarely would have faced when connecting to the NoSQL database.

Another thing that worked really well this Phase was Zoom. When creating our project presentation, we wanted to incorporate pre recorded demos showing off the features of the

website. Zoom helped with this tremendously. Equipped with a record button and screen share functionality, Zoom made it extremely easy for each team member to record their demo and export it to our presentation. Not only was recording easy, but it allowed us to split up the work and put our presentation together more quickly.

In addition to the above, refactoring went smoothly. In the example from the struggles section, we were having a hard time deciding where to place the method. Once we decided, the method extraction and the rest of the refactoring caused minimal issues. In particular, method extraction was really easy thanks to the refactoring features in the PyCharm IDE.

The backend used VSCode's debugger when finding things to refactor, and their Python extension proved to be very helpful in finding errors. The VSCode debugger shows all the variables in the current context in a very clear format, and we used this a lot when finding certain things within the file. In addition, VSCode's shortcut reference was also very helpful because we found ourselves commenting out a lot of things for testing purposes. Lastly, VSCode has a very satisfying way of snapping windows to all 4 corners and this was helpful so that we could see 4 files at once and quickly make changes between them.

REFERENCES

- [1] "Arpit Bhayani", "Fast and Efficient Pagination in MongoDB," Oct. 2020;
<https://www.codementor.io/@arpitbhayani/fast-and-efficient-pagination-in-mongodb-9095f1bqr>

This resource helped in implementing the functionality of pagination. An initial obstacle in the beginning was figuring out how to only populate a page with a certain number of entries. This website introduced us to a function called `find().limit()` and `find.skip().limit()`. The `.limit()` but a cap on how many entries from our database we wanted to populate the page with and the `skip().limit()` allowed us to skip the previous entries from our database that we already pulled from but still limit how many entries we wanted on page. The `find().limit()` was used on all our

model pages to populate the pages with the first 10 entries from each database while the `find.skip().limit()` was used to populate the remaining model pages.

[2] “Pagination,” Oct. 2020; <https://getbootstrap.com/docs/4.0/components/pagination/>

This resource helped in implementing the pagination user interface. The website provided a basic overview of pagination, explained how to work with icons, how to configure sizing and alignment, etc. This resource proved to be extremely useful in providing the foundation of our websites pagination user interface for phase II.

[3] “goldsmith”, “Wikipedia,” Oct. 2020; <https://github.com/goldsmith/Wikipedia>

This API helped with getting information from wikipedia pages. It gave us information about documentation for using the API. We used the search feature and it showed us extra parameters for the search to make it more precise so that we get the page that we are actually looking for. We also used the summary function for a small description of the model we are getting. In addition, it allowed us to get the HTML data of a page. We used this with Beautiful Soup later to get data that we couldn’t find in the other data.

[4] “sefakilic”, “goodreads,” Oct. 2020; <https://github.com/sefakilic/goodreads>

This link provided documentation for using the GoodReads API. The API initially uses HTML GET requests, but this library allowed us to easily get data about an Author. We had to initially use a GET request that gave us the ID of an author given their name. But with that ID we were able to get other information like a description, date of birth, birthplace, etc.

[5] “Google”, “Google Books APIs,” Oct. 2020; <https://developers.google.com/books>

This link provides documentation for the Google Books API. Initially we had to register a key so that we could use their API. Then we learned that this API proved very useful in finding books associated with a certain author, publisher, or book title. This API provided very useful information about a certain book.

[6] “TheSaint321”, “How to import packages into PyCharm using pip” Oct. 2020;
<https://stackoverflow.com/questions/46225875/how-to-import-packages-into-pycharm-using-pip>

This link provided information for getting packages for pip in PyCharm. We initially had to use this command to install the Wikipedia and Goodreads package into the Pycharm environment, because we couldn't find a way to import these packages under the packages menu in the Project Interpreter settings. Also, some of these packages had dependencies which were luckily in the packages menu so it was a quick import and set-up procedure for developmental use in our virtual environment.

[7] “Guru99”, “How to Click on image in Selenium Webdriver” Oct. 2020;
<https://www.guru99.com/click-on-image-in-selenium.html>

This link provided information for clicking links using Selenium Webdriver. We wanted to get a picture for some of the models and concerning Authors and Publishers, we couldn't easily get a thumbnail picture for these two models. So this link showed us how to click an image and get the link associated with it. This link is what we stored in the database.

[8] “MongoDB”, “Query and Projection Operators” Oct. 2020;
<https://docs.mongodb.com/manual/reference/operator/query/>

We used this link to initially understand the basic operations for using MongoDB. Specifically, it was important referencing this link when performing special and specific operations. For example, when trying to do comparisons or performing searches that match a certain set of letters. This was mainly important when browsing the database using MongoDB Compass which is a GUI application for MongoDB.

[9] “PyMongo”, “Tutorial” Oct. 2020;
<https://api.mongodb.com/python/current/tutorial.html>

This link was important during the entirety of using PyMongo. It gave us the documentation for basic functions when using PyMongo. More specifically, it gave us functions for CRUD operations. It also allowed us to test out the different operations that was provided by PyMongo

before using the functions to perform hundreds of operations on a larger set of models. It was also helpful in performing unittests as it showed us about the results class that is returned by all CRUD operations to inform the developer what the function did.

[10] “Beautiful Soup”, “Beautiful Soup Documentation” Oct. 2020;

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

This link provided us information about using Beautiful Soup. Beautiful Soup is a web scraping library in Python that takes a HTML file and parses in a bunch of linked lists so that we can easily get the data in it. It also showed us about using a find function to find a certain element in the file with a specific class or id. We learned that all tables in the Wikipedia page have the same class name so this proved to be very useful. Upon successfully finding an element with a certain class or id we were able to get the text associated with it.

[11] “db.collection.find()”, “MongoDB Documentation” Nov. 2020;

[**db.collection.find\(\) — MongoDB Manual**](#)

This link provided us information about how to use the find() function. The find() function was extremely useful in querying what data we wanted to show when a user wanted to filter and sort by a specific attribute. This website not only extended on the behavior of the find() method, but it provided examples of how to implement the various ways to use the find() method.

[12] “Error Message: MongoError: bad auth Authentication failed through URI string”,

“StackOverflow” Nov. 2020;

[**node.js - Error Message: MongoError: bad auth Authentication failed through URI string - Stack Overflow**](#)

This link helped us figure out why we were getting authentication failure message when trying to access our database.

[13] “ Debugging configurations for Python apps in Visual Studio Code”, “Visual Studio” May 2020;

[Debugging configurations for Python apps in Visual Studio Code](#)

This link helped us with setting up the debugger in VS Code and setting breakpoints to check if certain exceptions were handled properly.

[14] “Merging vs Rebasing”, “Atlassian”

[Merging vs. Rebasing | Atlassian Git Tutorial](#)

This link helped us fix our Git merge conflicts and enabled us to commit and pull changes properly from a certain branch.